Algorithms

Reference Notes

University of Waterloo

CS 341, Fall 2026

Revised 27 October 2026

Contents

	Preface	iv
1	Introduction	1
I	**Divide & Conquer**	**4**
2	Merge Sort	5
	Interlude: Padding Arguments	9
3	Counting Inversions	11
4	Maximum Subarray	14
5	Median and Selection	17
6	Closest Pair	20
7	Integer Multiplication	23
	Interlude: Master Theorem	26
8	Matrix Multiplication	28
	Interlude: Applications of Matrix Multiplication	30
9	Maximum Independent Set	32
II	**Graph Search**	**34**
	Prelude: Graphs	35
10	Reachability by Breadth-First Search	36
	Interlude: BFS Trees	38
11	Shortest Paths	40
12	Connected Components	42
13	Testing Bipartiteness	44
14	Reachability by Depth-First Search	47
	Interlude: DFS Trees	49
15	Cut Vertices	52
16	Cut Edges	56
17	Directed Reachability	58
18	Strong Connectivity	60
19	Directed Acyclic Graphs	62
20	Strongly Connected Components	65

III Greedy Algorithms **68**
21 Interval Scheduling 69
Interlude: Priority Queues 71
22 Interval Coloring 73
23 Total Completion Time 75
24 Shortest Paths with Nonnegative Weights 77
25 Minimum Spanning Trees 80
Interlude: Disjoint Sets 83
26 Minimum Spanning Trees (revisited) 85

IV Dynamic Programming **87**
Prelude: Dynamic Programming 88
27 Weighted Interval Scheduling 91
28 Subset Sum 94
Interlude: Pseudopolynomial Time 96
29 Knapsack 97
30 Longest Increasing Subsequence 99
31 Longest Increasing Subsequence (revisited) 101
32 Longest Common Subsequence 103
33 Edit Distance 105
34 Maximum Independent Set on Trees 107
35 Optimal Binary Search Trees 109
36 Shortest Paths with Negative Weights 111
37 Negative Cycles 113
38 All-Pairs Shortest Paths 115
39 Travelling Salesperson 117

V Max Flow / Min Cut **119**
40 Maximum Flow and Minimum Cut 120
41 Maximum Flow (revisited) 123
42 Edge-Disjoint Paths 125
43 Vertex-Disjoint Paths 128
44 Edge-Disjoint Paths in Undirected Graphs 130
45 Bipartite Matching 132
46 Minimum Vertex Cover in Bipartite Graphs 134
Interlude: Certificates and Perfect Matchings 136
47 Baseball Elimination 138
48 Project Selection 140

Acknowledgments 142
Further Reading 143
Index 144

Preface

These notes present the algorithmic solutions to the computational problems covered in the *CS 341: Algorithms* course at the University of Waterloo. Each of the main chapters of the notes covers a single computational problem that we see in class. The chapters include pseudocode for an algorithm that solves the problem, a proof of correctness, and an analysis of its time complexity. In this way, they offer models of what a complete presentation of an algorithmic solution to a computational problem looks like.

These notes do not aim to cover the introduction and discussion of many of the elegant mathematical ideas that underlie the design of each algorithm. These discussions are covered in the lectures themselves and can be found in the lecture notes of Lap Chi Lau available here:

`https://cs.uwaterloo.ca/~lapchi/cs341-2025/notes.html`

This version of the reference notes is being written in the Fall 2026 offering of the CS 341 course, taught by Rafael Oliveira and myself. Generative AI has been used extensively in the compilation of these notes. In turn, content generated this way owes much to the material developed by previous instructors of the course and to the many other excellent sources of exposition on algorithm design that have been produced over the years. See the *Acknowledgments* section for a list of all the instructors of CS 341 at the University of Waterloo between 1998 and 2026, and the *Further Reading* section for a few suggestions of alternative textbooks that present the topics covered in these notes in more detail.

The current version of the notes is written for the students who are currently enrolled in the course. I hope you find them useful! Please let me know if you do, and definitely let me know as you find ways that the notes could be even more helpful in the future.

Eric Blais
Waterloo, October 2026

Chapter 1

Introduction

The main topic of this course is the design and analysis of algorithms. We will explore these tasks through concrete examples: stating a problem, finding an algorithm, and proving that it is correct and efficient. For our first example, consider the following sorted-input version of 2SUM.

2SUM

Input: A sorted integer array A and an integer target t.
Output: Indices $i < j$ such that $A[i] + A[j] = t$, or a report that no such pair exists.

By default, when the input includes a single array, we let n denote its number of elements. The entries are indexed from 1 to n. Input arrays are nonempty unless stated otherwise, and repeated values are allowed. Here, *sorted* means $A[1] \leq A[2] \leq \cdots \leq A[n]$. The indices must be distinct, but the two values need not be. We write $A[\ell..r]$ for the entries with indices from ℓ to r, and for the empty sequence when $\ell > r$.

1.1 Main idea

Our algorithmic solutions begin with a main idea, illustrated on a concrete example. For 2SUM, keep two indices pointing to the smallest and largest remaining entries. If their sum is too small, advance the left index; if it is too large, retreat the right index. Sortedness lets us discard that endpoint without losing a possible solution.

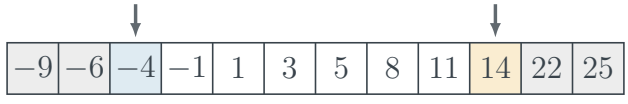


Figure 1.1: When considering the elements at positions 3 and 10 of the array A with target sum 17, we see that $A[3] + A[10] = -4 + 14 = 10 < 17$ so we can advance the left pointer to 4. Grey entries have already been discarded.

1.2 Algorithm

We describe the algorithm precisely using *pseudocode*. It records the operations and their order without the syntax of a particular programming language.

Algorithm TwoSum$(A[1..n], t)$

```
1 i ← 1; j ← n
2 while i < j do
3     s ← A[i] + A[j]
4     if s = t then return (i, j)
5     else if s < t then i ← i + 1
6     else j ← j − 1
7 return "no pair exists"
```

The signature TwoSum$(A[1..n], t)$ specifies the inputs and makes n available to the algorithm; we need no separate instruction to compute the array's length. The arrow $\leftarrow$ means assignment, whereas $=$ tests equality. Indentation indicates which instructions belong to a loop or a conditional branch. A **return** ends the current call.

To *solve* a problem, an algorithm must terminate on every input satisfying the input specification and return an output satisfying the output specification. We include a proof of correctness for every algorithm. Here, the key is to show that discarding an endpoint cannot remove the last possible solution.

Theorem 1.1 (Correctness)**.** TwoSum *solves the* 2SUM *problem.*

Proof. We maintain the following *loop invariant*: before each test of $i < j$, if a solution exists, then a solution exists using two distinct indices in $[i, j]$. Initially $[i, j] = [1, n]$, so the invariant holds.

Suppose $i < j$. If $A[i] + A[j] = t$, the returned indices satisfy the output specification. If $A[i] + A[j] < t$, then for every q with $i < q \leq j$, sortedness gives

$$A[i] + A[q] \leq A[i] + A[j] < t.$$

Thus no solution in the current interval uses i, and increasing i preserves the invariant. If $A[i] + A[j] > t$, then for every p with $i \leq p < j$,

$$A[p] + A[j] \geq A[i] + A[j] > t.$$

Thus no solution in the current interval uses j, and decreasing j preserves the invariant.

Each iteration that does not return decreases $j - i$ by one. Since the loop runs only while $j - i > 0$, the algorithm terminates. If it exits the loop, $[i, j]$ contains fewer than two distinct indices. The invariant then implies that no solution exists, justifying the final report. □

We also analyze the *running time*: the number of elementary operations performed. Our default model is the *Word-RAM* (word random-access machine). Data are stored in memory words, and an array entry can be accessed directly by its index. Reading or writing a word, comparing word-sized integers, and adding or subtracting them each take constant time. We assume that indices, input integers, and intermediate values such as $A[i] + A[j]$ fit in a constant number of words. Thus arithmetic on those values also takes constant time.

We state *worst-case* bounds, which apply to every permitted input of a given length. Writing $T(n) = O(f(n))$ means that there are constants $C > 0$ and n_0 such that $T(n) \leq Cf(n)$ for every $n \geq n_0$. In particular, $O(n)$ denotes a linear upper bound. Our formal running-time proofs establish upper bounds; when useful, a short remark afterwards explains why the bound is tight.

Theorem 1.2 (Running time)**.** TwoSum *takes $O(n)$ time in the Word-RAM model.*

Proof. Initially $j - i = n - 1$. Each iteration that does not return decreases $j - i$ by one, and the loop runs only while $j - i > 0$. There are therefore at most $n - 1$ iterations, including a possible iteration that returns. Each iteration performs a constant number of array accesses, arithmetic operations, assignments, and comparisons, each of constant cost. Initialization and the final report also take constant time. The total running time is at most $c(n - 1) + c_0$ for fixed constants c, c_0, and hence is $O(n)$. □

Tightness. If all entries are zero and $t = 1$, the left index advances $n - 1$ times before the algorithm reports failure. Thus the linear bound is tight for this algorithm.

Part I

Divide & Conquer

Chapter 2

Merge Sort

A simple divide-and-conquer strategy involves dividing the input into two halves, solving each half recursively, and combining the answers. One problem where this approach is particularly effective is sorting.

Sorting

Input: An integer array A.
Output: A sorted array B containing exactly the entries of A, with each integer appearing the same number of times.

2.1 Main idea

Split the array into equal halves, sort each half recursively, and merge the two sorted results. A single entry is already sorted. For example,

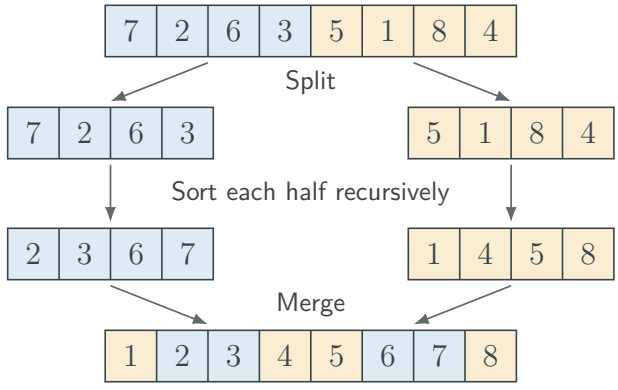


Figure 2.1: Split, sort, and merge. Shading records the original halves.

2.2 Auxiliary problem

To combine the sorted subarrays in MergeSort, we need a solution to the following problem.

Merging
Input: Two sorted integer arrays A and B.
Output: A sorted array C containing exactly all entries of A and B, with multiplicity.

To merge, repeatedly take the smaller first remaining entry. In Figure 2.1, we take 1 from the right array, then 2 and 3 from the left, and continue. Sortedness makes the next choice local: the smallest remaining entry is at the front of one of the two arrays.

Algorithm Merge($A[1..n], B[1..m]$)

```
1 allocate an array C[1..n + m]
2 i ← 1; j ← 1; k ← 1
3 while i ≤ n or j ≤ m do
4     if j > m or (i ≤ n and A[i] ≤ B[j]) then
5         C[k] ← A[i]; i ← i + 1
6     else
7         C[k] ← B[j]; j ← j + 1
8     k ← k + 1
9 return C
```

Lemma 2.1 (Correctness of Merge)**.** Merge *solves the* Merging *problem.*

Proof. We prove the following invariant at each evaluation of the **while** condition:

(i) $1 \le i \le n+1$, $1 \le j \le m+1$, and $k = i + j - 1$.

(ii) $C[1..k-1]$ is sorted and contains exactly the entries of $A[1..i-1]$ and $B[1..j-1]$, with multiplicity.

(iii) Every entry of $C[1..k-1]$ is at most every entry in the unconsumed suffixes $A[i..n]$ and $B[j..m]$.

Initially, $i = j = k = 1$ and all three prefixes are empty, so the invariant holds.

Suppose the invariant holds and the loop continues. The conditional takes the first entry of the only nonempty suffix, or the smaller of the two first entries when both are nonempty. Since the suffixes are sorted, this is a smallest unconsumed entry x. The selected input entry exists, and (i) gives $k \le n + m$, so the write is valid.

Appending x preserves sortedness by (iii) and records exactly the newly consumed entry, preserving (ii). Since x is at most every remaining entry, (iii) also remains true. Incrementing k and the chosen input index preserves (i).

Each iteration consumes one input entry, so the loop terminates after at most $n + m$ iterations. At termination, (i) and the false loop condition give $i = n + 1$, $j = m + 1$, and $k = n + m + 1$. By (ii), C is sorted and contains exactly all entries of A and B, with multiplicity. □

Lemma 2.2 (Running time of MERGE)**.** *On inputs of lengths $n, m \geq 1$,* MERGE *takes $O(n+m)$ time.*

Proof. The index k starts at 1, increases by one in each iteration, and equals $n+m+1$ when the loop terminates. Hence there are exactly $n+m$ iterations. Each performs $O(1)$ work, and allocating the output array takes $O(n+m)$ time. Adding these costs gives $O(n+m)$. □

2.3 Algorithm

For now, let $A[1..n]$ be an integer array with $n = 2^h$ for an integer $h \geq 0$. Then every recursive call with more than one entry splits its input into two equal, nonempty halves. The following algorithm returns a sorted array and does not modify its input. The *Padding Arguments* interlude will remove the assumption on the input length.

Algorithm MERGESORT($A[1..n]$)

```
1 if n = 1 then
2     return A
3 B ← MergeSort(A[1..n/2])
4 C ← MergeSort(A[n/2 + 1..n])
5 return Merge(B, C)
```

Theorem 2.3 (Correctness of MERGESORT)**.** MERGESORT *solves the* Sorting *problem on arrays of length $n = 2^h$, where $h \geq 0$ is an integer.*

Proof. We use induction on h. When $h = 0$, the array has one entry. The algorithm immediately returns A, which is already sorted.

Let $h \geq 1$, and suppose the claim holds for every array of length 2^{h-1}. Consider any array A of length $n = 2^h$. Each recursive call receives an array of length $n/2 = 2^{h-1}$. By the induction hypothesis, both calls terminate, B is a sorted rearrangement of the first half, and C is a sorted rearrangement of the second half.

Lemma 2.1 implies that MERGE(B, C) terminates and returns a sorted array containing exactly the entries of B and C. These are exactly the entries of A, with the same multiplicities, because the two halves partition the input. Thus MERGESORT(A) terminates and returns a sorted rearrangement of A, completing the induction. □

Theorem 2.4 (Running time of MergeSort)**.** MergeSort *takes* $O(n \log n)$ *time.*

Proof. Let $T(n)$ be the worst-case running time on arrays of length n, where n is a power of two. There are constants $a, b > 0$ such that

$$T(1) \le a, \qquad T(n) \le 2T(n/2) + bn \quad (n \ge 2). \tag{2.1}$$

Indeed, the two recursive calls each have input length $n/2$. Outside these calls, copying the slices takes $O(n)$ time, merging takes $O(n)$ time by Lemma 2.2, and the remaining work takes $O(1)$ time. Choose a to bound the base case and b to bound all this nonrecursive work.

We prove by induction on h that, for $n = 2^h$,

$$T(n) \le an + bn \log n. \tag{2.2}$$

For $h = 0$, the right-hand side is a, so the claim follows from $T(1) \le a$. For $h \ge 1$, assume the bound holds at $n/2 = 2^{h-1}$. Then

$$\begin{aligned} T(n) &\le 2T(n/2) + bn \\ &\le 2\left(a\frac{n}{2} + b\frac{n}{2}\log\frac{n}{2}\right) + bn \\ &= an + bn(\log n - 1) + bn \\ &= an + bn \log n. \end{aligned}$$

□

Tightness. At each of the $\log n$ merging levels, the algorithm writes n entries in total. Thus this implementation takes $\Theta(n \log n)$ time, regardless of the values being sorted.

Interlude

Padding Arguments

Padding an input means adding auxiliary data to obtain an input of a convenient size, together with a rule for recovering a solution to the original problem from a solution to the padded problem.

A padding argument must justify both the recovery rule and the effect on the running time. In particular, adding data must preserve the information needed to solve the original problem.

We can use padding to remove the power-of-two restriction in MergeSort. Pad the array to the next power of two, sort it, and discard the added entries. To ensure that those entries come last, introduce a formal symbol $+\infty$ and extend integer comparisons by

$$x < +\infty \quad \text{for every integer } x, \qquad +\infty \leq +\infty.$$

The symbol $+\infty$ is an auxiliary value, not an integer. It can be represented by a flag distinguishing it from ordinary entries, so storing it and comparing it with any entry still take constant time.

Algorithm Sort($A[1..n]$)

```
1 N ← the smallest power of 2 that is at least n
2 A' ← A padded with N − n copies of +∞
3 S ← MergeSort(A')
4 return S[1..n]
```

Theorem (Correctness of Sort)**.** Sort *solves the* Sorting *problem for arbitrary input lengths.*

Proof. A power of two at least n exists, so N is well-defined. Appending $N - n$ symbols to A produces an array A' of power-of-two length N.

The correctness proofs for Merge and MergeSort also apply to arrays whose entries are integers or $+\infty$. Indeed, Merge still selects a smallest unconsumed entry at each iteration: the first entry of each nonempty sorted suffix is its smallest entry, and the extended comparison selects the smaller of the two candidates. Thus the invariant proof for Merge remains valid. The induction for MergeSort then applies without change, since it uses only the correctness of the recursive calls and of Merge. Both algorithms terminate on these extended inputs as well.

The call to MergeSort therefore returns a sorted rearrangement S of A'. Every original entry is less than $+\infty$. Consequently, the first n entries of S are exactly the original integer entries, in sorted order and with their multiplicities preserved. Thus Sort terminates and returns the required output. □

Theorem (Running time of SORT)**.** SORT *takes* $O(n \log n)$ *time.*

Proof. If $n = 1$, then $N = 1$ and the procedure takes constant time. Suppose $n \geq 2$. Since N is the smallest power of two at least n, the smaller power of two $N/2$ satisfies $N/2 < n$. Thus $n \leq N < 2n$.

We can find N by repeated doubling from 1 in $O(\log n)$ time. Forming A' and copying the first n output entries take $O(N)$ time. The running-time proof for MERGESORT is unchanged: its recurrences depend only on the input length, and comparisons involving $+\infty$ still take constant time. Sorting therefore takes $O(N \log N)$ time, and

$$N \log N \leq 2n \log(2n) = 2n(1 + \log n) \leq 4n \log n.$$

The other steps take $O(n)$ time, so the total is $O(n \log n)$. □

Remark. The bound $N < 2n$ shows that padding increases the input length by less than a factor of two. The calculation above shows why this preserves the $n \log n$ running-time bound. For a different problem or a different running-time bound, both the recovery rule and the cost must be checked again. For later problems, we will not carry out these padding analyses explicitly.

Chapter 3

Counting Inversions

A divide-and-conquer algorithm can sometimes solve a problem more efficiently by doing more than the problem asks for. Counting inversions is one example: it is convenient to sort the array while counting its inversions.

Counting Inversions

Input: An integer array A.
Output: The number $\text{inv}(A)$ of pairs (i, j) with $1 \le i < j \le n$ and $A[i] > A[j]$.

3.1 Main idea

Split the array into two halves. An inversion lies within the left half, within the right half, or crosses between them. Recursion counts the first two kinds. To count the crossing inversions quickly, ask each recursive call to return a sorted copy of its half as well.

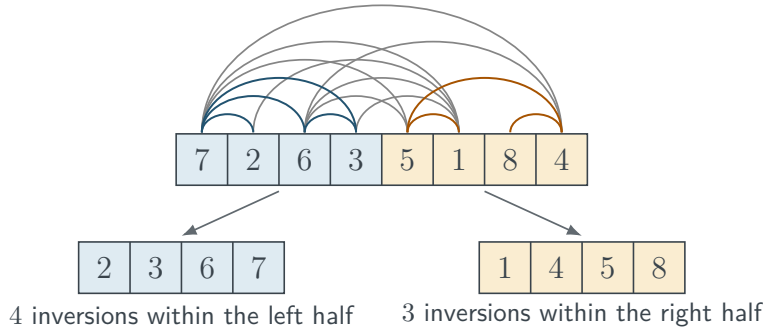


Figure 3.1: Each arc represents an inversion. Blue and gold arcs lie within the halves; grey arcs cross between them. The recursive answers include both a count and a sorted half.

There are 8 crossing inversions, giving $4 + 3 + 8 = 15$ inversions in total.

3.2 Auxiliary problem

To combine the recursive answers, merge the sorted halves while counting inversions between them.

Merging and Counting Cross Inversions

Input: Sorted integer arrays A and B.
Output: Their sorted merge and the number of pairs (i, j) with $A[i] > B[j]$.

When a left entry is written, every right entry already written is smaller than it. In the example, $2, 3, 6, 7$ contribute $1, 1, 3, 3$ crossing inversions. Count these pairs during the merge, taking the left entry first on a tie so equal values are not counted.

Algorithm MergeCount($A[1..n], B[1..m]$)

```
1  allocate C[1..n + m]
2  i ← 1; j ← 1; k ← 1; c ← 0
3  while i ≤ n or j ≤ m do
4      if j > m or (i ≤ n and A[i] ≤ B[j]) then
5          c ← c + (j − 1)
6          C[k] ← A[i]; i ← i + 1
7      else
8          C[k] ← B[j]; j ← j + 1
9      k ← k + 1
10 return (C, c)
```

Lemma 3.1 (Cross-inversion count)**.** MergeCount *solves the* Merging and Counting Cross Inversions *problem.*

Proof. The array operations and index updates are those of Merge, so its invariant and termination proof apply unchanged. For the count, maintain the additional invariant that c is the number of cross inversions whose entry from A has already been consumed.

Initially none has been consumed and $c = 0$. Suppose $A[i]$ is about to be written. Every previously consumed entry of B is strictly less than $A[i]$: when that entry was chosen, it was strictly smaller than the then unconsumed first entry of A, which is at most the current $A[i]$. There are $j - 1$ such entries. Every unconsumed entry of B is at least $A[i]$, because either B is exhausted or $A[i] \le B[j]$ and B is sorted. Thus exactly $j - 1$ cross inversions use this occurrence of $A[i]$. Line 5 counts all of them once. When an entry of B is consumed, the set of consumed entries of A is unchanged, so the invariant is preserved without changing c.

At termination all entries of A have been consumed, proving the count. □

Lemma 3.2 (Running time of MergeCount)**.** MergeCount *takes* $O(n + m)$ *time.*

Proof. As in Merge, there are exactly $n + m$ iterations. Each performs constant work, including the counter update. Allocating the output also takes $O(n + m)$ time. Adding these costs gives $O(n + m)$. □

3.3 Algorithm

Assume that n is a power of two. As in MergeSort, each call returns a newly sorted array; it also returns the inversion count.

Algorithm SortCount($A[1..n]$)

```
1 if n = 1 then return (A, 0)
2 (B, c_L) ← SortCount(A[1..n/2])
3 (C, c_R) ← SortCount(A[n/2 + 1..n])
4 (D, c_X) ← MergeCount(B, C)
5 return (D, c_L + c_R + c_X)
```

Theorem 3.3 (Correctness of inversion counting)**.** SortCount *solves the* Counting Inversions *problem for power-of-two input lengths, and also returns a sorted rearrangement of its input.*

Proof. Use induction on h, where $n = 2^h$. When $h = 0$, the only entry is already sorted and there are no index pairs. For $h \geq 1$, the induction hypothesis gives the sorted halves and their exact inversion counts.

Every inversion has both indices in the left half, both in the right half, or one in each. These cases are disjoint and exhaustive. Sorting within each half does not change the number of cross inversions: every left occurrence precedes every right occurrence in the original array, so only the comparison of their values matters. Lemma 3.1 therefore gives precisely the third count, together with the sorted rearrangement of both halves. Adding the counts proves the claim. Termination follows from the smaller recursive calls and the terminating merge. □

Theorem 3.4 (Running time)**.** SortCount *takes* $O(n \log n)$ *time.*

Proof. Copying the two slices and applying Lemma 3.2 cost $O(n)$ outside the recursive calls. Thus there are fixed $a, b > 0$ such that $T(1) \leq a$ and $T(n) \leq 2T(n/2) + bn$. The same induction as in Theorem 2.4 yields $T(n) \leq an + bn \log n$. Thus the running time is $O(n \log n)$ for $n \geq 2$. □

Chapter 4

Maximum Subarray

Divide-and-conquer often separates solutions into those lying within one half and those crossing between the halves. Here recursion handles the first two cases, while a simpler problem handles the crossing case.

Maximum Subarray

Input: An integer array A.
Output: Indices $1 \le i \le j \le n$ maximizing $\sum_{k=i}^{j} A[k]$. The chosen interval must be nonempty.

4.1 Main idea

An optimal interval lies entirely in the left half, entirely in the right half, or crosses the midpoint. Recursion handles the first two cases. For the third, choose a largest-sum suffix of the left half and a largest-sum prefix of the right half. Consider

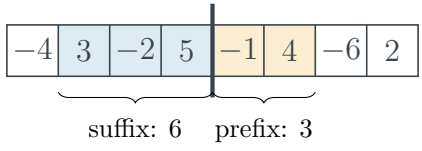


Figure 4.1: The best crossing interval joins a suffix and a prefix.

The best interval within the left half has sum 6, and the best within the right half has sum 4. The illustrated crossing interval has sum $6 + 3 = 9$, so the answer is $A[2..6]$.

4.2 Auxiliary problem

To combine the two recursive answers, we also need the best interval crossing the midpoint.

Maximum Crossing Subarray

Input: An integer array A of even length.
Output: A maximum-sum interval $A[i..j]$ with $i \le n/2 < j$, and its sum.

For even $n \ge 2$, put $h = n/2$. A crossing interval has $i \le h < j$.

Lemma 4.1 (Decomposition of a crossing interval)**.** *A maximum crossing sum is the sum of a maximum suffix sum of $A[1..h]$ and a maximum prefix sum of $A[h+1..n]$.*

Proof. Every crossing interval decomposes uniquely into a nonempty suffix $A[i..h]$ and a nonempty prefix $A[h+1..j]$. Each of its two sums is at most the corresponding maximum. Conversely, joining suffix and prefix intervals attaining those two maxima produces a crossing interval with sum equal to their sum. Hence the upper bound is attained. □

Scan leftward for a best suffix and rightward for a best prefix, then join the two intervals.

Algorithm CROSSING($A[1..n]$)

```
1 h ← n/2; s ← 0; L ← A[h]; ℓ ← h
2 for i ← h, h − 1, . . . , 1 do
3     s ← s + A[i]
4     if s > L then L ← s; ℓ ← i
5 s ← 0; R ← A[h + 1]; r ← h + 1
6 for j ← h + 1, h + 2, . . . , n do
7     s ← s + A[j]
8     if s > R then R ← s; r ← j
9 return (ℓ, r, L + R)
```

Lemma 4.2 (The crossing helper)**.** CROSSING *solves the* Maximum Crossing Subarray *problem.*

Proof. After iteration i of the first loop, $s = \sum_{k=i}^{h} A[k]$ and L is the maximum of the suffix sums starting at indices $i, i+1, \ldots, h$. The initialization and the update by comparison establish this invariant by induction over the loop iterations. At its end, ℓ attains the maximum suffix sum. The analogous invariant for the second loop gives a maximum prefix sum and its endpoint r. Lemma 4.1 proves the output specification. Initialization from actual entries ensures the chosen suffix and prefix are nonempty even for negative data. Both loops are finite, so the helper terminates. □

Lemma 4.3 (Running time of Crossing)**.** CROSSING *takes $O(n)$ time.*

Proof. The two loops have n iterations altogether, each performing constant work. Initialization and returning the triple take constant time. Adding these costs gives $O(n)$. □

4.3 Algorithm

Assume that n is a power of two. An algorithm will return the indices together with the sum of the chosen interval. Ties may be resolved in any fixed way.

Algorithm MaxSubarray($A[1..n]$)

```
1 if n = 1 then return (1, 1, A[1])
2 h ← n/2
3 (i, j, L) ← MaxSubarray(A[1..h])
4 (u, v, R) ← MaxSubarray(A[h + 1..n])
5 (a, b, X) ← Crossing(A)
6 return a largest-sum triple among
      (i, j, L), (u + h, v + h, R), and (a, b, X)
```

Theorem 4.4 (Correctness)**.** MaxSubarray *solves the* Maximum Subarray *problem for power-of-two input lengths, and also returns the maximum sum.*

Proof. Induct on h for inputs of length 2^h. The single-entry case has exactly one nonempty interval. For larger inputs, every interval is wholly in the left half, wholly in the right half, or crosses the midpoint. By the induction hypothesis, the two recursive calls return maxima in the first two classes. Reindexing the second half requires adding $n/2$ to both returned indices. The crossing lemma supplies a maximum in the third class. Selecting the largest of the three sums therefore attains the maximum over every nonempty interval. Each recursive call has smaller length and the helper loops are finite, proving termination as well. □

Theorem 4.5 (Running time)**.** MaxSubarray *takes* $O(n \log n)$ *time.*

Proof. Copying the slices and finding the crossing interval cost $O(n)$, so fixed constants give $T(1) \le a$ and $T(n) \le 2T(n/2) + bn$. The induction from Theorem 2.4 gives $T(n) \le an + bn \log n$, and hence $O(n \log n)$ for $n \ge 2$. □

Chapter 5

Median and Selection

Given an unsorted array, we seek a middle entry in the order that sorting would produce.

Median

Input: An array A of distinct integers.
Output: The lower median: the entry of rank h, where h is the unique integer satisfying $\frac{n}{2} \leq h \leq \frac{n+1}{2}$.

Finding the median of an array does not lead naturally to a divide-and-conquer algorithm: the desired entry will generally not be the median of a subproblem. We instead solve a more general problem that asks for the kth smallest entry of the array for an arbitrary rank k.

Selection

Input: An array A of distinct integers and an integer $1 \leq k \leq n$.
Output: The kth smallest entry of A.

5.1 Main idea

Choose a *pivot* and partition the array into entries below and above it. The desired rank tells us whether to return the pivot or recurse on one side. The median would be an ideal pivot, but finding it is the very problem we are trying to solve. Instead, we efficiently obtain a good enough pivot using the *median of medians*: group the entries in fives and recursively find the median of their group medians. This asks for a median on a much smaller input and still lets us discard a fixed fraction of the array.

1	2	3	8	9	14	15	20	21
4	5	6	11	12	17	18	23	24
7	**10**	**13**	**16**	**19**	**22**	**25**	**26**	**27**
34	35	36	31	32	33	28	29	30
43	44	45	40	41	42	37	38	39

Figure 5.1: Group medians are bold; their median is the dark pivot 19. The blue cells and pivot certify 15 entries at most 19; the gold cells and pivot certify 15 entries at least 19. Columns are arranged by median only for display.

5.2 Lemmas

For $n > 40$, partition the first $5q$ entries into groups of five, leaving fewer than five entries ungrouped. Let $M[1..q]$ contain the median of each complete group. The ungrouped entries are ignored only when choosing the pivot; they remain in the selection instance.

Lemma 5.1 (Quality of the pivot)**.** *If p is the lower median of M, each of the sets of entries strictly below and strictly above p has size at most $7n/10 + 3/2$.*

Proof. At least $q/2$ group medians are at most p, and at least $q/2$ are at least p. Each median in the first class has three entries in its group at most p. The groups are disjoint, so there are at least $3q/2$ entries at most p. The symmetric argument gives at least $3q/2$ entries at least p. Hence either strict side has at most $n - 3q/2$ entries. Since fewer than five entries were left over, $q \geq n/5 - 1$, and $n - 3q/2 \leq 7n/10 + 3/2$. □

5.3 Algorithm

The small groups can be sorted by comparing all their entries; their size is a fixed constant. The base case can likewise use any correct sorting procedure on at most forty entries. Unlike halving, partitioning about a pivot does not preserve power-of-two lengths. This analysis therefore keeps arbitrary lengths, using the fixed base case to absorb the small additive error in Lemma 5.1.

Algorithm SELECT$(A[1..n], k)$

```
1 if n ≤ 40 then sort A and return its kth entry
2 form the complete groups of five and their medians M[1..q]
3 h ← the lower middle rank in an array of length q
4 p ← SELECT(M, h)
5 partition the other entries into L[1..ℓ] below p
      and R[1..r] above p
6 if k = ℓ + 1 then return p
7 if k ≤ ℓ then return SELECT(L, k)
8 return SELECT(R, k − ℓ − 1)
```

Theorem 5.2 (Correctness of Select)**.** SELECT *solves the* Selection *problem. Selecting the lower middle rank solves the* Median *problem.*

Proof. Use strong induction on n. For $n \leq 40$, sorting supplies the required rank. For larger n, there are $q \leq n/5 < n$ group medians, so the first recursive call terminates and returns an entry p of M, hence of A. Every other entry lies in exactly one of L and R, because entries are distinct. Thus the rank of p in A is $\ell + 1$.

If $k = \ell + 1$, the answer is p. If $k \leq \ell$, the desired entry is the kth smallest in L. Otherwise it is the $(k - \ell - 1)$st smallest in R. The chosen rank is valid in its chosen array. Each chosen array has fewer than n entries, so the induction hypothesis proves the recursive answer and its termination. This covers every branch. Taking $k = h$, the lower middle rank in the Median specification, gives the lower median. □

Theorem 5.3 (Linear-time selection)**.** SELECT *takes $O(n)$ time in the worst case. In particular, the median can be found in linear time.*

Proof. Fix $b > 0$ so grouping, finding group medians, and partitioning cost at most bn outside the recursive calls. If the final call has length s, then, by Lemma 5.1,

$$q + s \le n/5 + 7n/10 + 3/2 \le 19n/20 \qquad (n > 40).$$

If there is no final recursive call, set $s = 0$ and charge zero for that call. Choose $K \ge 20b$ large enough that every base case costs at most Kn; there are only forty base lengths.

We prove the bound Kn by strong induction over all inputs of length n. The base cases hold by choice of K. At a larger length, both calls are smaller, so their combined cost is at most $K(q + s)$. Consequently the total is at most

$$K(q + s) + bn \le (19K/20 + b)n \le Kn. \qquad \square$$

Chapter 6

Closest Pair

In geometric divide-and-conquer, the recursive answers can simplify the problem of combining the two halves. Here they restrict where a closer pair crossing the dividing line can occur.

Closest Pair

Input: An array P of points with integer coordinates, $n \geq 2$. Assume their x-coordinates are distinct.
Output: Two distinct input points with minimum Euclidean distance.

6.1 Main idea

Split the points by a vertical line and solve each side recursively. Let δ be the smaller of the two distances obtained. A better pair must cross the line, and both its endpoints must lie within distance δ of it. For the eight points below, the dividing line is $x = 0$ and $\delta = \sqrt{10}$.

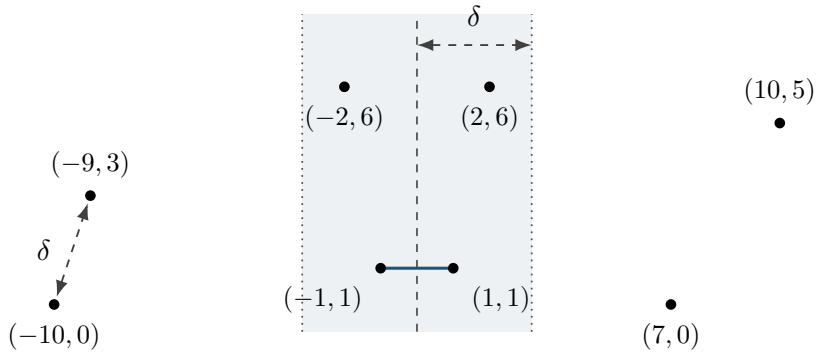


Figure 6.1: The pair far to the left uniquely defines δ. The strip extends δ to each side of the dividing line. The highlighted pair has distance 2.

The pair $(-1, 1), (1, 1)$ improves the two recursive answers. Within each half, points are separated by at least δ. This limits the number of nearby candidates in the strip and makes the combining step linear.

6.2 Lemmas

Write X for the points in increasing x-order, and Y for the same points in increasing y-order, breaking ties by x.

For even $n \geq 4$, the first $n/2$ points of X form L, and the remaining points form R. Let ℓ be halfway between their two boundary x-coordinates. Suppose the closest distances within L and R are both at least $\delta > 0$. Consider the strip $|p_x - \ell| < \delta$.

Lemma 6.1 (Seven successors suffice)**.** *In the strip's order by (y, x), any pair at distance less than δ occurs within seven positions of one another.*

Proof. Fix the earlier point p. A later point within distance δ has y-coordinate in $[p_y, p_y + \delta)$. Intersecting this horizontal range with the strip gives a rectangle of width 2δ and height δ. Divide each half of the rectangle, on its own side of $x = \ell$, into four squares of side $\delta/2$. Assign boundary points consistently to one square. Each square lies on one side of the dividing line and has diameter $\delta/\sqrt{2} < \delta$. It therefore contains at most one point: two points in it would contradict the separation within L or within R. There are eight squares and p occupies one of them. Hence at most seven later points in the strip have y-coordinate less than $p_y + \delta$. Every later point closer than δ is among those seven. □

Lemma 6.2 (Where an improving pair lies)**.** *Every pair with one point in each half and distance less than δ has both points in the strip.*

Proof. The line $x = \ell$ lies between their x-coordinates. The distance of either point from this line is at most the horizontal separation of the two points, which is at most their Euclidean distance and hence is less than δ. □

6.3 Algorithm

Assume that n is a power of two. Initially form X and Y using MergeSort, comparing the indicated coordinates and moving each whole point. The same merging invariant and induction proof apply.

Compare squared distances $D(p, q) = (p_x - q_x)^2 + (p_y - q_y)^2$, avoiding square roots. Scan the strip in y-order and compare each point with its next seven successors, as justified by Lemma 6.1.

Algorithm CLOSEST$(X[1..n], Y[1..n])$

```
 1 if n = 2 then return (X[1], X[2])
 2 h ← n/2; ℓ ← (X[h]_x + X[h + 1]_x)/2
 3 scan Y into Y_L with x < ℓ and Y_R with x > ℓ
 4 e_L ← CLOSEST(X[1..h], Y_L)
 5 e_R ← CLOSEST(X[h + 1..n], Y_R)
 6 e ← a closer pair among e_L, e_R; d ← D(e)
 7 form S[1..s] by retaining from Y points with (p_x − ℓ)^2 < d
 8 for i ← 1 to s do
 9     for j ← i + 1 to min(i + 7, s) do
10         if D(S[i], S[j]) < D(e) then e ← (S[i], S[j])
11 return e
```

In lines 3 and 7, scanning retains the order of surviving records.

Theorem 6.3 (Correctness of Closest)**.** *After the initial sorts,* CLOSEST *solves the* Closest Pair *problem for power-of-two input lengths.*

Proof. By induction on the length n. For $n = 2$, the only pair is returned. At a larger length, scanning Y produces the same two halves as splitting X, with their y orders preserved. The induction hypothesis gives a closest pair in each half. Let d be their smaller squared distance and put $\delta = \sqrt{d}$ for the proof. Distinct x-coordinates imply $d > 0$.

If a global closest pair has squared distance d, the initial choice of e is already optimal. Otherwise an optimal pair has distance less than δ and crosses the split, since neither half has a better pair. By Lemma 6.2, it lies in S; by Lemma 6.1, the inner loop compares its two points. The algorithm keeps the closest pair it has seen, so its final pair is optimal. All recursive lengths halve and all scans are finite, proving termination. □

Theorem 6.4 (Running time)**.** *Including the initial sorts, the algorithm takes* $O(n \log n)$ *time for power-of-two* $n \geq 2$*.*

Proof. Each non-base call performs two linear scans and at most seven distance comparisons per strip point. Copying subarrays is also linear. Thus $T(n) \leq 2T(n/2) + bn$ with a constant base cost at $n = 2$. The same induction used for MergeSort gives $O(n \log n)$, and the initial two sorts have that bound as well. Adding these costs gives the claimed bound. □

Chapter 7

Integer Multiplication

For numerical problems, the cost of manipulating bits matters. Integer multiplication illustrates how a less obvious choice of recursive subproblems can improve a divide-and-conquer algorithm.

Integer Multiplication

Input: Two nonnegative integers represented by bit arrays X and Y of length n, allowing leading zeroes.
Output: Their product, represented using at most $2n$ bits.

7.1 Main idea

Grade-school multiplication leads naturally to a divide-and-conquer algorithm: split both integers into high and low halves, compute the four products of these halves recursively, then shift and add the answers. Figure 7.1 illustrates this for two four-bit integers.

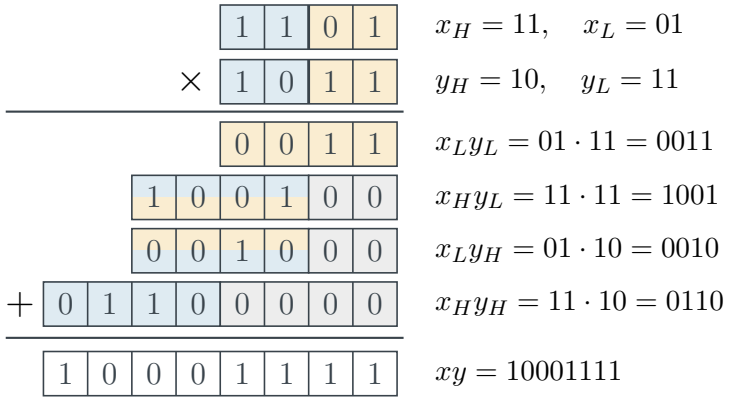


Figure 7.1: Grade-school multiplication with two-bit halves. The upper and lower colours of each partial product match its blocks from x and y, respectively. Grey cells are zeros appended by shifts.

The key idea behind faster integer multiplication is that a different choice of subproblems needs only three multiplications. Instead of computing the two cross products separately, we recover their sum from $x_H y_H$, $x_L y_L$, and $(x_H + x_L)(y_H + y_L)$.

7.2 Lemmas

This chapter counts bit operations. Assume that n is a power of two. For $n \geq 2$, put $h = n/2$ and write

$$x = x_H 2^h + x_L, \qquad y = y_H 2^h + y_L, \qquad 0 \leq x_H, x_L, y_H, y_L < 2^h.$$

These four numbers are the high and low halves of the input bit arrays.

Lemma 7.1 (Karatsuba's identity)**.** *Let* $z_H = x_H y_H$, $z_L = x_L y_L$, *and* $z_S = (x_H + x_L)(y_H + y_L)$. *Then*

$$xy = z_H 2^{2h} + (z_S - z_H - z_L)2^h + z_L.$$

Proof. Expanding the product of the two sums gives $z_S = z_H + x_H y_L + x_L y_H + z_L$. Therefore $z_S - z_H - z_L = x_H y_L + x_L y_H$. Substituting into the displayed expression gives the expansion of $(x_H 2^h + x_L)(y_H 2^h + y_L)$. □

7.3 Algorithm

The sums $x_H + x_L$ and $y_H + y_L$ may require $h + 1$ bits. We handle their carry bits separately so that every recursive call still uses h-bit inputs.

Algorithm KARATSUBA$(X[1..n], Y[1..n])$

1 **if** $n = 1$ **then return** the product of the two bits
2 $h \leftarrow n/2$; split x, y into high and low halves
3 $z_H \leftarrow$ KARATSUBA(x_H, y_H)
4 $z_L \leftarrow$ KARATSUBA(x_L, y_L)
5 $z_S \leftarrow (x_H + x_L)(y_H + y_L)$ using the recursive rule below
6 **return** $z_H 2^{2h} + (z_S - z_H - z_L)2^h + z_L$

To compute line 5, write the two sums as

$$x_H + x_L = u + c_x 2^h, \qquad y_H + y_L = v + c_y 2^h, \qquad 0 \leq u, v < 2^h, \quad c_x, c_y \in \{0, 1\}.$$

Here u, v are the low h bits and c_x, c_y are the carry bits. Compute uv by calling KARATSUBA on h-bit representations of u, v, including leading zeroes. Expanding the product then gives

$$z_S = uv + (c_x v + c_y u)2^h + c_x c_y 2^{2h}. \tag{7.1}$$

Each of $c_x v$ and $c_y u$ is either zero or the other factor, and $c_x c_y$ is a single-bit product. Multiplication by a power of two means shifting bits. The calls in lines 3 and 4 also use fixed h-bit representations.

Theorem 7.2 (Correctness)**.** KARATSUBA *solves the* Integer Multiplication *problem for power-of-two input lengths.*

Proof. By induction on the length n. At $n = 1$, the product is computed directly. At a larger length, all three recursive calls use $h = n/2$-bit inputs: the two pairs of halves and the low parts u, v of the sums. The induction hypothesis therefore supplies exactly z_H, z_L, uv. Equation (7.1) gives the exact product z_S of the two sums. Lemma 7.1 then proves that the returned expression is xy. All arithmetic can be carried out exactly on $O(n)$ bits; in particular the final value is less than 2^{2n}. All recursive calls have smaller length, which also proves termination. □

Theorem 7.3 (Bit complexity)**.** *The algorithm takes $O(n^{\log_2 3})$ bit operations.*

Proof. Let $T(n)$ be the worst-case number of bit operations. For $n \geq 2$, put $h = n/2$. For the third product, computing uv takes at most $T(h)$ operations, and computing $c_x c_y$ takes $O(1)$. These multiplications therefore cost $T(h) + O(1)$. The products $c_x v$ and $c_y u$ require only selecting or copying a factor or zero. Forming the two sums and assembling (7.1) by shifts and additions take $O(n)$ bit operations. Thus the complete third product costs $T(h) + O(n)$.

The other two products each cost at most $T(h)$. Splitting the inputs and the final shifts, additions, and subtractions also take $O(n)$ bit operations. Hence there are fixed $a, b > 0$ such that $T(1) \leq a$ and $T(n) \leq 3T(n/2) + bn$. Put $\rho = \log_2 3$ and choose $C \geq a + 2b$. We prove the stronger upper bound

$$T(n) \leq Cn^{\rho} - 2bn$$

by induction on $\log n$. At $n = 1$ it holds by the choice of C. At a larger size, the inductive step is

$$T(n) \leq 3\big(C(n/2)^{\rho} - 2b(n/2)\big) + bn = Cn^{\rho} - 2bn,$$

because $2^{\rho} = 3$. This proves the upper bound. □

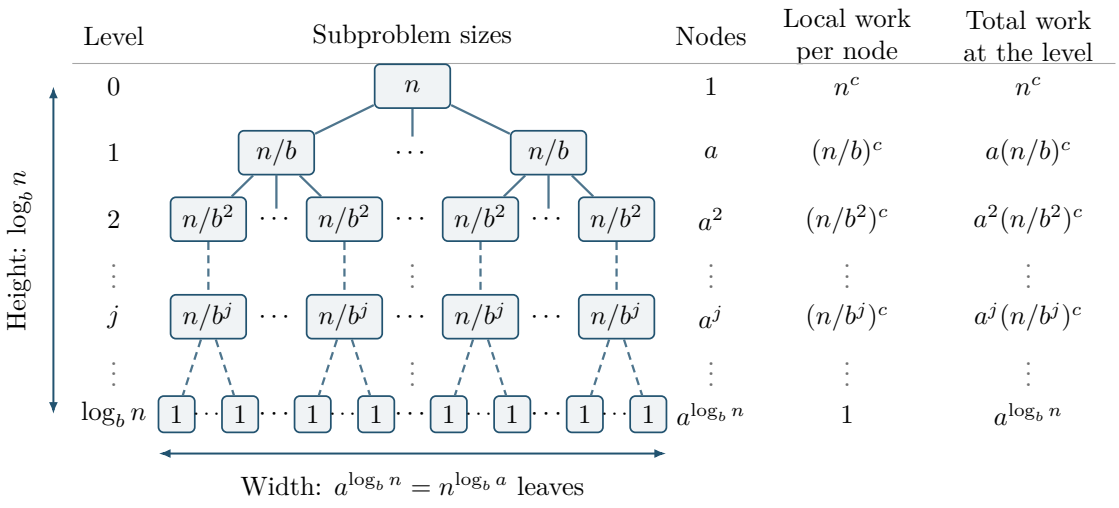
Tightness. This implementation makes all three recursive calls, even on zero operands. Its recursion tree has $3^{\log n} = n^{\log_2 3}$ single-bit leaves, each doing constant work. Thus the bound is $\Theta(n^{\log_2 3})$.

Interlude

Master Theorem

A recursion tree records the work done by recursive calls. Suppose a call of size $n > 1$ makes a calls of size n/b and does n^c work outside those calls. Assume that n is a power of b and that each call of size one does one unit of work.

At level j, there are a^j calls, each with input size n/b^j. The tree has height $\log_b n$ and $a^{\log_b n}$ leaves.



Dots omit branches; dashed connections skip intermediate levels.

Adding the work over all levels gives

$$n^c + a\left(\frac{n}{b}\right)^c + a^2\left(\frac{n}{b^2}\right)^c + \cdots + a^j\left(\frac{n}{b^j}\right)^c + \cdots + a^{\log_b n} \cdot 1.$$

Each term is a/b^c times the previous one, giving three cases:

1. **If $c > \log_b a$, the work decreases geometrically.** The sum is within a constant factor of the root's n^c work, giving $O(n^c)$.
2. **If $c = \log_b a$, all levels do the same work.** Each contributes n^c, and there are $\log_b n + 1$ levels, giving $O(n^c \log n)$.
3. **If $c < \log_b a$, the work increases geometrically.** The sum is within a constant factor of the leaf level's $a^{\log_b n} = n^{\log_b a}$ work, giving $O(n^{\log_b a})$.

We now restore constant factors and prove these bounds.

Theorem (Master Theorem)**.** *Fix integers $a \geq 1$ and $b \geq 2$ and a constant $c \geq 0$. Suppose a nonnegative function T, defined on powers of b, satisfies*

$$T(1) \leq A, \qquad T(n) \leq aT(n/b) + Cn^c \quad (n > 1),$$

where $A, C > 0$ are constants. Then

$$T(n) = \begin{cases} O(n^c), & c > \log_b a, \\ O(n^c \log n), & c = \log_b a, \\ O(n^{\log_b a}), & c < \log_b a. \end{cases}$$

Proof. Write $n = b^h$ and $q = a/b^c$. We first prove by induction on h that

$$T(n) \leq Aa^h + Cn^c \sum_{j=0}^{h-1} q^j. \tag{$*$}$$

For $h = 0$, the sum is empty and the claim is $T(1) \leq A$. For $h \geq 1$, the induction hypothesis at n/b gives

$$\begin{aligned} T(n) &\leq aT(n/b) + Cn^c \\ &\leq a\left(Aa^{h-1} + C(n/b)^c \sum_{j=0}^{h-2} q^j\right) + Cn^c \\ &= Aa^h + Cn^c\left(q\sum_{j=0}^{h-2} q^j + 1\right) \\ &= Aa^h + Cn^c \sum_{j=0}^{h-1} q^j. \end{aligned}$$

This proves $(*)$. Since $a^h = n^{\log_b a}$, it remains to bound the sum in the three cases.

Case 1: $c > \log_b a$**.** Then $q < 1$, so $\sum_{j=0}^{h-1} q^j \leq 1/(1-q)$. Using $n^{\log_b a} \leq n^c$ in $(*)$ gives

$$T(n) \leq An^{\log_b a} + \frac{C}{1-q} n^c \leq \left(A + \frac{C}{1-q}\right) n^c = O(n^c).$$

Case 2: $c = \log_b a$**.** Then $q = 1$, so the sum has h terms, all equal to one. Substituting $h = \log_b n$ in $(*)$ gives

$$T(n) \leq An^c + Cn^c \log_b n = O(n^c \log n).$$

Case 3: $c < \log_b a$**.** Then $q > 1$, so $\sum_{j=0}^{h-1} q^j = (q^h - 1)/(q-1) \leq q^h/(q-1)$. Since $n^c q^h = a^h = n^{\log_b a}$, using $(*)$ gives

$$T(n) \leq An^{\log_b a} + \frac{C}{q-1} n^c q^h = \left(A + \frac{C}{q-1}\right) n^{\log_b a} = O(n^{\log_b a}). \qquad \square$$

Remark. Some recurrences do not have the form covered by this theorem. For example, $T(n) \leq 2T(n/2) + Cn \log n$ has a different local-work term, while $T(n) \leq T(n/2) + T(n/4) + Cn$ has unequal subproblem sizes. Such recurrences can be analyzed directly using recursion trees: determine the work at each level and sum it over the levels of the tree.

Chapter 8

Matrix Multiplication

Matrix multiplication is a fundamental problem whose best recursive subproblems are far from obvious. Carefully chosen products of sums and differences reduce the number of recursive multiplications.

Matrix Multiplication

Input: Two $n \times n$ integer matrices A and B.
Output: The matrix $C = AB$, with $C[i, j] = \sum_{k=1}^{n} A[i, k]B[k, j]$.

Assume that n is a power of two. We count scalar arithmetic operations, assuming all intermediate entries fit in a constant number of words.

8.1 Main idea

For $n \geq 2$, partition each matrix into four square blocks of side $n/2$. Then

$$AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.$$

Computing these four blocks directly requires eight submatrix products. As in integer multiplication, we can find a different way to compute the same result using fewer products: Strassen's identities recover all four blocks from just seven products of sums and differences of the input blocks.

8.2 Lemmas

Define the following seven products:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), & M_2 &= (A_{21} + A_{22})B_{11}, \\ M_3 &= A_{11}(B_{12} - B_{22}), & M_4 &= A_{22}(B_{21} - B_{11}), \\ M_5 &= (A_{11} + A_{12})B_{22}, & M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

Lemma 8.1 (Strassen's identities)**.** *The blocks of AB are*

$$C_{11} = M_1 + M_4 - M_5 + M_7, \qquad C_{12} = M_3 + M_5,$$
$$C_{21} = M_2 + M_4, \qquad C_{22} = M_1 - M_2 + M_3 + M_6.$$

Proof. Distributing products and cancelling terms, without changing the order of factors, gives

$$\begin{aligned} M_1 + M_4 - M_5 + M_7 &= A_{11}B_{11} + A_{12}B_{21}, \\ M_3 + M_5 &= A_{11}B_{12} + A_{12}B_{22}, \\ M_2 + M_4 &= A_{21}B_{11} + A_{22}B_{21}, \\ M_1 - M_2 + M_3 + M_6 &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

These are the four blocks of AB. □

8.3 Algorithm

Algorithm STRASSEN$(A[1..n, 1..n], B[1..n, 1..n])$

```
1 if n = 1 then return the matrix [A[1,1]B[1,1]]
2 split A, B into their four blocks of side n/2
3 compute M_1, ..., M_7 above by seven recursive calls
4 form C_11, C_12, C_21, C_22 using Lemma 8.1
5 return the matrix with these four blocks
```

Theorem 8.2 (Correctness)**.** STRASSEN *solves the* Matrix Multiplication *problem for power-of-two dimensions.*

Proof. Induct on $\log n$. The scalar base case is the defining product. For a larger dimension, the induction hypothesis gives the exact seven block products, and Lemma 8.1 then gives every block of AB. All recursive calls have smaller dimension, proving termination. □

Theorem 8.3 (Arithmetic complexity)**.** STRASSEN *takes $O(n^{\log_2 7})$ scalar operations.*

Proof. There are seven half-dimension calls and $O(n^2)$ additional operations: copying blocks, adding and subtracting matrices, and assembling the output. Since $\log_2 7 > 2$, the Master Theorem gives the upper bound $O(n^{\log_2 7})$, with constant cost at dimension one. □

Tightness. There are $7^{\log n} = n^{\log_2 7}$ scalar leaf multiplications, so the bound is $\Theta(n^{\log_2 7})$.

Interlude

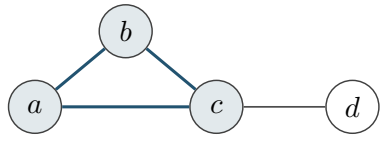
Applications of Matrix Multiplication

Matrix multiplication appears in many problems whose input is not initially a pair of matrices. Composing linear maps, computing all pairwise inner products of two collections of vectors, and combining relations between vertices are three examples. Expressing such a task as a matrix product allows an improvement in matrix multiplication to improve the resulting algorithm. Triangle finding gives a concrete example of this principle.

Triangle Finding

Input: A graph $G = (V, E)$.
Output: Three distinct pairwise adjacent vertices, or a report that no such triple exists.

The *adjacency matrix* of an n-vertex graph G is the $n \times n$ matrix A with $A[u, v] = 1$ if $\{u, v\} \in E$ and $A[u, v] = 0$ otherwise. The naïve search for a triangle checks every triple of vertices using this adjacency matrix and takes $O(n^3)$ time. Multiplying the adjacency matrix by itself can be used to find triangles more rapidly.



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 8.1: A graph and its adjacency matrix representation, with rows and columns following the order a, b, c, d.

As in the preceding chapter, assume that n is a power of two.

Algorithm TRIANGLE(G)

```
1 construct the adjacency matrix A
2 C ← STRASSEN(A, A)
3 for each edge {u, v} ∈ E do
4     if C[u, v] > 0 then
5         find w with A[u, w] = A[w, v] = 1 by scanning the vertices
6         return (u, v, w)
7 return no triangle
```

Theorem (Correctness of TRIANGLE)**.** TRIANGLE *solves the* Triangle Finding *problem.*

Proof. By the definition of matrix multiplication,

$$C[u, v] = (A^2)[u, v] = \sum_{w \in V} A[u, w]A[w, v].$$

The value of $A[u, w]A[w, v]$ is 1 when w is adjacent to both u and v and is 0 otherwise. Thus $C[u, v]$ counts their common neighbours.

If an edge $\{u, v\}$ has $C[u, v] > 0$, a common neighbour exists, so the scan in line 5 finds one. Since the graph has no loops, this vertex differs from both u and v. The returned triple therefore forms a triangle. Conversely, any triangle has an edge with a common neighbour, so its matrix entry is positive. The algorithm therefore finds a triangle whenever one exists. The matrix multiplication and all scans terminate. □

Theorem (Running time of TRIANGLE)**.** TRIANGLE *takes* $O(n^{\log_2 7})$ *time.*

Proof. A simple graph has $O(n^2)$ edges, so constructing A and scanning all edges take $O(n^2)$ time. Finding a common neighbour takes $O(n)$ time and happens at most once, since the algorithm then returns.

The running time $T(n)$ of STRASSEN satisfies $T(n) \leq 7T(n/2) + O(n^2)$, with a constant base case. Since $\log_2 7 > 2$, the Master Theorem gives $T(n) = O(n^{\log_2 7})$, which also bounds the total time after adding the $O(n^2)$ work outside this call. □

Chapter 9

Maximum Independent Set

Divide-and-conquer is useful even when the resulting algorithm takes exponential time. In that setting, removing just one more vertex in a recursive branch can significantly improve the running-time bound.

Maximum Independent Set

Input: A graph $G = (V, E)$.
Output: A largest subset $S \subseteq V$ containing no pair of adjacent vertices.

9.1 Main idea

Choose a vertex v. Every independent set either excludes v, or includes v and therefore excludes every neighbour of v. Solve these two cases recursively and keep the larger answer. For example, branch on c in Figure 9.1.

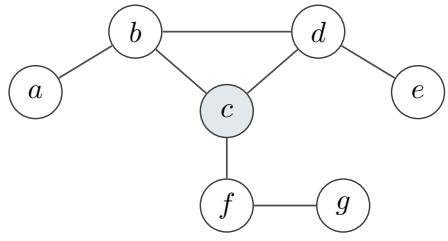


Figure 9.1: Excluding c gives an independent set of at most three vertices; including c gives $\{a, c, e, g\}$, with four vertices.

9.2 Algorithm

Recursive instances may have no vertices. Write $N[v]$ for v together with all its neighbours.

Algorithm IndependentSet(G)

```
1 if G has no edges then return V
2 choose a vertex v of maximum degree
3 S_0 ← IndependentSet(G − {v})
4 S_1 ← {v} ∪ IndependentSet(G − N[v])
5 return a larger set among S_0, S_1
```

Theorem 9.1 (Correctness)**.** IndependentSet *solves the* Maximum Independent Set *problem.*

Proof. Use strong induction on the number of vertices, including zero. When there are no edges, all vertices form an independent set and no larger subset is possible. Otherwise both recursive graphs have fewer vertices. The induction hypothesis gives an optimal independent set excluding v in the first branch. In the second branch, an independent set containing v cannot contain any neighbour of v, and its remaining vertices must form an independent set in $G - N[v]$. Conversely, every independent set in that smaller graph can be combined with v. Thus the second branch gives an optimal independent set containing v. Every independent set belongs to one of these two cases, so taking the larger answer is optimal. Strictly decreasing recursive sizes prove termination. □

Theorem 9.2 (A bound on the branching algorithm)**.** IndependentSet *runs in time* $O(n^2\varphi^n)$*, where* $\varphi = \frac{1+\sqrt{5}}{2}$*. In particular, since* $\varphi < 1.62$*, its running time is* $O(1.62^n)$*.*

Proof. Let $L(k)$ be the maximum number of leaves in a recursion tree starting from a graph with k vertices. A leaf has no edges and contributes one. At an internal call, the chosen vertex has degree at least one. Hence the two recursive graphs have $k-1$ and at most $k-2$ vertices. Strong induction gives $L(k) \leq \varphi^k$: the leaf case follows from $\varphi^k \geq 1$, and the internal case follows from

$$\varphi^{k-1} + \varphi^{k-2} = \varphi^k.$$

Every internal node has exactly two children. If the tree has I internal nodes and L leaves, counting its edges gives $2I = I + L - 1$, so it has $2L - 1$ nodes in total. Thus there are $O(\varphi^n)$ calls.

First convert the input to an adjacency matrix in $O(n^2)$ time. Using these matrices and retaining original vertex labels, testing for edges, choosing a maximum-degree vertex, constructing both induced subgraphs, and copying an answer all take at most $O(n^2)$ work per call, where n is the original vertex count. Multiplying this uniform cost by the call bound proves $O(n^2\varphi^n)$. Since $\varphi/1.62 < 1$, we have $n^2(\varphi/1.62)^n \to 0$ as $n \to \infty$, which gives $n^2\varphi^n = O(1.62^n)$. □

Remark. Exhaustive search branches on whether to include each vertex, making two recursive calls on the remaining $n-1$ vertices. Continuing until no vertices remain explores all 2^n subsets and takes $\Omega(2^n)$ time. In IndependentSet, including a vertex also deletes its neighbours, so this branch has at most $n-2$ vertices. This small change gives another exponential-time algorithm, but its smaller base yields a substantial improvement in the running-time bound.

Part II

Graph Search

Prelude

Graphs

A *graph* $G = (V, E)$ represents relationships between pairs of objects: V is its set of *vertices*, and E is its set of *edges*. Write $n = |V|$ and $m = |E|$. Unless stated otherwise, graphs are finite, undirected, and simple: an edge $uv = \{u, v\}$ joins two distinct vertices, and no edge is repeated. We assume $n \geq 1$ and index vertices by $V = \{1, \ldots, n\}$; figures may also use letters.

Two vertices joined by an edge are *neighbours*. The *adjacency list* Adj[u] contains the neighbours of u, and its length is the *degree* $\deg(u)$. We represent a graph by an array of these lists, one for each vertex.

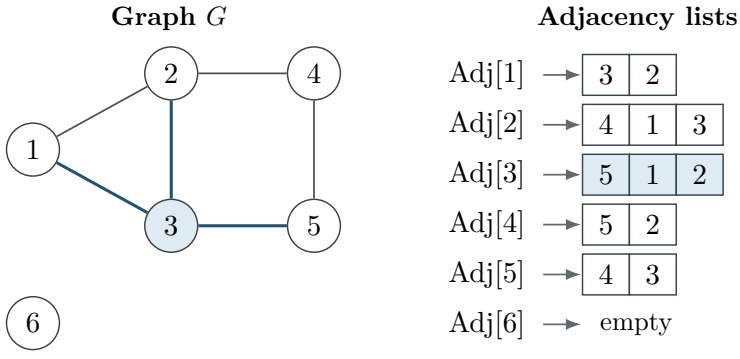


Figure 9.2: A graph and its adjacency lists. The highlighted list records the three neighbours of vertex 3: $5, 1, 2$. The lists need not be sorted. Vertex 6 is isolated, so its list is empty.

We can access a vertex's list in constant time and scan it in $O(1 + \deg(u))$ time. Each undirected edge uv contributes one entry to each of Adj[u] and Adj[v], so the lists contain $2m$ entries altogether. Including their n headers, a full scan takes $O(n + m)$ time. Neighbours may appear in any order.

A *walk* follows consecutive edges; a *path* has no repeated vertices. Its length is its number of edges; length-zero paths are allowed. A *cycle* repeats only its first vertex at the end and has at least three edges. Vertices joined by a path are *connected*, and maximal sets of pairwise connected vertices are the *components*. Write $\mathrm{dist}(u, v)$ for the shortest path length, or $+\infty$ if no path exists.

Chapter 10

Reachability by Breadth-First Search

The Reachability problem asks which vertices can be reached from a given starting vertex by following edges.

Reachability

Input: A graph G and a starting vertex s.
Output: A Boolean array indicating whether there is a path from s to each of the vertices in G.

10.1 Main idea

Breadth-first search (BFS) starts by visiting s itself, then its neighbours, then the still-unvisited neighbours of those neighbours, and so on. This systematic exploration expands the visited region one layer at a time.

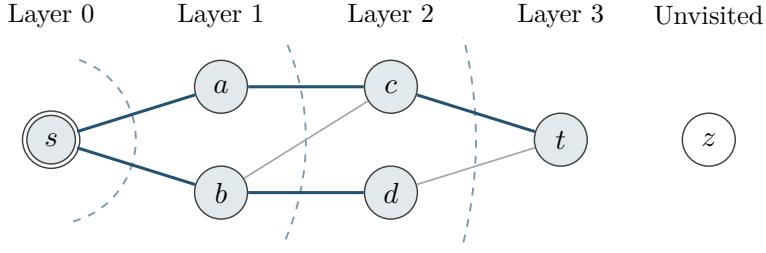


Figure 10.1: A possible exploration visits s, then a, b, then c, d, and finally t. Dashed arcs centred on s separate the layers. Blue edges show first discoveries; grey edges lead to vertices already discovered. Vertex z remains unvisited.

10.2 Algorithm

A queue keeps discovered vertices in the order in which they were visited. It inserts at the back and removes from the front, each in constant time. Mark each vertex when it enters the queue so that it enters only once.

Algorithm BFS(G, s)

```
1 set visited[v] ← false for all v ∈ V
2 create an empty queue Q
3 visited[s] ← true; enqueue s
4 while Q is not empty do
5     u ← dequeue from Q
6     for each v ∈ Adj[u] do
7         if not visited[v] then
8             visited[v] ← true; enqueue v
9 return visited
```

Theorem 10.1 (Correctness)**.** BFS *solves the* Reachability *problem.*

Proof. The algorithm terminates because each vertex is enqueued at most once and every adjacency list is finite. Correctness is established by proving two claims.

Every visited vertex is reachable from s: We prove this by induction on the order of discovery. Initially only s is visited. When a new vertex v is marked while scanning u, the induction hypothesis gives a path from s to u. Adding the edge uv gives a walk to v, from which repeated portions can be deleted to obtain a path.

Every vertex reachable from s is visited by termination: Fix a path $s = v_0, v_1, \ldots, v_k = v$. Induct on i to show that v_i is visited by termination. The case $i = 0$ holds initially. If v_{i-1} is visited, it is enqueued and eventually dequeued. Its adjacency scan either finds v_i already visited or marks it. Thus every reachable vertex is visited, and the returned array has the required values. □

Theorem 10.2 (Running time)**.** BFS *runs in time* $O(n + m)$*.*

Proof. Initializing the array costs $O(n)$ time. Each visited vertex is enqueued and dequeued once, and its adjacency list is scanned once. The queue operations therefore cost $O(n)$ in total, and the scans examine at most $2m$ adjacency entries. Each entry takes constant time to process, giving $O(n + m)$ total time. □

Interlude

BFS Trees

Recording who first discovers each vertex turns a BFS execution into a rooted tree. The depths in this tree restrict where the other edges of the graph can occur.

Whenever BFS discovers v while scanning u, call u the *parent* of v. The source s has no parent. These parent edges are the *tree edges*; other graph edges between reached vertices are *non-tree edges*. The *depth* of s is zero, and each child has depth one more than its parent. A tree is a connected graph with no cycle; a forest has no cycle.

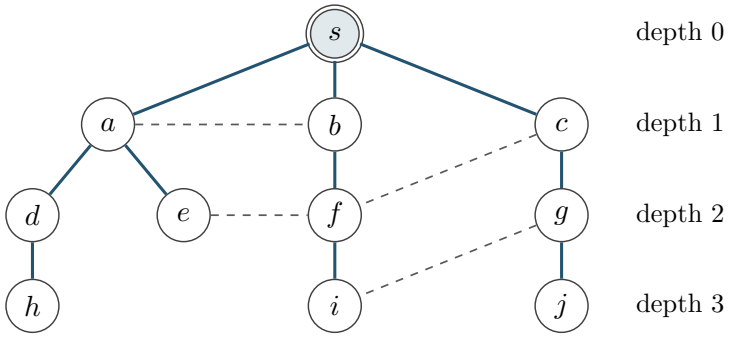


Figure 10.2: One BFS tree, shown by solid blue edges. Dashed non-tree edges can join vertices at the same depth, such as ab and ef, or at neighbouring depths, such as cf and gi. No edge skips a layer.

The following algorithm adds parent and depth arrays to BFS. A parent value $\perp$ means that no parent is assigned, and a depth of $+\infty$ marks an unvisited vertex.

Algorithm BFSTree(G, s)

```
set visited[v] ← false, parent[v] ← ⊥,
    and depth[v] ← +∞ for all v ∈ V
create an empty queue Q
visited[s] ← true; depth[s] ← 0; enqueue s
while Q is not empty do
    u ← dequeue from Q
    for each v ∈ Adj[u] do
        if not visited[v] then
            visited[v] ← true; parent[v] ← u
            depth[v] ← depth[u] + 1; enqueue v
return (parent, depth)
```

The extra assignments do not change which vertices BFS visits or their order. Thus the algorithm terminates and visits exactly the vertices reachable from s.

Lemma (BFS tree)**.** *The parent edges on the vertices reachable from s form a tree rooted at s.*

Proof. Every non-source vertex is discovered once and gets one parent, discovered earlier. Following parents therefore cannot repeat a vertex and must end at s, the only discovered vertex without a parent. Thus the parent edges connect all reached vertices. If they contained a cycle, the last-discovered vertex on it would have two incident cycle edges to earlier vertices. Both would have to be its parent edge, contradicting uniqueness. The parent graph is a tree. □

Induction on discovery order shows that the depth array records depths in this tree: s receives zero, and every other vertex receives one more than its parent's depth.

Lemma (Edges stay within adjacent layers)**.** *Every edge between reachable vertices joins vertices at the same depth or at depths differing by one.*

Proof. At the start of each nonempty queue iteration, if the first vertex has depth k, the queue contains only depths k and $k+1$, in that order. Indeed, this holds initially for the queue containing only s; removing its first vertex and appending newly discovered vertices of depth $k+1$ preserves the property.

Consider an edge uv, with u dequeued before v, and let k be the depth of u. If v has already been discovered when u is dequeued, it is still in the queue and has depth k or $k+1$. Otherwise, the scan of u discovers v and assigns it depth $k+1$. In either case, the depths of the endpoints differ by at most one. □

Theorem (Running time of BFSTree)**.** BFSTree *runs in time $O(n+m)$.*

Proof. Initializing the arrays costs $O(n)$ time. Each vertex enters and leaves the queue at most once, and each adjacency list is scanned at most once. The lists contain $2m$ entries in total, each requiring constant work; the parent and depth assignments add constant work per discovery. The total running time is therefore $O(n+m)$. □

The adjacency-list order can change which vertex first discovers a neighbour, and hence can change the BFS tree. For example, processing c before b in Figure 10.2 makes c the parent of f.

Chapter 11

Shortest Paths

The depths recorded by BFS solve the shortest-path problem in an unweighted graph.

Unweighted Shortest Paths

Input: A graph G and a starting vertex s.
Output: An array giving the length of a shortest path from s to each vertex in G, with $+\infty$ for unreachable vertices.

11.1 Main idea

The path in the BFS tree from s to a vertex at depth k has exactly k edges. Any other path must also reach depth k from depth zero, and each edge can increase the depth by at most one. Thus every path needs at least k edges, so the tree path is shortest.

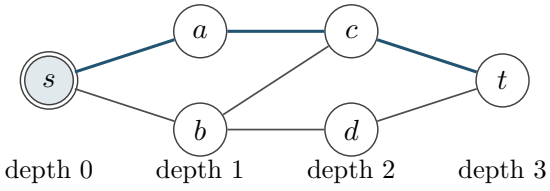


Figure 11.1: The path s, a, c, t uses three edges, one between each pair of successive layers. No path from s to t can use fewer edges, since no edge skips a layer.

11.2 Algorithm

Compute the BFS tree and return its depth array.

Algorithm BFSPaths(G, s)
1 (parent, depth) $\leftarrow$ BFSTree(G, s)
2 **return** depth

Theorem 11.1 (Correctness)**.** BFSPaths *solves the* Unweighted Shortest Paths *problem.*

Proof. BFSTree terminates and visits exactly the vertices reachable from s. For a reachable vertex v, write $h[v]$ for its depth. We prove that this is its shortest-path distance by establishing two claims.

There is a path from s to v of length $h[v]$: The parent edges form a tree rooted at s, and its path from s to v has exactly $h[v]$ edges.

Every path from s to v has at least $h[v]$ edges: Consider a path $s = v_0, v_1, \ldots, v_k = v$. The *Edges stay within adjacent layers* lemma gives $h[v_i] \leq h[v_{i-1}] + 1$ for each i. Starting with $h[s] = 0$, induction on i yields $h[v_i] \leq i$, and hence $h[v] \leq k$.

The two claims show that $h[v] = \text{dist}(s, v)$. Every unreachable vertex keeps its initial depth of $+\infty$, so the returned array has the required values. □

Theorem 11.2 (Running time)**.** BFSPaths *runs in time $O(n + m)$.*

Proof. The *Running time of BFSTree* theorem gives $O(n + m)$ time for the call to BFSTree. Returning its depth array does not increase this bound. □

Chapter 12

Connected Components

Repeating breadth-first search from new starting vertices turns a one-source search into a decomposition of the whole graph.

Connected Components

Input: A graph G.
Output: A label comp$[v]$ for each vertex such that two vertices have the same label exactly when they are connected, and the number of components.

12.1 Main idea

One search reaches exactly one connected component. Give all vertices it reaches a common label, then start another search at an unlabelled vertex. Keep earlier labels: they record components already completed.

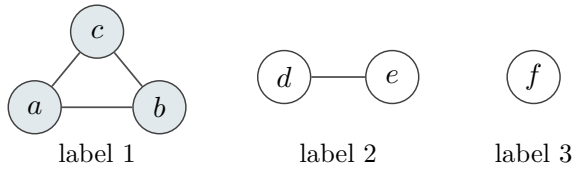


Figure 12.1: Three searches label the triangle, the edge, and the isolated vertex as three different components.

12.2 Algorithm

The algorithm performs a BFS from each vertex not reached by an earlier search. It initializes the labels only once.

Algorithm COMPONENTS(G)

```
1  set comp[v] ← 0 for all vertices; c ← 0
2  for each s ∈ V do
3     if comp[s] = 0 then
4        c ← c + 1; comp[s] ← c
5        create an empty queue Q; enqueue s
6        while Q is not empty do
7           u ← dequeue from Q
8           for each v ∈ Adj[u] do
9              if comp[v] = 0 then
10                comp[v] ← c; enqueue v
11 return (comp, c)
```

Theorem 12.1 (Correctness)**.** COMPONENTS *solves the* Connected Components *problem.*

Proof. Induct over completed searches. Initially no component has been labelled. Suppose the labelled vertices are exactly the union of some complete components, with a different positive label on each. An unlabelled start vertex s is in none of them; its entire component is therefore unlabelled. The reachability proof of Theorem 10.1 applies to the next queue search, with a nonzero label playing the role of a visited flag. It assigns the new label to exactly this component. Thus the induction assertion is preserved.

Every unlabelled vertex is eventually considered by the outer loop, so all vertices are labelled at termination. This proves both the partition and the count. Each queue search terminates by the BFS correctness proof, and the outer loop is finite. □

Theorem 12.2 (Running time)**.** COMPONENTS *runs in* $O(n + m)$ *time.*

Proof. Every vertex is enqueued and dequeued once over the entire execution, and every adjacency entry is scanned once. Including the one initialization and the outer loop, the cost is $O(n + m)$. □

Chapter 13

Testing Bipartiteness

A small addition to breadth-first search solves the fundamental problem of testing bipartiteness. The search also supplies a certificate: either a bipartition or an odd cycle.

Bipartiteness

Input: A graph G.
Output: A bipartition of V, or an odd cycle certifying that no bipartition exists.

A *bipartition* divides V into two disjoint sets so that every edge has one endpoint in each set.

13.1 Main idea

Assign a starting vertex colour 0. Every edge forces opposite colours, so give each newly discovered neighbour the opposite colour from its parent. An edge whose endpoints already have the same colour reveals a contradiction. Consider this four-cycle and the graph obtained by adding one diagonal:

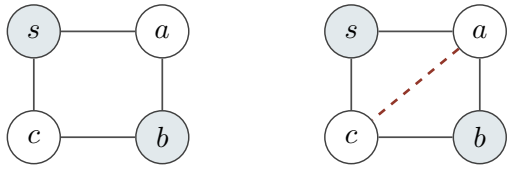


Figure 13.1: Shaded vertices have colour 0, unshaded vertices colour 1. The left graph has bipartition $\{s, b\}, \{a, c\}$. The added edge ac joins equal colours and closes the odd cycle s, a, c, s.

13.2 Algorithm

Colour vertices with 0 or 1. An uncoloured vertex has colour $\perp$. A queue search gives each newly reached neighbour the opposite colour. As in Chapter 12, restart at uncoloured vertices without resetting earlier colours.

Algorithm BIPARTITE(G)

```
1  set colour[v] ← ⊥ and parent[v] ← ⊥ for all v
2  for each s ∈ V do
3     if colour[s] = ⊥ then
4        colour[s] ← 0; d[s] ← 0; enqueue s in a new queue
5        while the queue is not empty do
6           u ← dequeue
7           for each v ∈ Adj[u] do
8              if colour[v] = ⊥ then
9                 colour[v] ← 1 − colour[u]
10                parent[v] ← u; d[v] ← d[u] + 1; enqueue v
11             else if colour[v] = colour[u] then
12                return the odd cycle obtained from u, v and their parents
13 return ({v : colour[v] = 0}, {v : colour[v] = 1})
```

To recover the cycle in line 12, let w be the lowest common ancestor of u and v in the current search tree. To find w, ascend the deeper endpoint until the depths agree, storing the traversed vertices. Then ascend both endpoints together until they meet, storing both chains. Join the two chains at that meeting vertex, reversing one chain, and close them with uv.

Lemma 13.1 (A conflicting edge yields an odd cycle)**.** *When line 12 is reached, the edge uv together with the tree path from u to v forms an odd cycle.*

Proof. Every completed queue search reaches its entire component, as proved for COMPONENTS. Thus there is no edge from the current search to an earlier search tree, and u, v lie in the same current tree. Each parent edge joins opposite colours, and the colour of a vertex is its depth modulo two. Let w be the last common vertex of the root paths of u and v, their lowest common ancestor. The tree path between them has length

$$(d[u] - d[w]) + (d[v] - d[w]) = d[u] + d[v] - 2d[w],$$

which is even because the endpoint colours agree. The edge uv is not a parent edge, so this path has at least two edges. Its vertices are distinct, and adding uv produces a simple odd cycle. □

Theorem 13.2 (Correctness)**.** BIPARTITE *solves the* Bipartiteness *problem.*

Proof. If the algorithm finishes without a conflict, every vertex has been coloured and every edge has been scanned. Its endpoints have different colours: a newly coloured endpoint receives the opposite colour, and a previously coloured one is checked. The two colour classes are therefore a bipartition.

If a conflict occurs, Lemma 13.1 gives an odd cycle. In any bipartition, the two parts must alternate along a cycle. After an odd number of edges this would put the starting vertex in the opposite part from itself, a contradiction. Thus the reported certificate correctly rules out a bipartition.

Each vertex enters a queue at most once and each adjacency list is finite. Cycle recovery strictly decreases the current depths. Thus all loops terminate. □

Corollary 13.3 (Odd-cycle characterization)**.** *A finite undirected graph is bipartite if and only if it has no odd cycle.*

Proof. A bipartition rules out an odd cycle by the alternating-parts argument. Conversely, on a graph with no odd cycle the algorithm cannot return the certificate in Lemma 13.1; it must return a bipartition. □

Theorem 13.4 (Running time)**.** BIPARTITE *runs in* $O(n + m)$ *time.*

Proof. Vertices enter a queue at most once and adjacency entries are scanned at most once from each endpoint. Initialization and all searches therefore cost $O(n + m)$. In cycle recovery, each ascent decreases a depth, giving at most $2n$ parent steps; assembling the cycle also takes $O(n)$ time. Recovery is performed at most once, so adding the costs gives $O(n + m)$. □

Chapter 14

Reachability by Depth-First Search

Recall that the Reachability problem asks which vertices can be reached from a given starting vertex by following edges.

Reachability (restated)

Input: A graph G and a starting vertex s.
Output: A Boolean array indicating whether there is a path from s to each of the vertices in G.

14.1 Main idea

Depth-first search (DFS) starts at s and follows unvisited neighbours as far as possible. When it has no unvisited neighbour left, it returns to the previous vertex and continues with any remaining unvisited neighbours. This systematically explores one branch before moving to the next.

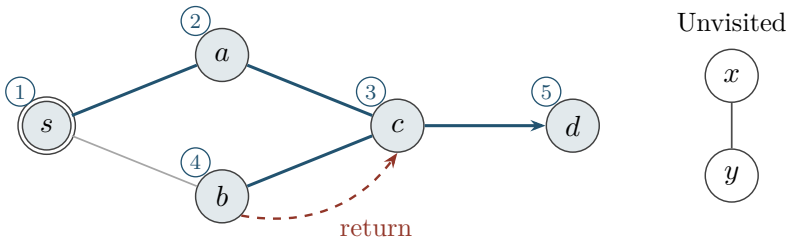


Figure 14.1: The numbered markers give discovery order s, a, c, b, d, taking a before b at s and b before d at c. Solid arrows show first discoveries along undirected edges. After exploring b, DFS returns to c before visiting d; the grey edge bs leads to an already visited vertex. Vertices x and y remain unvisited.

14.2 Algorithm

The recursive procedure Explore(u) visits u and explores each unvisited neighbour completely before continuing. It shares the graph and visited array with its caller.

Algorithm DFS(G, s)

```
1 set visited[v] ← false for all v ∈ V
2 procedure EXPLORE(u)
3     visited[u] ← true
4     for each v ∈ Adj[u] do
5         if not visited[v] then EXPLORE(v)
6 EXPLORE(s)
7 return visited
```

Theorem 14.1 (Correctness)**.** DFS *solves the* Reachability *problem.*

Proof. The algorithm terminates because each recursive call immediately marks a previously unvisited vertex, so there are at most n calls, each scanning a finite adjacency list. Correctness is established by proving two claims.

Every visited vertex is reachable from s: We prove this by induction on the order of discovery. Initially only s is visited. When a new vertex v is marked while scanning u, the induction hypothesis gives a path from s to u. Adding the edge uv gives a walk to v, from which repeated portions can be deleted to obtain a path.

Every vertex reachable from s is visited by termination: Fix a path $s = v_0, v_1, \ldots, v_k = v$. Induct on i to show that v_i is visited by termination. The case $i = 0$ holds initially. If v_{i-1} is visited, its call scans every neighbour before returning. Its adjacency scan either finds v_i already visited or calls EXPLORE(v_i), which marks it. Thus every reachable vertex is visited, and the returned array has the required values. □

Theorem 14.2 (Running time)**.** DFS *runs in time $O(n + m)$.*

Proof. Initializing the array costs $O(n)$ time. Each visited vertex has one recursive call, and its adjacency list is scanned once. The work outside the scans therefore costs $O(n)$ in total, and the scans examine at most $2m$ adjacency entries. Each entry takes constant time to process, giving $O(n + m)$ total time. □

Interlude

DFS Trees

Recording who first discovers each vertex turns a DFS execution into a rooted tree. Its non-tree edges have a different structure from those of a BFS tree: they join vertices to ancestors, never different branches. We consider undirected graphs throughout this interlude.

As for BFS, the vertex that first discovers v is its *parent*, and the source s has no parent. Parent edges are *tree edges*; other edges between reached vertices are *non-tree edges*. The *ancestors* of v are the vertices other than v on its tree path to the root. The *descendants* of v are the vertices other than v whose tree path to the root passes through v. The *subtree* rooted at v consists of v and its descendants.

Instead of depth, useful additional information to record is each vertex's *start time*, when it is first visited, and *finish time*, when its recursive call returns. A clock advances at every start and finish event.

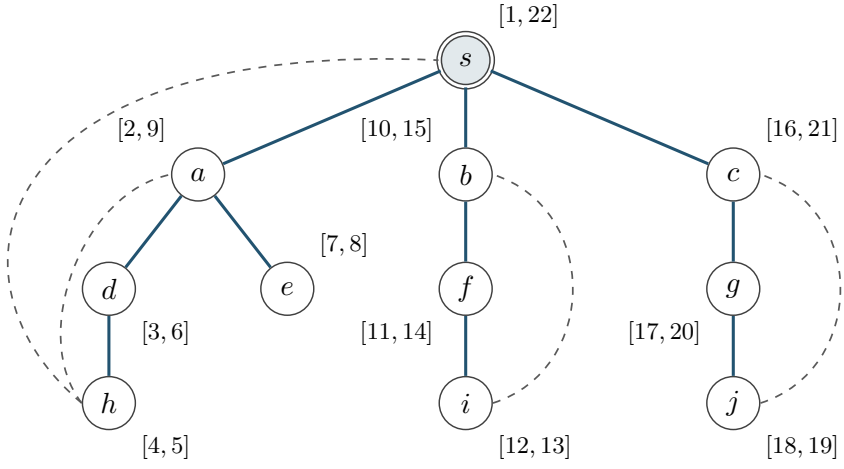


Figure 14.2: One DFS tree, shown by solid blue edges. The label beside each vertex gives [start, finish]. Dashed non-tree edges join descendants to the root (hs) or an internal ancestor (ha, ib, jc), and may skip several levels. No edge joins different branches.

The following algorithm adds parent, start, and finish arrays to DFS. A value of $\perp$ means that no parent or time has been assigned. The local procedure shares these arrays and the clock with its caller.

Algorithm DFSTree(G, s)

```
1  set visited[v] ← false and
       parent[v], start[v], finish[v] ← ⊥ for all v ∈ V
2  clock ← 0
3  procedure Explore(u)
4      visited[u] ← true
5      clock ← clock + 1; start[u] ← clock
6      for each v ∈ Adj[u] do
7          if not visited[v] then
8              parent[v] ← u; Explore(v)
9      clock ← clock + 1; finish[u] ← clock
10 Explore(s)
11 return (parent, start, finish)
```

The extra assignments do not change which vertices DFS visits or their order. Thus the algorithm terminates and visits exactly the vertices reachable from s.

Lemma (DFS tree)**.** *The parent edges on the vertices reachable from s form a tree rooted at s.*

Proof. Every non-source vertex is discovered once and gets one parent, discovered earlier. Following parents therefore cannot repeat a vertex and must end at s, the only discovered vertex without a parent. Thus the parent edges connect all reached vertices. If they contained a cycle, the last-discovered vertex on it would have two incident cycle edges to earlier vertices. Both would have to be its parent edge, contradicting uniqueness. The parent graph is a tree. □

The clock increments once on entering each call and once on leaving it. Consequently all recorded times are distinct and give the order of these events. Unvisited vertices keep $\perp$ in both time arrays.

Lemma (Nested intervals)**.** *For two distinct reached vertices u, v, the intervals*

$$[\text{start}[u], \text{finish}[u]] \quad \textit{and} \quad [\text{start}[v], \text{finish}[v]]$$

are disjoint or one strictly contains the other. The interval of u contains that of v exactly when u is an ancestor of v.

Proof. Two recursive calls are either nested or one finishes before the other begins. They cannot partially overlap, because a caller is suspended until the called computation returns. The clock records precisely the entry and exit events, so the intervals have the corresponding nesting or disjointness. A nested call is reached by a chain of recursive child calls; these are exactly the parent edges. Thus nesting is equivalent to the ancestor relation. Distinct event times make all endpoint inequalities strict. □

Lemma (Undirected non-tree edges)**.** *Every non-tree edge between reached vertices joins a vertex to an ancestor.*

Proof. Take an edge uv and suppose u is discovered first. At that moment v is unvisited. Before the call on u can finish, it examines the edge uv. If v is still unvisited, this examination calls the helper on v. Otherwise v was discovered while the call on u was active. In either case v is a descendant of u, because every discovery during that active call occurs in its recursive call tree. Thus the endpoints of uv are ancestor-related. □

Such a non-tree edge is called a *back edge*. In contrast to BFS, back edges can skip several levels, but cannot join different branches.

Theorem (Running time of DFSTree)**.** DFSTree *runs in time* $O(n + m)$.

Proof. Initializing the arrays costs $O(n)$ time. Each visited vertex has one recursive call, and each adjacency list is scanned at most once. There are at most $2m$ adjacency entries to process, each taking constant time. The parent assignments and clock updates add constant work per visited vertex, so the total running time is $O(n + m)$. □

As with BFS, changing the adjacency-list order can change the DFS tree. It can also change the start and finish times.

Chapter 15

Cut Vertices

The ancestor structure of an undirected DFS tree gives information that distance layers alone do not provide. It lets us determine whether removing one vertex disconnects the graph.

Cut Vertices

Input: A connected graph G.
Output: All vertices whose removal increases the number of components.

15.1 Main idea

After removing a non-root vertex u, each of its child subtrees must have a route to the part above u. In an undirected DFS tree, a non-tree edge can only lead to an ancestor. We therefore ask whether each child subtree has a back edge to an ancestor of u.

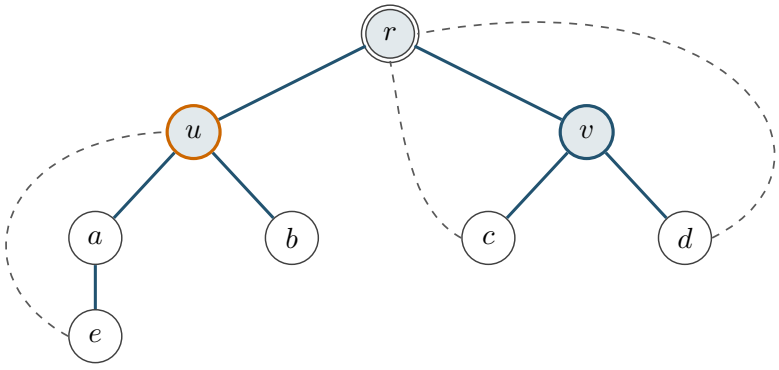


Figure 15.1: Solid edges form a DFS tree; dashed edges are back edges. The subtree $\{a, e\}$ has a back edge to u, but none to its ancestor r, so removing u disconnects it from the root. In contrast, both child subtrees of v have a back edge to r, so removing v leaves the graph connected.

15.2 Lemmas

Choose a vertex r and run DFSTree(G, r) from the DFS Trees interlude. Since G is connected, it produces a spanning tree rooted at r. Write T_u for the vertex set of the subtree rooted at u.

Lemma 15.1 (Subtree exits)**.** *For a child v of u, every edge from T_v to its complement is either the tree edge uv or a back edge from T_v to u or an ancestor of u.*

Proof. The only tree edge leaving T_v is its root's parent edge uv. By the *Undirected non-tree edges* lemma, any other edge has ancestor-related endpoints. Its endpoint outside T_v cannot be a descendant of a vertex inside T_v, since every such descendant is also in T_v. It must therefore be an ancestor outside this subtree, namely u or an ancestor of u. □

Lemma 15.2 (Characterization of cut vertices)**.** *A non-root vertex u is a cut vertex exactly when some child subtree T_v has no back edge to an ancestor of u. The root is a cut vertex exactly when it has at least two children.*

Proof. For a non-root u, suppose a child subtree has no such back edge. By Lemma 15.1, deleting u removes all its connections to the rest of the graph. Both T_v and the part containing the root are nonempty, so the graph becomes disconnected.

Conversely, suppose every child subtree has a back edge to an ancestor of u. The vertices outside T_u remain connected by their tree paths to the root, none of which uses u. Each child subtree is internally connected by its tree edges and has a back edge to this outside part. Hence all vertices remaining after deletion are connected. This includes the case of no children.

For the root, distinct child subtrees have no edges between them, by the back-edge property. Deleting the root therefore leaves one component for each child subtree. If there are at least two, it is a cut vertex. If there is one, the remaining tree is connected; if there are none, there are no remaining vertices. Neither of the last two cases increases the number of components. □

15.3 Auxiliary problem

To test the subtree condition efficiently, summarize how high a back edge from each subtree can reach.

Define

$$\mathrm{low}[u] = \min\Big(\{\mathrm{start}[u]\} \cup \{\mathrm{start}[a] : xa \text{ is a back edge},\ x \in T_u,\ a \text{ is an ancestor of } x\}\Big).$$

The parent edge is a tree edge and is excluded from this definition.

Subtree Low Values

Input: A connected graph G and its DFS parent, start, and finish arrays.
Output: The value $\mathrm{low}[u]$ for every vertex u.

Lemma 15.3 (Bottom-up recurrence for low)**.** *The value* low$[u]$ *is the minimum of* start$[u]$*, all* low$[v]$ *for children* v *of* u*, and all* start$[a]$ *for back edges from* u *itself to an ancestor* a*.*

Proof. A back edge counted in the definition starts either at u or inside one of its child subtrees. The first class supplies the direct back-edge terms. The second class is represented in the corresponding child's low value. Those child values also contain their own start times, but these are larger than start$[u]$ and cannot lower the minimum incorrectly. Taking the minimum therefore gives exactly the defining set of possibilities. □

Children have smaller finish times than their parents. To process vertices in that order, place each vertex u in an array at index finish$[u]$, then scan the array from left to right. All finish times lie in $\{1, \ldots, 2n\}$, so this takes $O(n)$ time.

Algorithm LowValues$(G, \text{parent}[1..n], \text{start}[1..n], \text{finish}[1..n])$

```
 1 initialize F[1..2n] ← ⊥
 2 for each u ∈ V do F[finish[u]] ← u
 3 for t = 1, ..., 2n with F[t] ≠ ⊥ do
 4    u ← F[t]
 5    low[u] ← start[u]
 6    for each v ∈ Adj[u] do
 7       if parent[v] = u then
 8          low[u] ← min(low[u], low[v])
 9       else if v ≠ parent[u] and start[v] < start[u]
10          low[u] ← min(low[u], start[v])
11 return low
```

Lemma 15.4 (Correctness of low values)**.** LowValues *solves the* Subtree Low Values *problem.*

Proof. By induction over vertices in increasing order of finish time. All children of the current vertex have already been processed, so their low values are correct by induction. The other updates consider exactly the non-tree edges from the current vertex to ancestors: the parent edge is excluded, and a smaller start time distinguishes an ancestor from a descendant. Lemma 15.3 now proves the current low value is correct, completing the induction.

All loops traverse finite lists, so the algorithm terminates. □

Lemma 15.5 (Running time of LowValues)**.** LowValues *runs in* $O(n + m)$ *time.*

Proof. Building and scanning F costs $O(n)$. The remaining loops visit each vertex and scan each adjacency list once, performing constant work per entry. Including allocation of the output array, the total cost is $O(n + m)$. □

15.4 Algorithm

Use Boolean flags for membership in the output set, so marking the same vertex more than once takes constant time per attempt.

Algorithm CutVertices(G)

```
 1 r ← any vertex of G
 2 (parent, start, finish) ← DFSTree(G, r)
 3 low ← LowValues(G, parent, start, finish)
 4 set all cut-vertex flags to false
 5 for each u ∈ V do
 6    c ← 0
 7    for each v ∈ Adj[u] do
 8       if parent[v] = u then
 9          c ← c + 1
10          if parent[u] ≠ ⊥ and low[v] ≥ start[u]
               then mark u
11    if parent[u] = ⊥ and c ≥ 2 then mark u
12 return the marked vertices
```

Theorem 15.6 (Correctness)**.** CutVertices *solves the* Cut Vertices *problem.*

Proof. The DFS data have the properties proved in the DFS Trees interlude, and the low values are correct by Lemma 15.4. For a child v of u, a back edge from T_v reaches an ancestor of u exactly when $\text{low}[v] < \text{start}[u]$. Indeed, start times increase along every root-to-leaf path, and every exit from T_v goes to an ancestor by Lemma 15.1.

Consequently the test with $\geq$ marks a non-root u exactly when some child subtree has no such exit. The root is marked exactly when it has at least two children. Lemma 15.2 therefore proves that all and only the cut vertices are returned. The subroutines terminate and the remaining loops are finite. □

Theorem 15.7 (Running time)**.** CutVertices *runs in* $O(n + m)$ *time.*

Proof. Computing the DFS tree and the low values takes $O(n + m)$ by the *Running time of DFSTree* theorem and Lemma 15.5. The remaining loops visit each vertex and scan each adjacency list once, performing constant work per entry. Initializing flags and collecting marked vertices costs $O(n)$. Adding the costs gives $O(n + m)$. □

Chapter 16

Cut Edges

The same DFS low values also identify cut edges. This closely related application shows how a small change in the property being tested changes a non-strict comparison into a strict one.

Cut Edges

Input: A connected graph G.
Output: All edges whose removal increases the number of components.

A cut edge is also called a *bridge*.

16.1 Main idea

A tree edge uv, with parent u and child v, is a cut edge precisely when the subtree T_v has no other edge connecting it to the rest of the graph. A back edge to u itself now suffices to keep that subtree connected: we are deleting the edge uv, not the vertex u.

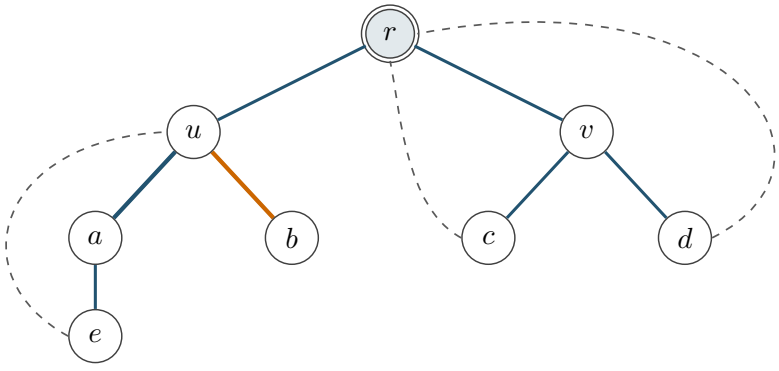


Figure 16.1: The same DFS tree as in the Cut Vertices chapter. The orange edge ub is a cut edge: deleting it isolates b. The thick blue edge ua is not a cut edge: the back edge eu keeps the subtree $\{a, e\}$ connected to u. Thus u can be a cut vertex even though ua is not a cut edge.

16.2 Lemmas

Use the DFS tree, subtree notation T_v, and low values from Chapter 15.

Lemma 16.1 (Characterization of cut edges)**.** *A tree edge uv, where u is the parent of v, is a cut edge exactly when T_v has no back edge to u or an ancestor of u. No non-tree edge is a cut edge.*

Proof. If there is no such back edge, Lemma 15.1 says uv is the only edge between T_v and its nonempty complement. Its deletion disconnects these two sets. If such a back edge exists, it reconnects the two tree pieces left after deleting uv, so the graph remains connected. Finally, deleting a non-tree edge leaves the entire spanning tree intact, so cannot disconnect the graph. □

16.3 Algorithm

Algorithm CutEdges(G)

```
1 r ← any vertex of G
2 (parent, start, finish) ← DFSTree(G, r)
3 low ← LowValues(G, parent, start, finish)
4 create an empty edge list B
5 for each v ∈ V with parent[v] ≠ ⊥ do
6     u ← parent[v]
7     if low[v] > start[u] then append uv to B
8 return B
```

Theorem 16.2 (Correctness)**.** CutEdges *solves the* Cut Edges *problem.*

Proof. The DFS data have the properties proved in the DFS Trees interlude, and the low values are correct by Lemma 15.4. For a child v of u, $\text{low}[v] \le \text{start}[u]$ holds exactly when a back edge from T_v reaches u or an ancestor of u. This follows from the definition of low and the increasing start times along an ancestor chain.

Thus the test with $>$ succeeds exactly when no such back edge exists. Lemma 16.1 proves that these are exactly the tree edges that are cut edges, and that no non-tree edge is a cut edge. Each tree edge is considered once, at its child, so the returned list is exact. The subroutines terminate and the final loop is finite. □

Theorem 16.3 (Running time)**.** CutEdges *runs in $O(n + m)$ time.*

Proof. Computing the DFS tree and low values takes $O(n + m)$. The final loop performs constant work for each non-root vertex, including at most one list append. It costs $O(n)$, so the total is $O(n + m)$. □

Chapter 17

Directed Reachability

A *directed graph* $G = (V, E)$ has ordered pairs of vertices as edges. An edge (u, v), written $u \to v$, goes from u to v; it does not imply that $v \to u$ is present. We allow edges in both directions and self-loops $u \to u$, but no repeated edges.

A *directed path* is a sequence of distinct vertices whose consecutive vertices are joined by edges in the indicated direction. We write $u \rightsquigarrow_G v$ when such a path exists, allowing a path of length zero from a vertex to itself. A *directed walk* may repeat vertices. Deleting the portions between repeated occurrences turns any directed walk into a path with the same endpoints.

Directed Reachability

Input: A directed graph G and a starting vertex s.
Output: A Boolean array indicating whether there is a directed path from s to each vertex in G.

The outgoing adjacency list $\text{Adj}_G[u]$ contains all vertices v with $u \to v$. Each edge appears in exactly one list, so the lists contain m entries altogether. We continue to write $n = |V|$ and $m = |E|$.

17.1 Main idea

The DFS algorithm from Chapter 14 also solves Directed Reachability without any changes: use the outgoing adjacency lists of G.

Starting at s, systematically follow outgoing edges and mark each vertex when it is first visited. The resulting flags identify all vertices reachable from s.

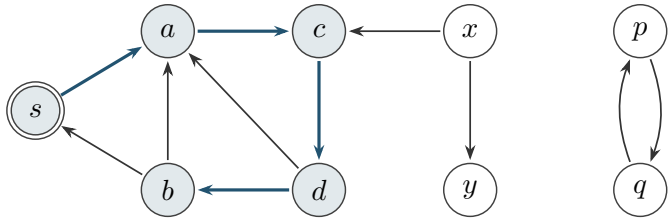


Figure 17.1: The shaded vertices s, a, b, c, d are reachable from s; thick arrows show one DFS search. Vertex x has an edge into the reached set, but none out of that set leads to x. Vertex y can be reached from x, but x is itself unreachable from s. The pair p, q is disconnected from the rest of the graph even when directions are ignored.

17.2 Algorithm

The algorithm is simply DFS(G, s). Its proofs of correctness and running time are identical for directed graphs; we include them here for completeness.

Lemma 17.1 (Correctness)**.** DFS *solves the* Directed Reachability *problem.*

Proof. In the call to DFS, each vertex has at most one recursive call, and every call scans a finite list, so the algorithm terminates. Correctness is established by proving two claims.

Every visited vertex is reachable from s: By induction over the order in which vertices are first visited. The initial vertex s is reachable by a path of length zero. Every later vertex v is visited along an edge $u \to v$ from an already visited vertex u. By induction, u is reachable from s, so following this edge also makes v reachable.

Every vertex reachable from s is visited before termination: Fix a directed path $s = v_0, v_1, \ldots, v_k$. By induction over i. The initial call visits $v_0 = s$. If v_{i-1} is visited, its call examines $v_{i-1} \to v_i$ before returning. It either finds v_i already visited or visits it recursively. Thus every vertex on the path is visited. □

Theorem 17.2 (Running time)**.** DFS *runs in time* $O(n + m)$.

Proof. Initializing the flags costs $O(n)$. Each visited vertex has one recursive call, and each outgoing adjacency list is scanned at most once. There are m adjacency entries altogether, each taking constant time to process. The total running time is therefore $O(n + m)$. □

Chapter 18

Strong Connectivity

A directed graph is *strongly connected* if, for every pair of vertices u, v, there is a directed path from u to v.

Strong Connectivity
Input: A directed graph G.
Output: Whether every vertex can reach every other vertex.

18.1 Main idea

Fix a vertex s. If every vertex can reach s and s can reach every vertex, then any journey from u to v can go through s. A search from s checks the paths *from* s. To check the paths *to* s, reverse every edge. The resulting *reversed graph* is

$$G^R = (V, E^R), \qquad E^R = \{(v, u) : (u, v) \in E\}.$$

A path from v to s in G becomes a path from s to v in G^R. Thus a second search from s in G^R checks whether every vertex can reach s in the original graph.

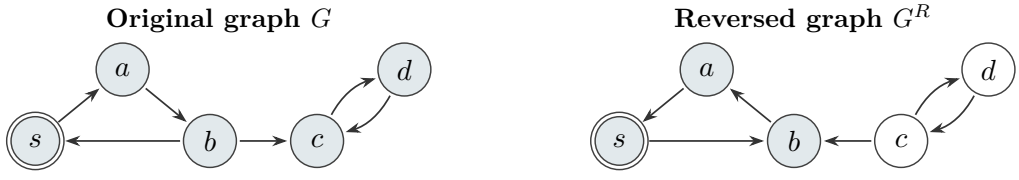


Figure 18.1: Shading shows the vertices reached from s. The first search reaches every vertex; the reversed search misses c, d, which cannot reach s in G. Thus G is not strongly connected.

18.2 Lemmas

Lemma 18.1 (Reversing paths)**.** *For every $u, v \in V$, we have $u \rightsquigarrow_G v$ if and only if $v \rightsquigarrow_{G^R} u$.*

Proof. Reversing the edges of a directed path from u to v gives a directed path from v to u in G^R. Since reversing G^R recovers G, the same argument proves the converse, including paths of length zero. □

Lemma 18.2 (One fixed vertex suffices)**.** *Fix any $s \in V$. The graph G is strongly connected if and only if $s \rightsquigarrow_G v$ and $v \rightsquigarrow_G s$ for every $v \in V$.*

Proof. If G is strongly connected, both reachability statements follow directly from its definition. Conversely, suppose both hold for every vertex. For arbitrary u, v, follow a path from u to s and then one from s to v. This gives a directed walk, and hence a directed path, from u to v. Thus G is strongly connected. □

18.3 Algorithm

Both calls to DFS start at the same vertex s, each with fresh visited flags.

Algorithm StronglyConnected(G)

```
1 choose any s ∈ V
2 F ← DFS(G, s)
3 G^R ← the reversal of G
4 B ← DFS(G^R, s)
5 for each v ∈ V do
6     if not F[v] or not B[v] then return false
7 return true
```

To implement reversal, initialize an empty outgoing list for each vertex and, for every edge $u \to v$ of G, insert u into v's list.

Theorem 18.3 (Correctness)**.** StronglyConnected *solves the* Strong Connectivity *problem.*

Proof. By Lemma 17.1, $F[v]$ is true exactly when $s \rightsquigarrow_G v$, and $B[v]$ is true exactly when $s \rightsquigarrow_{G^R} v$. The latter condition is equivalent to $v \rightsquigarrow_G s$ by Lemma 18.1. The final check returns true exactly when both conditions hold for every vertex. Lemma 18.2 proves that this is equivalent to strong connectivity. Both searches terminate. Reversal and the final check scan finite lists, so the whole algorithm terminates. □

Theorem 18.4 (Running time)**.** StronglyConnected *runs in $O(n + m)$ time.*

Proof. Each search takes $O(n + m)$ time by Theorem 17.2. Reversal initializes n lists and inserts one entry per edge, taking $O(n + m)$ time. The final check costs $O(n)$. Adding these costs gives $O(n + m)$. □

Chapter 19

Directed Acyclic Graphs

A *directed cycle* is a sequence $v_0, v_1, \ldots, v_{k-1}, v_0$, where $k \geq 1$, the first k vertices are distinct, and every step follows a directed edge. In particular, a self-loop is a cycle of length one. A directed graph is *acyclic* if it has no directed cycle; such a graph is called a *DAG*.

DAG Recognition

Input: A directed graph G.
Output: Whether G is acyclic.

19.1 Main idea

In a DAG, listing vertices in decreasing order of DFS finish time makes every edge point forwards. This gives a way to recognize acyclic graphs: run DFS, then check whether every edge follows this ordering.

A directed acyclic graph G **After adding $f \to c$**

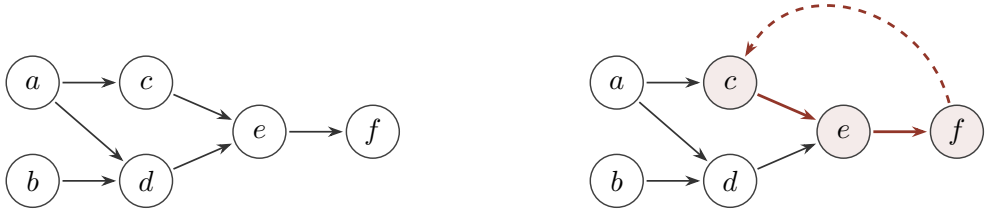


Figure 19.1: One order of increasing finish time is f, e, c, d, a, b. Reversing it gives b, a, d, c, e, f, with every edge pointing forwards in the left graph. Adding $f \to c$ leaves this order of finish times unchanged, but the new edge points backwards and closes the cycle $c \to e \to f \to c$.

19.2 Lemmas

Definition 19.1 (Topological ordering)**.** Let $G = (V, E)$ be a directed graph with n vertices. A *topological ordering* of G is a list $v_1, \ldots, v_n$ containing every vertex exactly once such that $(v_i, v_j) \in E$ implies $i < j$. Equivalently, if $p[v]$ denotes the position of v in the list, then $p[u] < p[v]$ for every edge $u \to v$.

Lemma 19.2. *Every directed graph with a topological ordering is acyclic.*

Proof. Let $p[v]$ be the position of v in a topological ordering. A directed cycle $v_0 \to v_1 \to \cdots \to v_{k-1} \to v_0$ would imply

$$p[v_0] < p[v_1] < \cdots < p[v_{k-1}] < p[v_0],$$

which is impossible. Thus there is no directed cycle. □

Lemma 19.3 (Finish times in a DAG)**.** *In a DFS of an acyclic directed graph G, for every edge $u \to v$, the vertex v finishes before u.*

Proof. Consider the moment when the call at u examines $u \to v$. If v is unvisited, the search recursively visits v and waits for that call to return before continuing. Hence v finishes before u. If v has already finished, the conclusion also holds.

The only remaining possibility is that v is visited but its call has not yet finished. The unfinished calls form the current chain of recursive calls, ending at u. Thus either $v = u$ or v is an ancestor of u in that chain. In the first case, $u \to v$ is a directed cycle of length one. In the second, the chain supplies a directed path from v to u through distinct vertices, and $u \to v$ closes a directed cycle. Both cases contradict acyclicity, so this possibility cannot occur. □

19.3 Algorithm

A search from one vertex need not reach the whole graph. The following routine restarts DFS at each still-unvisited vertex, keeping the same arrays and clock throughout. Its parent edges would form a collection of rooted trees, called a *DFS forest*. We need only the start and finish times. The local procedure shares the arrays, graph, and clock. Unlike in an undirected DFS tree, non-tree edges may join different branches or trees; the proof below uses the order of recursive calls.

Algorithm DFSTimes(G)

```
1 set start[v], finish[v] ← ⊥ for all v ∈ V; t ← 0
2 procedure Explore(u)
3     t ← t + 1; start[u] ← t
4     for each v ∈ Adj_G[u] do
5         if start[v] = ⊥ then Explore(v)
6     t ← t + 1; finish[u] ← t
7 for each u ∈ V do
8     if start[u] = ⊥ then Explore(u)
9 return (start, finish)
```

Lemma 19.4 (DFS times)**.** DFSTimes *assigns start and finish times to every vertex, using each integer from* 1 *to* $2n$ *exactly once, and runs in time* $O(n + m)$*.*

Proof. A call assigns its start time immediately, so each vertex has at most one call. Every call scans a finite list, so all calls terminate. The outer loop calls every vertex not reached earlier. Thus each vertex has exactly one call, and the clock advances once at its start and once at its finish, giving the claimed times.

Initializing the arrays and scanning the outer loop costs $O(n)$. Each adjacency list is scanned once, processing m entries altogether. All other work is constant per vertex, so the total is $O(n + m)$. □

A topological ordering must place the tail of every edge before its head. In decreasing order of finish time, this means that every edge $u \to v$ must satisfy $\text{finish}[u] > \text{finish}[v]$. We can check this directly, without constructing or sorting the ordering.

Algorithm IsDAG(G)

```
1 (start, finish) ← DFSTimes(G)
2 for each u ∈ V do
3     for each v ∈ Adj_G[u] do
4         if finish[u] ≤ finish[v] then return false
5 return true
```

Theorem 19.5 (Correctness)**.** IsDAG *solves the* DAG Recognition *problem.*

Proof. The call to DFSTimes terminates by Lemma 19.4, and the remaining loops scan finite lists. Correctness is established by proving two claims.

If G is acyclic, the algorithm returns true: For every edge $u \to v$, Lemma 19.3 gives $\text{finish}[u] > \text{finish}[v]$. No edge fails the check, so the algorithm returns true.

If the algorithm returns true, G is acyclic: All finish times are distinct by Lemma 19.4. List the vertices in decreasing order of finish time. Every edge $u \to v$ passes the check, so u precedes v in this list. It is therefore a topological ordering, and Lemma 19.2 implies that G is acyclic. □

Theorem 19.6 (Running time)**.** IsDAG *runs in time* $O(n + m)$*.*

Proof. The call to DFSTimes takes $O(n + m)$ time by Lemma 19.4. The remaining loops examine at most n vertices and m adjacency entries, with constant work per entry. Adding these costs gives $O(n + m)$. □

Chapter 20

Strongly Connected Components

A *strongly connected component* is a maximal set of vertices that can all reach one another.

Strongly Connected Components

Input: A directed graph G.
Output: Labels comp$[v]$ such that two vertices have the same label exactly when each can reach the other, and the number of components.

20.1 Main idea

A search in a component with no outgoing edges to other components reaches exactly that component. The last vertex to finish a DFS of the whole graph belongs to a component with no incoming edges from other components. Reversing the graph turns such a component into one with no outgoing edges. This suggests two passes: first record finish times in G, then search G^R, always starting at the unvisited vertex with largest finish time.

The *component graph* has one vertex for each strongly connected component. It has an edge $C \to D$, for $C \neq D$, when some edge of G goes from a vertex of C to a vertex of D.

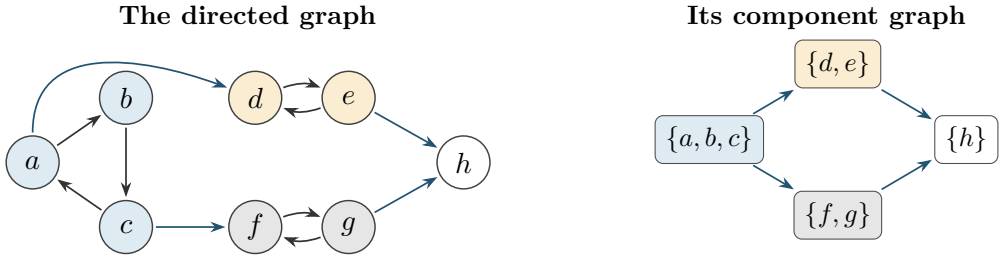


Figure 20.1: The component graph is acyclic. One possible order of increasing finish time is h, g, f, c, b, e, d, a. Searching G^R in decreasing order of finish time, while skipping labelled vertices, starts searches at a, d, f, h and recovers the four components shown.

20.2 Lemmas

Lemma 20.1 (Components and paths)**.** *The strongly connected components partition V. Every directed path between vertices of one component stays in that component. The graphs G and G^R have the same strongly connected components.*

Proof. Define $u \sim v$ when $u \rightsquigarrow_G v$ and $v \rightsquigarrow_G u$. Length-zero paths give reflexivity, and symmetry follows from the definition. Concatenating paths proves transitivity, after deleting repeated portions if necessary. Thus $\sim$ is an equivalence relation. Its classes are exactly the maximal sets of mutually reachable vertices, so they form the required partition.

If a path from u to v passes through x, where $u \sim v$, then $u \rightsquigarrow_G x$ and $x \rightsquigarrow_G v \rightsquigarrow_G u$. Hence $x \sim u$, proving the path claim. Finally, reversal exchanges the two reachability conditions by Lemma 18.1, so it preserves the classes. □

Lemma 20.2 (Acyclic component graph)**.** *The component graph is acyclic.*

Proof. A directed cycle of distinct components would make every component on the cycle reachable from every other: follow its intercomponent edges, joining them by paths inside each component. All vertices on that cycle would then be mutually reachable, contradicting that they lie in distinct strongly connected components. □

Run DFS on all of G, and write $f(C) = \max\{\text{finish}[v] : v \in C\}$ for the latest finish time in component C.

Lemma 20.3 (Finish times of components)**.** *If the component graph has an edge $C \to D$, then $f(C) > f(D)$.*

Proof. Let x be the vertex with the smallest start time in $C \cup D$. If $x \in C$, every vertex of $C \cup D$ is reachable from x by a path inside this union. All other vertices of the union are still unvisited, so DFS visits them before the call at x returns. Their calls finish before that call finishes. Hence $f(C) = \text{finish}[x] > f(D)$.

If $x \in D$, every vertex of D is reached and finishes by the time the call at x returns. No vertex of C can be reached from D, since $C \to D$ would then make the two components mutually reachable. Thus all vertices of C are still unvisited when x finishes, giving $f(C) > f(D)$ again. □

20.3 Algorithm

Use DFSTimes from the preceding chapter. To read vertices in decreasing order of finish time, set $F[\text{finish}[v]] = v$ in an initially empty array $F[1..2n]$, then scan it backwards. This takes $O(n)$ time and requires no sorting.

In the second pass, component labels also serve as visited flags. Initialize them only once. The local procedure shares G^R, comp, and the current label c with its caller.

Algorithm StrongComponents(G)

```
(start, finish) ← DFSTimes(G)
initialize F[1..2n] ← ⊥
for each v ∈ V do F[finish[v]] ← v
G^R ← the reversal of G
set comp[v] ← 0 for all v; c ← 0
procedure Label(u)
    comp[u] ← c
    for each v ∈ Adj_{G^R}[u] do
        if comp[v] = 0 then Label(v)
for t = 2n, ..., 1 with F[t] ≠ ⊥ do
    s ← F[t]
    if comp[s] = 0 then
        c ← c + 1; Label(s)
return (comp, c)
```

Reversal is implemented as in the Strong Connectivity chapter: initialize empty outgoing lists and insert u into v's list for each edge $u \to v$.

Theorem 20.4 (Correctness)**.** StrongComponents *solves the* Strongly Connected Components *problem.*

Proof. By induction over completed searches in the second pass. Initially no component has been labelled. Suppose the labelled vertices are exactly the union of some complete components, with a different positive label on each. Let s be the next unlabelled starting vertex, and let C be its component. The whole of C is unlabelled.

By the order of the outer loop, s has the largest finish time among unlabelled vertices. Hence $f(C) = \text{finish}[s]$ is larger than $f(D)$ for every other unlabelled component D. There can be no edge $D \to C$ from such a component, since Lemma 20.3 would give $f(D) > f(C)$. Thus in G^R, C has no edge to another unlabelled component.

By Lemma 20.1, all vertices of C can be reached from s within C in G^R. The next search therefore labels all of C. Every outgoing edge from C to another component leads to an already labelled vertex and is skipped, so it labels nothing outside C. This preserves the induction assertion.

Every vertex appears once in F, so all vertices are labelled by the end of the outer loop. The labels give exactly the components, and c counts them. Both DFS passes call each vertex at most once and scan finite lists; the other loops and reversal are finite. Thus the algorithm terminates. □

Theorem 20.5 (Running time)**.** StrongComponents *runs in* $O(n + m)$ *time.*

Proof. Each DFS visits every vertex once and scans each outgoing adjacency list once. Its other per-vertex work—clock updates in the first pass, label assignments in the second—is constant. Thus each pass costs $O(n + m)$, as does reversal.

Each pass initializes its shared arrays only once, and building and scanning F take $O(n)$ time. These additional costs are $O(n)$, giving $O(n + m)$ time altogether. □

Part III

Greedy Algorithms

Chapter 21

Interval Scheduling

An interval $I_i = [s_i, f_i]$ has integer endpoints $s_i < f_i$. Throughout the scheduling chapters, intervals are closed: sharing an endpoint counts as an overlap.

Interval Scheduling

Input: An array of intervals I.
Output: A largest set of pairwise disjoint intervals.

The array length is n, and I_i denotes its ith interval. When we reorder records, they retain their original indices.

21.1 Main idea

A *greedy algorithm* commits to one choice at a time. Choose the interval that finishes first, then repeat among intervals starting after it finishes. Finishing earlier leaves at least as many possibilities for the remaining choices.

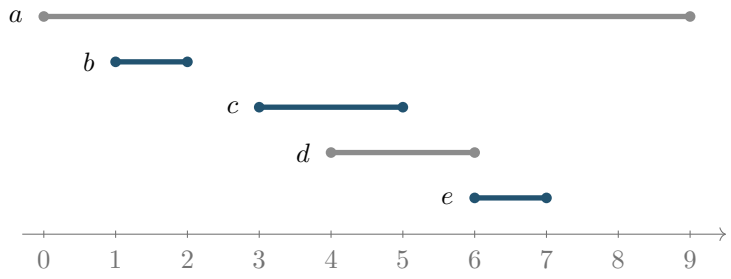


Figure 21.1: The greedy choices are $b = [1, 2]$, $c = [3, 5]$, and $e = [6, 7]$. The long interval a would exclude all the others.

21.2 Lemmas

Lemma 21.1 (An earliest finish is safe)**.** *In any nonempty collection of intervals, an interval g with smallest finishing time belongs to a largest pairwise disjoint subcollection.*

Proof. Take a largest disjoint subcollection and let o be its first interval in time order. Replace o by g. Every later interval starts after f_o, and $f_g \leq f_o$, so the replacement introduces no overlap. The number of chosen intervals is unchanged, proving the claim. □

21.3 Algorithm

Sort indices by finishing time, comparing the corresponding endpoints in MergeSort. Keep the original interval records attached to their indices. Only the last selected finishing time is needed to test compatibility.

Algorithm ScheduleIntervals($I[1..n]$)

```
1 order the indices as π_1, ..., π_n with f_{π_1} ≤ ··· ≤ f_{π_n}
2 create an empty list A; t ← −∞
3 for k = 1, ..., n do
4     i ← π_k
5     if t < s_i then
6         append i to A; t ← f_i
7 return A
```

The formal $-\infty$ makes the first interval eligible. Break sorting ties arbitrarily.

Theorem 21.2 (Correctness)**.** ScheduleIntervals *solves the* Interval Scheduling *problem.*

Proof. By induction over the number of selected intervals. We show that the current list is a prefix, in time order, of some optimal solution. The empty list has this property.

Suppose the assertion holds. Let t be the last selected finishing time, or $-\infty$ for the empty prefix. The intervals that may follow the prefix are precisely those starting after t. None was skipped earlier: an interval skipped earlier starts no later than an earlier selected finish, and hence no later than t.

The suffix of an optimal solution extending the prefix must be optimal among these remaining intervals; otherwise replacing it would improve the whole solution. The next selected interval has smallest finishing time among them. By Lemma 21.1, there is an optimal suffix beginning with that interval. This proves the inductive step.

When the scan ends, no interval can follow the selected list. Its optimal extension therefore has no additional intervals, so the list itself is optimal. The tests also ensure that its intervals are pairwise disjoint. Sorting terminates and the scan has n iterations, proving termination. □

Theorem 21.3 (Running time)**.** ScheduleIntervals *runs in time* $O(n \log n)$*.*

Proof. MergeSort orders the indices in $O(n \log n)$ time. Each comparison reads two finishing times and takes constant time. The scan performs constant work per interval and at most n list appends, for $O(n)$ additional time. Adding these costs proves the bound. □

Interlude

Priority Queues

A queue processes items by arrival order. A *minimum-priority queue* instead selects an item with smallest key. The order can change when new items arrive or existing keys decrease.

Each item has an identifier and a key, which is an integer or formal $+\infty$. Equal keys are allowed, and any minimum item may be chosen. The operations used below are:

- $\textsc{Insert}(Q, x, k)$ inserts a new item x with key k.
- $\textsc{MinKey}(Q)$ returns the smallest key without removing an item.
- $\textsc{ExtractMin}(Q)$ removes and returns a minimum-key item.
- $\textsc{DecreaseKey}(Q, x, k)$ replaces x's key by a smaller key k.

Minimum and extraction require a nonempty queue; decrease requires an item already present. An item can be inserted again after its removal.

Priority Queue Maintenance

Input: A valid sequence of these operations, starting with an empty queue.
Output: The required minimum keys and items, respecting all insertions, removals, and key decreases.

A *binary min-heap* stores the items in a complete binary tree: all levels except possibly the last are full, and the last is filled from left to right. Every parent's key is at most each child's key. Consequently the root has a minimum key.

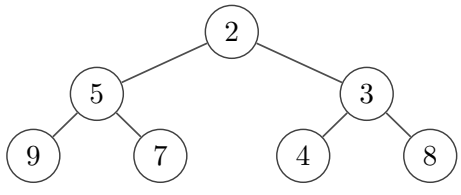


Figure 21.2: A heap orders parents against their children. It need not order siblings or an entire level.

Store the tree in an array in level order. Its parent and child positions can be computed in constant time. Identifiers lie in $\{1, \ldots, N\}$; a position array locates each present item and marks absent items. Every swap updates the two positions.

- *Insert:* append the item at the next tree position. While its key is smaller than its parent's, swap it with its parent.

- *Decrease:* lower the item's key and perform the same upward swaps.
- *Extract:* save the root item, move the last item to the root, and remove the last position. While the replacement is larger than a child, swap it with a smallest-key child. Return the saved item.
- *Minimum:* read the root's key.

Removing the only item simply leaves an empty heap. In Figure 21.2, decreasing 7 to 1 moves that item past 5 and then 2 to the root.

Theorem (Binary heap correctness)**.** *The binary-heap operations solve the* Priority Queue Maintenance *problem.*

Proof. By induction over the operations. We maintain the complete-tree shape, the parent–child key inequalities, and the correct item positions. The empty heap satisfies these conditions.

Insertion adds the next position, preserving the shape. A key decrease leaves the shape unchanged. In either case, only the changed item's edge to its parent can violate heap order. An upward swap fixes that edge; the displaced parent remains no larger than the children it acquires. Any remaining violation is again between the changed item and its new parent. Thus the upward loop restores heap order.

Extraction removes exactly the root item and the last position, so the remaining shape is complete. Only edges from the replacement item to its children can violate heap order. Swapping with a smallest child fixes the order above and beside it; only its new child edges can still violate the order. The downward loop therefore restores heap order. Each loop moves through distinct levels, so it terminates.

Swaps preserve the items and their keys, and the prescribed updates preserve their positions. The parent–child inequalities imply that the root is a minimum. Thus minimum and extraction return valid answers, completing the induction. □

Theorem (Binary heap running time)**.** *For at most N items, minimum-key and membership queries take $O(1)$ time; insertion, extraction, and decrease-key take $O(\log N)$ time. Initializing an empty queue takes $O(N)$ time, and building a queue of N given items by insertion takes $O(N \log N)$ time.*

Proof. A complete binary tree of height h has at least 2^h vertices, since its first h levels are full and its last level is nonempty. Thus $h \leq \log N$. Each update performs constant work per level and moves along at most one root-to-leaf path. Minimum reads the root; membership reads the position array.

Initializing the arrays takes $O(N)$ time. Building the heap then makes N insertions, each costing $O(\log N)$, for $O(N \log N)$ total time. □

Chapter 22

Interval Coloring

As in Chapter 21, intervals that share an endpoint overlap. A valid coloring assigns different colors to every pair of overlapping intervals.

Interval Coloring

Input: An array of intervals I.
Output: A color for each interval, using as few colors as possible, so overlapping intervals have different colors.

22.1 Main idea

Process intervals by starting time. Reuse an available color whenever possible; introduce a new color only when all existing colors are still occupied. At that moment, one interval of every existing color overlaps the new interval.

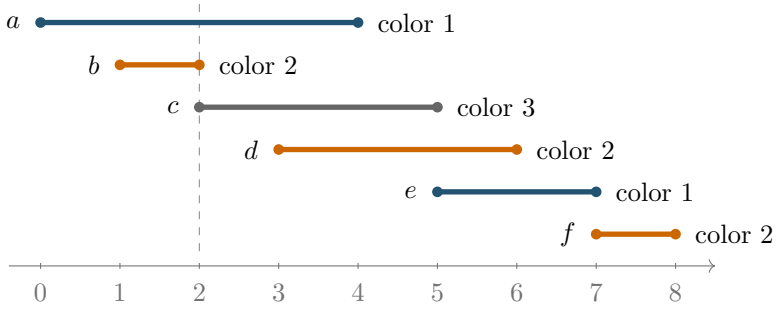


Figure 22.1: Three colors suffice. The intervals a, b, c all contain time 2, so two colors cannot suffice.

22.2 Lemmas

Lemma 22.1 (Simultaneous intervals)**.** *If q intervals contain a common time, every valid coloring uses at least q colors.*

Proof. Every pair of those intervals overlaps at that time. They must therefore receive pairwise different colors. □

22.3 Algorithm

Use two priority queues from the preceding interlude. B stores occupied colors, keyed by their last finishing times. F stores available colors, keyed by their color numbers. The latter queue lets us use the smallest available color.

Algorithm ColorIntervals($I[1..n]$)

```
order the indices as π_1, ..., π_n with s_{π_1} ≤ ··· ≤ s_{π_n}
create empty priority queues B, F; q ← 0
for k = 1, ..., n do
    i ← π_k
    while B is not empty and MinKey(B) < s_i do
        r ← ExtractMin(B); Insert(F, r, r)
    if F is empty then
        q ← q + 1; r ← q
    else r ← ExtractMin(F)
    color[i] ← r; Insert(B, r, f_i)
return (color, q)
```

Sort using MergeSort and break equal starting times arbitrarily. All queue identifiers are color numbers at most n.

Theorem 22.2 (Correctness)**.** ColorIntervals *solves the* Interval Coloring *problem.*

Proof. By induction over the processed intervals. After the while loop, B contains exactly the colors whose last assigned intervals finish at or after s_i; F contains the other existing colors. Indeed, the loop moves all and only colors with earlier finishing times into F, and each assignment puts its color back in B with its new finishing time.

Using a color from F therefore creates no overlap with an interval of that color. A new color has no earlier interval. Thus the coloring remains valid after every assignment.

Suppose the algorithm uses q colors, and consider the first interval assigned color q. At that moment F is empty. For each earlier color, its last interval starts no later than the new interval and finishes no earlier than its starting time. These $q-1$ intervals and the new interval all contain that starting time. Lemma 22.1 shows that every valid coloring needs at least q colors. The returned coloring is therefore optimal.

The outer loop has n iterations. Each pass through the inner loop removes an item from B, and there are at most n insertions into B. All heap operations terminate, so the algorithm terminates. □

Theorem 22.3 (Running time)**.** ColorIntervals *runs in time* $O(n \log n)$.

Proof. Sorting costs $O(n \log n)$ time. Initializing the output and queues costs $O(n)$. Each interval causes one insertion into B and at most one extraction from F. Each item inserted into B is removed at most once; each removal causes one insertion into F. There are therefore $O(n)$ heap updates altogether, each costing $O(\log n)$ by the *Binary heap running time* theorem. The other loop operations cost $O(n)$ in total. Adding these costs proves the bound. □

Chapter 23

Total Completion Time

Jobs run one at a time on one machine, without interruption or idle time, starting at time zero. Job i has positive integer processing time p_i. In an ordering $\pi_1, \ldots, \pi_n$, its kth job has *completion time* $\sum_{j=1}^{k} p_{\pi_j}$.

Minimum Total Completion Time

Input: An array of processing times p.
Output: An ordering of the jobs minimizing the sum of their completion times on one machine.

23.1 Main idea

Run shorter jobs first. A long job placed early delays every job after it. Exchanging a long job with the shorter job immediately after it reduces their total completion time while leaving all other completion times unchanged.



Figure 23.1: The final finishing time is 17 in both orders, but the sum of completion times is smaller when short jobs go first.

For example, swapping the first two jobs in the given order changes their completion times from $7, 9$ to $2, 9$. The later completion times remain $14, 17$, so the total decreases by 5.

23.2 Lemmas

Lemma 23.1 (Adjacent exchange)**.** *If two consecutive jobs have processing times $a > b$, swapping them decreases total completion time by $a - b$.*

Proof. Let t be the time when the pair starts. Before the swap, their completion times sum to $(t+a)+(t+a+b) = 2t+2a+b$. After the swap, they sum to $(t+b)+(t+b+a) = 2t+2b+a$. The difference is $a - b$. Earlier jobs are unchanged, and the pair still finishes at $t + a + b$, so every later completion time is unchanged. □

23.3 Algorithm

Algorithm ShortestFirst($p[1..n]$)

```
1 create the index array π[1..n] = (1, ..., n)
2 sort π by nondecreasing p_{π_k} using MergeSort
3 return π
```

Jobs with equal processing times may appear in either order.

Theorem 23.2 (Correctness)**.** ShortestFirst *solves the* Minimum Total Completion Time *problem.*

Proof. Take an optimal ordering. It cannot contain an adjacent pair of strictly decreasing processing times: Lemma 23.1 would give a strictly better ordering. Its processing times are therefore nondecreasing.

The algorithm also returns a nondecreasing ordering. All such orderings have the same sequence of processing times, differing only in the identities of equal-length jobs. Their prefix sums, and hence their total completion times, are equal. The returned ordering is thus optimal. The array construction and MergeSort terminate, proving termination. □

Theorem 23.3 (Running time)**.** ShortestFirst *runs in time $O(n \log n)$.*

Proof. Constructing the index array takes $O(n)$ time. MergeSort, with padding if needed, sorts it in $O(n \log n)$ time because each comparison reads two processing times and has constant cost. Returning the array adds constant time. □

Chapter 24

Shortest Paths with Nonnegative Weights

In a *weighted graph*, each edge e has an integer weight $w(e)$. The weight of a path or walk is the sum of its edge weights. The *weighted distance* from s to v is the minimum weight of an s–v path, or $+\infty$ if no such path exists.

Nonnegative Shortest Paths

Input: A directed graph G with nonnegative edge weights w, and a source s.
Output: Every weighted distance from s and parent pointers describing shortest paths to reachable vertices; use $+\infty$ for unreachable distances.

We store an edge's weight with its adjacency-list entry, so scanning the edge also reads its weight in constant time. This convention applies to the weighted graphs in later chapters.

24.1 Main idea

As in BFS, process vertices in increasing distance, using a priority queue to account for unequal edge weights. Keep a tentative distance for each vertex. Repeatedly settle the vertex with smallest tentative distance, then use its outgoing edges to improve the remaining distances. Nonnegative weights ensure that an unsettled vertex cannot offer a shorter route back to the vertex just settled.

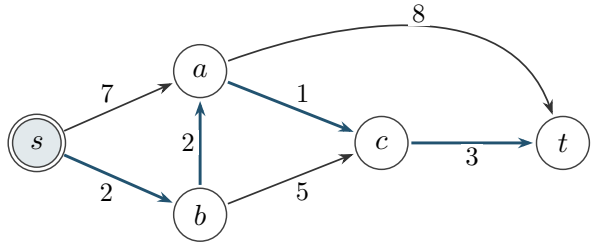


Figure 24.1: The blue edges form a shortest-path tree from s. Vertices are settled in the order s, b, a, c, t, at distances $0, 2, 4, 5, 8$.

24.2 Lemmas

Write $\delta(v)$ for the weighted distance from s to v. For a set S of reachable vertices containing s, define the *boundary distance* of $v \notin S$ by

$$b(v) = \min\{\delta(x) + w(x, v) : x \in S,\ (x, v) \in E\},$$

with minimum $+\infty$ when the set is empty.

Lemma 24.1 (A safe next vertex)**.** *Let $u \notin S$ have minimum boundary distance. If $b(u)$ is finite, then $b(u) = \delta(u)$. If $b(u) = +\infty$, every vertex outside S is unreachable from s.*

Proof. Every finite candidate $\delta(x)+w(x, v)$ describes a walk from s to v. Deleting repeated portions cannot increase its length, because all weights are nonnegative. Thus $\delta(v) \le b(v)$.

If $b(u)$ is finite, u is reachable. Choose a shortest path to u and let (x, y) be its first edge leaving S. Its prefix to x has length at least $\delta(x)$, and the part after y has nonnegative length. Therefore

$$b(u) \le b(y) \le \delta(x) + w(x, y) \le \delta(u).$$

Together with the reverse inequality, this proves equality.

If some vertex outside S is reachable, a path to it has a first edge leaving S, which gives a finite boundary distance. Hence an infinite minimum boundary distance implies that no such vertex is reachable. □

24.3 Algorithm

The queue contains the vertices not yet settled, keyed by their tentative distances. Testing whether a vertex is in the queue takes constant time using the position array from *Priority Queues*.

Algorithm DIJKSTRA(G, w, s)

```
 1 set d[v] ← +∞ and parent[v] ← ⊥ for all v ∈ V
 2 d[s] ← 0; build a priority queue Q containing each v with key d[v]
 3 while Q is not empty do
 4    u ← ExtractMin(Q)
 5    if d[u] = +∞ then break
 6    for each v ∈ Adj[u] do
 7       if v ∈ Q and d[u] + w(u, v) < d[v] then
 8          d[v] ← d[u] + w(u, v); parent[v] ← u
 9          DecreaseKey(Q, v, d[v])
10 return (d, parent)
```

Theorem 24.2 (Correctness)**.** DIJKSTRA *solves the* Nonnegative Shortest Paths *problem.*

Proof. Let S consist of the vertices whose adjacency lists have been processed. We maintain two properties: $d[x] = \delta(x)$ for every $x \in S$, and $d[v] = b(v)$ for every $v \notin S$.

The first extracted vertex is s. Nonnegative weights give $d[s] = 0 = \delta(s)$. Processing its outgoing edges sets each other label to its boundary distance for $S = \{s\}$, so the properties hold.

Suppose the properties hold before a later extraction. The queue selects a minimum boundary distance. If it is infinite, Lemma 24.1 shows that all remaining vertices are unreachable, and their infinite labels are correct. Otherwise that lemma gives $d[u] = \delta(u)$. Adding u to S introduces exactly the new boundary candidates $\delta(u) + w(u, v)$. Lines 6–9 take the minimum of each old label and its new candidate. Thus both properties are preserved. At termination every distance is correct.

Whenever a parent is assigned to v, that parent has already been settled and its distance will not change. The assignment records $d[v] = d[\text{parent}[v]] + w(\text{parent}[v], v)$. Following final parents therefore moves strictly backwards in settlement order, even when an edge has weight zero. Every finite-label vertex other than s has a parent, so the chain ends at s. Summing the recorded equalities gives a path of length $d[v] = \delta(v)$, as required.

Each iteration removes one vertex from Q, so there are at most n iterations. Every adjacency list is finite and the heap operations terminate. Hence the algorithm terminates. □

Theorem 24.3 (Running time)**.** DIJKSTRA *runs in time* $O((n + m) \log n)$.

Proof. Initializing the arrays costs $O(n)$ time. Building the queue by insertion costs $O(n \log n)$ by the *Binary heap running time* theorem. There are at most n extractions. Each adjacency list is scanned at most once, so at most m edges are examined and at most m decrease-key operations occur. Each heap update costs $O(\log n)$; all other work per examined edge is constant. Adding these costs gives $O(n + n \log n + m + m \log n) = O((n + m) \log n)$. □

Chapter 25

Minimum Spanning Trees

A *spanning tree* is a connected, acyclic subgraph containing every vertex. Its weight is the sum of its edge weights. Every connected graph has a spanning tree: repeatedly delete an edge of a cycle until no cycle remains. Such a deletion preserves connectivity.

Minimum Spanning Tree

Input: A connected graph G with positive edge weights w.
Output: A spanning tree of G with minimum total edge weight.

25.1 Main idea

Grow one tree, beginning at any vertex. At each step, add a cheapest edge joining the tree to a vertex outside it. The cut property will show that every partial tree obtained this way can still be extended to a minimum spanning tree.

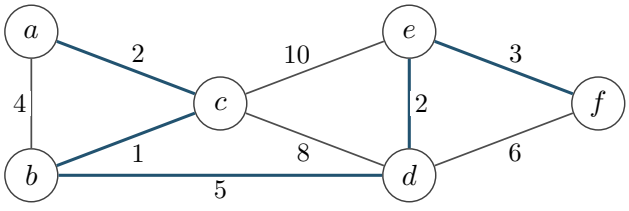


Figure 25.1: Starting at a, Prim's algorithm adds ac, cb, bd, de, ef. The resulting tree has weight $2 + 1 + 5 + 2 + 3 = 13$.

After adding ac and cb in Figure 25.1, the tree contains $\{a, b, c\}$. Its cheapest edge to the remaining vertices is bd, of weight 5. The cheaper edge de is not yet eligible: both its endpoints are outside the current tree.

25.2 Lemmas

For a nonempty proper subset $S \subset V$, the *cut* $\delta(S)$ is the set of edges with exactly one endpoint in S. A cut *respects* an edge set F when no edge of F crosses it.

Lemma 25.1 (Cut property)**.** *Suppose F is contained in some minimum spanning tree. If a cut respects F and e is a minimum-weight edge crossing that cut, then $F \cup \{e\}$ is contained in some minimum spanning tree.*

Proof. Choose a minimum spanning tree T containing F. If $e \in T$, there is nothing to prove. Otherwise, adding $e = uv$ to T creates one cycle: e together with the unique u–v path in T. The path is unique because two distinct paths would contain a cycle in T.

Since u and v lie on opposite sides of the cut, that path contains another crossing edge f. The cut respects F, so $f \notin F$. Deleting f from the cycle leaves a spanning tree $T' = T \cup \{e\} \setminus \{f\}$. Indeed, the remaining cycle edges still connect the endpoints of f, and deleting an edge of the only cycle leaves no cycle.

By the choice of e, $w(e) \le w(f)$, so $w(T') \le w(T)$. The minimality of T implies that T' is also minimum. It contains $F \cup \{e\}$, proving the claim. □

25.3 Algorithm

As in Dijkstra's algorithm, a priority queue chooses the next vertex. Here key[v] is the weight of a cheapest edge joining v to the current tree, and parent[v] records its other endpoint. Choose any root r.

Algorithm Prim(G, w, r)

```
1  set key[v] ← +∞ and parent[v] ← ⊥ for all v ∈ V
2  key[r] ← 0; build Q containing each v with key key[v]
3  create an empty edge list T
4  while Q is not empty do
5     u ← ExtractMin(Q)
6     if parent[u] ≠ ⊥ then append parent[u]u to T
7     for each v ∈ Adj[u] do
8        if v ∈ Q and w(u, v) < key[v] then
9           key[v] ← w(u, v); parent[v] ← u
10          DecreaseKey(Q, v, key[v])
11 return T
```

Theorem 25.2 (Correctness)**.** Prim *solves the* Minimum Spanning Tree *problem.*

Proof. Let S be the extracted vertices. After each iteration, we maintain three properties: T is a tree on S contained in some minimum spanning tree; each remaining key is the cheapest edge weight from S to its vertex, or $+\infty$ if no such edge exists; and each finite key has the recorded parent as a witness.

The first extracted vertex is r. No edge is added, and scanning its adjacency list establishes the key and parent properties. The empty edge set is a tree on $\{r\}$ and is contained in every spanning tree. Thus all properties hold.

Suppose they hold before a later iteration. Connectivity guarantees an edge from S to $V \setminus S$, so the minimum key is finite. The selected edge parent[u]u is a cheapest edge of

this cut. No edge of T crosses the cut, so Lemma 25.1 shows that adding the selected edge preserves containment in a minimum spanning tree. It attaches one new vertex to T, preserving the tree property. Scanning u's list compares each old key with exactly the newly available edge from u. It therefore preserves the key and parent properties for the enlarged set S.

At termination, $S = V$ and T is a spanning tree contained in a minimum spanning tree T^*. A proper subset of the edges of a tree is not connected, so $T = T^*$ and the returned tree is minimum. The loop removes one vertex per iteration, and each adjacency list and heap operation terminates. Hence the algorithm terminates. □

Theorem 25.3 (Running time)**.** Prim *runs in time* $O((n + m) \log n)$.

Proof. Initializing the arrays costs $O(n)$ time. Building the queue by insertion costs $O(n \log n)$ by the *Binary heap running time* theorem. There are n extractions. Each adjacency list is scanned once, so $2m$ adjacency entries are examined and at most $2m$ decrease-key operations occur. Each heap update costs $O(\log n)$; all other work per examined entry is constant. Adding these costs gives $O(n + n \log n + m + m \log n) = O((n + m) \log n)$. □

Interlude

Disjoint Sets

Some algorithms repeatedly join components and ask whether two vertices already belong to the same component. A *disjoint-set data structure* maintains a partition while supporting these two operations.

Initially every vertex is in its own set. The operation $\textsc{Find}(x)$ returns a representative of the set containing x; two vertices have the same representative exactly when they are in the same set. The operation $\textsc{Union}(x, y)$ merges their sets, doing nothing if they already coincide. A set's representative may change after a union.

Disjoint-Set Maintenance

Input: Vertices $1, \ldots, n$ and a sequence of find and union operations.
Output: The requested representatives, respecting all unions, starting from singleton sets.

Represent each set by a rooted tree. An array P stores parent pointers, with $P[r] = r$ at a root. The root represents the set; finding it means following parents. To join two sets, attach the smaller tree's root to the larger tree's root. Store each tree's size at its root.



Figure 25.2: Union by size attaches root 1 to root 3. Only vertices in the smaller tree gain a level.

These trees encode set membership. Their edges need not be edges of any graph whose components the data structure is tracking.

The arrays P and size are shared by the following routines. Only size entries at roots are used.

Algorithm $\textsc{MakeSets}(n)$

```
1 for v = 1, ..., n do P[v] ← v; size[v] ← 1
```

Algorithm FIND(x)
1 **while** $P[x] \neq x$ **do** $x \leftarrow P[x]$
2 **return** x

Algorithm UNION(x, y)
1 $a \leftarrow$ FIND(x); $b \leftarrow$ FIND(y)
2 **if** $a = b$ **then return**
3 **if** size$[a]$ < size$[b]$ **then** swap a and b
4 $P[b] \leftarrow a$; size$[a] \leftarrow$ size$[a]$ + size$[b]$

Theorem (Disjoint-set correctness)**.** MAKESETS, FIND, *and* UNION *solve the* Disjoint-Set Maintenance *problem.*

Proof. By induction over operations. Maintain a rooted forest whose vertex sets are exactly the current sets, with each root storing its set's size. Initialization gives singleton trees and correct sizes.

Following parents reaches the unique root of a tree, so FIND returns the required representative and leaves the invariant unchanged. If two roots agree, UNION correctly changes nothing. If they differ, linking one root to the other joins precisely the two trees and creates no cycle. The new root's size is their size sum. Thus union also preserves the invariant.

Every find follows a finite chain towards its root, so it terminates. Initialization has n iterations and union makes two finds and finitely many further operations. All three routines therefore terminate. □

Theorem (Disjoint-set running time)**.** MAKESETS *runs in time* $O(n)$*. Each* FIND *and* UNION *takes* $O(\log n)$ *time.*

Proof. Fix a vertex v. Its depth increases only when its tree is attached to another tree. If its current set has size k, the other set has size at least k, because we attach the smaller tree. After the union, v's set therefore has size at least $2k$.

Initially the set has size one. After h depth increases it has size at least 2^h, and it never has more than n vertices. Consequently $h \leq \log n$. This holds for every vertex, so each find follows at most $\log n$ parent pointers and takes $O(\log n)$ time. Union makes two finds and constant additional work, giving the same bound. Initialization makes n constant-time assignments of each kind and therefore takes $O(n)$ time. □

Chapter 26

Minimum Spanning Trees (revisited)

Recall that a minimum spanning tree connects all vertices without cycles and has the smallest possible total edge weight.

Minimum Spanning Tree (restated)
Input: A connected graph G with positive edge weights w.
Output: A spanning tree of G with minimum total edge weight.

26.1 Main idea

Consider all edges from cheapest to most expensive. Keep an edge exactly when its endpoints belong to different current components. Each kept edge joins two trees, so the chosen edges always form a forest.

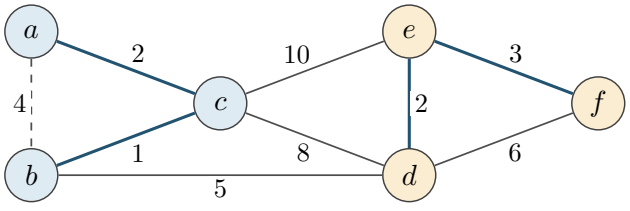


Figure 26.1: After keeping bc, ac, de, ef, there are two components. The next edge ab is skipped; bd then joins the components.

Figure 26.1 uses the graph from Figure 25.1. The edge ab would close a cycle inside $\{a, b, c\}$, so it is skipped. The edge bd produces the same minimum spanning tree of weight 13 that Prim found, though the edges are chosen in a different order.

26.2 Algorithm

Use the disjoint-set routines from the preceding interlude. List each undirected edge once, for example from its smaller-numbered endpoint, and sort this list using MergeSort. An empty list needs no sorting.

Algorithm KRUSKAL(G, w)

```
1 create an empty edge list T; MakeSets(n)
2 order the edges as e_1, ..., e_m with w(e_1) ≤ ··· ≤ w(e_m)
3 for i = 1, ..., m do
4     let e_i = uv
5     if Find(u) ≠ Find(v) then
6         append uv to T; Union(u, v)
7 return T
```

Theorem 26.1 (Correctness)**.** KRUSKAL *solves the* Minimum Spanning Tree (restated) *problem.*

Proof. By induction over the scanned edges. Maintain two properties: T is a forest contained in some minimum spanning tree, and the disjoint sets are exactly the connected components of T. Initially T is empty and all components are singletons, so both properties hold.

An edge within one component is skipped, preserving both properties. Now suppose $e_i = uv$ joins different components, and let C be u's current component. No edge of T crosses $\delta(C)$. Moreover, no previously scanned edge crosses this cut. If one did, its endpoints would have belonged to different components when scanned, so the algorithm would have joined them. They would still be in the same component now, a contradiction.

Every edge crossing $\delta(C)$ is therefore unprocessed or is e_i itself. Since the scan is in nondecreasing weight order, e_i is a minimum-weight edge of this cut. Lemma 25.1 shows that adding it preserves containment in a minimum spanning tree. It joins two trees without creating a cycle, and UNION merges exactly their sets. Both properties are preserved.

At the end, T is connected. Otherwise, connectivity of G would give an edge between two final components. Its endpoints were also in different components when it was scanned, so it would have joined them, a contradiction. Thus T is a spanning tree contained in a minimum spanning tree. As in Prim's proof, it must itself be minimum.

The scan has m iterations, and MergeSort and the disjoint-set routines terminate. Hence the algorithm terminates. □

Theorem 26.2 (Running time)**.** KRUSKAL *runs in time* $O((n + m) \log n)$.

Proof. Listing the edges and initializing the disjoint sets costs $O(n + m)$ time. Sorting costs $O(m \log m)$; lists with at most one edge need only constant time. The graph is simple, so $m \le n(n-1)/2 < n^2$ and $\log m \le 2 \log n$ when $m \ge 1$. Thus sorting costs $O(1 + m \log n)$.

Each of the m iterations makes two finds and at most one union. By the *Disjoint-set running time* theorem, these cost $O(\log n)$ per iteration; appending an edge costs constant time. Adding the costs gives $O(n + m + m \log n) = O((n + m) \log n)$. □

Part IV

Dynamic Programming

Prelude

Dynamic Programming

Dynamic programming replaces repeated recursive work by a table of answers to subproblems. The central choices are what each entry means and in what order to compute the entries.

The Fibonacci numbers satisfy $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$. A direct recursive evaluation expands each occurrence in the recurrence tree separately. For example, computing F_6 leads to two computations of F_4, with identical recursive work below them.

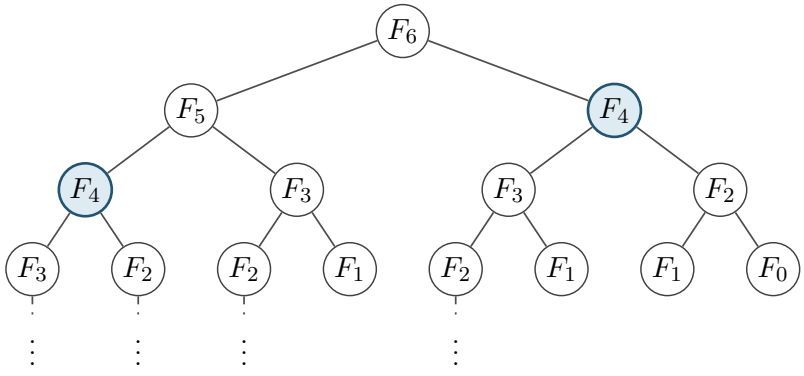


Figure 26.2: Part of the recurrence tree for F_6. The two highlighted occurrences of F_4 represent the same subproblem. Memorization saves its answer after the first computation and reuses it for the second.

Memorization keeps the recursive algorithm but saves each answer when it is first computed. A later call with the same arguments returns the saved answer. A *bottom-up* algorithm instead chooses an order in which every needed answer is already available.

Fibonacci Number

Input: A nonnegative integer n.
Output: F_n.

Top-Down Memorization Algorithm

The table F is shared by all calls to VALUE. The symbol $\perp$ marks an entry whose answer has not yet been computed.

Algorithm FIBONACCITOPDOWN(n)

```
1 initialize F[0..n] to ⊥; F[0] ← 0
2 if n ≥ 1 then F[1] ← 1
3 return VALUE(n)
4 procedure VALUE(i)
5     if F[i] = ⊥ then
6         F[i] ← VALUE(i − 1) + VALUE(i − 2)
7     return F[i]
```

Theorem (Top-down Fibonacci correctness)**.** FIBONACCITOPDOWN *solves the* Fibonacci Number *problem.*

Proof. We claim that every call to VALUE(i) returns F_i. We prove this by induction over i. For $i = 0$ or $i = 1$, the claim follows from the initialization. For $i \geq 2$, the first call computes the two smaller values correctly by the inductive hypothesis and stores their sum, F_i. Every later call returns this same saved value. In particular, the algorithm returns F_n. □

Theorem (Top-down Fibonacci running time)**.** FIBONACCITOPDOWN *performs* $O(n)$ *additions.*

Proof. Each entry $F[i]$ is computed at most once, since subsequent calls return its saved value. Each non-base entry requires one addition of Fibonacci numbers, and there are at most n such entries. □

Bottom-Up Algorithm

We can also fill the same table in the order $0, 1, \ldots, n$, so that the two entries needed for each addition have already been computed.

Algorithm FIBONACCI(n)

```
1 if n ≤ 1 then return n
2 F[0] ← 0; F[1] ← 1
3 for i = 2, ..., n do F[i] ← F[i − 1] + F[i − 2]
4 return F[n]
```

Theorem (Bottom-up Fibonacci correctness)**.** FIBONACCI *solves the* Fibonacci Number *problem.*

Proof. We claim that each computed entry $F[i]$ equals F_i. We prove this by induction over i. The claim holds for the initialized entries $F[0]$ and $F[1]$. For $i \geq 2$, the two smaller entries are already correct by the inductive hypothesis, so their sum is $F_{i-1} + F_{i-2} = F_i$. Thus the returned entry is F_n. When $n \leq 1$, the early return is correct by definition. □

Theorem (Bottom-up Fibonacci running time)**.** FIBONACCI *performs* $O(n)$ *additions.*

Proof. Each loop iteration computes one entry using one addition of Fibonacci numbers. There are at most n iterations, giving $O(n)$ additions. □

For the algorithms that follow, we first define the mathematical answer to each subproblem. The algorithm computes these answers by top-down memorization or in a bottom-up order. Its correctness proof typically uses induction, dividing possible solutions into cases to justify the update for each entry. The dependency relation must be acyclic: a subproblem cannot require its own answer before it is computed.

When the desired output is a solution rather than only its value, we also recover the choices that attain the computed optimum. Running-time proofs count the subproblems, the work per subproblem, and any work needed to recover the output.

Empty sums are zero, and ties between optimal choices may be broken arbitrarily. Tables may include index zero or an extra boundary row when the algorithm creates them explicitly. For the problems that follow, indices and values obey the word-cost convention of Chapter 1 unless stated otherwise.

Chapter 27

Weighted Interval Scheduling

The weight of a collection of intervals is the sum of their weights. As in Chapter 21, intervals that share an endpoint overlap.

Weighted Interval Scheduling

Input: An array of intervals I with nonnegative integer weights w.
Output: The maximum total weight of a pairwise disjoint subcollection of I.

Write $I[i] = [s_i, f_i]$ and $w[i]$ for its weight.

27.1 Main idea

Order the intervals by starting time. An optimal solution either skips the first interval or includes it and continues after its finish. Each possibility leaves a suffix of the same ordered list. Storing the best weight for each suffix lets us reuse these answers.

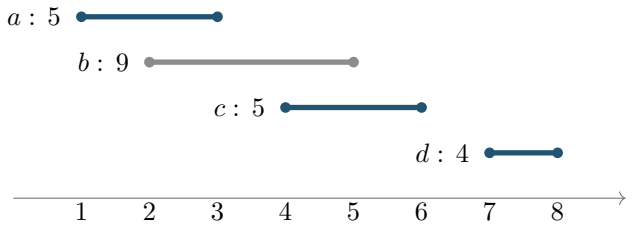


Figure 27.1: Taking a, c, d gives weight 14. Taking the individually heaviest interval b leaves only d, for weight 13.

27.2 Top-Down Memorization Algorithm

For the intervals sorted by starting time, let $\mathrm{OPT}(i)$ be the maximum weight of a pairwise disjoint subcollection of intervals $i, \ldots, n$. We store this value in $opt[i]$.

The subroutine $\textsc{Next}(I, i)$ returns the first index j with $s_j > f_i$, or $n + 1$ if none exists. Binary search finds this index in $O(\log n)$ time. Since $s_i < f_i$, we always have $\textsc{Next}(I, i) > i$.

The recursive procedure shares the sorted input and the table opt with the main call. An entry equal to $\perp$ has not yet been computed.

Algorithm WeightedIntervalsTopDown($I[1..n], w[1..n]$)

```
1 sort intervals and their weights by nondecreasing starting time
2 initialize opt[1..n] to ⊥; opt[n + 1] ← 0
3 return Value(1)
4 procedure Value(i)
5    if opt[i] = ⊥ then
6       opt[i] ← max{Value(i + 1), w[i] + Value(Next(I, i))}
7    return opt[i]
```

Theorem 27.1 (Correctness)**.** WeightedIntervalsTopDown *solves the* Weighted Interval Scheduling *problem.*

Proof. We claim that every call to Value(i) returns OPT(i). We prove this by induction over decreasing i. For $i = n + 1$, the empty suffix has weight zero, as stored in $opt[n + 1]$.

For $i \leq n$, let $j = \text{Next}(I, i)$. Among solutions omitting i, the maximum weight is OPT($i + 1$). A solution including i can use only intervals with index at least j after it. Conversely, any disjoint subcollection of that suffix can be combined with i. The maximum possible weight in this case is therefore $w[i] + \text{OPT}(j)$.

Since $i + 1, j > i$, the inductive hypothesis gives the correct suffix values when $opt[i]$ is first computed. The algorithm stores the larger of the two possible weights, which is OPT(i). Every later call returns this same saved value. In particular, the algorithm returns OPT(1), the maximum weight for the full input. □

Theorem 27.2 (Running time)**.** WeightedIntervalsTopDown *takes* $O(n \log n)$ *time.*

Proof. Sorting takes $O(n \log n)$ time, and initializing opt takes $O(n)$ time. Each of its n non-boundary entries is computed at most once. Computing an entry makes one call to Next, taking $O(\log n)$ time, and two recursive calls. A call for a saved entry takes constant time.

There are at most $2n + 1$ calls to Value in total, including the initial call, and at most n binary searches. Their total work is $O(n \log n)$. Adding the sorting and initialization costs proves the bound. □

27.3 Bottom-Up Algorithm

Both entries needed for $opt[i]$ have larger indices. We can therefore fill the same table from right to left.

Algorithm WeightedIntervals$(I[1..n], w[1..n])$

```
1 sort intervals and their weights by nondecreasing starting time
2 opt[n + 1] ← 0
3 for i = n, n − 1, . . . , 1 do
4     opt[i] ← max{opt[i + 1], w[i] + opt[Next(I, i)]}
5 return opt[1]
```

Theorem 27.3 (Correctness)**.** WeightedIntervals *solves the* Weighted Interval Scheduling *problem.*

Proof. We claim that each computed entry $opt[i]$ equals $\mathrm{OPT}(i)$. We prove this by induction over decreasing i. For $i = n + 1$, the empty suffix has weight zero, as stored in $opt[n + 1]$.

For $i \leq n$, let $j = \textsc{Next}(I, i)$. Among solutions omitting i, the maximum weight is $\mathrm{OPT}(i + 1)$. A solution including i can use only intervals with index at least j after it. Conversely, any disjoint subcollection of that suffix can be combined with i. The maximum possible weight in this case is therefore $w[i] + \mathrm{OPT}(j)$.

Since $i + 1, j > i$, the inductive hypothesis gives the correct suffix values when $opt[i]$ is computed. The algorithm stores the larger of the two possible weights, which is $\mathrm{OPT}(i)$. Thus the algorithm returns $\mathrm{OPT}(1)$, the maximum weight for the full input. □

Theorem 27.4 (Running time)**.** WeightedIntervals *takes* $O(n \log n)$ *time.*

Proof. Sorting takes $O(n \log n)$ time. Each of the n loop iterations makes one call to Next, taking $O(\log n)$ time, and otherwise takes constant time. Filling the table therefore takes $O(n \log n)$ time. Adding the costs proves the bound. □

Chapter 28

Subset Sum

Subset Sum

Input: An array of positive integers A and a nonnegative integer K.
Output: A set of indices whose entries sum to K, or a report that none exists.

28.1 Main idea

For each entry, either omit it and keep the same target, or include it and subtract its value from the target. Store the answer for every suffix and every remaining target between 0 and K.

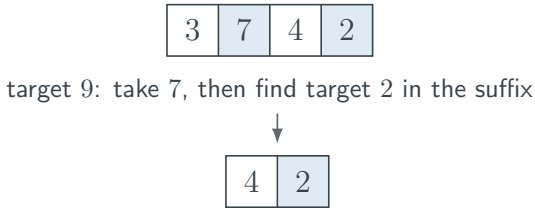


Figure 28.1: The highlighted entries 7 and 2 sum to 9. Their indices, rather than just their values, identify the chosen items.

28.2 Algorithm

Let $S(i, k)$ mean that some subset of indices $i, \ldots, n$ sums to k. The entry $D[i, k]$ records whether this is possible.

The table includes column zero. Short-circuit evaluation prevents accesses to a negative column.

Algorithm SubsetSum$(A[1..n], K)$

```
1  D[n+1, 0] ← true
2  for k = 1, ..., K do D[n+1, k] ← false
3  for i = n, n−1, ..., 1 do
4      for k = 0, ..., K do
5          D[i, k] ← D[i+1, k] or (A[i] ≤ k and D[i+1, k − A[i]])
6  if not D[1, K] then return "no solution"
7  create an empty list J; k ← K
8  for i = 1, ..., n do
9      if not D[i+1, k] then append i to J; k ← k − A[i]
10 return J
```

Theorem 28.1 (Correctness)**.** SUBSETSUM *solves the* Subset Sum *problem.*

Proof. We claim that each computed entry $D[i, k]$ equals $S(i, k)$. We prove this by induction over decreasing i. For $i = n + 1$, the empty suffix supplies only sum zero, as recorded in the last row.

For $i \leq n$, a subset summing to k either omits i, leaving target k in the next suffix, or includes i, leaving target $k - A[i]$. The second case is possible only when $A[i] \leq k$. Conversely, a solution to either remaining problem supplies a subset of sum k, adding i in the second case. By the inductive hypothesis, the next row correctly records both possibilities. Their Boolean disjunction therefore gives $D[i, k] = S(i, k)$.

A false entry $D[1, K]$ justifies the negative report. Otherwise, at the start of each recovery iteration, we maintain $D[i, k] = \text{true}$ and $\sum_{j \in J} A[j] + k = K$. Both hold initially. If skipping i is possible, they persist. If not, taking i must be possible; appending i and subtracting $A[i]$ preserves both properties. At row $n + 1$, a true entry requires $k = 0$, so the distinct selected indices sum to K. □

Theorem 28.2 (Running time)**.** SUBSETSUM *runs in time* $O(n(K + 1))$.

Proof. Initializing the last row costs $O(K + 1)$. There are $n(K + 1)$ other entries, each computed in constant time. Recovery examines at most n indices and appends each at most once. The total is $O(n(K + 1))$. □

Interlude

Pseudopolynomial Time

A bound can be polynomial in the value of an input integer without being polynomial in the number of bits used to write that integer.

A nonnegative integer K needs $\lceil \log(K+1) \rceil$ binary digits, with one digit used for zero. Thus increasing a target from 2^{10} to 2^{20} increases its representation from 11 bits to 21 bits, but increases the number of columns in the subset-sum table by roughly a factor of one thousand.

A running-time bound is *pseudopolynomial* when it is polynomial in the number of input objects and their numerical magnitudes, but need not be polynomial in their binary encoding length.

The $O(n(K+1))$ bound for SubsetSum is pseudopolynomial. Its correctness proof does not change with the size of K; what changes is how the table size compares with the length of the input.

Lemma (Exponential table size)**.** *There are subset-sum inputs of binary encoding length $O(b)$ for which* SubsetSum *computes at least 2^b table entries.*

Proof. Use the one-entry array $A[1] = 1$ and target $K = 2^b$. Their binary descriptions use $O(b)$ bits. The algorithm still fills row 1 in columns $0, \ldots, 2^b$, which gives the claim. □

Knapsack will have the same distinction, with capacity replacing target. For flows, a bound depending on the numerical value of a maximum flow has the same issue. A bound polynomial in the numbers of vertices and edges avoids this dependence on capacity magnitudes.

These comparisons concern the number of charged operations. Under the word-cost convention of Chapter 1, stored values fit in a constant number of words. A bit-cost analysis must additionally charge for arithmetic on the binary representations.

Chapter 29

Knapsack

Each item has a weight and a value and may be chosen at most once. The total weight must fit within the knapsack's capacity.

Knapsack

Input: Arrays of positive integer weights w and nonnegative integer values v, and a nonnegative integer capacity W.
Output: A set of items of total weight at most W and maximum total value.

29.1 Main idea

Keep the remaining capacity as a parameter. Taking an item spends some capacity and adds its value; omitting it leaves the capacity unchanged. In either case the item is removed from consideration.

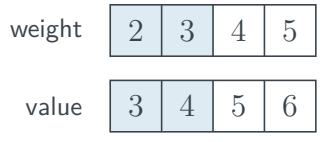


Figure 29.1: With capacity 5, the first two items give value 7. Taking the item of greatest individual value gives only 6.

29.2 Algorithm

Let $K(i, c)$ be the maximum value obtainable from items $i, \dots, n$ with capacity c. We store this value in $D[i, c]$.

Each transition moves to the next row, so no item can be used twice.

Algorithm KNAPSACK$(w[1..n], v[1..n], W)$

```
for c = 0, ..., W do D[n + 1, c] ← 0
for i = n, n − 1, ..., 1 do
    for c = 0, ..., W do
        D[i, c] ← D[i + 1, c]
        if w_i ≤ c then
            D[i, c] ← max{D[i, c], v_i + D[i + 1, c − w_i]}
create an empty list J; c ← W
for i = 1, ..., n do
    if D[i, c] > D[i + 1, c] then append i to J; c ← c − w_i
return J
```

Theorem 29.1 (Correctness)**.** KNAPSACK *solves the* Knapsack *problem.*

Proof. We claim that each computed entry $D[i, c]$ equals $K(i, c)$. We prove this by induction over decreasing i. For $i = n + 1$, the empty suffix has value zero, as stored in the last row.

For $i \leq n$, among solutions omitting i, the maximum value is $K(i + 1, c)$. A solution including i is possible only when $w_i \leq c$. It leaves capacity $c - w_i$ for the next suffix, whose best solution can conversely be combined with i. The maximum value in this case is therefore $v_i + K(i + 1, c - w_i)$. By the inductive hypothesis, the next row supplies the correct values for both cases. The algorithm stores the larger feasible value, which is $K(i, c)$.

At the start of each recovery iteration, the chosen value plus $D[i, c]$ equals $D[1, W]$, and the chosen weight plus c equals W. These equalities hold initially. A strict improvement over skipping means that taking i is feasible and attains $D[i, c]$, so the update preserves both equalities. Otherwise skipping attains the same value. At the last row the remaining value is zero and the remaining capacity is nonnegative. Thus J attains the optimum without exceeding capacity, and each item is considered only once. □

Theorem 29.2 (Running time)**.** KNAPSACK *runs in time* $O(n(W + 1))$.

Proof. The table has $(n + 1)(W + 1)$ entries, each initialized or computed in constant time. Recovery takes $O(n)$ time. Adding these costs proves the bound. □

Chapter 30

Longest Increasing Subsequence

A *subsequence* is obtained by deleting entries while preserving the order of those that remain; its entries need not be consecutive. It is *increasing* if each entry is strictly smaller than the next.

Longest Increasing Subsequence

Input: An integer array A.
Output: Indices $i_1 < \cdots < i_k$ of maximum length with $A[i_1] < \cdots < A[i_k]$.

30.1 Main idea

If a subsequence starts at i, its next index must be a later index with a larger value. The best continuation is the longest subsequence starting at one of those indices.



one longest subsequence: $2, 3, 6, 7$

Figure 30.1: The entries need not be consecutive. The alternative $2, 3, 4, 7$ is also optimal; ties need not be broken in a particular way.

30.2 Algorithm

Let $L(i)$ be the length of a longest increasing subsequence starting at index i. We store this length in $D[i]$.

A next pointer records the continuation responsible for an improvement.

Algorithm IncreasingSubsequence($A[1..n]$)

```
for i = n, n − 1, . . . , 1 do
    D[i] ← 1; next[i] ← ⊥
    for j = i + 1, . . . , n do
        if A[j] > A[i] and 1 + D[j] > D[i] then
            D[i] ← 1 + D[j]; next[i] ← j
choose i maximizing D[i]; create an empty list J
while i ≠ ⊥ do append i to J; i ← next[i]
return J
```

Theorem 30.1 (Correctness)**.** IncreasingSubsequence *solves the* Longest Increasing Subsequence *problem.*

Proof. We claim that each computed entry $D[i]$ equals $L(i)$ and that its next pointers describe an increasing subsequence of this length starting at i. We prove this by induction over decreasing i. For $i = n$, the one-entry subsequence is optimal, and the algorithm sets $D[n] = 1$ with no next pointer.

For $i < n$, an increasing subsequence starting at i either consists of i alone or continues at some $j > i$ with $A[j] > A[i]$. For a fixed such j, its maximum length is $1 + L(j)$: any continuation has length at most $L(j)$, and an optimal one can be prefixed by i. By the inductive hypothesis, $D[j] = L(j)$ and its pointers give such a continuation. The inner loop compares all these possibilities with the one-entry subsequence and records one attaining the maximum. This proves the claim for i.

Every nonempty increasing subsequence has a first index. Choosing an index maximizing $D[i]$ and following its pointers therefore returns a longest increasing subsequence. □

Theorem 30.2 (Running time)**.** IncreasingSubsequence *runs in time* $O(n^2)$*.*

Proof. The inner loop examines one pair for each $i < j$, at constant cost per pair. There are $n(n-1)/2$ pairs. Initialization, choosing the largest entry, and recovery take $O(n)$ additional time. The total is $O(n^2)$. □

Chapter 31

Longest Increasing Subsequence (revisited)

Longest Increasing Subsequence (restated)
Input: An integer array A.
Output: Indices $i_1 < \cdots < i_k$ of maximum length with $A[i_1] < \cdots < A[i_k]$.

31.1 Main idea

Process the array from right to left. Among increasing subsequences of the same length, the one with the largest first value is easiest to extend on the left. Keep one such representative for every achievable length, and use their ordered first values to find the best continuation.

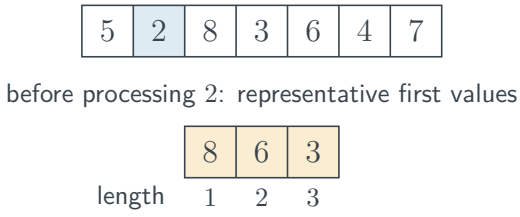


Figure 31.1: For the suffix $8, 3, 6, 4, 7$, representatives can be 8, $6, 7$, and $3, 6, 7$. Prefixing 2 to the last one creates length 4.

31.2 Lemmas

For the processed suffix let b_k be the largest possible first value of an increasing subsequence of length k, and let h be its maximum length. There are no representatives when the suffix is empty.

Lemma 31.1 (Ordered representatives)**.** *The values satisfy $b_1 > b_2 > \cdots > b_h$. For a new entry x on the left, let k be the first index with $b_k \le x$, or $h + 1$ if none exists. The longest subsequence starting at x has length k, and only b_k needs updating, to x.*

Proof. The second entry of a subsequence attaining b_{j+1} starts a length-j subsequence and is larger than b_{j+1}. Thus $b_j > b_{j+1}$.

If $k > 1$, prefixing x to the representative starting at $b_{k-1} > x$ gives length k; if $k = 1$, use x alone. A longer subsequence would need a length-k continuation starting above x. This contradicts $b_k \le x$ when $k \le h$, or the maximality of h otherwise.

For $j < k$, the old value $b_j > x$ is better. At length k, x is at least the old value, if one existed. No longer new subsequence exists, so only b_k needs updating. □

31.3 Algorithm

Store each representative by its first index $p[j]$. A next pointer is fixed when an index is processed; replacing a representative later does not change any previously recorded chain.

Algorithm FastIncreasingSubsequence($A[1..n]$)

```
 1 h ← 0
 2 for i = n, n − 1, . . . , 1 do
 3    ℓ ← 1; r ← h + 1
 4    while ℓ < r do
 5       j ← ⌊(ℓ + r)/2⌋
 6       if A[p[j]] > A[i] then ℓ ← j + 1 else r ← j
 7    k ← ℓ; next[i] ← ⊥
 8    if k > 1 then next[i] ← p[k − 1]
 9    p[k] ← i; h ← max{h, k}
10 create an empty list J; i ← p[h]
11 while i ≠ ⊥ do append i to J; i ← next[i]
12 return J
```

Theorem 31.2 (Correctness)**.** FastIncreasingSubsequence *solves the* Longest Increasing Subsequence (restated) *problem.*

Proof. We claim that after processing a suffix, h is its maximum increasing subsequence length and, for every $j \le h$, the chain starting at $p[j]$ has length j and the largest possible first value. We prove this by induction over the length of the processed suffix. The claim holds for the empty suffix, where $h = 0$.

For the next entry $A[i]$, Lemma 31.1 shows that the test in line 6 changes from true to false at the required index k. The binary search keeps k in $[\ell, r]$, discarding the left half after a true test and the right half otherwise, until it returns k. If $k > 1$, the inductive hypothesis supplies a valid length-$(k-1)$ continuation at $p[k-1]$; otherwise i stands alone. Lines 8–9 record this chain and perform the lemma's update. Previously recorded chains stay unchanged, so the claim is preserved.

After the whole array is processed, the chain starting at $p[h]$ uses increasing indices and values and has optimal length h. The algorithm returns exactly this chain. □

Theorem 31.3 (Running time)**.** FastIncreasingSubsequence *runs in time* $O(n \log n)$.

Proof. Each search starts with at most $n + 1$ candidates and halves their number up to rounding at each iteration, giving $O(\log n)$ time. The other work per input entry is constant. Recovery takes $O(n)$ time, so the total is $O(n \log n)$. □

Chapter 32

Longest Common Subsequence

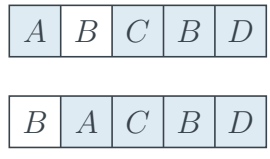
A *string* is an array of characters. A common subsequence of two strings is a string obtained from each by deleting characters without changing the order of those that remain.

Longest Common Subsequence
Input: Strings A and B.
Output: A longest string that is a subsequence of both inputs.

As with arrays, let n and m denote the string lengths. Input strings are nonempty, but suffixes may be empty. Comparing two characters and appending a character to an output list take constant time.

32.1 Main idea

At the front of two suffixes, either match their first characters or skip a character in one suffix. Matching is possible only when those characters agree. Compare the best result in each feasible case.



a longest common subsequence is A, C, B, D

Figure 32.1: Matching positions need not have the same indices. The highlighted characters occur in the same order in both strings.

32.2 Algorithm

Let $C(i, j)$ be the longest common subsequence length of $A[i..n]$ and $B[j..m]$. We store this length in $D[i, j]$. To compute an interior entry, compare $D[i + 1, j]$ and $D[i, j + 1]$, and also $1 + D[i + 1, j + 1]$ when $A[i] = B[j]$.

A recorded choice specifies which case attains an entry.

Algorithm CommonSubsequence($A[1..n], B[1..m]$)

```
set D[n+1, j] ← 0 and D[i, m+1] ← 0 for all boundary indices
for i = n, n−1, ..., 1 do
    for j = m, m−1, ..., 1 do
        choose the largest of the feasible candidates described above
        store its value in D[i, j] and its case in T[i, j]
i ← 1; j ← 1; create an empty string Z
while i ≤ n and j ≤ m do
    if T[i, j] = match then append A[i] to Z; i ← i + 1; j ← j + 1
    else if T[i, j] = skip A then i ← i + 1
    else j ← j + 1
return Z
```

Theorem 32.1 (Correctness)**.** CommonSubsequence *solves the* Longest Common Subsequence *problem.*

Proof. We claim that each computed entry $D[i, j]$ equals $C(i, j)$. We prove this by induction over the total length of the two suffixes. If either suffix is empty, only the empty common subsequence is possible, as recorded by the boundary value zero.

Otherwise, represent a common subsequence by its increasing pairs of matched positions. If its first pair is (i, j), the characters agree and its remaining length is at most $C(i+1, j+1)$. If not, it omits position i or position j: matching both in different pairs would put those pairs in opposite orders. Its length is therefore bounded by one of the two skip cases. Conversely, each skip case supplies a feasible common subsequence, and when $A[i] = B[j]$ we can prefix that character to an optimal subsequence of the two shorter suffixes. Every needed entry has smaller total suffix length and is already correct by the inductive hypothesis. The algorithm takes the largest feasible value, so $D[i, j] = C(i, j)$.

At the start of each recovery iteration, Z is common to the consumed prefixes and $|Z| + D[i, j] = D[1, 1]$. Both properties hold initially. A match appends equal characters and reduces the remaining value by one; a skip preserves that value. Thus both properties persist. At an empty suffix, the remaining value is zero, so Z is a common subsequence of optimal length. □

Theorem 32.2 (Running time)**.** CommonSubsequence *runs in time* $O(nm)$.

Proof. Boundary initialization costs $O(n + m)$. Each of the nm interior entries compares at most three constant-time candidates. Recovery has at most $n + m$ steps and uses constant-time appends. Since input lengths are positive, their sum is $O(nm)$, giving the bound. □

Chapter 33

Edit Distance

An *edit* inserts, deletes, or substitutes one character. The *edit distance* between two strings is the smallest number of edits needed to transform one into the other.

An *alignment* places the strings in two rows, inserting gaps so that the rows have equal length and no column has two gaps. Matching characters cost zero; each other column costs one.

Edit Distance
Input: Strings A and B.
Output: Their edit distance and an alignment attaining it.

33.1 Main idea

Align the strings from left to right. A column either pairs two characters, deletes an input character, or inserts an output character. After choosing its first column, the remaining alignment concerns two shorter suffixes.

C	A	R	T
C	U	$-$	T

Figure 33.1: An alignment of `CART` and `CUT` of cost 2: substitute A by U and delete R. A dash denotes a gap.

33.2 Lemmas

Lemma 33.1 (Alignment interpretation)**.** *The minimum alignment cost equals the minimum number of edits.*

Proof. An alignment gives an edit sequence of the same cost: keep matches, substitute mismatches, delete characters opposite gaps, and insert characters below gaps.

Conversely, track original characters through an edit sequence. Surviving originals keep their relative order and pair with their final characters. Place deleted originals and surviving inserted characters in gap columns between these pairs. Each deleted original required a deletion, each surviving inserted character an insertion, and each changed surviving original a substitution. These are distinct operations, so this alignment costs no more than the edit sequence. Taking minima proves equality. □

33.3 Algorithm

Let $E(i, j)$ be the minimum alignment cost of $A[i..n]$ and $B[j..m]$. We store this cost in $D[i, j]$. An interior entry compares $1 + D[i + 1, j]$ for deletion, $1 + D[i, j + 1]$ for insertion, and $[A[i] \neq B[j]] + D[i + 1, j + 1]$ for pairing. Here $[P]$ is 1 when P is true and 0 otherwise. Record a minimizing case, breaking ties arbitrarily. All recovery appends are to Z.

Algorithm EditDistance($A[1..n], B[1..m]$)

```
initialize D[n + 1, j] ← m − j + 1 and D[i, m + 1] ← n − i + 1
for i = n, n − 1, . . . , 1 do
    for j = m, m − 1, . . . , 1 do
        store the least of these three candidates in D[i, j] and its case in T[i, j]
i ← 1; j ← 1; create an empty column list Z
while i ≤ n and j ≤ m do
    if T[i, j] = delete then append (A[i], –); i ← i + 1
    else if T[i, j] = insert then append (–, B[j]); j ← j + 1
    else append (A[i], B[j]); i ← i + 1; j ← j + 1
append remaining characters, each opposite a gap, to Z
return (D[1, 1], Z)
```

Theorem 33.2 (Correctness)**.** EditDistance *solves the* Edit Distance *problem.*

Proof. We claim that each computed entry $D[i, j]$ equals $E(i, j)$. We prove this by induction over the total length of the two suffixes. If one suffix is empty, every remaining character must occupy its own gap column. The initialized boundary value is therefore correct.

Otherwise the first alignment column deletes $A[i]$, inserts $B[j]$, or pairs the two characters. Its cost is respectively 1, 1, or $[A[i] \neq B[j]]$. Each choice leaves suffixes of smaller total length, whose optimal costs are already correct by the inductive hypothesis. Every alignment has one of these first columns, and each can be followed by an optimal remaining alignment. Taking the minimum of these three costs therefore gives $D[i, j] = E(i, j)$.

Recovery consumes both strings in order, so its columns form a valid alignment. At every step, the cost already produced plus $D[i, j]$ equals $D[1, 1]$: this holds initially and is preserved by the recorded minimizing choice. The final gap columns cost the remaining boundary value. Thus the returned alignment has cost $D[1, 1] = E(1, 1)$, which equals the edit distance by Lemma 33.1. □

Theorem 33.3 (Running time)**.** EditDistance *runs in time* $O(nm)$.

Proof. Boundary initialization takes $O(n + m)$ time. Each of the nm interior entries compares three constant-time candidates. Recovery consumes a character on every step and takes $O(n + m)$ time. Adding these costs gives $O(nm)$ for positive input lengths. □

Chapter 34

Maximum Independent Set on Trees

An *independent set* contains no two adjacent vertices. Here we seek a largest independent set when the input graph is a tree.

Maximum Independent Set on a Tree

Input: A tree T.
Output: A largest set of pairwise nonadjacent vertices.

34.1 Main idea

Root the tree. Omitting a vertex leaves its child subtrees independent. Taking it excludes its children and leaves its grandchild subtrees independent. Compare these two possibilities at each vertex.

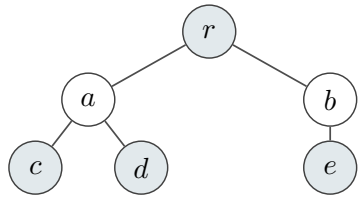


Figure 34.1: Taking r permits c, d, e, giving four vertices. Omitting r gives at most two from the subtree at a and one from the subtree at b.

34.2 Algorithm

Let $I(v)$ be the maximum independent-set size in the subtree at v. Write $\mathrm{ch}(v)$ and $\mathrm{gc}(v)$ for its children and grandchildren. We store $I(v)$ in $D[v]$ and record whether an optimal solution takes v.

Run DFSTree from any root r. Its parent array gives the children lists. Process vertices in increasing finish time, so every child is processed before its parent. The recovery procedure shares the output list J.

Algorithm TreeIndependentSet(T)

```
1 run DFSTree(T, r); obtain children lists and increasing finish order
2 for each vertex v in increasing finish order do
3    a ← Σ_{u∈ch(v)} D[u]; b ← 1 + Σ_{z∈gc(v)} D[z]
4    D[v] ← max{a, b}; take[v] ← (b ≥ a)
5 create an empty list J; Recover(r); return J
6 procedure Recover(v)
7    if take[v] then
8       append v to J; call Recover(z) for each z ∈ gc(v)
9    else call Recover(u) for each u ∈ ch(v)
```

Theorem 34.1 (Correctness)**.** TreeIndependentSet *solves the* Maximum Independent Set on a Tree *problem.*

Proof. We claim that $D[v] = I(v)$ and that the vertices added by Recover(v) form an independent set of size $I(v)$ in the subtree at v. We prove this by induction over subtree size. For a leaf, the algorithm sets $D[v] = 1$ and recovery takes v, which is optimal.

For an internal vertex v, all child and grandchild subtrees are smaller and have already been processed. Different child subtrees have no edge between them, since such an edge would create a cycle. The same is true of the grandchild subtrees.

Among independent sets omitting v, the maximum size is $\sum_{u\in\mathrm{ch}(v)} I(u)$: each child subtree contributes at most its optimum, and their optimal sets can be combined. Among sets including v, all children are excluded. The remaining vertices lie in the grandchild subtrees, whose optimal sets can be combined with v. The maximum size in this case is therefore $1 + \sum_{z\in\mathrm{gc}(v)} I(z)$.

By the inductive hypothesis, the algorithm computes both sizes correctly and recovery supplies the optimal sets needed in each case. It stores the larger size and recovers a set attaining it, proving the claim for v. At the root, this gives a maximum independent set of the whole tree. □

Theorem 34.2 (Running time)**.** TreeIndependentSet *runs in time* $O(n)$*.*

Proof. A rooted tree has one parent edge per nonroot vertex, hence $n - 1$ edges, so DFSTree takes $O(n)$ time. Children lists are built by one scan of the parent array. Place each vertex v at index finish[v] in an array of length $2n$, then scan the array to obtain the required order in $O(n)$ time. In all child sums together, each nonroot vertex occurs once. In all grandchild sums together, each vertex occurs at most once, under its unique grandparent. Enumerating grandchildren by children lists also examines each child-list header once per parent. Thus the table work is $O(n)$. Recovery calls act on disjoint descendant subtrees. No vertex is a call argument twice, and the same child and grandchild counts bound all scans by $O(n)$. The total is $O(n)$. □

Chapter 35

Optimal Binary Search Trees

A *binary search tree* stores smaller keys in each left subtree and larger keys in each right subtree. Its root has depth zero, so a successful search for a key at depth d makes $d + 1$ comparisons.

Optimal Binary Search Tree

Input: An increasing array of keys x and positive integer access frequencies f.
Output: A binary search tree minimizing $\sum_i f_i(1 + \text{depth}(x_i))$.

35.1 Main idea

Choose a root, solve the intervals on its two sides, and attach the two optimal subtrees. Attaching adds one comparison to every search in a subtree, so the recurrence includes the sum of all frequencies in the current interval.

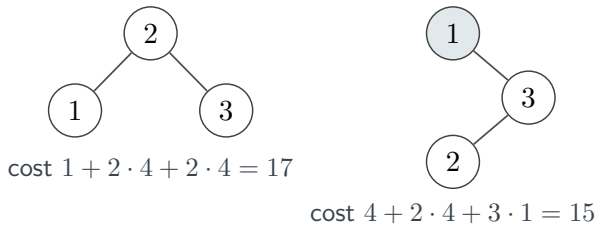


Figure 35.1: For keys $1, 2, 3$ with frequencies $4, 1, 4$, a balanced tree need not minimize the total search cost.

35.2 Algorithm

Let $C(i, j)$ be the optimal cost on keys $x_i, \ldots, x_j$, with depths measured from that subtree's root. We store this cost in $D[i, j]$. Put $F(i, j) = \sum_{k=i}^{j} f_k$.

Prefix sums $P[0] = 0$ and $P[j] = \sum_{k=1}^{j} f_k$ give $F(i, j) = P[j] - P[i - 1]$ in constant time. Empty-interval entries are initialized before any nonempty interval is processed.

Algorithm OptimalSearchTree($x[1..n], f[1..n]$)

```
1  compute P[0..n]; set D[i, i − 1] ← 0 for i = 1, . . . , n + 1
2  for ℓ = 1, . . . , n do
3     for i = 1, . . . , n − ℓ + 1 do
4        j ← i + ℓ − 1
5        choose r ∈ {i, . . . , j} minimizing D[i, r − 1] + D[r + 1, j]
6        R[i, j] ← r; D[i, j] ← P[j] − P[i − 1] + D[i, r − 1] + D[r + 1, j]
7  return Build(1, n)
8  procedure Build(i, j)
9     if i > j then return an empty tree
10    r ← R[i, j]
11    return a root x_r with children Build(i, r − 1) and Build(r + 1, j)
```

Theorem 35.1 (Correctness)**.** OptimalSearchTree *solves the* Optimal Binary Search Tree *problem.*

Proof. We claim that $D[i, j] = C(i, j)$ and that Build(i, j) returns a binary search tree on exactly $x_i, \ldots, x_j$ attaining this cost. We prove this by induction over interval length. An empty interval has cost zero, and Build returns the empty tree.

For a nonempty interval, a tree rooted at x_r must place keys $x_i, \ldots, x_{r-1}$ on the left and $x_{r+1}, \ldots, x_j$ on the right. Their internal costs are at least $C(i, r-1)$ and $C(r+1, j)$. Every search also compares its key with the root, adding $F(i, j)$. Conversely, attaching optimal trees for these two intervals attains the resulting cost.

Both intervals are shorter, so the inductive hypothesis supplies their correct costs and trees. The algorithm tries every root and stores the minimum cost, giving $D[i, j] = C(i, j)$. Build attaches the two optimal subtrees to the recorded root, so its tree attains this value. For $(i, j) = (1, n)$, it is therefore an optimal binary search tree for the full input. □

Theorem 35.2 (Running time)**.** OptimalSearchTree *runs in time* $O(n^3)$*.*

Proof. Prefix sums and empty entries take $O(n)$ time. There are $O(n^2)$ nonempty intervals, each trying at most n roots with constant work per root. Recovery creates one node per key and at most two child calls per node, taking $O(n)$ time. The total is $O(n^3)$. □

Chapter 36

Shortest Paths with Negative Weights

A *negative cycle* is a directed cycle whose total edge weight is negative.

Shortest Paths without Negative Cycles

Input: A directed weighted graph G with no negative-weight cycle, and a source s.
Output: The weighted distance from s to every vertex, using $+\infty$ if unreachable.

36.1 Main idea

Repeatedly scan the edges to improve distance estimates. After one scan, every one-edge route has been considered; after two scans, every two-edge route has been considered, and so on. Without negative cycles, a shortest route never needs more than $n - 1$ edges.

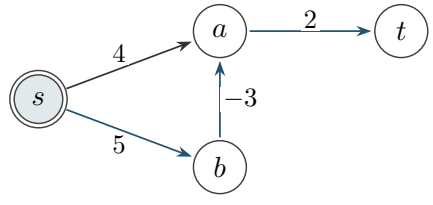


Figure 36.1: The blue path to t has weight 4. Scanning edges in the order $(a, t), (b, a), (s, b), (s, a)$ gives $d[t] = +\infty, 6, 4$ after the first, second, and third scans.

36.2 Lemmas

Write $\delta(v)$ for the minimum path weight from s to v, and let $D_k(v)$ be the minimum weight of a walk from s to v using at most k edges. A missing walk has value $+\infty$, and $+\infty + a = +\infty$ for finite a.

Lemma 36.1 (Removing cycles)**.** *In a graph with no negative cycle, every walk can be shortened to a path with the same endpoints and no greater weight. In particular,* $D_{n-1}(v) = \delta(v)$.

Proof. If a walk repeats a vertex, choose a repeated segment with no internal repetition. This is a directed cycle, so its weight is nonnegative. Deleting it preserves the endpoints and does not increase weight. Each deletion removes edges, so repeating this step ends at a path, which has at most $n - 1$ edges. The two minima therefore agree. □

Lemma 36.2 (Bounded-edge recurrence)**.** *$D_0(s) = 0$ and $D_0(v) = +\infty$ for $v \neq s$. For $k \geq 1$,*

$$D_k(v) = \min\big(\{D_{k-1}(v)\} \cup \{D_{k-1}(u) + w(u,v) : (u,v) \in E\}\big).$$

Proof. A walk with at most k edges either uses at most $k-1$ edges or ends with an edge (u,v) after such a walk. Each case has at least the corresponding candidate weight, and every finite candidate is attained. Taking the minimum proves the recurrence. □

36.3 Algorithm

Bellman–Ford uses one distance array, updating it immediately during a scan. These updates can propagate farther than one edge in a scan; the proof therefore uses inequalities, not equality with D_k.

Algorithm BellmanFord(G, w, s)

```
1 set d[v] ← +∞ for every v; d[s] ← 0
2 for k = 1, ..., n − 1 do
3     for each edge (u, v) do
4         d[v] ← min{d[v], d[u] + w(u, v)}
5 return d
```

Theorem 36.3 (Correctness)**.** BellmanFord *solves the* Shortest Paths without Negative Cycles *problem.*

Proof. We claim that after k scans, every finite label $d[v]$ is the weight of a walk from s to v, and $d[v] \leq D_k(v)$ for every vertex v. We prove this by induction over k. For $k = 0$, initialization gives $d[v] = D_0(v)$; the only finite label is zero at s, attained by the zero-edge walk.

During scan k, each new finite label extends a witnessing walk by one edge, preserving the first claim. Labels only decrease, so the inductive hypothesis gives $d[u] \leq D_{k-1}(u)$ when (u,v) is scanned. Its update ensures $d[v] \leq D_{k-1}(u) + w(u,v)$. The final $d[v]$ is therefore at most every incoming-edge candidate in Lemma 36.2, and remains at most $D_{k-1}(v)$. The lemma gives $d[v] \leq D_k(v)$, proving the second claim.

The witnessing-walk property and Lemma 36.1 give $d[v] \geq \delta(v)$. Thus after $n-1$ scans,

$$\delta(v) \leq d[v] \leq D_{n-1}(v) = \delta(v).$$ □

Theorem 36.4 (Running time)**.** BellmanFord *runs in time $O(n + nm)$.*

Proof. Initialization and forming an edge list take $O(n+m)$ time. Each of the $n-1$ scans takes $O(m+1)$ time, including loop overhead. The total is $O(n + nm)$, including graphs with no edges. □

Chapter 37

Negative Cycles

Recall that a negative cycle is a directed cycle whose total edge weight is negative.

Negative Cycle
Input: A directed weighted graph G.
Output: A negative-weight directed cycle, or a report that none exists.

37.1 Main idea

Allow a walk to start at any vertex, so every component is covered. Compare the best walks using at most $n-1$ edges with those using at most n. An optimal walk giving a strict improvement must repeat a vertex; the first repetition reveals a negative cycle.

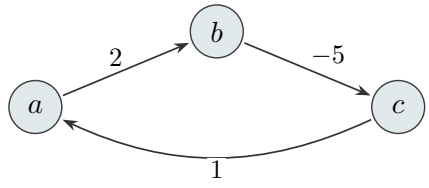


Figure 37.1: The cycle has weight -2. Traversing it again makes a walk cheaper; restricting the number of edges is what keeps each subproblem finite.

37.2 Lemmas

Let $H_k(v)$ be the minimum weight of a walk ending at v, starting anywhere, and using at most k edges. Thus $H_0(v) = 0$ for every vertex. Its recurrence is the bounded-edge recurrence of Lemma 36.2, with this different base row.

Lemma 37.1 (Detection and recovery)**.** *A negative cycle exists if and only if $H_n(v) < H_{n-1}(v)$ for some v. For any such v, an optimal walk attaining $H_n(v)$ uses exactly n edges, and every contiguous segment of that walk that is a simple cycle has negative weight.*

Proof. If no negative cycle exists, deleting repeated portions produces a path with at most $n-1$ edges and no greater weight. Hence the two rows are equal.

Conversely, suppose the rows are equal and put $h(v) = H_{n-1}(v)$. The recurrence gives $h(v) \le h(u) + w(u, v)$ for every edge. Summing these inequalities around any directed cycle cancels all h terms, giving total cycle weight at least zero. Thus no negative cycle exists, proving the equivalence.

Now let $H_n(v) < H_{n-1}(v)$. An attaining walk cannot have fewer than n edges, or it would be eligible for $H_{n-1}(v)$. Its $n+1$ vertices contain a repetition. If a repeated segment forming a simple cycle had nonnegative weight, deleting it would give a walk with at most $n-1$ edges and weight at most $H_n(v)$, a contradiction. Thus that cycle is negative. □

37.3 Algorithm

A recorded choice is either *stay* or an incoming edge. To trace a walk from (k, v), a stay moves to $(k-1, v)$; an edge (u, v) moves to $(k-1, u)$ and prepends that edge to the walk. For an improving entry in row n, the lemma guarantees that the resulting walk has n edges.

Algorithm NegativeCycle(G, w)

```
set D[0, v] ← 0 for every v
for k = 1, ..., n do
    set D[k, v] ← D[k − 1, v] and T[k, v] ← stay for every v
    for each edge (u, v) do
        if D[k − 1, u] + w(u, v) < D[k, v] then
            D[k, v] ← D[k − 1, u] + w(u, v); T[k, v] ← (u, v)
if D[n, v] = D[n − 1, v] for every v then return "no negative cycle"
choose v with D[n, v] < D[n − 1, v]; trace its recorded walk P
scan P from its start until the first repeated vertex
return the segment between its two occurrences
```

Theorem 37.2 (Correctness)**.** NegativeCycle *solves the* Negative Cycle *problem.*

Proof. We claim that $D[k, v] = H_k(v)$ and that the recorded choices describe a walk attaining this value. We prove this by induction over k. For $k = 0$, the zero-edge walk at each vertex has weight zero, as recorded in the base row.

For $k \geq 1$, a walk either uses at most $k-1$ edges or has a last edge (u, v) preceded by a walk using at most $k-1$ edges. By the inductive hypothesis, the preceding row gives the optimum for each of these possibilities, together with an attaining walk. Taking the minimum of the stay candidate and all incoming-edge candidates therefore gives $H_k(v)$, and recording its choice preserves an attaining walk.

Lemma 37.1 now justifies reporting that no negative cycle exists exactly when the last two rows agree. Otherwise the traced walk has n edges, so its $n+1$ vertices include a repetition. The first repetition closes a cycle with no repeated interior vertex. By the lemma this cycle is negative, so the returned cycle is correct. □

Theorem 37.3 (Running time)**.** NegativeCycle *runs in time* $O(n(n+m))$.

Proof. Each of n rows copies n entries and scans m edges, for $O(n+m)$ time per row. Tracing takes $O(n)$ time. An array recording each vertex's first position in the walk finds the first repetition in $O(n)$ time, including initialization; extracting its segment also takes $O(n)$. These costs give $O(n(n+m))$ in total. □

Chapter 38

All-Pairs Shortest Paths

All-Pairs Shortest Paths

Input: A directed weighted graph G with no negative-weight cycle.
Output: The weighted distance between every ordered pair of vertices.

38.1 Main idea

Allow intermediate vertices one at a time. When vertex k becomes available, a shortest route either avoids it or combines a route to k with a route from k.

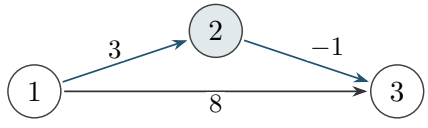


Figure 38.1: Allowing vertex 2 as an intermediate improves the distance from 1 to 3 from 8 to $3 + (-1) = 2$.

38.2 Algorithm

Let $F_k(i, j)$ be the minimum path weight from i to j with all intermediate vertices in $\{1, \ldots, k\}$. The zero-edge path has weight zero; missing paths have value $+\infty$. We store this value in $D[k, i, j]$.

Keep the preceding layer intact while computing the next one.

Algorithm FloydWarshall(G, w)

```
set D[0,i,j] ← +∞ for all i,j and D[0,i,i] ← 0 for all i
for each edge (i,j) do D[0,i,j] ← min{D[0,i,j], w(i,j)}
for k = 1,...,n do
    for i = 1,...,n do
        for j = 1,...,n do
            D[k,i,j] ← min{D[k−1,i,j], D[k−1,i,k] + D[k−1,k,j]}
return D[n,·,·]
```

Theorem 38.1 (Correctness)**.** FloydWarshall *solves the* All-Pairs Shortest Paths *problem.*

Proof. We claim that each computed entry $D[k, i, j]$ equals $F_k(i, j)$. We prove this by induction over k. For $k = 0$, a path has no intermediate vertices, so it is either a single edge or a zero-edge path. Initialization records exactly these values; a nonnegative self-loop does not change the zero diagonal.

For $k \geq 1$, a shortest permitted path either avoids k internally or visits k once and splits into two paths whose intermediate vertices are less than k. If k is an endpoint, the first case suffices. Its weight is therefore at least

$$\min\{F_{k-1}(i, j),\ F_{k-1}(i, k) + F_{k-1}(k, j)\}.$$

Conversely, a finite first candidate is attained by a permitted path. For a finite second candidate, concatenate attaining paths to and from k. Deleting repetitions cannot increase the resulting walk's weight by Lemma 36.1, and leaves a permitted path. Thus the displayed minimum equals $F_k(i, j)$, including when no permitted path exists.

By the inductive hypothesis, the preceding layer supplies the correct values in this minimum. The update therefore gives $D[k, i, j] = F_k(i, j)$. In the final layer all vertices are permitted internally, so the returned entries are precisely the weighted distances. □

Theorem 38.2 (Running time)**.** FloydWarshall *runs in time* $O(n^3)$*.*

Proof. Initialization takes $O(n^2 + m) = O(n^2)$ time for a simple directed graph. The three nested loops compute n^3 entries, each in constant time. Returning the final layer by reference takes constant time. The total is $O(n^3)$. □

Chapter 39

Travelling Salesperson

A *complete graph* has an edge between every pair of distinct vertices. A *tour* is a cycle that visits every vertex exactly once.

Travelling Salesperson

Input: A complete graph on at least three vertices with nonnegative edge weights w.

Output: A minimum-weight cycle visiting every vertex exactly once.

39.1 Main idea

Fix vertex 1 as the start. A partial tour is summarized by its last vertex and the set of other vertices visited. Try each possible penultimate vertex, then close the completed path back to 1.

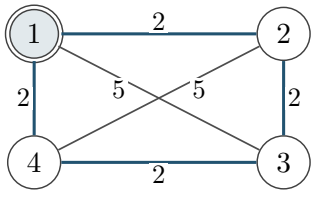


Figure 39.1: For visited set $\{2, 3, 4\}$ and last vertex 4, the best path is $1, 2, 3, 4$ of weight 6. Closing it gives a tour of weight 8.

39.2 Algorithm

For a nonempty $S \subseteq \{2, \ldots, n\}$ and $v \in S$, let $C(S, v)$ be the minimum weight of a path from 1 to v visiting exactly $\{1\} \cup S$. We store this weight in $D[S, v]$ and an attaining penultimate vertex in $P[S, v]$.

Use subset masks as table indices. Membership and deletion of a vertex are word operations under the cost model for this exponentially sized table: its addresses already require $O(n)$ bits. Store the weights in a matrix, built in $O(n^2)$ time if needed.

Algorithm TravellingSalesperson(G, w)

```
1  U ← {2, ..., n}
2  for each v ∈ U do D[{v}, v] ← w(1, v); P[{v}, v] ← ⊥
3  for k = 2, ..., n − 1 do
4     for each S ⊆ U with |S| = k, and each v ∈ S, do
5        choose u ∈ S \ {v} minimizing D[S \ {v}, u] + w(u, v)
6        store the minimum in D[S, v] and u in P[S, v]
7  choose v ∈ U minimizing D[U, v] + w(v, 1); S ← U
8  create an empty list J
9  while S ≠ ∅ do
10    prepend v to J; u ← P[S, v]; S ← S \ {v}; v ← u
11 return the cycle (1, J, 1)
```

Theorem 39.1 (Correctness)**.** TravellingSalesperson *solves the* Travelling Salesperson *problem.*

Proof. We claim that $D[S, v] = C(S, v)$ and that its recorded predecessors describe a path attaining this value. We prove this by induction over $|S|$. For $S = \{v\}$, the only possible path is the edge from 1 to v, whose weight is stored in the table.

For $|S| > 1$, every eligible path has a penultimate vertex $u \in S \setminus \{v\}$. Removing its last edge leaves a path counted by $C(S \setminus \{v\}, u)$, so its weight is at least $C(S \setminus \{v\}, u) + w(u, v)$. Conversely, appending (u, v) to an optimal such path is valid because v is not yet visited. The inductive hypothesis supplies the correct cost and an attaining path for each smaller set. Trying every possible u and recording a minimum therefore proves the claim for (S, v).

Every tour starting at 1 consists of a path visiting all vertices followed by its closing edge to 1, and every such closure is a tour. Line 7 chooses the least possible total weight. Recovery follows the recorded predecessors, removing each last vertex from the set and prepending it to J. It therefore reconstructs the selected path, visiting every vertex exactly once. Closing this path returns an optimal tour. □

Theorem 39.2 (Running time)**.** TravellingSalesperson *runs in time* $O(n^2 2^n)$.

Proof. There are at most 2^{n-1} subsets and at most n choices of last vertex per subset. Each entry tries at most n predecessors, at constant cost per candidate. Subsets can be listed and grouped by cardinality in $O(n2^n)$ time by scanning their masks. Initialization, choosing the last vertex, and recovery take $O(n^2)$ additional time. These costs are bounded by $O(n^2 2^n)$. □

Part V

Max Flow / Min Cut

Chapter 40

Maximum Flow and Minimum Cut

A *flow network* is a directed graph with nonnegative integer edge capacities c, a source s, and a distinct sink t. A *flow* assigns $0 \le f(e) \le c(e)$ to each edge and has equal total inflow and outflow at every vertex except s, t. Its value $|f|$ is outflow minus inflow at s.

An s–t *cut* is a set S with $s \in S$ and $t \notin S$. Its capacity $c(S)$ is the sum of capacities of edges leaving S.

Maximum Flow

Input: A flow network G with capacities c and source and sink s, t.
Output: An s–t flow of maximum value.

Minimum Cut

Input: The same network.
Output: An s–t cut of minimum capacity.

40.1 Main idea

Start with zero flow and repeatedly find a route along which more flow can be sent. A route may use an edge forwards to increase its flow or backwards to cancel some existing flow. These options form the *residual graph*.

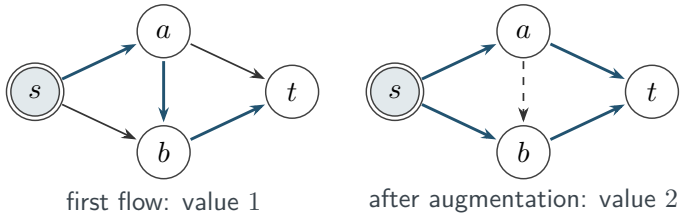


Figure 40.1: All capacities are 1. After sending flow along s, a, b, t, the residual route s, b, a, t cancels the flow on (a, b) and raises the value to 2. Blue edges carry one unit.

40.2 Lemmas

For each original edge $e = (u, v)$, the residual graph G_f has a forward arc e^+ from u to v of capacity $c(e) - f(e)$ when positive, and a backward arc e^- from v to u of capacity $f(e)$ when positive. Each residual arc retains its original edge and its sign. Keep different

arcs separate even when their endpoints coincide; this also handles original edges in both directions.

For a simple s–t path P in G_f, let b be its smallest residual capacity. *Augmenting* along P adds b to every original edge used forwards and subtracts b from every original edge used backwards.

Lemma 40.1 (Augmentation)**.** *Augmenting a feasible flow along a residual path gives a feasible flow of value $|f| + b$. If capacities and the old flow are integral, so is the new flow.*

Proof. A forward change is at most $c(e) - f(e)$; a backward change is at most $f(e)$. Thus every new flow remains between zero and capacity. A residual arc from u to v increases net outflow at u by b and decreases net outflow at v by b, whichever sign it has. The two changes at every internal path vertex cancel. Net outflow at s increases by b, and at t decreases by b. Conservation and the value assertion follow. With integral data the residual capacities, their minimum, and all updated flows are integers. □

Write $\delta^+(S)$ and $\delta^-(S)$ for the original edges leaving and entering S.

Lemma 40.2 (Flow across a cut)**.** *For every feasible flow and every s–t cut,*

$$|f| = \sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e) \le c(S).$$

Proof. Sum net outflow over the vertices of S. Conservation makes every term zero except the term $|f|$ at s. An edge internal to S appears once positively and once negatively and cancels. Edges leaving and entering S give the displayed difference. Bound the first sum by its capacities and the subtracted sum below by zero. □

Lemma 40.3 (A certificate of optimality)**.** *If t is unreachable from s in G_f, let S be the reachable set. Then $|f| = c(S)$; f is a maximum flow and S is a minimum cut.*

Proof. No residual arc leaves S. Thus each original edge leaving S is saturated, or it would have a forward residual arc. Each original edge entering S has zero flow, or its backward arc would leave S. Lemma 40.2 now gives $|f| = c(S)$. Every flow value is at most this cut capacity, and every cut capacity is at least this flow value, so both are optimal. □

40.3 Algorithm

A graph search returns a simple residual path when one exists and the reachable set otherwise. Store parent *arcs*, so parallel residual arcs cannot be confused. Residual lists can be rebuilt by scanning the original edges after each augmentation.

Algorithm FordFulkerson(G, c, s, t)

```
1 set f(e) ← 0 for every edge e
2 repeat
3    construct G_f; search from s
4    if t is not reached then return (f, the reached set)
5    recover a simple residual path P from s to t
6    b ← min{residual capacity of a : a ∈ P}
7    for each arc of P belonging to an original edge e do
8       add b to f(e) for a forward arc, or subtract b for a backward arc
```

Theorem 40.4 (Correctness)**.** FordFulkerson *solves both the* Maximum Flow *and* Minimum Cut *problems, returning an integral maximum flow.*

Proof. Zero flow is feasible and integral. Lemma 40.1 preserves both properties and raises the value by a positive integer at every augmentation. By Lemma 40.2, every value is bounded by $c(\{s\})$. Consequently only finitely many augmentations occur. Each search and update terminates, so eventually a search finds no path. Lemma 40.3 then proves that both returned objects are optimal. □

Corollary 40.5 (Max-flow/min-cut theorem)**.** *For integer-capacity networks, the maximum flow value equals the minimum cut capacity, and an integral maximum flow exists.*

Proof. The terminating algorithm returns an integral flow and a cut of equal value by Lemma 40.3. Weak duality in Lemma 40.2 shows that these are the two optima. □

Theorem 40.6 (Running time)**.** FordFulkerson *takes* $O((F+1)(n+m))$ *time, where* F *is the maximum flow value.*

Proof. Each augmentation raises the integral value by at least one, so there are at most F augmentations. Constructing a residual graph, searching it, and updating a path each take $O(n+m)$ time: there are at most $2m$ residual arcs. Include initialization and the final unsuccessful search to obtain $O((F+1)(n+m))$. □

The dependence on F is pseudopolynomial, as in *Pseudopolynomial Time*. The next chapter removes it by specifying which augmenting path to choose.

Chapter 41

Maximum Flow (revisited)

Maximum Flow and Minimum Cut

Input: A flow network G with capacities c and source and sink s, t.
Output: A maximum flow and a minimum s–t cut.

41.1 Main idea

Use BFS to choose an augmenting path with the fewest arcs. After an augmentation, BFS distances in the residual graph never decrease. An arc that blocks a shortest path can block another only after its tail has moved to a later BFS layer.

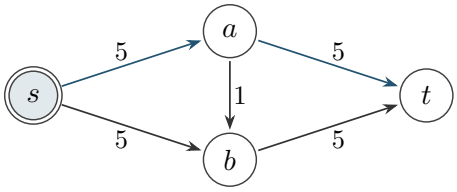


Figure 41.1: BFS selects a two-edge route, such as s, a, t, and sends five units at once. A route through (a, b) has three edges and a bottleneck of one.

41.2 Lemmas

Distances in this chapter count residual arcs, regardless of capacities. Unreachable vertices have distance $+\infty$.

Lemma 41.1 (Distances do not decrease)**.** *After augmentation along a BFS path, every residual distance from s is at least its previous value.*

Proof. Write d for the old distances. Every old residual arc (u, v) with reachable tail satisfies $d(v) \le d(u) + 1$. Every newly appearing arc is the reverse of an arc on the chosen shortest path. For such a new arc (u, v), the old distances satisfy $d(v) = d(u) - 1$, so the same inequality holds, and both endpoints were reachable.

Follow any path from s in the new residual graph. By induction over its arcs, every vertex on it was previously reachable, and its old distance is at most the length of the new path prefix ending there. Apply this to a new shortest path to obtain the claimed inequality. A previously unreachable vertex cannot become reachable by this argument. □

Lemma 41.2 (Number of augmentations)**.** *BFS path selection permits $O(nm)$ augmentations.*

Proof. An augmentation saturates at least one residual arc; charge the augmentation to one such arc, called critical. Fix a particular oriented residual arc $a = (u, v)$, retaining its original-edge identity. When it is critical, $d(v) = d(u) + 1$ because it lies on a shortest path. Before that same arc can be critical again, it must regain positive capacity. This requires using its paired reverse arc (v, u) in a later augmentation. At that later time, $d'(u) = d'(v) + 1$. By Lemma 41.1,

$$d'(u) = d'(v) + 1 \geq d(v) + 1 = d(u) + 2.$$

The tail's distance therefore increases by at least two between successive critical occurrences. A reachable distance is at most $n - 1$, so each oriented arc is critical only $O(n)$ times. There are at most $2m$ such arc identities, proving the bound. □

41.3 Algorithm

Only the search rule changes; the residual representation and augmentation operation are exactly those of Chapter 40. Before the repeated searches, discard isolated vertices other than s, t and renumber the remaining vertices, retaining their original identifiers for the output. Place discarded vertices outside the returned cut. This does not change any flow or cut capacity.

Algorithm EdmondsKarp(G, c, s, t)

```
1 set f(e) ← 0 for every edge
2 repeat
3    construct G_f and run BFS from s, storing parent arcs
4    if t is not reached then return (f, the reached set)
5    recover the BFS path to t and augment by its bottleneck capacity
```

Theorem 41.3 (Correctness)**.** EdmondsKarp *solves the* Maximum Flow and Minimum Cut *problem and returns an integral flow.*

Proof. A BFS path is a valid simple residual path, so every iteration is an iteration allowed by FordFulkerson. Theorem 40.4 therefore proves feasibility, integrality, termination, and optimality of both returned objects. Lemma 41.2 gives an additional termination bound independent of the capacity magnitudes. □

Theorem 41.4 (Running time)**.** EdmondsKarp *runs in time $O(n + m + nm^2)$.*

Proof. Remove isolated vertices other than s, t before the searches, retaining the original edge records. This costs $O(n + m)$. The remaining network has at most $2m + 2$ vertices and m edges. Each residual construction, BFS, and augmentation then costs $O(m + 1)$ time. By Lemma 41.2, there are $O(nm)$ augmentations, plus a final unsuccessful search. When $m \geq 1$, this costs $O(nm^2 + m)$; when $m = 0$, there is just the initial search. Isolated vertices can be placed outside the returned cut without changing its capacity. Including preprocessing gives $O(n + m + nm^2)$. □

Chapter 42

Edge-Disjoint Paths

Paths are *edge-disjoint* if no edge belongs to more than one of them. They may still share vertices.

Edge-Disjoint Directed Paths

Input: A directed graph G and distinct vertices s, t.
Output: A largest collection of pairwise edge-disjoint s–t paths.

42.1 Main idea

Give every edge capacity one and find a maximum integral flow. Extract one path at a time from edges carrying positive flow. Each extraction removes one unit, so a unit-capacity edge cannot appear twice.

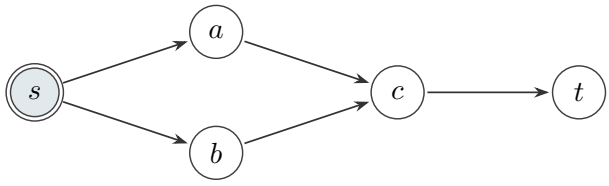


Figure 42.1: There are two routes to c, but both need (c, t). The one-edge cut leaving $\{s, a, b, c\}$ certifies that at most one edge-disjoint path can reach t.

42.2 Auxiliary problem

The following extraction procedure also applies to larger integral capacities. It need not account for flow circulating around cycles.

Flow Path Extraction

Input: A feasible integral s–t flow f of nonnegative value F.
Output: F unit paths, with each edge e used at most $f(e)$ times.

Lemma 42.1 (A positive-flow path)**.** *Every feasible flow of positive value has an s–t path using only positive-flow edges.*

Proof. Let S be the vertices reachable from s through positive-flow edges. If $t \notin S$, no positive flow leaves S. The cut identity in Lemma 40.2 would give $|f| = 0 - \sum_{e \in \delta^-(S)} f(e) \leq 0$, a contradiction. Thus t is reachable, and a search supplies a simple path. □

Algorithm ExtractFlowPaths(G, f, s, t)

```
copy f into g; F ← |f|; create an empty list P
repeat F times
    search the positive-g edges for a simple s–t path P
    append P to P; subtract one from g(e) on every edge of P
return P
```

Lemma 42.2 (Correctness)**.** ExtractFlowPaths *solves the* Flow Path Extraction *problem.*

Proof. Before iteration j, maintain that g is an integral feasible flow of value $F - j + 1$. Lemma 42.1 supplies the required path. Its edges have integer flow at least one. Subtracting one along it preserves nonnegativity and conservation and reduces the value by one. This proves the invariant. Since g never becomes negative, the extracted paths use each edge at most its initial flow. The finite number of terminating searches proves termination and the output claim. □

Lemma 42.3 (Running time)**.** ExtractFlowPaths *runs in time $O((F + 1)(n + m))$.*

Proof. Initialization takes $O(n + m)$ time. Each of F searches takes $O(n + m)$ time, and recording and subtracting its path takes $O(n)$. Adding the costs proves the bound. □

42.3 Algorithm

Algorithm EdgeDisjointPaths(G, s, t)

```
set c(e) ← 1 for every edge
(f, S) ← FordFulkerson(G, c, s, t)
return ExtractFlowPaths(G, f, s, t)
```

Theorem 42.4 (Correctness)**.** EdgeDisjointPaths *solves the* Edge-Disjoint Directed Paths *problem.*

Proof. Any k edge-disjoint paths give a feasible unit-capacity flow of value k by adding their unit flows. Thus their number is at most the maximum flow value F. The algorithm obtains an integral maximum flow and extracts F paths. Unit capacities and the extraction guarantee make these paths edge-disjoint, so the upper bound is attained. Both called algorithms terminate, proving termination. □

Theorem 42.5 (Running time)**.** EdgeDisjointPaths *runs in time* $O(n(n+m))$.

Proof. The original graph is simple, so the cut $\{s\}$ has capacity at most $n-1$ and $F \le n-1$. Theorem 40.6 and Lemma 42.3 each give $O(n(n+m))$ time. Constructing capacities costs $O(m)$, which is within the same bound. □

Corollary 42.6 (Edge-disjoint paths and edge cuts)**.** *The maximum number of edge-disjoint directed s–t paths equals the minimum number of edges whose deletion makes t unreachable from s.*

Proof. A unit-capacity cut has capacity equal to its number of outgoing edges, and deleting them destroys all s–t paths. Conversely, after any successful deletion, let S be the vertices still reachable from s. Every original edge leaving S must have been deleted. Thus the smallest deletion size equals the smallest cut capacity. Apply Theorem 40.5 and the path–flow correspondence above. □

Chapter 43

Vertex-Disjoint Paths

The *internal vertices* of an s–t path are all its vertices except s and t. Paths are *internally vertex-disjoint* if they share no internal vertex.

Internally Vertex-Disjoint Directed Paths

Input: A directed graph G and distinct vertices s, t.
Output: A largest collection of distinct s–t paths sharing no internal vertex.

43.1 Main idea

Replace each vertex other than s, t by an entrance and an exit joined by one edge. Incoming edges end at the entrance and outgoing edges start at the exit. Edge-disjoint paths in this new graph can use each internal vertex at most once in total.

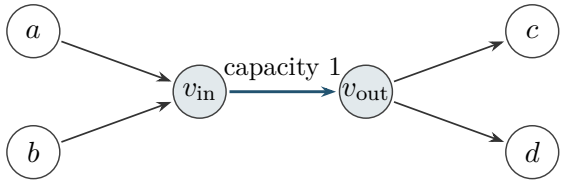


Figure 43.1: All passages through v must use the same middle edge. The two entrance choices and two exit choices cannot bypass its capacity.

43.2 Lemmas

Let H be the split graph. For an internal vertex v, add $(v_{\text{in}}, v_{\text{out}})$. Replace each original edge (u, v) by $(\text{out}(u), \text{in}(v))$, where $\text{in}(s) = \text{out}(s) = s$ and likewise for t.

Lemma 43.1 (Path correspondence)**.** *There are k internally vertex-disjoint paths in G if and only if there are k edge-disjoint paths in H.*

Proof. Expand each original path through the split edge of every internal vertex it uses. Distinct internally vertex-disjoint paths cannot share an original edge either: sharing one would share an internal endpoint, except for the single possible edge (s, t), which cannot be two distinct paths. Their expansions are therefore edge-disjoint.

Conversely, every internal entrance in H has only its split edge as an outgoing edge. An s–t path entering it must traverse that edge and then leave from the exit. Contracting these pairs gives a path in G. Edge-disjoint paths in H cannot use the same split edge, so their contractions share no internal vertex. They are distinct, including the possible direct path from s to t. □

43.3 Algorithm

Algorithm VertexDisjointPaths(G, s, t)

```
1 construct the split graph H
2 P ← EdgeDisjointPaths(H, s, t)
3 contract each entrance–exit pair in every path of P
4 return the resulting paths
```

Theorem 43.2 (Correctness)**.** VertexDisjointPaths *solves the* Internally Vertex-Disjoint Directed Paths *problem.*

Proof. The called algorithm terminates and returns a maximum collection of edge-disjoint paths in H. Lemma 43.1 converts them to a feasible collection of the same size in G. A larger collection in G would expand to a larger one in H, contradicting optimality. Construction and contraction are finite, proving termination. □

Theorem 43.3 (Running time)**.** VertexDisjointPaths *runs in time* $O(n(n+m))$.

Proof. The split graph has $2n-2$ vertices and $m+n-2$ edges, and is constructed in $O(n+m)$ time. The preceding chapter's bound is therefore $O(n(n+m))$ on this graph. There are at most $n-1$ paths, each with $O(n)$ vertices, so contraction costs $O(n^2)$, within the same bound. □

Corollary 43.4 (Vertex-disjoint paths and separators)**.** *If* $(s,t) \notin E$*, the maximum number of internally vertex-disjoint* s*–*t *paths equals the minimum number of vertices other than* s, t *whose deletion makes* t *unreachable from* s*.*

Proof. Every separating vertex set must meet each path, and disjoint interiors require distinct meeting vertices. Thus its size is at least the maximum number k of paths.

By Corollary 42.6, the split graph has an s–t cut with k outgoing edges. Replace each of its split edges by its original vertex. Replace each of its other edges by an internal endpoint in G; such an endpoint exists because (s,t) is absent and (t,s) cannot leave an s–t cut. These choices give at most k vertices. Any original path avoiding them would expand to a path avoiding the edge cut, which is impossible. They therefore separate s from t and attain the bound. □

Chapter 44

Edge-Disjoint Paths in Undirected Graphs

In an undirected graph, paths that traverse an edge in opposite directions still share that edge.

Edge-Disjoint Undirected Paths

Input: An undirected graph G and distinct vertices s, t.
Output: A largest collection of pairwise edge-disjoint s–t paths.

44.1 Main idea

Replace each edge by a unit-capacity arc in each direction. If a flow uses both directions, subtract their common amount. This leaves all net flow balances unchanged and prevents two extracted paths from using the same undirected edge in opposite directions.

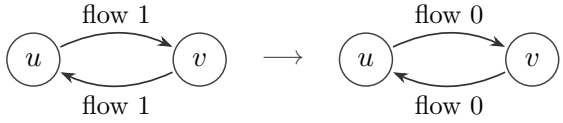


Figure 44.1: Canceling equal opposite flows changes neither vertex's net outflow. The same operation is performed for each original edge.

44.2 Lemmas

Lemma 44.1 (Cancellation)**.** *In the bidirected network, subtracting* $\min\{f(u,v), f(v,u)\}$ *from both opposing arcs preserves feasibility, integrality, and flow value, and leaves at least one direction unused.*

Proof. Both flows remain nonnegative and cannot exceed their old capacities. At each endpoint, incoming and outgoing flow decrease by the same amount, so net outflow is unchanged. Conservation and the value at s are therefore unchanged. Subtraction preserves integrality, and one of the two new values is zero. □

44.3 Algorithm

Algorithm UndirectedDisjointPaths(G, s, t)

```
1 replace each edge {u, v} by arcs (u, v), (v, u) of capacity one, obtaining H
2 (f, S) ← FordFulkerson(H, c, s, t)
3 for each original edge, cancel the common flow on its two arcs
4 P ← ExtractFlowPaths(H, f, s, t)
5 return these paths with directions forgotten
```

Theorem 44.2 (Correctness)**.** UndirectedDisjointPaths *solves the* Edge-Disjoint Undirected Paths *problem.*

Proof. Orient any collection of k undirected edge-disjoint paths from s to t. Their unit flows form a feasible flow of value k in H. Thus the optimum path count is at most the maximum flow value F.

The algorithm finds an integral maximum flow. Cancellation preserves its value and feasibility by Lemma 44.1. Extraction then returns F directed paths, using each arc at most once. Opposite arcs cannot both occur because one now has zero flow. Forgetting directions therefore gives F undirected edge-disjoint paths and attains the upper bound. All called algorithms and finite scans terminate. □

Theorem 44.3 (Running time)**.** UndirectedDisjointPaths *runs in time* $O(n(n+m))$.

Proof. The network has n vertices and $2m$ arcs. The maximum flow value is at most the source degree, hence at most $n-1$. Construction and cancellation take $O(n+m)$ time. The flow computation and extraction each take $O(n(n+m))$ time by their earlier bounds. Forgetting directions costs at most the size of the extracted paths and fits the same bound. □

Chapter 45

Bipartite Matching

A *matching* is a set of edges with no shared endpoint. It is *perfect* if every vertex is an endpoint of a matching edge.

Maximum Bipartite Matching

Input: A bipartite graph $G = (X \cup Y, E)$ with its two parts specified.
Output: A largest set of edges with no shared endpoint.

45.1 Main idea

Send flow from a new source through X, then along original edges to Y, and finally to a new sink. A unit of flow chooses a pair. Capacity one from the source and into the sink prevents repeated partners.

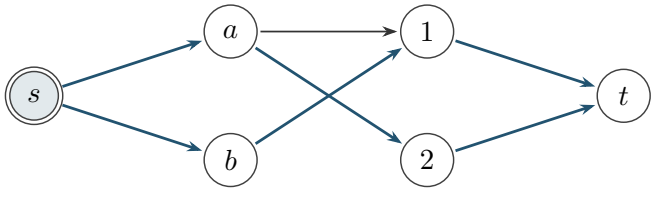


Figure 45.1: The matching $\{(a, 2), (b, 1)\}$ corresponds to a flow of value 2. Choosing $(a, 1)$ first can be undone through a backward residual arc.

45.2 Lemmas

The *matching network* has vertices $\{s, t\} \cup X \cup Y$. Add arcs (s, x) of capacity one, arcs (y, t) of capacity one, and (x, y) for every original edge, of capacity $M = n + 1$. A capacity of one on the middle arcs would also suffice here; the larger finite capacity will let the next chapter read vertex covers from cuts of this same network.

Lemma 45.1 (Matching–flow correspondence)**.** *A matching of size k gives an integral flow of value k. An integral flow of value k in the matching network gives a matching of size k, consisting of the positive-flow middle arcs.*

Proof. For each matched pair (x, y), send one unit on s, x, y, t. Distinct matched endpoints ensure the unit capacities, and every internal vertex has equal inflow and outflow.

Conversely, integrality and the unit-capacity edge into each x allow at most one positive outgoing middle arc, carrying one unit. The unit-capacity edge out of each y allows at

most one such incoming arc. These arcs are therefore a matching. Conservation gives their number as the total flow out of s, which is the flow value k. □

45.3 Algorithm

Building the matching network means making exactly the vertices and arcs specified above; one scan of the vertices and edges suffices.

Algorithm BipartiteMatching(G, X, Y)

```
1 construct the matching network H and its capacities c
2 (f, S) ← FordFulkerson(H, c, s, t)
3 return {{x, y} ∈ E : f(x, y) = 1}
```

Theorem 45.2 (Correctness)**.** BipartiteMatching *solves the* Maximum Bipartite Matching *problem.*

Proof. The maximum flow returned is integral. By Lemma 45.1, its positive middle arcs form a matching of the same size. A larger matching would give a flow of larger value, contradicting maximality. The flow algorithm and the finite construction and extraction scans terminate, completing the proof. □

Theorem 45.3 (Running time)**.** BipartiteMatching *runs in time* $O(n(n+m))$.

Proof. The network has $n + 2$ vertices and $m + n$ edges. Its flow value is at most $\min\{|X|, |Y|\} \le n$. Theorem 40.6 therefore gives $O(n(n+m))$ time. Construction and extraction take $O(n+m)$ and $O(m)$ time, respectively, within the same bound. □

Chapter 46

Minimum Vertex Cover in Bipartite Graphs

A *vertex cover* is a set of vertices containing at least one endpoint of every edge.

Minimum Bipartite Vertex Cover

Input: A bipartite graph $G = (X \cup Y, E)$ with its two parts specified.
Output: A smallest vertex set containing an endpoint of every edge.

46.1 Main idea

Use the matching network and find a minimum cut S. Select the vertices of X outside S and the vertices of Y inside S. An uncovered original edge would cross the cut on an expensive middle arc, which a minimum cut cannot afford.

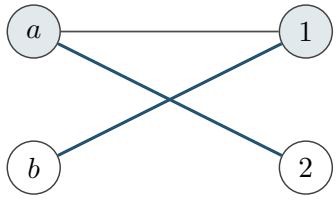


Figure 46.1: The cover $\{a, 1\}$ meets every edge. The two blue matching edges force every cover to contain at least two vertices.

46.2 Lemmas

Use the matching network of Chapter 45, including middle-arc capacity $M = n + 1$.

Lemma 46.1 (Cuts and covers)**.** *For a minimum cut S, the set $C = (X \setminus S) \cup (Y \cap S)$ is a vertex cover and has size $c(S)$. Conversely, each vertex cover C gives a cut of capacity $|C|$.*

Proof. The cut $\{s\}$ has capacity $|X| \leq n < M$, so a minimum cut cannot have a middle arc leaving it. Thus no edge has $x \in X \cap S$ and $y \in Y \setminus S$, precisely the condition that every edge meets C. The only arcs leaving S are then the unit arcs from s to $X \setminus S$ and from $Y \cap S$ to t. Their number is $|C|$.

Conversely, for a vertex cover C, take $S = \{s\} \cup (X \setminus C) \cup (Y \cap C)$. A middle arc leaving S would be an edge with neither endpoint in C, impossible. The crossing unit arcs correspond exactly to the vertices of C, giving capacity $|C|$. □

46.3 Algorithm

Algorithm BipartiteVertexCover(G, X, Y)

```
1 construct the matching network H with middle-arc capacity n + 1
2 (f, S) ← FordFulkerson(H, c, s, t)
3 return (X \ S) ∪ (Y ∩ S)
```

Theorem 46.2 (Correctness)**.** BipartiteVertexCover *solves the* Minimum Bipartite Vertex Cover *problem.*

Proof. The flow algorithm returns a minimum cut. By Lemma 46.1, the output is a vertex cover of that capacity. Every smaller cover would give a smaller cut, a contradiction. Construction and recovery are finite, and the flow algorithm terminates, so the whole algorithm terminates. □

Theorem 46.3 (Running time)**.** BipartiteVertexCover *runs in time* $O(n(n+m))$.

Proof. The network and flow computation are identical to those in Theorem 45.3. Extracting the cover takes $O(n)$ time by testing membership in the reached set, within that bound. □

Corollary 46.4 (Kőnig's theorem)**.** *In a bipartite graph, the maximum matching size equals the minimum vertex-cover size.*

Proof. Lemma 45.1 identifies the maximum matching size with the maximum flow value. Lemma 46.1 identifies the minimum cover size with the minimum cut capacity. These are equal by Theorem 40.5. □

Interlude

Certificates and Perfect Matchings

An optimal solution is especially useful when a second object certifies that no better solution exists. Flows and cuts, and matchings and covers, give concrete examples of this principle.

Any matching has size at most any vertex cover: its edges have disjoint endpoints, so covering them requires a different vertex for each edge. Thus a matching and a cover of the same size certify each other's optimality. Kőnig's theorem guarantees such a pair for bipartite graphs. Likewise, a feasible flow and a cut of equal value certify both optima by the cut inequality.

These certificates are useful even to someone who did not run the algorithm. Checking a matching, checking a cover, and comparing their sizes suffice to verify the claimed optimum. No enumeration of all matchings is required.

For $U \subseteq X$, write $N(U)$ for the vertices in Y adjacent to at least one vertex of U.

Theorem (Hall's theorem)**.** *A bipartite graph with $|X| = |Y|$ has a perfect matching if and only if $|N(U)| \geq |U|$ for every $U \subseteq X$.*

Proof. If a perfect matching exists, the matched neighbours of the vertices of U are $|U|$ distinct elements of $N(U)$. This proves necessity.

Conversely, suppose there is no perfect matching. By Theorem 46.4, there is a vertex cover C with $|C| < |X|$. Put $U = X \setminus C$. Since C covers every edge, every neighbour of U belongs to $Y \cap C$. Hence

$$|N(U)| \leq |Y \cap C| < |X| - |X \cap C| = |U|.$$

This violates the stated condition and proves sufficiency. □

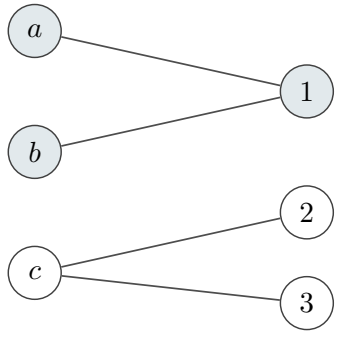


Figure 46.2: The set $U = \{a, b\}$ has only one neighbour. This small obstruction certifies that no perfect matching exists.

A graph is *d-regular* when every vertex has degree d.

Lemma (Regular bipartite graphs)**.** *Every finite d-regular bipartite graph with $d \geq 1$ has a perfect matching.*

Proof. Counting edges from both parts gives $d|X| = d|Y|$, so $|X| = |Y|$. For any $U \subseteq X$, its $d|U|$ incident edges end in $N(U)$, whose vertices have at most $d|N(U)|$ incident edges in total. Thus $|U| \leq |N(U)|$. Hall's theorem gives a perfect matching. □

Chapter 47

Baseball Elimination

Each remaining game has exactly one winner. We ask whether the remaining wins can be assigned so that a designated team finishes strictly first.

Possible League Winner

Input: An array of current wins w, a matrix of remaining game counts g, and a designated team x.
Output: Whether x can finish with strictly more wins than every other team; if so, a feasible allocation of the remaining wins.

The wins and game counts are nonnegative integers. The matrix g is symmetric with zero diagonal; g_{ij} counts games between teams i and j. An allocation records how many of those games each team wins, without listing games individually.

47.1 Main idea

Let x win all its remaining games. Its final total is then fixed. For each remaining game between other teams, send a unit of flow to its winner. A capacity at each team limits how many additional wins it can receive without catching x.

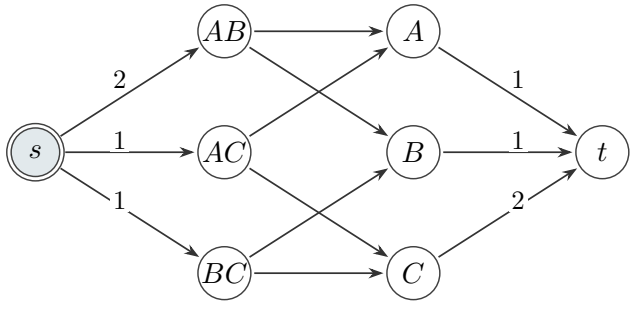


Figure 47.1: Team x can reach 10 wins; A, B, C currently have $8, 8, 7$. The capacities to t are $1, 1, 2$. Split the two AB games evenly and give both remaining games to C; each rival finishes with 9.

47.2 Lemmas

Put $W = w_x + \sum_j g_{xj}$ and $b_i = W - 1 - w_i$ for each rival $i \neq x$. If a budget b_i is negative, x cannot finish strictly first. Otherwise create a vertex p_{ij} for each rival pair with $g_{ij} > 0$, and a vertex for every rival. Add (s, p_{ij}) of capacity g_{ij}, (p_{ij}, i) and (p_{ij}, j) each of capacity g_{ij}, and (i, t) of capacity b_i. Let $R = \sum_{i<j,\ i,j\neq x} g_{ij}$.

Lemma 47.1 (Schedule–flow correspondence)**.** *Team x can finish strictly first if and only if all budgets are nonnegative and the constructed network admits an integral flow of value R.*

Proof. If a winning allocation exists, change every remaining game involving x to a win for x. This increases its total and can only decrease a rival's total, so it remains strictly first with W wins. Rival i can therefore receive at most b_i further wins. Send each rival game's unit of flow from its pair vertex to its winner and then to t. These flows satisfy the capacities and conservation, and all R units leave s.

Conversely, a flow of value R saturates all source arcs, since their capacities sum to R. Conservation and integrality split each pair's g_{ij} games into integer numbers of wins for i and j. Each rival receives at most b_i wins, so finishes with at most $W - 1$. Give x all its remaining games; it finishes with W and is strictly first. □

47.3 Algorithm

The output counts are obtained directly from game-to-team arc flows.

Algorithm PossibleWinner$(w[1..n], g[1..n, 1..n], x)$

```
1 W ← w_x + Σ_j g_xj; compute b_i ← W − 1 − w_i for each i ≠ x
2 if some b_i < 0 then return "impossible"
3 construct the network above and compute R
4 (f, S) ← EdmondsKarp(H, c, s, t)
5 if |f| < R then return "impossible"
6 assign all games involving x to x
7 for each rival pair with g_ij > 0, assign its two arc flows as wins
8 return "possible" and the allocation (using zero for absent pairs)
```

Theorem 47.2 (Correctness)**.** PossibleWinner *solves the* Possible League Winner *problem.*

Proof. A negative budget makes strict victory impossible even if x wins all its remaining games. Otherwise the flow algorithm returns an integral maximum flow. If its value is below R, no flow of value R exists, so Lemma 47.1 justifies the negative report. If its value is R, the same lemma proves that the recovered allocation makes x strictly first. No larger value is possible because the source capacities total R. All construction and recovery scans are finite, and Edmonds–Karp terminates. □

Theorem 47.3 (Running time)**.** PossibleWinner *runs in time $O(n^6)$.*

Proof. Reading game counts, building the network, and recovering the allocation cost $O(n^2)$ time. There are $O(n^2)$ game and team vertices and $O(n^2)$ arcs. Theorem 41.4 bounds the flow computation by $O(n^2 + n^2(n^2)^2) = O(n^6)$. This bound counts operations on the game counts; it does not expand one record per game. □

Chapter 48

Project Selection

Each project earns a profit and requires some equipment. Equipment is bought at most once and can serve every project that needs it.

Project Selection

Input: Arrays of nonnegative integer project profits p and equipment costs c, and each project's required equipment.
Output: Projects to undertake and equipment to buy, meeting every chosen project's requirements and maximizing total profit minus total cost.

48.1 Main idea

Put chosen projects and purchased equipment on the source side of a cut. Cutting a source-to-project arc pays for profit forgone; cutting an equipment-to-sink arc pays for equipment bought. A sufficiently expensive project-to-equipment arc forbids choosing a project without its equipment.

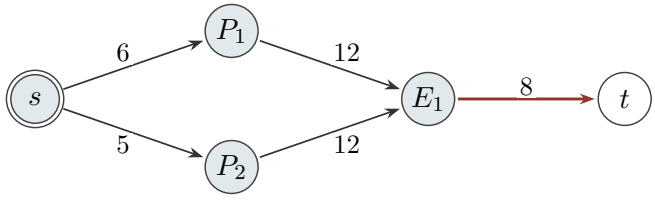


Figure 48.1: Both projects require the same equipment. Either alone loses money, but together they yield $6 + 5 - 8 = 3$. The cut with all project and equipment vertices on the source side has capacity 8.

48.2 Lemmas

Write $P = \sum_i p_i$ and set $M = P + 1$. Build a network with a vertex for each project and equipment item, source arcs (s, i) of capacity p_i, sink arcs (j, t) of capacity c_j, and arcs (i, j) of capacity M for every equipment requirement. The finite choice of M replaces a formal infinite capacity.

Lemma 48.1 (Selections and cuts)**.** *A minimum cut selects a feasible set of projects and equipment on its source side, of net profit $P - c(S)$. Conversely, every feasible selection of net profit z gives a cut of capacity $P - z$.*

Proof. The cut $\{s\}$ has capacity $P < M$, so no minimum cut can have a requirement arc leaving it. Thus every selected project has all its required equipment on the source side. The crossing arcs are precisely those for unselected project profits and selected equipment costs:

$$c(S) = \sum_{i \notin S} p_i + \sum_{j \in S} c_j = P - \left(\sum_{i \in S} p_i - \sum_{j \in S} c_j \right).$$

This proves feasibility and the claimed profit.

Conversely, put s, all selected projects, and all purchased equipment on the source side. Feasibility prevents any requirement arc from leaving it. The same identity gives capacity $P - z$. □

48.3 Algorithm

Keep project and equipment vertex types distinct even when their numerical indices agree. Let r be the total number of listed requirements.

Algorithm ProjectSelection($p[1..n], c[1..m]$, requirements)

1 $P \leftarrow \sum_i p_i$; $M \leftarrow P + 1$
2 construct the network above
3 $(f, S) \leftarrow$ EdmondsKarp(H, capacity, s, t)
4 **return** the project vertices in S and the equipment vertices in S

Theorem 48.2 (Correctness)**.** ProjectSelection *solves the* Project Selection *problem.*

Proof. The flow algorithm returns a minimum cut. By Lemma 48.1, the returned projects and equipment are feasible and have profit $P - c(S)$. A feasible selection with larger profit would give a cut with smaller capacity, contradicting minimality. The network construction and output scan are finite, and Edmonds–Karp terminates, proving termination. □

Theorem 48.3 (Running time)**.** ProjectSelection *runs in time $O((n+m)(n+m+r)^2)$.*

Proof. The network has $n + m + 2$ vertices and $n + m + r$ arcs. Construction takes $O(n + m + r)$ time and recovery takes $O(n + m)$. By Theorem 41.4, the flow computation takes $O((n + m)(n + m + r)^2)$ time, which also bounds the other work. □

Acknowledgments

The topics covered in these notes and the specific presentation of the different algorithms owes much to the many instructors of the *CS 341: Algorithms* course at the University of Waterloo over the years:

Spyros Angelopoulos
Andrew Arnold
Shalev Ben-David
Therese Biedl
Eric Blais
Daniel G. Brown
Trevor Brown
Forbes Burkowski
Jonathan Buss
Eowyn Cenek
Timothy M. Chan
Paul Church
William Cowan
Reza Dorrigiv
Rudolf Fleischer
Elena Grigorescu
Nathan Harms
Meng He
M. Jason Hinek
Armin Jamshidpey
Kevin Lanctot
Lap Chi Lau
Amit Levi
Ming Li
Alejandro López-Ortiz
Yang Lu
Anna Lubiw
Bin Ma
J. Ian Munro
Yakov Nekrich
Naomi Nishimura
Rafael Oliveira
Richard Peng
Mark Petrick
Prabhakar Ragde
Collin Roberts
Masoomeh Rudafshani
Semih Salihoğlu
Luke Schaeffer
Éric Schost
Dale Schuurmans
Jeffrey Shallit
Murray Sherk
Tim Smith
Douglas R. Stinson
Arne Storjohann
Santhoshini Velusamy
Tomáš Vinař
Hong Zhou
Eugene Zima

The current offering of CS 341 can typically be found at

`https://student.cs.uwaterloo.ca/~cs341/`

Further Reading

The following textbooks, listed on the CS 341 course website, offer complementary accounts of algorithm design and analysis.

Algorithms
Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. McGraw-Hill, 2008.

A concise treatment that emphasizes the mathematical idea behind each algorithm. Useful for seeing how divide and conquer, graph algorithms, greedy methods, and dynamic programming fit into a common approach.

Algorithms
Jeff Erickson. First edition, 2019.

Detailed explanations of design techniques, correctness arguments, and graph algorithms, with extensive exercises. A useful companion for practising how to turn an algorithmic idea into a complete proof.

Introduction to Algorithms
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Fourth edition, MIT Press, 2022.

A broad reference with detailed analyses and substantial coverage of data structures. Useful when a short treatment here raises questions about implementation, related algorithms, or more advanced results.

The Algorithm Design Manual
Steven S. Skiena. Third edition, Springer, 2020.

An application-oriented view of algorithm design, with a catalogue of algorithmic problems and techniques. Useful for recognizing a familiar problem inside a new application and finding suitable methods.

Algorithm Design
Jon Kleinberg and Éva Tardos. Addison-Wesley, 2006.

Organized around design techniques and the problems that motivate them. Especially useful for further examples of greedy arguments, dynamic programming, and network-flow applications, with substantial problem sets.

Index

0–9
2SUM, 1–3

A
adjacency lists, 35
adjacency matrix, 30
alignment, 105
arrays
    conventions, 1
    sorted, 1
articulation vertices, *see* cut vertices
augmenting paths, 121

B
back edges, 51
baseball elimination, 138–139
Bellman–Ford algorithm, 111–112
BellmanFord, 112
BFS, 37
BFSPaths, 40
BFSTree, 38
binary search, 91
binary search trees, 109
    optimal, 109–110
Bipartite, 45
bipartite graphs
    recognition, 44–46
BipartiteMatching, 133
BipartiteVertexCover, 135
bipartition, 44
boundary distance, 78
breadth-first search, 36–37
    tree structure, 38–39
bridges, *see* cut edges

C
certificates of optimality, 136–137
Closest, 21
closest pair of points, 20–22
ColorIntervals, 74
CommonSubsequence, 104
complete graphs, 117
completion time, 75
    minimizing total, 75–76
component graph, 65
Components, 43
connected components, 42–43
connectedness, 35
correctness, 2
counting inversions, *see* inversions, counting
Crossing, 15
cut edges, 56–57
cut property, 81
cut vertices, 52–55
CutEdges, 57
cuts
    in flow networks, 120
    undirected graphs, 81
CutVertices, 55
cycles, 35
    directed, 62

D
DAG, *see* directed acyclic graphs
degree, 35
depth-first search, 47–48
    forest, 63
    tree structure, 49–51
DFS, 48
DFSTimes, 63
DFSTree, 50
Dijkstra, 78
Dijkstra algorithm, 77–79
directed acyclic graphs, 62–64
directed graphs, 58
disjoint paths
    edge-disjoint, directed, 125–127
    edge-disjoint, undirected, 130–131
    internally vertex-disjoint, 128–129
disjoint sets, 83–84
distance
    weighted, 77
divide and conquer, 5
dynamic programming, 88–90
    bottom-up evaluation, 88
    memorization, 88

E
EdgeDisjointPaths, 126
edges, 35
edit distance, 105–106
EditDistance, 106
Edmonds–Karp algorithm, 123–124
EdmondsKarp, 124
exchange arguments, 69
ExtractFlowPaths, 126

F
FastIncreasingSubsequence, 102
Fibonacci, 90
Fibonacci numbers, 88–90
FibonacciTopDown, 89
Find, 84
finish time, 49
flow networks, 120
flows, 120
Floyd–Warshall algorithm, 115–116
FloydWarshall, 115
Ford–Fulkerson algorithm, 120–122
FordFulkerson, 121

G
graph reversal, 60
graphs
    adjacency lists, 35
    adjacency matrix, 30
    basic definitions, 35
greedy algorithms, 69

H
Hall's theorem, 136
heaps
    binary min-heaps, 71–72

I
IncreasingSubsequence, 99
independent set, 107
    general graphs, 32–33
    trees, 107–108
IndependentSet, 32
induction, 7
integer multiplication, 23–25
interval coloring, 73–74
interval scheduling
    unweighted, 69–70
    weighted, 91–93
inversions, counting, 11–13
    crossing inversions, 12
IsDAG, 64

K
Karatsuba, 24
Karatsuba multiplication, 23–25
Knapsack, 97
knapsack, 97–98
Kőnig's theorem, 135
Kruskal, 86
Kruskal algorithm, 85–86

L
longest common subsequence, *see* subsequences
longest increasing subsequence, *see* subsequences
loop invariants, 2
low values, 54
LowValues, 54

M
MakeSets, 83
Master Theorem, 26–27
matching networks, 132
matchings, 132
    bipartite graphs, 132–133
    perfect, 132, 136–137
matrix multiplication
    Strassen algorithm, 28–29
    triangle finding, 30–31
max-flow/min-cut theorem, 122
maximum flow, 120–122
maximum independent set, *see* independent set
maximum subarray, 14–16
    crossing the midpoint, 15
MaxSubarray, 16
median, 17–19
median of medians, 17
memorization, 88
Menger's theorem
    edge version, 127
Merge, 6
merge sort, 5–8
MergeCount, 12
MergeSort, 7
merging, 6
minimum cut, 120–122
minimum spanning trees
    Kruskal algorithm, 85–86
    Prim algorithm, 80–82
minimum vertex cover, *see* vertex cover
multiplication
    integers, 24
    matrices, 29

N
negative cycles, 111
    finding a cycle, 113–114
NegativeCycle, 114
neighbours, 35
Next, 91
non-tree edges
    BFS trees, 38
    DFS trees, 49

O
OptimalSearchTree, 110

P
padding arguments, 9–10
paths, 35
    directed, 58
    internal vertices, 128
pivot, 17
PossibleWinner, 139
Prim, 81
Prim algorithm, 80–82
priority queues, 71–72
project selection, 140–141
ProjectSelection, 141
pseudocode, 2
pseudopolynomial time, 96

R
reachability
    breadth-first search, 36–37
    depth-first search, 47–48
    directed graphs, 58–59
recurrences
    Master Theorem, 26–27
recursion trees, 26–27
recursive branching, 32–33
regular graphs, 137
residual graphs, 120
running time, 3
    worst case, 3

S
ScheduleIntervals, 70
Select, 18
selection, 17–19
shortest paths
    all pairs, 115–116
    negative weights, 111–112
    nonnegative weights, 77–79
    unweighted graphs, 40–41
ShortestFirst, 76
Sort, 9
SortCount, 13
sorting
    merge sort, 5–8
spanning trees, 80
start time, 49
Strassen, 29
strings, 103
strong connectivity, 60–61
StrongComponents, 67
strongly connected components, 65–67
StronglyConnected, 61
subsequences, 99
    longest common, 103–104
    longest increasing, faster, 101–102
    longest increasing, quadratic, 99–100
subset sum, 94–95
SubsetSum, 94

T
topological ordering, 62
tours, 117
travelling salesperson, 117–118
TravellingSalesperson, 118
tree edges, 38
TreeIndependentSet, 108
trees
    ancestors, 49
    depth, 38
    descendants, 49
    parent, 38
    subtrees, 49
Triangle, 30
triangle finding, 30–31
two-pointer method, *see* 2SUM
TwoSum, 2

U
UndirectedDisjointPaths, 131
Union, 84
union–find, *see* disjoint sets

V
vertex cover, 134
    bipartite graphs, 134–135
VertexDisjointPaths, 129
vertices, 35

W
walks, 35
    directed, 58
weighted graphs, 77
WeightedIntervals, 93
WeightedIntervalsTopDown, 92
Word-RAM model, 3