CSCI3160 Design and Analysis of Algorithms

Ryan Chan

December 11, 2025

Abstract

This is a note for **CSCI3160 Design and Analysis of Algorithms**.
Contents are adapted from the lecture notes of CSCI3160, prepared by Xiao Liang and Yufei Tao, as well as some online resources.
This note is intended solely as a study aid. While I have done my best to ensure the accuracy of the content, I do not take responsibility for any errors or inaccuracies that may be present. Please use the material thoughtfully and at your own discretion.
If you believe any part of this content infringes on copyright, feel free to contact me, and I will address it promptly.
Mistakes might be found. So please feel free to point out any mistakes.

Contents

Chapter 1

Introduction

Computer science is a subject where we first define a computation model, which is a simple yet accurate abstraction of a computing machine, and then we gradually build up a theory based on this model.

Thus, the first thing to do is to come up with a computation model. For the study of algorithms, we need to consider the following criteria:

1. **Mechanically Implementable:** it can run real algorithms.

2. **Sufficiently General:** it can capture all the natural steps of problem-solving.

3. **Sensitive Enough:** it can distinguish differences in resource usage (time and space).

We choose the **Random Access Machine (RAM)** as our model.

1.1 The RAM Model

The RAM model has a **memory** and a **CPU**.

Memory is defined as an infinite sequence of cells, each containing the same number of w bits, with addresses 1, 2, and so on. The CPU contains a fixed number of registers, each of which has w bits. A word is a sequence of w bits, where w is called the word length. In other words, each memory cell and CPU register stores a word.

1.1.1 Atomic Operations

We say that there are a few atomic operations that the CPU can perform.

1. **Register (Re-)Initialization**

We can set a register to a fixed value or to the content of another register.

2. **Arithmetic**

This operation takes two integers stored in two registers and performs basic arithmetic calculations.

3. **Comparison / Branching**

This operation takes two integers stored in two registers, compares them, and determines the result of the comparison.

4. **Memory Access**

This operation takes a memory address that is currently stored in a register, then performs either reading (to register) or writing (to memory).

5. **Randomness**

`RANDOM(x, y)` returns an integer chosen **uniformly** at random in $[x, y]$, where $x \leq y$. The resulting random integer is then placed in a register.

An execution is defined as a sequence of atomic operations, where the cost (running time) of an execution is the length of such a sequence, i.e., the number of atomic operations it performs.

1.1.2 Algorithms

We first take a look at some terminologies.

An input refers to the initial state of the registers and the memory before an execution starts.

An algorithm is a description that, given an input, can be utilized to **unambiguously** produce a sequence of atomic operations — namely, the execution of the algorithm. In other words, it should always be clear what the next atomic operation should be, given the outcomes of all the previous atomic operations.

The cost of an algorithm on an input is the length of its execution on that input (i.e., the number of atomic operations required).

The space of an algorithm on an input is the largest memory address accessed by the algorithm's execution on that input.

We define an algorithm as **deterministic** if it never invokes the atomic operation `RANDOM`; otherwise, the algorithm is **randomized**.

1.1.3 Expected Running Time

A deterministic algorithm has a fixed cost for the same input, whereas for a randomized algorithm, the cost is a random variable, since for each input the cost might change every time the algorithm is executed.

Algorithm 1.1: Find a One

```
1 while r ≠ 1 do
2     r = RANDOM(0, 1)
3 return r = 1
```

Here, we cannot know how many times line 2 will be executed, as each execution can produce a new sequence of atomic operations.

Thus, we use the **expected cost** to evaluate the cost of a randomized algorithm.

Definition 1.1.1. Let X be a random variable that represents the cost of an algorithm on a given input. The expected cost of the algorithm on that input is the expectation of X.

We use the expected running time, rather than other metrics, for the following reasons:

1. Ease of computation
2. Linearity of expectation: it allows us to break the analysis into smaller parts
3. Concentration bounds

Theorem 1.1.1 (Markov's Inequality)**.** Suppose that a random variable $X \geq 0$ only takes non-negative values. Then, for every $t > 0$,

$$\mathbb{P}(X \geq t) \leq \frac{\mathbb{E}[X]}{t}.$$

The theorem here shows that if the average value of X is small, then it is less likely that X is large. This provides an upper bound on the probability that a non-negative random variable is much larger than its expectation.

Let T be the random variable representing the running time of a randomized algorithm, and suppose $T \geq 0$ and $\mathbb{E}[T] = \mu$. Then, for any $c > 1$,

$$\mathbb{P}(T \geq c\mu) \leq \frac{1}{c}.$$

Example (Las Vegas Algorithm). A Las Vegas algorithm always gives the correct result but has a random running time.

Suppose $\mathbb{E}[T] = \mu$, and we run this algorithm with a timeout of $c\mu$. Then we have

$$\mathbb{P}[\text{timeout}] \leq \frac{1}{c}.$$

This shows that if we run the algorithm for c times its expected running time, then the probability that it fails will be less than or equal to $\frac{1}{c}$.

For example, if we run for 3μ steps, then it will fail with probability $\leq \frac{1}{3}$. Therefore, if we repeat it 3 times, we will succeed with probability $1 - \left(\frac{1}{3}\right)^3 = 0.963$.

1.2 Efficiency of the Worst Input

By looking at the cost of an algorithm on the worst input, we are able to measure the speed of an algorithm.

Definition 1.2.1. Define $\mathcal{I}_n$, where n is an integer, to be the set of all inputs to a problem that have the same problem size n. Given an input $I \in \mathcal{I}_n$, the cost $X_{\mathcal{A}}(I)$ of an algorithm $\mathcal{A}$ is the length of its execution on I.

The worst-case cost of $\mathcal{A}$ under the problem size n is the maximum $X_{\mathcal{A}}(I)$ over all $I \in \mathcal{I}_n$.

The worst expected cost of $\mathcal{A}$ under the problem size n is the maximum $\mathbb{E}[X_{\mathcal{A}}(I)]$ over all $I \in \mathcal{I}_n$.

Example (Dictionary Search). In memory, a set S of n integers has been arranged in ascending order in the memory cells from address 1 to n. The value of n has been placed in Register 1 of the CPU. Another integer v has been placed in Register 2, where n is the problem size, and $\mathcal{I}_n$ is the set of all possible (S, v).

We want to determine whether v exists in S.

The worst-case cost of the binary search algorithm is $\mathcal{O}(\log n)$. This means that on any input in $\mathcal{I}_n$, the maximum number $f(n)$ of atomic operations performed by the algorithm grows no faster than $\log_2 n$.

Example (Randomized Algorithm). Consider the following randomized algorithm:

Algorithm 1.2: Find a Zero

Data: A is an array of size n that contains at least one 0

```
1 while A[r] ≠ 0 do
2     r = RANDOM(1, n)
3 return r
```

The expected cost depends on the input array. For example, if all numbers in A are 0, then the algorithm finishes in $\mathcal{O}(1)$. If it has only one 0, the algorithm will finish in $\mathcal{O}(n)$ because each $A[r]$ has a $\frac{1}{n}$ probability of being 0, and we need to repeat n times in expectation to find the 0.

Thus, we have worst-case cost $= \infty$ and worst expected cost $= \mathcal{O}(n)$.

Example (Find a Zero). Let A be an array of n integers, among which half are 0. Design an algorithm to report an arbitrary position of A that contains a 0.

Algorithm 1.3: Find a Zero

```
while A[r] ≠ 0 do
    r = RANDOM(1, n)
return r
```

The algorithm finishes in $\mathcal{O}(1)$ expected time on every input A.

Proof. Let X be the number of iterations until the algorithm picks a zero. Each iteration chooses $r \in \{1, \ldots, n\}$.

Then we have

$$p = \mathbb{P}[A[r] = 0] = \frac{\#\text{ of zeros}}{n} = \frac{\frac{n}{2}}{n} = \frac{1}{2}.$$

Then X is a geometric random variable with success probability $p = \frac{1}{2}$.

For a geometric random variable, we have

$$\mathbb{E}[X] = \frac{1}{p} = \frac{1}{\frac{1}{2}} = 2.$$

Since each iteration takes $\mathcal{O}(1)$ time to pick a random index, we need two such iterations in expectation. Thus, the expected running time is $2 \times \mathcal{O}(1) = \mathcal{O}(1)$. ■

Remark. In contrast, any deterministic algorithm must probe at least $\frac{n}{2}$ integers of A in the worst case. In other words, any deterministic algorithm must have a worst-case time of $\Theta(n)$ — provably slower than the above randomized algorithm (in expectation).

1.3 Basic Techniques

In algorithm, there are three basic techniques we can utilize.

1.3.1 Recursion

When dealing with a sub-problem (the same problem but with a smaller input), consider it solved, and use the sub-problem's output to continue the algorithm design.

Example (The Hanoi Tower Problem). There are 3 rods A, B, and C. On rod A, n disks of different sizes are stacked, such that no disk of a larger size is above a disk of a smaller size. The other two rods are empty.

It is allowed to move the top-most disk of a rod to another, while no disk of a larger size can be put above a disk of a smaller size. The goal is to design an algorithm to move all the disks to rod B.

To move the largest disk, i.e., the n-th disk, to rod B, we first need to move all disks above it to rod C. For all disks, we use this method; thus, if we ignore the last disk, the remaining problem is to move $n - 1$ disks. This is the same problem with a smaller size.

Suppose that the algorithm performs $f(n)$ operations to solve a problem of size n, where $f(1) = 1$. By recursion, we have

$$f(n) \geq 1 + 2 \times f(n - 1).$$

Solving this gives

$$f(n) \geq 2^n - 1.$$

Thus, the best time complexity for solving this problem is $\Omega(2^n)$, where n is the number of disks.

1.3.2 Repeating till Success

Given a set S of n integers in an array and an integer $k \in [1, n]$, we want to find the k-th smallest integer of S.

Definition 1.3.1 (Rank). The rank of an integer $v \in S$ is the number of elements in S smaller than or equal to v.

For example, suppose that $S = (53, 92, 85, 23, 35, 12, 68, 74)$, then the rank of 53 is 4.

We can obtain the rank of v in $\mathcal{O}(|S|)$ time.

Example (Sub-problem of The k-Selection Problem). Assume n is a multiple of 3. We want to obtain a subproblem of size at most $\frac{2n}{3}$ with exactly the same result as the original problem.

Our goal is to produce a set S' and an integer k' such that $|S'| \leq \frac{2n}{3}$, $k' \in [1, |S'|]$, and the element with rank k' in S' is the element with rank k in S, where it is possible that $k \neq k'$.

Consider the following algorithm A_{sub}:

We first take an element $v \in S$ uniformly at random. Then we divide S into S_1 and S_2, where S_1 is the set of elements in S that are less than or equal to v, and S_2 is the set of elements in S greater than v.

If $|S_1| \geq k$, then return $S' = S_1$ and $k' = k$. Otherwise, return $S' = S_2$ and $k' = k - |S_1|$.

This algorithm succeeds if $|S'| \leq \frac{2n}{3}$, and fails otherwise.

We repeat this algorithm until it succeeds.

Lemma 1.3.1. The algorithm succeeds with probability at least $\frac{1}{3}$.

Proof. Let S be a set of size n and let $v \in S$ be chosen uniformly at random. Let the rank of v in S be $\text{rank}(v)$.

Define $S_1 = \{x \in S : x \leq v\}$, $S_2 = \{x \in S : x > v\}$. If $|S_1| \geq k$, return $S' = S_1$ and $k' = k$; otherwise $S' = S_2$ and $k' = k - |S_1|$. The sub-problem succeeds if $|S'| \leq \frac{2n}{3}$.

If $|S_1| \geq k$, then $|S'| = |S_1| = \text{rank}(v)$, and success requires

$$\text{rank}(v) \leq \frac{2n}{3}.$$

If $|S_1| < k$, then $|S'| = |S_2| = n - \text{rank}(v)$, and success requires

$$n - \text{rank}(v) \leq \frac{2n}{3} \implies \text{rank}(v) \geq \frac{n}{3}.$$

Then we have

$$\text{rank}(v) \in \left[\frac{n}{3}, \frac{2n}{3}\right].$$

Since v is chosen uniformly at random, every rank is equally likely, and the number of ranks in the success range is at least $\frac{2n}{3} - \frac{n}{3} = \frac{n}{3}$. Thus we have

$$\mathbb{P}[\text{success}] \geq \frac{\frac{n}{3}}{n} = \frac{1}{3}.$$

■

Remark. Choosing $\frac{2n}{3}$ is not mandatory, but rather a convenient threshold since it gives a smaller sub-problem. For example, if we say it succeeds when $|S'| \leq \frac{9n}{10}$, we have the range $[\frac{n}{10}, \frac{9n}{10}]$, giving the probability of success $\frac{8}{10}$, which is high. However, this is simply because we have a larger range to choose from, making the sub-problem larger.

In general, if an algorithm succeeds with a probability at least $c > 0$, then the number of repeats needed

for the algorithm to succeed for the first time is at most $\frac{1}{c}$ in expectation.

1.3.3 Geometric Series

Definition 1.3.2 (Geometric Series). A geometric sequence is an infinite sequence of the form

$$n, cn, c^2n, c^3n, \dots$$

where n is a positive number and c is a constant satisfying $0 < c < 1$. It holds in general that

$$\sum_{i=0}^{\infty} c^i n = \lim_{i\to\infty} n \cdot \frac{1-c^i}{1-c} = \frac{n}{1-c} = \mathcal{O}(n).$$

The summation $\sum_{i=0}^{\infty} c^i n$ is called a geometric series.

Consider again the k-th selection problem. Using the previous repeating technique, we can convert the problem into a subproblem with size at most $\lceil \frac{2n}{3} \rceil$ in $\mathcal{O}(n)$ expected time. We can use the geometric series to find the expected running time:

$$\begin{aligned} a \cdot n + a \cdot \frac{2}{3} \cdot n + a \cdot \left(\frac{2}{3}\right)^2 \cdot n + \cdots &= a \cdot n + a \cdot \sum_{i=1}^{\infty} \left(\frac{2}{3}\right)^i \cdot n \\ &= a \cdot n + a \cdot \mathcal{O}(n) \\ &= \mathcal{O}(n). \end{aligned}$$

Next, we analyze the running time of A_{sub}. It takes $\mathcal{O}(1)$ to select the element v, and dividing S into S_1 and S_2 takes $\mathcal{O}(n)$. Comparing $|S_1|, |S_2|$, and k also takes $\mathcal{O}(n)$. Thus, by the linearity of expectation, every conversion takes $\mathcal{O}(n)$ running time in expectation.

By the previous lemma, we need to repeat the algorithm 3 times in expectation until it succeeds, and each execution takes $\mathcal{O}(n)$ time. Therefore, the k-th selection algorithm takes $\mathcal{O}(n)$ time in expectation.

Chapter 2

Divide and Conquer

We take a look at divide and conquer, which guarantees strong performance. When dividing, we utilize recursion to reduce the problem into sub-problems, and the conquer part tackles the original problem.

2.1 Sorting

We consider merge sort in this example.

Problem 2.1.1. Given an array A of n distinct integers, produce another array where the same integers are arranged in ascending order.

Divide

Let A_1 be the array containing the first $\lceil \frac{n}{2} \rceil$ elements of A, and A_2 be the array containing the remaining elements of A. We sort A_1 and A_2 recursively.

Conquer

Merge the two sorted arrays in ascending order, which can be done in $\mathcal{O}(n)$ time.

Analysis

Let $f(n)$ denote the worst-case cost of the algorithm on an array of size n. Then,

$$f(n) \leq 2f\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n),$$

which gives[1]

$$f(n) = \mathcal{O}(n \log n).$$

2.2 Counting Inversions

Problem 2.2.1. Given an array A of n distinct integers, count the number of inversions, where an inversion is a pair (i, j) such that $1 \leq i < j \leq n$ and $A[i] > A[j]$.

Divide

Let A_1 be the array containing the first $\lceil \frac{n}{2} \rceil$ elements of A, and A_2 be the array containing the remaining elements of A. We solve the problem recursively on A_1 and A_2.

Conquer

It remains to count the number of *crossing inversions* (i, j) where $i \in A_1$ and $j \in A_2$. Using merge sort, we can sort A_1 in $\mathcal{O}(n \log n)$ time. For each element $e \in A_2$, we count how many crossing inversions

[1] See Master Theorem A.

e produces using binary search. In total, there are $\frac{n}{2}$ binary searches performed, each taking $\mathcal{O}(\log n)$ time, giving $\mathcal{O}(n \log n)$ time for this step.

Analysis

A simple comparison of each pair takes

$$(n-1) + (n-2) + \cdots + 1 = \mathcal{O}(n^2).$$

Let $f(n)$ denote the worst-case cost of the algorithm on an array of size n. Then,

$$f(n) \leq 2f\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n \log n),$$

which gives

$$f(n) = \mathcal{O}(n \log^2 n).$$

2.3 Dominance Counting

Problem 2.3.1. Denote $\mathbb{Z}$ as the set of integers. Given a point p in $\mathbb{Z}^2$, denoted by $p[1], p[2]$ its x- and y-coordinate, given two distinct points p and q, we say that q dominates p if $p[1] \leq q[1]$ and $p[2] \leq q[2]$.

Let P be a set of n points in $\mathbb{Z}^2$ with distinct x-coordinates. Find for each point $p \in P$ the number of points in P that are dominated by p.

Assume that, without loss of generality, the points are given in ascending order in the x-coordinates, i.e.

$$p_1[1] < p_2[1] < \cdots < p_n[1].$$

Divide

Let l be the vertical line such that P has $\lceil \frac{n}{2} \rceil$ points on each side of the line, where P_1 is the set of points of P on the left of l, and P_2 is the set of points on the right of l. We solve the problem recursively on P_1 and P_2.

Conquer

It remains to count for each point $p_2 \in P_2$ how many points in P_1 it dominates. We sort P_1 by its y-coordinate. Then for each point $p_2 \in P_2$, we obtain the number of points in P_1 dominated by p_2 using binary search.

Analysis

Let $f(n)$ denote the worst-case cost of the algorithm on n points. In total, there are $\frac{n}{2}$ binary searches performed, each taking $\mathcal{O}(\log n)$ time, giving $\mathcal{O}(n \log n)$ time for this step. Then,

$$f(n) \leq 2f\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n \log n),$$

which gives

$$f(n) = \mathcal{O}(n \log^2 n).$$

2.4 Matrix Multiplication (Strassen's Algorithm)

Given two $n \times n$ matrices A and B, compute their product AB. Assume for simplicity that n is a power of 2. We can divide each of A and B into 4 sub-matrices of order $\frac{n}{2}$.

Suppose we want to compute

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

We need to perform 8 order-$\frac{n}{2}$ matrix multiplications in the most trivial case, i.e.

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

In the trivial case, we need $\mathcal{O}(n^3)$ time.

In a non-trivial case, we have

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}$$

where

$$\begin{aligned} p_1 &= A_{11}(B_{12} - B_{22}) \\ p_2 &= (A_{11} + A_{12})B_{22} \\ p_3 &= (A_{21} + A_{22})B_{11} \\ p_4 &= A_{22}(B_{21} - B_{11}) \\ p_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ p_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ p_7 &= (A_{11} - A_{21})(B_{11} + B_{12}) \end{aligned}$$

If $f(n)$ is the worst-case time of computing the product of two order-n matrices, then each of $p_i, 1 \leq i \leq 7$ can be computed in $f(\frac{n}{2}) + \mathcal{O}(n^2)$ time, where $f(\frac{n}{2})$ is exactly one multiplication between two order-$\frac{n}{2}$ matrices, and $\mathcal{O}(n^2)$ is the cost of matrix additions and subtractions.

Therefore,

$$f(n) \leq 7f\left(\frac{n}{2}\right) + \mathcal{O}(n^2)$$

can be solved to

$$f(n) = \mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.81}).$$

> **Note.** For n being a power of 2, we can recursively split the matrix into sub-matrices, and it still takes $\mathcal{O}(n^{2.81})$. This also works for any n since we can pad with zeros if n is not a power of two. Given that the logic here does not depend on n, the running time remains $\mathcal{O}(n^{2.81})$.

2.5 Fast Fourier Transform

For degree-d polynomials,

$$A(x) = \sum_{i=0}^{d} a_i x^i, \quad B(x) = \sum_{i=0}^{d} b_i x^i,$$

their product $C(x) = A(x)B(x) = \sum_{k=0}^{2d} c_k x^k$ with

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

where $a_i, b_i = 0$ if $i > d$.

To do such multiplication it takes $\Theta(d^2)$. To speed up, we again use divide and conquer.

2.5.1 Point-wise Multiplication

First, we take a look at the representation of a polynomial. Consider two representations of $A(x)$:

1. Coefficients $a_0, a_1, \ldots, a_d$
2. Values at $d+1$ distinct points (point–value pairs):

$$(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_d, A(x_d))$$

Note. A degree-d polynomial is uniquely determined by its values at any $d+1$ distinct points.

If we use point–value representation, given $C(x) = A(x)B(x)$, for any point z we have $C(z) = A(z)B(z)$. Thus, it takes linear time to compute all the values of $C(x)$, i.e. $(2d+1) = \mathcal{O}(d)$ multiplications.

2.5.2 Root of Unity

Before obtaining $C(x)$, we need to evaluate $A(x)$ and $B(x)$. Evaluating a degree-n polynomial at one point costs at least $\mathcal{O}(n)$ time; since there are n points in total, this takes $\Theta(n^2)$. However, consider the following.

A polynomial can be decomposed as (assume n is even):

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\ &= (a_0 + a_2 x^2 + \cdots + a_n x^n) + x(a_1 + a_3 x^2 + \cdots + a_{n-1} x^{n-2}) \\ &= A_{\text{even}}(x^2) + x \cdot A_{\text{odd}}(x^2) \end{aligned}$$

where

$$A_{\text{even}}(x) = a_0 + a_2 x + \cdots + a_n x^{\frac{n}{2}}, \quad A_{\text{odd}}(x) = a_1 + a_3 x + \cdots + a_{n-1} x^{\frac{n}{2}-1}.$$

Then, if we consider paired points $\pm x_i$, we have

$$A(x_i) = A_{\text{even}}(x_i^2) + x_i A_{\text{odd}}(x_i^2), \quad A(-x_i) = A_{\text{even}}(x_i^2) - x_i A_{\text{odd}}(x_i^2).$$

Then the evaluation on points is reduced to the evaluation of $\{x_0^2, \ldots, x_{\frac{n}{2}-1}^2\}$ plus a linear-time combination.

However, this reduction cannot utilize recursion; therefore, we choose another set of points using the concept of complex numbers.

Theorem 2.5.1 (Fundamental Theorem of Algebra)**.** Every polynomial of degree $n \geq 1$ with complex coefficients has exactly n roots in $\mathbb{C}$, counted with multiplicity. Equivalently, for

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0, \quad (a_n \neq 0),$$

there exist $\zeta_1, \ldots, \zeta_n \in \mathbb{C}$ such that

$$p(z) = a_n \prod_{k=1}^{n} (z - \zeta_k),$$

where each root appears according to its multiplicity.

Explanation. Consider our problem: we have a polynomial $C(x)$ that can be determined by $n = 2d+1$ points. According to the theorem, every polynomial of degree $2d$ has exactly $2d$ roots in $\mathbb{C}$. Hence, we can use these complex roots to find a convenient set of points that make evaluation easier. ⊛

Definition 2.5.1. An n-th root of unity, where n is a positive integer, is a complex number z satisfying

$$z^n = 1.$$

There are n distinct n-th roots of unity, given by

$$z_k = \cos\left(\frac{2k\pi}{n}\right) + i \sin\left(\frac{2k\pi}{n}\right), \quad \forall k \in \{0, 1, \ldots, n-1\}.$$

Let

$$\omega_n = \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right),$$

then we have

$$\begin{aligned}
\omega_n^0 &= \cos(0) + i\sin(0) = 1,\\
\omega_n^1 &= \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right),\\
\omega_n^2 &= \cos\left(\frac{4\pi}{n}\right) + i\sin\left(\frac{4\pi}{n}\right),\\
&\vdots\\
\omega_n^{n-1} &= \cos\left(\frac{2(n-1)\pi}{n}\right) + i\sin\left(\frac{2(n-1)\pi}{n}\right).
\end{aligned}$$

Thus, we can write the n-th roots of unity as

$$\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-2}, \omega_n^{n-1}.$$

By Euler's formula,

$$e^{i\theta} = \cos(\theta) + i\sin(\theta),$$

we have

$$\omega_n^k = \cos\left(\frac{2k\pi}{n}\right) + i\sin\left(\frac{2k\pi}{n}\right) = e^{i\frac{2k\pi}{n}}.$$

Moreover, we note some important facts about the roots of unity:

Proposition 2.5.1. For all $n \in \mathbb{N}$, there are exactly n distinct n-th roots of unity, i.e.,

$$U_n = \{\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}\}.$$

Proposition 2.5.2. For all even $n \in \mathbb{N}$, it holds that

$$\omega_n^{k+\frac{n}{2}} = -\omega_n^k, \quad \forall k \in \{0, 1, \ldots, \tfrac{n}{2} - 1\}.$$

Proposition 2.5.3. 3. For all even $n \in \mathbb{N}$, squaring each element of U_n gives the set $U_{\frac{n}{2}}$, i.e., the set of $\frac{n}{2}$-th roots of unity. More explicitly,

$$\omega_n^{2k} = \omega_{\frac{n}{2}}^{k \bmod (\frac{n}{2})}.$$

Proof. Let $\omega_n = e^{\frac{2\pi i}{n}}$ be a primitive n-th root of unity. Consider the square of ω_n^k:

$$(\omega_n^k)^2 = (e^{\frac{2k\pi i}{n}})^2 = e^{\frac{2(2k)\pi i}{n}} = e^{\frac{2k\pi i}{\frac{n}{2}}} = \omega_{\frac{n}{2}}^k.$$

Thus, for each $k = 0, 1, \ldots, n-1$, squaring ω_n^k gives $\omega_{n/2}^k$. ■

Remark.

- Since k ranges over $0, \ldots, n-1$, some values of $\omega_{n/2}^k$ repeat (because k modulo $\frac{n}{2}$).
- Therefore, the set of all $(\omega_n^k)^2$ is exactly

$$\{\omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{n/2-1}\} = U_{n/2},$$

the set of $\frac{n}{2}$-th roots of unity.

2.5.3 Fast Fourier Transform

Now we can look at the details of the Fast Fourier Transform (FFT).

Given a polynomial $A(x)$ of degree at most $n-1$, where $n = 2^m$ for some $m \geq 0$.

If $n = 1$, then $\deg A(x) \leq 0$, and $A(x) = a_0$.

If $n \geq 2$,

1. Split $A(x)$ into its even and odd parts.
2. Make two recursive FFT calls of size $\frac{n}{2}$ to evaluate both $A_{\text{even}}(y)$ and $A_{\text{odd}}(y)$ on all the points in

$$U_{\frac{n}{2}} = \{\omega_{\frac{n}{2}}^0, \omega_{\frac{n}{2}}^1, \ldots, \omega_{\frac{n}{2}}^{\frac{n}{2}-1}\}.$$

3. For all $k \in \{0, 1, \ldots, n-1\}$, compute

$$A(\omega_n^k) = A_{\text{even}}(\omega_n^{2k}) + \omega_n^k A_{\text{odd}}(\omega_n^{2k}),$$

using the values of $A_{\text{even}}(y)$ and $A_{\text{odd}}(y)$ at $\omega_n^{2k} = \omega_{\frac{n}{2}}^{k \bmod (\frac{n}{2})}$.

4. Output the values $(A(\omega_n^k))_{k=0}^{n-1}$.

Then we have the time complexity

$$f(n) = \begin{cases} \mathcal{O}(1), & \text{if } n = 1; \\ 2f\left(\frac{n}{2}\right) + \mathcal{O}(n), & \text{if } n \geq 2. \end{cases}$$

Explanation. Suppose we want to evaluate a polynomial at $n = 8$ points.

1. Split into even and odd parts:

$$A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$$

   This reduces the 8-point evaluation to two 4-point evaluations at x^2.

2. Recursively split each 4-point evaluation into even/odd again, reducing to 2-point evaluations at x^4.

3. Split again until reaching 1-point evaluations (just the coefficients themselves).

4. Combine results on the way back using

$$A(\omega_n^k) = A_{\text{even}}(\omega_{n/2}^k) + \omega_n^k \, A_{\text{odd}}(\omega_{n/2}^k),$$

   which takes $\mathcal{O}(n)$ time per level.

Key idea: At each level, the recursive results are reused for multiple points; only the multiplication by ω_n^k differs. This divide-and-conquer gives total complexity $\mathcal{O}(n \log n)$. ⊛

The last step is to turn the point-value pair representation back to the coefficient representation, which is called interpolation. We can accomplish the interpolation in $\mathcal{O}(n \log n)$ time by using the inverse FFT.

2.5.4 Analysis

We can summarize all the operations here.

First, we select the smallest $n \geq 2d + 1$ where $n = 2^m$, $m \geq 0$.

Then we do the evaluation using FFT for both $A(\omega_n^k)$ and $B(\omega_n^k)$, which takes $\mathcal{O}(n \log n)$.

We then do the point-wise multiplication, i.e. $C(\omega_n^k) = A(\omega_n^k)B(\omega_n^k)$, which takes $\mathcal{O}(n)$.

Last, we do the interpolation using inverse FFT, taking $\mathcal{O}(n \log n)$ time.

Thus, we can finish a degree-n polynomial multiplication in time $\mathcal{O}(n \log n)$.

Chapter 3

Greedy Algorithms

Greedy algorithms enforce a simple strategy, i.e. make the locally optimal decision at each step.

3.1 Activity Selection

Problem 3.1.1. Given a set S of n intervals of the form $[s, f]$ where s, f are integers, we want to output a subset $T \subseteq S$ of disjoint intervals with the largest size $|T|$.

Remark. One can think of $[s, f]$ as the duration of an activity, and consider the problem as picking the largest number of activities that do not have time conflicts.

Greedy Algorithm

The idea is to pick the earliest-finishing interval, keep it, and discard all intervals that overlap it. Repeat until none remain.

Repeat until S becomes empty:

1. Add to T the interval $\mathcal{I} \in S$ with the smallest finish time.
2. Remove from S all intervals intersecting $\mathcal{I}$ (including $\mathcal{I}$).

The greedy algorithm picks the earliest finishing activity, leaving the most room for future choices.

Proof. Let $G = \{g_1, g_2, \ldots, g_k\}$ be the greedy solution, and $O = \{o_1, o_2, \ldots, o_m\}$ be an arbitrary optimal solution, where each g_i, o_i is an interval.

Since there is no overlap, it holds that for each i, $f_{o_{i-1}} \leq s_{o_i}$. It must be $k \leq m$; otherwise, O would not be an optimal solution.

Assume g_1 is the earliest finishing interval, as well as o_1, which might be different. Then we have $f_{g_1} \leq f_{o_1}$.

We can replace o_1 with g_1 in O. Since g_1 finishes no later than o_1, intervals in O remain disjoint.

Repeating this process, assume $k < m$. The replacement stops at

$$O = \{g_1, g_2, \ldots, g_k, o_{k+1}, o_{k+2}, \ldots, o_m\}.$$

Note that the finish time of g_k is earlier than the start time of all intervals $o_{k+1}, o_{k+2}, \ldots, o_m$, where disjointness still holds. Then the greedy algorithm would not stop at g_k, because there are still intervals remaining in the original set S. This leads to a contradiction. Thus, it must be $k = m$, i.e. the greedy algorithm gives an optimal solution. ■

Analysis

This algorithm can be implemented in $\mathcal{O}(n \log n)$ time.

3.2 Minimum Spanning Trees

Consider $G = (V, E)$ an undirected graph, where w is a function that maps each edge e of G to a positive integer value $w(e)$, which we call the weight of e.

An undirected weighted graph is defined as a pair (G, w). Assume G is connected, i.e. every pair of vertices has a path between them.

A tree is defined as a connected undirected acyclic graph.

Definition 3.2.1 (Spanning Tree). A spanning tree T of a connected undirected weighted graph $G = (V, E)$ is a tree that spans G with vertex set $V_T = V$ and edge set $E_T \subseteq E$.

Problem 3.2.1 (Minimum Spanning Tree). Given a connected undirected weighted graph (G, w) with $G = (V, E)$, the goal of the minimum spanning tree (MST) problem is to find a spanning tree of the smallest cost.

Note. MSTs may not be unique.

3.2.1 Prim's Algorithm

Choose an arbitrary vertex as the starting point. This algorithm grows a tree T by including one vertex at a time. At any moment, it divides the vertex set V into:

1. the set S of vertices that are already in T;
2. the set of other vertices $V \setminus S$.

If an edge connects a vertex in S and a vertex in $V \setminus S$, we call it a cross edge.

Greedy Algorithm

The idea is to repeatedly take the lightest cross edge. Check CSCI3100: Prim's Algorithm.

Proposition 3.2.1. If $T = (E_T, V_T)$ is a tree and an edge $e \notin E_T$ connects two vertices in V_T, then the following graph

$$T' \coloneqq (E_T \cup \{e\}, V_T)$$

contains exactly one cycle.

Proof. Since T is connected, there is already a unique path between the endpoints of e. Adding e creates a cycle by joining the ends of this path. As trees are acyclic, this must be the only cycle. ■

Proof. By definition, T is a tree, so it is connected. Thus, there is a unique path in T such that

$$P_{u \to v} = u \to \cdots \to v$$

and adding an edge $e = (u, v)$ forms a closed loop. Hence, T' contains at least one cycle.

Suppose there exists another cycle C' in T'. Then C' must involve at least one edge not in T since T is acyclic. The only edge not in T is e; therefore, C' must also contain e. Removing e from C' gives a path between u and v which is different from $P_{u \to v}$, yet T has only one unique path between u and v, thus a contradiction. ■

Proposition 3.2.2. Let C be a set of edges that form a cycle in a connected graph. Removing any edge e from C keeps the graph connected.

Proof. Since the removed edge was part of a cycle, there remains an alternative path between its endpoints. Thus, all vertices remain reachable from one another. ■

Proposition 3.2.3. Let $G = (V, E)$ be an undirected graph, and let T be a sub-graph of G. Then the following statements are equivalent:

1. T is a spanning tree of G,
2. T is connected and has exactly $|V| - 1$ edges.

Proof. Assume T is a spanning tree of G. Then by definition, T is connected, acyclic, and spans all vertices in V. Since any tree on $|V|$ vertices has exactly $|V| - 1$ edges, T is connected and has exactly $|V| - 1$ edges.

Conversely, assume T is connected and has exactly $|V| - 1$ edges. T must be acyclic; otherwise, removing one edge from the cycle would not break connectivity. However, it is impossible to have a connected graph of $|V|$ vertices with fewer than $|V| - 1$ edges. T must also span all vertices; otherwise, T can contain at most $|V| - 1$ vertices. Since T has exactly $|V| - 1$ edges, these conditions imply that T would contain a cycle, contradicting the acyclic property above. ■

Prim's Algorithm. Let $G = (V, E)$ be a connected, undirected graph with edge weights. Let T be the tree built by Prim's algorithm, $T^\star$ any MST of G, E_k the first k edges selected by Prim, and $S_k \subset V$ the set of vertices included so far.

We maintain the invariant that in the k-th step, the tree formed by E_k is a subtree of some MST.

Initially, $E_0 = \varnothing$, and any MST contains E_0. Thus, the invariant holds for the base case.

Assume that after k steps the invariant is true. Let $e = (u, v)$ be the next edge chosen by Prim, where $u \in S_k, v \in V \setminus S_k$. Edge e is the minimum-weight edge crossing the cut.

Case 1: $e \in T^\star \Rightarrow E_{k+1}$ is a subtree of $T^\star$, so the invariant holds.

Case 2: $e \notin T^\star$. Since $T^\star$ is a spanning tree, there is a unique path in $T^\star$ between u and v. This path must contain some edge f crossing the cut. Define

$$T' \coloneqq T^\star + e - f,$$

i.e., T' is obtained by adding edge e to $T^\star$ and removing edge f.

It is clear that $E_{k+1} = E_k \cup \{e\}$ is a sub-graph of T'. By Proposition 3.2.1, $T^\star + e$ contains a cycle; by Proposition 3.2.2, T' is connected. Since $T^\star$ is an MST and has exactly $|V| - 1$ edges (Proposition 3.2.3), T' also has $|V| - 1$ edges, so by Proposition 3.2.3 it is a spanning tree.

As e is the lightest edge across the cut, $w(e) \leq w(f)$, so $w(T') \leq w(T^\star)$. Hence, T' is an MST.

After $|V| - 1$ steps, Prim has selected $|V| - 1$ edges. By induction, this set of edges P forms a subtree of some MST, denoted T_{final}. By Proposition 3.2.3, T_{final} has exactly $|V| - 1$ edges, so $P = T_{\text{final}}$. ■

Analysis

This algorithm can be implemented in $\mathcal{O}((|V| + |E|) \cdot \log |V|)$ time.

3.3 Huffman Codes

Consider an alphabet Σ. An **encoding** is a function that maps each letter in Σ to a binary string, where such a string is called a **code-word**.

Given that not all symbols occur with equal frequency, we can assign shorter code-words to more frequent symbols in order to reduce the **average number of bits per letter**. However, we must enforce a constraint that no letter's code-word is a prefix of another letter's code-word. An encoding satisfying this constraint is called a **prefix code**.

For each letter $\sigma \in \Sigma$, let $\mathit{freq}(\sigma)$ denote the frequency of σ, and let $\mathit{len}(\sigma)$ denote the number of bits in the code-word of σ. Then, given an encoding, its average length is

$$\sum_{\sigma \in \Sigma} \mathit{freq}(\sigma) \cdot \mathit{len}(\sigma).$$

Problem 3.3.1 (The Prefix Code Problem). Given an alphabet Σ and the frequency of each letter in it, find a prefix code for Σ with the smallest possible average length.

We can utilize a binary tree to represent prefix codes.

Definition 3.3.1 (Code Tree). A **code tree** on Σ is a binary tree T such that every leaf node corresponds to a unique letter in Σ, and every letter in Σ corresponds to a unique leaf node in T. For every internal node, its left edge is labeled 0 and its right edge is labeled 1.

Then we can generate the prefix code for a letter $\sigma \in \Sigma$ from the tree T by concatenating the bit labels of the edges on the path from the root to σ.

Lemma 3.3.1. Every prefix code can be generated by a code tree.

Since the code word length corresponds to the level of the node in T, to solve the problem we can simply find a tree T that minimizes the average height.

Greedy Algorithm

Let $n = |\Sigma|$. Create a set S of n stand-alone leaf nodes, each corresponding to a distinct letter in Σ. Then repeat the following steps until $|S| = 1$:

1. Remove from S two nodes u_1 and u_2 with the smallest frequencies.

2. Create a new node v with u_1 and u_2 as its children, and set the frequency of v to be the sum of the frequencies of u_1 and u_2.

3. Add v back into S.

Here the prefix code derived from the tree is called **Huffman code**.

Lemma 3.3.2. In an optimal code tree, every internal node of T must have two children.

Proof. Assume for contradiction, T is optimal but contains an internal node v with only one child.

Let the single child-subtree of v contain the leaf set L and

$$W = \sum_{x \in L} p_x$$

be the total weight of those leaves. Since all $p_x > 0$, we have $W > 0$.

Now form a new tree T' by bypassing v, i.e., remove v and attach its only child in v's place. Then all leaves in L move up one level, so each such leaf's depth decreases by exactly 1.

Let $C(T)$ denote the expected cost of tree T,

$$C(T) = \sum_{\text{leaves } x} p_x \cdot d_T(x)$$

Then the new cost will be

$$C'(T) = C(T) - \sum_{x \in L} p_x = C(T) - W$$

Because $W > 0$, we have $C'(T) < C(T)$, leading to contradiction. ■

Lemma 3.3.3. Let σ_1, σ_2 be two letters in Σ with the lowest frequencies. There exists an optimal code tree where σ_1 and σ_2 have the same parent.

Proof. Without loss of generality, assume $freq(\sigma_1) \leq freq(\sigma_2)$. Let T be any optimal code tree. Let p be an arbitrary internal node with the largest level in T. By Lemma 3.3.2, p must have two leaves. Let x and y be letters corresponding to those leaves such that $freq(x) \leq freq(y)$. Swap σ_1 with x and σ_2 with y, which gives a new code tree T'. Note that both σ_1 and σ_2 are children of p in T'. ■

Theorem 3.3.1. Huffman code produces an optimal prefix code.

Proof. Consider σ with a size of n.

For $n = 2$, Huffman algorithm will encode one letter with 0 and the other letter with 1, which is optimal.

Assume the theorem is correct for $n = k - 1$ where $k \geq 3$.

Consider the following:

Σ: an alphabet of size k, where σ_1 and σ_2 are two letters with the lowest frequencies.

T: an optimal code tree on Σ, where leaves σ_1 and σ_2 have the same parent p.

T_{huff}: output of Huffman's algorithm on Σ.

And for $n = k - 1$, we have

Σ': an alphabet constructed from Σ by removing σ_1 and σ_2 and adding a letter $\sigma^\star$ with frequency $freq(\sigma_1) + freq(\sigma_2)$.

T': the tree obtained by removing leaves σ_1 and σ_2 from T.

T'_{huff}: output of Huffman's algorithm on Σ'.

Then we have

$$\text{average height of } T_{huff} = \text{average height of } T'_{huff} + freq(\sigma_1) + freq(\sigma_2),$$

$$\text{average height of } T = \text{average height of } T' + freq(\sigma_1) + freq(\sigma_2),$$

$$\text{average height of } T_{huff} \leq \text{average height of } T',$$

which give

$$\text{average height of } T_{huff} \leq \text{average height of } T$$

■

Analysis We can implement this algorithm in $\mathcal{O}(n \log n)$ time.

Chapter 4

Dynamic Programming

4.1 Pitfall of Recursion

Let A be an array of n positive integers, with function

$$f(k) = \begin{cases} 0, & \text{if } k = 0 \\ \max\limits_{i=1}^{k}(A[i] + f(k-i)), & \text{if } 1 \le k \le n \end{cases}$$

To compute $f(n)$, we can utilize the following:

Algorithm 4.1: $f(n)$

```
1 if k = 0 then
2     return 0
3 ans ← −∞
4 for i ← 1 to k do
5     tmp ← A[i] + f(k − i)
6     if tmp > ans then
7         ans ← tmp
8 return ans
```

This recursion gives $\Omega(2^n)$ because it computes $f(x)$ for the same x repeatedly. A recursive algorithm does considerable redundant work if the same sub-problem is encountered over and over again.

We resolve such by dynamic programming, where we resolve sub-problems according to a certain order, remembering the output of every sub-problem to avoid re-computation.

For the same problem, we consider solving the problem with order $f(1), f(2), \cdots, f(n)$. Then we need $\mathcal{O}(n^2)$ time in total.

Algorithm 4.2: Dynamic programming for $f(n)$

```
1 Initialize: ans[n], ans[0] ← 0
2 for k ← 1 to n do
3     ans[k] ← −∞
4     for i ← 1 to k do
5         tmp ← A[i] + ans[k − i]
6         if tmp > ans[k] then
7             ans[k] ← tmp
```

In the following, we will see examples of dynamic programming, where the core idea is to break the problem into sub-problems that can be solved in a **nice order**, so that later sub-problems can reuse the solutions of earlier ones.

The key to solving a problem is identifying its recursive structure — how the original problem relates to its sub-problems — which then guides the design of a dynamic programming algorithm.

4.2 Rod Cutting

Problem 4.2.1. Consider a rod of length n with an array P of length n where $P[i]$ is the price for a rod of length $i, i \in [1, n]$. How to cut the rod into segments of integer lengths to maximize the revenue?

Example. Consider the price array P

length i	1	2	3	4
price $P[i]$	1	5	8	9

There are 8 ways to cut the rod of length 4:

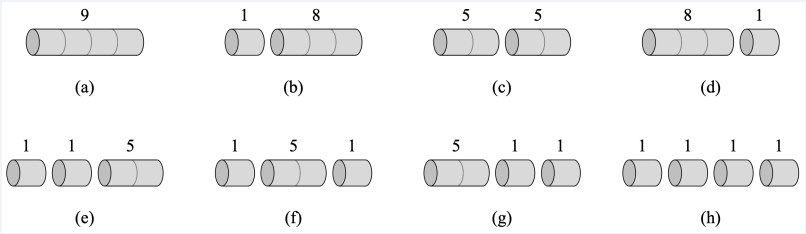


Figure 4.1: Rod Cutting

The optimal cutting method is (c), which has a revenue of 10.

Intuition. No matter how we cut, we must make a first cut, but we do not know which first cut is optimal. Therefore, we can consider all possible first cuts, then choose the one that maximizes revenue. For each choice of the first cut, we cut the remaining rod optimally and select the maximum revenue among all possible cutting strategies.

Let $\text{opt}(n)$ denote the optimal revenue from cutting a rod of length n. For $n \geq 1$, conditioned on choosing a first segment of length i, the revenue is

$$P[i] + \text{opt}(n - i),$$

and this holds for any choice of first cut. Since there are n possible first cuts, we have

$$\text{opt}(n) = \max_{i=1}^{n} \big(P[i] + \text{opt}(n - i)\big).$$

Using dynamic programming, $\text{opt}(n)$ can be computed in $\mathcal{O}(n^2)$ time[1].

If we want not only the optimal revenue but also the corresponding cutting strategy, we can record which sub-problem yields $\text{opt}(n)$. Define

$$\text{bestSub}(n) = k \quad \text{if the optimal revenue is obtained by making the first cut at length } k.$$

After computing all $\text{bestSub}(i)$ for $i \in [1, n]$, we can reconstruct the optimal cutting method using another array in $\mathcal{O}(n)$ time. Therefore, the overall complexity of the rod cutting problem remains $\mathcal{O}(n^2)$.

[1] The algorithm used can be found in 4.2

Note. The technique of using the bestSub function to reconstruct an optimal solution is known as the *piggyback technique*.

4.3 Longest Increasing Subsequences

Problem 4.3.1. Given an array of integers $A[1 \dots n]$, how can we find the length of the **longest strictly increasing subsequence** (LIS)?

Example. Consider the array

$$A = [10, 9, 2, 5, 3, 7, 101, 18].$$

Then we have LIS $= [2, 3, 7, 101]$ with length 4.

Remark. The elements in the LIS do not need to be contiguous in the original array A; there may be gaps between them. The only requirement is that they appear in strictly increasing order.

Using a brute-force method, i.e., trying all possible subsequences, requires $\mathcal{O}(2^n)$ time.

Intuition. The idea is similar to the rod-cutting problem: we try to determine the best last element to place in the LIS, then reduce the problem to a sub-problem. If we choose an element as the last element of an increasing subsequence, the optimal solution can be found by examining all valid previous elements. This approach works nicely with an appropriate ordering.

Let $\text{dp}[i]$ be the length of the LIS ending at index i. Using the recurrence relation, we have

$$\text{dp}[i] = 1 + \max\{\text{dp}[j] : j < i \text{ and } A[j] < A[i]\}, \qquad \text{and if no such } j \text{ exists, then } \text{dp}[i] = 1.$$

Let $\text{len}_{\text{LIS}}(i)$ denote the length of the LIS that **ends at** $A[i]$. Then

$$\text{len}_{\text{LIS}}(i) = 1 + \max_{j<i \text{ and } A[j]<A[i]} \{\text{len}_{\text{LIS}}(j)\},$$

where we append $A[i]$ to the LIS that ends at $A[j]$ only if $A[j] < A[i]$. After computing $\text{len}_{\text{LIS}}(i)$ for each $i \in \{1, 2, \dots, n\}$, the LIS length is

$$\max_{i \in \{1,2,\dots,n\}} \{\text{len}_{\text{LIS}}(i)\}.$$

Algorithm 4.3: Longest Increasing Subsequence (LIS)

Input: Array $A[1 \dots n]$
Output: Length of the Longest Increasing Subsequence

```
Initialize: dp[1...n] = 1
for i ← 1 to n do
    for j ← 1 to i − 1 do
        if A[j] < A[i] then
            dp[i] ← max(dp[i], dp[j] + 1)
return max(dp[1...n])
```

This algorithm runs in $\mathcal{O}(n^2)$ time. As before, we can use the piggyback technique to recover the actual longest increasing subsequence by maintaining an extra array to record the predecessors of each element in the LIS.

4.4 Edit Distance

Problem 4.4.1. Given two strings

$$A = a_1 a_2 \cdots a_m \quad \text{and} \quad B = b_1 b_2 \cdots b_n,$$

we allow three operations: **insertion**, **deletion**, and **replacement**. The **Edit Distance** is defined as the minimum number of such operations required to transform A into B. How can we compute the edit distance?

Example. Consider $A =$ `kitten` and $B =$ `sitting`:

1. `kitten` $\rightarrow$ `sitten` (replace 'k' with 's')
2. `sitten` $\rightarrow$ `sittin` (replace 'e' with 'i')
3. `sittin` $\rightarrow$ `sitting` (insert 'g' at the end)

Therefore, the edit distance is 3.

Intuition. If we can find the best way to align two strings, i.e., maximize the overlap of characters, then we can minimize the number of edit operations.

Consider the endpoints of the alignment in both strings. Let the last characters be a_m in A and b_n in B, and write

$$A = \underbrace{a_1 a_2 \cdots a_{m-1}}_{\alpha} a_m, \quad B = \underbrace{b_1 b_2 \cdots b_{n-1}}_{\beta} b_n.$$

We have three possible alignments for the last characters:

$A =$	αa_m	
$B =$	β	b_n

Table 4.1: Case 1

$A =$	α	a_m
$B =$	βb_n	

Table 4.2: Case 2

$A =$	α	a_m
$B =$	β	b_n

Table 4.3: Case 3

Denote the edit distance between two strings by $D(\text{str}_1, \text{str}_2)$. Then we can see that

$$D(A,B) = \begin{cases} D(\alpha a_m, \beta) + 1, & \text{if case 1 happens,} \\ D(\alpha, \beta b_n) + 1, & \text{if case 2 happens,} \\ D(\alpha, \beta) + 1, & \text{if case 3 happens and } a_m \neq b_n, \\ D(\alpha, \beta), & \text{if case 3 happens and } a_m = b_n. \end{cases},$$

which can be rewritten as

$$D(A,B) = \min\left\{D(\alpha a_m, \beta) + 1,\ D(\alpha, \beta b_n) + 1,\ D(\alpha,\beta) + [a_m == b_n]\right\}, a_m == b_n \coloneqq \begin{cases} 1, & \text{if } a_m \neq b_n, \\ 0, & \text{if } a_m = b_n. \end{cases}.$$

Consider the sub-problem. Let

$$A[1\ldots i] = a_1 a_2 \cdots a_i \quad \text{and} \quad B[1\ldots j] = b_1 b_2 \cdots b_j,$$

then we have

$A =$	$a_1 \ldots a_{i-1} a_i$	
$B =$	$b_1 \ldots b_{j-1}$	b_j

Table 4.4: Case 1

$A =$	$a_1 \ldots a_{i-1}$	a_i
$B =$	$b_1 \ldots b_{j-1} b_j$	

Table 4.5: Case 2

$A =$	$a_1 \ldots a_{i-1}$	a_i
$B =$	$b_1 \ldots b_{j-1}$	b_j

Table 4.6: Case 3

$$D(i,j) = \min\left\{D(i,j-1)+1, D(i-1,j)+1, D(i-1,j-1)+[a_i == b_j]\right\}.$$

Consider also the boundary case, where $D(i,0) = i$ is to delete all chars from A and $D(0,j) = j$ is to insert all chars of B.

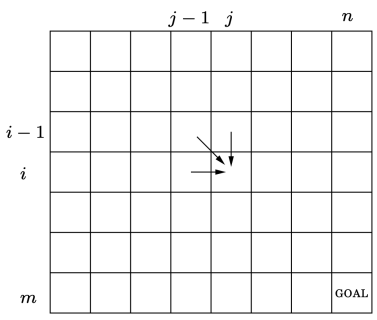
Then, for all $0 \le i \le m$ and $0 \le j \le n$, we have

$$D(i,j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min\{D(i,j-1)+1, D(i-1,j)+1, D(i-1,j-1)+[a_i == b_j]\}, & \text{if } i,j \neq 0 \end{cases}$$

The edit distance between A and B is given by the value $D(m,n)$.

To solve the problem $D(i,j)$, we only need to know the solution to three sub-problems $D(i-1,j)$, $D(i,j-1)$ and $D(i-1,j-1)$. Therefore, we can utilize the **dynamic programming table**.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Figure 4.2: Dynamic Programming Table

Explanation. We first fill all the cells in the first row and column, which is simple insertion and deletion. Consider $D(i,j-1)$ as deletion (or insertion), same for $D(i-1,j)$, and $D(i-1,j-1)$ is to find if the current character is the same. If not the same, then we find the minimum of these three and add 1, else if the char is the same, we simply use the value in $[i-1,j-1]$. ⊛

By filling the table in proper order, we can solve the problem in $\mathcal{O}(mn)$ time.

Algorithm 4.4: Edit Distance

Input: Strings $A[1\ldots m], B[1\ldots n]$

Initialize:
$D[0][j] \leftarrow B[j]$ for $j = 0\ldots n$
$D[i][0] \leftarrow A[i]$ for $i = 0\ldots m$
for $i \leftarrow 1$ **to** m **do**
  for $j \leftarrow 1$ **to** n **do**
    if $A[i] = B[j]$ **then**
      $D[i][j] \leftarrow D[i-1][j-1]$
    else
      $$D[i][j] \leftarrow 1 + \min \begin{cases} D[i-1][j] & // \text{ delete} \\ D[i][j-1] & // \text{ insert} \\ D[i-1][j-1] & // \text{ replace} \end{cases}$$

We can again use piggyback to retrieve the edit sequence.

4.5 Optimal BST

Chapter 5

Graph

5.1 White Path Theorem

5.2 Single Source Shortest Paths

5.2.1 SSSP with Arbitrary Weights

5.3 All-Pairs Shortest Paths

Chapter 6

Approximation Algorithms

Appendix A

Master Theorem

We can use the Master Theorem to solve recurrence problems.

Consider

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \quad a \geq 1, \quad b > 1,$$

where

$$f(n) = \mathcal{O}(n^k \log^p n).$$

Case 1: if $\log_b a > k$, then

$$\mathcal{O}(n^{\log_b a}).$$

Case 2: if $\log_b a = k$, then

- if $p > -1$,

$$\mathcal{O}(n^k \log^{p+1} n),$$

- if $p = -1$,

$$\mathcal{O}(n^k \log \log n),$$

- if $p < -1$,

$$\mathcal{O}(n^k).$$

Case 3: if $\log_b a < k$, then

- if $p \geq 0$,

$$\mathcal{O}(n^k \log^p n),$$

- if $p < 0$,

$$\mathcal{O}(n^k).$$