CMSC351

Hari

Contents

Chapter 1

Introduction

1.1 Introduction

Definition 1 (Algorithm)**.** An algorithm is a finite list of step by step instruction for solving a problem.

We typically think of algorithms in terms of optimizing the time and space for efficiency.

Example (Algorithm)**.** Tournament assignment.

Example (Sorting Algorithms)**.** Two sorting algorithms:

- Slow algorithm (bubble sort): $4n^2$ instructions
- Fast algorithm (merge sort): $80n \log n$ instructions

Two computers:

- 10 billion instructions / second
- 10 million instructions / second

Time to sort 10 million numbers:

- Slow algorithm, fast computer: ≈ 11 hours
- Fast algorithm, slow computer: $\approx \frac{1}{2}$ hours

Example (Comparisons)**.** Say we are sorting around 10^7 numbers. To calculate this we know there will be $4 \cdot \left(10^7\right)^2$ instructions. Dividing by the speed of the computer, we have

$$\frac{4 \cdot \left(10^7\right)^2}{10^{10}}$$

For the second one, we have

$$\frac{80 \cdot 10^7 \cdot \log 10^7}{10^7}$$

Note. This course is meant to teach the basics of algorithms. It is an analogy to (for example) discrete math, where we learn how to write proofs. It is hard to "write an algorithm to do X" just like it is hard to "prove X".

Example (Minimum Coin Count). This is an example of dynamic programming. Let us say we have coins with some worth, for example $C = \{1, 10, 25\}$. So in this case we have quarters, dimes, and pennies. How can we minimize this?

One first thought might be to select **greedily**, namely taking the maximum possible. This works for say $35 = 25 + 10$, but notice that this does not work in the case of 30. If we greedily select, we will get $25 + 1 + 1 + 1 + 1 + 1$ which is 6 coins total. However, we can do it in 3 with $30 = 10 + 10 + 10$.

However, lets say we know the minimum coins necessary for x cents, call this $A[x]$ If we know the values of $A[0]$ to $A[x-1]$, is there an easy way to find $A[x]$? Yes!

- If we select the 1 cent coin, then we get $A[x-1] + 1$
- If we select the 10 cent coin, then we get $A[x-5] + 1$ (when $x \geq 10$)
- If we select the 25 cent coin, then we get $A[x-25] + 1$ (when $x \geq 25$)

So then we just take the minimum of these three. And finally we arrive at the algorithm:

Algorithm 1 Min Coins

```
Require: $C$ = list of coin denominations
 1: procedure Min Coins
 2:     $A \leftarrow$ empty list which can grow
 3:     $A[0] \leftarrow 0$
 4:     for $i = 1, \ldots, x$ do
 5:         $n \leftarrow \infty$
 6:         for all $c \in C$ do
 7:             if $i - c \geq 0$ then
 8:                 $n \leftarrow \min(n, A[i-c] + 1)$
 9:             end if
10:         end for
11:         $A[i] \leftarrow n$
12:     end for
13:     return $A[n]$
14: end procedure
```

Example (Number of ways to coin count). Suppose we want to solve the problem of how many ways we can obtain n cents. First, note that with for two coins, we have that

$$\begin{aligned}\#\text{ of ways to get } n \text{ using } c_1 \text{ and/or } c_2 &= \#\text{ of ways to get } n \text{ using } c_1 \\ &+ \#\text{ of ways to get } n - c_2 \text{ using } c_1 \text{ and/or } c_2\end{aligned}$$

Notice that if we can either use 0 of c_2 or 1 or more of c_2. This generalizes for the r-coin case as well.

$$\begin{aligned}\#\text{ of ways to get } n \text{ using } c_1, \ldots, c_r &= \#\text{ of ways to get } n \text{ using } c_1, \ldots, c_{r-1} \\ &+ \#\text{ of ways to get } n - c_r \text{ using } c_1, \ldots, c_r\end{aligned}$$

Note that this is only true if $n \geq c_r$.

Let us have an array A such that $A[0] \ldots A[k]$ contains the number of ways to obtain 0 to k cents using all r coins. However, let $A[k+1] \ldots A[n]$ contain the number of ways to obtain 0 through k cents using the first $r-1$ coins. To update $A[k+1]$ to use all r coins, we see that if $n \geq c_r$,

$$A[k+1] = A[k+1] + A[k+1-c_r]$$

Then the algorithm is simple from there.

Algorithm 2 Coin Count

```
Require: $C$ = list of coin denominations
function COINCOUNT(C, n)
    $A \leftarrow [1, 0, 0, \dots 0]$
    for $c \in C$ do
        for $i = 1 \dots n$ do
            if $i \geq c$ then
                $A[i] \leftarrow A[i] + A[i - c]$
            end if
        end for
    end for
    return $A[n]$
end function
```

Chapter 2

Order Notation and Sorting

2.1 Order Notation

Definition 2 (Big O Notation)**.** We state that $f(x) = \mathcal{O}(g(x))$ if

$$\exists x_0, C > 0 \text{ such that } \forall x \geq x_0,\ f(x) \leq Cg(x)$$

We think of this as saying that *eventually* $f(x)$ is smaller than some constant multiple of $g(x)$.

Example (Big O)**.** Note that $42000x \log x = \mathcal{O}(x^2)$ with
$C = 10$ because eventually

$$42000x \log x \leq 10x^2$$

. However, eventually in this case means $x_0 \approx 67367$.

Definition 3 (Big Omega)**.** We have $f(x) = \Omega(g(x))$ if

$$\exists x_0, C > 0 \text{ such that } \forall x \geq x_0,\ f(x) \geq Cg(x)$$

Definition 4 (Big Theta)**.** We have $f(x) = \Theta(g(x))$ if

$$\exists x_0, C_1 > 0, C_2 > 0 \text{ such that } \forall x \geq x_0, C_1 g(x) \leq f(x) \leq C_2 g(x)$$

Remark. You must write order notation in simplest form for exams.

Remark. When we write say $\Theta(n^2)$, this actually forms a set. This set contains all functions which are bounded above and below asymptotically by n^2. For example, we can write the set of all functions $2^{\Theta(n^2)}$. Please note this is different from the set of functions $\Theta(2^{n^2})$.

The basic idea is that $\mathcal{O}$ provides an upper bound, Ω provides a lower bound, and Θ provides a tight bound.

Remark. When most people talk informally about Big O, typically it refers to Θ instead.

Theorem 1 (Equivalence)**.** $f(x) = \Theta(g(x))$ if and only if $f(x) = \mathcal{O}(g(x))$ and $f(x) = \Omega(g(x))$.

Example (All together)**.** We show $3x \lg x + 17 = \mathcal{O}(x^2)$. Consider the expression $3x \lg x + 17$. Note two things,

- If $x > 0$, $\lg x < x$
- If $x \geq \sqrt{17}$, $x^2 \geq 17$

Therefore, if $x \geq 5$, both these statements are true, and we have

$$\begin{aligned} 3x \lg x + 17 &\leq 3 \cdot x \cdot x + 17 \\ &\leq 3x^2 + x^2 \\ &\leq 4x^2 \end{aligned}$$

Therefore, since $x_0 = 5$ and $C = 4$, we have that

$$3x \lg x + 17 \in \mathcal{O}(x^2)$$

Example (Shenanigans)**.** Consider the function $f(x) = 0.001x^2(1 + \cos(x\pi))$.

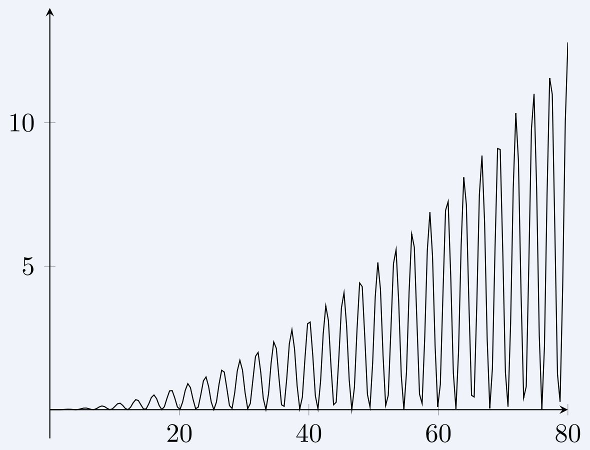


Figure 2.1: Graph of f(x)

First we note that $0.001x^2(1+\cos(x\pi)) \leq 0.001x^2(1+1) = 0.002x^2$ for $x \geq 0$. Therefore, $f(x) \in \mathcal{O}(x^2)$. However, note that $0.001x^2(1+\cos(x\pi)) = 0.001x^2(1-1) = 0$ when x is odd. Therefore, $f(x) \notin \Omega(x^2)$, so $f(x) \notin \Theta(x^2)$.

2.2 Limits and Big O

There are a few alternative ways of proving $\mathcal{O}, \Omega$, and Θ. Note that these are unidirectional, and the converse is not necessarily true.

Theorem 2 (Limit Theorem)**.** Provided that $\lim\limits_{n\to\infty} f(n)$ and $\lim\limits_{n\to\infty} g(n)$ exist (they may be ∞), then we have the following:

(a) If $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$, then $f(n) = \mathcal{O}(g(n))$

(b) If $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$, then $f(n) = \Omega(g(n))$

Proof 1. Proof of (a) only. Suppose we have

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L \neq \infty$$

By definition of the limit, this means

$$\forall \epsilon > 0, \exists n_0 \text{ such that } n \geq n_0 \implies L - \epsilon < \frac{f(n)}{g(n)} < L + \epsilon$$

Specifically, if $\epsilon = 1$, then we get

$$\exists n_0 \text{ such that } n \geq n_0 \implies \frac{f(n)}{g(n)} < L + 1$$

So when $n \geq n_0$, we have

$$f(n) < (L+1)g(n)$$

so

$$f(n) \leq$$

$\square$

Example (Limit)**.** We have $n^2 = \mathcal{O}(3^n)$ Note that we can apply L'hopital's Rule:

$$\lim_{n\to\infty} \frac{n^2}{3^n} = \lim_{n\to\infty} \frac{2n}{\ln 3 \cdot 3^n} = \lim_{n\to\infty} \frac{2}{\ln 3 \cdot \ln 3 \cdot 3^n} = 0$$

Since $0 \neq \infty$, we have that $n^2 = \mathcal{O}(3^n)$.

2.3 Common Functions

Computer scientists (and our exams), almost always expect our Big O results to be in terms of simple functions. For example, we would never say that $f(n) = \mathcal{O}(n^2+1)$, even if it may be true, we typically state $f(n) = \mathcal{O}(n^2)$.

The following is a list of some standard functions in increasing size,

$$1, \lg n, n, n \lg n, n^2, n^2 \lg n, n^3, \ldots$$

Note that each of these is $\mathcal{O}$ of anything to the right. For example, $n = \mathcal{O}(n \lg n)$ and $n^2 = \mathcal{O}(n^3)$. Note that

$$n^k = \mathcal{O}(n^k \lg n)$$

and

$$n^k \lg n = \mathcal{O}(n^{k+1})$$

In addition, we have that

$$n^k = \mathcal{O}(b^n)$$

Lastly, above everything else is $\mathcal{O}(n!)$.

Remark. So how do we compare the size of two functions? For example, $n^{\log n}$ and $(\log n)^n$. There are two ways to do this, either taking a limit of the ratio and use LHopital's rule, or take the log of both sides.

In this case, we can take the log of both sides. Then we get $\log n \log n = (\log n)^2$ versus $n \log\log n$. The first one in this case is **polylog**, which grows slower than polynomials (including first degree polynomials).

Note. In essence, the "largest term" always wins in a Θ sense. This can be proven rigorously.

Note. It's tempting to think that if $f(x)$ and $g(x)$ are both positive functions defined on $[0, \infty)$ with positive derivatives and if $f'(x) > g'(x)$ for all x, then eventually $f(x) > g(x)$. This is not true.

To see why, let us say both f and g approach the same asymptote. We can see that the difference of the two functions $f - g$ will approach zero (as $f' > g'$). However, if they both approach the same asymptote, f' will never actually pass g'. A nice example of this is

$$f(x) = 1 - \frac{1}{(x+1)^2}$$

$$g(x) = 1 - \frac{1}{e^{x+1}}$$

Note. Note that $5^n \neq \mathcal{O}(2^n)$.

Note. Note $\log_2 n = \Theta(\log_5 n)$ and $\log_5 n = \Theta(\log_2 n)$. This generalizes, where when $b, c > 1$ (why do we need this?), $\Theta(\log_b x) = \Theta(\log_c x)$.

We can see intuitively that this is true since by log base rules when $b, c > 1$, we can re express the function as a constant multiple of $\ln x$. However, when $b, c < 1$, the power is inverted so the function decreases instead of increasing.

Note. We can have two functions $f(x)$ and $g(x)$ which are not constant multiples of one another and which satisfy $f(x) = \mathcal{O}(g(x))$ and $g(x) = \mathcal{O}(f(x))$. For example, $f(x) = 4x^3$ and $g(x) = 4x^3 + 3$.

Note. Note that $\log_b x = \mathcal{O}(x^c)$ when $0 < c < 1$. Roots grow faster than logarithms.

2.4 Related Asymptotic Notions

Definition 5 (Little o)**.** We say $f(x) = o(g(x))$ if

$$\exists x_0, C > 0 \text{ such that } \forall x \geq x_0, f(x) > Cg(x)$$

Definition 6 (Little omega)**.** We say $f(x) = \omega(g(x))$ if

$$\exists x_0, C > 0 \text{ such that } \forall x \geq x_0, f(x) < Cg(x)$$

Theorem 3 (Related Limit Theorems)**.** Provided that $\lim_{n\to\infty} f(n)$ and $\lim_{n\to\infty} g(n)$ exist, (they may be ∞),

(a) If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, we say $f(n) = o(g(n))$

(b) If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, we say $f(n) = \omega(g(n))$

Chapter 3

Rigorous Time

3.1 Rigorous Time

Algorithms take time. Running code takes time.

Formally speaking, each assignment takes its own time.

Algorithm 3 assign

```
1: a = 0                                                    ▷ $c_1$
2: b = 0                                                    ▷ $c_2$
3: c = 0                                                    ▷ $c_3$
```

Formally, the total time here is $c_1 + c_2 + c_3 = \Theta(1)$. However, typically we bundle these constant assignment times together and call this all together time 1.

3.2 For Loops

Algorithm 4 for

```
1: $s = 0$                                                  ▷ $c_1$
2: for $i = 1, \dots n$ do               ▷ $c_2$ a total of $n$ times (maintenance)
3:     // Comment                                ▷ 0 a total of $n$ times
4: end for
```

The total time is then $c_1 + c_2 n + (0) \cdot n = \Theta(n)$.

Algorithm 5 sum

```
1: $s = 0$                                                  ▷ $c_1$
2: for $i = 1, \dots n$ do                      ▷ $c_2$ a total of $n$ times
3:     $s = s + i$                              ▷ $c_3$ a total of $n$ times
4: end for
```

The total time is now $c_1 + c_2 n + c_3 n = \Theta(n)$. However, typically we disregard the maintenance line as typically when analyzing time complexity the behavior inside the loop is linear or greater.

Therefore, we typically think of this as $c_1 + c_3 n = \Theta(n)$. Sometimes, we even disregard the constant term and think of this as $c_3 n = \Theta(n)$. If we consider assignments to be time 1, as we usually do, we now get the standard expression:

$$T(n) = n = \Theta(n)$$

The same applies to while loops.

3.3 Conditionals: the fun stuff

.

Let say we have the following algorithm.

Algorithm 6 hi

```
1: if $a < b$ then
2:     print "hi"
3: end if
```

If the conditional takes c_1 time and the print takes c_2, then the time required is

- If $a < b$, time is $c_1 + c_2$
- If $a \geq b$, time is c_1

In this case, note that since both are constant time, we can state that the whole algorithm is just constant time c_1.

However, and here is the issue: what do we do with this?

Algorithm 7 unknown

```
1: if $a < b$ then
2:     Unknown (could be not constant time)
3: end if
```

We can't just absorb this whole thing into a constant time c_1.

Algorithm 8 ifsum

```
1: if $a < b$ then
2:     $s = 0$                                   ▷ $c_1$
3:     for $i = 1, \ldots n$ do                  ▷ $c_2$ a total of $n$ times
4:         $s = s + i$                           ▷ $c_3$ a total of $n$ times
5:     end for
6: end if
```

Notice that

- If $a < b$, then the time is $c_1 + c_2 + c_3n + c_4n = \Theta(n)$
- If $a \geq b$, then the time is $c_1 = \Theta(1)$

Definition 7 (Best Case and Worst Case). The **Best Case** performance of an algorithm is the behavior of an algorithm under optimal conditions, minimizing the performance cost. The **Worst Case** performance of an algorithm is the behavior under the worst conditions, maximizing the performance cost.

Example (If Sum). From our previous If Sum example,

- In best case, $a \geq b$, and the time is $T(n) = 1 = \Theta(1)$.
- In the worst case, $a < b$, and the time is $T(n) = n + n = \Theta(n)$.

Remark. Any line that takes constant time can be ignored provided there is adjacent code which takes nonzero time.

Remark. With loops, as long as the body takes nonzero time we can ignore the maintenance line.

3.4 Average Case Complexity

To calculate the average case complexity, we take the sum of the running time for each input multiplied by the probability of that input. We could define it as the expected value of the function.

Example (If Sum). From the previous if sum example, if $a \geq b$ half of the time and $a < b$ half the time, then we get

$$1 \cdot \frac{1}{2} + (n) \cdot \frac{1}{2} = \frac{1}{2}n + \frac{1}{2} = \Theta(n)$$

Chapter 4

Maximum Contiguous Sum

4.1 Algorithm 1: Naive

Here is our first, naive implementation of MCS.

Algorithm 9 Maximum Contiguous Sum

```
Require: $A$ is a list of length $n$
 1: $M \leftarrow 0$
 2: for $i = 1 \dots n$ do
 3:     for $j = i \dots n$ do
 4:         $S \leftarrow 0$
 5:         for $k = i \dots j$ do
 6:             $S \leftarrow S + A[i]$
 7:         end for
 8:         $M \leftarrow \max(M, S)$
 9:     end for
10: end for
11: return $M$
```

No matter what the case is, note that we always get

$$\begin{aligned} T(n) &= 1 + \sum_{i=0}^{n} \sum_{j=i}^{n} \left(1 + \left(\sum_{k=i}^{j} 1 \right) + 1 \right) \\ &= 1 + \sum_{i=0}^{n} \sum_{j=i}^{n} (1 + j - i + 1 + 1) \end{aligned}$$

Note that since $-i+1$ is constant, we can move it by setting $j = j - i + 1$. Therefore,

$$\begin{aligned}
T(n) &= 1 + \sum_{i=0}^{n} \sum_{j=1}^{n-i+1} 2 + j \\
&= 1 + \sum_{i=0}^{n} n - i + 1 + \frac{(n-i+1)(n-i+2)}{2} \\
&= 1 + \sum_{i=0}^{n} n - i + 1 + \frac{i^2 - i(n+2) - i(n+1) + (n^2+3n+2)}{2} \\
&= 1 + n^2 - \frac{1}{2}n^2 - \frac{1}{2}n + n - \frac{n(n+1)(n+2)}{4} - \frac{n(n+1)^2}{4} + \frac{n^3+3n^2+2}{2} + \dots \\
&= \Theta(n^3)
\end{aligned}$$

Note that this is the same in all cases, since the loop will always run through the whole list.

4.2 Second Naive attempt

Here is our second "brute-force" algorithm.

Algorithm 10 Maximum Contiguous Sum

```
Require: $A$ is a list of length $n$
1: $M \leftarrow 0$
2: for $i = 1 \ldots n$ do
3:     $S \leftarrow 0$
4:     for $j = i \ldots n$ do
5:         $S = S + A[j]$
6:         $M = \max(M, S)$
7:     end for
8: end for
9: return $M$
```

Lets look at the time complexity of this.

$$\begin{aligned}
T(n) &= 1 + \sum_{i=1}^{n} 1 + \sum_{j=i}^{n} 2 \\
&= 1 + \sum_{i=1}^{n} 1 + 2\left(n - i + 1\right) \\
&= 1 + \sum_{i=1}^{n} 2 + 2n - \sum_{i=1}^{n} i \\
&= 1 + 2n + 2n^2 - \frac{n \cdot (n+1)}{2} \\
&= 1 + 2n + 2n^2 - \frac{1}{2}n^2 - \frac{1}{2}n \\
&= \Theta(n^2)
\end{aligned}$$

Note that again this is also true for not only the standard time complexity, but also for best case, worst case, and average case.

4.3 Kadane's Algorithm: Dynamic Programming

Theorem 4 (MCS)**.** Define M_i as the maximum contiguous sum ending at (and including) index i where $1 \leq i \leq n$. Then we have $M_1 = A[1]$ and $M_i = \max\left(M_{i-1} + A[i], A[i]\right)$.

Proof 2. Clearly $M_1 = A[1]$ as there is only one contiguous sum up to the first index, which is the first value.

Denote by C_i the set of contiguous sums ending at index i and denote by C_{i-1} the set of contiguous sums ending at index $i - 1$. Then we have

$$C_i = \{x + A[i] | x \in C_{i-1}\} \cup \{A[i]\}$$

Therefore,

$$\begin{aligned}
M_i = \max(C_i) &= \max\left(\{x + A[i] | x \in C_{i-1}\} \cup \{A[i]\}\right) \\
&= \max\left(\{x + A[i] | x \in C_{i-1}\}, A[i]\right) \\
&= \max\left(M_{i-1} + A[i], A[i]\right)
\end{aligned}$$

□

This means we can progress through the list, finding the maximum contiguous sum ending at and including index i, until we reach the end. Since the maximum contiguous sum overall must end somewhere, by looping through the whole list, we can find the maximum of each M_i.

Algorithm 11 Kadane's Algorithm for MCS

```
Require: $A$ is a list of length $n$
1: $M \leftarrow A[1]$
2: $M_i = A[1]$
3: for $i = 2 \dots n$ do
4:     $M_i = \max(M_i + A[i], A[i])$
5:     $M = \max(M, M_i)$
6: end for
7: return $M$
```

We see that the time complexity of this algorithm is $\Theta(n)$.

Chapter 5

Binary Search and Recurrence

5.1 Binary Search

Definition 8 (Search)**.** Given a sorted list of elements and a target element, find the index of the target element or fail if the element does not exist.

The algorithm idea goes as follows.

- We look at the middle of the list. If the element is not in the middle, we can compare the target to the element to see if its less or greater.
- Concentrate the search to the upper or lower half depending on if the target is greater than or less than the middle element.
- Repeat on the sublist.

Algorithm 12 Binary Search

```
Require: $A$ is a sorted list of length $n$.
Require: $t$ is a target element.
function BINSEARCH($A$, $t$)
    $L \leftarrow 1$
    $R \leftarrow n$
    while $L \leq R$ do
        $C = \left\lfloor \frac{L+R}{2} \right\rfloor$
        if $t < A[C]$ then
            $R = i - 1$
        else if $t > A[C]$ then
            $L = i + 1$
        else
            return $i$
        end if
    end while
    return None
end function
```

5.2 Time Complexity

Notice that the inside of the while loop is all constant time. However, we have to figure out exactly how long the while loop takes. How do we do that?

Lets look at three cases.

- **Best Case:**

  If the target is immediately located on the first check then the total time is $T(n) = c_1 + c_2 = \Theta(1)$.

- **Worst Case:**

  The worst case occurs when the target is never found. Consider the length of each iteration. At first, it is n. After the first iteration, it is $\frac{n}{2}$ (If n is odd then its ± 0.5 but this does not matter). After that the sublist length is $\frac{n}{4}$. This continues until the sublist length is $\frac{n}{2^k}$. This iterates until $L = R$, when the sublist is of length 1. Therefore,

$$\frac{n}{2^k} = 1$$
$$2^k = n$$
$$k = \lg n$$

  However, there is also one extra iteration when $L = R$. Therefore,

$$T(n) = c_1 + c_2\,(1 + \lg n) + c_3 = \Theta(\lg n)$$

- **Average Case:** This is a bit tricky, as we need look at all possible positions of the target within the list.

  Lets look at a few examples. If the list has 3 elements, it will be found after 1 iteration if its in the middle, and 2 if its at the ends.

  When the list has 7 elements, if the target is in the middle (1 element), it will be found after 1 iteration. For two of the elements, it will take 2 iterations, and for 4 of the elements, it will take 3 iterations.

  When the list has 15 elements, we get

  Table 5.1: 15 elements

iterations	elements
1	1
2	2
3	4
4	8

  If we let N be the number of iterations, we get

$$\begin{aligned}\sum_{i=1}^{N}(\text{probability})(\text{time}) &= \sum_{i=1}^{N}\frac{2^{i-1}}{n}(c_1+ic_2)\\ &= \frac{1}{n}\left(\sum_{i=1}^{N}2^{i-1}+c_2\sum_{i=1}^{N}i2^{i-1}\right)\\ &= \Theta(\lg n)\end{aligned}$$

5.3 Recurrence

In the previous binary sort case, we could consider the algorithm as a recursive algorithm. After we compare it, we then perform the same algorithm on a sublist of size $\frac{n}{2}$. So therefore, we have a recurrence relation for $T(n)$ as following:

$$T(n) = 1 + T(\left\lfloor\frac{n}{2}\right\rfloor)$$

How do we then find what $T(n)$ is? We have two goals for recurrence relations.

1. Find a closed form expression for $T(n)$.
2. Find Θ for $T(n)$ if finding the closed form is difficult.

5.4 Digging Down and Closed Form Expressions

Example (Digging Down). $T(n) = 2T(\frac{n}{2}) + \frac{1}{2}n$, where $T(1) = 7$. How exactly do we solve this? We can try expanding this out and "digging down". We know that

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + \frac{1}{2}n \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{1}{2}\left(\frac{n}{2}\right)\right) + \frac{1}{2}n \\
&= 4T\left(\frac{n}{4}\right) + n \\
&= 4\left(2T\left(\frac{n}{8}\right) + \frac{1}{2}\left(\frac{n}{4}\right)\right) + n \\
&= 8T\left(\frac{n}{8}\right) + \frac{3}{2}n \\
&= 8\left(2T\left(\frac{n}{16}\right) + \frac{1}{2}\left(\frac{n}{8}\right)\right) + \frac{3}{2}n \\
&= 16T\left(\frac{n}{16}\right) + 2n
\end{aligned}$$

We see that the general expression for the above is

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \frac{k}{2}n$$

This ends when $\frac{n}{2^k} = 1$, where $T(1) = 7$. This is when $k = \lg n$. The expression then becomes

$$T(n) = 2^{\lg n}T(1) + \frac{1}{2}n\lg n = 7n + \frac{1}{2}n\lg n = \Theta(n\lg n)$$

5.5 The Recurrence Theorem

This calculation seems very tedious to do over and over again. Is there a way to simplify this (or maybe just get the time complexity we care about?) much more easily? Yes!

Theorem 5 (Recurrence Theorem). Suppose $T(n)$ satisfies the recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

for positive constants $a \geq 1$ and $b \geq 1$ and where $\frac{n}{b}$ can mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$ (it does not matter). Then we have:

1. If $f(n) = \mathcal{O}(n^c)$ and $\log_b a > c$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^c)$ and $\log_b a = c$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$

3. If $f(n) = \Theta\left(n^c \lg^k n\right)$ and $\log_b a = c$, then $T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$ (extension of 2)
4. If $f(n) = \Omega(n^c)$ and $\log_b a < c$ then $T(n) = \Theta(f(n))$.

> **Note.** For case 4, $f(n)$ must also satisfy a regularity condition, which states that $\exists C < 1$ and n_0 such that $af\left(\frac{n}{b}\right) \leq Cf(n)$ for all $n \geq n_0$. This is almost always true.

> **Note.** Though we have that in the conditions f could be $\mathcal{O}$ and Ω, it is typically easier to find the Θ and then consider which case it is.

Proof 3. (When $f(n) = 0$).

Note that we can solve the recurrence relation easily as if we know

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) \\
&= a^2T\left(\frac{n}{b^2}\right) \\
&= a^3T\left(\frac{n}{b^3}\right) \\
&= \dots \\
&= a^kT\left(\frac{n}{b^k}\right)
\end{aligned}$$

When $k = \log_b n$, we have

$$\begin{aligned}
T(n) &= a^kT(1) \\
&= a^{\log_b n}T(1) \\
&= a^{\frac{\log_a n}{\log_a b}}T(1) \\
&= \left(a^{\log_a n}\right)^{\frac{1}{\log_a b}} T(1) \\
&= n^{\frac{1}{\log_a b}}T(1) \\
&= n^{\log_b a}T(1) \\
&= \Theta(n^{\log_b a})
\end{aligned}$$

(Intuition for $f(n)$).

- If this new time requirement $f(n)$ is at most polynomially smaller than the recursive part ($\mathcal{O}$), then the recursive part dominates, therefore

$$f(n) = \mathcal{O}(n^c),\ c < \log_b a \implies T(n) = \Theta(n^{\log_b a})$$

- If this new time requirement $f(n)$ is the same (meaning Θ) polynomially as the recursive part, they combine and introduce a logarithmic factor. This is difficult to prove.

$$f(n) = \Theta(n^c),\ c = \log_b a \implies T(n) = \Theta\left(n^{\log_b a} \lg n\right)$$

- If this new time requirement $f(n)$ is at least (meaning Ω) polynomially larger than the recursive part, then this time requirement is the dominating factor.

$$f(n) = \Omega(n^c),\ c > \log_b a \implies T(n) = \Theta(f(n))$$

> **Note.** Don't forget the regularity condition!

$\square$

Example (Recurrence Theorem). Consider the prefix $2T\left(\frac{n}{4}\right)$. We see that $\log_4 2 = 0.5$.

- If $T(n) = 2T\left(\frac{n}{4}\right) + n^{\frac{1}{3}}$. We see that $c = \frac{1}{3}$. Since $0.5 > \frac{1}{3}$, and $f(n) = \mathcal{O}(n^{\frac{1}{3}})$, we see that $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{0.5})$.

- If $T(n) = 2T\left(\frac{n}{4}\right) + n^{\frac{1}{2}}$. We see that $c = \frac{1}{2}$. Since $0.5 = 0.5$, and $f(n) = \Theta(n^c)$, we have that $T(n) = \Theta(n^{\log_4 2} \lg n) = \Theta(n^{0.5} \lg n)$.

- If $T(n) = 2T\left(\frac{n}{4}\right) + n \lg n$, we see that $f(n) = n \lg n = \Omega(n^c)$, where $c = 1$. We see that $0.5 < 1$. Therefore, $T(n) = \Theta(f(n)) = \Theta(n \lg n)$.

  > **Note.** Note that $f(n) = n \lg n$ satisfies the regularity condition because:
  >
  > $$af\left(\frac{n}{b}\right) = 2f\left(\frac{n}{4}\right) = \frac{n}{2} \ln \frac{n}{4} \leq \frac{1}{2} n \ln n \leq \frac{1}{2} f(n)$$
  >
  > when $n \geq n_0 = 1$.

- If $T(n) = 2T\left(\frac{n}{4}\right) + 17$, we see that $f(n) = \Theta(n^c)$ where $c = 0$. Therefore, since $0.5 > 0$, we see that $T(n) = \Theta(n^{0.5})$

Example (Non example). Here are some examples where this theorem does not apply.

- Suppose $T(n) = 2T\left(\frac{n}{4}\right) + 3T\left(\frac{n}{2}\right) + n$. Here this theorem does not apply.

> **Note.** There is another method that often applies called the Akra-Bazzi method. It applies to recurrences of the form
>
> $$T(n) = f(n) + \sum_{i=1}^{k} a_i T\left(b_i n + h_i(n)\right)$$

- Suppose $T(n) = 2T\left(\frac{n}{4}\right) + f(n)$ and you know that $f(n) = \mathcal{O}(n^2)$. Since we only know $f(n) = \mathcal{O}(n^2)$, we can only check Case 1. We see that $c = 2$. However, $\log_b a = 0.5 < 2$. We need $c < \log_b a$ to use case 1, and therefore this theorem does not apply here.

Chapter 6

Sorting Algorithms

6.1 Bubble Sort

Definition 9 (Standard Bubble Sort)**.** Here is the naive implementation of Bubble Sort.

Algorithm 13 Bubble Sort

```
Require: $A$ is a list of length $n$
for $i = n \dots 2$ do
    for $j = 1 \dots i - 1$ do
        if $A[j] > A[j+1]$ then
            $A[j] \leftrightarrow A[j+1]$
        end if
    end for
end for
return $A$, sorted.
```

Here we can analyze multiple parts of our naive algorithm.

- **Comparisons:** We see that the number of comparisons is equal to

$$\begin{aligned}\mathrm{Cmp}(n) &= \sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 \\ &= \sum_{i=2}^{n} i-1 \\ &= \frac{1}{2}n^2 - \frac{1}{2}n - 1 - n + 1 \\ &= \frac{1}{2}n^2 - \frac{1}{2}n - n \\ &= \binom{n}{2} + \dots \\ &= \Theta(n^2)\end{aligned}$$

> **Note.** Note that the $\binom{n}{2}$ is important. We are comparing every pair of elements selected from the list.

- **Exchanges:**
  - **Best case:** In the best case, there are no exchanges at all as the input is sorted. Therefore, 0.
  - **Worst case:** We have to swap every element after each comparison. Therefore, it is $\Theta(n^2) = \binom{n}{2} + \dots$.
  - **Average case:** Note that each permutation of the list is equally likely. We can also notice that the number of "inversions" or out of order pairs is equal to the number of exchanges made. We finally note that every pair is equally likely to be inverted or not. Therefore, for each $\binom{n}{2}$ pairs, there is a $\frac{1}{2}$ probability that they are inverted. So the total comparisons is
  $$\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4}$$
- **Time:** Note that it is always the same ($\Theta(n^2)$), as we always go through the loop no matter what (see: comparisons).
- **Space:** It always uses $\Theta(1)$ auxiliary space: two indices and possibly a swap variable.

> **Remark.** Our naive BubbleSort is stable. This means that the order of identical entries is preserved.

Remark. Our naive BubbleSort is also in-place, which means we do not have to create a new list to sort, we only move elements within.

Definition 10 (Modified Bubble Sort)**.** Here is a slightly better version of BubbleSort. We note that every pass through the list in BubbleSort, it sends the largest element to the end. If we don't do any swaps for the last k elements, we don't have to go all the way to the end on the next pass, only up to $n - k$.

Algorithm 14 Modified Bubble Sort

```
Require: $A$ is a list of length $n$
 1: $i \leftarrow n - 1$
 2: while $i > 0$ do
 3:     $t \leftarrow 1$
 4:     for $j = 1 \ldots i - 1$ do
 5:         if $A[j] > A[j+1]$ then
 6:             $A[j] \leftrightarrow A[j+1]$
 7:             $t \leftarrow j$
 8:         end if
 9:     end for
10:     $i \leftarrow t - 1$
11: end while
12: return $A$, sorted.
```

Note that if there are no swaps done at all, then this instantly exits. In this case, for the time complexity, we have

- Best Case: We only have one pass, giving us $\Theta(n)$.
- Worst Case: We still need $\binom{n}{2}$ swapped pairs, giving us $\Theta(n^2)$
- Average Case: Doing some combinatorics again, we swap about half the time, still getting $\Theta(n^2)$.

6.2 Insertion Sort

Idea behind insertion sort: We pass through the list from left to right. If we encounter an entry smaller than some of its predecessors, then we need to move it as far left as is appropriate. We shift as many elements to the right needed to make room for it and then insert it into the proper position.

Definition 11 (Insertion Sort with Sentinel)**.** Here we use a "sentinel" value, a specific value used for comparison right at the beginning of the list.

Algorithm 15 Insertion Sort with Sentinel

```
Require: $A$ is a list of length $n$ (1-indexed)
 1: $A[0] \leftarrow -\infty$
 2: for $i = 2 \dots n$ do
 3:     $t \leftarrow A[i]$
 4:     $j \leftarrow i - 1$
 5:     while $t < A[j]$ do
 6:         $A[j+1] \leftarrow A[j]$
 7:         $j \leftarrow j - 1$
 8:     end while
 9:     $A[j+1] \leftarrow t$
10: end for
11: return $A$, sorted.
```

We can now analyze this in multiple ways.

- **Comparisons:**
  - **Best case:** In the best case, the list is already sorted. We do $n-1$ comparisons.
  - **Worst case:** In the worst case, we have to swap every time (reverse order). Therefore, we get
$$\sum_{i=2}^{n}\sum_{j=0}^{i-1} 1 = \sum_{i=2}^{n} i = \frac{1}{2}n^2 + \frac{1}{2}n - 1 = \frac{(n-1)\cdot(n+2)}{2}$$
  - **Average case:** For average case, we see that for every "insertion" of each element, it has a $\frac{1}{i}$ probability of going i steps down. Therefore, we get the sum
$$\sum_{i=2}^{n}\sum_{j=1}^{i} \frac{1}{i}\cdot(i-j+1) = \frac{(n-1)\cdot(n+4)}{4}$$
- **Assignments and reads (from list):** From the algorithm, we see that for each comparison that goes through, we do 1 assignment, and for each comparison at all, we do 2 assignments (plus 1 from the sentinel).
- **Time complexity:**
  - **Best case:** Note that in the best case, it is sorted so there is only one pass, so it behaves as $\Theta(n)$.
  - **Worst case:** Just like comparisons, in worst case we get
$$3\cdot\frac{(n-1)\cdot(n-2)}{2} + 1 = \Theta(n^2)$$
  - **Average case:** On average we can still analyze based on inversions, which gives us
$$T(n, I) = \underbrace{n+1}_{\text{Pre / Post while}} + \underbrace{I}_{\text{Inversions: while loop checks and passes}} + \underbrace{n-1}_{\text{While loop checks and fails}}$$

Note that the average number of inversions is

$$\frac{(n)\cdot(n-1)}{4}$$

So therefore, $T(n) = \Theta(n^2)$.

- **Space:** Space is $\Theta(1)$.

Definition 12 (Insertion Sort without Sentinel)**.** We can actually get rid of the sentinel from insertion sort.

Algorithm 16 Insertion Sort without Sentinel

```
Require: $A$ is a list of length $n$ (1-indexed)
 1: for $i = 2 \ldots n$ do
 2:     $t \leftarrow A[i]$
 3:     $j \leftarrow i - 1$
 4:     while $j > 0$ and $t < A[j]$ do
 5:         $A[j+1] \leftarrow A[j]$
 6:         $j \leftarrow j - 1$
 7:     end while
 8:     $A[j+1] \leftarrow t$
 9: end for
10: return $A$, sorted.
```

Remark. Note that here, we lose all our comparisons to the sentinel (this is constant so it doesn't really matter in the long term but eh). In the average case, for each n values, in the old algorithm we compared it to the end with probability $\frac{1}{i}$. Therefore, we now get in the average case

$$\frac{(n-1)\cdot(n+4)}{4} - \sum_{i=2}^{n}\frac{1}{i} = \frac{(n-1)\cdot(n+4)}{4} - H_n + 1$$

where H_n is the n-th harmonic number.

6.3 Selection Sort

Idea: We find the largest element, and move it to the last value. Then we find the largest element in the subarray not including the last, and put that at the end. And we continue this until the array is sorted.

Definition 13 (Selection Sort (algorithm))**.** Here is the pseudocode for the selection sort.

Algorithm 17 Selection Sort (algorithm)

```
Require: $A$ is a list of length $n$
 1: for $i = n \dots 2$ do
 2:     $k \leftarrow 1$
 3:     for $j = 2 \dots i$ do
 4:         if $A[j] > A[k]$ then
 5:             $k \leftarrow j$
 6:         end if
 7:     end for
 8:     $A[k] \leftrightarrow A[i]$
 9: end for
10: return $A$, sorted.
```

Similar to BubbleSort, we can see that this algorithm is typically inefficient.

- **Comparisons:** The total comparisons it makes is $\frac{n(n-1)}{2}$.
  - **Best Case:** $\Theta(n^2)$, left as exercise.
  - **Worst Case:** $\Theta(n^2)$, left as exercise.
  - **Average Case:** $\Theta(n^2)$, left as exercise.
- **Exchanges:** Worst case, the exchanges it makes is $n - 1$.
  - **Best Case:** $\Theta(1)$, left as exercise.
  - **Worst Case:** $\Theta(n)$, left as exercise.
  - **Average Case:** $\Theta(n)$, left as exercise.
- **Time:** $\Theta(n^2)$ in all cases.
- **Space:** $\Theta(1)$ in all cases.

Remark. Note that selection sort is NOT stable.

Remark. The Cocktail Sort is where we swap both the minimum and maximum at the same time. This takes 3 comparisons for 2 elements, but sorts the list 50% faster, for a total net 25% speedup.

Definition 14 (Selection Sort). Any sorting algorithm in which we find the largest element, then the second largest, then the third largest, and so on.

Example (Selection Sort (def)). These are all examples of the type of “selection sort”.

- Selection Sort
- Bubble Sort

- Heap Sort (we will see this soon!)

6.4 Mergesort

Idea: We divide our list into two adjacent half size sublists. Assuming these are sorted recursively from this algorithm, we "merge" the two lists by combining the two sorted arrays into one larger sorted array.

Definition 15 (Merge Sort)**.** This is the pseudocode for our mergesort algorithm.

Algorithm 18 Merge Sort

```
Require: $A$ is a list of length $n$
 1: function MergeSort(A)
 2:     if $n = 1$ then
 3:         return $A$
 4:     end if
 5:     $m \leftarrow \lfloor \frac{n}{2} \rfloor$
 6:     $L = A[1 \dots m]$
 7:     $R = A[m+1 \dots n]$
 8:     $L = \text{MergeSort}(L)$
 9:     $R = \text{MergeSort}(R)$
10:     $A = \text{Merge}(A, L, R)$
11:     return $A$
12: end function
13: procedure Merge(A, L, R)
Require: $A$ is a list of length $n$
Require: $L$ is a list of length $m$
Require: $R$ is a list of length $n - m$
14:     $i \leftarrow 0$
15:     $j \leftarrow 0$
16:     $k \leftarrow 0$
17:     while $i < m$ and $j < n - m$ do
18:         if $L[i] \leq R[j]$ then
19:             $A[k] \leftarrow L[i]$
20:             $i \leftarrow i + 1$
21:             $k \leftarrow k + 1$
22:         else
23:             $A[k] \leftarrow R[j]$
24:             $j \leftarrow j + 1$
25:             $k \leftarrow k + 1$
26:         end if
27:     end while
28:     while $i < m$ do
29:         $A[k] \leftarrow L[i]$
30:         $i \leftarrow i + 1$
31:         $k \leftarrow k + 1$
32:     end while
33:     while $j < n - m$ do
34:         $A[k] \leftarrow R[j]$
35:         $j \leftarrow j + 1$
36:         $k \leftarrow k + 1$
37:     end while
38:     return $A$
39: end procedure
```

If we consider merging, we can see that in the best case, the number of comparisons is m where

$m \le n - m$. This is when the last element of the shorter sorted list is smaller than the first element of the other list ($L_m < R_1$).

The worst case is $m+(n-m)-1$ comparisons, when L_m and R_{n-m} are the two largest elements. The average case when $m = n - m$ is $2m - 2 + \frac{2}{m-1}$.

Remark. Note that if $m = 1$, we can be more efficient and do the merge procedure using binary search.

To evaluate the time complexity, we can observe that we have two recursive calls and then a merge step which takes $\Theta(n)$. Therefore, we have that

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n)$$

With $T(1) = c$ and $f(n) = \Theta(n)$. Therefore, this results in $T(n) = \Theta(n \lg n)$. Note that this is best, worst, and average case.

A full analysis: By digging down we see that if $T(n) = 2T\left(\frac{n}{2}\right) + n$,

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

Then if $k = \lg n$, we get

$$T(n) = 2^{\lg n} \cdot c_1 + \lg n \cdot n = n + n \lg n = \Theta(n \lg n)$$

Space: We can see that the auxiliary space satisfies the recurrence relation $S(n) = S\left(\frac{n}{2}\right) + f(n)$ where $f(n) = \Theta(n)$. This is because the first mergesort subcall takes $S\left(\frac{n}{2}\right)$ space, which is then freed and used again in the second call. Then the merge step requires $f(n)$. Therefore, the total space used is $\Theta(n)$.

Remark. Mergesort is not an in place sort. However, it is stable.

Remark. Mergesort "can" be made to be in place using linked lists.

6.5 Heap Sort

Definition 16 (Complete Binary Tree)**.** A complete binary tree is a binary tree in which all levels are completely filled, except possibly for the last level, and the last level has all entries as far left as possible.

Example (Complete Binary Tree)**.**

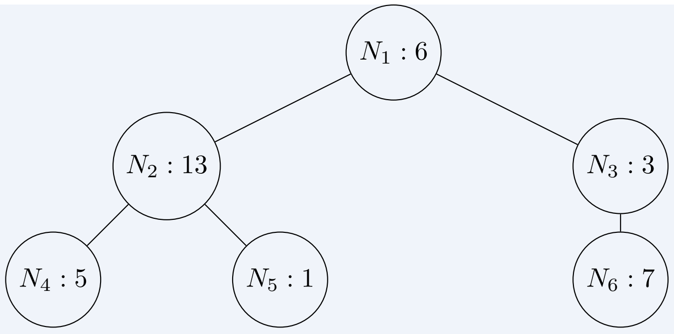


Figure 6.1: A Complete Binary Tree

Note. We begin the node indexing at the root node at $i = 1$ instead of 0.

Theorem 6 (Complete Binary Trees and Arrays)**.** Every complete binary tree can be expressed as an array such that:

- If a node has index i, then its left and right children have indices $2i$ and $2i + 1$.
- If a node has even index i then its parent has index $\frac{i}{2}$ and if it has an odd index i then its parent has index $\frac{i-1}{2}$. We denote this as $\left\lfloor \frac{i}{2} \right\rfloor$.

Proposition 1. Consider a complete binary tree expressed as an array.

- If there are n nodes total, then the largest node with children is the node with index $\left\lfloor \frac{n}{2} \right\rfloor$.
- If a node with index i has children then all nodes with smaller indices have children.
- If there are n nodes total, then the nodes with indices $1, 2, \dots, \left\lfloor \frac{n}{2} \right\rfloor$ have children.

Definition 17 (Max Heap)**.** We define a max heap as a complete binary tree in which each node's value is greater than or equal to that node's children's value if that node has children. In other words, keys non-strictly decrease as we go down the branches. This is known as the **heap property**.

Example (Max Heap)**.**

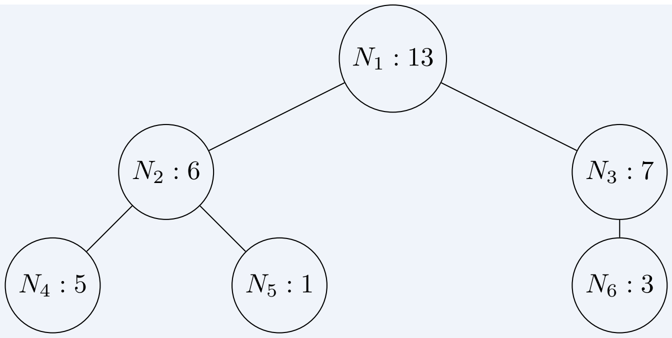


Figure 6.2: A Max Heap

Given a complete binary tree, it is possible to rearrange the nodes so as to obtain a max heap.

Definition 18 (Max Heapify). We define the maxheapify function as follows.

Algorithm 19 Max Heapify

```
Require: $A$ is a list of length $n$
Require: $i$ is an index of a node in $A$
function MAXHEAPIFY(A, i)
    $V \leftarrow A[i]$
    $V_i \leftarrow i$
    if $A[2i]$ exists then
        $V \leftarrow \max(V, A[2i])$
        $V_i \leftarrow$ index associated with the max $V$ (either $i$ or $2i$)
    end if
    if $A[2i+1]$ exists then
        $V \leftarrow \max(V, A[2i+1])$
        $V_i \leftarrow$ index associated with the max $V$ (either $i$, $2i$, or $2i+1$)
    end if
    $A[i] \leftrightarrow A[V_i]$
    if $V_i \neq i$ then
        MaxHeapify$(A, V_i)$
    end if
end function
```

The maxheapify function takes an element at index i and floats the key down the tree as far as necessary to ensure that the subtree rooted at index i is also a max heap. The algorithm swaps the node down until either

(a) It is larger than the two child nodes, in which case it gets swapped to its own spot and the routine finishes.

(b) It is recursively called until the value is sifted down to a leaf, in which case it stays at its own spot and the routine finishes.

We can see that if i is the index of the node we are max heapify-ing then this node is in level $\lfloor \lg i \rfloor$.

- In the best case, we compare with the children and there is no issue, so it is a constant time $\Theta(1)$.
- In the worst case we need to check all the way to the bottom level. The bottom level is $\left\lfloor \frac{n}{2} \right\rfloor$, so this is $\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1$. Note that $\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1 \leq 1 + \lg n$, so this is $\mathcal{O}(\lg n)$.

Definition 19 (Converting Binary Tree to Max Heap)**.** We can convert a Complete Binary Tree to a Max Heap. To do so we start at the nodes with children and max heapify each successive node as we progress towards the root.

Algorithm 20 Converting to Max Heap

```
Require: $A$ is a list of length $n$
1: procedure ConvertToHeap(A)
2:     for $i = \left\lfloor \frac{n}{2} \right\rfloor, \ldots 0$ do
3:         MaxHeapify$(A, i)$
4:     end for
5:     return $A$ as a max heap.
6: end procedure
```

We are running this process on $\left\lfloor \frac{n}{2} \right\rfloor$ nodes.

- In the best case, we get a $\Theta(1)$ function $\left\lfloor \frac{n}{2} \right\rfloor$ times. Since we know that $\frac{n}{2} \leq \left\lfloor \frac{n}{2} \right\rfloor \leq \frac{n}{2} + 1$ this is $\Theta(n)$.
- In the worst case we are running a $\mathcal{O}(\lg n)$ process $\left\lfloor \frac{n}{2} \right\rfloor$ times. Since we know that $\left\lfloor \frac{n}{2} \right\rfloor \leq \frac{n}{2} + 1$, we know this is $\mathcal{O}(n \lg n)$.

Definition 20 (Heapsort)**.** Note that a max binary heap is structured nicely such that extracting keys in a sorted manner is easy. There are several ways to do this, but we will consider the way where we observe that the largest key is at the root node.

Algorithm 21 Heap Sort

```
Require: $A$ is a list of length $n$.
1: procedure Heapsort(A)
2:     ConvertToHeap$(A)$
3:     for $i = n, \ldots 2$ do
4:         $A[1] \leftrightarrow A[i]$
5:         MaxHeapify$(A[1 : (i - 1)], 1)$
6:     end for
7: end procedure
```

What we do is we first convert our array into a max binary heap. We then take the root node, and swap it for the last node. Finally, we cut the last node off the tree, and heapify the remaining subtree. We then repeat until all the nodes have been cut off the tree.

In the worst case, note the following.

- First, converting to the max heap will take $\mathcal{O}(n \lg n)$.
- For each $i = n \ldots 2$, we swap i with node 1, we cut node i off the tree, and then we max heapify node 1. This is $\mathcal{O}(\lg(i-1))$. Therefore, the total time required is

$$\begin{aligned}\mathcal{O}(n \lg n) + \sum_{i=2}^{n} \Theta(1) + \Theta(1) + \mathcal{O}(\lg(i-1)) &\leq \mathcal{O}(n \lg n) + \sum_{i=2}^{n} \mathcal{O}(\lg(i-1)) \\ &\leq \mathcal{O}(n \lg n) + \sum_{i=2}^{n} \mathcal{O}(\lg n) \\ &\leq \mathcal{O}(n \lg n) + (n-1)\mathcal{O}(\lg n)\end{aligned}$$

Therefore, the time complexity is $\mathcal{O}(n \lg n)$.

Note. We can actually use Stirling's approximation here. The number of sifts or max heapify operations we must do is $n - 1 \sim n$. The number of levels per sift is $\sim \lg n - i$ on the i-th run. We do 2 comparisons per level per sift. So the total number of comparisons is $\sim 2n \lg n$. We get

$$\begin{aligned}\sum_{i=0}^{y} 2\lg(n-i) &= \sum_{i=1}^{n} 2\lg i \\ &= 2\sum_{i=1}^{n} \lg i \\ &= 2\lg(1 \cdot 2 \cdot 3 \ldots n) \\ &= 2\lg(n!) \\ &\approx 2\lg\left(\sqrt{2\pi n}\frac{n}{e}^{n}\right) \\ &= 2\left(n\lg n - n\lg e + \frac{1}{2}\lg n + C\right) \\ &= 2n\lg n + \mathcal{O}(n) \\ &= \mathcal{O}(n\lg n)\end{aligned}$$

In the best case, we have two options.

- If we start with a heap of distinct elements which is already a max heap, converting to a max heap will be $\Theta(n)$. However, the swap, cut, and max heapify still occur, which still takes $\mathcal{O}(n \lg n)$.

- If we start with a heap of identical elements, then converting to a max heap will be $\Theta(n)$. However, the swaps, cuts, and max heapify this time will be $\mathcal{O}(n) + \sum_{i=2}^{n} \Theta(1) + \Theta(1) + \Theta(1) = \mathcal{O}(n)$.

Remark. Heapsort uses $\mathcal{O}(1)$ auxiliary space.

Remark. Heapsort is unstable.

Remark. Heapsort is in-place.

Remark. Heapsort is rarely used as a general sorting algorithm because Quicksort is usually better. However, max heaps are used frequently in things such as priority queues and scheduling. The reason is insertion and deletion is $\Theta(\lg n)$ on a max heap instead of $\Theta(n)$ on a list.

6.6 Quicksort

Intuition: We pick out an item, which we call the pivot value. Then we rearrange the list and put all smaller elements to the left, all larger items to the right, and the pivot in between. We then recursively apply this to the left and right sublists. The choice of the pivot is nuanced. For now, we will pick the last value in the list.

Algorithm 22 Quicksort

```
Require: $A$ is a list of length $n$
1: procedure QUICKSORT($A$, $L$, $R$)
2:     $I$ = Partition($A$, $L$, $R$)
3:     Quicksort($A$, $L$, $I - 1$)
4:     Quicksort($A$, $I + 1$, $R$)
5: end procedure
```

The intuition behind this algorithm is we partition to put all larger values above the pivot and all smaller values below the pivot. Then, using the pivot index I, we quicksort below it and above it.

How do we partition? So say we are given an list to partition

$$A_0, A_1, A_2, \dots A_r$$

using A_r as the pivot. We process the list from left to right, keeping a sublist of small and large values. The small values will be in the sublist $A[0 : i]$, and the large values will be in $A[i + 1 : j]$.

$$\underbrace{A_0, A_1, A_2, \dots A_i,}_{\text{smaller}} \underbrace{A_{i+1}, A_{i+2}, \dots A_j,}_{\text{larger}} \underbrace{A_{j+1}, A_{j+2}, \dots A_{r-1},}_{\text{unprocessed}} A_r$$

We start the procedure by having an empty small list at index $A[0:0]$, and an empty large list at $A[0:0]$. We then process each value moving left to right.

Case 1: The next value is larger than the pivot. We can just increase the size of our large list from $A[i+1:j]$ to $A[i+1:j+1]$.

$$\underbrace{A_0, A_1, A_2, \dots A_i,}_{\text{smaller}} \underbrace{A_{i+1}, A_{i+2}, \dots A_j, \boldsymbol{A_{j+1}},}_{\text{larger}} \underbrace{A_{j+2}, \dots A_{r-1},}_{\text{unprocessed}} A_r$$

Case 2: The next value is smaller than the pivot. Then we just swap $A[i+1]$ with $A[j+1]$, moving the small value to the beginning of our large list. Then we increase the size of the small list to $A[0:i+1]$ and shift over the large list to $A[i+2:j+1]$.

$$\underbrace{A_0, A_1, A_2, \dots A_i, \boldsymbol{A_{j+1}}}_{\text{smaller}} \underbrace{A_{i+2}, \dots A_j, \boldsymbol{A_{i+1}},}_{\text{larger}} \underbrace{A_{j+2}, \dots A_{r-1},}_{\text{unprocessed}} A_r$$

Finally, we put the pivot in the middle.

Algorithm 23 Partitioning

```
Require: $A$ is a list of length $n$.
Require: $p$ is the beginning index of the run we want to partition.
Require: $r$ is the last index (and the value of the pivot in this case).
 1: $X \leftarrow A[r]$
 2: $i \leftarrow p - 1$                          ▷ $i$ is ending index of small list
 3: for $j = p \dots r - 1$ do                   ▷ $j$ is ending index of large list
 4:     if $A[j] \le X$ then
 5:         $i \leftarrow i + 1$
 6:         $A[i] \leftrightarrow A[j]$
 7:     end if
 8: end for
 9: $A[i+1] \leftrightarrow A[r]$
10: return $i + 1$
```

Remark. Pivot value choice. Note that if we choose the last value of the list as the pivot, if the list is close to sorted, it takes longer to run. Therefore, choosing the last value as the pivot maybe less than ideal. One other way to choose the pivot value is randomly. Another way might be to choose the pivot value as the median, but this is somewhat challenging as well.

We note that if $T(n)$ is the time complexity of a call to Quicksort and k is the pivot position, we get

$$T(n) = T(k) + T(n-k-1) + (n-1)$$

The partitioning call is linear, so we have an additional $n - 1$.

1. **Worst Case:** The worst case occurs when the pivot index is the largest (or smallest) value. This creates two recursive calls of size 0 and $n - 1$. Therefore, we get

$$T(n) = \begin{cases} 0 & n \le 1 \\ T(n-1) + (n-1) & \text{otherwise} \end{cases}$$

Therefore, we get $(n-1)+(n-2)+(n-3)+\dots 2+1$, which is $\binom{n}{2}$. By digging down, we see that $T(n)=\Theta(n^2)$.

2. **Best Case:** The best case occurs when the pivot index is perfectly in the center, splitting the list into two half length sublists. This gives

$$T(n) = T\left(\left\lfloor\frac{n-1}{2}\right\rfloor\right) + T\left(\left\lceil\frac{n-1}{2}\right\rceil\right) + n - 1$$

This results in $T(n)=\Theta(n\lg n)$.

Proof 4. Base case: $n=1$. We see that $an\lg n = a\cdot 1\cdot 0 = 0$.

Assume this is true for $i = 1\dots n$.

Consider $a\cdot(n+1)\cdot\lg(n+1)$. We see that

$$\begin{aligned}T(n) &= T\left(\left\lfloor\frac{n-1}{2}\right\rfloor\right) + T\left(\left\lceil\frac{n-1}{2}\right\rceil\right) + n - 1\\ &\le 2T\left(\frac{n}{2}\right) + n - 1\end{aligned}$$

If S is continuous and monotonic. Then

$$\begin{aligned}T(n) &\le 2T\left(\frac{n}{2}\right) + n - 1\\ &\le 2a\left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) + n - 1\\ &\le an\lg\left(\frac{n}{2}\right) + n - 1\\ &\le an\left(\lg n - \lg 2\right) + n - 1\\ &\le an(\lg n - 1) + n - 1\\ &\le an\lg n + (1-a)n - 1\\ &\le an\lg n\end{aligned}$$

For our inductive step to be correct, we need $1-a\le 0$, or $a\ge 1$. Therefore, we pick $a=1$. And by induction, we get

$$T(n)\le n\lg n$$

□

3. **Average Case:** Note that every partition is always $\Theta(n)$, as it has to go through the whole list. So the only choice the algorithm depends on is where the pivot value ends up. Therefore, we get

$$T(n) = \begin{cases} 0 & n \leq 1 \\ \frac{1}{n} \sum_{k=0}^{n-1} T(k) + T(n-k-1) + n & \text{otherwise} \end{cases}$$

This ends up being $\mathcal{O}(n \lg n)$.

We can make an approximation. Since the pivot in the best case is $\frac{n}{2}$, and the pivot in the worst case is 0 or $n-1$, then we can assume on average the pivot ends up at $\frac{n}{4}$. Therefore, we get the recurrence

$$T(n) = \begin{cases} 0 & n \leq 1 \\ T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n - 1 & \text{otherwise} \end{cases}$$

Proof 5. We can guess $T(n) \leq an \lg n$ for $n \geq 1$. We see that $an \lg n = a \cdot 1 \cdot 0 = 0$.

Assume this is true for $i = 1 \dots n$.

Consider $a \cdot (n+1) \cdot \lg(n+1)$. We see that

$$\begin{aligned}
T(n) &= T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n - 1 \\
&\leq a\left(\frac{n}{4}\right)(\lg n - \lg 4) + a\frac{3n}{4}\lg\left(\frac{3n}{4}\right) + n - 1 \\
&= a\left(\frac{n}{4}\right)(\lg n - \lg 4) + a\left(\frac{3n}{4}\right)(\lg 3 + \lg n - \lg 4) + n - 1 \\
&= a\left(\frac{n}{4}\right)(\lg n - 2) + a\left(\frac{3n}{4}\right)(\lg n + \lg 3 - 2) + n - 1 \\
&= \left(a\left(\frac{n}{4}\right)\lg n - 2a\left(\frac{n}{4}\right)\right) + \left(a\left(\frac{3n}{4}\right)\lg n + a\left(\frac{3n}{4}\right)\lg 3 - 2a\left(\frac{3n}{4}\right)\right) + n - 1 \\
&= an \lg n + \left(-\frac{1}{2} + \frac{3}{4}\lg 3 - \frac{3}{2}\right)an + n - 1 \\
&= an \lg n - (2 - \frac{3}{4}\lg 3)an + n - 1 \\
&= an \lg n - ((2 - \left(\frac{3}{4}\right)\lg 3)a - 1)n - 1
\end{aligned}$$

This has to be less than $an \lg n$ for our induction to work. Therefore, we need $(2 - \left(\frac{3}{4}\right)\lg 3)a - 1 \geq 0$, so $a \geq 1.23$ approximately. Therefore, if we pick $a = 1.23$, we see that for $n \geq 1$,

$$T(n) \leq an \lg n$$

Therefore, $T(n) = \mathcal{O}(n \lg n)$. ☐

Remark. Quicksort is in place: It uses a constant amount of extra space.

Remark. Quicksort is not stable, meaning it does not preserve the order of equal elements.

Remark. Note that when looking at the algorithm, we execute two lines of recursive quicksort calls. Therefore, we must store the $(q+1, r)$ on a stack to execute after we've completed the recursive call to the first quicksort.

- If the pivot is the largest element every time, we see that we will get n elements on the stack: $(n+1, n), (n, n-1), (n-1, n-2), \ldots$.
- If the pivot is the smallest element, now we get zero elements on the stack, since the first Quicksort algorithm always finishes. Every time we execute a recursive call at stage i, (i, n) is pushed on the stack, and then removed immediately after.
- If the pivot is the median element, then we see that the stack will recursively get $((n+1)/2+1, n), ((n+1)/4+1, (n+1)/2-1), ((n+1)/8+1, (n+1)/4-1), \ldots$. Therefore, on average we see that the stack height is $\Theta(\log n)$.

Not a fundamental definition, but we say that an algorithm is in place if it uses $\mathcal{O}(1)$ extra variables and $\mathcal{O}(\log n)$ extra index variables on average. If the index variables take $\mathcal{O}(\log n)$ bits, we say that on average it takes $\mathcal{O}((\log n)^2)$ bits.

Chapter 7

Other Algorithms

7.1 Integer Multiplication

Let us introduce the problem with addition first.

Algorithm 24 Addition

```
procedure ADD($x, y, z$)
    $C \leftarrow 0$
    for $i = 0 \ldots n-1$ do
        $(C, z[i]) \leftarrow \text{AddDigits}(x[i], y[i], C)$
    end for
    $z[n] \leftarrow C$
end procedure
```

Adding two digits and a carry is an **atomic add**. If the time for an atomic add is α, then the best we can do for addition is $\alpha n = \Theta(n)$. We cannot do any better for addition.

Now let us look at multiplication. Consider multiplication of two n digit numbers.

Example (Mult). Standard Multiplication example.

Table 7.1: Multiplication

		1	0	2
		2	5	7
		7	1	4
	5	1	0	
2	0	4		
2	6	2	1	4

Multiplying two digits is an **atomic multiply**. The time for an atomic multiply is μ.

To analyze the speed of the standard multiplication algorithm, we just have to count the number of atomic adds and atomic multiplies (ignoring carries).

For multiplies, we have n^2 atomic multiplies as every digit on the bottom is multiplied by every digit on the top. For additions, we sum by column, right to left: $0+2+4+6+\ldots+2(n-1)+2(n-1)+\ldots+8+6+4+2+0$. This is equal to

$$2\sum_{i=0}^{n-1} 2i = 4\sum_{i=0}^{n-1} i = 2n(n-1)$$

Therefore, the total time with respect to the atomic adds and multiplies is

$$n^2\mu + 2n(n-1)\alpha$$

Attempt 2: What if we try a recursive method to multiply? For two digit multiplication, notice that we have

$$(10a+b)(10c+d) = 10^2 ac + 10(ad+bc) + bd$$

In this case we are ignoring carries.

If we look at a 4 digit multiplication, we can notice that if we split up the 4 digits into 2 x 2 digits, we can treat (ac, bd, ad, bc) as two digit multiplications.

We can then recurse. If we are multiplying

$$\underbrace{y_{n-1}\cdots y_{n/2}}_{c}\,\underbrace{y_{n/2-1}\cdots y_0}_{d}$$

$$\underbrace{x_{n-1}\ldots x_{n/2}}_{a}\,\underbrace{x_{n/2-1}\cdots x_0}_{b}$$

We can see that

$$xy = 10^n ac + 10^{n/2}(ad+bc) + bd$$

Then we see that the recursive time to multiply is

$$M(n) = 4M\left(\frac{n}{2}\right) + A(n) + A(n)$$

Where $A(n)$ is the time for additions, as the inner addition takes one, and the outer two takes another. Therefore, we get the recurrence relation

$$M(n) = 4M\left(\frac{n}{2}\right) + 2\alpha n$$

Where $M(1) = \mu$.

Therefore, we get

$$\begin{aligned} M(n) &= 4M\left(\frac{n}{2}\right) + 2\alpha n \\ &= 4^2 M\left(\frac{n}{4}\right) + 4 \cdot 2\alpha\frac{n}{2} + 2\alpha n \\ &= 4^3 M\left(\frac{n}{8}\right) + 4^2 \cdot 2\alpha\frac{n}{4} + 4 \cdot 2\alpha\frac{n}{2} + 2\alpha n \\ &= 4^k M\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 4^i \left(2\alpha\frac{n}{2^i}\right) \\ &= 4^k M\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2\alpha 2^i \\ &= 4^k M\left(\frac{n}{2^k}\right) + 2\alpha n \sum_{i=0}^{k-1} 2^i \\ &= 4^k M\left(\frac{n}{2^k}\right) + 2\alpha n \left(2^k - 1\right) \end{aligned}$$

When $k = \lg n$, we get $M(1) = \mu$, so the result is

$$4^{\lg n}\mu + 2\alpha n\left(2^{\lg n} - 1\right)$$

which is also just

$$n^2\mu + 2\alpha n(n-1)$$

But this is also just n^2.

7.2 Karatsuba

We can use a sneaky idea. We know that if we're multiplying ab and cd, we get

$$(10a + b)(10c + d) = 10^2 ac + 10(ad + bc) + bd$$

Note that

$$(a + b)(c + d) = ac + (ad + bc) + bd$$

so then $ad + bc = (a + b)(c + d) - (ac + bd)$.

The middle expression can be calculated using two expressions we already have (ac), and (bd), plus one additional multiplication.

Remark. Throughout this time, we have been ignoring multiplies by 100 and 10. This is because in any base n system, we can perform the multiplies through left shifts by n digits, which is $\Theta(n)$ for the number of digits. Therefore, in this case it is constant.

The full product we have is

$$10^2ac + 10\left((a+b)(c+d) - (ac+bd)\right) + bd$$

Therefore, we can now recurse. Suppose x and y are both n digit numbers. Then

$$\underbrace{y_{n-1}\cdots y_{n/2}}_{c}\underbrace{y_{n/2-1}\cdots y_0}_{d}$$

$$\underbrace{x_{n-1}\cdots x_{n/2}}_{a}\underbrace{x_{n/2-1}\cdots x_0}_{b}$$

We can see that

$$xy = 10^n ac + 10^{n/2}((a+b)(c+d) - (ac+bd)) + bd$$

Therefore, we have

$$M(n) = 3M\left(\frac{n}{2}\right) + 2A\left(\frac{n}{2}\right) + 2A(n) + A(n)$$

We get the two $n/2$ additions from $a+b$ and $c+d$. The other two are from $ac+bd$ and the subtraction, and the final one is from the total sum.

Note. We are playing fast and loose with powers of 2, half sizes, and so on, to keep the explanation tidy and avoid floor and ceiling functions. Note that sometimes, $a+b$ and $c+d$ can become non atomic multiplications if they grow too large, so we are avoiding the in depth explanation here.

Remark. For the $a+b$ and $c+d$, if they are two digit (or over $n/2$ digit), we can apply Karatsuba again on the inside.

Remark. In the case where one number is one digit and the other has more, then it is $\Theta(n)$ to multiply. The actual implementation of Karatsuba's algorithm uses this as the base case.

Remark. If the numbers have differing numbers of digits, we split by the shorter one to guarantee that both of them can actually be split. We additionally split from the right hand side (the units digit) to ensure the left shifting works properly.

Algorithm 25 Karatsuba Multiplication

```
Require: $A, B$ are integers with $n$ digits in base $b$
 1: procedure KARATSUBA($A, B$)
 2:     $A_T \leftarrow A[n-1 : n/2]$
 3:     $A_B \leftarrow A[n/2-1 : 0]$
 4:     $B_T \leftarrow A[n-1 : n/2]$
 5:     $B_B \leftarrow A[n/2-1 : 0]$
 6:     $R_2 \leftarrow \text{Karatsuba}(A_T, B_T)$
 7:     $R_0 \leftarrow \text{Karatsuba}(A_B, B_B)$
 8:     $R_1 \leftarrow \text{Karatsuba}(A_T + A_B, B_T + B_B) - (R_2 + R_0)$
 9:     $R \leftarrow R_2 \cdot b^2 + R_1 \cdot b + R_0$        ▷ use left shifts for base multiplication
10:     return $R$
11: end procedure
```

We have

$$\begin{aligned}
M(n) &= 3M\left(\frac{n}{2}\right) + 4\alpha n \\
&= 3^2 M\left(\frac{n}{2^2}\right) + 3 \cdot 4\alpha\frac{n}{2} + 3\alpha n \\
&= 3^k M\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 3^i \left(4\alpha\frac{n}{2^i}\right) \\
&= 3^k M\left(\frac{n}{2^k}\right) + 4\alpha n \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i \\
&= 3^k M\left(\frac{n}{2^k}\right) + 4\alpha n \frac{1.5^n - 1}{1.5 - 1}
\end{aligned}$$

When $k = \lg n$, we get $M(1) = \mu$, so we get

$$\begin{aligned}
3^{\lg n}\mu + 4\alpha n \left(\frac{\frac{3^{\lg n}}{2^{\lg n}} - 1}{\frac{1}{2}}\right) &= 3^{\lg n}\mu + 8\alpha n \left(\frac{3^{\lg n}}{2^{\lg n}} - 1\right) \\
&= \mu n^{\lg 3} + 8\alpha n \left(\frac{n^{\lg 3}}{n} - 1\right) \\
&= \mu n^{\lg 3} + 8\alpha \left(n^{\lg 3} - n\right) \\
&= (\mu + 8\alpha)\, n^{\lg 3} - 8\alpha n \\
&\approx \Theta(n^{1.58})
\end{aligned}$$

We get a large constant factor, but smaller exponent than the standard algorithm.

Remark. Note that usually, the atomic value is actually base 2^{64} and not base 10. Karatsuba's only has a good speedup for very large multiplication. There is also a generalization on top of Karatsuba's algorithm, known as Toom-Cook. With Toom-Cook, we can multiply two three-digit numbers with 5 multiplies, giving $\Theta(n^{\log_3 5}) \approx \Theta(n^{1.46})$. In general, we can get $\Theta(n^c)$ for c arbitrarily close to 1, but with a larger constant term.

Remark. Fourier transform methods such as Strassen and Schonhage Straiten allow us to do multiplication in $\Theta(n \lg n)$ with a large constant factor as well.

7.3 Summation Approximations and Techniques

Say we have the sum

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Note that since this is $1+2+3+4+5\ldots+n-1+n$, we can bound it. In this case, we have a lower bound of

$$1+1+1+1+1\ldots+1+1=n$$

And an upper bound of

$$n+n+n+n+n\ldots+n+n=n^2$$

Therefore,

$$n \le \sum_{i=1}^{n} i \le n^2$$

We can also **split a sum**, like following.

$$\sum_{i=1}^{n} i = \sum_{i=1}^{\frac{n}{2}} i + \sum_{i=\frac{n}{2}+1}^{n} i$$

Now note that we can lower bound this instead with

$$1 \cdot \left(\frac{n}{2}\right) + \left(\frac{n}{2}+1\right)\left(\frac{n}{2}\right) = \frac{n^2}{4} + n$$

So instead we get

$$\frac{n^2}{4} + n \le \sum_{i=1}^{n} i \le n^2$$

Now let us consider

$$\sum_{i=1}^{n} \frac{1}{i}$$

We see that the lower bound is $\frac{1}{n} \cdot n = 1$, and the upper bound is $1 \cdot n = n$. However, we can improve our upper bound. Note the following:

$$\begin{aligned}\sum_{i=1}^{n} \frac{1}{i} &= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \dots \frac{1}{n} \\ &\leq \underbrace{\frac{1}{1}}_{1} + \underbrace{\frac{1}{2} + \frac{1}{2}}_{1} + \underbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}}_{1} + \frac{1}{8} + \dots\end{aligned}$$

But how many 1 s are there? We see that we have $1 + 2 + 4 + 8 + 16 \dots$ number of elements. If we have k ones, then we have

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1$$

elements. Therefore, since $n = 2^k - 1$, we have $k = \lg(n+1)$. So we have

$$1 \leq \sum_{i=1}^{n} \frac{1}{i} \leq \lg(n+1)$$

We can actually generalize these concepts.

Theorem 7 (Monotonically Increasing Bounds)**.** Let $f(i)$ be a function that can be extended to the real line and is monotonically increasing (nondecreasing). Then,

$$\int_{m-1}^{n} f(x)\,\mathrm{d}x \leq \sum_{i=m}^{n} f(i) \leq \int_{m}^{n+1} f(x)\,\mathrm{d}x$$

Proof 6. Intuitively, we can think of putting the sum as a series of rectangles. The upper bound arises when the boxes at i to $i+1$ have height $f(i)$, and the lower bound arises when the boxes at i to $i+1$ have height $f(i+1)$. □

Theorem 8 (Monotonically Decreasing Bounds)**.** Let $f(i)$ be a function that can be extended to the real line and is monotonically decreasing. Then,

$$\int_{m}^{n+1} f(x)\,\mathrm{d}x \leq \sum_{i=m}^{n} f(i) \leq \int_{m-1}^{n} f(x)\,\mathrm{d}x$$

7.4 Locality

Definition 21 (Temporal Locality)**.** Temporal locality is when if a memory location is referenced, then it is likely that the same location will be referenced again in the near future. In essence, we access one location multiple times over time, which allows specific memory locations to be put in a cache.

Definition 22 (Spatial Locality). Spatial locality is if a memory location is referenced at a particular time, then it is likely that **nearby** locations will be accessed in the near future.

We can see that a lot of our sorts have spatial locality. For example, in Bubble Sort, we go left to right, or in Merge Sort, we merge going left to right on two sublists. However, note that Heap Sort is NOT spatially local, as the indices of the children are twice as large as the index of the parent. Therefore, we have to jump and access memory at much farther indices.

7.5 Comparison Based Sorting Algorithms

Table 7.2: Comparison Based Sorting Algorithms

	Bubble	Insert	Select	Merge	Heap	Quick
Worst Cmp	$\frac{n^2}{2}$	$\frac{n^2}{2}$	$\frac{n^2}{2}$	$n\lg n$	"$n\lg n$"	$\frac{n^2}{2}$
Avg Cmp	$\frac{n^2}{2}$	$\frac{n^2}{4}$	$\frac{n^2}{2}$	$n\lg n$	$n\lg n$	$1.4n\lg n$
Worst Moves		$\frac{n^2}{2}$		?	"$n\lg n$"	?
Avg Moves		$\frac{n^2}{4}$		?	$n\lg n$	?
Worst Xchg	$\frac{n^2}{2}$		n			
Avg Xchg	$\frac{n^2}{2}$		n			
In place	Yes	Yes	Yes	Y/N	Yes	Yes
Spatial Loc	Yes	Yes	Yes	Yes	No	Yes

Chapter 8

Non Comparison-Based Sorting

8.1 Non Comparison Sorting

We have always been talking about comparison based sorting until know. This means that we compare a pair of values and depending on which one is greater we do a specific thing. For every comparison based algorithm, we can build a **decision tree** based on exactly what decision we make at what point in time.

Lemma 1. We have $\lg(n!) = \Theta(n \lg n)$

Proof 7. First we will prove Ω.

$$\begin{aligned}
\lg(n!) &= \lg n + \lg(n-1) + \ldots \lg 2 + \lg 1 \\
&= \left(\lg(n-0) + \lg(n-1) + \ldots \lg\left(n - \left\lfloor \frac{n}{2} \right\rfloor\right)\right) + \text{the rest} \\
&\geq \lg(n-0) + \lg(n-1) + \ldots \lg\left(n - \left\lfloor \frac{n}{2} \right\rfloor\right) \\
&\geq \lg \frac{n}{2} + \lg \frac{n}{2} + \ldots \lg \frac{n}{2} \\
&\geq \left(1 + \left\lfloor \frac{n}{2} \right\rfloor\right)\left(\lg \frac{n}{2}\right) \\
&\geq \left(1 + \left\lfloor \frac{n}{2} \right\rfloor\right)(\lg n - \lg 2) \\
&\geq \frac{1}{2} n (\lg n - 1) \\
&\geq \frac{1}{4} n \lg n \text{ if } n \geq 4
\end{aligned}$$

The last step isn't obvious, but

$$\frac{1}{2}n(\lg n - 1) \geq \frac{1}{4}n \lg n \implies \frac{1}{4}n \lg n \geq \frac{1}{2}n \implies \lg n \geq 2 \implies n \geq 4$$

Now for $\mathcal{O}$.

$$\lg n! = \lg n + \lg(n-1) + \ldots \lg 2 + \lg 1 \leq \lg n + \lg n + \lg n \ldots = n \lg n$$

Together, for $n \geq 4$, we have

$$\frac{1}{4}n \lg n \leq \lg n! \leq n \lg n$$

Fun fact: we also see that

$$\lg n! \leq n \lg n \leq 4 \lg n!$$

Which means $n \lg n \in \Theta(\lg n!)$. Therefore, since

$$\lg n! \in \Theta(n \lg n)$$

They are the exact same time complexity! □

Theorem 9 (Comparison Bounds). Any comparison sort requires, in the worst case, $\Omega(n \lg n)$ comparisons.

Proof 8. Any comparison sorting algorithm needs to manage $n!$ possible permutations of the list. Each permutation is a leaf in the decision tree, so the number of leaves must be $n!$. In general if h is the height of a tree, then the number of leaves is at most

$$n! \leq 2^h$$

Therefore, we have

$$h \geq \lg(n!)$$

Let d be the number of comparison based decisions necessary in the worst case. In the worst case, we follow the tree down the lowest leaf, so $d = h \geq \lg n!$. Therefore, by the previous lemma, we have that

$$d \geq \lg n! \geq \frac{1}{4}n \lg n = \Omega(n \lg n)$$

□

8.2 Counting Sort

If we have an array of (preferably small) nonnegative integers and we know the maximum, we can sort the array. The way this works is we create a new list, where each $C[i]$ counts the number of instances of i in the original array.

Definition 23 (Counting Sort)**.** We define counting sort as follows.

Algorithm 26 Counting Sort

```
Require: $A$ is a list of length $n$, where each value is an integer between 0 and $k-1$, inclusive.
 1: for $i = 0 \dots k-1$ do
 2:     $C[i] \leftarrow 0$
 3: end for
 4: for $j = 1 \dots n$ do
 5:     $C[A[j]] \leftarrow C[A[j]] + 1$
 6: end for
 7: $t \leftarrow 0$
 8: for $i = 0 \dots k-1$ do
 9:     for $j = 1 \dots C[i]$ do
10:         $t \leftarrow t + 1$
11:         $B[t] \leftarrow i$
12:     end for
13: end for
14: return $B$, which is a sorted version of $A$
```

Note. If A is given but we don't have k, then we can simply loop once through A to calculate k. This process is $\Theta(n)$.

Note that in all cases, we have that

- Initializing C: $\Theta(k)$
- Iterating through A : $\Theta(n)$
- Iterating through C and updating B: Since C has k values in it, and B has n values in it, the whole thing is $\Theta(k+n)$.

So the total time for best case, worst case, and average case is $\Theta(n+k)$.

Remark. The $n+k$ might be confusing, but we can think of it as a single variable. So $\Theta(n+k)$ means that if $n+k \geq m_0$, then

$$B \cdot (n+k) \leq T(n,k) \leq C \cdot (n+k)$$

Note. If k is a fixed constant and we let n vary then the time complexity is $\Theta(n)$.

Note. If k is not fixed but we can guarantee that $k \leq n$ then the time complexity is $\Theta(n)$.

> **Note.** Counting Sort is not in place, however this implementation is not stable.

Algorithm 27 Stable Counting Sort

```
Require: $A$ is a list of length $n$, where each value is an integer between 0 and $k-1$, inclusive.
 1: for $i = 0 \dots k-1$ do
 2:     $C[i] \leftarrow 0$
 3: end for
 4: for $j = 1 \dots n$ do
 5:     $C[A[j]] \leftarrow C[A[j]] + 1$
 6: end for
 7: for $i = 1 \dots k-1$ do
 8:     $C[i] \leftarrow C[i-1] + C[i]$
 9: end for
10: for $j = n \dots 1$ do
11:     $B[C[A[j]]] \leftarrow A[j]$
12:     $C[A[j]] \leftarrow C[A[j]] - 1$
13: end for
14: return $B$, which is a sorted version of $A$
```

Our new algorithm is still $\Theta(n+k)$. However, this time, we form the partial sums of the values in C. Then we go backwards through A, putting the value in $A[i]$ in the proper location of B by using C to determine the index.

What the partial sums represent, is that at each index i we have $C[i]$ is the number of values less than or equal to i in A. If we know $C[i]$, and we find 3 values of i, we can put them at $C[i]$, $C[i]-1$, $C[i]-2$. If we go backwards through A and do this, our algorithm is now stable.

8.3 Bucket Sort

The intuition behind bucket sort is to put each value into a "bucket" corresponding to its first digit. We then sort each bucket, and combine.

Definition 24 (Bucket Sort)**.** This is the algorithm for Bucket Sort.

Algorithm 28 Bucket Sort

```
Require: $A$ is a list of $n$ numbers uniformly distributed in range $[0, 1)$
 1: for $i = 0 \dots n-1$ do
 2:     bucket $B[i] \leftarrow \varnothing$
 3: end for
 4: for $i = 1 \dots n$ do
 5:     put $A[i]$ into bucket $B[\lfloor n \cdot A[i] \rfloor]$
 6: end for
 7: for $i = 0 \dots n-1$ do
 8:     BubbleSort bucket $B[i]$
 9: end for
10: concatenate buckets $B[0], B[1], \dots B[n-1]$
11: return concatenated buckets $B$
```

We see that

- The time to initialize the buckets is $\Theta(n)$
- The time to put the numbers in the buckets is $\Theta(n)$
- The time to sort the buckets is harder.
  - If there is one bad bucket of size N, we get $\Theta(n^2)$.
  - If there are $\sqrt{n}$ mediocre buckets of size $\sqrt{n}$, we get $\Theta(n^{\frac{3}{2}})$.
- The time to concatenate the buckets is $\Theta(n)$.

The average size of a bucket is 1. This is because there are n values and n buckets, so even if all are placed into 1 bucket, we still have $n/n = 1$ on average. The probability that a bucket is empty is

$$\frac{(n-1)^n}{n^n} = \left(\frac{n-1}{n}\right)^n = \left(1 - \frac{1}{n}\right)^n \sim \frac{1}{e} \approx 0.37$$

That means the average number of empty buckets is

$$n\left(1 - \frac{1}{n}\right)^n \sim \frac{n}{e} \sim 0.37n$$

What is the probability that a bucket has exactly 1 number?

$$n \cdot \frac{1}{n} \cdot \left(\frac{n-1}{n}\right)^{n-1} = \left(1 - \frac{1}{n}\right)^{n-1} \sim \frac{1}{e} \approx 0.37$$

What is the probability that a bucket has exactly two numbers? We need a pair of numbers each to go to the same bucket, and the rest of the values go to different buckets. Therefore, we choose the pair, and then place them in the same bucket, and then place the others.

$$\binom{n}{2}\frac{1}{n} \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-2} \sim \frac{1}{2e} \approx 0.18$$

The probability of 3 numbers follows similarly, giving

$$\binom{n}{3}\frac{1}{n^3}\left(1-\frac{1}{n}\right)^{n-3} \sim \frac{1}{6e} \approx 0.06$$

For 4, we have

$$\binom{n}{4}\frac{1}{n^4}\left(1-\frac{1}{n}\right)^{n-4} \sim \frac{1}{24e} \approx 0.015$$

The probability a bucket has at most 4 number is

$$\sim \frac{1}{e}+\frac{1}{e}+\frac{1}{2e}+\frac{1}{6e}+\frac{1}{24e} \approx 0.996$$

On average, how many numbers are in the largest bucket? The answer is

$$\sim \frac{\ln n}{\ln\ln n}$$

Therefore, on the average case, almost all buckets have few values, so most of them take constant time to sort. On top of that, no bucket has very many items, so no bucket has a really large sorting time.

The average time for all n sorts is totally $\Theta(n)$.

Proof 9. First, we note that all operations other than the sort are $\Theta(n)$. Therefore, we have

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1}\mathcal{O}(N_i^2)$$

Where N_i is the random variable denoting the number of elements placed in bucket $B[i]$. To analyze the average case running time of bucket sort, we have to compute the expected value of the running time, where we take the expectation over the input distribution. We get

$$\begin{aligned} E(T(n)) &= E\left(\Theta(n) + \sum_{i=0}^{n-1}\mathcal{O}(N_i^2)\right) \\ &= \Theta(n) + \sum_{i=0}^{n-1} E\left(\mathcal{O}(N_i^2)\right) \\ &= \Theta(n) + \sum_{i=0}^{n-1}\mathcal{O}(E(N_i^2)) \end{aligned}$$

We claim that $E(N_i^2) = 2 - \frac{1}{n}$.

Let X_{ij} be the indicator random variable that $A[j]$ falls into bucket i. Then we see

$$N_i = \sum_{n=1}^{n} X_{ij}$$

Then

$$\begin{aligned}
E(N_i^2) &= E\left(\left(\sum_{j=1}^{n} X_{ij}\right)^2\right) \\
&= E\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right) \\
&= E\left(\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\sum_{1\le k\le n, k\ne j} X_{ij}X_{ik}\right) \\
&= \sum_{j=1}^{n} E\left(X_{ij}^2\right) + \sum_{1\le j\le n}\sum_{1\le k\le n; k\ne j} E\left(X_{ij}X_{ik}\right)
\end{aligned}$$

We evaluate the summations separately. Indicator random variable X_{ij} is 1 with probability $\frac{1}{n}$ and 0 otherwise. Therefore,

$$E\left(X_{ij}^2\right) = 1^2\cdot\frac{1}{n} + 0^2\cdot\left(1-\frac{1}{n}\right) = \frac{1}{n}$$

When $k \ne j$, the variables X_{ij} and X_{ik} are independent, so

$$\begin{aligned}
E\left(X_{ij}X_{ik}\right) &= E\left(X_{ij}\right)E\left(X_{ik}\right) \\
&= \frac{1}{n}\cdot\frac{1}{n} \\
&= \frac{1}{n^2}
\end{aligned}$$

Therefore, we get

$$\begin{aligned}
E\left(N_i^2\right) &= \sum_{j=1}^{n}\frac{1}{n} + \sum_{1\le j\le n}\sum_{1\le k\le n, k\ne j}\frac{1}{n^2} \\
&= n\cdot\frac{1}{n} + n(n-1)\cdot\frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n}
\end{aligned}$$

Therefore, the average case running time for bucket sort is

$$\Theta(n) + n \cdot \mathcal{O}\left(2 - \frac{1}{n}\right) = \Theta(n)$$

□

However, why do we use bubble sort rather than an asymptotically faster sorting algorithm?

- It is required for the book's analysis: it is an ad hoc proof.
- Almost all buckets are small, so a quadratic sorting algorithm is better (May want a special purpose sorting algorithm for each small value of n.)

Note. In the general case, if we have numbers in the range $[0, R)$, we can instead put

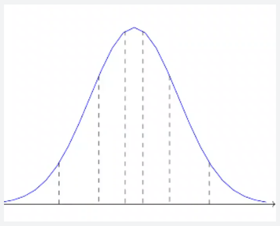
$$A[i] \in B\left[\left\lfloor \frac{n \cdot A[i]}{R} \right\rfloor\right]$$

If the numbers are in the range $[Q, R)$, we get

$$A[i] \in B\left[\left\lfloor \frac{n \cdot A[i]}{R - Q} \right\rfloor\right]$$

We can also sort integers this way.

Note. This typically works only for uniformly distributed bucket sort. If it is nonuniform, we take areas under the distribution of the values to be approximately equal, and split buckets based on that.



8.4 Radix Sort

Definition 25 (Radix Sort)**.** Given a set of integers of the form $x_d x_{d-1} \dots x_2 x_1$, the radix sort algorithm is as follows:

Algorithm 29 Radix Sort

```
Require: $A$ is a list of integers
1: for $i = 1 \dots d$ do
2:     stable sort $A$ using digit $i$
3: end for
4: return $A$, sorted.
```

Definition 26 (Radix)**.** The radix is the range of letters or digits.

Example (Radix)**.** For example, for numbers in base 10, the radix is 10.

Let us look at the analysis for radix sort. Let d be the number of digits. Suppose the time complexity of the underlying sort is $\Theta(f)$ for some function f. Then the radix sort time complexity is $\Theta(df)$.

For example, say we use counting sort as the underlying sort. We know that counting sort is $\Theta(N+k)$. In this case, if R is the radix, then the counting sort has time $\Theta(N+R)$. Therefore, our radix sort has time $\Theta(d(N+R))$.

How can we optimize our time complexity based on what we are sorting? Let S be the size or "range" of one value we are sorting. This means every number i falls in $0 \dots S$ inclusive. Then

$$S = R^d$$

Then we note that

$$\log S = D \log R \implies D = \frac{\log S}{\log R}$$

Therefore, the time complexity for Radix Sort is now

$$\Theta\left(\frac{\log S}{\log R}(N+R)\right)$$

Simplifying this, we get

$$\begin{aligned}\Theta\left(\frac{\log S}{\log R}(N+R)\right) &= c\frac{\log S}{\log R}(aN+bR)\\ &= bc\frac{\log S}{\log R}\left(\frac{a}{b}N+R\right)\\ &= \gamma\frac{\log S}{\log R}(\alpha N+R)\end{aligned}$$

Since γ and $\log S$ are constants, we need to minimize

$$\frac{\alpha N + R}{\ln R}$$

Taking the derivative and setting it equal to 0, we get

$$R = \frac{\alpha N}{\ln R - 1}$$

We can see that $R \sim \frac{\alpha N}{\ln N}$.

$$\frac{\alpha N}{\ln N} \sim \frac{\alpha N}{\ln\left(\frac{\alpha N}{\ln N}\right) - 1} \sim \frac{\alpha N}{\ln N - \ln\ln N} \sim \frac{\alpha N}{\ln N}$$

Substituting this in for R, we get

$$\Theta\left(\frac{N \log S}{\log N}\right)$$

If we are doing this in binary, then we get

$$\Theta\left(\frac{N \lg S}{\lg N}\right)$$

Note. If we are sorting different length numbers, we pad on the left with 0s. If we are alphabetizing different length words, then we pad on the right with a _ blank.

Note. While we analyzed using Θ, we can always substitute in $\mathcal{O}$ or Ω and also substitute in worst, best, or average case as needed, provided we understand the details.

Example (Binary). Suppose our list contains n integers between 0 and 255 inclusive. Let us use base $b = 2$ so that we need $d = 8$ digits. If we use Radix Sort with Counting Sort, we get $\Theta(8(n+2)) = \Theta(n)$.

Example (Size). Suppose our list contains n integers between 0 and $2^n - 1$ inclusive. As the list gets longer, so can the values. Let us use $b = 2$ so we need $d = n$ digits. If we use Radix Sort and Counting Sort, we get $\Theta(n(n+2)) = \Theta(n^2)$. In this case, it is better to use something like Quick Sort with average case $\Theta(n \lg n)$.

Example. Suppose our list contains n integers between 0 and $n-1$ inclusive. If we choose $b = n$, we only need $d = 1$ digit. Then with Radix Sort and Counting Sort we have $\Theta(1(n+n)) = \Theta(n)$. However, note in this case that we are only doing a single iteration of counting sort.

Remark. The auxiliary space is dependent upon the underlying sort. Radix Sort is in place only if the underlying sort is as well. However, we know that Radix Sort is stable since we have chosen a sort which is stable.

Chapter 9

Selection

9.1 Selection

Definition 27 (Selection)**.** The selection problem: In a list of n values, find the k-th smallest.

How do we solve this? We can perform a quicksort-like algorithm, but only focus on the location of our k-th smallest element. We can partition based on the median, and if k falls in the lower partition, we continue. If k falls in the larger partition, the new k becomes $k-(q-p+1)$. Therefore, we get

Algorithm 30 Naive Selection

```
Require: $A$ is a list which contains $A[p \dots r]$.
function Selection($A, p, r, k$)
    $s \leftarrow$approximate_median($A, p, r$)
    $q \leftarrow$partition($A, p, r, s$)
    if $k < q - p + 1$ then
        return Selection($A, p, q - 1, k$)
    else if $k > q - p + 1$ then
        return Selection($A, q + 1, r, k - (q - p + 1)$)
    else
        return $q$
    end if
end function
```

We can do this better by keeping k fixed and just considering slices over $p \to r$ on top of the list.

Algorithm 31 Naive Selection 2

```
Require: $A$ is a list of length $n$.
procedure Selection($A, p, r, k$)
    $s \leftarrow$approximate_median($A, p, r$)
    $q \leftarrow$partition($A, p, r, s$)
    if $k < q$ then
        Selection($A, p, q - 1, k$)
    else if $k > q$ then
        Selection($A, q + 1, r, k$)
    end if
    return $A[k]$
end procedure
```

Now our k-th smallest element will be in index k, but no guarantees about the other elements. This function can also be made nonrecursive, in the same vein as binary search.

Algorithm 32 Naive Nonrecursive Selection

```
Require: $A$ is a list of length $n$.
$q \leftarrow -1$
while $q \neq k$ do
    $s \leftarrow$approximate_median($A, p, r$)
    $q \leftarrow$partition($A, p, r, s$)
    if $k < q$ then
        $r \leftarrow q - 1$
    else if $k > q$ then
        $p \leftarrow q + 1$
    end if
end while
return $A[k]$
```

- In the worst case, note that this is quadratic as we might pivot right on the edge, which takes $\frac{n}{2}$ recursive calls with $n - 1$ time to partition. Therefore, we get $\frac{n(n-1)}{2}$.
- In the best case, for these algorithm, we split into equal sizes. The partition takes $n - 1$ time, so we get
$$T(n) = T\left(\frac{n}{2}\right) + n - 1$$
where $T(1) = 0$. This is approximately equal to $2n$.
- In the average case, we can make a simplifying assumption that the average case must be at the $\frac{1}{4}$ mark. Therefore, we get
$$T(n) \leq T\left(\frac{3n}{4}\right) + n - 1$$

Note that we are making a pessimistic assumption that we are always on the worse side. Therefore it is less than or equal to. This approximately is equal to

$$T(n) = 4n$$

By the geometric series.

We can also do a more exact bound on the average case instead of splitting it at $\frac{1}{4}$, but still under the assumption that we end up on the larger side. Then we get

$$T(n) = \sum_{q=1}^{n-1} \frac{1}{n} T(\max(q-1, n-q)) + n - 1$$

Note that we can rewrite this as

$$T(n) = \frac{1}{n} 2 \sum_{q=\frac{n}{2}}^{n-1} T(q) + n - 1$$

If we guess that $T(n) \leq an$, we can now solve for this recurrence, as by induction we have

$$T(n) \leq \frac{2}{n} \sum_{q=\frac{n}{2}}^{n-1} aq + n - 1$$

Now chugging out the algebra we get $a \geq 4$.

9.2 Selection in Worst Case Linear Time

We can actually do selection in worst case linear time. The intuition is as follows:

- Split our array into a 5 by $\frac{n}{5}$ grid.
- Find the median of each column.
- Rearrange each column (with 5 elements) to put the median in the center, small above, and large below.
- Now find the median of the medians of the columns.
- Sort each column by its median, putting the columns with smaller medians to the left and larger to the right.
- Finally, we do a normal selection, but partition using this median of medians as a pivot.

Intuitively, notice that our partition will now guarantee $\sim \frac{1}{4}$ of the values are less and $\sim \frac{1}{4}$ of the values are greater. If we split our rectangular grid into 4 regions, the top left will be less and the bottom right will be greater. More precisely, we know $\frac{3n}{10}$ is guaranteed to be less than or greater than the pivot.

- Finding the median of each column takes $\frac{n}{5}\frac{5\cdot(5-1)}{2}$ for sorting, which gives us $2n$.

- Finding the median of medians is a recursive call to selection, giving $T\left(\frac{n}{5}\right)$.
- Pivoting takes $n-1$.
- The recursive call on one side is guaranteed to be less than or equal to $T\left(\frac{7n}{10}\right)$.

In total, we have

$$T(n) \leq 2n + T\left(\frac{n}{5}\right) + n - 1 + T\left(\frac{7n}{10}\right)$$

If we make the guess that $T(n) \leq an$, we get that $T(n) \leq 30n$.

Chapter 10

Graphs

10.1 Graphs

Definition 28 (Graph)**.** A graph is a tuple (V, E) where V is a set of **vertices** and E is a set of ordered tuples (v_0, v_1) called **edges** where $v_0, v_1 \in V$.

Definition 29 (Undirected Graph)**.** An undirected graph is a graph where if $(v_0, v_1) \in E$, $(v_1, v_0) \in E$. In an undirected graph, (v_0, v_1) and (v_1, v_0) are considered a single edge.

Definition 30 (Adjacency Matrix)**.** We can represent a graph as an adjacency matrix, Where each row corresponds to an incident vertex and each column corresponds to the end vertex. Namely, $A[i, j] = 1$ if $(v_i, v_j) \in E$ and $A[i, j] = 0$ otherwise.

Note that the adjacency matrix needs $\Theta(n^2)$ storage. However, since it only contains 0's and 1's, it can be packed into words.

Definition 31 (Adjacency List)**.** An adjacency list is a list of linked lists such that the linked list at $A[i]$ contains every single vertex reachable from i with one edge.

Note that the adjacency list needs $\Theta(m + n)$ storage where m is the number of edges and n is the number of vertices.

To figure out if there is an edge from x to y, we see that:

- Matrix representation: $\Theta(1)$
- List representation: In the worst case, we get $\Theta(n)$

To get all the edges in the graph G, we see that

- Matrix representation: $\Theta(n^2)$
- List representation: $\Theta(n + m)$

Note. The reason the list representation is $n + m$ is that there are m values in the list total, as there are m edges. We also have to go through all the vertices once by traversing through the list, so that contributes the n. If the graph is incredibly sparse, note that the traversal will dominate over reading out the edges.

Definition 32 (Eulerian Cycle)**.** An Eulerian Cycle is a cycle which uses each graph edge exactly once. Formally, it is a sequence of edges

$$(v_0, v_1), (v_1, v_2), \dots (v_i, v_{i+1}), \dots (v_n, v_0)$$

Where each edge shows up exactly once.

Definition 33 (Degree)**.** The degree of a vertex v is the number of edges containing v.

Theorem 10 (Euler)**.** An undirected, connected graph $G = (V, E)$ has an Eulerian Cycle if and only if every vertex has even degree.

Proof 10. If G contains an Eulerian Cycle, let us traverse this cycle. Every time we visit a vertex, we come in on an incident edge and leave through a different edge. Therefore, the trail removes 2 from the degree of the vertex. So, in the end, each vertex must have a degree of $2k$ for some k, meaning they are all even.

Conversely, we can proceed by induction. Assume each vertex has even degree. Since G is connected, then it contains some circuit C. Consider the graph X' when we remove all the edges in the circuit C. Then X' may be disconnected but each vertex still has even degree. Hence we can use induction to each component of X' to get a closed Eulerian trail for each component. Each component has at least one vertex in common with C. Once we reach this common vertex, traverse that component of X' and return back to that vertex. Therefore, by induction, our graph is Eulerian. □

An algorithm then follows from the proof. Given an adjacency list, we start at the first list and the first value in that list. We then go to the list with that index, and take off the first value. We continue in this fashion taking off the first element and jumping to the list of that vertex until we have formed a cycle that loops back. We then traverse this cycle until we find a vertex with a nonzero number of elements in the list, and repeat there, joining that cycle to our original cycle.

Algorithm 33

```
Require: $A$ is an adjacency list
function EulerianCycle(A)
    $C \leftarrow$ empty array
    while Adjacency list is not empty do
        $S \leftarrow$ first nonempty adjacency list index
        $C_S \leftarrow$ empty array
        $C_S$.append($S$)
        $i \leftarrow A[S]$.data
        while $i \neq S$ do
            $C_S$.append($i$)
            Delete edge $i$ from adjacency list
            $i \leftarrow A[i]$.data
        end while
        Splice together the two cycles $C_S$ and $C$ based on common index
    end while
end function
```

Note that this runs in $\Theta(m+n)$. Effectively it runs in $\Theta(m)$ since the inner while loop is actually constant.

10.2 Trees

Definition 34 (Tree)**.** A **tree** is a graph which is undirected, connected, and contains no cycles (acyclic).

Definition 35 (Spanning Tree)**.** A **spanning tree** of a connected graph G is a subset of the graph which is a tree and includes all vertices of G.

Definition 36 (Minimum Spanning Tree)**.** A **minimum spanning tree** of a connected weighted graph G is a spanning tree of G with the smallest possible edge weight sum.

10.3 Kruskal's Algorithm

Kruskal's algorithm provides a method to find a minimum spanning tree of a weighted graph. We start with a graph with no edges and all vertices. We then add the smallest edge possible into the graph without creating a cycle, and continue in this fashion until we reach the end of the set of edges. At this point, we have a MST of the graph.

Algorithm 34 Kruskal's Algorithm

```
function KRUSKAL($G$)
    Sort the edges so that $e_1 \leq e_2 \leq e_3, \ldots \leq e_n$
    $T \leftarrow \varnothing$
    for $i = 1 \ldots m$ do
        Put $e_i$ in $T$
        if $T$ has a cycle then
            Remove $e_i$
        end if
    end for
    return $T$
end function
```

10.4 Prim's Algorithm

The premise behind Prim's algorithm is to consider a subgraph G', which is initally a single vertex. As we progress, we add the lowest cost edge which connects one vertex in the subgraph to a vertex outside the subgraph. We then add that vertex into the subgraph, and continue.

Algorithm 35 Prim's Algorithm

```
function PRIM($G$, $s$)
    $d$ is a list of size $n$
    $p$ is a list of size $n$
    $T$ is the initial minimum spanning tree
    for $i = 1 \ldots n$ do
        $d[v_i] \leftarrow \infty$
    end for
    $O \leftarrow G$
    $d[s] \leftarrow 0$
    while $O \neq \varnothing$ do
        $u \leftarrow \min(O)$ (with respect to distance $d$)
        for each $v$ adjacent to $u$ do
            if $v \in O$ and weight of $(u, v) < d[v]$ then
                $d[v] \leftarrow$ weight of $(u, v)$
                $p[v] \leftarrow u$
            end if
        end for
        Add $u$ and the edge $(p[u], u)$ to $T$
    end while
end function
```

Each iteration of the while loop while the "outside" is not empty removes 1 vertex, so the loop is $\mathcal{O}(n)$. However, inside that, extraction of the minimum takes $\mathcal{O}(n)$, and going through adjacent vertices is worst case $\mathcal{O}(n)$, so we get $\mathcal{O}(n(n+n)) = \mathcal{O}(n^2)$.

We could instead of using a distance array and extracting the minimum, use a heap, keeping distances sorted.

Algorithm 36 Heap Prim's

```
function HeapPrim($G$, $W$)
    for $i = 1 \ldots n$ do
        MinHeap[$i$] $\leftarrow i$
        WhereInHeap[$i$] $\leftarrow i$
        $d[i] \leftarrow \infty$
        outside[$i$] $\leftarrow T$
        parent[$i$] $\leftarrow NIL$
    end for
    $d[i] \leftarrow 0$
    for $i = n \ldots 1$ do
        $u$ ←MinHeap[1]
        MinHeap[1] ←MinHeap[$i$]
        WhereInHeap[MinHeap[1]] ← 1
        SiftDown($1, i-1$)                              ▷ Keeping track of WhereInHeap
        for each $v$ ∈adj[$u$] do
            if $v$ ∈outside and $W[u, v] < d[v]$ then
                $d[v] \leftarrow W[u, v]$
                prev[$v$] $\leftarrow u$
                SiftUp(WhereInHeap[$v$] )               ▷ Keeping track of WhereInHeap
            end if
        end for
    end for
end function
```

Here, we see that we first loop through n times. We then perform a sift, which is a heap operation. Afterwards, we loop through the adjacent edges and perform a sift. Therefore, amortizing the edge loop, we get $\mathcal{O}(n \log n) + \mathcal{O}(m \log n) = \mathcal{O}((m + n) \log n)$.

10.5 Dijkstra's Algorithm

Dijkstra's algorithm is used to find the shortest paths between nodes in a connected, positive weighted graph. There are three types of shortest path problems:

- Single source, single sink
- Single source
- All pairs

It is very difficult to solve the single source single sink problem without solving the single source problem for all vertices along the way.

Idea behind Dijkstra's algorithm: We keep a list of distances to every vertex. Starting from a vertex, we consider all adjacent vertices and update their distances. We then look at the shortest one not covered, and continue, looking at its adjacent vertices.

Algorithm 37 Dijkstra's Algorithm

```
function Dijkstra($G, W, s$)
    for vertices $v \in V[G]$ do
        $d[v] \leftarrow \infty$
        $\pi[v] \leftarrow NIL$
    end for
    $O \leftarrow V[G]$
    $d[s] \leftarrow 0$
    while $O \neq \varnothing$ do
        $u \leftarrow$ PopMin($O$)                ▷ With respect to $d$
        for all $y$ adjacent to $u$ do
            if $y \in O$ and $d[u] + W[u, v] < d[v]$ then
                $d[v] \leftarrow d[u] + W[u, v]$
                $\pi[v] \leftarrow u$
            end if
        end for
    end while
    return $\pi$
end function
```

This algorithm is similar to the breadth first search (and Prim's algorithm). We have an expanding shell of visited nodes from the source node, and updating the distances as we go.

In the end, we are left with an array, each index corresponding to a vertex. At each value we have its parent, so to find the smallest path to a vertex from s, we find that vertex in the list, find its parent, and continue by checking the parent in the list until we reach s.

Note the similarity to Prim's algorithm. Here we can in addition either use an adjacency matrix or a heap to pop the minimum value from the vertices which have not been visited yet.

With an adjacency matrix, this has time complexity $\mathcal{O}(n^2)$, and with a heap, this has time complexity $\mathcal{O}((n + m) \log n)$ (similar to Prim's algorithm).

Chapter 11

Complexity Theory

11.1 NP Completeness

We typically say a problem is "easy" if it can be solved in at most polynomial time, and "hard" if it requires at least exponential time to solve. Now this does not necessarily mean if a problem is "easy" it can be solved: Imagine an algorithm which takes $\Theta(n^{100})$ time. And conversely, we could have an algorithm which takes $\Theta(1.001^n)$ time. And what do we consider $\Theta(n^{\lg n})$?

Now, typically "natural problems" in polynomial time tend to have small polynomial degree, and "natural problems" tend to have exponential times with a large base (such as 2 or 3). And rarely there are problems between polynomial time and exponential time.

It's not even clear what a computer is, or even how to measure "time". We could have parallel computers, there is cost to access memory, and so on. In the next few sections, we will discuss the concept of NP-completeness, and determining the difficulty of problems, along with reductions between problems.

11.2 Decision and Optimization Problems

Definition 37 (Decision Problem)**.** A decision problem is a Yes/No question.

Example.

- Does a graph G have an Eulerian cycle?
- Does a Boolean formula $\mathcal{F}$ have a satisfying assignment?
- Is an integer n prime?

Definition 38 (Optimization Problem)**.** An optimization problem is finding the "best" solution to a problem.

Example.

- Find an optimal traveling salesman tour of weighted graph G.
- Find a Eulerian cycle in a graph G.
- Find a satisfying assignment for a Boolean formula $\mathcal{F}$
- Factor an integer n into primes.

Optimization (or search) problems have analogous decision problems.

- Eulerian cycle:
  - Find an Eulerian cycle in graph G.
  - Does graph G have a Eulerian cycle?
- Satisfiability:
  - Find a satisfying assignment for Boolean formula $\mathcal{F}$.
  - Is Boolean formula $\mathcal{F}$ satisfiable?
- Coloring:
  - Find a minimum coloring of undirected graph G.
  - Given undirected graph G and bound k, does G have a coloring with k or fewer colors?
- Factoring:
  - Factor integer n.
  - Given integer n and bound b, does n have a prime factor less than or equal to b?

Definition 39 (Analogous Problems)**.** A decision problem and an optimization problem are analogous if they are equivalent up to polynomial time.

Definition 40 (Class P)**.** The class **P** consists of the decision problems which are solvable in polynomial time.

Example.
The following are examples of decision problems in P.

- Value of Boolean formula
- 2-SAT
- Shortest path
- Eulerian cycle

- Minimum spanning tree

Definition 41 (Class NP)**.** The class **NP** consists of the decision problems where YES answers are verifiable in polynomial time, with a "witness" or "certificate".

Example.

- Formula satisfiability
- 3-SAT
- Hamiltonian cycle
- Traveling salesman problem
- Knapsack problem

We know that $\mathbf{P} \subseteq \mathbf{NP}$. However, one of the major unsolved problems is if $\mathbf{P} = \mathbf{NP}$ or not.

Theorem 11 (Cook-Levin)**.** Formula satisfiablility is solvable in polynomial time if and only if every problem in **NP** is solvable in polynomial time.

Definition 42 (NP-complete)**.** A problem is NP-complete if if the problem is solvable in polynomial time, then every problem in NP is solvable in polynomial time.

Clearly we see that Formula satisfiability is NP-complete. There are quite a few more NP-complete problems:

- 3-SAT
- Hamiltonian cycle
- Traveling salesman problem
- Knapsack problem
- Many more!

NP is partitioned into

- P: The "easy" problems.
- NP-complete: The "hard" problems.
- NPI (NP Intermediate): The "intermediate" problems.

Most "natural problems" are either in P or NP-complete.

11.3 Reductions

We will start this section with an example reduction.

Definition 43 (Connected)**.** An undirected graph G is connected if there is a path between every pair of vertices.

Definition 44 (Connected Component)**.** A connected component of an undirected graph G is a maximal connected subgraph of G.

Definition 45 (k-component Graph)**.** A k-component graph G is an undirected graph that consists of exactly k connected components.

Note that a connected graph is a 1-component graph.

Definition 46 (Coloring Problem)**.** The coloring problem is given a graph G, color G with the minimum number of colors.

Definition 47 (Decision Coloring Problem)**.** The decision version of the coloring problem is, given a graph G and an integer k, can G be colored with k or fewer colors?

Theorem 12. 1-component graph coloring is in **P** if and only if 2-component graph coloring is in **P**.

Proof 11. (1 to 2). Assume that 1 component graph coloring can be solved in polynomial time. Say a 2 component graph consists of components A and B.

Algorithm 38 2 component coloring

```
1: $C \leftarrow A + B + (a_i, b_i)$        ▷ Insert edge between $A$, $B$)
2: $c \leftarrow$OneColor($C$)
3: $c$
```

Here we just check if A is k-colorable and then check if B is k-colorable by inserting an edge between A and B. Note that if solving 1 component decision takes $\Theta(n^k)$, then this 2 coloring algorithm takes $\Theta(n^k)$ which is polynomial time.

(2 to 1). On the other hand, assume 2 component graph coloring can be solved in polynomial time.

Algorithm 39 1 component coloring

```
1: $G' \leftarrow G \cup \{v\}$        ▷ Add isolated vertex to graph
2: $c \leftarrow$TwoColor$(G')$
3: $c$
```

Note that this tells us whether a k coloring exists for the graph G, since a singular isolated vertex will always have a k coloring. This algorithm takes $\Theta(n^k)$ if checking two components takes $\Theta(n^k)$. □

Note that we must only use one call to the "oracle" we have access to within our algorithm, and we must be doing this for decision problems only. The standard way to write what we just did is

$$\text{2-component coloring} \leq_{\mathbf{P}} \text{1-component coloring}$$

$$\text{1-component coloring} \leq_{\mathbf{P}} \text{2-component coloring}$$

Theorem 13 (Formula and Circuit)**.** Formula SAT and Circuit SAT are equivalent.

$$\text{Formula SAT} \leq_{\mathbf{P}} \text{Circuit SAT}$$

$$\text{Circuit SAT} \leq_{\mathbf{P}} \text{Formula SAT}$$

Proof 12. (Informal).

(First statement). We can always construct a circuit for any formula F and solve that.

(Second). We must now convert a circuit into a formula. **We then have to convert it to conjunctive normal form**. Finally we can solve that formula.

For each wire of the circuit, we introduce a boolean variable x_v. Now we must construct a formula based on the gates of the circuit.

For a NOT gate, let x_u be the input and x_v be the output. We then add the clauses

$$(x_u \vee x_v) \wedge (\overline{x_u} \vee \overline{x_v})$$

We see that this statement guarantees x_v must be $\overline{x_u}$.

For an OR gate, let x_u and x_w be the inputs and x_v be the output. Then we can add the clauses

$$(x_v \vee \overline{x_u}) \wedge (x_v \vee \overline{x_w}) \wedge (\overline{x_v} \vee x_u \vee x_w)$$

For an AND gate, let x_u and x_w be the inputs and x_v be the output. Then add the clauses

$$(x_v \vee \overline{x_u} \vee \overline{x_w}) \wedge (\overline{x_v} \vee x_u) \wedge (\overline{x_v} \vee x_w)$$

Now for input wires we want 0, we add the statement $\overline{x_i}$, and for input wires we want 1, we add x_i. For an unknown input, we add nothing.

To force this to be 3-SAT, we add in auxiliary variables and force each of those variables to be 0 through a combination of 3-SAT conjunctions. Then we can pad variables

$$x \mapsto (x \vee a_1 \vee a_2)$$

□

Corollary. Circuit SAT is NP-complete.

Proof 13. By Cook Levin, Formula SAT is NP-complete. Therefore, Circuit SAT is as well. □

To show a problem A is NP-complete:

- Show that A is in NP.
- For some NP-complete problem B, show

$$B \leq_{\mathbf{P}} A$$

The second requirement is showing that our problem A is **NP-Hard**, or at least as hard as NP problems.

Example (Coloring)**.** The decision version of the coloring problem states: Given an undirected graph G, and an integer k, can the vertices of G be colored with at most k colors so that no two neighboring vertices have the same color?

Theorem 14. Coloring is in NP.

Proof 14. Assume the graph is represented by an $n \times n$ adjacency matrix $A[i, j]$. Assume that the k colors are represented by integers from $\{1, 2, \dots, k\}$. The witness is an assignment of colors to vertices (ie, an array color$[i]$).

To verify, we loop through each vertex and check all adjacent colors. If none of them are the same, then that vertex passes. We see that this takes $\mathcal{O}(n^2)$, and so is in NP. □

Theorem 15. SAT is in NP.

Proof 15. We have a certificate of assignments of TRUE's and FALSE's to variables. We substitute the assignments into the formula, and reduce the formula. If the value is TRUE, it is verified. Therefore, the time to verify is $\mathcal{O}(n)$ for n variables. □

Theorem 16. Coloring and Satisfiability are NP-complete.

Theorem 17. If a decision problem A is in P, then its complement $\overline{A}$ is in P.

Definition 48 (co-NP)**.** If a problem A is in NP, then its complement $\overline{A}$ is in co-NP.

Note that

- P $\subseteq$ NP
- P $\subseteq$ co-NP

The true structure of NP and co-NP is unknown. This is where the (quite famous) problem of $P \neq NP$ comes in. This helps us understand the structure of NP. Some questions include

- P = NP $\cap$ co-NP?
- NP = co-NP?
- P = NP?

Example (Factoring)**.** Factoring: Given integer N and bound B, does n have a prime factor less than or equal to B?

- In NP.
- In co-NP (witness is the factorization of N).

Example (Graph Isomorphism)**.** Given two graphs G and H, are they isomorphic?

- In NP.
- Not known to be in co-NP.
- In **probabilistic** co-NP.

There are many more complexity classes.

Definition 49 (PSPACE)**.** PSPACE is the set of all decision problems that can be solved by Turing machines using $\mathcal{O}(n^k)$ space.

One unsolved problem is whether P=PSPACE.

Definition 50 (EXPTIME)**.** The set of all decision problems that are solvable in exponential time.

Definition 51 (EXPSPACE)**.** The set of all decision problems solvable in exponential space.

Example (Known Heirarchy). $P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq EXPSPACE$

11.4 Decision and Optimization Equivalence

If we have an optimization problem, typically we can convert it to an equivalent decision problem. They should be equivalent up to optimization time.

Example (Coloring).
(Decision version). Given an undirected graph G and an integer k, can the vertices of G be colored with at most k colors so that no two neighboring vertices have the same color?
(Optimization version). Given an undirected graph G, find the minimum coloring of the graph.

Theorem 18. The optimization coloring problem and the decision coloring problem are equivalent up to polynomial time.

Proof 16. (Opt $\implies$ Dec). Assume optimization version of coloring is in P with time $\mathcal{O}(n^r)$. Consider the following algorithm:

Algorithm 40 Dec from Opt

```
function DEC($G, k$)
    $G, C \leftarrow$Opt($G$)
    $K \leftarrow$Count colors in $C$
    if $k < K$ then
        return False
    else
        return True
    end if
end function
```

We see that the time for this decision version is $\mathcal{O}(n^r) + \mathcal{O}(n) = \mathcal{O}(n^r)$.
(Dec $\implies$ Opt). Assume decision version of coloring is in P with time $\mathcal{O}(n^r)$.

Algorithm 41 Opt from Dec

```
function OPT($G$)
    $b \leftarrow 1$
    for $i = 1 \dots n$ do
        if Dec$(G, i)$ = False then
            $b \leftarrow i$
        end if
    end for
    Create $K_b$ (complete graph with $b$ vertices, each colored different)
    for $i = 1 \dots n$ do
        for $j = 1 \dots b$ do
            $G' \leftarrow$ Connect vertex $i$ to every vertex in $K_b$ except vertex $j$.
            if Dec$(G', b)$ then
                break
            end if
        end for
        color[i]$\leftarrow j$
    end for
end function
```

The intuition behind this is that we find the boundary with the number of colors possible. We then go through the vertices and try to color them with one of the colors, and check whether it is still possible. To "color" one of the colors, we connect a 5 clique where all vertices are connected but one.

Note that this is polynomial time. Therefore, the optimization and decision version are equivalent. □

Chapter 12

Miscellaneous Data Structures

12.1 Bloom Filters

So far, all the data structures covered have been exact data structures. For example, arrays, graphs, heaps, and so on. However, Bloom filters are a probabilistic data structure, meaning that it sacrifices some precision on data stored for speed.

Bloom filters are a data structure in which searching and insertion is very fast, but deletion is impossible. If a key has been stored, then searching will tell us that it has been stored. However, even if a key is not in the Bloom filter, there is a slight chance that searching tells us it is in the filter. In other words, there can be false positives, but no false negatives.

Definition 52 (Bloom Filter)**.** Given a set S, a bloom filter is a bit array B with m bits, and a set of k hash functions $h_1, \dots h_k$ such that $h_i : S \to \mathbb{Z}_m$. The Bloom filter consists of two operations: insertion and search.

- (Insertion). To insert a key in S, we set

$$B[h_1(x)] = \ldots = B[h_k(x)] = 1$$

- (Search). To search for a key, we check if

$$B[h_1(x)] = B[h_2(x)] = \ldots B[h_k(x)] = 1$$

  If its true, return true. Else, return false.

We can see that if we insert multiple elements, their hash functions could collide.

Example (Insertion)**.** Let $m = 20$, and we have the following hash functions:

$$h_1(x) = x \pmod{20}$$

$$h_2(x) = 3x \pmod{20}$$
$$h_3(x) = 7x \pmod{20}$$

Our set of keys is $\mathbb{Z}_{10}$.

Our bit array starts as

$$B = 00000000000000000000$$

After inserting $x = 1$, we get

$$B = 01010001000000000000$$

Inserting $x = 4$ we get

$$B = 01011001100010000000$$

Finally, inserting $x = 7$, we get

$$B = 01011001110010000000$$

Note that in the last insertion, the bits 7 and 1 were already set.

Note. Notice that it is possible for a key x which was never inserted that the bits $B[h_1(x)] \dots B[h_k(x)]$ were all set to 1 by other keys.

Theorem 19 (False Positive with Independence)**.** Let B be a bloom filter with m bits and k hashes, and say there are n elements within the bloom filter. Then the probability of a false positive with bitwise independence is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k$$

Proof 17. The probability that a bit is not set to 1 by a hash function is

$$1 - \frac{1}{m}$$

For k hash functions, the probability that a bit is not set by any of them is

$$\left(1 - \frac{1}{m}\right)^k$$

With n inserted items, the probability a single bit is not set is

$$\left(1 - \frac{1}{m}\right)^{nk}$$

The probability it is set is then

$$1 - \left(1 - \frac{1}{m}\right)^{nk}$$

We get a false positive if all k of the bits searched for are set, so the solution is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k}$$

□

However, here we are assuming the events $B[h_1(x)] = 1$ and $B[h_2(x)] = 1$ are independent.

Theorem 20 (General Formula for False Positives). Let B be a bloom filter with m bits and k hashes, and say there are n elements within the bloom filter. Then the probability of a false positive is

$$\frac{1}{m^{k(n+1)}} \sum_{i=1}^{m} i^k i! S(kn, i) \binom{m}{i}$$

where $S(n, k)$ is the Stirling number of the second kind.

Proof 18. For any subset $I \subseteq \{1, \ldots, m\}$, let E_I be the event that that specific subset of bits is set. Let A be the event of a false positive. Then we note that

$$P(A) = \sum_{I \subseteq \{1, \ldots m\}} P(A|E_I) \cdot P(E_I)$$

Now note for a specific subset of bits I, we have that

$$P(A|E_I) = \left(\frac{|I|}{m}\right)^k$$

Since each hash function needs to hit one of the values in the subset out of m to achieve a false positive.

However, we need to know $P(E_I)$. The number of possible events is the number of functions from kn to m, or the number of ways to assign the kn bits set to the bloom filter. This is m^{kn}.

How many of those events gives us the subset I? We see that this is the number of surjections from kn to $|I|$, as each of the $|I|$ bits must be set.

To count the number of surjections, we see that it is the same as partitioning the kn values into $|I|$ nonempty subsets (which is $S(kn, |I|)$ from discrete), and then permuting those subsets. Therefore, we get $|I|!S(kn, |I|)$.

Therefore, the probability

$$P(E_I) = \frac{|I|!S(kn, |I|)}{m^{kn}}$$

Combining everything, we get

$$\begin{aligned}
P(A) &= \sum_{I\subseteq\{1,\dots,m\}} P(A|E_I)P(E_I) \\
&= \sum_{I\subseteq\{1,\dots,m\}} \left(\frac{|I|}{m}\right)^k \cdot \frac{|I|!S(kn,|I|)}{m^{kn}} \\
&= \frac{1}{m^{k(n+1)}} \sum_{i=1}^{m} i^k i! S(kn,i) \binom{m}{i}
\end{aligned}$$

□