A Gentle Introduction to Algorithms

Kevin Zhu & Axel Li

Originally written as official course notes for
UC Berkeley CS 170, Spring 2022

Covering efficient algorithms, divide and conquer, the fast Fourier transform, graphs, shortest paths, greedy algorithms, minimum spanning trees, linear programming, network flow, and zero-sum games.

Contents

Chapter 1

Efficient Algorithms

These notes are serve as a concise presentation of topics taught in CS170, Efficient Algorithms and Intractable Problems, at UC Berkeley. Everything in these notes are in scope for exams, excluding sections specifically designated as [EXTRA] and anything in the footnotes.

In these notes, we will assume knowledge of programming, data structures and mathematical proofs.

We will use Pythonic psuedocode. Assume lists are implemented as Python lists, so accessing an element at a given index takes constant time.[1]

1.1 Motivation

Algorithms[2] form the foundations of modern computing technology. Wireless networks (WiFi), for example, were only made possible by an algorithm for efficient Fourier transforms, and GPS navigation applications by Dijkstra's algorithm. In this course, we will explore the (fast) Fourier transform and Dijkstra's, along with a multitude of other algorithms that form the theoretical foundations of computer science. During this process, we hope to impart insights on how to devise algorithms, prove their correctness, and analyze their time and space complexity - skills that are essential for any computer scientist or software engineer. Welcome to CS170!

1.2 Algorithms

An algorithm is a finite sequence of well-defined instructions used to perform a computation. There may exist many algorithms to solve a given problem, although for practical purposes, we focus on the most efficient ones.

When discussing an algorithm, we generally break the process down into three parts: a description of the algorithm, a proof of correctness, and runtime/memory analysis. We may rigorously express an algorithm by its pseudocode; or, more often, we opt for a

[1] This is because a Python list is implemented as a (dynamic) array (e.g Java ArrayList), and an array is stored as a contiguous block of memory. So, accessing at an index is some arithmetic, then following a pointer to a memory address.

[2] The term 'algorithm' is named after the 9th century mathematician and polymath al-Khwarizmi, known for creating fundamental arithmetic algorithms and inventing algebra, among other things.

high-level description to succinctly highlight the main ideas and omit implementation details. Both of these approaches are equally acceptable for presenting an algorithm.

Example: Finding the maximum element in a list of integers (assume the list is finite and nonempty, so the maximum is well-defined).

High-Level Description:

Assign the first number in the list as the max element. For each remaining number in the list: if this number is larger than max, replace max with this number. When there are no numbers left in the list to iterate over, return max.

Pseudocode:

Algorithm 1 Maximum Element of Nonempty List

```
1: procedure GET_MAX(lst)
2:     max_element ← lst[0]
3:     for num in lst do
4:         if num > max_element then
5:             max_element ← num
6:     return max_element
```

Proof of Correctness:

Observe that the value of max_element is always assigned to be equal to some number in the provided list. This implies that max_element cannot be strictly greater than every number in the list, and is less than or equal to the true maximum element of the list. Simultaneously, the max variable must be greater than or equal to every value in the list due to the for loop. These two inequalities imply that the returned value is exactly equal to the maximum value of the list.

Runtime:

The initialization of the max is $O(1)$, and the loop does $O(1)$ for each element in the list, which has n elements. Thus, the overall runtime is $O(n)$.

1.3 Asymptotics

The widely used method to measure the performance of an algorithm by its running time (also known as "time complexity") is by counting the number of basic computer steps as a function of the size of the input(s), usually denoted by n. As most algorithms finish quickly on modern day computers when n is small, we are most interested in the asymptotic behavior, i.e when n is large. To achieve this, we use *big O notation*.

Intuitively, big O notation[3] allows us to compare two functions asymptotically; for two functions f and g, $f = O(g)$ is an analog of $f \leq g$.

Formally, let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals representing an algorithm's running time with an input of size n. We say $f(n) = O(g(n))$ if there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.[4]

Example: An algorithm that takes $f(n) = 5n^3 + 4n + 3$ steps for an input of size n is $O(n^3)$, as we may set $c = 12$. Similarly, the same algorithm is also $O(n^{100})$ and $O(2^n)$, however these are less informative. To avoid these absurdities, we generally consider the tightest big O bound we can.

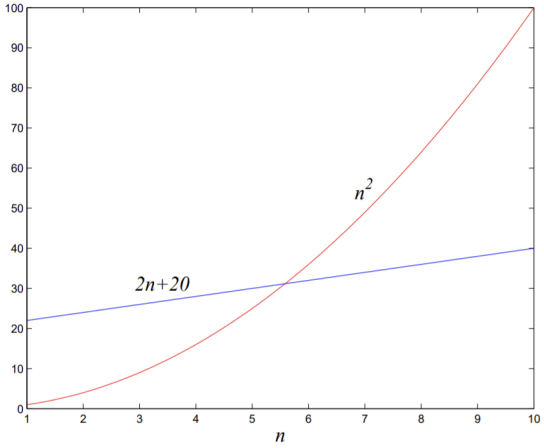
As stated before, $f(n) = O(g(n))$ is an analog of $f \leq g$. Similarly,

$f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$ is an analog of $f \geq g$.

$f(n) = \Theta(g(n)) \iff (f(n) = O(g(n)) \wedge g(n) = O(f(n)))$ is an analog of $f = g$.

The default in this class is to use $O()$ even if $\Theta()$ is eligible.

Figure 1.1: n^2 grows faster than $2n + 20$. Source: DPV, pg. 16



Usefully, big O may also be defined in terms of limits. This is generally the preferred method for determining asymptotic relationship between two functions. The definitions

[3]We are using Donald Knuth's definition of Big-O Notation. There is a slightly different definition used in analytic number theory.

[4]This is technically an abuse of notation, as the equal sign here is not symmetric: $O(n) = O(n^2)$ is true, but $O(n^2) = O(n)$ is certainly false. It'd be more fitting to say $n \in O(n^2)$, but the equals sign is customary.

are as follows:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \iff f(n) = O(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \iff f(n) = \Omega(g(n))$$

This is actually a simplification - these definitions only hold when the limit is defined.[5] Trying the limit test on our example above, $\lim_{n\to\infty} \frac{f(n)}{n^3} = 5$ after applying L'Hopital's rule. Thus, $f = O(n^3)$ and $f = \Omega(n^3)$, so $f = \Theta(n^3)$.

Below are a few general rules for big O notation. Try to justify each statement yourself (Exercise 0.1):

- Exterior multiplicative constants can be omitted, e.g. $14n^2 = O(n^2)$, $3^{n+1} = O(3^n)$.
- Lower order terms can be eliminated, e.g. $n^2 + n \ln n + 10 = O(n^2)$
- Exponentials > polynomials > logarithms > constants, e.g. $3 = O(log(n))$, $\ln n = O(n)$, $n^2 = O(2^n)$.
- Exponentials of higher base dominate those of lower base: a^n dominates b^n if $a > b$, e.g. $2^n = O(3^n)$.
- Polynomials of higher degree dominate those of lower degree: n^a dominates n^b if $a > b$, e.g. $n = O(n^2)$
- Logarithm bases do not matter: $\log_a n = \Theta(\log_b n)$ for any a, b.

Big O notation is also used for measuring the memory usage of an algorithm (also known as "space complexity"). Space complexity generally refers to the additional amount of memory used in an algorithm, i.e excluding the size of the inputs. Usually, this is straightforward to calculate and does not require the analysis above.

It is important to remember that big O is just a model for simplification of analysis. It disregards many important factors of real-life computing speed such as the time it takes reading from disk or the effects of large constant factors. Ultimately, however, it is a very useful framework that we will use for the rest of the class to analyze efficiency.

1.4 Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

[5] Mathematicians bypass the edge case of an undefined limit using supremum or infimum.

The above sequence is known as the Fibonacci numbers. The Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}.$$

We would like to devise an algorithm to calculate the nth Fibonacci number. The first algorithm we may come up with, in pseudocode:

Pseudocode:

Algorithm 2 Naive Fibonacci Algorithm

```
1: procedure FIB(n)
2:     if n = 0 then return 0
3:     if n = 1 then return 1
4:     return FIB(n − 1) + FIB(n − 2)
```

This algorithm is obviously correct, as we have directly transcribed the definition. However, how efficient is it?

Let's do an inductive analysis: let $T(n)$ be the number of computer steps used to calculate F_n. Assuming arithmetic operations are constant, the function's runtime is a constant plus the amount of time of the recursive calls: $T(n) = T(n-1) + T(n-2) + O(1)$. We can observe $T(n-1) \geq T(n-2)$, so $T(n) \geq T(n-2) + T(n-2) = 2T(n-2)$. Furthermore, using the same reasoning, $T(n-2) \geq 2T(n-4)$, so $T(n) \geq 2 * 2T(n-4)$, and so on.

Thus, $T(n) >= 2^{N/2}$ follows with an inductive argument with $T(0) = 1, T(1) = 2$ as the base case. So, this algorithm is on the order of exponential time.

What can we do to improve this algorithm? We may observe that we recalculate FIB(x) for most values of x many times in our recursion (e.g FIB(3) is calculated from scratch for every $x > 3$). To remove this repetition, we can instead store the values of FIB(x) in an array, so we can access them in constant time instead of recalculating them.

Pseudocode:

Algorithm 3 Fibonacci Linear-time Algorithm

```
1: procedure FIB(n)
2:     if n = 0 then return 0
3:     lst ← [0..n − 1]
4:     lst[0] ← 0
5:     lst[1] ← 1
6:     for i in [2..n] do
7:         lst[i] ← lst[i − 1] + lst[i − 2]
8:     return lst[n − 1]
```

Now, each FIB(x) is only calculated once! Since the calculation of FIB(x) for any x is $O(1)$ and there are n values calculated up to FIB(n), the total running time of the algorithm is $O(n)$. At the expense of $O(n)$ memory, we have now reduced our algorithm to linear time![6]

1.5 Bit Complexity and Addition

How many digits are in a non-negative[7] number N in base b? With k digits, we can express numbers up to $b^k - 1$; for example in base 10, the largest number that can be represented with $k = 3$ digits is 999. Thus solving for k, we find a number N has $\lceil \log_b N + 1 \rceil$, or $O(\log N)$ digits. In this course, unless we specify otherwise we will be using base 2, i.e binary, so we may use 'digits' and 'bits' interchangeably.

Let's revisit the elementary algorithm for how to add two numbers. Recall we are interested in measuring an algorithm's runtime with respect to the size of the inputs. For the addition problem, the size of the inputs are the number of digits in the two numbers. Hence, we are interested in the *bit complexity* of the algorithm. Suppose we have two numbers x and y that each are n bits long. Using the elementary algorithm, each individual bit summation is calculated using some fixed amount of time (say, using a truth table), and there are at most $n + 1$ digits in the result. Thus, the entire algorithm runs in $O(n)$.

So, it's true that adding larger numbers takes longer than adding smaller numbers. In the rest of the course, however, we shall treat addition and all other basic arithmetic operations as a constant time operation unless otherwise specified.[8]

[6] This is actually an example of dynamic programming, which will be covered in detail later in this course.

[7] Note that we are only concerned about non-negative integers, though there are several schemes to handle negative numbers by using an extra bit such as Two's Complement, which is covered in CS61C.

[8] Why can we make this assumption? One reason is that in many languages such as Java, ints have a fixed length representation of 32 bits. This means that all basic arithmetic operations involving ints

1.6 Subroutines

When devising new algorithms, we often take an existing algorithm and use it as a subroutine. For example, in multiplication, which we'll cover in the next note, we will use addition as a subroutine. It is useful to think of the subroutine algorithm as a 'black-box', meaning we only care about the inputs and outputs, and ignore all the internal details. In fact, all recursive algorithms take advantage of subroutines, as they use a smaller version of themselves (i.e subproblems) as subroutines! Any algorithm taught in lecture may be used as a subroutine without proof if unmodified in any way. If modified, it must be proven from scratch, as even a small modification can affect an algorithm's correctness.

The process of using an algorithm without modification as a subroutine is formally called a *reduction*, which will be a focus of the second half of this course.

are constant, as the length of all ints are fixed. The downside is that there are limits to the size of numbers that can be represented with ints, but this is generally not a problem since we rarely have to deal with numbers larger than 32 bits. If we do need to work with arbitrarily large numbers then we must use some other representation and arithmetic can no longer be assumed to be constant time.

Chapter 2

Divide and Conquer

2.1 Introduction

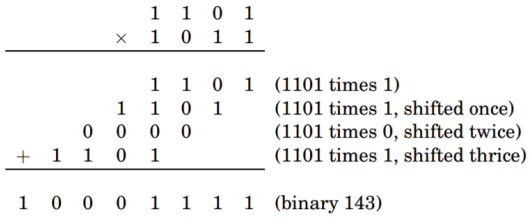
Divide and conquer is the first algorithmic paradigm we will cover in this course; that is, a framework of algorithms that follow the same structure. Divide and conquer is characterized by dividing the problem into some number of subproblems that are each solved recursively, and combining the results intelligently to retrieve the final answer. In this note, we will explore a few algorithms of this nature. To begin, we will examine multiplication, where applying divide and conquer with a neat trick allows us to beat $O(n^2)$.[1]

2.2 Multiplication

2.2.1 Grade School Algorithm

To recap, the grade school algorithm for multiplying two numbers x and y is to separately multiply x by each digit of y, left-shift these results by the appropriate amounts, then add them up. For example, to calculate $13 * 11$ in binary, which is $1101 * 1011$, we would perform the following:

Figure 2.1: Grade School Multiplication. Source: DPV, pg. 24



Bit complexity: we compute n rows, each row having up to $2n$ bits, as each row is left-shifted up to n times. We then do the addition step row by row (not by columns!): we add row 1 (00001101) to row 2 (00011010), then add the result to row 3 (00000000),

[1] Kolmogorov, among other mathematicians, conjectured multiplication would be $\Omega(n^2)$. The fastest algorithm now is $O(n \log n)$ discovered in 2019, inspired by FFT, which we'll learn in the next note.

and so on until we reach the last row.[2] As adding one row to another takes $O(n)$ time and there are $n-1$ additions to perform, the total time taken to add these rows is $(n-1)*O(n)$, which is $O(n^2)$.

2.2.2 Naive Divide and Conquer

A number can be decomposed as the sum of its left and right half: $x = x_L * 2^{n/2} + x_R$ where n is the number of digits of x, in binary (e.g $x = 1011$ is equivalent to $10*100+11$, as 100 in binary is 2^2 in decimal). Let's use this fact to devise a divide and conquer algorithm: we split both x and y into two halves, recursively compute the products of these smaller numbers, and use those to construct our answer. More explicitly, we decompose x and y each into two halves as $x = x_L * 2^{n/2} + x_R$ and $y = y_L * 2^{n/2} + y_R$, then we construct our answer $x * y = x_L * y_L * 2^n + (x_L * y_R + x_R * y_L) * 2^{n/2} + x_R * y_R$, where we solve $x_L * y_L$, $x_L * y_R$, $x_R * y_L$, and $x_R * y_R$ recursively.[3]

Pseudocode:

Algorithm 4 Naive Divide and Conquer Multiplication

```
1:  function MULTIPLY(x, y, n)
2:      if n = 1 then
3:          if x = 1 and y = 1 then
4:              return 1
5:          else
6:              return 0
7:      x_L ← x/2^{n/2}
8:      x_R ← x % 2^{n/2}
9:      y_L ← y/2^{n/2}
10:     y_R ← y % 2^{n/2}
11:     m_LL ← MULTIPLY(x_L, y_L, n/2)
12:     m_LR ← MULTIPLY(x_L, y_R, n/2)
13:     m_RL ← MULTIPLY(x_R, y_L, n/2)
14:     m_RR ← MULTIPLY(x_R, y_R, n/2)
15:     return 2^n * m_LL + 2^{n/2} * (m_LR + m_RL) + m_RR
```

Correctness and bit complexity: The algorithm is correct by induction on n, simply using the distributive property for the inductive step to justify our expression for $x * y$. As for the bit complexity, let's denote $T(n)$ as the overall running time on n-bit inputs. The four $n/2$-bit multiplications are computed recursively, and constructing the answer

[2] We do the addition row by row since for the purposes of this class, computers can only add two numbers at a time. Therefore, to compute the sum of three numbers, we would have to add the first two, then add the third number to the result.

[3] If n is odd, we can simply left pad with a zero.

involves 3 additions that we know take $O(n)$ to calculate, as well as two multiplications by powers of two that are easily calculated by just left-shifting, which is also $O(n)$. So, we get the recurrence relation $T(n) = 4T(n/2) + O(n)$. As we'll see later, this is equivalent to $O(n^2)$, the same as the grade school algorithm.

2.2.3 Karatsuba's Algorithm

However, Karatsuba made a clever observation[4] using the distributive property: $x_L * y_R + x_R * y_L = (x_L + x_R) * (y_L + y_R) - x_L * y_L - x_R * y_R$. Now, to construct our expression for $x * y$, we only need to perform three multiplications instead of four - we'll still calculate $x_L * y_L$ and $x_R * y_R$, but instead of also calculating $x_L * y_R$ and $x_R * y_L$, we'll calculate $(x_L + x_R) * (y_L + y_R) - x_L * y_L - x_R * y_R$ from the observation. (Note we've already calculated $x_L * y_L$ and $x_R * y_R$, so there's only one new multiplication here: $(x_L + x_R) * (y_L + y_R)$). There is a slight drawback in that we introduce some additions and subtractions, but those are still done in linear time. The reduction of a multiplication is much more significant: our resulting recurrence relation is $T(n) = 3T(n/2) + O(n)$. Each recursive call now produces one less subproblem, and this effect compounds. This reduces the runtime from $O(n^2)$ to $O(n^{1.59})$!

Pseudocode:

Algorithm 5 Karatsuba's Algorithm

```
1:  function MULTIPLY(x, y, n)
2:      if n = 1 then
3:          if x = 1 and y = 1 then
4:              return 1
5:          else
6:              return 0
7:      x_L ← x/2^(n/2)
8:      x_R ← x % 2^(n/2)
9:      y_L ← y/2^(n/2)
10:     y_R ← y % 2^(n/2)
11:     m_L ← MULTIPLY(x_L, y_L, n/2)
12:     m_C ← MULTIPLY(x_L + x_R, y_L + y_R, n/2)
13:     m_R ← MULTIPLY(x_R, y_R, n/2)
14:     return 2^n * m_LL + 2^(n/2) * (m_C − m_L − m_R) + m_RR
```

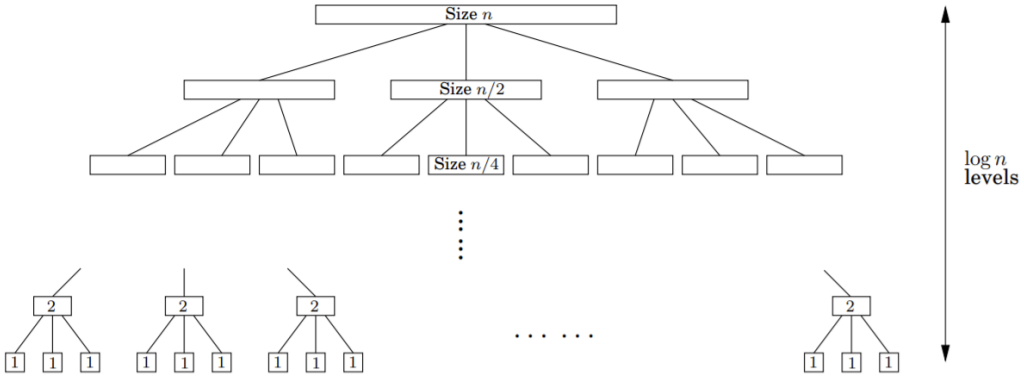
[4]Gauss made the same observation centuries earlier when multiplying complex numbers.

2.3 Master Theorem

2.3.1 Time Complexity Analysis

How do we solve these recurrences? In the example above, we can derive the running time of $O(n^{1.59})$ by examining the pattern of the recursive calls:

Figure 2.2: Recursive call structure. Source: DPV, pg. 58



At each level of recursion, the subproblems are halved in size until they reach 1,[5] therefore the height of the tree is $\log_2 n$. The branching factor is 3, and the result is that at depth k in the tree there are 3^k subproblems, each of size $n/2^k$.

Each subproblem (not subtree) takes a linear amount of time, correlating with the size of the subproblem. Therefore a subproblem of size $n/2^k$ can be solved in $O(n/2^k)$ time, so at depth k of the tree, $3^k * O(n/2^k) = (3/2)^k * O(n)$ time is spent to solve all subproblems.

This means that the total time spent to solve all subproblems is a geometric series, starting at $O(n)$ and ending at $O((\frac{3}{2})^{\log_2 n} * n) = O(3^{\log_2 n} * \frac{n}{n}) = O(3^{\log_2 n}) = O(3^{\log_3 n / \log_3 2}) = O(n^{1/\log_3 2}) = O(n^{\log_2 3})$, which is roughly $O(n^{1.59})$. Since the sum of any increasing geometric series is, within a constant factor, simply the largest term in the series, this is also the time complexity of solving all subproblems combined.

2.3.2 Master Theorem

The analysis above is standard for divide and conquer algorithms, so let's solve the general form and just reuse the results from here on. Most divide-and-conquer algorithms tackle a problem of size n by recursively solving subproblems of size n/b, then combining these answers in $O(n^d)$ time, for some $a, b, d > 0$ (in the multiplication algorithm,

[5] In practice, the base case is not 1, but at 16 or 32 bits depending on the processor, as they can multiply these numbers in a single operation.

$a = 3$, $b = 2$, and $d = 1$). The running time is therefore $T(n) = aT(\lceil n/b \rceil) + O(n^d)$. We can now derive a closed-form solution for given values of a, b, d.

Master Theorem: If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

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

Proof: (For the sake of convenience, let's assume that n is a power of b, allowing us to ignore the rounding effect in $\lceil n/b \rceil$.)

As the size of the subproblems decreases by a factor of b with each level of recursion, there are $\log_b n$ levels. The tree has a branching factor of a, so at depth k there are a^k subproblems, each of size n/b^k. The total time complexity of all subproblems at depth k is $a^k * O(n/b^k)^d = O(n^d) * (a/b^d)^k$.

As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio a/b^d, and there are three cases:

1. $a/b^d < 1$: The sum is dominated by the first term, $O(n^d)$.
2. $a/b^d > 1$: The sum is dominated by the last term, $O(n^{\log_b a})$, as shown in the multiplication algorithm.
3. $a/b^d = 1$: All $O(\log n)$ terms of the series are equal to $O(n^d)$, and the total complexity is $O(n^d \log n)$.

2.4 Matrix Multiplication

Matrix multiplication can also be done with a divide-and-conquer algorithm. To recap, the product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, where the (i, j)th entry is

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

This formula implies an $O(n^3)$ algorithm for matrix multiplication, as there are n^2 entries, each taking $O(n)$ time. However, similarly to grade school multiplication, there is a more efficient divide-and-conquer algorithm.

Matrix multiplication is easy to break into subproblems, as it can be performed block-wise. We can carve an $n \times n$ matrix into four $n/2 \times n/2$ matrices:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad\qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

And their product can be expressed in terms of these blocks and is exactly as if the blocks were single elements.

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

However, this particular divide-and-conquer algorithm has eight size-$n/2$ products and $O(n^2)$-time additions, resulting in the recurrence relation $T(n) = 8T(n/2) + O(n^2)$, which comes out to $O(n^3)$ as well.

Strassen discovered with very clever algebra, matrix multiplication can be broken down into only seven $n/2 \times n/2$ subproblems:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ P_3 &= (C + D)E \\ P_4 &= D(G - E) \\ P_5 &= (A + D)(E + H) \\ P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F) \end{aligned}$$

The new recurrence relation is $T(n) = 7T(n/2) + O(n^2)$, which works out to $O(n^{\log_2 7})$, roughly $O(n^{2.81})$.

The intuition behind how he discovered this result is not in scope for this class.

2.5 Sorting

2.5.1 Mergesort

Let's try applying divide and conquer to some other familiar problems, such as sorting a list. The natural approach would be to divide the list into two halves, recursively sort each half, then somehow merge the two sublists together. This is precisely Mergesort.
Pseudocode:

Algorithm 6 Mergesort

```
1: function MERGESORT(A[0, n − 1])
2:     if n = 1 then
3:         return A[0]
4:     else
5:         return MERGE( MERGESORT(A[0, ⌊n/2⌋ − 1]), MERGESORT(A[⌊n/2⌋, n − 1])
   )
```

To complete this description, we need to specify the merge step. How do we efficiently merge two sorted lists $A[0..n-1]$ and $B[0..m-1]$ into one sorted list $Z[0..m+n-1]$? The key observation is that the smallest element, $Z[0]$, must be $A[0]$ or $B[0]$, whichever is smaller. Why? WLOG, let $A[0]$ be the smallest element. By A being sorted, $A[0]$ is smaller than every other element in A, and $A[0]$ is smaller than $B[0]$, which is smaller than every other element in B. Next, $Z[1]$ must be either $A[1]$ or $B[0]$ by the same logic. Notice our subproblem still has two sorted lists - we can solve this recursively.
Pseudocode:

Algorithm 7 Merge

```
1: function MERGE(A[0, n − 1], B[0, m − 1])
2:     if n = 0 then
3:         return B
4:     else if m = 0 then
5:         return A
6:     if A[0] ≤ B[0] then
7:         return A[0] ∘ MERGE(A[1, n − 1], B[0, m − 1])
8:     else
9:         return B[0] ∘ MERGE(A[0, n − 1], B[1, m − 1])
```

Note the circle sign there means concatenation.

Proof Sketch: The merge correctly produces the sorted list since we take the smallest number out of the remaining numbers at each step, as shown in the analysis in the section above. The recursive step follows by induction on n.

Runtime Analysis: The merge step takes $O(n)$ time, as each element is considered once and completed in constant time. The recursive step breaks down the problem into two subparts. Thus, $T(n) = 2T(n/2) + O(n) = O(nlogn)$ from Master Theorem.

As a final note, notice we do all the recursive steps before the merge step. So, during this algorithm's execution, this will create the entire tree of subproblems before ever merging. This suggests we could also do a "bottom-up" formulation where we just start at the singleton arrays and iteratively merge up to the final answer. This may be implemented with a queue.

Pseudocode:

Algorithm 8 Iterative Mergesort

```
1: function ITERATIVE_MERGESORT(A[0, n − 1])
2:     Q = []                                   ▷ initialize empty queue
3:     for i = 0 to n − 1 do
4:         Q.PUSH([A[i]])
5:     while |Q| > 1 do
6:         Q.PUSH(MERGE( Q.POLL(), Q.POLL() ))
7:     return Q.POLL()
```

2.6 Median and Selection

Given an unsorted list of numbers, suppose we would like to find some summary statistics to give us a sketch of the data. One choice is the arithmetic mean, i.e the total sum divided by the number of elements, which can be computed easily in $O(n)$. Another choice is the median, i.e the $\lfloor \frac{n+1}{2} \rfloor$-th smallest number, which is often preferable since it is more robust to outliers. However, the median is trickier to calculate - we could resort to an $O(n \log n)$ sort, but that is inefficient, as we likely don't need to construct the entire ordering to search for just a single key element. Let's design an $O(n)$ algorithm. To help us use recursion, we will generalize the median finding problem to selection.

A selection algorithm is an algorithm to find the kth smallest number (aka kth order statistic) in a list for any value of k. For example, $k = 1$ is the minimum element, $k = \lfloor \frac{n+1}{2} \rfloor$-th is the median, and $k = n$ is the maximum (note we are 1-indexing for k).

2.6.1 Quickselect

Let's use divide and conquer. Using our usual format, we could try dividing into two sublists and try to recursively find the kth smallest element in each. But, then how do we use those two to get the kth smallest element of the original list? There's not enough information here, so we'll try a different approach.

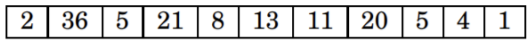
How else can we divide the list? One way is through the use of a partition on a pivot v: for a number v, we can split our list into three categories: elements smaller than v, elements greater than v, and elements equal to v.[6] There are ways to do this in $O(n)$, though they are not the focus of this class.[7] Note the sublists are not necessarily ordered.

[6] We could just use two groups by combining the equals group with one of the other groups, which would yield roughly the same results if there aren't many duplicates.

[7] Two ways include the Hoare and Lomuto partition schemes. This problem is called the Dutch National Flag Problem, proposed by Dijkstra.

Example: Splitting the following array on $v = 5$.

Figure 2.3: Array to be split. Source: DPV, pg. 64



yields the following subarrays

Figure 2.4: Subarrays after splitting. Source: DPV, pg. 64

This tells us an ordering on S_L, S_V, and S_R: every element in S_R is at least as large as the elements in S_V, and every element in S_V is at least as large as the elements in S_L. Thus, we can narrow down our search to only one of these sublists. For example above, if $k = 7$, we only need to look at S_R, as the 7th smallest element is in the sublist containing the 6th - 11th smallest numbers. Explicitly,

$$\text{SELECTION}(S, k) = \begin{cases} \text{SELECTION}(S_L, k) & k \leq |S_L| \\ v & |S_L| < k \leq |S_L| + |S_v| \\ \text{SELECTION}(S_R, k - |S_L| - |S_v|) & k > |S_L| + |S_v| \end{cases}$$

So, our algorithm would be to pick a value v, partition our list with respect to v, then recurse on the relevant sublist. We haven't specified how to pick v yet, but are we on the right track to an efficient algorithm? Suppose we were able to split the list by roughly half on each recursive step. Then, $T(n) = T(n/2) + O(n)$ is equal to $O(n)$ by the Master Theorem.

2.6.2 Pivot Selection

Random (Quickselect)

How do we select v? We must do it quickly, and it should split the list well. One way would be just picking an element at random from the list.

Runtime: In the worst case, if we're really unlucky, our algorithm would pick the smallest/largest element as the pivot on every single iteration, reducing the size of our list by only one; this would make our algorithm run in $(n)+(n-1)+(n-2)+\cdots+1 = O(n^2)$. However, in the best case it picks the middle element every time, which was $O(n)$ from before. What we care about is the average case.

We'll need to calculate the expected running time. Our pivot is random, so to give us some more structure for analysis, let's define some boundaries: call a pivot that falls between the 25-75th percentile a "good" pivot. So, 50% of pivots are good, and it takes on expectation two recursive iterations to get a good pivot by expectation of a

geometric distribution. A good pivot reduces our list by at least 25%. So, letting $T(n)$ be the expected runtime, $T(n) \leq T(\frac{3}{4}n) + O(n)$, which is $O(n)$ by Master Theorem. This algorithm is called Quickselect[8].

Deterministic (Deterministic-select) [EXTRA]

We can actually guarantee an $O(n)$ worst case with a clever scheme to get a pivot called median of medians as follows:

1. Group the array into $\lfloor n/5 \rfloor$ groups of 5 elements each
2. Find the median of each group (as each group has a constant number of elements, finding each individual median is $O(1)$)
3. Create a new array only with the $\lfloor n/5 \rfloor$ medians, and find the true median of this array by recursively calling the selection algorithm on it.
4. Return this median as the pivot.

This pivot is guaranteed to be "good", specifically in the 30-70th percentile. Why? It's greater or equal to than half of the medians, since it's the median of the medians. And each of those medians are greater or equal to the smallest three elements of their respective five element list, since they're the medians of their respective list. So, $p \geq \frac{1}{2} * \frac{n}{5} * 3$, thus $p \geq 3/10 * n$. With a symmetrical argument, $p \leq 7/10 * n$.

Runtime: Let's construct the recurrence. There are two recursive calls: one at step 3 of medians of medians to find the median of the medians, and one from our selection algorithm to recurse on our desired sublist. As for the work per recursive call, we do a linear scan to construct our list of medians, and another linear scan to do our partition in our selection algorithm. Thus $T(n) \leq T(n/5) + T(7n/10) + O(n)$.

We can't solve this with Master Theorem unfortunately. Instead, we can use induction to show $T(n) \leq c * n$ for some constant $c > 0$, thus proving it's $O(n)$ by the definition of Big-O.

Applying the inductive hypothesis, $T(n) \leq c(n/5) + c(7n/10) + d * n \leq (9/10 * c + d) * n$ for some constant $d > 0$.

Remember, we need to just show the existence of some c. To finish showing $T(n) \leq c * n$, we need to just pick c large enough that $(9/10 * c + d) \leq c$, i.e $c \geq 10d$. As for the base case, running deterministic-select on a single element list is trivially constant time.

In practice, efficient implementations of both these algorithms both perform well, with small preference towards Quickselect.[9]

[8] If you are familiar with Quicksort, you might have seen a strong similarity with it and Quickselect - both are divide and conquer algorithms using a random partition with "quick" in their name. This is because they were created by the same person, Tony Hoare.

[9] We can also use selection for sorting, though it's not very efficient - if we iteratively select the ith

2.7 Binary Search

Binary Search is the classic algorithm for finding a target number in a sorted list. When searching for a number in a sorted list, it is inefficient to iterate it in its entirety as there is useful structure in a sorted list. A single comparison, $A[i] < x$, tells us not only that x is greater than $A[i]$, but also everything to the left of it, allowing us to discard an entire subarray per comparison. To maximize our number of elements discarded in the worst case, we'll compare our target number to the middle element and recurse on either the left or right subarray depending on the results of the comparison.[10] Here is an iterative pseudocode of the algorithm, though it can also be done recursively.

Pseudocode:[11]

Algorithm 9 Binary Search

```
1:  function BINARY_SEARCH(A[0, n − 1], k)
2:      l ← 0
3:      h ← n − 1
4:      while l < h do
5:          m ← ⌊(l + h)/2⌋
6:          if A[m] < k then
7:              l ← m + 1
8:          else if A[m] > k then
9:              h ← m − 1
10:         else
11:             return m
12:     return -1
```

smallest element for $i = 0..n-1$, we will have constructed the entire sorted list. This is Selection sort.

[10] Quickselect and Binary Search are different from the other algorithms presented in this note, as their subproblems are not combined; instead, they discard all subproblems except for one, and recurse on that single subproblem. As such, some computer scientists prefer to label this type of algorithm as "*decrease* and conquer".

[11] Implementing binary search may be trickier than you'd expect due to edge cases. There was a bug in Java's implementation of binary search until 2006 due to integer overflow (our pseudocode does not account for it for sake of clarity, though it's an easy fix).

Chapter 3

The Fast Fourier Transform

3.1 Introduction

So far in this class, we've discussed efficient divide and conquer algorithms for multiplying numbers and matrices. The last divide and conquer algorithm we will discuss is the Fast Fourier Transform (FFT) algorithm, which we will use to help us multiply polynomials. The FFT has been described as "the most important numerical algorithm of our lifetime" due to its important practical applications in signal processing. Due to its importance (and complexity), we'll dive much deeper into the details than usual.

In this note, we will derive the most common variant of the FFT, called the Cooley-Tukey FFT[1] in the context of speeding up polynomial multiplication. We hope to make its derivation intuitive, though feel free to skip straight to the algorithm. Finally, we will introduce FFT in relation to the Discrete Fourier Transform (DFT), and discuss some additional applications.

For this note, assume arithmetic operations take constant time.

3.2 Polynomial Multiplication

Forward: This section isn't necessary for understanding the FFT algorithm, but it will help build context on the derivation. Also, some of the homework/exam problems in this class involve reductions to polynomial multiplications (i.e re-expressing the problem input in terms of polynomials, then multiplying them efficiently using FFT as a black-box algorithm).

First, we'll discuss two naive ways of multiplying polynomials. Given two polynomials $A(x) = a_0 + a_1 + \cdots + a_{n-1}x^{n-1} + a_n x^n$ and $B(x) = b_0 + b_1 + \cdots + b_{n-1}x^{n-1} + b_n x^n$, we would like to calculate the product polynomial, $C(x) = A(x)B(x)$. What does the product polynomial look like in general? The easiest way to see this is to try multiplying two arbitrary low-degree polynomials by hand and rearranging the terms in order of degree. You should see that for any k, $c_k x^k$ in $C(x)$ is the result of adding

[1]The algorithm is named after Tukey, who created the algorithm during a meeting discussing detection of nuclear-weapon tests in the Soviet Union. It was later discovered that Gauss had already invented this algorithm in 1805.

all the pairs of monomials whose degree adds up to k. Explicitly:

$$\begin{aligned} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ &\vdots \\ c_k &= \sum_{j=0}^{k} a_j b_{k-j} \end{aligned}$$

3.2.1 Naive algorithm 1: Multiplying in coefficient form

When multiplying two polynomials, a common manual method is to use the distributive property and write out all of the terms, then combine terms of the same degree. A naive algorithm to multiply polynomials can be easily fashioned using this approach.

To calculate the runtime of this algorithm, we must first determine how many different terms there are after distributing. Since each polynomial has $n+1$ terms, the product of the two polynomials will have $(n+1)^2 = O(n^2)$ terms in it, each of which can be calculated in constant time. Afterwards, combining terms takes $O(n^2)$ time as well. Thus, this algorithm runs in $O(n^2)$.

3.2.2 Naive Algorithm 2: Multiplying in value form, but with inefficient conversions

Recall that a polynomial of degree n is usually expressed in its coefficient representation: $A(x) = a_0 + a_1 + \cdots + a_{n-1}x^{n-1} + a_n x^n$. However, we can uniquely represent $A(x)$ with any $n+1$ points on the polynomial (e.g a line is uniquely determined by two points). This is the value representation: $(x_1, A(x_1)), (x_2, A(x_2)), \ldots, (x_{n+1}, A(x_{n+1}))$.

Note this holds for any $n+1$ points of our choosing. Intuitively, there are $n+1$ coefficients in the coefficient form and $n+1$ points in the value representation - we have an equal number of degrees of freedom, and they should thus encode the same amount of information, though we won't provide a proof.

This is useful, since multiplying point-wise is much more efficient than multiplication in coefficient form, which we'll discuss more in detail below.

So, given two degree n polynomials in coefficient representation, our algorithm would be:

1. Evaluate $A(x)$ and $B(x)$ each at the same set of $2n+1$ x-coordinates of your choice (i.e converting them to value representation).
2. Multiply each of these points together.

3. Interpolate the $2n+1$ points to retrieve $C(x)$ (i.e converting the points back into coefficient representation).

Note that although each $A(x)$ and $B(x)$ can be uniquely represented by $n+1$ points, the product polynomial is of degree $2n$, and therefore requires $2n+1$ points to represent it uniquely.

Runtime: To calculate the runtime, let's first analyze how to evaluate a polynomial at a given arbitrary x-coordinate, x_0. Note that to calculate, say, x_0^9, we can reuse our result of x_0^8 and multiply that by x_0 instead of recalculating it from scratch. Taking advantage of this fact, we can rewrite the polynomial:

$$A(x) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-1} + x_0 a_n)))$$

and evaluate this expression starting from the innermost parentheses.[2] This involves n additions and n multiplications, and is thus $O(n)$ time. Since we do this for $2n+1$ points, evaluation takes $O(n^2)$ in total.

Next, multiplying two points at the same x-coordinate is simply multiplying their y values together, which is $O(1)$. Since we do this for $2n+1$ points, this step takes $O(n)$ in total.

Finally, interpolation can be done in $O(n^2)$ as well using linear algebraic techniques, though they aren't the focus of this class.[3]

So, in total this algorithm is still $O(n^2)$. Our bottleneck is converting between coefficient and value representation. This is where the FFT (and the inverse FFT, aka IFFT) come into play.

3.3 FFT Derivation

How do we speed up evaluation? For simplicity of calculations, from here on, let's say we're given a single polynomial, $A(x)$, as a degree $n-1$ polynomial, so it needs n points to uniquely define it. Remember we have the choice of whichever n points we like - perhaps we can come up with a better set of points than just picking them arbitrarily.

3.3.1 Observation 1: Positive-Negative Pairs

Say you wanted to evaluate $A(x) = x^2$, for example. A simple observation is that $A(x) = A(-x)$ for any x, since our polynomial is symmetric over the y axis. Gener-

[2] This is called Horner's method.

[3] One way is to create a system of equations produced by plugging in each point in the coefficient form and solving for the coefficients. Another is Lagrange Interpolation, which is sometimes covered in CS70 and EECS16B.

alizing this, $A(x) = A(-x)$ is true for any polynomial consisting of only even degree terms (e.g $A(x) = 2 + x^2$), as the -1s are taken to only even powers.

To utilize this observation, let's choose our n points as $n/2$ positive-negative pairs: $\{x_1, -x_1, x_2, -x_2, \ldots, x_{n/2}, -x_{n/2}\}$; then, we essentially only need to evaluate half of our original points, since we can easily retrieve $A(-x)$ from each $A(x)$.

Right now, this only holds for even polynomials. But we can actually take advantage of this observation for all polynomials: we can decompose a polynomial into its even and odd terms, then factor an x out of the odd terms to create another even polynomial:

Example: $A(x) = 2+3x+5x^2+7x^3+11x^4+13x^5 = (2+5x^2+11x^4)+x(3+7x^2+13x^4)$.

Now with this decomposition, since the polynomials within the parentheses (which we'll call 'subpolynomials' from now on) are even, $A(x) = A(-x)$ for those subpolynomials. For example, try evaluating $A(x)$ at $x = 1$ and $x = -1$. The subpolynomials yield 18 and 23 respectively for both $x = 1$ and $x = -1$. This is great, as we can evaluate the two subpolynomials at $x = 1$ once, and use the results for both $A(1)$ and $A(-1)$. Here, $A(1) = 18 + 1 \cdot 23$ and $A(-1) = 18 + -1 \cdot 23$.

This isn't actually much speed-up though, as cutting our number of evaluations in half still yields $1/2 \cdot O(n^2) = O(n^2)$. However, if we can do this *recursively*, the effect compounds.

3.3.2 Observation 2: Subpolynomials in x^2

Notice in the example above that all the monomial terms in the parentheses are polynomial in x^2 (simply because even numbers are divisible by two). So, we can formulate these subpolynomials as functions of x^2, which allows us to use recursion since the degree (and thereby number of necessary points) are now halved in these two polynomials! Combining our two observations, we can express our polynomial $A(x)$ and also $A(-x)$ as:

$$A(x) = A_e(x^2) + xA_o(x^2)$$
$$A(-x) = A_e(x^2) - xA_o(x^2)$$

where

$$A_e(x^2) = a_0 + a_2x + \cdots + a_{n-2}x^{n/2-1}$$
$$A_o(x^2) = a_1 + a_3x + \cdots + a_{n-1}x^{n/2-1}$$

Example: Using the same example above, $A_e(x^2) = 2 + 5x + 11x^2$ and $xA_0(x^2) = x(3 + 7x + 13x^2)$.

So, instead of evaluating $A(x)$ at n points: $\{x_1, -x_1, x_2, -x_2, \ldots x_{n/2}, -x_{n/2}\}$, we can instead evaluate $A_e(x^2)$ and $A_0(x^2)$ at $n/2$ points: $\{x_1^2, x_2^2, \ldots x_{n/2}^2\}$ and use the above formulas to retrieve $A(x)$ and $A(-x)$ for each point.

This relation suggests a recursive algorithm, as we desire to recursively evaluate $A_e(x^2)$ and $A_o(x^2)$ in the same way, i.e further breaking them into their respective even and odd subpolynomials at $n/4$ points that are positive-negative paired, and so on.

However, there is a glaring issue. We chose our original input to consist of positive-negative pairs to reduce our work by half, but after one step of recursion, our points are no longer positive-negative paired - they are all positive, as x^2 is always positive for real numbers.

3.3.3 Observation 3: Complex Numbers

So, we turn to the complex numbers. Reverse-engineering what the points must be, at the bottom layer, let's say we have $x = 1$. At the previous layer of recursion, we then needed $x^2 = 1$, so $x = 1$ and $x = -1$. At the layer before that, we then needed $x^2 = 1$ and $x^2 = -1$, so $x = 1, -1, i, -i$. We can see in general, at the top layer we need the set of x such that $x^k = 1$. These are known as the roots of unity.

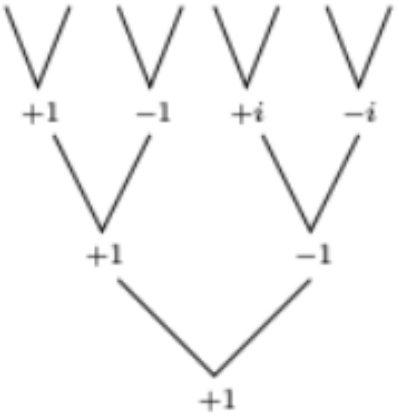


Figure 3.1: 4th roots of unity. Source: DPV, pg. 64

3.3.4 Roots of Unity

The nth roots of unity, denoted as ω_n^k for $k \in \{0, 1, \ldots, n-1\}$, are defined to be the set of x such that $x^n = 1$. Written using Euler's formula, they are $\omega_n = e^{2\pi ik/n}$ since taken to the nth power yields $e^{2\pi ik}$, which will always evaluate to 1 for any value of k, as they are multiples of 2π radians.

Notice that squaring a nth root of unity produces a $n/2$th root of unity. Algebraically, you can imagine squaring as placing a $/2$ in the denominator of the exponent, creating an $n/2$th root. Though it may be clearer to see geometrically:

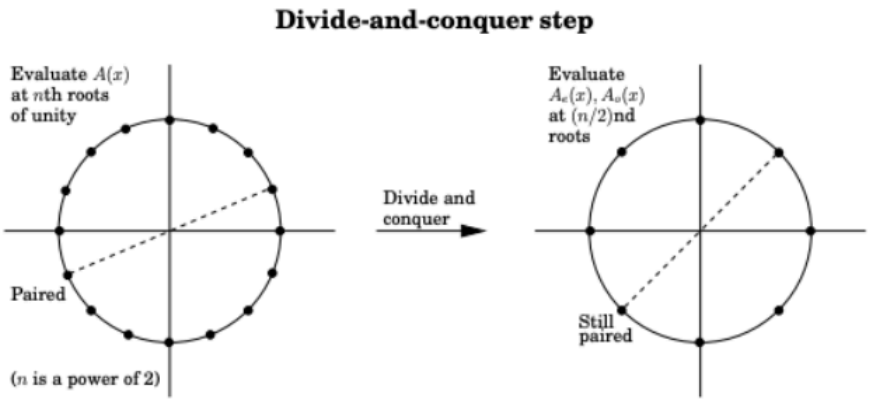


Figure 3.2: Source: DPV, pg. 74

So, the nth roots of unity when squared creates $n/2$th roots of unity, which are still positive-negative paired - this allows us to continue our recursion! Thus, we have our algorithm.

3.4 FFT Algorithm

The FFT algorithm evaluates a polynomial at the nth roots of unity. Here is the psuedocode:

Pseudocode:

Algorithm 10 FFT

```
1: function FFT((a_0, a_1, ..., a_{n-1}), ω)
2:     if ω = 1 then
3:         return a_0
4:     E_0, E_1, ..., E_{n/2-1} ← FFT((a_0, a_2, ..., a_{n-2}), ω^2)
5:     O_0, O_1, ..., O_{n/2-1} ← FFT((a_1, a_3, ..., a_{n-1}), ω^2)
6:     for j = 0 to n/2 − 1 do
7:         A_j ← E_j + ω^j O_j
8:         A_{j+n/2} ← E_j − ω^j O_j
9:     return (A_0, A_1, ..., A_{n-1})
```

(Note that n must be a power of 2, so we must round the number of coefficients up to the next power of two, padding with zeros if necessary.)

In this pseudocode, the input is the coefficients of a polynomial $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ in a list, and the A_i's that are returned represents the original polynomial $A(x)$ evaluated at ω^i i.e. the nth roots of unity. E_i and O_i represent the even and odd

polynomials evaluated at $(\omega^i)^2$, respectively. We put the positive pairs of the roots of unity in the first half of the returned list, A, and the negative pairs in the second half.

Runtime: Let $T(n)$ be the amount of time required to calculate the FFT with the nth roots of unity. At each recursive step, we create two subproblems (i.e the subpolynomials $A_e(x^2)$ and $A_o(x^2)$) each half of the original size, as they're polynomial in x^2, which means their degree (and thereby number of points) is halved. We evaluate n points in total using the two formulas (notice the for-loop over $n/2$ doing two assignments per loop) each in constant time since it's just a few arithmetic operations, yielding $O(n)$. Thus, we have the recurrence relation $T(n) = 2T(n/2) + O(n)$. By the Master Theorem, this algorithm runs in $O(n \log n)$ time, a *significant* improvement over the naive $O(n^2)$ approach!

As a sanity check, what gave us our speedup? Suppose we didn't use positive-negative pairs and just used $A(x) = A_e(x^2) + xA_o(x^2)$ recursively at n arbitrary points. In order to evaluate our original n points, we'd need to solve $A_e(x^2)$ and $A_o(x^2)$ at the full n points (in contrast with $n/2$ points) And so on - every layer requires the full n points. The runtime of that would be back to $O(n^2)$ (Exercise 2.3). Thus, our speedup came from reducing our points in half recursively.

3.4.1 Matrix Generalization: DFT, Inverse FFT

Why is this algorithm called the Fast Fourier Transform? Let's generalize outside the scope of polynomials. Evaluating a polynomial can be interpreted using linear algebra as a 'transformation' from the coefficient 'basis' to the value 'basis', so it can represented as a matrix-vector multiplication:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \tag{3.1}$$

Above, our choice of n points determines our transformation matrix, which we multiply on our vector of coefficients to give us a resulting vector of values.

The FFT chooses our n points to be the roots of unity. This transformation matrix is known as the Discrete Fourier Transform (DFT) matrix.

$$D_n(\omega) = \begin{bmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \dots & \omega^{0\cdot(n-1)} \\ \omega^{01\cdot 0} & \omega^{1\cdot 1} & \dots & \omega^{1\cdot(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{(n-1)\cdot 0} & \omega^{(n-1)\cdot 1} & \dots & \omega^{(n-1)\cdot(n-1)} \end{bmatrix} \tag{3.2}$$

where $\omega = e^{\frac{2\pi}{n}i}$ is a nth root of unity.

This visualization is helpful as we can think of polynomial interpolation as applying the inverse DFT matrix. Using linear algebra, it turns out that the inverse of the DFT matrix is closely related the DFT matrix, namely $D_n(\omega)^{-1} = \frac{1}{n} D_n(\omega^{-1})$. Thus, the FFT algorithm can be modified easily to compute the inverse as well. All you have to do change the input of the psuedocode to take the inverse roots of unity, then divide by n at the end![4]

3.5 Application

3.5.1 Revisiting Polynomial Multiplication

We can now improve naive algorithm 2 using FFT and IFFT! Given two degree n polynomials in coefficient representation, let N equal to $2n + 1$ rounded up to the nearest power of 2. Our improved algorithm is as follows:

1. Evaluate $A(x)$ and $B(x)$ at the Nth roots of unity (i.e converting them to value representation).
2. Multiply each of these points together.
3. Interpolate these points using IFFT to retrieve $C(x)$ (i.e converting the points back into coefficient representation).

Runtime: FFT in $O(N \log N)$ + multiplication in $O(N)$ + IFFT in $O(N \log N)$ yields $O(N \log N)$. Note that $N = O(n)$, so the algorithm runs in $O(n \log n)$.

3.5.2 Integer Multiplication [EXTRA]

Recall that when we write an integer in base b as $(a_n \dots a_1 a_0)_b$, what we really mean is

$$a_n b^n + \cdots + a_1 b + a_0$$

This is precisely $A(x) = a_n x^n + \cdots + a_1 x + a_0$ evaluated at b! Thus, if we wanted to multiply two integers in base b, we could think of it as multiplying two polynomials, then writing down the coefficients of the resulting polynomial as the digits of our answer.

If it were this simple, why did it take researchers until 2020 to arrive at an $O(n \log n)$ algorithm for integer multiplication? Well, there's no guarantee that the coefficients are actually representable as digits. In other words, we need to carry, and that takes time.

[4]Note that in other sources the normalization factor of $\frac{1}{n}$ for the inverse DFT matrix only may be split to be $\frac{1}{\sqrt{n}}$ in front of both the DFT matrix and its inverse.

3.5.3 Convolution [EXTRA]

Suppose we had two discrete time signals $x[n]$ and $y[n]$, i.e. functions from integers to real values. The convolution of x and y is also a signal (i.e. function) defined by the following:

$$(x * y)[n] = \sum_{k=-\infty}^{\infty} x[k]y[n-k]$$

However, this almost the exact same formula as the product of two polynomials! In particular, if we have $C(x) = A(x)B(x)$, then the coefficients of $C(x)$ are

$$c_i = \sum_{k=0}^{\min(i,n-1)} a_k b_{i-k}$$

If x and y are only nonzero for $t = 0, 1, \ldots, n-1$, then convolution and polynomial multiplication are identical.

3.5.4 Cross Correlation [EXTRA]

Cross correlation can be thought of as a sliding dot product between two signals. In particular, we write

$$(x \star y)[n] = \sum_{k=-\infty}^{\infty} x[k]y[n+k]$$

where x and y are discrete time signals. The nth term computes the dot product of x and y shifted to the left by n.

Notice how this equation is similar to the equation for convolution, just with a $+k$ instead of a $-k$. In fact, the cross correlation between two signals is exactly the same as convolution between two signals, but with one of them reversed in time!

To see this, let $w[n] = x[-n]$. Then

$$\begin{aligned}
(w \star y)[n] &= \sum_{k=-\infty}^{\infty} w[k]y[n+k] \\
&= \sum_{k=-\infty}^{\infty} x[-k]y[n+k] \\
&= \sum_{i=-\infty}^{\infty} x[i]y[n-i] \\
&= (x * y)[n]
\end{aligned}$$

Chapter 4

Graphs and Depth-First Search

4.1 Introduction

So far, we've only worked with lists of data. For the next few notes, we'll consider algorithms on graphs, which are useful for encapsulating relationships in the data. Before we go into the algorithms, we'll first review graphs broadly in mathematics and computing. Then, we'll discuss graph search using depth first search (DFS). We will define pre/post order numbers, and edge types based on the tree formed by a DFS to use as a useful framework to help us understand how DFS allows us to decompose the structural relationships in graphs, namely for cycle detection, DAG linearization, and finding (strongly) connected components.

4.2 Graph representation

Recall the structure of a graph from a discrete mathematics course. A graph G consists of a set of vertices (often called 'nodes'), and a set of edges between pairs of vertices, denoted V and E, respectively. The edges of a graph may be either directed or undirected. An undirected edge is denoted $e = \{u, v\}$, while an edge in a directed graph is denoted $e = (u, v)$, where u and v are the two incident vertices.

Also, a tree is a minimally connected graph, where the removal of any edge disconnects the graph. Equivalently, it can be defined as a connected graph with no cycles. As such, $|E| = |V| - 1$ for any tree.

Recall the graph abstract data type from a data structures course. Graphs should support the operations of:

1. Retrieving the set of neighbors of a given vertex
2. Testing if two vertices are adjacent to each other
3. Addition and removal of vertices and edges

The two most common data structures to efficiently support this are the adjacency matrix and adjacency list.

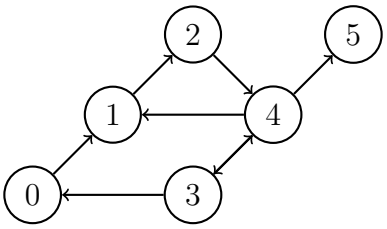
Below, we'll describe the two data structures for directed graphs, as undirected edges can be expressed as two directed edges (i.e $\{v_i, v_j\}$ is equivalent to (v_i, v_j) and (v_j, v_i)).

4.2.1 Adjacency Matrix

In an adjacency matrix, we represent a graph as a $|V| \times |V|$ matrix. The entry at index (i, j) of the matrix is an indicator (1 if True, 0 if False) indicating if there exists a directed edge from v_i to v_j. For simplicity, we'll assume our adjacency matrix is implemented as a list of lists[1].

A cute observation is that this implies that for an undirected graph, its corresponding adjacency matrix is symmetric.

Example:



The adjacency matrix for the previous graph is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4.2.2 Adjacency List

In an adjacency list, we represent a graph as a list of length $|V|$. The entry at index i in the list is a collection containing all the vertices that v_i has an outgoing edge pointing to it. For simplicity, we'll assume that our adjacency list is implemented as a list of lists.[2]

Similarly, this implies for an undirected graph, if v_j exists in v_i's list, then v_i exists in v_j's list.

Example: The adjacency list for the previous graph is:

[1] list as in Python list, not the list abstract data type.

[2] Why not use a list of hashsets or binary trees, as that would allow us to find an edge in $O(1)$ or $O(\log(\text{outdeg}(v_i)))$ time, respectively? In addition to overhead costs unexpressed in asymptotic notation, ultimately, they are unnecessary for most algorithms, which usually just involve iterating over the edges.

Vertex	Neighbors
0	1
1	2
2	4
3	0, 4
4	1, 3, 5
5	

4.2.3 Tradeoffs

For both of our data structures, we've assumed they are implemented as a list of lists. But they act differently. The inner lists of the adjacency matrix will always be length $|V|$, and we can identify a desired vertex by its corresponding index. On the other hand, the inner lists of the adjacency list only include a vertex if the corresponding edge exists in the graph, rendering its indices meaningless, and each has a length of only outdeg(v_i) entries.

Below is a table demonstrating the tradeoffs:

Implementation	Space	Find Neighbors	Test Adjacency
Adj Matrix	$O(\|V\|^2)$	$O(\|V\|)$	$O(1)$
Adj List	$O(\|V\| + \|E\|)$	$O(\deg(v_i))$	$O(\deg(v_i))$

Here, we have not considered putting weights on edges, though we can easily modify our implementations to accommodate it. In this note, edge weights are irrelevant, though it will be necessary in the next note.

Generally, we prefer to use adjacency lists, as most graphs in practice are very large and sparse, meaning that there aren't many edges, adjacency lists are much more memory efficient and have a faster find neighbors operation. Thus, for the rest of this note, we will assume our graph is given as an adjacency list for any runtime calculations.

4.3 DFS

4.3.1 Inspiration

Suppose you are exploring a maze. Without any tools, you may end up walking in cycles and never explore the entire maze. The age-old method of systematically exploring a maze is to use a ball of string and a piece of chalk. The process can be summarized as going as far as you can until you see a dead-end while unraveling the ball of string and marking every junction in your path. At the dead-end, backtrack using your string until you reach a point where there exists an unmarked junction. Then repeat if there are still unexplored junctions. This guarantees that you reach every junction reachable from your start point.

Translating this into computer science, a junction is a vertex and a corridor between two junctions is an edge. The chalk is a length $|V|$ list of booleans, denoted 'visited', where each ith entry corresponds to whether or not we've visited v_i yet in our algorithm. The ball of string is a stack, where each entry is a vertex to begin or continue visiting. Though for clarity, we'll create the stack implicitly using recursion. Let's call this algorithm EXPLORE(v).

4.3.2 Explore/DFS Algorithm

To summarize, given a vertex v, EXPLORE(v) marks the vertex as visited, then recursively calls EXPLORE on each of its neighbors. By the nature of how recursive algorithms execute on a computer (i.e a stack, which is LIFO), this gives us the "depth-first" behavior. This holds for both undirected and directed graphs. Below is the pseudocode. We'll explain PREVISIT(v) and POSTVISIT(v) in the next section. [3]

Algorithm 11 Explore Algorithm

```
1: function EXPLORE(G, v)
2:     visited[v] ← True
3:     PREVISIT(v)
4:     for (v, w) ∈ E do
5:         if not visited[u] then
6:             EXPLORE(G,w)
7:     POSTVISIT(v)
```

EXPLORE finishes once it visits all the vertices reachable from the start vertex. This isn't necessarily the entire graph, as there can exist many vertices unreachable from the vertex. DFS solves this issue by just repeatedly restarting EXPLORE at a new, unvisited part of the graph.

Explicitly, DFS simply runs EXPLORE as a subroutine for all unvisited vertices. Below is the pseudocode for DFS. EXPLORE is also re-written for reference.

[3] Implementation details vary. In our implementation, for each neighbor, we only explore it if it hasn't been visited yet. In other implementations, we instead explore all its neighbors, and include a base case at the top of the function to immediately return if the node has already been visited. Other implementations may also opt to save space by bookkeeping if a node has been visited through modifying the graph in some way instead of explicitly tracking a visited list or set of vertices.

Algorithm 12 DFS

```
1: function DFS(G)
2:     visited ← [False] * |V|
3:     for v ∈ V do
4:         if not visited[v] then
5:             EXPLORE(G, v)
6: function EXPLORE(G, v)
7:     visited[v] ← True
8:     PREVISIT(v)
9:     for (v, u) ∈ E do
10:        if not visited[u] then
11:            EXPLORE(G,u)
12:    POSTVISIT(v)
```

Note that EXPLORE and DFS are essentially the same algorithm. We make a distinction between them to stay consistent with DPV, though many other sources will use DFS and explore synonymously.

Runtime: First, constructing the visited list takes $O(|V|)$ time. Second, for each vertex, EXPLORE looks through $O(\deg(v_i))$ edges. Since the sum of the degrees of a graph is $|E|$ (or, $2|E|$ for undirected graphs), this step yields $O(|V| + |E|)$. Thus in total, we have $O(|V| + |E|)$.

In other words, we are using an amortized analysis for the second part of our analysis - throughout the entire algorithm in total, each edge will only be considered twice for undirected graphs or once for directed graphs, yielding $O(|V| + |E|)$.

This runtime, $O(|V| + |E|)$, is often called linear time, where linear refers to "linear in terms of the vertices and edges".

4.3.3 Preorder/Postorder Numbers

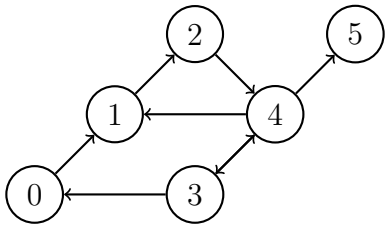
Now, let's return to PREVISIT and POSTVISIT. These are optional procedures performed at the first time you arrive at the vertex and the last time you leave the vertex, respectively. The naming is rather confusing as they are called during a visit (not before or after as the name suggests) - perhaps it'd be clearer as pre-(explore neighbors) and post-(explore neighbors).

In any case, as an example, you could define PREVISIT(v) to just print v. Here, running DFS will just print the vertices in the order you first reach each vertex.

To keep track of the order our DFS traverses each vertex, we'll keep track of a global 'clock' that we'll use to mark for each vertex when the DFS first arrives at it and last

leaves from it. These are called the preorder and postorder numbers. So, the interval $[\text{pre}[v], \text{post}[v]]$ is exactly the range of time that v was on the stack.

Example: The preorder and postorder numbers of DFS starting at vertex 0, assuming we break ties in favor of vertices with a smaller label:



Vertex	**preorder**$[v]$	**postorder**$[v]$
0	0	11
1	1	10
2	2	9
3	4	5
4	3	8
5	6	7

preorder$[v]$ - The first time we arrive at v.

postorder$[v]$ - The last time we leave from v.

Explicitly, we could implement this by defining in the main body of the DFS function a variable c (for 'clock') and two length V lists called pre and post that will contain the preorder and postorder number of v_i at the ith index for all i. In the body of the previsit or postvisit procedures, we increment c, then assign relevant index of the pre or post lists to c.

Note that these numbers are specific to a particular DFS traversal on a given start vertex. It doesn't make sense to refer to the preorder/postorder numbers of a graph, as there is a numbering for each start vertex and each of its possible tiebreaking schemes.

So, the pre/post order numbers tells us when each vertex in visited in the context of a DFS traversal. This information will be useful for understanding the intuition the algorithms later in this note.

4.3.4 DFS Trees

By nature of DFS never revisiting any vertices, the path that the DFS actually traverses through is acyclic and thus defines a tree, called a DFS tree. The start vertex of the DFS is known as the "root" of the tree.

Note that every time we finish exploring what is reachable, when we restart the DFS at a new section of the graph, this starts a new tree. So, in total we have a DFS forest.

We can classify all the edges of a graph with respect to a DFS tree.

For undirected graphs:

- *Tree edges* are the edges that we actually traverse in the DFS (i.e the edges in the DFS tree).
- *Non-tree edges* are all the other edges in the graph.

For directed graphs, we can further break down non-tree edges based on ancestry in the DFS tree. Informally, an ancestor of a vertex is defined as a parent, grandparent, great-grandparent, etc. A descendant of a vertex is defined as a child, grandchild, great-grandchild, etc. The root is the ancestor of every vertex:

- *Tree edges* are the edges that we actually traverse in the DFS (i.e the edges in the DFS tree).
- *Forward edges* are edges that go from a vertex to one of its descendants in the DFS tree to a descendant.
- *Back edges* are edges that go from a vertex to one of its ancestors in the DFS tree.
- *Cross edges* are edges that go from a vertex to neither ancestor nor descendant.

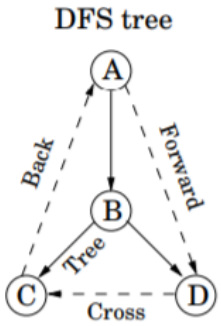


Figure 4.1: Types of DFS edges. Source: DPV, pg. 95

Note that both tree edges and forward edges go from an ancestor to its descendant. The difference is that a tree edge goes directly from a parent to a child.

Regarding cross edges, they could define any indirect familial relationship (e.g sibling, cousin, aunt/uncle, nephew/niece, etc.), or they could be from a completely different family (i.e another DFS tree in the DFS forest). For example, imagine adding a new vertex, E to the example above. Any edge from E to any other vertex would be a cross edge. Try experimenting with adding other vertices/edges to see the other familial relationships!

How can we compute these edge classifications directly in the DFS? For undirected graphs, during our DFS, any edge to an unvisited vertex is a tree edge, and the rest (i.e edges to a visited vertex) are non-tree edges.

For directed graphs, given two vertices u and v, what does it mean if u is an ancestor of v? u is an ancestor of v iff u is discovered first and v is discovered during EXPLORE(u) (either immediately or after some recursive calls). Remember, we have the relative times of when vertices are being explored (i.e on the stack) from the pre/post numbers! Here, $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$. So, this identifies tree/forward edges. We can also use this to identify back edges by symmetry - u is a descendant of v if v is an ancestor of u, so we can just swap u and v above. Finally, as cross edges mean there's no direct lineage, u and v should not be on the stack at the same time - v should have already finished.

Here is a precise summary in interval notation. '[' denotes pre and ']' denotes post:

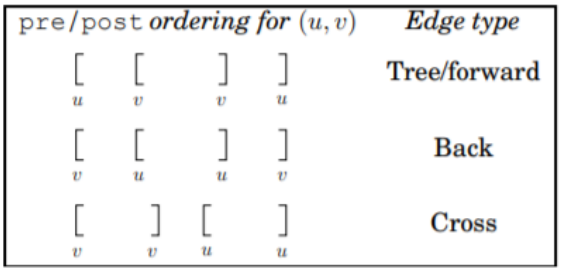


Figure 4.2: Types of DFS pre/postorder intervals. Source: DPV, pg. 95

Note that this is comprehensive, that is, no other orderings can possibly exist (i.e if u is placed on the stack before v, it must finish after v due to LIFO of a stack. The other case would be u completely finishing before v, but that contradicts how DFS works, as it should visit v before completing).

4.4 Cycle Detection

Now, with the framework given by the pre/post-order numbers and DFS tree edges, we'll move onto the applications of DFS. Cycle detection follows immediately from the previous discussion on DFS edge classification. Recall that a cycle occurs when there are two distinct paths between two vertices.

For undirected graphs, the moment we see a non-tree edge, our graph is therefore not a tree. Explicitly, when we loop over the neighbors in EXPLORE, if any neighbor is already visited, we can immediately return that there is a cycle.

For directed graphs, it is not as simple, as a non-tree edge can be either a forward, back,

or cross edge (i.e just because we see an already visited neighbor, unlike undirected graphs, it doesn't mean that neighbor can reach us!). We are only interested in back edges.

There exists a cycle iff there exists a back edge. This should make sense, since a back edge implies there is a path from an ancestor to a descendant, but also another path (by following the back edge) from a descendant to an ancestor. And given a cycle, running DFS from an arbitrary root node in the cycle yields the last edge a back edge, since it points to the root. So, we can just run DFS, keeping track of pre/post numbers, then scan the edges to see if there exists a pre/post ordering corresponding with a back edge.

There are other methods to detect cycles as well - check out the exercises.

4.5 Topological Sort/Linearization of a DAG

When a directed graph has no cycles at all, it is called a directed acyclic graph (DAG). DAGs are really interesting due to their hierarchical substructure - for example, treating each vertex as a task and each edge as a dependency, a DAG means there is an order you can do all your tasks (whereas if there is a cycle, there is no solution)[4]

Any vertex that doesn't have any incoming edges is called a source vertex, and any vertex that doesn't have any outgoing edges is a sink vertex. There can be multiple sources or sinks, and every DAG must have at least one source and at least one sink (it must start and end somewhere!). If you remove a sink, then informally, "the second to last" vertex becomes the new sink, and similarly for a source. If the graph consists of just a single vertex, it is both a source and a sink.

Given a DAG, we want to topologically sort it (also called linearization), which means to return an ordering of the vertices such that for every edge (u, v) in the graph, u comes before v in the ordering (as such, the first vertex in the topological ordering is a source, and the last vertex in the ordering is a sink). Thus, all dependency constraints are satisfied.

Example:

Note that the topological ordering is not unique (e.g if you have two sources, either one can come first).

To topological sort a DAG, we can simply just run a DFS starting at any vertex. Imagine running DFS on a DAG. You'll quickly see that a vertex cannot finish until its descendants finish. In other words, for all edges (u, v), $\text{post}[u] > \text{post}[v]$. We can see this as given two vertices u and v with an edge (u, v), try running DFS from v first

[4]In fact, the substructure given by a DAG forms the basis of all dynamic programming algorithms, which we'll see in a future note.

and running DFS from u first. In the former case, EXPLORE(v) will finish before we even start exploring u. And in the latter case, u calls EXPLORE(v), which will then finish before backtracking to u.

Alternatively, we can just see this is true from figure 2 in the DFS tree section, as in the two cases besides back edges (which can't exist in a DAG as they are acyclic), post[u] > post[v].

Note the highest postorder number is taken by a source vertex, and the lowest postorder number is taken by a sink. We can just return the vertices in order of decreasing postorder number.

4.6 Connectivity

4.6.1 Undirected Graphs

Recall two vertices u and v are connected to each other if there is a path between them (i.e u can reach v and v can reach u). In an undirected graph, if u can reach v, this implies v can reach u, by following the same path backwards.

An undirected graph can be decomposed into its connected components, that is, maximal disjoint subsets of the graph where every pair of vertices in a component can reach each other.

DFS directly yields us our connected components, as EXPLORE(v) visits everything reachable from v. Every time we get stuck, we've concluded a connected component. When we restart, we create a new connected component.

Explicitly, we can just define a list denoted '*cc*' and an index 'i' in the main body of the DFS. Every time we call EXPLORE on a new vertex in the outer loop, we increment the index. In the EXPLORE(v) function, we can assign append v to the current index of *cc*.

Note that each DFS tree is a *spanning tree* of a connected component.[5]

4.6.2 Directed Graphs - Kosaraju's Algorithm)

In directed graphs, connectivity is defined the same way (i.e u can reach v and v can reach u), but is now called strong connectivity. A directed graph is thus built up of strongly connected components (SCCs).

However, unlike undirected graphs, in a directed graph, if u can reach v, it does not necessarily mean v can reach u, which makes the strongly connected components trick-

[5]Spanning trees turn out to be very important in computer science. We'll cover spanning trees in detail in a future note.

ier to compute. A naive algorithm would be to run DFS from all vertices, keeping track of the set of vertices reachable from each vertex, then merging them into groups to retrieve strongly connected components. This would be $O(|V| * (|V| + |E|))$, which isn't great.

One important property of SCCs is that if we treat each SCC as a 'meta-node', and an edge between SCCs as a 'meta-edge', our resulting 'meta-graph' forms a DAG. This should make sense, as suppose it wasn't a DAG (i.e has a cycle). In that case, the cycle of meta-nodes should merge into a single meta-node, as the nodes in the cycle are strongly connected to each other.

As the metagraph is a DAG, it now has at least one source SCCs and at least one sink SCCs.

Example: SCC's

Below, we'll derive an intuitive algorithm that can be done in linear time, using DFS twice. Though feel free to skip to the algorithm.

Derivation

Try just running DFS on a graph at various start points. Ideally, DFS would traverse in a way such that the vertices belong to the same SCC, so we can isolate and extract it. However, DFS may very likely travel across to another SCC without fully exploring an SCC. How can we guarantee that we visit only nodes within a SCC?

Observation 1: Exploring sink SCC Suppose we already had our SCC metagraph (we don't actually yet, that's the goal of the algorithm. But it's useful for intuition). You may have noticed that if we start DFS from a vertex belonging to a sink SCC, then it can't accidentally leave the SCC without fully exploring the component, as there aren't any outgoing edges. So, for a vertex v in a sink SCC, EXPLORE(v) yields precisely the sink connected component. Then we can effectively "remove" this sink SCC, yielding a new sink SCC, and we can iterate from there.

However, we have no way of finding a vertex in a sink SCC. Unlike a normal DAG, iterating the adjacency list won't suffice, as the degrees of the vertices don't tell us anything here.

Observation 2: Generalizing a DAG property Recall this property in a DAG: for all edges (u, v), post$[u] >$ post$[v]$. We can generalize it with SCCs: Say we have two SCCs C and D. For all edges between two SCCs, the highest postorder number in C is greater than the highest postorder in D. We can justify this similarly - either DFS starts from a vertex in D or a vertex in C. In the former case, everything in D will finish before anything in C, thus every postorder number in C will be higher than anything in D. In the latter case, C cannot be finished exploring without travelling to D and

then backtracking to C (otherwise it would contradict reachability in EXPLORE). So, the highest postorder in C is greater than all the postorders in D.

Thus, the highest postorder number overall, by transitivity, must be in a source SCC.

However, note that the lowest postorder number is not necessarily in a sink vertex, as if we start DFS from a vertex that connects to a lower SCC, but it ends up travelling other vertices in the source first, the other vertices will get stuck and finish before ever travelling to the lower SCC (Exercise). So that doesn't work.

But we're looking for a sink SCC... luckily sources and sinks are quite similar - they are special being the first and last members of a DAG.

Observation 3: Reverse Graph If we reverse all the edges, then sinks become sources and sources become sinks. So, we use the reverse graph as a proxy - we can find a source in G^R by selecting the vertex with the highest postorder number, which gives us a sink in G that we desired for observation 1: running EXPLORE in a sink yields us exactly the sink connected component.

Furthermore, suppose we finish exploring this component. Recall from DAGs that removing a source transforms the second place vertex into a source. Referring back to our postorders again, the unvisited vertex with next highest postorder is the new source, as the relative ordering of the postorders remains fixed from observation 2. And it corresponds to a new sink in G. We can repeat this process to retrieve all the SCCs in reverse topological order of G.

Kosaraju's Algorithm

Thus, we have our algorithm:

1. Create a copy of G with the edges reversed, called G^R.
2. Run DFS on G^R, storing the postorder numbers
3. Run DFS on G, tiebreaking in decreasing order of the postorder numbers from G^R.

For the last step, just like in the undirected graph algorithm, we can just store the SCCs in a list '*scc*'. Every time we jump to a new unvisited vertex in the outer loop, that is a new SCC.

Notably, not only does this give us the strongly connected components, but it gets us them in reverse topological order! This should make sense, as we essentially generalized the topological sorting algorithm, with a few additional modifications.

Chapter 5

Shortest Paths

5.1 Introduction

Suppose you are driving in an unfamiliar city and are in need of gas. How can you find the nearest gas station, and the path to it?

Expressed mathematically, this is the problem of finding the minimum weight path from one source vertex s to all other vertices in a weighted directed graph G. Since the shortest path may not be well defined if there exists cycles with negative total weight, so we should also be able to detect these cases as well. In this note, we shall examine four different algorithms which solve this problem:

Name	**Assumptions**	**Runtime**
BFS	Edges with weight 1	$O(\|E\| + \|V\|)$
Dijkstra's	Nonnegative edge weights	$O((\|E\| + \|V\|) \log \|V\|)$
Bellman-Ford	None	$O(\|E\|\|V\|)$
DAG Shortest Paths	Graph is acyclic	$O(\|E\| + \|V\|)$

All of these algorithms use the same fundamental principles of updating known information when looking at an edge, and we shall use this paradigm throughout this note. For the same content presented with a different focus and order, refer to the textbook.

5.1.1 Setup

Define arrays dist and prev such that for vertex v, dist[v] and prev[v] are the distance and previous vertex of the best known path from s to v so far. At the start of the algorithm, the distance to s is set to zero and the distance to all other vertices is set to infinity. Also let $\textsc{adj}(u)$ return the edges leading out of u and $\textsc{w}(u, v)$ return the weight of the directed edge (u, v). (If the edge doesn't exist, let this be infinity.)

Let us examine the following helper function:

Algorithm 13 Update Function

```
1: function UPDATE(u, v)
2:     if dist[u] + W(u, v) < dist[v] then
3:         dist[v] = dist[u] + W(u, v)
4:         prev[v] = u
5:         return True
6:     return False
```

Effectively, this function checks if there is a better path to v by navigating to u first, then using the edge (u, v). The update function has a couple of convenient properties:

- dist[v] will never become less than the actual optimal distance to v.
- Excess calls will not negatively impact the shortest paths algorithm.
- If u comes just before v in the optimal path from s to v, and dist[u] is optimal, then dist[v] will be optimal after the function finishes.

With these properties, we have the following observation: Suppose the shortest path from s to v exists and is $s = u_0, u_1, u_2, \ldots, u_n = v$. If we call UPDATE$(u_{i-1}, u_i)$ for $i = 1, \ldots, n$ **in that order but not necessarily consecutively**, then dist[v] and prev[v] will contain the optimal path from s to v.

This means that in theory, all our shortest paths algorithm needs to do is find the right sequences of edges to update! Of course, we don't know that ahead of time; that's up to the algorithm to figure out. The general structure is as follows:

Algorithm 14 General Shortest Paths Algorithm

```
1: function SHORTEST_PATH(G, s)
2:     INITALIZE()
3:     other setup
4:     while not finished do
5:         (u, v) ← next edge
6:         UPDATE(u, v)
7:     return dist, prev
8: function INITIALIZE()
9:     dist ← [∞] * |V|
10:    prev ← [null] * |V|
11:    dist[s] ← 0
```

5.2 Bellman Ford

Suppose for now that our input graph has no negative cycles. This implies that for each vertex, there exists a shortest path from the source to it with no more than $|V|-1$ edges long. Otherwise, there would be a cycle of some sort that we could remove to get a path which is no worse.

Bellman-Ford uses the most direct approach to solve the shortest paths problem without any extra assumptions: it simply updates all edges $|V|-1$ times in total! This guarantees that all shortest paths from s will have its edges updated in the necessary order somewhere in the sequence. In particular, the ith edge along any particular shortest path is updated in the ith iteration of the outside loop, thus guaranteeing the desired relative order:

Algorithm 15 Bellman-Ford

```
1: function BELLMAN_FORD(G, s)
2:     INITALIZE()
3:     for i = 1, ..., |V| − 1 do
4:         for (u, v) ∈ E do
5:             UPDATE(u, v)
6:     return dist, prev
```

Well, what about negative cycles? Similar to the logic before, the length of a path to a negative cycle plus one traversal of said cycle is at most $|V|$ edges long. This means that if we update all edges one extra time, at least one element of dist will be reduced on the $|V|$th iteration if and only if a negative cycle exists somewhere in the graph. We can thus run the outer loop one extra time and see if any updates are successful to determine if there is a negative cycle.

As Bellman-Ford does a constant amount of work for each edge $|V|$ times, the total runtime is $O(|E||V|)$.

5.3 DAG Shortest Paths

Now we shall take a look at graphs with specific assumptions that will allow us to use more intelligent algorithms. In the case that the graph is acyclic, all paths from s will visit vertices in topological order. Thus, we can sort the vertices and update the edges in topological order. Unlike before, we do not have to worry about negative cycles because no cycles exist in the first place.

Algorithm 16 DAG Shortest Paths

```
1: function DAG_SHORTEST_PATHS(G, s)
2:     INITALIZE()
3:     u_1, u_2, ..., u_|V| ← TOPOLOGICAL_SORT(G)
4:     for i = 1, ..., |V| do
5:         for (u_i, v) ∈ ADJ(u_i) do
6:             UPDATE(u_i, v)
7:     return dist, prev
```

Topological sort takes $O(|E| + |V|)$ time, and we examine each vertex and edge once afterwards, so our total runtime is $O(|E| + |V|)$.

5.4 Breadth First Search

Now, we shift our focus to a different scenario; instead of making assumptions about the structure of G, we shall make assumptions about the edge weights. We shall start with the simplest assumption that all edge weights are 1. Note that this implies that there cannot be any negative cycles, as all cycles will have positive weight.

Here, the length of a shortest path is equal to the number of edges along that path. This means that a vertex of distance $k + 1$ from s must have at least one neighbor which is a distance of k from s. Conversely, to find the vertices $k + 1$ away from s, we only need to look at the neighbors of all vertices k away from s. Doing this iteratively with $k = 0, 1, \ldots, |V| - 1$, we can find the shortest path to all vertices.

Algorithm 17 Breadth First Search

```
1: function BFS(G, s)
2:     INITALIZE()
3:     visiting ← [s]
4:     to_visit ← []
5:     for k = 0, 1, ..., |V| − 1 do
6:         for u ∈ visiting do
7:             for (u, v) ∈ ADJ(u) do
8:                 if UPDATE(u, v) then
9:                     to_visit.ADD(v)
10:        visiting ← to_visit
11:        to_visit ← []
12:    return dist, prev
```

During the kth iteration, vertices in visiting are all k away from s, and we put vertices

we find that are $k+1$ away from s in to_visit. Here we are using UPDATE() to simultaneously check if a vertex has been visited before, and if not, update it.

Note that we can actually simplify our pseudocode a bit. Instead of moving collections around, we can simply use a queue which does the work of both visiting and to_visit by storing the current vertices of interest that we wish to explore. This results in cleaner code but more complex invariants:

- At any given time, the queue contains only vertices which are distance k or $k+1$ from s, for some integer k.
- For each k, there is some point in time where the queue contains all vertices of distance k and nothing else.

For more details, check out the textbook. Regardless of how we write our pseudocode, each vertex and edge is examined once so our runtime is $O(|E| + |V|)$.

5.5 Dijkstra's

Finally, we shall consider a more general case where we assume G has nonnegative edge weights. Because a minimum weight path may use more edges than a minimum edge path, we need to adjust our approach.

Similar to BFS, we know that a vertex which is x away from the source s must have at least one neighbor which is $\leq x$ away from s since the edge weights are nonnegative. Conversely, to find all vertices of distance x away, we need to examine the neighbors of all vertices with distance $< x$.

Suppose we are in the middle of our algorithm and have a set of vertices for which we already know the optimal distance. The next closest vertex v which is not in this set must be a neighbor of one of these vertices (call it u), and u must be the second to last vertex in the shortest path from s to v.

This inspires a functionally similar approach as BFS which repeatedly does the following:

- Choose the vertex with the shortest known path which we have not visited yet.
- Update the outgoing edges of the chosen vertex.

We can get the vertex with the shortest known path using a priority queue, and remove it from the queue once we visit it. We shall use the following priority queue operations:

- MAKE_QUEUE(V): Initializes and returns a priority queue with the vertices $v \in V$ as elements and dist[v] as the keys.
- Q.EMPTY(): Returns if the queue is empty.

- Q.DELETE_MIN(): Returns the element with the smallest key, and removes it.
- Q.DECREASE_KEY(v): Notifies the priority queue that the key for the following element has decreased.

All together, our pseudocode is as follows:

Algorithm 18 Dikjstra's

```
1: function DIJKSTRAS(G, s)
2:     INITALIZE()
3:     Q ← MAKE_QUEUE(V)
4:     while !Q.EMPTY() do
5:         u ← Q.DELETE_MIN()
6:         for (u, v) ∈ ADJ(u) do
7:             if UPDATE(u, v) then
8:                 Q.DECREASE_KEY(v)
9:     return dist, prev
```

For the sake of brevity, we shall ignore the possible implementations for a priority queue. One common implementation is a binary heap, which takes $O(|V|)$ to initialize, $O(1)$ to check for emptiness, and $O(\log |V|)$ for each of the other operations. We have $O(|E| + |V|)$ log time operations, for a total runtime of $O((|E| + |V|) \log |V|)$

Chapter 6

Greedy Algorithms and MSTs

6.1 Introduction

Suppose you are given a set of n items, each of which has a specific weight w_i, and a backpack which can hold a maximum total weight of W. What is the maximum number of items you can carry? One algorithm for solving such a problem is to repeatedly add the item with the least weight to the backpack. This is an example of a greedy algorithm: algorithms that repeatedly make a locally optimal choice.

Although greedy algorithms are not generally applicable due to their shortsightedness, they are simple and efficient when they do work. Generally the most interesting part of greedy algorithms is proving that they are globally optimal, which is often done with exchange arguments. In the following sections, we shall explore various problems that can be solved with greedy algorithms, and show how to prove their optimality.

6.1.1 Exchange Arguments

In this section, we shall take a look at the structure of exchange arguments used to prove the optimality of greedy algorithms using the example previously mentioned.

Let the items be sorted by increasing weight, so $w_1 \leq w_2 \leq \cdots \leq w_n$ (from here we refer to the items by their weights). Consider any optimal solution which doesn't take w_1 as its lightest item. Clearly, we can modify this solution by replacing its lightest item with w_1, as the number of items carried will remain the same and the total weight will either decrease or stay the same. Therefore, there exists some optimal solution which includes w_1, and we can place w_1 in our backpack with confidence.

We now repeat this by considering a similar problem with items $w_2, \ldots, w_n$ and total capacity $W - w_1$. By the same logic, there exists some optimal solution which uses w_2 as its lightest weight for this problem, or equivalently uses w_2 as its second lightest weight in the original problem. Therefore we are safe to place w_2 in our backpack next.

This logic can be applied iteratively until we run out of capacity, resulting in the greedy solution. Since we prove that the greedy solution is still as good as any other solution with every extra item we add, the greedy solution must be an optimal solution (though it might not necessarily be the only optimal solution).

Overall, this argument has an inductive flavor, and considers hypothetical exchanges to show that each step taken can lead to a valid optimal solution.

6.2 Minimum Spanning Trees

Imagine you are given a nonnegatively weighted undirected graph $G = (V, E)$. How can you select a subset of edges $E' \subseteq E$ such that $G' = (V, E')$ has a single connected component and the sum of the weights of the edges in E' is minimized? For example, this could be used to determine the minimum total cost of building roads between pairs of cities such that all cities are connected.

Clearly E' shouldn't contain any cycles, as we could remove any arbitrary edge within that cycle to get a better solution. At the same time, E' must connect the graph. Therefore, the optimal E' is a spanning tree of G, namely a tree that include all vertices of G. This problem of finding the spanning tree with smallest total edge weight cost is known as the minimum spanning tree problem.

(Note that the minimum spanning tree problem can also be defined if the weights may be negative. However, the optimal solutions of the two problems mentioned above may not be the same. We shall focus on finding the minimum cost spanning tree in this note.)

6.2.1 Cut Property

Before we can jump into algorithms, we need to first define a few useful concepts. Define a cut as any partition of the vertices V into two subsets S and $V - S$, and define an edge to cross the cut if it has one vertex in S and one vertex in V.

The min cut property states that, for any cut $(S, V \setminus S)$, the smallest weight edge crossing the cut is part of some minimum spanning tree. We prove this with an exchange argument:

Let e be an edge with lightest weight across the cut. Consider an arbitrary MST T. Let us add e to T; since T was a tree, adding e creates a cycle and there is now some other edge e' which crosses the cut and is redundant with e. If we now remove e', then we will get another spanning tree $T' = T \cup \{e\} \setminus \{e'\}$, with total weight $\text{weight}(T') = \text{weight}(T) + w(e) - w(e') \leq \text{weight}(T)$. Therefore, T' is a minimum spanning tree which includes e, so e is part of some minimum spanning tree.

6.2.2 Prim's Algorithm

Prim's algorithm starts with one vertex and builds up a minimum spanning tree from that vertex. More specifically, it starts with a set U containing any arbitrary vertex from V and an empty set T, then repeatedly does the following:

- Find the vertex $v \notin U$ with the smallest edge to any vertex $u \in U$.
- Add $\{u, v\}$ to T, and add v to U.

This works by applying the cut property to U and $V \setminus U$ at each step, so the final spanning tree stored in T is optimal. To run this algorithm efficiently, we can create a min heap for the vertices, with the keys being the minimum distance from the vertex to U, and update it as we run our algorithm. Each heap operation takes $O(\log |V|)$ time, and there may be up to one update per edge/vertex, so Prim's algorithm takes $O(|E| \log |V|)$ time to run.

6.2.3 Kruskal's Algorithm

While Prim's algorithm has a local focus, building up a single tree into a spanning tree, Kruskal's algorithm has a global focus. It starts with an empty set T, and repeats the following:

- Find the smallest edge e which won't create an cycle when added to T.
- Add e to T.

At each step, e is optimal because we can consider any cut which has all the vertices/edges of T on one side and has e crossing the cut. (This is thanks to the condition that e won't create a cycle, which guarantees at least one such cut exists.) We can then apply the cut property, as usual, to prove the optimality of Kruskal's as a whole.

To run Kruskal's algorithm efficiently, we will want to sort the edges by weight first, which takes $O(|E| \log |E|)$ time. We can then iterate through the edges in sorted order; checking if e would create a cycle is equivalent to checking if its vertices are already connected in T, which can be efficiently done with a disjoint set data structure at a cost of $O(\log^* |V|)$ time per operation and up to $|E|$ operations in total. Thus, the total runtime is dominated by sorting, and the algorithm takes $O(|E| \log |E|)$ time total.

While Prim's and Kruskal's leverage the same min-cut property for their optimality, they way they build up the minimum spanning tree is completely different. If the edge weights are all unique, then only one MST exists and both algorithms will find it. Kruskal's also can find any MST given proper tiebreaking.

6.3 Huffman Encoding

Let's suppose you have a message of length m using letters from some alphabet (say the modern Latin alphabet) of size n, and you wish to encode the message in binary (meaning a sequence of zeros and ones). What's the most efficient way to do so? Clearly one way is to assign each letter a k-length binary code word, where $k = \lceil \log_2 n \rceil$, then substitute for each letter its corresponding code word. However, if we use variable length code words and take into account the frequency of each letter, we can do better!

One thing to watch out for is that we want the decoding of the binary string to be unambiguous. To avoid issues, we shall only consider prefix-free codes, where no

codeword is a prefix of another codeword, Prefix-free codes can be represented using a binary tree, where leaves correspond to letters and the path from the root to a leaf corresponds to the codeword for that letter (e.g. a left corresponds to a zero, and a right corresponds to a one).

Using the tree representation, we can calculate the average encoding length per letter across the entire encoded message. Let f_i be the frequency of the ith letter, and d_i be its depth in the tree, meaning that the corresponding codeword is d_i bits long. The average length per letter is thus

$$\sum_{i=1}^{n} f_i d_i$$

This formula already suggests a greedy approach. Clearly the letters with the smallest frequencies must be at the deepest parts of the prefix tree, as otherwise we could use a swap to get a more efficient encoding.

WLOG let the first and second letters have the smallest two frequencies. These two letters must be at the bottom of the tree (say with depth d) and WLOG we can place them next to each other as children of the same internal node. We shall now treat these two letters as a single meta-letter in the tree at depth $d-1$ and with frequency $f_{\{1,2\}} = f_1 + f_2$, then run this algorithm recursively with the meta-letter and the other letters.

The reason this works is that the cost incurred by these two letters is $f_1 d + f_2 d$, which we can rewrite as $(f_1 + f_2)(d-1) + (f_1 + f_2)$ where the second part is simply a constant. Thus, we can write our problem as minimizing

$$\begin{aligned}\operatorname*{argmin}_d \sum_{i=1}^{n} f_i d_i &= \operatorname*{argmin}_d \left(f_1 d + f_2 d + \sum_{i=3}^{n} f_i d_i \right) \\ &= \operatorname*{argmin}_d \left(f_{\{1,2\}}(d-1) + \sum_{i=3}^{n} f_i d_i \right)\end{aligned}$$

which is equivalent to finding the optimal encoding using the meta-letter (combining the first and second letters) and the rest of the letters (from 3 to n). This recursive algorithm generates what is called a Huffman encoding, and is the optimal prefix-free encoding.

Chapter 7

Linear Programming, Flow, and Zero-Sum Games

7.1 Linear Programs

7.1.1 Introduction

Linear programs are a class of optimization problems that consist of a linear objective function and linear constraints. LPs are useful because they are widely applicable and have some powerful properties.

Example: Suppose you run a factory which produces products A and B. Product A sells at \$36 per unit, while product B sells at \$14 per unit. Due to limited machinery, you can only make up to 20 units of product A and 30 units of product B per day. In addition, both products use the same raw materials, of which your supply is limited, so you can only make 40 total units of product each day. What is the most amount of money you can make each day?

Let x_1 and x_2 be the number of units of product A and B we make per day, respectively. We can express this problem as the following linear program:

$$\begin{aligned} \max 36x_1 + 14x_2 : \\ x_1 &\leq 20 \\ x_2 &\leq 30 \\ x_1 + x_2 &\leq 40 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

Visually, this is equivalent to finding the point furthest along the direction $(36, 14)$ which is within the feasible set (the set of all points (x_1, x_2) which satisfy all constraints).

7.1.2 Standard Form

The standard form of a LP is

$$\begin{aligned} \max\, & c^\top x : \\ & Ax \leq b \\ & x \geq 0 \end{aligned}$$

Here, $x \in \mathbb{R}^n$ is a vector which we wish to find, while $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$ are matrices and vectors determined by the problem. The inequalities used are element-wise inequalities. In total, there are n scalar variables and m inequality constraints, excluding nonnegativity constraints.

Example: Express the previous example LP in standard form.

We can rewrite the LP as

$$\begin{aligned} \max & \begin{bmatrix} 36 \\ 14 \end{bmatrix}^\top \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} & \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 20 \\ 30 \\ 40 \end{bmatrix} \\ & \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq 0 \end{aligned}$$

7.1.3 Ways to Express Linear Programs

Sometimes linear programs may be expressed in different forms. Here we present a short list of ways to transform between different representations:

1. Minimization and Maximization: To flip between the two, multiply the coefficients of the objective function by -1.
2. Inequalities and Equalities: To convert inequalities of the form $a_i^\top x \leq b_i$ to equalities, introduce a slack variable s_i and use $a_i^\top x + s_i = b_i$, $s_i \geq 0$ instead. To convert an equality constraint $a_i^\top x = b_i$ to an inequality, simply replace it with two inequalities $a_i^\top x \leq b_i, a_i^\top x \geq b_i$. (While the latter method is mathematically correct, it can be numerically unstable.)
3. Unconstrained Variables: If a variable x_i is unconstrained, we can split it into two nonnegative variables x_i^+ and x_i^-. Whenever we have x_i in our original LP, replace it with $x_i^+ - x_i^-$. (Note that while we can convert back and forth without problem, there may be multiple values of x_i^+ and x_i^- such that $x_i = x_i^+ - x_i^-$.)

7.1.4 Duality

Example: Show algebraically that the optimal solution to the example problem is 1000 per day.

Consider the following:

$$\begin{aligned} &22(x_1 \leq 20) \\ &+14(x_1 + x_2 \leq 40) \\ \hline &=36x_1 + 14x_2 \leq 1000 \end{aligned}$$

By scaling and combining the inequalities, we have proven that the optimal value of the LP must be no more than 1000! Since we know a solution that achieves 1000, that solution is the optimal solution.

While here we happened to know exactly the right combination of inequalities to get our desired result, we can generalize this approach to get what is called the dual LP. Let us assign a nonnegative scaling factor y_i for each of the inequalities (excluding the nonnegativity inequalities). Adding all the inequalities up, we get

$$y_1(1x_1 + 0x_2) + y_2(0x_1 + 1x_2) + y_3(1x_1 + 1x_2) \leq 20y_1 + 30y_2 + 40y_3$$

If we find nonnegative y_i such that $y_1 + y_3 \geq 36$ and $y_2 + y_3 \geq 14$, then we can upper bound the objective function for our original LP (known as the primal LP) by $20y_1 + 30y_2 + 40y_3$. This results in the dual LP:

$$\begin{aligned} &\min \begin{bmatrix} 20 \\ 30 \\ 40 \end{bmatrix}^\top \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} &\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \geq \begin{bmatrix} 36 \\ 14 \end{bmatrix} \\ &\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \geq 0 \end{aligned}$$

Notice that the dual LP has exactly one variable for each constraint and one constraint for each variable, respectively, of the primal LP.

More generally, the relationship between a primal LP and a dual LP is as follows:

Primal:	Dual:
$\max c^\top x$	$\min b^\top y$
$Ax \leq b$	$A^\top y \geq c$
$x \geq 0$	$y \geq 0$

By the construction of the dual, the optimal value of the primal LP will always be less than or equal to the optimal value of the dual LP; this is known as weak duality. However, we can actually make a stronger statement: the optimal values of the primal and the dual are the same! This is known as strong duality, and while we will not prove it in general, we shall see some specific examples of strong duality later in this note.

7.1.5 LP Algorithms

The original algorithm use to solve LPs was the Simplex algorithm, which traversed from vertex to vertex greedily along the polytope defined by the LP inequalities. While it performs well in real life applications, it has a worst case exponential time complexity.

Later, interior-point algorithms (which move within the interior of the polytope rather than along its edges) were developed and proven to have polynomial time complexity. However, Simplex still is a popular method due to its efficiency in practical applications.

7.2 Max Flow and Min Cut

7.2.1 Modeling Max Flow

Suppose a utilities company needs to transport water through a series of (one way) pipes with different capacities from one point to another. What is the maximum rate at which it can do so?

We can model this problem as a graph problem. Create a source vertex s and sink vertex t for the origin and destination, and also create vertices where the pipes intersect. For each pipe, draw a directed edge and give it weight corresponding to its capacity. We wish to find the maximum flow where edge capacities are satisfied and flow is conserved at all intermediate vertices.

Mathematically speaking, each edge e has a nonnegative capacity c_e and a flow f_e along it. In order for edge capacities to be satisfied, we must have $0 \leq f_e \leq c_e$. In order for flow to be conserved, each vertex v except s and t must satisfy $\sum_{(u,v)\in E} f_{uv} = \sum_{(v,w)\in E} f_{vw}$. Finally, we wish to maximize the size of the flow, which is equal to the quantity of flow out of s (or equivalently the flow into t): $\sum_{(s,v)\in E} f_{sv}$. This leads to the natural representation of max flow as a LP, which we know how to solve.

7.2.2 Max Flow Algorithms

Clearly we can solve the max flow problem using standard LP algorithms. However, due to the special formation of the problem specific algorithms exist that are more efficient.

Suppose we have a graph G and source/sink vertices s, t on which we wish to compute maximum flow. Given any flow f, we can compute a residual graph G_f. For each edge $(u, v) \in E$ with flow f_{uv} and maximum capacity c_{uv} along that edge:

- Create an edge (u, v) in G_f with capacity $c_{uv} - f_{uv}$.
- Create an edge (v, u) in G_f with capacity f_{uv}.

The edge (u, v) tracks how much more flow can be pushed from u to v along the edge. Similarly, the edge (v, u) tracks how much flow currently going from u to v can be undone, which is equivalent to flow backwards from v to u.

We can then have the following greedy algorithm which repeatedly does the following:

- Find a path from s to t along the residual graph G_f. If no such path exists, halt.
- Add the maximum possible flow along this path to the current flow, and update the residual graph.

If all capacities are integers no greater than some constant C, then this program is guaranteed to terminate as the flow must increase by an integer amount each iteration. However, each iteration takes up to $O(|E|)$ time, and since the maximum flow is as large as $C|E|$, there may be as many as $C|E|$ iterations for a runtime of $O(C|E|^2)$! Alternatively, if we somehow know the maximum flow of the graph, F, ahead of time, then there will be at most F iterations each costing $O(|E|)$ time, resulting in a $O(F|E|)$ runtime. This is known as the Ford-Fulkerson algorithm.

We can actually do better if we add extra conditions on what path we select every iteration. If we choose the shortest path using BFS, then we are actually guaranteed $O(|E|^2|V|)$ time! This is known as the Edmonds–Karp algorithm. If we are more selective about the specific shortest path we choose, then we can achieve $O(|E||V|^2)$ time with Diniz's algorithm. We won't discuss the details of those algorithms here, but search them up if you're interested.

7.2.3 Max Flow Min Cut Duality

In this section we shall prove that the previous algorithm framework actually gives the optimal flow. First, some definitions:

- A (s, t)-cut is a partitioning of the vertices into two subsets L and R such that s is in S and t is in R.
- The capacity of a cut is the sum of the capacities of the edges crossing the cut from left to right i.e. edges from L to R.

Cuts are important because their sizes give upper bounds to any possible flow in a network. Intuitively, cuts can be thought of as cross sections of the graph, especially if we imagine the graph as a physical network of pipes. All flow from s to t must pass

through this cross section at least once somewhere, so the size of any cross section is an upper bound on the amount of flow possible.

The Max-Flow Min-Cut theorem states that the size of the maximum flow and the size of the minimum cut in a graph is the same. Consider the flow f output by our algorithm; let L be the set of vertices reachable from s in the residual graph G_f and R be the rest of the vertices. By the termination conditions of our algorithm, any edge from L to R in the original graph must be at maximum capacity, while any edge from R to L must have no flow. This implies that the size of the flow f is exactly equal to the size of the cut L, R, which proves the theorem.

With this theorem, we can finally prove that our algorithm is optimal. Since our algorithm simultaneously finds a flow and a (s, t)-cut which have the same size, the flow and cut it finds are optimal for the corresponding max flow and min cut problems!

In terms of linear programming, max flow and min cut are duals of each other, and the Max-Flow Min-Cut theorem is the manifestation of strong duality specifically for this type of problem. More specifically, the min cut problem has one variable per vertex which indicates whether it is in the left or right side of the cut, and one variable per edge which indicates whether it spans the cut.

One small asterisk is that the max flow dual has variables that are real valued, while the min cut problem is actually an instance of 0-1 integer linear programming, where all variables are boolean values of 0 or 1. However, it turns out that the optimal solution to the max flow dual always has variables that are boolean valued, so everything works out fine. (This is because the max flow LP satisfies certain mathematical properties which guarantees the integrality of the optimal solution, though we won't discuss them here.)

7.3 Zero Sum Games

7.3.1 Modeling Zero Sum Games

In this section, we shall examine matrix games, which are played by two opposing players, known as the row and column players, respectively. Each of the player picks a row/column of the matrix, respectively, and the result of the game is the value at the specified row and column of the matrix. The row player's objective wants to maximize this value, while the column player wants to minimize it.

One classic example would be rock paper scissors; letting rock be the first row/column, paper be the second, and scissors be the third, we can represent the game with the

following matrix:

$$G = \begin{bmatrix} 0 & -1 & +1 \\ +1 & 0 & -1 \\ -1 & +1 & 0 \end{bmatrix}$$

If the row player picks paper and the column player picks scissors, then the row player would win, as indicated by the $+1$ at the second row and first column.

From this example, we can see that there may not necessarily exist optimal pure strategies for either player, in which they always select one option. However, there is always a optimal mixed strategy for each player, in which they select each option with some probability. For a matrix game $G \in \mathbb{R}^{m \times n}$, if the row player chooses row i with probability r_i and the column player chooses column j with probability c_j, then the expected payoff is $\sum_{i,j} G_{ij} r_i c_j = r^\top G c$, where $r \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$ are column vectors containing respective probabilities.

7.3.2 Column and Row Player Duality

Suppose instead of players making decisions secretly, the one player is forced to announce their strategy to the other.

While this appears to be a disadvantage, we shall see that it doesn't actually matter thanks to LP strong duality!

Let us assume the column player is forced to announce their strategy; in other words we wish to find $\min_c \max_r r^\top G c$ with the proper constraints. In this case, there exists an optimal pure strategy for the row player in response. We can therefore model the game as the following LP:

$$\begin{aligned} &\min z \\ z &\geq \sum_j G_{ij} c_j \, \forall i \\ c &\geq 0 \\ \sum_j c_j &= 1 \end{aligned}$$

where z is a dummy variable used to represent the best possible payoff given a pure strategy from the row player.

On the other hand, the row player announcing their strategy first results in the following

LP:

$$\begin{aligned} &\max z \\ z &\leq \sum_i r_i G_{ij} \, \forall j \\ r &\geq 0 \\ \sum_i r_i &= 1 \end{aligned}$$

These two LP's happen to be duals of each other! This means that both players have mixed strategies that can guarantee the best payoff for themselves, regardless of how their opponent reacts.

Chapter A

Exercises

A.1 Chapter 1: Efficient Algorithms

1. Justify each of the statements in the Big O section of this note.
2. Find a tight bound for the runtime of Selection sort. Selection sort iteratively finds the minimum element of the active list, moves it to the front, then removes it from the active candidates.
3. Find a tight bound for the summation $\sum_{i=0}^{n} 2^i$.
4. Improve the linear time fibonacci algorithm above to $O(1)$ space instead of $O(n)$.
5. Can the bit complexity of addition be improved? Justify why or why not.

A.2 Chapter 2: Divide and Conquer

1. Using Karatsuba's algorithm, find the product of $10 * 11$ (show your work).
2. Use the Master Theorem to solve the following recurrence relations:
   (a) $T(n) = 8T(n/2) + O(n^2)$
   (b) $T(n) = 4T(n/4) + O(n)$
   (c) $T(n) = 4T(n/2) + O(n^5)$
3. Devise a divide and conquer algorithm very similar to Mergesort to find the k smallest numbers in $O(nlogk)$ and analyze its runtime.
4. Run the Quickselect algorithm to find the 3rd smallest number (1-indexed) on $A = [3, 5, 2, 4, 1]$. Experiment what the algorithm's execution would be on different choices of pivots.
5. We define a peak element as an element that is at least as large as all of its neighbor(s). Given a list, devise an algorithm to return an index of a peak element (if there are multiple, just return one of them).

A.3 Chapter 3: The Fast Fourier Transform

1. Suppose you want to multiply a polynomial of degree 5 and a polynomial of degree 9 with each other. What root of unity do you need to use in the FFT algorithm?
2. Use the DFT to compute the product of $1 + 2x$ and $3 + 4x^2$.
3. Suppose we just used the $A(x) = A_e(x^2) + xA_o(x^2)$ trick recursively to evaluate a degree $n-1$ polynomial at n arbitrary points. What would the runtime of this algorithm be?
4. A *homogeneous* polynomial is defined to have terms all of the same degree, and can consist of multiple variables (e.g $x^5+2x^3y^2+9xy^4$). Find an efficient algorithm to multiply two homogeneous polynomials, each of two variables x and y.

A.4 Chapter 4: Graphs and Depth-First Search

1. How many times do we 'visit' u if u is of degree d in an undirected graph in a DFS traversal, using our pseudocode above? Let's define 'visit' intuitively how you'd expect; explicitly, we 'visit' u every independent instance of executing some amount of code within the explore(u) function.
2. Given a directed graph G and an edge (u, v) in G, find an efficient algorithm to determine if a cycle exists with (u, v) in it.
3. Give an efficient algorithm to determine whether there exists a vertex s in a directed graph G such that every vertex in G is reachable from s.
4. https://leetcode.com/problems/number-of-islands/
5. How could you modify DFS slightly to identify back edges in a DFS tree without using the pre/post order numbers?
6. https://leetcode.com/problems/course-schedule/

A.5 Chapter 5: Shortest Paths

A.6 Chapter 6: Greedy Algorithms and MSTs

A.7 Chapter 7: Linear Programming, Flow, and Zero-Sum Games

Chapter B

Solutions

B.1 Chapter 1: Efficient Algorithms

1. Apply the limit test for each statement. The solutions for each individual property are omitted for brevity.
2. On the first iteration, Selection sort examines n elements, then $n-1$, then $n-2$, and so on down to the last element. The runtime is $n+n-1+n-2+\cdots+1 = n(n+1)/2 = O(n^2)$.
3. By the formula for a finite geometric series, the summation is equal to $2^{n+1}-1 = O(2^n)$.
4. Notice that to calculate $\text{FIB}(n)$, we only need the results of $\text{FIB}(n-1)$ and $\text{FIB}(n-2)$. Thus, it is inefficient to store an entire array of $\text{FIB}(x)$ for all x when we only need the last two entries. We can make do with just two variables that we iteratively update.
5. No. As a lower bound, to even read the inputs or write the answer requires $O(n)$ time, where n is the number of bits in the larger input.

B.2 Chapter 2: Divide and Conquer

1. $10 * 11 = 1 * 1 * 100 + (1 * 1 + 0 * 1) * 10 + 0 * 1 = 100 + 10 + 0 = 110$
2. Use the Master Theorem to solve the following recurrence relations:
   (a) $a = 8, b = 2, d = 2, d < \log_b a \rightarrow O(n^3)$
   (b) $a = 4, b = 4, d = 1, d = \log_b a \rightarrow O(n \log n)$
   (c) $a = 4, b = 2, d = 5, d > \log_b a \rightarrow O(n^5)$
3. Algorithm: Split the list into two sublists and find the k smallest numbers in each recursively. Merge the two sublists using the same operation as in Mergesort, but stop after k iterations. At the base case (i.e n/k lists of length k), use a sort to explicitly sort each lists

   Runtime analysis: Master theorem won't work here, since in our base case we do more work than in the rest of our subproblems. At the base case, we sort n/k sublists each of length k, yielding a runtime of $n/k * O(klogk) = O(nlogk)$

For the rest of the subproblems, the merge operation takes k time, as we stop at k iterations. Summing the work per recursion layer everything besides the base case then forms a geometric series $k + 2k + 4k + ... + n/2k * k$, which is $O(n)$. Thus, the runtime is dominated by the base case, $O(nlogk)$.

4. Here's one possibility. Suppose $v = 2$. We partition A into [1], [2], and [3,5,4], so we recurse into the sublist [3, 5, 4], now with $k = 1$. Suppose $v = 4$. Then, our partition yields [3], [4], and [5]. Recursing into [3] reaches our base case, so we return 3.

5. This is Spring 2021 Midterm 1 Question 2. Please refer to the solutions here: https://tbp.berkeley.edu/exams/7225/download/

Feedback Form: https://forms.gle/cM1io6eXyH65ytMP8

B.3 Chapter 3: The Fast Fourier Transform

1. The end result of the product will be a degree 14 polynomial, which is determined with 15 points. Recall that the number of points we evaluate our polynomials at is always a power of two, so we round up and use the 16th roots of unity.

2. Since the end polynomial is of degree 3 and is determined by 4 points, we use the 4th roots of unity. We first evaluate the input polynomials at the 4th roots of unity:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 1+2i \\ -1 \\ 1-2i \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 3 \\ 0 \\ 4 \\ 0 \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \\ 7 \\ -1 \end{bmatrix}$$

We then compute the pointwise product:

$$\begin{bmatrix} 3 \\ 1+2i \\ -1 \\ 1-2i \end{bmatrix} \circ \begin{bmatrix} 7 \\ -1 \\ 7 \\ -1 \end{bmatrix} = \begin{bmatrix} 21 \\ -1-2i \\ -7 \\ -1+2i \end{bmatrix}$$

Finally, we do polynomial interpolation with the inverse DFT:

$$\frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} 21 \\ -1-2i \\ -7 \\ -1+2i \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 4 \\ 8 \end{bmatrix}$$

Our answer is $3 + 6x + 4x^2 + 8x^3$.

3. Note that we have to be careful when using Master Theorem here, as the number of points where we are evaluating the polynomials stays static while the degree of the polynomials themselves get halved at each step. Since we're evaluating n points at each recursive step anyway, let's just compute the amount of time it takes to evaluate a single point and multiply by n.

   Let $T(n)$ be the amount of time taken to evaluate a degree n polynomial at a single arbitrary point. By splitting our polynomial into odd and even terms and evaluating those separately, we have $T(n) = 2T(n/2) + O(1)$. By Master Theorem, our runtime is $O(n)$.

   Thus, the total amount of time taken to evaluate a degree n polynomial at n arbitrary points is $nT(n) = O(n^2)$, which is no better than the naive approach. We also could have reached this conclusion by observing that evaluating a polynomial at any point trivially takes $\Omega(n)$ time, so evaluating at n arbitrary points takes $\Omega(n^2)$ time.

   Overall, this reaffirms what we know already: evaluating a polynomial at a single point must take linear time; the only way to save time is if we are evaluating a polynomial at many points with some sort of symmetry!

4. Let us first examine the properties of homogeneous polynomials of degree n with two variables x and y. By definition, any term in the polynomial must be some constant times $x^a y^b$, where $a + b = n$ and $a, b \geq 0$. Since there is only one degree of freedom, we can focus on the powers of x when multiplying two such polynomials, then put the powers of y back at the end.

   Thus, our algorithm consists of the following: given two polynomials of two variables x, y and of degree m and n respectively, remove all instances of y. Multiply the polynomials using FFT, then add back in powers of y to each term to ensure the degree of all terms in the result is $m + n$.

B.4 Chapter 4: Graphs and Depth-First Search

1. d times for $d \geq 1$, or 1 for $d = 0$. For $d \geq 1$, we visit u once when we first arrive at the u, and for the $d - 1$ other neighbors, we visit u again after finishing exploring the neighbor. For $d = 0$, the outer code in DFS will allow us to visit u once in its entirety.

2. Let the edge be (u, v). There exists a cycle with the edge in it if and only if u is reachable from v. This is achievable using DFS, and our algorithm will take linear time with respect to the number of edges and vertices.

3. We claim such a vertex s exists if and only if there is exactly one source strongly

connected component. If there exists one source component, then pick any vertex in that component to be s; clearly s will be able to reach all other vertices in the entire graph. If there is more than one source component, then no matter how s is picked at least one of those components will be entirely unreachable.

Thus, our algorithm consists of using the SCC-DAG algorithm to find the strongly connected components of G, then verifying that only one source exists. This takes linear time with respect to the vertices and edges.

If time permits, in a future date, we'll add extra sections on a stack implementation of DFS, alternate algorithms for DAG topological sort (Kahn's algorithm) and finding SCCs (Tarjan's algorithm).

B.5 Chapter 5: Shortest Paths

B.6 Chapter 6: Greedy Algorithms and MSTs

B.7 Chapter 7: Linear Programming, Flow, and Zero-Sum Games