CS170 Notes

Daniel Liu

Contents

0 Prologue 5

0.1 Fibonacci sequence . . . 5

0.2 Big-O notation . . . 5

1 Algorithm with numbers 6

1.1 Basic Arithmetic . . . 6

1.1.1 Addition . . . 6

1.2 Multiplication and division . . . 6

1.3 Modular Arithmetic . . . 6

1.3.1 Modular Addition and Multiplication . . . 6

1.3.2 Modular Exponentiation . . . 7

1.3.3 Euclid's Algorithm for Greated Common Divisor . . . 7

1.3.4 An Extension of Euclid's Algorithm . . . 7

1.3.5 Modular Division . . . 8

1.4 Primality Testing . . . 8

1.4.1 Generating Random Primes . . . 9

1.5 Cryptography . . . 9

1.5.1 Private-Key Schemes: One-time pad and AES . . . 9

1.5.2 RSA . . . 10

1.6 Universal Hashing . . . 10

1.6.1 Hashing . . . 10

1.6.2 Hash Functions . . . 10

2 Divide and Conquer Algorithms 12

2.1 Multiplication . . . 12

2.2 Recurrence Relations . . . 12

2.3 Mergesort . . . 12

2.4 Medians . . . 13

2.5 Matrix Multiplication . . . 13

2.6 Fast Fourier Transform . . . 14

2.6.1 An Alternative Representation of Polynomials . . . 14

2.6.2 Evaluation by Divide and Conquer . . . 14

2.6.3 Interpolation . . . 15

2.6.4 A closer look at the fast Fourier transform . . . 15

3 Decompositions of Graphs 16

3.1 Why graphs? . . . 16

3.1.1 How is a graph represented? . . . 16
3.2 Depth-first search in undirected graphs . . . 16
3.2.1 Exploring mazes . . . 16
3.2.2 Depth-first search . . . 17
3.2.3 Connectivity in undirected graphs . . . 17
3.2.4 Previsit and postvisit orderings . . . 17
3.3 Depth-first search in directed graphs . . . 17
3.3.1 Types of Edges . . . 17
3.3.2 Directly acyclic graphs . . . 18
3.4 Strongly connected components . . . 18
3.4.1 Defining connectivity for directed graphs . . . 18
3.4.2 An efficient algorithm . . . 19

4 Paths in Graphs **20**
4.1 Distances . . . 20
4.2 Breadth-first Search . . . 20
4.3 Lengths on Edges . . . 20
4.4 Dijkstra's Algorithm . . . 20
4.4.1 An Adaptation to BFS . . . 20
4.4.2 An Alternative Derivation . . . 21
4.4.3 Running Time . . . 21
4.5 Priority queue implementations . . . 22
4.5.1 Array . . . 22
4.5.2 Binary Heap . . . 22
4.5.3 d-ary heap . . . 22
4.6 Shortest paths in the presence of negative edges . . . 22
4.6.1 Negative edges . . . 22
4.6.2 Negative cycles . . . 22
4.7 Shortest Paths in DAGs . . . 22

5 Greedy Algorithms **23**
5.1 Minimum Spanning Trees . . . 23
5.1.1 A Greedy Approach . . . 23
5.1.2 The cut property . . . 23
5.1.3 Kruskal's Algorithm . . . 24
5.1.4 A Data Structure for Disjoint Sets . . . 24
5.1.5 Prim's Algorithm . . . 25
5.2 Huffman Encoding . . . 25
5.3 Horn Formulas . . . 26

5.4 Set Cover 27

6 Dynamic Programming **28**

6.1 Shortest Paths in DAGs, Revisited 28

6.2 Longest Increasing Subsequence 28

6.3 Edit Distance 29

6.4 Knapsack 29

6.4.1 Knapsack with Repetition 30

6.4.2 Knapsack without Repetition 30

6.5 Chain Matrix Multiplication 30

6.6 Shortest Paths 31

6.6.1 Shortest Reliable Paths 31

6.6.2 All-pairs Shortest Paths 31

6.7 The Traveling Salesman Problem 32

6.8 Independent Sets in Trees 32

0 Prologue

0.1 Fibonacci sequence

Definition 0.1 (Fibonacci Sequence).

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

The naive approach is to return `fib1(n-1) + fib1(n-2)`. Let $T(n)$ be the number of computer steps to compute `fib1(n)`. We know that:

- $T(n) \leq 2$ for $n \leq 1$.
- $T(n) = T(n-1) + T(n-2) + 3$ for $n > 1$. There are three computer steps, two checks and one addition.

$T(n)$ is therefore exponential in n, so the algorithm is impractically slow.

We can rewrite the code in polynomial time. First, create an array of size n, and set `arr[0] = 0, arr[1] = 1` and do `arr[i] = arr[i - 1] + arr[i -2]` in a linear for loop. This algorithm is linear in n, which is a reasonable runtime.

0.2 Big-O notation

We will express runtime as a function of the size of the input.

- O is the *upper bound*, an analog of $f \leq g$. $f = O(g)$ if $\exists c > 0, f(n) \leq c \times g(n)$
- Ω is the *lower bound*, an analog of $f \geq g$. $f = \Omega(g)$ if $\exists c > 0, f(n) \leq c \times g(n)$
- Θ is the *tightest bound*, an analog of $f = g$. $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$

We do not just use the total number of steps that a computer runs; there are a few rules that are used to further simplify.

1. Multiplicative constants are omitted. For example, $14n^2$ becomes n^2.
2. n^a dominates n^b if $a > b$. For example, n^2 dominates n.
3. Any exponential dominates any polynomial. For example, 3^n dominates n^5.
4. Any polynomial dominates any logarithm. For example, n dominates $(\log n)^3$

1 Algorithm with numbers

1.1 Basic Arithmetic

1.1.1 Addition

It is a basic rule that the sum of any three single digit numbers is at most two digits long. This rule allows us to define a method of addition that works in any base, since we know each carry is always a single digit and at any step at most three single digit numbers are added.

With k digits in base b, we can express numbers up to $b^k - 1$. To represent the number $N \geq 0$ in base b, we find that $\lceil \log_b(N+1) \rceil$ digits are needed to write N in base b. To convert logarithms from base a to base b, we use the equation $\log_b N = (log_a N)/(log_a b)$. The size of an integer N in base a is its size in base b times a constant factor $\log_a b$. Therefore, the size of an integer in a different base in big-O notation is simple $O(\log N)$.

We want to find out how long it takes to add two binary numbers x and y. Suppose x and y are both n bits long, then the sum of x and y is at most $n+1$ bits, and each individual bit is constant time. The total runtime of the algorithm is linear and in the form $c_0 + c_1 n$, where c_0 and c_1 are some constant.

1.2 Multiplication and division

The traditional human method of multiplication takes $O(n^2)$ runtime. Another method is to line up the two numbers x and y, floor divide x by 2 and double y. Keep going until $x = 1$. Then strike out all of the rows where x is even, and add the remaining ys. This method can be represented by the following recursive formula:

$$x \cdot y = \begin{cases} 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is even} \\ x + 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is odd} \end{cases}$$

Each recursive call requires a division by 2 (right shift), a test for odd/even, a multiplication by 2 (left shift), and possibly one addition, which is $O(n)$ bit operations. The algorithm terminates after $O(n)$ recursive calls, since each call reduces the number of bits by 1. Therefore, the total runtime of this algorithm is $O(n^2)$.

For division, to divide an integer x by another integer $y \neq 0$ means to find a quotient q and a remainder r where $x = yq + r$ and $r < y$.

1.3 Modular Arithmetic

Modular arithmetic deals with restricted ranges of integers. x *modulo* N is the remainder when x is divided by N. If $x = qN + r$, with $0 \leq r < N$, then $x \mod N = r$.

Proposition 1.1 (Substitution Rule). If $x \equiv x' \mod N$ and $y \equiv y' \mod N$, then
$x + y \equiv x' + y' \mod N$ and $xy \equiv x'y' \mod N$

Additionally, Associativity, Commutativity, and Distributivity all hold for modular arithmetic.

1.3.1 Modular Addition and Multiplication

To add two numbers x and y modulo N, we start with regular addition. If the sum exceeds $N-1$, we just need to subtract N. Therefore, the runtime is $O(n)$, where $n = \lceil \log N \rceil$.

To multiply two mod-N numbers, x and y, we can use regular multiplication and reduce the answer modulo N. The resulting product can be as large as $(N-1)^2$, but this is still at most $2n$ bits long. We then can compute the remainder by dividing it by N, which takes $O(n^2)$.

1.3.2 Modular Exponentiation

We wish to calculate $x^y \mod N$ for large values of x, y, and N. To ensure the numbers don't grow too large, we can repeatedly multiply by $x \mod N$ until reaching $x^y \mod N$.

$$x \mod N \to x^2 \mod N \to \cdots \to x^y \mod N$$

This algorithm is exponential in the size of y.

We can do better by starting with x and squaring repeatedly modulo N.

$$x \mod N \to x^2 \mod N \to x^4 \mod N \to \cdots \to x^{2^{\lfloor \log y \rfloor}} \mod N$$

Each square takes $O(\log^2 N)$ to compute, with $\log y$ multiplications. To determine $x^y \mod N$, we need to separate y into its powers of 2. For example:

$$x^{25} = x^{16} \cdot x^8 \cdot x^1$$

Rewriting this idea recursively:

$$x^y = \begin{cases} (x^{\lfloor y/2 \rfloor})^2 & \text{if } y \text{ is even} \\ x \cdot (x^{\lfloor y/2 \rfloor})^2 & \text{if } y \text{ is odd} \end{cases}$$

Let n be the size in bits of x, y, and N (whichever is largest). The algorithm halts after at most n recursive calls, multiplying n bit numbers every call. Therefore the runtime of exponentiation is $O(n^3)$

1.3.3 Euclid's Algorithm for Greated Common Divisor

Given two integers a and b, we want to find the largest integer that divides both of them, also known as the *Greatest Common Divisor*.

Theorem 1.2 (Euclid's Rule). If x and y are positive integers with $x \geq y$, then $gcd(x, y) = gcd(x \mod y, y)$.

Proof. If we can show $\gcd(x, y) = \gcd(x - y, y)$, above statement can be derived by repeatedly subtracting y from x. Any integer that divides x and y must divide $x - y$, so $\gcd(x, y) \leq \gcd(x - y, y)$. Any integer that divides $x - y$ and y must divide both x and y, so $\gcd(x, y) \geq \gcd(x - y, y)$. ■

Lemma 1.3. If $a \geq b$, then $a \mod b < a/2$.

Proof. If $b \leq a/2$ then $a \mod b < b \leq a/2$, else $b > a/2$ and $a \mod b = a - b \leq a/2$. ■

This means that after any two consecutive rounds of Euclid's rule, a and b are at least halved in value or their length decreases by one bit. Two n-bit integers will reach the base case by $2n$ calls. This gives a total runtime of $O(n^3)$, since each call takes quadratic-time for division, and there are at most $2n$ calls.

1.3.4 An Extension of Euclid's Algorithm

Lemma 1.4. If d divides a and b, and $d = ax + by$ for $\exists x, y$, then $d = \gcd(a, b)$

Proof. d is a common divisor of a, b, so $d \leq \gcd(a, b)$. Since $\gcd(a, b)$ is common divisor of a, b, it must divide $ax + by = d$, which implies $gcd(a, b) \leq d$. Therefore $\gcd(a, b) \leq d$ ■

Lemma 1.5. For any positive integers a and b, the extended Euclid algorithm returns integers x, y, d such that $\gcd(a, b) = d = ax + by$

Proof. We attempt to prove this by induction.

1. Base case $b = 0$, returns a which is $\gcd(a, b)$.
2. Inductive hypothesis: Assume $\gcd(b, a \mod b) = bx' + (a \mod b)y'$.
3. Inductive step: $(a \mod b) = (a - \lfloor a/b \rfloor b)$, $d = \gcd(a, b) = \gcd(b, a \mod b) = bx' + (a \mod b)y' + bx' + (a - \lfloor a/b \rfloor b)y' = ay' + b(x' - \lfloor a/b \rfloor y')$. Therefore $d = ax + by$ with $x = y'$ and $y = x' - \lfloor a/b \rfloor y'$

■

1.3.5 Modular Division

We say that x is the multiplicative inverse of $a \mod N$ if $ax \equiv 1 \mod N$. There can be at most one $x \mod N$ that satisfies this condition, denoted by a^{-1}. However, the multiplicative inverse may not always exist. For example, if a and N are both even, $a \mod N = a - kN$, so the inverse does not exist. Because $\gcd(a, N)$ divides $ax \mod N$, then if $gcd(A, N) > 1$, $ax \not\equiv 1 \mod N$ When $\gcd(a, N) = 1$, the extended Euclid can give us x, y where $ax + Ny = 1$, so $ax \equiv 1 \mod N$.

Theorem 1.6. For any $a \mod N$, a has a multiplicative inverse if and only if it is relatively prime to N. If the inverse exists, it can be found in $O(n^3)$ time by running the extended Euclid algorithm.

1.4 Primality Testing

Factoring is hard, but determining primality is easy. We hope to find some test that determines if a number is prime without trying to factor the number.

Theorem 1.7 (Fermat's Little Theorem). If p is prime, then for every $1 \leq a < p$, $a^{p-1} \equiv 1 \mod p$

Proof. Let S be the nonzero integers modulo p, $S = \{1, 2, \ldots, p-1\}$. If we multiply the numbers in S by $a \mod p$, we just permute them. This is because none of the numbers can equal each other when multiplying by $a \mod p$, since they are unique from the start. Therefore, $S = \{1, 2, \ldots, p-1\} = \{a \cdot 1 \mod p, a \cdot 2 \mod p, \ldots, a \cdot (p-1) \mod p\}$. Multiplying all the elements together, we get $(p-1)! \equiv a^{p-1} \cdot (p-1)! \mod p$. Dividing by $(p-1)!$, we get the theorem. ■

This proof suggests that if $a^{N-1} \equiv 1 \mod N$, for $a < N$, then N is prime. However, Fermat's Theorem is not if and only if, so it does not work with certain choices of a.

Lemma 1.8. If $a^{N-1} \not\equiv 1 \mod N$ for a relatively prime to N, it must hold for at least half the choices of $a < N$

Proof. Fix some value for a which $a^{N-1} \not\equiv 1 \mod N$. Every element $b < N$ that passes Fermat's test with respect to N has a twin, $a \cdot b$ that fails the test: $(a \cdot b)^{N-1} \equiv a^{N-1} \cdot b^{N-1} \equiv a^{N-1} \not\equiv 1 \mod N$. All elements $a \cdot b$ for a fixed a but different b are distinct, so there is a one-to-one function $b \mapsto a \cdot b$, where there are at least as many elements which fail the test as pass it. ■

We can then test primality with a low error probability by selecting k numbers less than N, then testing each one with N. The probability that the algorithm returns yes when N is not prime is $\leq \frac{1}{2^k}$.

1.4.1 Generating Random Primes

Primes are very common. A random n bit number has around $\frac{1}{n}$ chance of being prime.

Theorem 1.9 (Lagrange's Prime Number Theorem). Let $\pi(x)$ be number of primes $\leq x$. Then $\pi(x) \approx x/(\ln x)$, or $\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1$

To generate a random n-bit prime, we can just pick a random n-bit number N, run the primality test on it, then output N if it passes the test else repeat the process.

Definition 1.10 (Carmichael numbers). There are non-prime numbers that can fool Fermat tests for all a, known as *Carmichael numbers*. We can use the following algorithm to check if N is composite.

- Write $N - 1$ in the form $2^t u$.
- Choose a random base a and check the value of $a^{N-1} \mod N$ by starting at $a^u \mod N$ and repeatedly squaring.
- If $a^{N-1} \not\equiv 1 \mod N$, then N is composite.
- Otherwise somewhere in the test we must have run into a 1. If this is after the first position and the preceding value was not -1, then N is composite.

Combining with the earlier Fermat test, at least three-fourths of the possible values of a between 1 and $N-1$ will reveal a composite N, even with Carmichael numbers

1.5 Cryptography

Suppose Alice wants to send a message to Bob, but Eve can intercept the message and try to decrypt it.

1.5.1 Private-Key Schemes: One-time pad and AES

The encryption function e is an invertible function whose inverse is the decryption function d.

Definition 1.11 (One-time Pad). Alice and Bob must meet beforehand to choose a binary string r with n bits. Alice's encryption is the bitwise exclusive-or, or $e_r(x) = x \oplus r$. $e_r(x)$ is its own inverse: $e_r(e_r(x)) = (x \oplus r) \oplus r = x \oplus 0 = x$ To be secure, we choose r at random, so Eve can get no information about x without knowing r. The downside to this method is that it is only a one-time use, since sending another message gives Eve $x_1 \oplus x_2$, which could reveal important information such as repeated bits.

1.5.2 RSA

In public-key cryptography, anybody can send a message to anybody else using publically available information.

Theorem 1.12. Pick two primes, p and q, and let $N = pq$. For any e relatively prime to $(p-1)(q-1)$ and $d = e^{-1} \mod (p-1)(q-1)$, the following is true:

- The mapping $x \mapsto x^e \mod N$ is a biject on $\{0, 1, \dots N-1\}$
- $\forall x \in \{0, \dots, N-1\}$, $(x^e)^d \equiv x \mod N$.

Proof. If the mapping $x \mapsto x^e \mod N$ is invertible, it must be a bijection, so statement 2 implies statement 1. To prove statement 2, e is invertible modulo $(p-1)(q-1)$ since it is relatively prime. Since $ed \equiv 1 \mod (p-1)(q-1)$, $\exists k : ed \equiv 1 + k(p-1)(q-1)$. We can show $x^{1+k(p-1)(q-1)} - x \equiv 0 \mod N$ by Fermat's little theorem, since $x^{p-1} \equiv 1 \mod p$ and $x^{q-1} \equiv 1 \mod q$, so this is also divisible by N. ■

1.6 Universal Hashing

1.6.1 Hashing

Hashing is a method of storing data items in a table to support insertions, deletions, and lookups. We give a short "nickname" to each of the items, and each item with the same "nickname" stored in a linked list.

1.6.2 Hash Functions

It is difficult to create a good hash function, since it must be "Random" enough to scatter data round but "consistent" enough to get the same result every time.

Definition 1.13. Let the number of buckets be a prime number. For any four coefficients $a_1, \dots, a_4 \in \{0, 1, \dots, n-1\}$, define $h_a(x_1, \dots, x_4) = \sum_{i=1}^{4} a_i \cdot x_i \mod n$

Theorem 1.14. Consider any pair of distinct IP addresses $x = (x_1, \dots, x_4)$ and $y = (y_1, \dots, y_4)$. If the coefficients $a = (a_1, a_2, a_3, a_4)$ are chosen uniformly at random from $\{0, 1, \dots, n-1\}$, then $\Pr\{h_a(x_1, \dots, x_4) = h_a(y_1, \dots, y_4)\} = \frac{1}{n}$. In other words, the chance that x and y collide under h_a is the same as it would be if each were assigned nicknames randomly and independently

Proof. Since $x = (x_1, \dots, x_4)$ and $y = (y_1, \dots, y_4)$ are distinct, the quadruples must differ in some component. Assume that $x_4 \neq y_4$. We wish to compute the probability $\Pr\{h_a(x_1, \dots, x_4) = h_a(y_1, \dots, y_4)\}$, or that the probability $\sum_{i=1}^{4} a_i \cdot x_i \equiv \sum_{i=1}^{4} a_i \cdot y_i \mod n$. This can be rewritten as $\sum_{i=1}^{3} a_i \cdot (x_i - y_i) \equiv a_4 \cdot (y_4 - x_4) \mod n$ Looking for probability that the last drawn number a_4 is such that the above equation holds. If the left-hand side evaluates to c, a_4 must be $c \cdot (y_4 - x_4)^{-1} \mod n$ out of n possible values, which is with probability $1/n$. ■

Let $\mathcal{H} = \{h_a : a \in \{0, \dots, n-1\}^4\}$, so $\mathcal{H}$ is a random hash function from our family. For any two distinct data items x and y, exactly $|\mathcal{H}|/n$ of all hash functions in $\mathcal{H}$ map x and y to the same bucket, where n is the number of buckets.

A family of hash functions with this property is called universal. This means that for any two data items x and y, the chance of collision is the same as if we map x and y to buckets uniformly at random. We can apply this any general table. First choose a table size n to be some prime number that is larger than the number of expected items in the table. Next assume the size of the domain of all items is $N = n^k$, a power of n. Then each data item can be consider a k-tuple of integers modulo n, and $\mathcal{H} = \{h_a : a \in \{0, \dots, n-1\}^n\}$ is a universal family of hash functions.

2 Divide and Conquer Algorithms

Divide and Conquer refers to a set of algorithms which involve breaking a problem into subproblems that are smaller instances of the same type of problem, recusively solving the subproblems, and appropriately combining the answers.

2.1 Multiplication

We can apply this idea to regular multiplication. Let x and y be two n-bit integers, with n being a power of 2. If x and y are written in binary, we can split them both in half and rewrite as $2^{n/2}x_L + x_R$, $2^{n/2}y_L + y_R$. Then $xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_l + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$. This algorithm requires four recursive calls for $x_L y_L$, $x_L y_R$, $x_R y_L$, $x_R y_R$, so $T(n) = 4T(n/2) + O(n)$, which is a runtime of $O(n^2)$.

To optimize this, we can exchange $(x_L y_R + x_R y_L) = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$ for three multiplications instead. This only requires three recursive calls, so $T(n) = 3T(n/2) + O(n)$, giving runtime of $O(n^{1.59})$. The height of the tree is $\log_2 n$, the branching factor is 3, so at depth k there are 3^k subproblems. For each subproblem it takes a linear amount of work to combine answers. Therefore, the total time at depth k is $3^k \times O(\frac{n}{2^k}) = (\frac{3}{2})^k \times O(n)$. The work that is done increases geometrically by a factor of $3/2$ per level, so the overall runtime is $O(n^{\log_2 3}) \approx O(n^{1.59})$. We can see that a small difference in speed per level.

2.2 Recurrence Relations

Theorem 2.1 (Master Theorem)**.** If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for $a > 0, b > 0, d \geq 0$, then

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

Proof. Assume that n is a power of b for convenience. Notice the size of the subproblems decrease by a factor of b with each level, so the height is $\log_b n$ levels. The branching factor is a, so the kth level is made of a^k subproblems, each with size n/b_k. The total work done for a level is $a^k \times O(\frac{n}{b^k})^d = O(n^d) \times (\frac{a}{b^d})^k$. As k goes from 0 to $\log_b n$, the numbers form a geometric series with ratio a/b^d.

- If the ratio is less than 1, the series is decreasing and the sum is $O(n^d)$.
- If the ratio is greater than 1, the series is increasing and the sum is given by last tern $O(n^{\log_b a})$.
- If the ratio is exactly 1, all $O(\log n)$ terms are equal to $O(n^d)$.

■

2.3 Mergesort

The process of mergesort involves splitting the list into two halves, recursively sorting each half, then merging the two sorted sublists. Merges are linear, so the overall time taken by mergesort is $T(n) = 2T(n/2) + O(n) = O(n \log n)$

There is a $n \log n$ lower bound for sorting with comparisons. We can depict sorting algorithms as a tree, with each node representing a comparison and the leaves representing a permutation of the elements. There are therefore $n!$ leaves, so the optimal depth and runtime is $\log(n!)$. In the worse case, there are $\Omega(n \log n)$ comparisons, so it follows that mergesort is an optimal comparison-based sorting algorithm.

2.4 Medians

We want to find the median of a list of numbers. We can generalize this problem take in a list of numbers, S, and integer k, then return the kth smallest element of S. The median is the output if we input $k = \lfloor |S|/2 \rfloor$.

The divide and conquer strategy for this problem is as follows:

- For any number v, split S into three categories: S_L, S_V, S_R, which are lists of numbers less than, equal to, or greater than v.
- The search can be narrowed down to one of the sublists, since we know the size of each sublist and which element we want.

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

The worse case efficiency for this algorithm is $\theta(n^2)$, if v is always the largest or smallest number in S. On average, the runtime lies close to the best-case time.

Lemma 2.2. On average a fair coin needs to be tossed two times before a "heads" is seen.

Proof. Let E be the expected number of tosses before a heads is seen. We finish if we get heads first, else we need to repeat. Therefore, $E = 1 + \frac{1}{2}E$, so $E = 2$. ∎

Therefore after two split operations on average, the array will shrink to at most $3/4$ths of its size. Let $T(n)$ be the expected running time of the algorithm on an array of size n.

$$T(n) \leq T(3n/4) + O(n)$$

We can then conclude that $T(n) = O(n)$.

2.5 Matrix Multiplication

The product of two $n \times n$ matrices X and Y is an $n \times n$ matrix $Z = XY$, with (i, j)th entry equal to $Z_{ij} = \sum_{k=1}^{n} X_{ik}Y_{kj}$. We can find this by taking the dot product of the ith row of X and jth column of Y and iterating through the number of rows in X or the number of columns in Y. Using brute force on this problem implies a $O(n^3)$ runtime.

Using a divide and conquer strategy, we find that:

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

Therefore we can recursively compute $AE, BG, AF, BH, CE, DG, CF, DH$, and do $O(n^2)$ additions. The total run-time: $T(n) = 8T(n/2) + O(n^2)$, which is still $O(n^3)$. We can further simplify this to 7 $n/2 \times n/2$ 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 $P_1 = A(F - H), P_2 = (A + B)H, P_3 = (C + D)E, P_4 = D(G - E), P_5 = (A + D)(E + H), P_6 = (B - D)(G + H), P_7 = (A - C)(E + F)$. The new runtime for this problem is $T(n) = 7T(n/2) + O(n^2)$, which is a runtime of $O(n^{\log_2 7}) \approx O(n^{2.81})$.

2.6 Fast Fourier Transform

The product of two degree d polynomials $C(x) = A(x) \cdot B(x) = c_0 + c_1 x + \cdots + c_{2d}x^{2d}$, where $c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0 = \sum_{i=0}^{k} a_i b_{k-i}$.

2.6.1 An Alternative Representation of Polynomials

A degree-d polynomial can either be characterized by its values at any $d+1$ distinct points, $A(x_0), A(x_1), \ldots, A(x_d)$, or its coefficients, $a_0, a_1, \ldots, a_d$. To find $C(x)$ with value representation, it takes linear time. We just need to find $2d+1$ points, then at any point z find the value $A(z) \times B(z)$. We define two procedures from switching between the two representations. *Evaluation* is translating the input coefficients to values, and *Interpolation* is translating the value representation to integer coefficients.

2.6.2 Evaluation by Divide and Conquer

If we choose n points, $\{x_0, x_1, \ldots, x_n\}$ to be positive and negative pairs, then the computations for finding $A(x_i) \cdot B(x_i)$ overlaps with finding $A(-x_i) \cdot B(-x_i)$. Generally, $A(x) = A_e(x^2) + xA_o(x^2)$, when we group the polynomial powers into odd or even, then factor out x for the odd. Comparing the $A(x_i)$ and $A(-x_i)$ calculations, we see $A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2)$ and $A(-x_i) = A_e(x_i^2) - x_i A_o(x_i^2)$. Therefore, we can reduce the computations by only evaluating $n/2$ points at the positions $\{x_0^2, \ldots, x_{n/2-1}^2\}$. Each of these evaluations, $\{A_e(x_0^2), A_o(x_0^2), \ldots, A_e(x_{n/2-1}^2), A_o(x_{n/2-1}^2)\}$, are also evaluation problems. However, this trick only works at the top level, unless the $n/2$ evaluation points are also plus minus pairs for the next levels, which can only occur with complex numbers. We need to pick $n \geq (d+1)$ points where $n = 2^k$, $k \in \mathbb{Z}$. The initial n points should be the n complex solutions to $z^n = 1$, or the nth roots of unity. We can easily define the nth roots of unity as n evenly spaced points on a circle of radius 1. Using polar coordinates, the n roots are $\omega = e^{\frac{2\pi i}{n}}$. This means that ω^j and $\omega^{j+n/2}$ are plus minus paired, since they are across from each other on the circle. When we square the terms to evaluate $A_e(x^2)$ and $A_o(x^2)$ at $\{1, \omega^2, \omega^4, \ldots, \omega^{2(n/2)-1}\}$, we get the $n/2$ points of unity. We can think of this as if the spacing between the points in the circle of radius 1 are now twice as far apart, meaning that the points are placed down around the circle twice, leading to two pairs of points overlapping on each other. Putting this all together, we have the following recursive formula for the evaluation part of FFT:

1. Input: n coefficients that represent the polynomial, where n is a power of 2.
2. Base case: $n = 1 \Rightarrow P(1)$. At one point we simply need to evaluate P at the point.
3. Recursive step: Call Evaluate on $P_e(x^2)$ and $P_o(x^2)$ on the $n/2$ points of unity $\{\omega^0, \omega^2, \ldots \omega^{n-2}\}$. $P_e(x^2)$ has the coefficients $\{p_0, p_2, \ldots, p_{n-2}\}$, and $P_o(x^2)$ has the coefficients $\{p_1, p_3, \ldots, p_{n-1}\}$.
4. From these calls, we get the evaluated terms at those points, $y_e = \{P_e(\omega^0), \ldots, P_e(\omega^{n-2})\}$ and $y_o = \{P_o(\omega^0), \ldots, P_o(\omega^{n-2})\}$.
5. Now, we can calculate the evaluation at the original n points, using the following equations:
$$P(\omega^j) = P_e(\omega^{2j}) + \omega^j P_o(\omega^{2j}) \tag{1}$$
$$P(-\omega^j) = P_e(\omega^{2j}) - \omega^j P_o(\omega^{2j}) = P(\omega^{j+n/2}) = P_e(\omega^{2j}) + \omega^{j+n/2} P_o(\omega^{2j}) \quad \text{using } -\omega^j = \omega^{j+n/2} \tag{2}$$
6. We can also use the following fact to clean up the equation:
$$y_e[j] = P_e(\omega^{2j}) \tag{3}$$
$$y_o[j] = P_o(\omega^{2j}) \tag{4}$$
7. Return: We return the following evaluation at the nth roots of unity
$$y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$$

2.6.3 Interpolation

We found the values are equal to FFT(coefficients, ω). It turns out that the coefficients are $\frac{1}{n}$FFT(values, ω^{-1}). The matrix representation between the two representations is the Vandermonde matrix:

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

If $x_0, \dots, x_{n-1}$ are distinct numbers, then M is invertible. Evaluation is multiplication by M, while interpolation is multiplication by M^{-1}. We are able to invert the Vandermonde matrix in $O(n^2)$ instead of $O(n^3)$.

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \dots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

Theorem 2.3 (Inversion Formula). $M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$ Because ω^{-1} is also an nth root of unity, interpolation or multiplication by $M_n(\omega)^{-1}$ is just an FFT operation, but with ω replaced by ω^{-1}

Lemma 2.4. The columns of matrix M are orthogonal to each other.

Proof. Take the inner product of any columns j and k of matrix M.

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \dots + \omega^{(n-1)(j-k)}$$

This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio $\omega^{(j-k)}$, which evaluates to 0, except for when $j = k$, where the terms are all 1 and the sum is n. Since the result is 0, the vectors are orthogonal to each other. ■

This means that $MM* = nI$, since $(MM*)_{ij}$ is the inner product of the ith and jth columns of M. This implies $M^{-1} = (1/n)M*$.

2.6.4 A closer look at the fast Fourier transform

The following describes

1. Takes an input vector $a = \{a_0, \dots, a_{n-1}\}$ and complex number ω whose powers $1, \omega, \omega^2, \dots, \omega^{n-1}$ are the complex nth roots of unity
2. Multiplies vector a by the matrix $M_n(\omega)$

We can split matrices using divide and conquer. We can divide the matrix in half both ways, and split the input vector in evens and odds. Therefore, the Product of $M_n(\omega)$ with vector $(a_0, \dots, a_{n-1})$, is the produce of $M_{n/2}(\omega^2)$ with $(a_0, a_2, \dots, a_{n-2})$ and with $(a_1, a_3, \dots, a_{n-1})$. This divide and conquer strategy gives us a runtime of $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

3 Decompositions of Graphs

3.1 Why graphs?

A graph is specified by a set of vertices or nodes V and edges E between pairs of vertices. An edge between x and y denoted by e = $\{x, y\}$ means that x shares a border with y, and the graph is undirected. This relationship is symmetric, so y also shares a border with x. An edge between x and y denoted by e = (x, y) means that there is a directed edge going from x to y, but not necessarily from y to x.

3.1.1 How is a graph represented?

We can use an **adjacency matrix** to represent a graph.

Definition 3.1 (Adjacency Matrix)**.** If there are $n = |V|$ vertices $v_1, \ldots, v_n$, the adjacency matrix is an $n \times n$ array whose (i, j)th entry is

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

For undirected graphs, the matrix is symmetric since both (u, v) and (v, u) will be equal. Using this formula, the presence of an edge can be checked in constant time with one memory access. However, the matrix takes up $O(n^2)$ space, which is wasteful if the graph does not have many edges.
An alternative representation is the **adjacency list**, which consists of $|V|$ linked lists, one per vertex. The linked list of vertex u holds the names of vertices that are connected to u. Therefore the total size of the datastructure is $O(|E|)$, and checking if an edge exists is no longer constant time since it requires checking the adjacency list connected to u. For undirected graphs, if v is in u's adjacency list, u is in v's adjacency list.

To determine which representation, adjacency matrix or adjacency list is better, we have to look at $|E|$ in relation to $|V|$. $|E|$ can be as small as $|V|$ or as large as $|V|^2$. If $|E|$ is large, we call this graph *dense*, and it is better to use an adjacency matrix. If $|E|$ is smaller, we call this graph *sparse*, and it is better to use an adjacency list.

3.2 Depth-first search in undirected graphs

3.2.1 Exploring mazes

Depth-first search is a linear-time procedure that determines which parts of the graph are reachable from a given vertex. We maintain a boolean variable for each vertex indicating whether or not it has been visited already. To return to previous junctions, we can use the stack by pushing a new vertex or poping to return to the previous one. Instead of using a stack explicitly, we can use recursion. The following pseudocode shows a simple depth-first search implementation.

```
procedure explore(G, v)
visited(v) = true
previsit(v)
for each edge (v, u) in E
  if not visited(u): explore(u)
postvisit(v)
```

A general form of reasoning to prove graph algorithms is as follows:

- Hypothesis: For any $k \geq 0$, all nodes within k hops from v get visited

- Induction: Showing if all nodes k hops away are visited, then so are all nodes $k+1$ hops away

The edges that the depth-first search algorithm takes are the minimum amount of edges required to visit all the vertices connected to v, so they are called *tree edges*. The edges that are not visited by the algorithm are called *back edges*.

3.2.2 Depth-first search

The previous depth-first search code only visits the portion of the graph reachable from the starting point. To check the rest of the graph, we must call the function on unvisited vertices until all vertices have been visited.
To check the runtime of the algorithm, we notice that each vertex is visited only once. During this visit, we are doing some fixed amount of work (marking the visited spot, pre/postvisit) and looping through the adjacent edges. The total work done in the first step for all vertices is $O(|V|)$. The loop, in total, will check over each edge, $\{x, y\} \in E$ twice, one during the call on x and one during the call on y, making the overall time $O(|E|)$. The total runtime of the depth-first search algorithm is therefore $O(|V| + |E|)$, which is linear in the size of its input.

3.2.3 Connectivity in undirected graphs

An undirected graph is *connected* if there is a path between any pair of vertices. Separate regions of vertices are each called a *connected component*, since each are subgraphs that are connected. We can adapt the depth-first search algorithm to check the connected components of a graph by assigning each node an integer identifying the connected component it belongs to.

3.2.4 Previsit and postvisit orderings

We can take a look at the previsit and postvisit orderings by keeping track of a clock counter, initially set to 1, and defining previsit and postvisit as the following:

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

Property. For any nodes u and v, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained inside the other. We can see that the interval represents the time when a vertex is on a stack, and the last-in, first-out behavior proves the previous property.

3.3 Depth-first search in directed graphs

3.3.1 Types of Edges

We can run the same algorithm on directed graphs without any modifications.
Ancestor and descendant relationships can be read from pre and post numbers. For edge (u, v)

- *Tree edges* are part of the DFS forest. pre(u) < pre(v) < post(v) < post(u)
- *Forward edges* lead from a node to a *nonchild* descendant in the DFS tree. pre(u) < pre(v) < post(v) < post(u)

- *Back edges* lead to an ancestor in the DFS tree. pre(v) < pre(u) < post(u) < post(v)
- *Cross edges* lead to neither descendant nor ancestor and lead to a node that has already been completely explored. pre(v) < post(v) < pre(u) < post(u)

3.3.2 Directly acyclic graphs

A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to \cdots \to v_k \to v_0$. A graph without cycles is *acyclic*.

Property. A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Proof. One direction: If (u, v) is a back edge, there is a cycle consisting of this edge together with the path from v to u in the search tree.
Conversely: If the graph has a cycle $v_0 \to v_1 \to \cdots \to v_k \to v_0$, look at the first node discovered in the cycle, or the one with the one with the lowest pre number, v_i. All other v_j of the cycle is reachable from it, and therefore will be descendants on the search tree. Then the edge $v_{i-1} \to v_i$ will have to lead from a node to its ancestor and is by definition a back edge. ■

Directed Acyclic Graphs are used to model relations like causalities, hierarchies, and temporal dependencies. An example is many tasks with some having dependencies. We can model each task as a node, and there is an edge from u to v if u is a precondition for v. If the graph has a cycle, there is no ordering that can work. If the graph is a dag, we want to try and *linearize* or *topologically sort* the graph, or order the vertices such that each edge goes from an earlier vertex to a later vertex and all precedence constraints are satisfies.
We can linearize all dags by ordering them in decreasing order of their post numbers.

Property. In a dag, every edge leads to a vertex with a lower post number. Otherwise, there would be a back edge that would make the graph not be a dag. If a dag is linearized, the vertex with the smallest post number must be a *sink* (no outgoing edges), and the vertex with the highest post number must be a *source* (no incoming edges).

Property. Every dag has at least one source and at least one sink. This allows us to come up with another approach to linearization:

1. Find a source, output it, and delete it from the graph
2. Repeat until the graph is empty

3.4 Strongly connected components

3.4.1 Defining connectivity for directed graphs

Definition 3.2. Two nodes u and v of a directed graph are *connected* if there is a path from u to v and a path from v to u.

We can then partition V into disjoint sets that are called *strongly connected components*. If we shrink each set into one meta-node, and draw an edge from one meta-node to another if there is an edge between their components, the resulting *meta-graph* must be a dag.

Property. Every directed graph is a dag of its strongly connected components.

The connectivity of a directed graph has two tiers, the top being a dag, while the bottom being the strongly connected component within the dag's meta-nodes.

3.4.2 An efficient algorithm

We can depose a directed graph into its strongly connected components in linear time using depth-first search.

Property. If the explore subroutine is started at node u, then it will terminate when all nodes reachable from u have been visited. Therefore, if we call explore on a node that is in the sink meta-node of the meta-graph, we will retrieve exactly that component.

Property. The node that receives the highest post number in a depth-first search must lie in a source strongly connected component.

Property. If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', the highest post number in C is bigger than the highest post number in C'.

Proof. We have to consider two cases of the depth-first search, based on which component it visits first. If it visits component C before C', all of C and C' will be traversed, and the first node in C that is visited will have a higher post number than any node of C'. If C' gets visited first, then the depth-first search will be stuck after seeing C' but before seeing any of C. ■

We can then find the sink meta-node by reversing graph G to create G^R, then finding the node with the highest post number in G^R. It's strongly connected component would then be the sink strongly component in G. We can then keep deleting the sink in G^R to successively output the strongly connected components.
The final algorithm (which runs in linear time) looks like this:

1. Run depth-first search on G^R
2. Run the undirected connected components algorithm on G, and during the depth-first search, process the vertices in decreasing order of their post numbers from step 1

4 Paths in Graphs

4.1 Distances

The *distance* between two nodes is the length of the shortest path between them. We can imagine the distance from a node by imagining the paths as strings and nodes as balls connected to the string. The distance between nodes is how far they hang down when you lift them up.

4.2 Breadth-first Search

We can see that lifting the nodes partitions the graphs into layers based on distance from the starting node.

Definition 4.1 (Breadth-First Search). Breadth first search traverses the nodes by layer, since it is easy to get the nodes at distance $d + 1$ if we have all the nodes at distance d.

1. Initially, the queue Q consists of only s, the one node at distance 0
2. For each subsequent distance $d = 1, 2, 3, \ldots$, there is a point in where Q contains all the nodes at distance d and nothing else.
3. As these nodes are processed and ejected, their unvisited neighbors at position $d + 1$ are injected to the end of the queue.

```
for all u in V:
  dist[u] = inf

dist(s) = 0
Q = [s]
while Q is not empty:
  u = eject(Q)
  for all edges (u, v) in E:
    if dist(v) = inf:
      inject(Q, v)
      dist(v) = dist(u) + 1
```

The overall runtime is $O(|V| + |E|)$, since each vertex is pushed and popped from the queue once, making $2|V|$ queue operations. For each vertex, the loop looks at each edge once in direct graphs or twice in undirected graphs, taking $O(|E|)$ time.

4.3 Lengths on Edges

Breadth-first search treats all edges as having the same length. In general, however, each edge $e \in E$ has a length l_e. If $e = (u, v)$, we will also write $l(u, v)$ or l_{uv}.

4.4 Dijkstra's Algorithm

4.4.1 An Adaptation to BFS

One solution to weighted graphs is to break a graph G's edges into unit-length pieces through "dummy" nodes. This is done by replacing an edge $e = (u, v)$ of E by adding $l_e - 1$ dummy nodes between u and v, producing a graph G'. We

can then run BFS on G'. One issue with this approach is if G has very long edges.
We can solve this problem with alarms, with estimated times of arrival. When we actually reach a node or an alarm clock rings, we can update the remaining alarm clocks. We can implement this system by using a priority queue, which allows us to tell which alarm will go off next.

```
for all u in V:
  dist(u) = inf
  prev(u) = nil
dist(s) = 0

H = makequeue(V)
while H is not empty:
  u = deletemin(H)
  for all edges (u, v) in E:
    if dist(v) > dist(u) + l(u, v):
      dist(v) = dist(u) = l(u, v)
      prev(v) = u
      decreasekey(H, v)
```

4.4.2 An Alternative Derivation

We can compute the shortest paths in using a different method. We start from the point s, then consider the closest nodes and moving on to those further away. When the "known region" is some subsets of vertices R that includes s, the next addition should be the node outside R closest to s. To find the node v closest to s and outside R, we first consider u, the closest vertex to v in R. u must be closer to s than v is, assuming that all edge lengths are positive. So the shortest path from s to v is a known shortest path extended by a single edge. We can thus find the path to v by trying all the single-edge extensions from the currently known shortest paths, finding the shortest such path, then adding it and the endpoint to R.

4.4.3 Running Time

Dijkstra's algorithm is structurally identical to breadth-first search, but the priority queue operations take logarithmic time. There are at most $|V|$ insert operations, $|V|$ deletemin and $|V| + |E|$ insert/decreasekey oeprations. This means that total time for Dijkstra's algorithm varies based on the priority queue operation used.

Implementation	deletemin	insert/deletekey	overall
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary Heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
d-ary heap	$O(\frac{d \log \|V\|}{\log d})$	$(\frac{\log \|V\|}{\log d})$	$O((\|V\| \cdot d + \|E\|) \frac{\log \|V\|}{\log d})$
Fibonacci Heap	$O(\|V\|)$	$O(1)$	$O(\|V\| \log \|V\| + \|E\|)$

To determine which implementation is best, we need to consider if a graph is *sparse* or *dense*. For all graphs, $|E|$ is less than $|V|^2$. If it is $\Omega(|V|^2)$, the array implementation is faster. A binary heap becomes preferable if $|E|$ dips below $|V|^2 / \log |V|$.
A d-ary heap is a generalization of a binary heap (which has $d = 2$). The optimal choice is $d \approx |E|/|V|$, or the average degree of the graph. For sparse graphs ($|E| = O(|V|)$), the runtime is $O(|V| \log |V|)$. For dense graphs ($|E| = \Omega(|V|^2)$), the runtime is $O(|V|^2)$. For intermediate density graphs ($|E| = |V|^{1+\delta}$), the runtime is $O(|E|)$.
The Fibonacci heap is not touched upon much, except that the insert operation takes varying amounts of time that averages at $O(1)$, so the *amortized* cost of heap insert is $O(1)$.

4.5 Priority queue implementations

4.5.1 Array

We can implement a priority queue as an unordered array of key values for all elements, with the initial values set to ∞. To insert or decreasekey, we only need to change a key value, taking $O(1)$ time. To deletemin, we have to perform a scan of the list, taking $O(n)$ time.

4.5.2 Binary Heap

In the Binary Heap, elements are stored in a complete binary tree. We enforce the constraint that the key value of each node is less than or equal to its children. Therefore, the root contains the smallest element.
To insert, we place a new element at the bottom and "bubble up" by comparing to its parent and swapping the two if it is smaller. The number of swaps is at most the height of the tree, which is $\log_2 n$ with n elements. A decreasekey does the same, except it bubbles up an element already inside the tree from its current position. To deletemin, we return the root value and replace it with the last node in the tree, also the rightmost node in the bottom row of the tree.
We can represent the binary tree using an array. The node at position j in the array will have its parent in position $\lfloor \frac{j}{2} \rfloor$ and children in position $2j$ and $2j + 1$.

4.5.3 d-ary heap

A d-ary heap is similar to a binary heap, except each node has d children instead of 2. The height of the tree is therefore $\Theta(\log_d n) = \Theta((\log n))/(\log d)$. Inserts are therefore sped up to $\Theta(\log d)$. However, deletemin operations are now $O(d \log_d n)$.
To represent a d-ary heap as an array, the node at position j on the array has parent $\lceil \frac{j-1}{d} \rceil$ and children $\{(j-1)d + 2, \ldots, \min\{n, (j-1)d + d + 1\}\}$.

4.6 Shortest paths in the presence of negative edges

4.6.1 Negative edges

Dijkstra's algorithm assumes that the shortest path from the starting point s to any node v must pass through nodes that are closer than v. This assumption no longer holds when the edge weights are negative, since a negative edge weight from a node farther than v to s could be the shortest.
To account for this, we can update all the edges $|V| - 1$ times, by checking the connected edges and getting the min distance. This is called the Bellman-Ford algorithm, which takes a runtime of $O(|V| \cdot |E|)$.

4.6.2 Negative cycles

If there is a negative cycle, it would not make sense to ask the shortest path. Going around the cycle causes the shortest path to continue decreasing. We can detect negative cycles by performing the Bellman-Ford algorithm one more time. If some dist value changes during the last cycle, we know there must be a negative edge.

4.7 Shortest Paths in DAGs

We now take a look at DAGs, which are graphs without cycles, and therefore We observe that in any path of a dag, the vertices appear in increasing linearized order. We can therefore topologically sort the dag by depth-first search, then visit the vertices in the sorted order and update the edges of each.

5 Greedy Algorithms

Greedy algorithms build up a solution by always choosing the next piece that offers the most obvious and immediate benefit. This will only work on certain tasks, where thinking ahead is not needed.

5.1 Minimum Spanning Trees

Suppose you have to network a collection of computers by linking pairs of them. Each edge on the graph represents a possible connection, and the edge weight represents the maintainance cost.

Property. Removing a cycle edge cannot disconnect a graph.

This means that the solution should be connected and acyclic (or a tree), since if the graph was cyclic removing an edge would reduce the cost. These types of trees are called minimum spanning trees.
Input: An undirected graph $G = (V, E)$ with edge weights w_e. *Output:* A tree $T = (V, E')$ with $E' \subseteq E$ that minimizes

$$\text{weight}(T) = \sum_{e \in E'} w_e$$

.

5.1.1 A Greedy Approach

Kruskal's minimum spanning tree starts with an empty graph, then selects edges from E. It repeatedly adds the next lightest edge that doesn't produce a cycle.

Property. A tree on n nodes has $n - 1$ edges.
This can be seen by building the tree one edge at a time. The first node doesn't need an edge, but every other node requires an edge to connect to the main tree.

Property. Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.
We simply need to prove that G is acyclic, then it will satisfy the definition of a tree. We can run the following iterative procedure: while the graph contains a cycle, remove one edge from the cycle. The process terminates when some graph $G' = (V, E')$, $E' \subseteq E$, which is acyclic and connected, so G' is a tree. However, we can see $|E'| = |V| - 1$ by the previous property. This means that $E' = E$, $G' = G$, no edges were removed, and G was acyclic to start with.

Property. An undirected graph is a tree if and only if there is a unique path between any pair of nodes.
First we solve the forward direction. In a tree, any two nodes can only have one path between them. If there were two paths, the union of these paths would contain a cycle. For the backwards direction, we note that if a graph has a path between any two nodes, then it is connected. If these paths are unique, the graph is also acyclic, since a cycle would require two paths between a pair of nodes.

5.1.2 The cut property

Property (Cut Property). Suppose edges X are part of a minimum spanning tree of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$. Then $X \cup \{e\}$ is part of some MST.

A cut is any partition of vertices into two groups, S and $V - S$. The property states that it is safe to add the lightest edge across any cut, provided X has no edges across the cut. To see why the property holds, we can assume that edges X are a part of some MST T. Let's assume that e is not in T. We can construct a different MST T' containing $X \cup \{e\}$ by altering T and changing one of its edges. We do this by adding edge e to T. Since T is connected, it already has a path between the endpoints of e. Therefore, it must have another edge e' across the cut $(S, V - S)$. If we remove this edge, we are left with $T' = T \cup \{e\} - \{e'\}$. Since T' is connected, has the same number of vertices and edges as T, then T' is a MST. If we compare the two tree weights, we get that

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e')$$

Since both e and e' are edges across S and $V - S$, and we assumed e is the lightest edge, $w(e) \leq w(e')$, therefore weight$(T') \leq$ weight(T). Since T is an MST, $w(e) = w(e')$.

5.1.3 Kruskal's Algorithm

To justify Kruskal's algorithm, we notice that at any moment the edges already chosen form a partial solution, or a group of connected components part of the final solution. The next edge to be added, e, should connect two components, T_1 and T_2. Since e is the lightest edge that doesn't produce a cycle, it satisfies the cut property.
In order to test if a candidate edge $u - v$ should be added, we also need to check if u and v lie in different components. We can model the algorithm's state using a collection of disjoint sets, each of which contains the nodes of a component. Initially, each node is a component by itself. We repeatedly test pairs of nodes to see if they belong to the same set. Whenever we add an edge, we merge the two components of the nodes.
The final algorithm uses $|V|$ makeset, $2|E|$ find, and $|V| - 1$ union operations.

5.1.4 A Data Structure for Disjoint Sets

Union by rank We can store sets as a directed tree, where nodes of the tree are elements of the set, and each have parents that eventually lead to the root of the tree. The root element is a representative for the entire set, with its parent pointer being a self-loop. Each node also has a rank that can be interpreted as the height of the subtree hanging from the node. When using the union operation, we should make sure to keep the trees shallow by making the root of the shorter tree point to the root of the taller tree. This will ensure that the height of the tree will only increase if the two trees are equally tall. Instead of using height, we can use the rank of the tree, which is why the algorithm is called *union by rank*.

Property. For any x, rank$(x) <$ rank$(\pi(x))$

Property. Any root node of rank k has at least 2^k nodes in its tree.

Property. If there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k. This means that the maximum rank is $\log n$.

Path Compression The total time for Kruskal's algorithm becomes $O(|E| \log |V|)$ for sorting the edges, with another $O(|E| \log |V|)$ for the union and find operations. We can further improve the performance of the algorithm by using path compression. While following the pointer to the root of the tree, we can change all the pointers to point directly to the tree. This turns the average time per operation and the amortized cost to be $O(1)$ from $O(\log n)$.

A randomized algorithm for minimum cut Let's define a cut $(S, \overline{S})$ in the graph. Suppose the graph we are working with is unweighted, and the edges are ordered uniformly at random. If we define the size of a cut $(S, \overline{S})$ as the number of edges crossing between S and $\overline{S}$, there is at least a $\frac{1}{n^2}$ probability that the random cut is the minimum cut in the graph. This means that repeating the process $O(n^2)$ times and outputting the smallest cut gives a high probability of finding the smallest cut. We then have an $O(mn^2 \log n)$ algorithm for unweighted minimum cuts, which can be reduced to $O(n^2 \log n)$.
To see why the probability for a minimum cut at each iteration is at least $\frac{1}{n^2}$, we can check Kruskal's algorithm. At any stage of the algorithm, the vertex set V is partitioned into connected components. The only edges we can add to the tree have endpoints in separate components. Let's call the size of the minimum cut C. The number of edges incident to each component is at least C. So, if there are k components in the graph, there will be at least $\frac{kC}{2}$ number of eligible edges that can be cut. The chance that the next eligible edge is from the minimum cut is at most $\frac{C}{(\frac{kC}{2})} = \frac{2}{k}$. Thus, the probability of leaving the minimum cut intact is at least $\frac{k-2}{k}$, and the chance that Kruskal's algorithm leaves the minimum cut intact until the last edge is at least

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} = \frac{1}{n(n-1)}$$

5.1.5 Prim's Algorithm

In Prim's algorithm, an intermediate set of edges X forms a subtree, where S is the set of this tree's vertices. On each iteration, X grows by one edge, which is the lightest edge between a vertex in S and a vertex outside S. This is similar to Dijkstra's algorithm, except that the value of a node is weight of the lightest incoming edge from set S, while in Dijkstra's the weight is the path from the start node.

5.2 Huffman Encoding

In an MP3 audio compression, the sound is encoded in three steps:

1. It is digitized by sampling at regular intervals, translating to a sequence of real numbers $s_1, s_2, \dots s_T$.
2. Each sample s_t is quantized, or approximated by a nearby number from a finite set Γ, which is carefully chosen to be indistinguishable to the human ear.
3. The resulting string of length T over alphabet Γ is encoded in binary.

Let's focus on the last step. For a simple example, assume that T is 130 million and Γ only has four values, A, B, C, D. We might choose to encode with 2 bits per symbol, with 00 for A, 01 for B, 10 for C, and 11 for D, requiring 260 megabits. However, let's say that the four symbols are not equally abundant.

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

We may then consider a *variable-length* encoding, where just one bit is used for frequent characters at the expense of three or more for other characters. In order to ensure uniqueness, we need to insist on a *prefix-free* property, where no codeword can be a prefix of another codeword. We can represent any prefix-free encoding with a full binary tree, where every node has either zero or two children, with the leaves as symbols and codeword generated by a path from a root to a leaf.
We want to find the optimal coding tree, given the frequencies, $f_1, f_2, \dots f_n$ of n symbols. Then we want to minimize the

overall length of the encoding,

$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth of } i\text{th symbol in tree})$$

Since we are representing as a tree, we can define each internal node to be the sum of the frequencies of its descendant leaves, or how many times the internal node is visited during encoding or decoding. During encoding, every time we visit a nonroot internal node, one bit gets output. Thus, another useful way of writing the cost function is the sum of the frequencies of all leaves and internal nodes, except the root.
The two symbols with the lowest frequencies should be at the bottom of an optimal tree, otherwise swapping them with symbols with a lower frequency would improve the encoding. This suggests us to use a greedy algorithm: find the two symbols with the smallest frequencies, i and j, make them the children of a new node which then has frequency $f_i + f_j$. Then, the cost of the tree is cost $f_1 + f_2$ plus the cost of a tree with $n-1$ leaves of frequencies $(f_1 + f_2), f_3, f_4, \ldots, f_n$. The resulting algorithm can make use of a priority queue, and takes $O(n \log n)$ time if a binary heap.

Entropy Let's say we have a horse race with four horses. We record the probability of them getting first, second, third, or other in their previous races. By determining the total number of bits required to encode the horse track records, we can determine how compressible and therefore predictable the horses are.
In general, suppose there are n possible outcomes with probabilities $p_1, p_2, \ldots p_n$. If a sequence of m values is drawn from the distribution, the ith outcome will pop up roughly mp_i times. For simplicity, assume that the p_i's are all powers of 2, or in the form $\frac{1}{2^k}$.
By induction, the number of bits needed to encode the sequence is

$$\sum_{i=1}^{n} mp_i \log(\frac{1}{p_i})$$

The average number of bits needed to encode a single draw from the distribution is

$$\sum_{i=1}^{n} p_i \log(\frac{1}{p_i})$$

This is the *entropy* of the distribution, or a measure of how much randomness it contains.

5.3 Horn Formulas

Horn formulas are a framework for computers to express logical facts and derive conclusions. The most primitive object is a *boolean*, which is either true or false. A *literal* is a variable x or its negation $\overline{x}$. There are two types of *clauses*:

1. *Implications*: A left-hand side of an AND of any number of positive literals and right-hand side is a single positive literal.
$$(z \wedge w) \implies u$$
2. *Negative clauses*: An or of any number of negative literals.
$$(\overline{u} \vee \overline{v} \vee \overline{y})$$

Given a set of clauses, we want to determine if there is a set of some variables that satisfies all the clauses, or a *satisfying assignment*.
One strategy is to start with all the variables being false, then set some of them to true only if we absolutely have to or else an *implication* would be violated. Once we are done and all the *implications* are satisfied, we then turn to the *negative clauses* and make sure they are satisfied as well. This follows a greedy scheme:

```
Input: A Horn formula
Output: A satisfying assignment, if one exists
set all variables to false
while there is an implication that is not satified:
  set the right-hand variable of the implication to true

if all pure negative clauses are satisfied: return the assignment
else: return ‘‘formula is not satisfiable‘‘
```

We have to convince ourselves that if the algorithm finds no satisfying assignment, there really is none. This is satisfied by the fact we are being "stingy," which maintains the invariant that:

If a certain set of variables is set to true, then they must be true in any satisfying assignment.

Therefore, if the truth assignment after the while loop does not satisfy the negative clauses, there can be no satisfying truth assignment.

5.4 Set Cover

Set cover takes in a set of elements B and sets $S_1, \ldots, S_m \subseteq B$. Then, it outputs the selection of S_i whose union is B, and the number of sets picked, or the cost. This problem seems to have a greedy solution, where we pick the set S_i with the largest number of uncovered elements each time. However, this greedy scheme is not optimal.

Claim Suppose B contains n elements that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.
Let n_t be the number of elements not covered after t iterations of the greedy algorithm. Since these remaining elements are covered by the optimal k sets, there must be some set with at least n_t/k of them. Therefore, the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t(1 - \frac{1}{k})$$

By repeated application, this implies $n_t \leq n_0(1 - \frac{1}{k})^t$. We can use the following inequality:

$$1 - x \leq e^{-x} \text{for all } x \text{, with equality if and only if } x = 0$$

Thus:

$$n_t \leq n_0(1 - \frac{1}{k})^t < n_0(e^{-\frac{1}{k}})^t = ne^{-\frac{t}{k}}$$

At $t = k \ln n$, n_t is less than $ne^{-\ln n} = 1$, meaning no elements remain to be covered.
The ratio between the greedy solution and optimal solution is always less than $\ln n$, which is called the *approximation factor*.

6 Dynamic Programming

6.1 Shortest Paths in DAGs, Revisited

We can use dynamic programming to study the shortest paths in directed acyclic graphs (dags). Dags can be linearized, meaning that their nodes can be lined up so that all edges go from left to right. To find the shortest path to any node, we simply have to compare the distances of the paths of the incoming edges. We can therefore use the following algorithm to compare the distances in a single pass:

```
initialize all dist() values to infinity
dist(s) = 0
for each v in V\{s}, in linearized order:
  dist(v) = min_{(u, v) in E} {dist(u) + l(u, v)}
```

Notice the algorithm is solving a collection of subproblems, namely finding $\{\texttt{dist}(u) : u \in V\}$. We can start with the obvious $\texttt{dist}(s) = 0$. Then, we can proceed to progressively large "subproblems," by looking at the distances that are further and further along.
This is where dynamic programming comes in. Dynamic programming solves a problem by first identifying a collection of subproblems and tackling them one by one starting from the smallest, then using the answers to figure out larger ones. The dag is *implicit* in all dynamic programming problems. Its nodes are the subproblems we define, and the edges are dependencies.

6.2 Longest Increasing Subsequence

The longest increasing subsequence problem involves taking a sequence of numbers, $a_1, \dots, a_n$, and outputting a subsequence which is strictly increasing and the longest possible. We can represent all increasing subsequences as a dag, where each node represents a number, and an edge is drawn from a node to another if the destination is larger. This translates the problem to finding the longest path in the dag.

```
for j = 1, 2,... n:
  L(j) = 1 + max{L(i) : (i, j) in E}
return max_j L(j)
```

L(j) is the length of the longest path ending at j. Since the path to node j must pass through one of its predecessors, L(j) is 1 plus the maximum $L(\cdot)$ value of the values before it. The final answer is the largest L(j), since any ending position is valid.
This is dynamic programming since we have defined appropriate subproblems, namely finding L(j). The runtime of the algorithm requires constructing an adjacency list of the graph and checking the indegree of j. Constructing the list takes linear time, and checking the indegree takes time linear in $|E|$. This is at most $O(n^2)$, the maximum being if the array is sorted in increasing order.
To get the actual path of the shortest path, we need to store prev(j), which is a pointer to the previous node of the longest path. By following the pointers, we can reconstruct the shortest path.

Recursion? No, thanks. Recursion by itself is a bad idea because it will take exponential time. The same subproblems will get solved over and over again, creating a tree with exponentially many nodes. Recursion only works well in divide and conquer, where problems are divided into *substantially* smaller subproblems, so the tree only has logarithmic depth. A typical dynamic programming problem, in contrast, has only slightly smaller subproblems, but form a tree where many of the nodes are repeated.

Programming? The programming in dynamic programming initially meant "planning," so dynamic programming means to optimally plan multistage processes. The dag can be thought of all possible ways for a process to evolve, with each node being a state and the outgoing edges representing actions to new states.

6.3 Edit Distance

If we have two different strings, we can align them by using gaps. Then we can determine the cost of the string, or how many columns where the letters differ. Edit distance can also be thought as the minimum number of edits (insertions, deletions, and substitutions) required to transform the first string into the second.
To apply dynamic programming, we first need to determine the subproblems. We can look at the edit distance between some prefix of the first string and the second string. We can call this subproblem $E(i, j)$, where $x[1 \dots i]$ and $y[1 \dots j]$ are valid substrings.
For the rightmost column, we can either insert a gap at $y[j]$, $x[i]$, or keep both of them. Therefore, we can divide $E(i, j)$ into three smaller subproblems, $E(i-1, j)$, $E(i, j-1)$, or $E(i-1, j-1)$. We can then define $E(i, j)$ as

$$E(i, j) = \texttt{min}\{1 + E(i-1, j), 1 + E(i, j-1), \text{diff}(i, j) + E(i-1, j-1)\}$$

where diff(i, j) is 0 if $x[i] = y[j]$ and 1 otherwise.
The answers to the subproblems of $E(i, j)$ is a two-dimensional table. We need to ensure that $E(i, j-1)$, $E(i-1, j)$, and $E(i-1, j-1)$ are handled before $E(i, j)$. Therefore, both filling the rows from left to right or columns from top to bottom work.
We also have to consider the base cases of the dynamic programming, or the smallest subproblems. These cases are $E(0, j) = j$ and $E(i, 0) = i$. This is because we have to add the first j or i letters to match the prefix from an empty substring.

```
for i = 0, 1, 2,..., m:
  E(i, 0) = i
for j = 0, 1, 2,..., n:
  E(0, j) = j
for i = 1, 2,..., m:
  for j = 1, 2,..., n:
    E(i, j) = min{E(i-1, j) + 1, E(i, j-1) + 1, E(i-1, j-1) + diff(i, j)}
return E(m, n)
```

The underlying dag has the form $(i-1, j) \rightarrow (i, j)$, $(i, j-1) \rightarrow (i, j)$, $(i-1, j-1) \rightarrow (i, j)$. We can also add weights on the edges, so the edit distances are given by the shortest paths in the dag. All the weights are set to 1, except where $\{(i-1, j-1) \rightarrow (i, j) : x[i] = y[i]\}$, which has a length of 0. Each move downw is a deletion, each move right is an insertion, and each diagonal move is either a match or a substitution.

Common Subproblems There are a few common subproblems that arise repeatedly.

1. The input is $x_1, x_2, \dots x_n$, and a subproblem is $x_1, x_2, \dots x_i$. The number of subproblems is linear.
2. The input is $x_1, \dots x_n$, and $y_1, \dots y_n$, and a subproblem is $x_1, \dots x_i$ and $y_1, \dots y_j$. The number of subproblems is $O(mn)$.
3. The input is $x_1, \dots x_n$ and a subproblem is $x_i, x_{i+1}, \dots x_j$. The number of subproblems is $O(n^2)$.
4. The input is a rooted tree, and a subproblem is a rooted subtree.

6.4 Knapsack

During a robbery, a burglar needs to decide what to put in his bag, which can hold a total weight of at most W pounds. There are n items, with weights $w_1, \dots w_n$ and value $v_1, \dots v_n$. He wants to find the most valuable combination of items into the bag. There are two variations of the problem, the first that allows repetition, and the second that only has one of each item.

6.4.1 Knapsack with Repetition

We define $K(w) =$ maximum value achievable with a knapsack of capacity w. Then, we get the following relation:

$$K(w) = \texttt{max}_{i:w_i \leq w}\{K(w - w_i) + v_i\}$$

.

```
K(0) = 0
for w = 1 to W:
  K(w) = max{K(w - w_i) + v_i : w_i <= w}
return K(W)
```

The algorithm fills up a one-dimensional table of length $W+1$. Each entry can take up to $O(n)$ to compute, so the overall runtime is $O(nW)$. The algorithm can be represented by a dag, and finding the longest path in it.

6.4.2 Knapsack without Repetition

Our earlier subproblem cannot be used because we don't know whether item n has already been used when looking at $K(w - w_n)$. Therefore, we need to add a second parameter, $0 \leq j \leq n$:

$$K(w, j) = \text{maximum value achievable using a knapsack of capacity } w \text{ and items } 1, \ldots j$$

Then, our answer is $K(W, n)$. We can also express $K(w, j)$ in terms of subproblems by using $K(w, j) = \texttt{max}\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$. The result is filling out a 2D table, with $W + 1$ rows and $n + 1$ columns, taking $O(nW)$ time.

```
Initialize all K(0, j) = 0 and all K(w, 0) = 0
for j = 1 to n:
  for w = 1 to W:
    if w_j > w: K(w, j) = K(w, j-1)
    else: K(w, j) = max{K(w, j - 1), K(w, w_j, j-1) + v_j}
```

Memoization Dynamic programming also suggests a recursive algorithm, since it builds upon subproblems. However, a naive recursive solution is very inefficient, since it solves the same subproblems over and over again. An algorithm can use a hash table to store the values of $K(\cdot)$ that have already been computed, and whenever calculating some $K(w)$, the algorithm first checks if the answer is already in the table. This trick if called memoization.
The constant factor in the big-O notation is larger because of recursion overhead. However, sometimes memoization pays off since dynamic programming solves every subproblem that could conceivably needed, while memoization only ends up solving the ones actually used.

6.5 Chain Matrix Multiplication

If we want to multiply matrices, we can change the grouping of multiplication, since matrix multiplication is associative. This means that $A \times (B \times C) = (A \times B) \times C$.
Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications. Therefore, rearranging the parenthesis in matrix multiplication will change the cost. We want to determine the optimal order of multiplying matrices with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots m_{n-1} \times m_n$.

Notice we can represent the order of multiplication as a binary tree, with the individual matrices in the trees, interior nodes being intermediate products, and the root being the final product. Since the number of possible trees is exponential in n, we cannot try every tree.
In order for the tree to be optimal, its subtrees must also be optimal. For $1 \leq i \leq j \leq n$, let us define the cost

$$C(i,j) = \text{minimum cost of multiplying } A_i \times A_{i+1} \times \cdots \times A_j$$

The smallest subproblem is when $i = j$, where nothing is multiplied and $C(i,i) = 0$. For $j > 1$, we split the product by choosing k where $i \leq k < j$, creating two subtrees $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdot \times A_j$. The cost of the subtree is then $C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j$. We just need to find the splitting point k for which the equation is the smallest.

```
for i = 1 to n: C(i, i) = 0
for s = 1 to n - 1:
  for i = 1 to n - s:
    j = i + s
    C(i, j) = min(C(i, k) + C(k+1, j) + m_{i-1} * m_k * m_j : i <= k < j)
return C(1, n)
```

The subproblems create a two-dimensional table, with each entry taking $O(n)$ time, making the overall runtime $O(n^3)$.

6.6 Shortest Paths

6.6.1 Shortest Reliable Paths

Sometimes we want to find the shortest path within a certain number of edges, since more edges may cause more unreliability. Suppose we want to find the shortest path in a graph G from s to t that uses at most k edges. Let us define, for each vertex v and integer $i \leq k$, $\texttt{dist}(v,i)$ to be the length of the shortest path from s to v that uses i edges. The starting values $\texttt{dist}(v,0)$ are ∞ for all vertices except s, which is 0. Then the equation is

$$\texttt{dist}(v,i) = \texttt{min}_{(u,v)\in E}\{\texttt{dist}(u,i-1) + l(u,v)\}$$

6.6.2 All-pairs Shortest Paths

If we want to find the shortest path between all pairs of vertices, we can run the previous algorithm on all vertices $|V|$ times, making a runtime of $O(|V|^2|E|)$. However, there is a more efficient dynamic programming algorithm known as the *Floyd-Warshall* algorithm, which takes $O(|V|^3)$.
First, number the vertices in V as $\{1, 2, \ldots n\}$, and let $\texttt{dist}(i,j,k)$ denote the length of the shortest path from i to j which only nodes $\{1, 2, \ldots k\}$ can be used as intermediates. Initially, $\texttt{dist}(i,j,0)$ is the length of the direct path between i and j if it exists, and ∞ otherwise. To check with using k as an intermediate node between i and j, we can simply check if $\texttt{dist}(i,k,k-1) + \texttt{dist}(k,j,k-1) < \texttt{dist}(i,j,k-1)$. This is because we have already calculated the optimal path from i to k and from k to j using lower numbered nodes, so we only have to check one more path.

```
for i = 1 to n:
  for j = 1 to n:
    dist(i, j, 0) = inf
for all (i, j) in E:
  dist(i, j, 0) = l(i, j)
for k = 1 to n:
  for i = 1 to n:
    for j = 1 to n:
      dist(i, j, k) = min(dist(i, k, k-1) + dist(k, j, k-1), dist(i, j, k - 1))
```

6.7 The Traveling Salesman Problem

A traveling salesman wants to visit each target city once and return home, with the minimum possible length. We denote the cities as $1, \dots n$, with the salesman's hometown being 1, and let $D = (d_{i,j})$ be a matrix of intercity distances. The goal is to design a tour that starts and ends at 1, includes all other cities, and has minimum total length.
The brute force solution takes $O(n!)$ time. Dynamic programming can yield a faster solution, although not in polynomial time.
The most obvious subproblem is a partial tour, where we have started at city 1, have visited a few cities, and now are in city j. We need to know all the cities visited so far, and j, the current city. The appropriate subproblem, then, is as follows:

For a subset of cities $S \subseteq \{1, 2, \dots n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot both start and end at 1. To express $C(S, j)$ in terms of a smaller subproblem, we need to consider what to pick as the second-to-last city. It must be some $i \in S$, so the overall path length is the distance from 1 to i, namely $C(S - j, i)$, plus the length of the final edge, d_{ij}. We need to pick the best such i:

$$C(S, j) = \texttt{min}_{i \in S : i \neq j} C(S - j, i) + d_{i,j}$$

```
C({1}, 1) = 0
for s = 2 to n:
  for all subsets S in {1, 2,..., n} of size s and containing 1:
    C(S, 1) = inf
    for all j in S, j != 1:
      C(S, j) = min(C(S - {j}, i) + d_{ij} : i in S, i != j)
return min_j C({i, ..., n}, j) + d_{j1}
```

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve. The runtime is then $O(n^2 2^n)$.

6.8 Independent Sets in Trees

A subset of nodes $S \subset V$ is an *independent set* of graph $G = (V, E)$ if there are no edges between the nodes in the set. If the graph is a tree, we can find the largest independent set of nodes in a graph in linear time. Just like the matrix multiplication problem, we can use the subtrees to naturally define the subproblem.
Start by rooting the tree at any node r. Each node defines a subtree which hangs from it. Let $I(u)$ be the size of the largest independent set of subtree hanging from u. Our final goal is $I(r)$.
Our dynamic programming problem should start at the subproblems, or the roots, and go up. We can split the computation into two cases, either including u or not.

$$I(u) = \texttt{max} \left\{ 1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w) \right\}$$

On time and memory The amount of time it takes to run a dynamic programming algorithm is the total number of edges in a dag. We just visit the nodes in linearized order, examine its inedges, and typically perform a custom amount of work per edge. The amount of memory required is not based on any part of the dag, however. We technically only need to store the value of a particular subproblem until the larger subproblems depending on it have been solved. For example, in Floyd-Warshall we only need two $|V| \times |V|$ arrays to store the dist values, one for odd values of k and one for even values.