1. Foundations

Time Complexity

Asymptotic Rate of Growth:

- We use asymptotic rate of growth to compare algorithms since we are more interested in which algorithm scales better (is more efficient as input size increases), and their long-run behaviour
- Small or specific function values are insufficient for comparison since they may be outliers or too small a value to appreciate the efficiencies of an algorithm
- However, when handling input below a known size, we must consider whether asymptotic analysis is better or not
- NOTE: log < polynomial < exponential < factorial

Big-Oh: A function $f(n)$ is in the class $O\big(g(n)\big)$ if $f(n)$ is eventually (for large enough n) bounded above by a constant multiple of $g(n)$

- Formally, $f(n) = O\big(g(n)\big)$ if and only if there exist constants $n_0,\ C > 0$ such that $f(n) \leq C \cdot g(n)$ for all $n \geq n_0$
- Implies that an algorithm running in $f(n)$ time scales at least as well as one running in $g(n)$ time → the $f(n)$ algorithm is at least as fast as the $g(n)$ one
- On a graph, $C \cdot g(n)$ is above $f(n)$ for all $n \geq n_0$

Big-Omega: A function $f(n)$ is in the class $\Omega\big(g(n)\big)$ if $f(n)$ is eventually (for large enough n) bounded below by a constant multiple of $g(n)$

- Formally, $f(n) = \Omega\big(g(n)\big)$ if and only if there exist constants $n_0, c > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$
- Implies that an algorithm running in $f(n)$ time scales at least as badly as one running in $g(n)$ time → the $f(n)$ algorithm is no faster than the $g(n)$ one
- On a graph, $c \cdot g(n)$ is below $f(n)$ for all $n \geq n_0$

Big-Theta: A function $f(n)$ is in the class $\Theta\big(g(n)\big)$ if $f(n)$ is in the class of both $O\big(g(n)\big)$ and $\Omega\big(g(n)\big)$

- Formally, $f(n) = \Theta\big(g(n)\big)$ if and only if there exist constants $n_0, c, C > 0$ such that $c \cdot g(n) \leq f(n) \leq C \cdot g(n)$ for all $n \geq n_0$
- An algorithm running in $f(n)$ time scales as well as one running in $g(n)$ time → the $f(n)$ algorithm is as fast as the $g(n)$ one
- On a graph, $c \cdot g(n)$ is below $f(n)$, and $C \cdot g(n)$ is above $f(n)$ for all $n \geq n_0$

Sum Property: If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = O(g_1 + g_2)$

- This property justifies the ignoring of non-dominant terms → if f_2 has a lower asymptotic bound than f_1, then the bound on f_1 also applies to $f_1 + f_2$
- Useful for analysing algorithms with multiple stages executed sequentially → e.g. a two stage algorithm where stage 1 (f_1) is linear and stage 2 (f_2) is quadratic, we treat the overall algorithm ($f_1 + f_2$) as quadratic since that is the most expensive stage (higher lower bound)
- $O(g_1 + g_2)$ is often written $O(\max(g_1, g_2))$ since $g_1 + g_2 \leq 2\max(g_1, g_2)$
- The sum property also applies to Ω and Θ notation

Product Property: If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 \cdot f_2 = O(g_1 \cdot g_2)$

- Useful for analysing algorithms with multiple nested parts → if each execution of the inner part takes f_2 time, and it is executed f_1 times, we can bound each part and multiply the bounds
- The product property also applies to Ω and Θ notation

Data Structures

Binary Heaps

- Complete binary tree where parents are >= (max heap) or <= (min heap) all children
- Are actually arrays
- Time Complexities:
  - Build heap: O(n)
  - Find Maximum: O(1)
  - Delete Maximum: O(logn)

    - Insert: O(logn)

Binary Search Trees

- Everything left of a node is smaller, and everything right is bigger
- Time Complexities:
    - Search: O(h)
    - Insert/Delete: O(h)

Frequency Table

- An array with all possible elements storing the frequency those elements appear in another array or data structure

Hash Tables

- Stores key-value pairs
- Values are hashed by a hash function to produce a key that is the index the key-value pair will be stored at
- Best Case Time Complexities:
    - Search: O(1)
    - Updating value in key-value pair: O(1)
    - Insert/Delete: O(1)
- Worst Case Time Complexities:
    - Search: O(n)
    - Updating value in key-value pair: O(n)
    - Insert/Delete: O(n)

Proofs

Propositions: true/false statements

- The goal of an argument is usually to establish a relationship between propositions

Notation	Meaning	Formal Term
$\neg P$	not P	Negation
$P \wedge Q$	P and Q	Conjunction (And)
$P \vee Q$	P and\or Q	Disjunction (Or)
$P \rightarrow Q$	If P then Q	Implication

$\forall$	For all	For all
$\exists$	For some	For some

Proof by Induction: A proof that involves proving that the first proposition is true and that each proposition implies the next one to prove a sequence of propositions

- Prove the first proposition P_1 is true, then use that to prove P_2 and use that to prove P_3 and so on, such that it must be true $\forall P$ if P_1 is true, which it is since we proved it

Proof by Contradiction: If the assumption $\neg P$ leads to a contradiction $(Q \wedge \neg Q)$ for some other proposition, then $\neg P$ is impossible, so P must hold true

- Assume a proposition P is false, then produce another proposition for which the assumption that P is false clearly results in a wrong or impossible scenario, thus contradicting the assumption that P is false and requiring it to be true

Algorithmic Reasoning: Whenever we present an algorithm, we must justify its correctness and efficiency

- Need to prove our claims that it always gives the right answer, and that it runs in the time complexity we claim it does

2. Divide and Conquer

Divide and Conquer: divide a large difficult problem into smaller, workable problems that you solve (conquer), and combine the solutions to solve the large problem

- Known examples include binary search, merge sort,

We have two main types of problems

- Decision problems: "Given X, can you do Y?" → yes or no answer, and justify
- Optimisation problems: "What is the smallest X for which you can do Y?" → explicit answer of X, and justify

Merge Sort:

- Divide: Split the array into two equal parts → O(1)
- Conquer: Sort each part recursively → log_2(n) recursions
- Combine: Merge the two sorted subarrays → O(n)

Every time a smaller number is placed from the second subarray during the merging step, there are k inversions, where k is the number of elements left in the first subarray → the smaller number that was placed forms an inversion with all k elements in the first subarray (the k elements are all larger, else they would have already been sorted in the first subarray and placed)

Quicksort:

- Divide: choose a pivot and partition the array around it → O(n)

- Conquer → sort both sides of the partition recursively → log_2(n) recursions
- Combine: Pass the answer up the recursion tree O(1)

Recurrences

Recurrences arise in estimations of time complexity for divide-and-conquer algorithms

Suppose a divide and conquer algorithm reduces a problem of size n into smaller problems of size n/b, then the solution of the recurrence is given by

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

- $a \geq 1$ is the integer factor the number of subproblems grows by on each level, and $b > 1$ is the real number factor the size of the subproblems shrink by at every level
- $f(n)$ is the overhead costs associated with splitting and combining these smaller problems
- The critical exponent is $c = \log_b a$, and the critical polynomial is n^c

Master Theorem: A theorem that provides us with the growth rate of a time complexity solution, and (where relevant) the approximate sizes of the constants involved

1. If $f(n) = O(n^{c-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^c)$
2. If $f(n) = \Theta(n^c \cdot (\log n)^k)$ for some $k \geq 0$, then $T(n) = \Theta(n^c \cdot (\log n)^{k+1})$
3. If $f(n) = \Omega(n^{c+\varepsilon})$, for some $\varepsilon > 0$, AND if, for some $k < 1$ and for all $n > n_0$, $a \cdot f\left(\frac{n}{b}\right) \leq k \cdot f(n)$, then $T(n) = \Theta\big(f(n)\big)$

NOTE: If none of these conditions hold, the Master Theorem is NOT applicable

- ALSO not applicable in cases where we cannot determine b (e.g. size of tree) or if $f(n)$ does not fit any condition

We structure responses using the Master Theorem like so:

1. Let $T(n) = \cdots$

2. Then the critical exponent is $c = \cdots$, so the critical polynomial is $n^c = \cdots$
3. Now, $f(n) = \cdots = [MASTER\ THEOREM\ CONDITION]$
4. This satisfies the condition for $[MASTER\ THEOREM\ CASE]$ with [case conditions; e.g. $k = \cdots < 1$], so $T(n) = \cdots$

Models of Computation

The uniform model of computation is where all arithmetic operations are assumed to take constant time.

In the logarithmic model of computation, potentially huge inputs are thought to have size equal to their width (the number of symbols required to express them

- In base 2, this is the number of bits in the number → an integer N has $n = \log_2 N$ bits
- Hence, integer addition requires the n bits to be added and things to be carried over, so addition now takes linear O(n) time
- Multiplication requires each of the n bits in N1 to be multiplied with each of the n bits in N2, and then for n O(n) additions to occur → $O(n^2)$
    - Least significant bit of N1 multiplied against all bits of N2 and result shifted 0 bits, then second least significant bit of N1 multiplied against all bits of N2 and result shifted 1 bit, and so on until most significant bit of N1 multiplied against all bits of N2 and result shifted n – 1 bits

An $O(n^2)$ divide-and-conquer algorithm to multiply two n-digit numbers:

1. If n = 1, just evaluate the product and return it
2. Otherwise split the input numbers A and B
    - A_0, B_0 are the least significant $\frac{n}{2}$ bits of A and B
    - A_1, B_1 are the most significant $\frac{n}{2}$ bits of A and B
    - $AB = A_1B_1 \cdot 2^n + (A_1B_0 + A_0B_1) \cdot 2^{\frac{n}{2}} + A_0B_0$
3. Now recursively compute:
    a. $X = A_0B_0$
    b. $Y = A_0B_1$
    c. $Z = A_1B_0$
    d. $W = A_1B_1$
4. Compute $W \cdot 2^n + (Y + Z) \cdot 2^{\frac{n}{2}} + X$, and return the result

Karatsuba Trick: An $\Theta\left(n^{\log_2 3}\right)$ divide and conquer algorithm that multiplies two n-digit numbers

- We rearrange the previous expression of AB to get
$$AB = A_1B_1 \cdot 2^n + \left((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0\right) \cdot 2^{\frac{n}{2}} + A_0B_0$$
- We save one round of multiplication every round by using this expression (don't need to calculate A_0B_1 and A_1B_0, but need to compute $(A_1 + A_0)(B_1 + B_0)$) → each problem is split into 3 subproblems, not 4
- We just use this expression instead in step 3:
    a. $X = A_0B_0$
    b. $W = A_1B_1$
    c. $V = (A_1 + A_0)(B_1 + B_0)$
- Then we compute $W \cdot 2^n + (V - W - X) \cdot 2^{\frac{n}{2}} + X$, and return the result

Extensions to the Karatsuba trick → Splitting A and B into more than 2 parts

- We can achieve $T(n) = O(n^{1+\varepsilon})$ by doing this
- However, if we split A and B into p + 1 parts, the constant factors involved are around p^p, so it is only worthwhile for extremely large inputs
- ONLY rely on asymptotic estimates if we know the constants hidden by asymptotic notation are reasonably small → There also exists a 2021 $O(n \log n)$ algorithm that may be faster for astronomically large numbers

Extensions to the Karatsuba trick → Multiplying Polynomials

- Split $P_A(x) = A_{p-1}x^{p-1} + \cdots + A_2x^2 + A_1x + A_0$, and the same for $P_B(x)$
- Objective is to compute the coefficients (not limited to integers) of $P_A(x)P_B(x)$ with as few multiplications as possible

Convolutions

SUPER Shitty convolution understanding:

1. Make 2 sequences A and B
2. Reverse B, which is shorter than or equal to A

3. Multiply all terms $A_i,\ B_j$ such that $i + j = k$ and sum these results to get C_k (k starts at 0)
4. Shift B right by one and repeat to get C_{k+1}

Coefficient representation: Represent a degree n polynomial in an n-sized array where the element at the i^{th} index is the coefficient of x^i

- Addition: O(n) term by term
- Evaluation: O(n) using Horner's rule
- Multiplication: O(n^2) every term in A multiplies with every term in B

Value representation: Represent a degree n polynomial with a sequence of n+1 values

- Fix a selection of inputs $x_0, \dots, x_n$, then the corresponding values $P(x_0), \dots, P(x_n)$ represent a unique polynomial of degree up to n
- Cannot evaluate non-fixed input using value representation, but can multiply polynomials in linear time by multiplying pointwise → $P_A P_B(x_0) = P_A(x_0)P_B(x_0)$
- BUT, in order to uniquely determine the product polynomial, we need to have enough input points → you want to ensure you have 2n + 1 points, since the product of two n degree polynomials is a 2n degree polynomial

Strategy to multiply polynomials fast

1. Evaluate $P_A(x)$ and $P_B(x)$ at 2n+1 distinct points → $x_0, \dots, x_{2n}$
2. Multiply them point by point (2n+1 multiplications of large numbers) to get 2n+1 values of the product polynomial $P_C(x)$
3. Reconstruct the coefficients of $P_C(x) = C_{2n}x^{2n} + C_{2n-1}x^{2n-1} + \cdots + C_1 x + C_0$

- But if we can't evaluate and reconstruct quickly, this is still O(n^2) time
- To hasten the process while maintaining computational accuracy, we want to choose inputs x_i that do not grow or shrink massively with large powers → 1 does not, so we want to take all solutions to $|x^n| = 1$, which must include complex numbers if we want 2n+1 points

NOTE: We assume multiplying complex numbers is constant time

Complex number: $z = a + bi$

- Modulus $|z| = r = \sqrt{a^2 + b^2}$
- Argument $\arg(z) = \theta \in (-\pi, \pi]$, such that $a = rcos(\theta)$, $b = rsin(\theta)$
- And so $z = a + bi = r(\cos(\theta) + \sin(\theta)\, i) = re^{i\theta}$ → convert to polar form
- To multiply two complex numbers, we multiply the moduli and add the arguments:
  - If $z = re^{i\theta}$, and $w = se^{i\phi}$, then $zw = (rs)e^{i(\theta+\phi)}$
  - If $z = re^{i\theta}$, then $z^n = r^n e^{i(n\theta)}$
- This makes computing powers of complex numbers really easy if the modulus is 1, since $1^n = 1$

We are interested in the roots of unity of order m for our inputs

- Roots of unity of order m: $z^m = r^m e^{i(m\theta)} = 1$
  - This requires $r = 1$, and $m\theta = 2\pi k$ → $\theta = \frac{2\pi k}{m}$
- Let $\omega_m = e^{\frac{2\pi i}{m}}$. Then $z = \omega_m^k = (\omega_m)^k$ → all roots of unity of order m can be written as powers of ω_m (the first root of unity of order m) → there are only m distinct values of ω_m, as $\omega_m^m = \omega_m^0$
- The product of two roots of unity of order m is given by $\omega_m^i \cdot \omega_m^j = \omega_m^{i+j}$, which is also a root of unity of order m → the set of all roots of unity of order m is closed under multiplication, but not addition
- Cancellation lemma: We also have that, for all positive integers l, m, and integers k, $\omega_{lm}^{lk} = \omega_m^k$ → $\left(\omega_{2m}^k\right)^2 = \omega_{2m}^{2k} = (\omega_{2m}^2)^k = \omega_m^k$ → the squares of the roots of unity of order 2m are just the roots of unity of order m

Discrete Fourier Transform

- Let $C = \langle C_0, C_1, \ldots C_{m-1} \rangle$ be a sequence of m real or complex numbers that is the coefficient representation of polynomial $P_C(x) = \sum_{j=0}^{m-1} C_j x^j$
- Let $\hat{C} = \langle \widehat{C_0}, \widehat{C_1}, \ldots, \widehat{C_{m-1}} \rangle = P_C(\omega_m^k)$ for all $0 \le k \le m - 1$ → the values of P_C at the m roots of unity of order m → $\hat{C}$ is a sequence that is the value representation of P_C → $\hat{C}$ is known as the DFT of C
- That is, the DFT of C is just the value representation of P_C if we use the m^{th} roots of unity as input points
- The DFT of C must have the same length as the original sequence C

- Calculating a DFT without FFT will take O(m^2) time

Fast Fourier Transform: an O(n logn) algorithm for calculating the DFT of an n-length sequence

1. Given a sequence A that is the coefficient representation of P_A, pad it with 0s until its length is the smallest power of 2 greater than 2n+1, and let this length be m
2. Split the sequence into the even and odd power terms
    a. Then rearrange to factor x out of the odd sequence, and make both sequences polynomials in terms of x^2 → e.g. $x^4 = (x^2)^2$

    $$P_A(x) = (A_0 + A_2x^2 + \cdots + A_{m-2}x^{m-2}) + (A_1x + A_3x^3 + \cdots + A_{m-1}x^{m-1})$$
    $$= \left(A_0 + A_2x^2 + \cdots + A_{m-2}(x^2)^{\frac{m-2}{2}}\right) + x\left(A_1 + A_3x^2 + \cdots + A_{m-1}(x^2)^{\frac{m-2}{2}}\right)$$

    b. Define $A^{[0]} = \langle A_0, A_2, A_4, \dots, A_{m-2}\rangle$, and $A^{[1]} = \langle A_1, A_3, A_5, \dots, A_{m-1}\rangle$, so that we have

    $$P_{A^{[0]}}(y) = A_0 + A_2y + A_4y^2 + \cdots + A_{m-2}y^{\frac{m-2}{2}}$$
    $$P_{A^{[1]}}(y) = A_1 + A_3y + A_5y^2 + \cdots + A_{m-1}y^{\frac{m-2}{2}}$$
    $$P_A(x) = P_{A^{[0]}}(x^2) + x \cdot P_{A^{[1]}}(x^2)$$

3. Now that we have two polynomials $P_{A^{[0]}}(y)$ and $P_{A^{[1]}}(y)$ that each have m/2 terms each, recursively repeat step 2 until we have sequences with length 1, at which point we evaluate it and then combine
4. Combine by just placing all values in $A^{[0]}$ then all values in $A^{[1]}$ into A

NOTES:

- In step 3, because m is a power of two, by the cancellation lemma, $y = x^2 = \omega_m^{2k} = \omega_{\frac{m}{2}}^{k}$
- In step 3, even though there are only m/2 distinct roots of unity of order m/2 in each subproblem, since $\omega_{\frac{m}{2}}^{\frac{m}{2}} = 1$, we can just simplify the later terms as repetitions of the earlier terms, and the subproblems have correctly halved in size:

$$P_A(\omega_m^k) = P_{A^{[0]}}((\omega_m^k)^2) + \omega_m^k \cdot P_{A^{[1]}}((\omega_m^k)^2)$$
$$= P_{A^{[0]}}\left(\omega_{m/2}^k\right) + \omega_m^k \cdot P_{A^{[1]}}\left(\omega_{m/2}^k\right)$$

- Thus the recurrence is $T(m) = 2\,T\left(\frac{m}{2}\right) + c\,m = \Theta(m\log m) = \Theta(n\log n)$ (case 2)

Inverse FFT: Converts $\hat{A}$ to A in $\Theta(n \log n)$ time

- Do FFT, but replace ω_m with $\overline{\omega_m} = e^{-\frac{2\pi i}{m}}$, and divide all results by m, then combine as usual

Divide and Conquer strat for multiplying polynomials

1. Use FFT to compute $\hat{A}$ and $\hat{B}$ (convert to value representation using the m^{th} roots of unity as inputs → $O(n \log n)$ time
2. Multiply point by point to get $\hat{C}$, the value representation of P_C also using the m^{th} roots of unity → $\Theta(m) = O(n)$
3. Use inverse FFT to convert $\hat{C}$ into C, the coefficient representation of P_C, and thereby obtain the product polynomial P_C → $O(n \log n)$

Note:

- P_A and P_B do not have to be of the same degree, as the lesser one just gets more 0s padded to its coefficient sequence

The coefficient sequence C is the convolution of sequences A and B

- $A \star B$ denotes the convolution of sequences A and B
- $P_C(x) = P_A(x) \cdot P_B(x)$, and hence its coefficient sequence, can be found in $\Theta(n \log n)$ time using the above algorithm
- Recall that the coefficients of $P_C(x)$ are given by

$$C_t = \sum_{i+j=t} A_i B_j$$

- Hence, FFT is actually used to calculate the convolution of two sequences, though it should be noted that one of the sequences are actually reversed for a convolution??

3. Greedy

Greedy Algorithms: Those that solve a problems by dividing it into stages, and only considers the choice that appears best at each stage

- This reduces the search space, but it is not always clear whether the locally optimal choice (best choice at a given stage) leads to the globally optimal outcome (best solution)
- Greedy algorithms are easy to describe, but hard to justify correctness as they do not consider all possibilities → need to justify why you didn't consider other possibilities, and why the possibilities your algo considered aren't worse than the others
- Usually, greedy algorithms cannot be used when there are two heuristics → e.g. the block stacking formatif
- Examples include BFS, Prim's, Kruskal's

Optimal Selection

Optimal Selection: We are given n things and need to choose things

- Choosing where to place cell towers among n house
- Choosing how to pick the most activities out of n activities without overlapping

Exchange Argument: Show that any alternative solution can be transformed into a greedy solution that is better (or at least not worse)

- Suppose the greedy makes selections $G = \{g_1, g_2, \dots g_r\}$, and some alternative solution selected $A = \{a_1, a_2, \dots, a_s\}$, both selections in some order
  - Prove that no other valid selection/solution can be better (in the case of minimum selections, we want $s \geq r$)
  - Find the first mismatched selection $g_k \neq a_k$, then consider what would happen if we exchanged (swapped) a_k for g_k in the alt solution, and call this swapped sequence of selections $A' = \{a_1, a_2, \dots, g_k, \dots a_s\}$
    - Would A' be valid and result in a better (or at least not worse) outcome?

    - Repeat with comparing G and A’ → find the next mismatch, exchange it to form A”, then prove its valid and no worse than A’ → repeat until we transform A into G
    - Once A matches G, consider how the greedy algo would work, and if it would end there → did they make the same number of selections → does r = s

Greedy Stays Ahead: Greedy solution is ahead (better) of alt solution at any stage, and it stays ahead, then it will remain ahead at the end and ‘win’

- Example: the tower coverage problem → cover as many houses as you can with as few cell towers as possible
    - For all k, the kth tower in the greedy solution is at least as far east (right) as the kth tower in any other solution
    - If this is true, then it would be impossible to cover all the houses using fewer towers

Greedy Stays Ahead → tower coverage problem

- We want to cover as many houses as we can with as few cell towers as possible
- Start with something like induction:
    1. Base case: k = 1 → the greedy places its first towers 5km east of the first house → if an alt placement had its first tower further east, it would not cover the first house, so the claim is true for k = 1
    2. Assumption: suppose the claim is true for some k – 1, and consider the kth tower → the greedy placed towers at g_{k-1} and g_k, and there is a house h such that
    $$g_{k-1} + 5 < h = g_k - 5$$
    3. Then if alt placed towers at a_{k-1} and a_k, the induction assumption/hypothesis tells us that $a_{k-1} \leq g_{k-1}$, so if $a_k > g_k$, then house h will not be covered by either of the towers on either side of it.
    4. Therefore, $a_k \leq g_k$, as required

- Proof:
    1. Suppose the greedy uses r towers, placed at $g_1, \dots, g_r$, and there is a house h at $g_r - 5$, which isn’t covered by the previous tower. Then $h > g_{r-1} + 5$ if it is to be covered
    2. From the induction above, we know that for any other placement of r – 1 towers at $a_1, \dots, a_{r-1}$, we have $a_{r-1} \leq g_{r-1}$

3. Therefore house h can't be covered by the first $r - 1$ towers no matter how they are placed, so at least r towers are needed, and thus the greedy solution is optimal

Optimal Ordering

Optimal Ordering: Given n things, order them in some way to do something optimally

- We prove a way of ordering is optimal by considering adjacent inversions → will a swap in two adjacent and unordered elements result in a better solution?
- If we have two heuristics, we can find a compromise that allows us to consider both at the same time

Proof by contradiction → tape storage problem I → store n files of different lengths on a tape in such a way that search time is minimised, and each file is equally likely to be used

- We want to look through the shortest total length of files on the tape to reach any given file

1. Suppose the files are not stored in ascending order of length. Then there must be two consecutive files that are out of order; say they have lengths $l_i > l_{i+1}$
2. Swapping them reduces the scaled retrieval time from $\ldots + (n - i + 1)l_i + (n - i)l_{i+1} + \cdots$ to $\ldots + (n - i + 1)l_{i+1} + (n - i)l_i + \cdots$.
3. Since $l_i > l_{i+1}$, we have fewer instances of the larger l_i term, and so resolving this adjacent inversion gives us a reduced scaled retrieval time
4. Thus, any arrangements that aren't sorted by length can be improved upon, and resolving all adjacent inversions one-by-one would gradually return us a sorted sequence, and so the best arrangement is the one where files are ordered from shortest to longest

Job Lateness Problem → we have a start time T_0 and a list of n jobs, each with duration times t_i and deadlines d_i. Only one job can be performed at a time, and all jobs have to be completed

- If a job i is completed at $f_i > d_i$, we say it has incurred lateness $l_i = f_i - d_i$
- Schedule all jobs to minimise the largest lateness → two heuristics → prioritise short duration jobs, and prioritise jobs with earlier deadlines
- In this case, a compromise would not work → we should just prioritise deadlines → deadlines do not change, and so by prioritising a shorter duration job with a later deadline, the job that should be done will not until after the shorter duration job, and so

the shorter duration is added to the lateness of the correct job → inversions would fuck up lateness, so resolve it and that will produce a job list scheduled by deadlines

- Suppose we only have two jobs i and j with $t_i < t_j$ and $d_i > d_j$. If we prioritised by duration, we would have $f_i = T_0 + t_i$, and so $f_j = T_0 + t_i + t_j$ → $l_i = T_0 + t_i - d_i$ and $l_j = T_0 + t_i + t_j - d_j$, but if we resolved the inversion, we could have $l_i = T_0 + t_i - d_i$ and $l_j = T_0 + t_i - d_j$, and since $t_i < t_j$ and $d_i > d_j$, we would have a reduction if we resolved the inversion, and so we should → resolving all would give us a sorted list

Applications to Graphs

Strongly Connected Component: A component of a directed graph $G = (V, E)$ where there is a path from u to v and v to u, for all $v, u \in V$ in the component

- To find a strongly connected component
  1. Construct $G_{rev} = (V, E_{rev})$ → G with all edges in reverse direction
  2. Find the set of vertices reachable from v in both G and G_{rev} by using DFS/BFS, and the common vertices form a strongly connected component
  3. Repeat for all v that are not part of a strongly connected component, until every v is in one
- O(V(V + E)) time complexity → O(V) repeat for all v, and BFS/DFS O(V + E)

Tarjan's Algorithm: An algorithm to find strongly connected components in O(V + E) time

1. Do a regular DFS on the graph with a stack
2. When an item is pushed onto the stack, mark it as in-stack, unmarking it when popped
3. If we want to push a vertex that is already marked as in-stack, we have a strongly connected component → each vertex on the stack after this vertex can reach/be reached from this vertex → each vertex on the stack after this vertex is part of the strongly connected component

Condensation Graph:

- The strongly connected components are disjoint sets that partition V
- Let C_G be the set of all strongly connected components of G
- Define the condensation graph $\sum_G = (C_G, E^*)$, where

$$E^* = \{(C_{u_1}, C_{u_2}) \big| (u_1, u_2) \in E, C_{u_1} \neq C_{u_2}\}$$

- That is, the vertices of $\sum_G$ are the strongly connected components of G
- The edges of $\sum_G$ are the edges in G that are not within a strongly connected component (duplicates are ignored → edges that connect two different strongly connected components
- Note that $\sum_G$ is directed and acyclic

Topological Sorting: An ordering of vertices in a graph such that all vertices can reach all vertices that are later in the order, but none that are before it

- Let $G = (V, E)$, and $n = |V|$ → a topological sort of G is a linear ordering (enumeration) of its vertices σ → $V \rightarrow \{1, \ldots, n\}$ such that if there exists an edge $(v, w) \in E$, then v precedes w in the ordering → $\sigma(v) < \sigma(w)$
- Directed acyclic graphs have at least 1 topographical sort
- The vertex with no incoming edges is the first vertex in the order
- To find a topographical sort:
  1. Initialise an empty list L of vertices, an array D containing the in-degrees of the vertices, and a set S of vertices with no incoming edges
  2. While S is non-empty, select a vertex u in S, remove it, and then append it to L
  3. For every outgoing edge $e = (u, v)$, remove the edge from the graph and decrement D[v] accordingly
  4. If D[v] is now 0, insert v into S
  5. Repeat step 2-4 until S is non-empty
  6. If there are no edges remaining, L is a topological ordering. Otherwise, the graph has a cycle
- O(V + E) time complexity
- Often good to get the topological sort of a graph then using that to make the probem easier to solve

Dijkstra's Algorithm: Algorithm to find the single source shortest path → the shortest paths from a source vertex to all other vertices

- By using a min heap to store the unvisited vertices and their known distance from the source vertex (shortest distance at top), finding and removing the min vertex takes O(log V) time → updating or inserting in the heap also takes O(log n) → Dijkstra has time complexity O((V + E) log V)
- Although using a Fibonacci heap would be even faster (O(m + n logn)), we don't use that in 3121

Minimum Spanning Tree: The MST of G is the tree subgraph of G that has the minimum total weight

- Prim's Algorithm: Start with any vertex and gradually build up the MST from that edge by adding the smallest edge where one vertex is in the MST and the other isn't, repeating until n – 1 edges have been added
- Kruskal's Algorithm: Add the lightest edge to the MST if it does not form a cycle, repeating until n – 1 edges have been added → forms multiple components that are trees → forests
  - To quickly determine if adding an edge will form a cycle or not, we track which component each vertex is in → if u and v are in the same component, don't add (u, v) since that will create a cycle → if u and v are not in the same component, add (u, v) and merge the components
  - To track and merge components quickly use the union-find data structure

Union-Find: The union-find data structure handles disjoint sets and supports three operations by using three arrays

- Three arrays
  - Set → set[i] = j → item i belongs to set j
  - Size → size[j] → the number of elements in set j
  - Elements → element[j] → a linked list of the items in set j
- Note: If $Set[i] \neq i$, then item i has already been merged into some other set, so Size[i] = 0, and Elements[i] is an empty list
- Initialise(S) → returns a structure where all items (vertices) are in distinct single-element sets → Set[i] = i, Size[i] = 1, Elements[i] = [i -> null] → O(n) = O(|S|) time
- Find(a) → returns the set that item a belongs to → O(1) time
- Union(A, B) → merges the sets A and B to get $A \cup B$
  - The first k union operations run in total time O(k logk) → aggregate time complexity → don't use time complexity of a single operation
  - Some Union ops use Union(a, b) and need you to use Find to get A B
    1. Find the sets a and b belong to → A = Set[a], B = Set[b]
    2. If $Size[A] < Size[B]$, do Union(B, A) instead
    3. Merge the smaller set B into the larger set A
       a. For each m in Elements[B], update Set[m] = a
       b. Update Size[A] = Size[A] + Size[B], and Size[B] = 0
       c. Append Elements[B] to Elements[A], and replace Elements[B] with an empty list
- Note: the new value of Size[A] is at least twice the value of Size[B] → $Size[A] \geq Size[B]$

- Note: the first k union operations can touch at most 2k items of S → the set containing an item m after the first k operations can have at most 2k items → since each union operation that changes Set[m] at least doubles Size[Set[m]], Set[m] has changed at most log(2k) times → the first k union operations will cause at most 2k log(2k) updates in the set array → k logk aggregate time complexity

Kruskal's Algorithm with the Union-Find data structure

1. Sort the m edges of graph G from smallest to largest → O(m logm) = O(m logn)
2. Use Find op to check that u, v are in different components → O(1)
3. If they are in the same component, don't add, otherwise add (u, v) to the MST and do Union(u, v) to merge
4. Repeat 2-3 until you have added n – 1 edges → n – 1 union ops → O(n logn)

- Total time complexity is O(m logn + m + n logn) = O(m logn)

4. Flow Networks

A Flow Network $G = (V, E)$ is a directed graph in which each edge $e = (u, v) \in E$ has a positive integer capacity $c(u, v) > 0$

- Two special vertices → source s and sink t → we want to send 'flow' from s to t
- A flow in G is a function $f: E \rightarrow [o, \infty), f(u, v) \geq 0$, which satisfies
  1. The Capacity Constraint: for all edges $e = (u, v) \in E$, we require $f(u, v) \leq c(u, v)$ → the flow through any edge does not exceed its capacity
  2. Flow Conservation: for all vertices $v \in V \backslash \{s, t\}$, we require
  $$\sum_{(u,v)\in E} f(u, v) = \sum_{(v,w)\in E} f(v, w)$$
  → the flow into a vertex (other than s or t) is equal to the flow out of that vertex

- The value of a flow is defined as
  $$|f| = \sum_{v:(s,v)\in E} f(s, v) = \sum_{v:(v,t)\in E} f(v, t)$$
  → The value of the flow is the units of flow arriving at the sink, or equivalently the units leaving the source
  - Every vertex other than s and t can only redirect flow, they can't add or remove units, so what we send is what we get → we wouldn't send more than what we can receive at a given path

Integrality Theorem:If all capacities are integers, then there is a flow of maximum value such that $f(v, u)$ is an integer for each edge $(u, v) \in E$ → there is at least one solution entirely in integers

- We only consider integer solutions in this course

Maximum Flow Problem: What is the maximum possible flow for a graph G

- Residual Flow Network: Given a flow network, the residual flow network is the network made up of the remaining capacities of each edge → no flow, only residual capacities
- Suppose the original flow network has an edge (v, w) with capacity c, and that f units of flow are sent from v to w, then in the residual flow network we have
  - An edge (v, w) with capacity c – f

    - An edge (w, v) with capacity f
- Sending flow on the 'virtual' edge (w, v) counteracts the already assigned flow from v to w in the flow network
    - If the flow network had edges (v, u) and (u, v), we combine the capacities of the original and virtual edges in the same direction in the residual network
    - If the edge has capacity 0 (f = 0 or f = c), we don't need to add them in

Augmenting Path: an augmenting path is a path from s to t in the residual flow network

- Allows us to consider both directions of each edge → sending more flow forwards, or sending flow backwards to counteract earlier incorrectly assigned flow
- The capacity of an augmenting path is the capacity of its bottleneck (smallest capacity) edge
- We send the capacity of the augmenting path along the augmenting path, recalculating the flow and residual capacities for each edge used
- Suppose we have an augmenting path of capacity f, including an edge from v to w. To correct any mistakes and consider both directions in the residual flow network, we:

    SUPER DODGY ASK TUTOR TO CHECK

    1. Cancel up to f units of flow being sent from w to v → increase the residual capacity from w to v by f → add f units to the backwards edges
    2. Add the remainder of these f units to the flow being sent from v to w → reduce the residual capacity from v to w by f → subtract f units to the forwards edges

Ford-Fulkerson Algorithm

- Keep adding flow through new augmenting paths, until there are no paths from s to t → when there are no more augmenting paths, you have achieved the largest possible flow in the network
    1. Start with the residual flow network of the unaltered flow network
    2. Find any augmenting path (using BFS or DFS)
    3. Send f (the capacity of the augmenting path) along the augmenting path → for all (v, u) on the augmenting path
        a. Subtract f from f(u, v), THEN add f – f(u, v) to f(v, u)
        b. Subtract f from f(u, v) then add f to f(v, u) in the flow network
        c. Subtract f from (v, u)
        d. Add f to (u, v)
    4. Repeat 2-3 until step 2 fails

- If all capacities are integers, every augmenting path increases flow by at least 1 unit, and so eventually we must have some edges will full capacity until there is no path from s to t

Max-Flow Min-Cut Theorem: The maximal amount of flow in a flow network is equall to the capacity of the cut of minimal capacity

- A cut in a flow network is any partition of the vertices of the underlying graph into two subsets S and T such that:
  - $S \cup T = V$
  - $S \cap T = \emptyset$
  - $s \in S$ and $t \in T$
- The capacity c(S, T) of a cut (S, T) is the sum of capacities of all edges leaving S and entering T

$$c(S,T) = \sum_{(u,v)\in E} \{c(u,v) : u \in S, v \in T\}$$

- Given a flow f, the flow f(S, T) through a cut (S, T) is the total flow through edges from S to T minus the total flow through edges from T to S → how much flow is actually going from the source side to the sink side

$$f = (S,T) = \sum_{(u,v)\in E} \{f(u,v): u \in S, v \in T\} - \sum_{(u,v)\in E} \{f(u,v): u \in T, v \in S\}$$

  - For any flow f, the flow through any cut (S, T) is equal to the value of the flow → $f(S,T) = |f|$ → and for any cut, $|f| = f(S,T) \leq c(S,T)$
- To find the min-cut, flood the

Proof of Correctness

- Firstly, the algorithm will terminate → Integrality theorem → if all capacities are integers, then every augmenting path increases flow by at least 1 unit, and so eventually edges will hit full capacity and there will be no path from s to t, so the algorithm terminates
- Assumption: Assume the Ford-Fulkerson algorithm has terminated. Then there are no more augmenting paths from the source s to the sink t in the last residual flow network
  1. Define S to be the source s and all vertices u such that there is a path in the residual flow network from s to u, and define T to be everything else
  2. Since there are no more augmenting paths from s to t, clearly the sink t belongs to T
- Claim: All edges from S to T are fully occupied with flow (, and all edges from T to S are empty

1. Suppose an edge (u, v) from S to T has any capacity remaining. Then in the residual flow network, the path from s to u could be extended to a path from s to v. This contradicts our assumption that $v \in T$
2. Suppose an edge (y, x) from T to S has any flow in it. Then in the residual flow network, the path from s to x could be extended to a path from s to y. This contradicts our assumption that $y \in T$
3. So we must have that all edges from S to T are occupied with flows to their full capacity, and that there is no flow from T to S. This means the flow across the cut (S, T) is precisely equal to the capacity of this cut → $f(S,T) = c(S,T)$

- Conclusion: Thus, such a flow is maximal and the corresponding cut is a minimal cut
- FUCKING I THINK → instead of comparing different flows, we show that the flow produced by the algo is equal to the capacity of a cut, and hence that all other flows must be less than or equal to the solution → the algo can also find the minimal cut of a flow network

Ford-Fulkerson Time Complexity

- The number of augmenting paths can be up to the value of the max flow $|f|$
- Each augmenting path is found in $O(V + E)$ time (DFS or BFS), and in any sensible network we have $V \leq E + 1$, so we can write this as $O(E)$
- Therefore the overall time complexity is $O(E|f|)$ → This can grow exponentially, so we want a better way?? → Edmonds-Karp algo

Edmonds-Karp Algorithm: always choose the augmenting path which consists of the fewest edges

- Same shit as Ford-Fulkerson, except we have to choose the fewest edge augmenting path, not just any
- Finding the augmented path using BFS will guarantee the fewest edge augmenting path in $O(V + E) = O(E)$ time
- Using the fewest edge augmenting path, the number of augmenting paths is bounded by $O(VE)$
- Time complexity is $O(VE^2)$
- Since Edmonds-Karp is a specialisation of Ford-Fulkerson, we say that the time complexity is whichever of $O(VE^2)$ and $O(E|f|)$ is more constraining → $O(VE^2)$ if few vertices and edges, but high capacity, $O(E|f|)$ if many vertices and edges but really low capacity → we say Edmonds-Karp has time complexity $O(\min(E|f|, VE^2))$

Note: Don't use Dinic's $O(V^2E)$ and Preflow-Push $O(V^3)$ algorithms in 3121

Applications of Flow Networks

Networks with multiple sources and sinks can be reduced to networks with a single source and sink by adding a super-source and super-sink which connect to all sources and sinks with infinite capacity

- If there is some constraint on how much flow can be sent from each source, we can place that constraint on the capacity of the edge from the super-source to that source

Networks with vertex capacities $C(v_i)$ that limit the throughput of the flow going to and from the vertex v_i can be reduced to networks with only edge capacities

- If a vertex v_i has capacity $C(v_i)$, then we can create two vertices u_i, w_i and create an edge between them of capacity $C(v_i)$ → u_i and w_i are then referred to as the in and out vertices of v_i
- All edges going to v_i now go to u_i, and all edges leaving v_i now leave from w_i
- Problems involving grids with capacities can also be solved like this by forming edges from each cell to their adjacent ones, and treating a cell as a vertex with capacity
  - If grid has n rows and m columns, time complexity of Edmonds-Karp will be $O\left((n+m)(nm)^2\right)$

Vertex Disjoint Paths

- Vertex disjoint paths are paths who share no vertices
- To find the max number of vertex disjoint paths:
  1. Create a super source connected to all starting points, and a super sink connected to all end points
  2. For every other non-starting and non-ending vertex, treat them as vertices with capacity and create v_{in} and v_{ou}
  3. Assign all edges capacity 1
  4. Run Edmonds-Karp and the max flow is equal to the min cut, and since edges have capacity 1, the min cut is the minimum number of vertex disjoint edges, and hence the maximum number of vertex disjoint paths

- There are at most 2n vertices and exactly n+m edges in the created network → Using $O(VE^2)$, we get $O(n(n+m)^2)$ → but low capacity so we can use $O(E|f|) = O\big(n(n+m)\big)$

Bipartite Matchings

- Bipartite graphs are those where its vertices can be split into two disjoint sets A and B such that every edge has one end in A and the other end in B
- A matching in a graph $G = (V, E)$ is a subset $M \subseteq E$ such that each vertex of the graph belongs to at most one edge in M → the maximum matching is the largest possible size of set M
- To find the maximum matching:
  1. Create a super source and super sink → connect the super source to all vertices in A, and connect all vertices in B to the super sink
  2. Add all edges from original graph
  3. Assign all edges capacity 1
  4. Run Edmonds-Karp and the flow is the maximum matching
- Common examples of bipartite graph scenarios → whenever we have two different types of nodes, when we work with grids, adjacent cells form bipartite graphs

Time Complexity

- When considering the time complexity, we need to think about how many edges and vertices there are after creating the new networks to solve the problems in terms of the original number of edges and vertices
- Also consider which Edmonds-Karp time complexity is more restrictive
  - $O(VE^2)$ if there are few vertices and edges, but edge capacities are high
  - $O(E|f|)$ if there are many vertices and edges, but edge capacities are low
-

5. Dynamic Programming

Intro

Shits like divide and conquer, except we use the solutions to subproblems together to get the solution to the main problem

- Solving a large problem recursively by building from (carefully) chosen subproblems → subproblems should be chosen such that:
    - Answers to subproblems (child instances) can be combined to answer a larger subproblem (parent instance)
    - The same subproblem occurs several times in the recursion tree → when we solve a subproblem, we store the solution so that we can solve the subproblem again when it appears in O(1) time by looking it up
- Divide and conquer breaks a larger problem into disjoint subproblems that are then solved recursively and recombined, whereas dynamic programming solves subproblems and uses the solutions to solve other subproblems required to solve the main problem

For 3121, dynamic programming algorithms are generally executed:

- In an iterative (bottom-up) sense → start with the base case, then solve subproblems that gradually increase in size until we can solve the main problem using the recurrence
    - Think of it as we start from the base case and fill out the lookup table storing subproblem solutions gradually, and using those stored solutions to solve larger subproblems until we can solve the main problem
    - Can be implemented recursively (top-down), but we usually talk about it as iterative
- By using a lookup table (usually an array indexed by the parameters of the subproblems) to store the solutions to subproblems

- Parameter refers to the main point of the problem → e.g. tower of Hanoi how many moves for a tower of height n, n is the parameter

Building dynamic programming algos

- Need to solve the subproblems in an order that respects the dependencies → tower of Hanoi, solve smaller towers before solving big towers
- The original problem may be one of our subproblems, or it may be solved by combining results from several subproblems → need to describe the process of combining
- Time complexity is usually given by multiplying the number of distinct subproblems by the average time taken to solve a subproblem using the recurrence

To justify dynamic programming algorithms, we need to justify briefly that:

- The recurrence correctly relates subproblems to each other → uses solutions to one subproblem to solve another one correctly
- The base cases are solved correctly
- The overall answer has been correctly solved
- Can use induction, but not required → good way of thinking about them tho

To develop dynamic programming algorithms:

1. Choose an order to build up the solution → order must respect the dependencies
2. Pick a tentative (non-fixed) subproblem specification, starting with just enough parameters to answer the overall problem
3. Try to make a recurrence between these subproblems
    a. If unable to determine which transitions are allowed using only the step 2 parameters, add more and try it until it stabilises
    b. Solve the necessary base cases
4. Analyse time complexity
    a. Removing unnecessary parameters may reduce the number of subproblems
    b. Using data structures can reduce time complexity

One Parameter, Constant Recurrence

We only use one parameter, and subproblems take constant time to solve

- Time complexity is hence just the maximum number of subproblems

Hopscotch dynamic programming algorithm development example:

1. Choose an order → start from smaller numbered squares to larger ones
2. Subproblem specification → how many ways are there to reach the ith square
3. Recurrence → num(i) = num(i – 1) + num(i – 2), because every square can be reached by any sequence of moves to reach i – 1 then a step, or any sequence of moves to reach i – 2 then a jump → base case num(0) = 1, num(1) = 1 since the only way to reach square 1 is by taking 1 step
4. Overall solution → is a subproblem num(n)
5. Time complexity → num(i – 1) + num(i – 2) takes O(1) since its just addition, and the order ensures that num(i – 1) and num(i – 2) have already been solved (O(1) lookup time for solved subproblems) → O(n) subproblems worst case, so O(n) overall

One Parameter, Linear Recurrence

We only use one parameter, and subproblems take linear time to solve

- Time complexity is hence just the maximum number of subproblems multiplied by the cost of solving each subproblem

Longest increasing subsequence example:

1. Order → consider the longest increasing subsequence of a subsequences from index 1 to i starting with small values of i

2. Subproblem specification → find the longest increasing subsequence up to index I → Q(i) is the subproblem with index i, and opt(i) is the optimal solution to Q(i)
3. Recurrence → Find all indices j < i such that A[j] < A[i], and choose m = max(A[j]), then extend that sequence by A[i] → if no such j exists, opt(i) = 0 → the longest subsequence up to index i is opt(i) = opt(m) + 1 → base case if i = 1, longest subsequence = 1, since there are no previous indices to consider
4. Overall solution → NOT A SUBPROBLEM → find the max opt() in the lookup table storing solutions, and that is the answer → consider case where A[n] < A[j] for all j
5. Time complexity → finding m requires finding the maximum by iterating over all indices smaller than i, which takes O(n), so we have O(n) subproblems taking O(n) time → $O(n^2)$ time complexity

Note:

- If we wanted to find the actual sequence, we would track the previous index, and that index's previous index to track the path → predecessor array type shit
- The notation for the k that minimises opt() is $argmin_{1<k<n} opt()$ → returns k, not the minimum result → argmax also exists

Graph Application

A topological ordering of vertices is where all edges point left to right → a vertex earlier in the order only has edges to vertices later in the order

- An acyclic directed graph always has at least one valid topological ordering → can be found in $O(|V| + |E|)$ time → good to always put DAG in topological ordering then solve from there → all dependencies are 'one way', and so we can use dynamic programming

Shortest single-source path → $G = (V, E)$ and each edge e has corresponding weight $w(e)$, which may be negative

- Let $n = |V|$ and $m = |E|$
- For a general graph, if all edge weights are positive, we can solve using Dijkstra in $O(m \log n)$, otherwise we can use the Bellman-Ford algorithm to find the single-source shortest path in $O(nm)$
- For a DAG, we can solve in $O(n + m)$ using dynamic programming

Shortest path in DAG → $O(n+m)$

- Subproblem: the shortest path to each vertex
- Recurrence: Consider the path to all vertices v with an edge to t, sum the path with the weight of edge (v, t), then select the minimum weight option
- Base Case: travelling from a vertex to itself costs 0
- **Subproblem**: for all $t \in V$, let $P(t)$ be the problem of determining $opt(t)$, the length of the shortest path from s to t
- **Recurrence:** For all $t \neq s$,
$$opt(t) = \min[opt(v) + w(v,t)],$$
where $(v,t) \in E$
- **Base Case**: $opt(s) = 0$
- **Order of Computation**: Solve in topological order → needs to be DAG
- **Time Complexity**: We iterate over all n vertices, but only consider each edge once (at its endpoint when we consider what edges go to t), so we have $O(n+m)$

5 mc, 3 short answer, 2 algo design

Single shortest paths in non-DAGs

- Dijkstra's does not work with graphs that have negative weights → we still disallow cycles of negative total weight since some vertices would have weight negative infinity → MAYBE detect cycles of negative infinity weight and condense them to a single vertex of negative infinity, and any vertices that can include that negative infinity vertex has negative infinity weight → 0 weight cycles are fine
- Observe that, for any vertex t, there is a shortest s-t path without any cycles → every shortest s-t path contains any vertex v at most once, and thus has at most n – 1 edges

Bellman-Ford

- For every vertex t, find the weight of a shortest s-t path consisting of at most i edges for $0 \leq i \leq n-1$
- Suppose the path $p = s \to \cdots \to v \to t$, where $p' = s \to \cdots \to v$, then p' must itself be the shortest path from s to v of at most i – 1 edges for p to be optimal, which is another subproblem

- **Subproblems**: for all $0 \leq i \leq n-1$, and all $t \in V$, let $P(i,t)$ be the problem of determining $opt(i,t)$, the length of a shortest path from s to t, which contains at most i edges
- **Recurrence**: for all $i > 0$ and $t \neq s$,
$$opt(i,t) = \min_{v|(v,t)\in E}\{opt(i-1,v) + w(v,t)\}$$
- **Base Cases**: $opt(i,s) = 0$, and for $t \neq s$, $opt(0,t) = \infty$
- **Order of Computation**: Solve subproblems $P(i,t)$ in increasing order of i, and any order of t, since $P(i,t)$ depends only on $P(i-1,v)$ for various v
- **Overall Answer**: the list of values $opt(n-1,t)$ over all vertices t → shortest path from s to all vertices t
- **Time Complexity**: i from 0 to n – 1, so n iterations, and in each iteration, each edge (v,t) is considered only once, so time complexity is $O(nm)$
- Can reconstruct the shortest s-t oath using predecessor array → or defining a subproblem pred(i, t) to be the immediate predecessor of vertex t on a shortest s-t path of at most i edges
    - The additional recurrence is:
$$pred(i,t) = argmin_{\left(v\middle|(v,t)\,\in\,E\right)}\{opt(i-1,v) + w(v,t)$$
- NOTE: Bellman-Ford builds a table of size $O(n^2)$ with a new row for each 'round' (i) → since including $opt(i-1,t)$ as a candidate for $opt(i,t)$ doesn't change the correctness of the recurrence, we can instead maintain a table with only one row of size $O(n)$ and overwrite it at each round

All Pairs Shortest Paths

- We want to find the weight of the shortest path from every vertex s to every other vertex t without running a single shortest path algo on every vertex

Floyd-Warshall - preamble

- Label vertices $v_1, v_2, \dots, v_n$, and let S be the set of vertices allowed as intermediate vertices → initially S is empty, and we add vertices one at a time
- The shortest path from s to t using the first k vertices as intermediates is an improvement on the answer to the previous round in cases where going from $s\ to\ v_k\ to\ t$ is shorter while still using only the first k vertices as intermediates

Floyd-Warshall – Dynamic Programming

- Subproblem: we determine $opt(i,j,k)$, the length of the shortest path from i to j using only vertices $v_1, v_2, \dots, v_k$ intermediate vertices
- **Recurrence:** for all $1 \le i,j,k \le n$,

$$opt(i,j,k) = \min[opt(i,j,k-1), opt(i,k,k-1) + opt(k,j,k-1)]$$

  - Second case $opt(i,k,k-1) + opt(k,j,k-1)$ is shortest path from v_i to v_k + shortest path from v_k to v_j using all vertices up to, but not including, k (since k is already used)
- **Base Cases:**

$$opt(i,j,0) = \begin{cases} 0, & i = j \\ w(i,j), & (v_i, v_j) \in E \\ \infty, & otherwise \end{cases}$$

- **Order of Computation**: Solve subproblems in increasing order of k, and any order of i and j, since $P(i,j,k)$ depends only on $P(i,j,k-1)$, $P(i,k,k-1)$, and $P(k,j,k-1)$
- **Overall Answer:** The table of values $opt(i,j,n)$ over pairs of vertices i and j → cell [i][j] contains the weight of the shortest path from v_i to v_j
- **Time Complexity:** $O(n^3)$ subproblems that each take $O(1)$ time → $O(n^3)$ time complexity

Applications to String Matching

We can speed up string comparisons by using hashing

- We compare hash values in constant time
- Let $A_s = a_s a_{s+1} \dots a_{s+m-1}$ be an m long substring of A starting at s
- We can compute hash values of strings quickly by mapping the characters in the alphabet of size d to an integer $0 \le i \le d-1$ → e.g. a = 0, b = 1, c = 2, etc.
  - a_i, b_i refer to the ID of the symbol a_i and b_i
  - We can identify a string as a sequence of IDs
  - We can view these IDs as digits in base d and construct the hash value of the string $B = b_1 b_2 \dots b_m$ as follows

    $$h(B) = d^{m-1} \times b_1 + d^{m-2} \times b_2 + \dots + d \times b_{m-1} + b_m$$

    which can be evaluated efficiently using Horner's rule

    $$h(B) = b_m + d \times \left(b_{m-1} + d \times \left(b_{m-2} + \dots + d \times (b_2 + d \times b_1)\right)\right)$$

    and thus requiring only m – 1 additions and m – 1 multiplications → linear time computation

- Choose some large prime number p and define the actual hash we use $H(B) = h(B)\ mod\ p$ → $h(B)$ can be impractically big, so we want to limit the value to mod p → don't want p to be too big or integer overflow but only matters for implementation
- Now for each contiguous substring $A_s = a_s a_{s+1} \dots a_{s+m-1}$, we compute $H(A_s) = d^{m-1}a_s + d^{m-2}a_{s+1} + \cdots + d^1 a_{s+m-2} + a_{s+m-1}\ mod\ p$ → But note that there's a lot of overlap between each substring → we can use a rolling hash to compute them → $H(A_{s+1}) = d \times H(A_s) - d^m a_s + a_{s+m}\ mod\ p$ → can compute $H(A_{s+1})$ from $H(A_s)$ in constant time
- We can now compare the hash values of $H(B)$ and $H(A_s)$ → if $H(B) = H(A_s)$, there might by a match so we check if A_s matches B character by character

Rabin-Karp Algorithm

1. Compute $H(B)$ and the base case $H(A_1)$ in $O(m)$ time using Horner's rule
2. Then compute the $O(n)$ subsequent values of $H(A_s)$ in constant time using the recurrence
$$H(A_{s+1}) = d \times H(A_s) - d^m a_s + a_{s+m}\ mod\ p$$
3. Compare $H(A_s)$ with $H(B)$, and checking the strings A_s and B character-by-character if they match

- Since p is large and prime, false positives are unlikely, so the algorithm runs quickly in the average case → $O(n)$ average case → but we cannot achieve useful bounds for worst case performance when hashing → $O(mn)$ worst case

Knuth-Morris-Prat Algorithm → $O(n + m)$ string matching algo

- When we hit a conflict in A_s and B, we try to salvage part of the matched section
- Let $B_k = b_1 b_2 \dots b_k$ denote a prefix of length k of the pattern B → the first k characters of B
- Suppose we are partway through the text, and the k most recent characters matched with B_k
  - If the next character of the text is b_{k+1}, we can extend our match from length k to length k + 1
  - If the next character of the text is NOT b_{k+1}, we cannot continue the current partial matching, but can try to extend some shorter partial matching
    - Suppose B = xyxyxzx, and we have read the text up to xyxyx, we would have maintained a partial match of length 5 → if the next character in the text is y, we cannot extend to length 6, but can fall back to our partial match xyx, which has length 3, and extend that to length 4 with the y

        - xyxyx → we take the highlighted match of length 3, then extend that to length 4 with the y to get xyxy, which is a match of length 4 to string B
        - The shorter partial matching we should extend is the longest partial matching that is both a prefix and suffix of B_k
            - If it wasn't a prefix, it can't be the start of a new partial match
            - If it wasn't a suffix, it isn't the most recent sequence of characters in the text, so it wouldn't be contiguous

- Let $\pi(k)$ be the length of the longest string which is both a prefix and suffix of B_k
    - Anytime we fail to extend our current partial match (of length k), we can instead try to extend the longest prefix-suffix match (of length $\pi(k)$)
    - This allows us to skip some starting points that the naïve algorithm would have tried, and also to skip rematching some characters
    - If we can't extend the $\pi(k)$ long partial match, try $\pi\big(\pi(k)\big)$, or if we can only extend the $\pi(k)$ long partial match for l more characters, try $\pi(\pi(k) + l)$
- The algo works by maintaining two pointers l and r into the text, which record the left and right boundaries (inclusive) of our current partial match
    - Initially, l = 1, r = 0
    - $w = r - l + 1$ → w is the length of the current partial match
    - If we have a match, we extend the length of the partial match by moving r right 1 (incrementing by 1)
    - If we do not have a match, move l right by $w - \pi(w)$ → will reduce w to $\pi(w)$, then check if we can extend it
    - If a match of length 0 cannot be extended, increment both l and r by 1

KMP

1. Compare the next character of the text a_{r+1} to b_{w+1}
    a. If the characters match, extend the current partial match → increase r by 1
        i. If $w = r - l + 1 = m$, report a match for the entire pattern and increase l by $w - \pi(w)$ (reduce w to $\pi(w)$) , and try to extend the new current partial matching with the current a and b
    b. If the characters do not match and $w \neq 0$, increase l by $w - \pi(w)$ (reduce w to $\pi(w)$), and try to extend that match with the current a and b
    c. If the characters do not match and w = 0, increase both l and r by 1
2. Repeat step 1 until there is no next character

- After each comparison, at least one of the pointers l or r must move forward 1, so each pointer can take up to n values, so we have $O(n)$ steps and each step is $O(1)$ character comparison and pointer incrementation → $O(n)$ overall time complexity

Failure function $\pi(k)$

- The longest prefix-suffix of B_{k+1} is, ideally, the extension of the longest prefix-suffix of B_k, namely $B_{\pi(k)}$
- If $b_{k+1} = b_{\pi(k)+1}$, then $\pi(k+1) = \pi(k) + 1$
- If $b_{k+1} \neq b_{\pi(k)+1}$, but $b_{k+1} = b_{\pi(\pi(k))+1}$, then $\pi(k+1) = \pi(\pi(k)) + 1$,
- And so on with base case $\pi(1) = 0$
- Subproblem: let $P(k)$ be the problem of determining $\pi(k)$, the length of the longest prefix-suffix of B_k
- Recurrence: for $k \geq 1$,

$$\pi(k+1) = \begin{cases} \pi(k) + 1, & if\ b_{k+1} = b_{\pi(k)+1} \\ \pi(\pi(k)) + 1, & if\ b_{k+1} = b_{\pi(\pi(k))+1} \\ & \dots \end{cases}$$

- Base Case: $\pi(1) = 0$
- Order of Computation: Increasing order of k
- Overall answer: The entire list of values $\pi(k)$
- Time complexity: $O(m)$

6 . Intractable Problems

Algorithms is about solving problems systematically and efficiently

- Intractable problems are problems that are very difficult to solve systematically and efficiently

Decision Problems

Decision Problems

- Problems where the answer is either yes or no
- E.g. Determining if there is a Hamiltonian Path in a graph, or Is there a path shorter than this number, or is this number a prime number
- A decision problem $A(x)$ is in class P (polynomial time, denoted $A(x) \in P$) if there exists a polynomial time algorithm which solves it
    - The set of all decision problems solvable in polynomial time
- A decision problem $A(x)$ is in class NP (non-deterministic polynomial time, denoted $A(x) \in NP$) if there is a problem $B(x, y)$ (often referredto as a verifier) such that:
    - For every input x, $A(x)$ is true if and only if there is some y for which $B(x, y)$ is true, and
    - The truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x only
    - We call y the certificate → it provides evidence for why x is a YES instance of problem A
    - Generally problems of the form ‘does there exist…’ → e.g. does there exist a Hamiltonian path
    - NP is the set of all decision problems with the property that ‘yes’ answers are verifiable in polynomial time

- P is a subset of NP, since an algorithm that solves the problem in polynomial time can also verify yes answers in polynomial time

Polynomial Time

- A (sequential) algorithm is said to be polynomial time, if for every input, it terminates in polynomially many steps in the length of the input → length = length of binary/unary representation of input → number of symbols needed to describe the input precisely
    - E.g. knapsack runs in $O(nC)$ time, but in binary C has length $\log_2 C$
    - Supposing the length of the input is denoted by n, the worst case complexity satisfies $T(n) = O(n^k)$ for some integer k
    - E.g. merge sort requires $n \log n$ comparison, and $n \log n = O(n^2)$ so it runs in polynomial time, but
- Polynomial time problems are said to be tractable → polynomial time is usually efficient and computable (the worst case we come across may be $O(n^4)$, but we don't really get worse than that)

Length of Inputs: Integers

- Length of inputs refers to the number of symbols required to describe the input precisely
    - If input x is an integer, then $|x| = \lceil \log_2 x \rceil$ can be taken to be the number of bits in the binary representation of x
- The definition of polynomial time computability is quite robust with respect to how we represent inputs → we could instead define the length of int x as the number of digits in its decimal representation instead of binary → $\log_{10} x$
    - This can only change the constants involved in the expression $T(n) = O(n^k)$, where n is the number of symbols used to describe x, but , but not the asymptotic bound

Length of Input: Arrays

- If the input is an array A, then each entry A[i] contributes $\log_2 A[i]$ bits for a total of $\log_2 A[1] + \cdots + \log_2 A[n]$ bits
- If the array entries are guaranteed to be at most M, then this can be abbreviated to $n \log_2 M$ → a polynomial time algorithm therefore runs in time polynomial in n and $\log M$
    - Knapsack has input size $n \log C$, but DP runs in $O(nC)$, and since $C = 2^{\log_2 C}$, we know that C isn't bounded by any power of $\log_2 C$, and hence that this is not a

polynomial time algorithm → pseudo-polynomial → it is polynomial in the inputs themselves, but not in the size of the inputs

Length of Input: Weighted Graphs

- If the input is a (simple) weighted graph G with edge weights up to W, then G can be described using:
  - Adjacency List → for each vertex, store a list of edges it is incident with alongside their integer weights in binary → $O(E \log_2 W)$ space
  - Adjacency Matrix → for each $v_i, v_j \in V$, store the weight of the edge from v_i to v_j (or some default value if no edge) in cell [i][j] → $O(V^2 \log_2 W)$ space
  - Sparse graphs have $E \leq V^2$, so adjacency lists are much more concise → but doesn't matter when discussing polynomial time shit
- E.g. Ford-Fulkerson runs on a graph with E edges of capacity up to C, so the size of the input is $O(E \log_2 C)$
  - Since FF runs in $O(E|f|)$, and $|f| \leq EC$, it is not a polynomial time algorithm
  - Edmonds-Karp, however, runs in $O(VE^2)$, and since $V \leq E - 1$, this is a polynomial time algorithm

Satisfiability problem (SAT) → we have a propositional formula in CNF form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, where each clause C_i is a disjunction (∨) of propositional values or their negations, is there an evaluation of propositional variables which makes the formula true?

- SAT is in class NP, since we can determine in polynomial time whether the formula is true if we are given an evaluation of the propositional variables
- However, SAT is in class P IF each clause contains exactly two variables (2SAT), as we can solve it in linear time using SCCs and topological sort

Is every problem in NP also in P?

- The **"$P \neq NP$"** hypothesis → NP is a strictly larger class of decision problems P
- No actual proven answer
- More complexity Classes:
  - NP-H is the set of all decision problems V with the property that every problem in NP is polynomially reducible to V
    - You can think of NP-H as problems at least as hard as everything in NP
    - A decision problem V is NP-H if every other NP problem is polynomially reducible to V
    - NOTE: not every NP-H problem is in NP

- NP-C is the intersection of NP and NP-H
  - Every problem in NP-C is about as hard as every other problem in NP-C
  - You can think of NP-C as the hardest problems in NP
  - NP-C problems are 'universal' → if we had an algorithm that can solve any NP-C problem V, then we can solve every other NP problem U by reduction
    1. Compute the reduction f(x) of U to V in polynomial time
    2. Run the algorithm that solves U on instance f(x)
  - A polynomial time algorithm for any NP-C problem would imply that P = NP

- If we assume P = NP
  - The intersection of NP-H and P is NP-C
    - If P = NP, we can use them interchangeably in the definition of NP-C
  - We can solve every problem in NP-C in polynomial time
    - Every problem in NP-C belongs to NP by definition, so if P = NP, every problem in NP-C belongs to P, and can thus be solved in polynomial time
  - Every problem in NP also belongs to NP-H
    - If P = NP, then every problem in NP is as hard as any other

Reductions

- Solving by reduction is when we map (reduce) a new problem to an old problem we know how to solve → this gives an algorithm for a new problem → the new algorithm involves applying the mapping function, then applying the known algorithm for the old problem
- NOTE: reductions do not need to be surjective → they may map all instances of U to some specific instances of V → but an algo for V solves all instances anyways
- Let U and V be two decision problems → U is polynomially reducible to V if and only if there exists a function $f(x)$ such that:
  1. $f(x)$ maps instances of U to instances of V
  2. $f(x)$ maps YES instances of U to YES instances of V, and NO instances of U to NO instances of V → $U(x)$ is YES if and only if $V\left(f(x)\right)$ is YES
  3. $f(x)$ is computable by a polynomial time algorithm
- If there is a polynomial reduction from U to V, we can conclude that U is no harder than V
- If we can solve V in polynomial time, then we can also solve U in polynomial time, and if we can't solve V in polynomial time, then we can't solve U in polynomial time

Consequences of Reductions (assuming P $\neq$ NP)

- Suppose U and V are decision problems in class NP, and that there exiwsts a polynomial reduction f from U to V
    - If V is in class P also, then U is in class P
    - If U is in class NP-C also, then V is in class NP-C
    - Anything else cannot give us any deductions

NOTE: Every instance of SAT is polynomially reducible to an instance of 3SAT

- This can be proven by introducing more propositional variables and replacing every clause by a conjunction of several clauses
- NO CLUE HOW THIS WORKS

Cook's Theorem: Every NP problem is polynomially reducible to the SAT problem

- For every NP decision problem U, there is a polynomial time computable function f such that:
    1. For every instance x of U, f(x) is a valid propositional formula for the SAT problem
    2. U(x) is true if and only if the formula f(x) is satisfiable
- SAT is in NP and NP-H (following Cook's Theorem), and so it is in NP-C
- Since NP problems map to SAT which maps to 3SAT, all NP problems map to 3SAT
- DO NOT NEED TO KNOW HOW TO PROVE IT

NP-C problems

- Travelling Salesman Problem → is there a path that visits all vertices of a weighted directed graph exactly once with total weight at most L
- Register allocation → can we colour vertices of G with at most K colours so that no edge has both vertices of the same colour
- Vertex Cover → can we choose k vertices so that every edge in an undirected unweighted graph is incident to at least one of the chosen vertices
    - NOTE: 3SAT can be reduced to VC
- Set Cover → Given m bundles that contain a subset of n items, is is possible to choose k bundles which together contain all n items

Proving NP-Completeness

- Theorem: Let V be an NP-C problem, and let W be another NP problem. If V is polynomially reducible to W, then W is also NP-C

- Proof Outline:
  - Let g(x) be a polynomial reduction of V to W, and let U be any other NP problem
  - Since V is NP-C, there is a polynomial reduction f(x) of U to V
  - Then $(g \circ f)(x)$ is a polynomial reduction of V to W
- Proof:
  1. Claim that $(g \circ f)(x)$ is a reduction of U to W
  2. Since f is a reduction of U to V, U(x) is true iff V(f(x)) is true
  3. Since g is a reduction of V to W, V(f(x)) is true iff W(g(f(x))) is true
  4. Thus, U(x) is true iff W(g(f(x))) is true → if $(g \circ f)(x)$ is a reduction of U to W
  5. Since f(x) is the output of a polynomial time computable function, the length $|f(x)|$ of the output f(x) must be at most a polynomial in |x| → for some polynomial (with positive coefficients) P, we have $|f(x)| \leq P(|x|)$
  6. Since g(y) is polynomial time computable as well, there exists a polynomial Q such that, for every input y, computation of g(y) terminates after at most $Q(|y|)$ steps
  7. Thus, the computation of $(g \circ f)(x)$ terminates in at most
     a. P(|x|) many steps for the computation of f(x), PLUS
     b. $Q(|f(x)|\ ) \leq Q(P|x|)$ many steps for the computation of $g(y)$, where y = f(x)
  8. In total the computation of $(g \circ f)(x)$ terminates in at most $P(|x|) + Q\big(P(|x|)\big)$ many steps, which is polynomial in $|x|$
  9. Therefore $(g \circ f)(x)$ is a polynomial reduction of U to W
  10. But U could be any NP problem, and so we have now proven that any NP problem is polynomially reducible to the NP problem W → W is NP-C

Optimisation Problems

Linear Programming → the restrictions (constraints) and costs (objective function) deal only with linear functions of the variables

- There are polynomial time algorithms for linear programming
- The SIMPLEX algo is worst case exponential, but has good average case performance
- The objective function is the 'real-value' function that is to be minimised or maximised by using the constraints
  - Typically takes the form $ax_1 + bx_2$, where x and y are the decision variables we pick, and a and b are any constants
    1. Graph all constraints as inequalities
    2. Find their intersection → the feasible region

3. Set the objective function to equal 0, rearrange for x_2, then graph the line, and the area above is where the objective function will be positive
4. Find the point maximising or minimising the objective function in the feasible region → intuitively, you should check corners or edges depending on the objective function

- Variables are typically assumed to be non-negative
    - ≥ constraints can be negated to make ≤ constraints → e.g.
    $$4x_T + x_O \geq 7 \;\rightarrow\; -4x_T - x_O \leq\; -7$$
    - Equality constraints can be represented by a pair of inequality constraints → e.g.
    $$x_L + x_T + x_O = 3 \rightarrow \begin{cases} \;\; x_L + x_T + x_O \leq 3 \\ -x_L - x_T - x_O \leq -3 \end{cases}$$
    - This allows minimisation problems to be converted to maximisation problems by negating the objective function → e.g.
    $$\min[x_L + 0.8x_T + 0.6x_O] \rightarrow \max[-x_L - 0.8x_T - 0.6x_O]$$

We want to be able to write problems as linear programs so that we can use LP Solvers as black box.

- To do this we need to answer the following:
    - What variables are we trying to optimise
    - Are these variables restricted to integers, or can they be any non-negative real number
    - What (linear) function of the variables are we trying to optimise
    - Do we want to maximise or minimise this function
    - What are the constraints on the variables, or combinations of variables
- After specifying answers to these questions, an LP solver can produce an optimal solution under these constraints