Introduction to Algorithms

Aryaman Arora

Contents

1 Algorithm Analysis

1.1 Asymptotic notation

For the upper bound, we have $O(g(n)$. $f(n) = O(g(n))$ means that:

$$\exists\, c > 0, n_0 > 0$$
$$\forall\, n \geq n_0, 0 \leq f(n) \leq cg(n)$$

When we use $O(n)$, we tend to prefer the tightest upper bound possible. Keep in mind that exponentials (higher base better) dominate polynomials dominate logarithms.

For lower bounds, we have $\Omega(g(n))$, with inverted rules. And when the same function $g(n)$ is both the lower and upper bound, we can say that $f(n) = \Theta(g(n))$.

2 Graphs

2.1 Properties

Definition 2.1. A *directed graph* is formally $G = (V, E)$, where V is a set of vertices and E is a set of edges represented by ordered pairs of vertices (u, v), i.e. an edge $u \to v$. An *undirected graph* is only different in that the edges are unordered pairs $\{u, v\}$, i.e. $u \leftrightarrow v$.

The number of edges is such that $|E| < |V|^2$, implying $\log E = O(\log V)$ (we don't need to write the cardinality bars inside big-O).

Definition 2.2. A *connected graph* is such that $|E| \geq |V| - 1$ (i.e. every vertex can be reached from any every other text).

In a connected graph, $\log E = \Theta(\log V)$ so in big-O we can reduce e.g. $O(E + V)$ to $O(E)$.

2.2 Representation

An *adjacency matrix* stores edges as a matrix A such that:

$$A_{ij} = \begin{cases} 1 & (i, j) \in E \\ 0 & (i, j) \notin E \end{cases}$$

This takes $O(V^2)$ space to store and time to find edges, and $O(1)$ time to check if an edge exists.

An *adjacency list* stores edges as an array of lists, such that A_i is the list of nodes outgoing from i. This takes $O(V + E)$ space to store and time to find edges, $O(|A_i|)$ time to check if an edge exists.

2.3 Search

When we traverse the graph, we keep track of $v.d$ (the distance of node v) and $v.\pi$ (the predecessory). We also track $v.color$ for internal use; a white vertex is unvisited, a grey one is being processed, and a black one is already visited (so we don't visit it again).

2.3.1 Bread-first search

Explore in order of distance.

Set all distances to ∞. Process in groups that we call *frontiers*. We start at some grey node s that is in the current frontier, add all of its outgoing white nodes (which we turn grey and set the properties of) to the next frontier, and so on. We replace the current frontier with the new one once it is empty. We can also use a queue instead of two lists.

In an unweighted graph, BFS finds shortest paths to nodes, which is a chain that can be required from checking $v.\pi$.

Runtime is $O(V + E)$, each vertex is reached only once and every edge is checked once or twice (undirected).

Proof. By induction. The base case is that we assume vertices with distance $\leq i$ are labelled correctly and those with $v.d > i$ have not been discovered. Every shortest path of length $i + 1$ necessarily is made up of a path of length i plus the end node. Since we assume all of length i are in the current frontier, then this shortest path has to be discovered because there is an outgoing node from the current frontier. □

2.3.2 Depth-first search

Where BFS uses a queue, DFS uses a stack. We explore the graph maximally at a time, and track $v.d$ (start time) and $v.f$ (end time).

Algorithm 1: DFS_VISIT(G, u)

```
u.d ← time;
time ← time + 1;
u.color ← grey;
for v ∈ G.Adj_u do
    if v.color = white then
        v.π ← u;
        DFS_VISIT(v);
u.color ← black;
u.f ← time;
time ← time + 1;
```

Algorithm 2: DFS(G)

```
for u ∈ G.V do
    if u.color = white then
        DFS_VISIT(u);
```

Note that all the start–end times are nested ($u.d < v.d < v.f < u.f$ means v is reachable from u) or disjoint (there is no path between the two). The parenthetical properties only matter in directed graphs.

A *back edge* is $u \to v$ such that $v.d < u.d < u.f < v.d$ (i.e. it goes to an ancestor that is on the call stack). A back edge is part of a cycle (prove w/ start–end nesting). If no back edges are found, the graph is acyclic.

Runtime is $\Theta(V + E)$, same as BFS. Note that $\Theta(E)$ element is found by the summation $\sum_u \Theta(1 + out(u))$, which is the inner loop in the algorithm (checking edges).

2.3.3 Topological sort

On a directed acyclic graph (DAG), a *topological sort* is an ordering of the vertices such that $\forall\ (u, v) \in E$, u is before v in the ordering. E.g. for a schedule of classes such that each class has requirements, a topological sort will find an order in which the classes can be taken.

Two simple algorithms for this, both in $\Theta(V + E)$.

Algorithm 3: PEEL(G)

ready $\leftarrow \{\}$;
maintain count of indegree of each vertex u;
for $i \in \{0, 1, \ldots, V\}$ **do**
    choose vertex u with no incoming edges;
    remove it from graph;
    remove all edges (u, v) and add v to *ready* if its indegree is 0;

Algorithm 4: DFS_TOPO(G)

DFS(G);
output vertices $u \in V$ in reverse order of $u.f$ (modify DFS to push finished vertices to list);

Proof. Let's say there is some edge (u, v) such that u is after v in the topological ordering. In the first algorithm, this is impossible because that would mean v was removed from the graph before u, but it has an incoming edge from u so it can't be removed until u is.

In the second algorithm, this would mean u has to have a finish time after v (since it's earlier in the call stack) so we couldn't have been before u in the ordering. If it is, then that's a back edge, so the graph won't be a DAG in the first place. □

2.3.4 Strongly connected components

Definition 2.3. A *strongly connected component* of a graph is $C \subseteq V$ such that for all $u, v \in C$ there is a path $u \rightsquigarrow v$ and $v \rightsquigarrow u$, and there is no vertex outside of C for which this property holds with a vertex in C.

The SCCs of the graph form a DAG, which is called the *condensation* of the graph. After all, if there was a cycle of SCCs, that cycle would form an SCC (since every vertex is able to be visited from every other vertex in a cycle).

Algorithm 5: SCC(G)

DFS(G);
build G^T;
call DFS(G^T) in reverse order of finish times, each tree created is an SCC;

Proof. Todo □

3 Amortised analysis

We want to come up with tighter bounds for the worst case cost of some operation(s). For example, an array that doubles size when it is full: what is the cost of n inserts? $O(n^2)$?

3.1 Aggregate analysis

We do know the exact cost of the i-th insert:

$$c_i = \begin{cases} 2i & i \text{ is a power of 2} \\ 1 & \end{cases}$$

Thus, the exact cost of n inserts is $\sum_{i=1}^{n} c_i$. The costs for the doubling when we do n inserts is at most $2n + n + n/2 + \ldots + 1 \leq 4n$. The costs for the normal inserts is $< n$. So, we can deposit a cost of 5 for every insertion and so we can say the runtime of this operation is $\Theta(5n)$.

3.2 Accounting method

This is useful for a sequence of many kinds of operations. We assign some costs to each operation that deposite some value 'in the bank' which we can draw on later.

For example, we can analyse a stack with operations push(), pop(), and multipop() [which removes all elements on the stack]. If we assign an amortised cost of 2 to push(), then we can say the amortised costs of pop() and multipop() are 0 since we prepay for the popping of each element we insert.

3.3 Potential method

This is just a complicated version of the latter that is probably impractical to use. Think in recurrences, I guess (consider rate of growth of the total cost, see if you can put a constant bound on it).

The idea is to make a state function $\Phi(S)$ and then calculate the change in cost during a change of state. The state function should 'absorb' the cost.

4 Data Structures

4.1 Disjoint set

The fundamental ideas for a *disjoint set* are the union operation, which combines two sets, and the find operation, which checks what set an item is in. The specifics of how these are implemented is the only interesting part. Basically, we want a disjoint set to be faster than DFS for connected components in an undirected graph is.

One idea is a doubly-linked list, where union is $O(1)$ but find is $O(n)$ where n is the size of the list.

A better idea is **pointers from each element to the set ID**, which brings find down to $O(1)$, but the runtime for union is more complicated, how do we make it less than $O(n)$? If we do union of the small list into the bigger list every time, then it is $\Theta(n)$ in the worst case.

We can calculate an $O(n \log n)$ amortised cost for this! The basic idea by the aggregate method is that an element that undergoes k changes in the set it is in must be in a set with at least 2^k elements. This is because if it undergoes a change, then it has been been merged into a set with more elements than the set which it was in, i.e. the set size has at least doubled. Prove by induction.

Another idea is **maintaining a forest of sets**, using a heuristic (like ranking) to keep the height of the tree low, or to be precise $O(\log n)$. Same runtime we get. The runtime for finding undirected graph components then is $O(E + V \log V)$.

The final improvement is if we do **path compression** when finding set (i.e. we move the element we queried up to the first level of the set tree, which isn't a costly operation but saves a lot of time). That reduces everything to amortised $O(\alpha(n))$ which is near-linear.

4.2 Runtimes

The runtimes you need to know follow.

Disjoint Set	MAKESET	UNION	FIND	
lists w/ union by weight	$O(\log n)$	$O(\log n)$	$O(1)$	amortised
up trees w/ union by rank	$O(1)$	$O(\log n)$	$O(\log n)$	worst case
same with path compression	$O(\alpha(n))$	$O(\alpha(n))$	$O(\alpha(n))$	amortised

Priority Queue	INSERT	EXTRACTMIN	DECREASEKEY	
binary (min) heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	worst case
balanced BST (AVL)	$O(\log n)$	$O(\log n)$	$O(\log n)$	worst case
Fibonacci heap	$O(1)$	$O(\log n)$	$O(1)$	amortised

Searchable Dynamic Sets	ANY	
BBST (AVL, red-black)	$O(\log n)$	worst case
splay trees	$O(\log n)$	amortised
B-Trees (w/ constant fanout)	$O(\log n)$	worst case

For maps, a hash table is $O(1)$ expected time for all operations.

4.3 Minimum Spanning Tree

Definition 4.1. A *spanning tree* is a subset of edges in a connected graph that fully spans all of the vertices. A *minimum spanning tree* does so with the minimum possible total edge weight. A *cut* is some partition of the graph $(S, V \setminus S)$.

An edge crosses the cut if $u \in S$ and $v \notin S$. A light edge is the cut with the minimum possible weight. A cut respects a partial MST A if no edges in A cross the cut.

Proof. Todo. □

4.3.1 Kruskal's algorithm

Sort the edges, add the minimum weight edge available if it is a cut relative to the subset we have built up in our partial MST (i.e. light edges only). Use disjoint set data structure to maintain set membership.

The sort takes $O(E \log E)$, and we further run $O(2E)$ finds, $O(V)$ unions, and $O(V)$ makesets. Runtime is dominated by the sort, so with path compression it's $O(E\alpha(V) + E \log E) = O(E \log V)$.

4.3.2 Prim's algorithm

Continually add the lightest edge to a node on the frontier of the subgraph we have. Maintain the lightest edge weight connecting unprocessed vertices to processed vertices in a set (i.e. Fibonacci heap).

We run $O(V)$ inserts, $O(E)$ decrease keys, and $O(V)$ extract mins. That gives a total runtime of $O(V + E + V \log V) = O(E + V \log V)$ with Fibonacci heap.

4.4 Shortest paths

Consider a directed graph $G(V, E)$ with edge weights $w(u, v)$. A path is a list of edges connecting adjacent nodes: $p = x_1 \to x_2 \to \ldots \to x_n$. The weight of a path is $w(p) = \sum_{(u,v)\in p} w(u, v)$. A path is a *shortest path* if no other path from its start to its end exists with lower $w(p)$.

The *shortest path distance*, $\delta(u, v)$, is the weight of the shortest path from u to v. If there is no path from u to v, then $\delta(u, v) = \infty$. If there is a negative cycle we can leverage in the graph, then $\delta(u, v) = -\infty$.

One property of a shortest path is *optimal substructure*: a shortest path is comprised of shortest paths. This also gives us the *triangle inequality*: $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$ for all $x \in V$.

> **Proof.** If there is a shortest path $s \rightsquigarrow t$ that includes a path $u \rightsquigarrow v$ that isn't the shortest, we can simple replace that subpath with whatever path is the shortest (since we can break the whole path down to $s \rightsquigarrow u \rightsquigarrow v \rightsquigarrow t$, changing a portion still maintains the path) and reduce the weight of the whole path. So, the original could not have been the shortest path.
>
> For the triangle inequality, the proof is that the shortest path is shorter than all other valid paths by definition, so it has to be less than the diversion we pick with x. If x is on the shortest path though, then it's equal. □

For now, we're interested in single-source shortest paths, for fixed s to all other nodes in the graph. The tool we use is *relaxations*; we start with conservative overestimates of $\delta(s, v)$ and gradually relax the distance as we process edges. Then we just have to decide what order and how many times to do this. The proof that this is okay can be shown through induction; note that a successful relaxation always gives us a valid path $s \rightsquigarrow u \to v$.

Algorithm 6: RELAX((u, v))

```
if v.d > u.d + w(u, v) then
    v.d ← u.d + w(u, v);
    v.π ← u;
```

4.4.1 Bellman–Ford algorithm

Algorithm 7: BELLMANFORD(G, s)

```
for v ∈ V do
    v.d ← ∞;
    v.π ← ∅;
s.d ← 0;
for i ∈ {1, ..., |V| − 1} do
    for (u, v) ∈ E do
        RELAX((u, v));
```

If we can relax any more at the end of this, then that means there is a negative weight cycle. At minimum, at the end of the i-th iteration of Bellman–Ford, we will have considered all paths of $\leq i$ edges. The longest possible path for a graph of $|V|$ nodes includes $\leq |V| - 1$ edges.

The runtime is $\Theta(VE)$.

4.4.2 Dijkstra's algorithm

This time we use the same idea as Prim's, examining vertices at the frontier of the subgraph we've looked at already.

Algorithm 8: DIJKSTRA(G, s)

for $v \in V$ **do**
  $v.d \leftarrow \infty$;
  $v.\pi \leftarrow \varnothing$;
Q is a priority queue with all vertices except s set to ∞;
$S \leftarrow \varnothing$;
while Q **do**
  $u \leftarrow Q$.EXTRACTMIN();
  $S = S \cup \{u\}$;
  for $v \in G.A_u$ **do**
    RELAX(u, v);
    Q.DECREASEKEY$(v, v.d)$;

The runtime is $O(E + V \log V)$ with a Fibonacci heap, better than Bellman–Ford.

Proof. Let us assume we reach a vertex u and do not find the shortest path to it. That means Dijkstra missed some vertices on the shortest path to u before reaching it. The first such edge is (x, y).

$$\begin{aligned} \delta(s,u) &< u.d \\ \delta(s,x) + w(x,y) &\leq \delta(s,u) \\ x.d + w(x,y) &\leq \delta(s,u) \\ y.d &\leq x.d + w(x,y) \\ u.d &\leq y.d \end{aligned}$$

There'd have to be a negative edge/cycle. □

4.5 Max flow

Definition 4.2. A **flow network** is a directed graph, with edge capacities $0c(u, v) \geq 0$, distinguished source (s) and sink (t) nodes. We assume for every $v \in V$ there is a path $s \rightsquigarrow v \rightsquigarrow t$ and no antiparallel edges.

Definition 4.3. A **flow** is a function f satisfying the following two constraints:

- Capacity constraint: $0 \leq f(u, v) \leq c(u, v)\ \forall\ u, v$
- Flow conservation: $\forall\ u \in V | \{s, t\} : \sum_{x \in V} f(x, u) = \sum_{y \in V} f(u, y)$ (what goes in must come out)

The **value** of a flow is the net flow out of s: $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$ (usually that second term is 0).

In the **max flow problem** we want to maximise the total flow from a source s to a sink t along a network of edges with some capacities. Our restrictions are that

each flow can only travel one way along an edge, the incoming flow to a node must equal its outgoing flow, and the flow along an edge is no greater than its capacity.

The Edmonds–Karp algorithm handles this by repeatedly BFSing from the source, finding the maximum flow one can send along that path (i.e. the minimum capacity of an edge along it), and reducing that capacity from the graph. It also maintains negative capacities for reversed edges, which we can leverage to redirect flows. The runtime is $O(VE^2)$, because the max path length is V and there are $O(VE)$ possible critical events (filling of capacities of some edge). BFS is $O(E+V) = O(E)$ since this is a connected graph, so we multiply and get $O(VE^2)$.

5 Divide and conquer

Divide and conquer is an algorithmic problem-solving paradigm. There are three parts of any D&C algorithm.

- *divide* a problem into smaller subproblems (constant fraction smaller)
- *conquer* problems, solving recursively (induction step)
- *combine* subproblem solutions to solve the whole problem

Some examples of common D&C approaches are binary search and merge sort.

5.1 Master theorem

Beisdes induction, guessing and checking, etc. the best tool at our disposal for solving recurrences (e.g. to calculate the runtime of a recursive algorithm) is the Master Theorem. This is derived from using the expanded recursion tree of a recurrence.

Given a recurrence of the form, such that $a \geq 1$, $b > 1$ are constants and $f(n)$ is an asymptotically positive function:

$$T(n) = aT(n/b) + f(n)$$

The master method gives us three cases for solving this.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for a constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for a constant $\epsilon > 0$, then $T(n) = \Theta(f(n))$.[1]

[1] Also, $af(n/b) \leq cf(n)$ for some constant $c < 1$ asymptotically.

5.2 Some algorithms

5.2.1 Merge sort

- *divide* array in half
- *conquer* by sorting each half

- *combine* by merging the two sorted subarrays

Algorithm 9: MERGESORT(A, p, r)

$q \leftarrow \lfloor (p+r)/2 \rfloor$;
$a \leftarrow$ MERGESORT(A, p, q);
$b \leftarrow$ MERGESORT$(A, q+1, r)$;
return MERGE(a, b);

The recurrence for the runtime of this recursive algorithm is $T(n) = \Theta(n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) \approx 2T(n/2) + \Theta(n)$ (cost of merging two sorted arrays with total length n is always $\Theta(n)$).

By case 2 of the Master Theorem, the overall runtime of this algorithm is $\Theta(n \log n)$.

This algorithm takes a lot of extra space but it has a very reliable runtime.

5.2.2 Quicksort

- *divide* array around some element x (called the pivot) and reorder elements such that to the left is $\leq x$ and right is $> x$
- *conquer* by sorting each half recursively
- *combine* by doing nothing

Algorithm 10: QUICKSORT(A, p, r)

if $p < r$ **then**
    $q \leftarrow$ PARTITION(A, p, r);
    QUICKSORT$(A, p, q-1)$;
    QUICKSORT$(A, q+1, r)$;

The best parts about this algorithm are that it takes no extra space (sorts in-place) and has little constant factor overhead.

Worst-case runtime. As for the runtime recurrence, it's a little harder to calculate. Let's try for the worst-case runtime. We know PARTITION is always $\Theta(n)$. As for the size of each recursive step, if we select the kth largest element as the pivot, then we recurse to sort subarrays of size $k-1$ and $n-k$. So, $T(n) = T(k-1) + T(n-k) + \Theta(n)$.

We don't know k because it's random. In the worst case though, we know the pivot is the smallest/largest element, so $k = 1$ or $k = n$. We can simply solve the recurrence without needing any theorem.

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ \Theta(f(n)) &= \Theta(n) + \Theta(n-1) + \ldots + \Theta(1) \\ &= \Theta\left(\frac{n(n-1)}{2}\right) \\ &= \Theta(n^2) \end{aligned}$$

Best-case runtime. In the best case, the element is exactly in the middle in terms of rank, so we get the exact same recurrence as merge sort and have $\Theta(n \log n)$.

Average-case runtime. We have to learn some discrete probability to calculate the average (=expected) runtime. A random variable X has some number of outcomes with value x and an associated probability. The expectation of a *discrete* random variable is simply $E[X] = \sum Pr(X = x) \cdot x$. Also, they have a property called **linearity of expectation**: $E[X + Y] = E[X] + E[Y]$.

So, we frame the problem as counting the average number of comparisons. Each pair of elements can only be compared once at most. We define an indicator random variable, X_{ij} which is 1 if i and j are compared and 0 if they are not. Thus we have $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$, and we can use linearity of expectation to simplify to:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

To solve this, we need to calculate the probability that i and j are compared. They will not be compared if a pivot is selected at some point that is between them in value; they'll get separated to different subproblems then. So, $Pr(X_{ij} = 1) = 2/(j - i + 1)$. We can plug this in.

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} \\ &\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \\ &\leq 2 \sum_{i=1}^{n} \Theta(\log n) \\ &= O(n \log n) \end{aligned}$$

5.2.3 Order statistic

The order statistic problem is the issue of finding the ith largest element in an unsorted array. We can use the quicksort method to do this, except we only recurse on the side with the ith largest element at each step, for an average runtime of $\Theta(n)$ (worst-case $\Theta(n^2)$).

5.2.4 Matrix multiplication

The basic idea is:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

The naive algorithm is $\Theta(n^3)$, but can we find something better?

One useful property is that we can recursively treat a matrix as a 2×2 matrix of submatrices. This takes 8 (recursive) multiplications and $\Theta(n^2)$ additions,

giving us:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Using the Master theorem shows that this is still $\Theta(n^3)$. Can we somehow reduce the 8 recursive multiplications? Strassen's method (see textbook) lets us push it down to just 7 multiplications, which gives us a runtime of $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$.

5.2.5 Closest pair of points

Naive algorithm compares every pair, so $\Theta(n^2)$. We split along the median x coordinate, recursively find the closest pair in each half, then given the closest pair we've found so far has distance d, we check all point crossings with x-coordinates between $x - d$ and $x + d$. This comes to $\Theta(n \log^2 n)$ or even just $\Theta(n \log n)$ with pre-sorting.

6 Dynamic programming

Dynamic programming is a technique for solving problems that are made up of overlapping subproblems. Besides that, there needs to be *optimal substructure*: an optimal solution to a problem contains optimal solutions to subproblems.

6.1 Some algorithms

6.1.1 Fibonacci

$F_k = F_{k-1} + F_{k-2}$. We memoise and convert to an iterative algorithm that uses the previously calculated values for a runtime of $\Theta(n)$ to calculate the nth Fibonacci number.

6.1.2 Subset sums

Is there a subset of an array of numbers with the sum we are looking for? Go through each prefix of the array and build up the solution; when we are at the ith element we add its value to every possible sum we know we can get from before. Runtime is $\Theta(nk)$ where k is the sum we are looking for.

6.1.3 Rod cutting

You have some set of k rods with length l_i and price p_i. You have one big rod of length n that you want to cut up and sell for maximum profit. The way to do this is iterate on lengths $1 \ldots n$ and check every rod, maximising total profit calculated as $f(n) = \max f(n - l_i) + p_i$. Runtime is $\Theta(nk)$.

6.1.4 Matrix multiplication chain

The optimal parenthesisation of a chain of matrix multiplications (cost is $\Theta(pqr)$ to multiply $p \times q$ matrix by $p \times r$ matrix) can be found through dynamic programming. Use a 2d array, recursively check every way to cut into pieces, sort of like merge sort but with every possible partition checked. Runtime is $\Theta(n^3)$.

7 A lower bound on sorting algorithms

Using the decision tree analysis and applying it to the process of comparisons in a sorting algorithm, we can find a lower bound on runtime for sorting algorithms (really, number of comparisons) to ensure the algorithms we've come up with are actually optimal.

Basically, a sorting algorithm should be able to map any possible set of permutations of length n to the permutation $1, 2, \ldots, n$. Any permutation is possible as an input. An execution of the algorithm can be analysed as a path from the root to the (sorted) leaf.

There are $n!$ permutations of a sequence of length n. So, our decision tree has $n!$ leaves. We can solve for minimum tree height to get a lower bound on a sorting algorithm: $n! \leq 2^h$. This gives us a runtime of $\Omega(n \log n)$, which is indeed how fast our best algorithms are.

> **Remark.** *Is it possible to construct a comparison-based data structure that allows insertion of n elements in $O(1)$ amortised time and extract-min in $O(1)$ amortised time?* No. If such a data structure existed than it would also allow sorting in $\Theta(n)$ time, which is not possible solely through comparisons as we found above.
> To sort in linear time relative to the size of the largest number in the array, though, we can use counting sort.

8 Computational tractability

> **Definition 8.1.** P = problems with polytime algorithms, e.g. $O(n^k)$ for every constant k.

We use polynomial time as a standard because it works across different models of computation (including RAM as in this class).

> **Definition 8.2.** NP = problems whose candidate solutions can be verified in polytime.

An example of a problem that is in NP but not in P is the **Hamiltonian cycle problem**: given a graph, is there a simply cycle that hits every vertex? Testing a cycle is easy, just check if the candidate edges are in the graph.[2] But finding a solution is not.

[2] Just $O(V)$ with adjacency matrix!

It's obvious that $P \subseteq NP$ but not whether $P = NP$ (which is a huge unsolved problem).

> **Definition 8.3.** An NP-complete problem is as hard as any other problem in NP. Finding a polytime solution for one would prove that $P = NP$.

8.1 Formalisation

We need some formal definitions of algorithmic problems to actually make sense of what this P/NP business is.

Definition 8.4. An **optimisation problem** has multiple valid solutions with some value, and we want to return the best one (e.g. shortest paths). A **decision problem** has a yes/no answer (e.g. is there a path from node s to node t with weight $\leq k$?).

P and NP are defined with respect to decision problems, because these are easier and more lax in their solution boundaries (a decision problem will have a greater superset of the solutions to an optimisation problem on the same idea).

Definition 8.5. A **problem instance** is a mathematical input to a decision problem. We use this kind of notation:

$$\textsc{Path}(\langle G, s, t, k\rangle) = \begin{cases} 1 & \text{if path exists} \\ 0 & \text{else} \end{cases}$$

Let n be the length of the binary encoding of input x. Problem Q is polytime computable if there exists an algorithm A and constant c such that, for any input x with length n, A terminates in $O(n^c)$ time and $A(x) = Q(x)$ (it gives the correct answer). The set of polytime computable problems is P.

For NP, it is similar. We want a verification algorithm A that correctly assesses the validity of a certification y for problem instance x, that too with the same time constraint as above. Such a problem is polytime verifiable.

8.2 Polytime reducibility

A problem Q_1 is polytime reducible to problem Q_2 (denoted as $Q_1 \leq_p Q_2$) if there exists a polynomial algorithm/function such that $Q_1(x) = Q_2(f(x))$.

The Hamiltonian cycle problem $\leq_p$ travelling salesman problem.

Proof. Given the input $\langle G\rangle$ for the cycle problem, construct $\langle G', c, k\rangle$ problem instance for TSP with same yes/no answer.

- $G' = (V, E')$ where E' is all possible edges.
- $c(u,v) = \begin{cases} 0 & \text{if } (u,v) \in E \\ 1 & \text{else} \end{cases}$
- $k = 0$

We can construct the correct answer with this. □

8.3 NP-hardness

Definition 8.6. A problem Q is NP-hard if $\forall$ problems $Q' \in NP$, it is true that $Q' \leq_p Q$.
Furthermore, a problem is NP-complete if $Q \in NP$ and it is NP-hard.

The strategy to prove a problem is NP-complete is:

1. Prove $Q \in NP$ by providing a fast verification algorithm
2. Prove that Q is NP-hard by coming up with way to reduce a known NP-complete problem to it

3-SAT problem is useful (find a way to make a conjunction of 3-clause disjunctions be true). As long as some instance of a problem is mappable to 3-SAT then it's intractable (NP-hard).[3]

[3] See this on Stack Overflow.

8.3.1 Example: Clique problem

This is a decision problem, CLIQUE(G, k), does the graph G have a clique (fully-connected component) of k vertices?

Theorem 8.1. CLIQUE is NP-complete.

Proof. First, we need to prove that CLIQUE $\in NP$. Given a certificate of k vertices, we can check in $\Theta(|k|^2)$ (which is polytime) whether it is fully-connected by checking if every pair has an edge between it. So, this problem is in NP.

Second, we need to show that it is NP-hard by coming up with a way to reduce a known NP-complete problem to it in polytime. We will use 3-SAT; we'll transform an instance of the 3-SAT problem into the k-CLIQUE problem.

For the r-th clause, we make three vertices corresponding to each boolean variable in it. We construct an edge between two vertices (v_i^r, v_j^s) if $r \neq s$ (they are in different clauses) and $l_i^r \neq \neg l_j^s$ (their corresponding boolean values can be true at the same time without contradiction). □

Lemma 8.2. Now we claim that the 3-SAT problem instance ϕ is satisfiable iff there is a k-size clique in this graph G.

Proof. We need to prove in both directions.

- $\phi \implies$ CLIQUE: If ϕ is true, then in our satisfying assignment there must be at least one literal per clause that is true. Choose any one in each clause that is true. The corresponding set of vertices is a clique, because by our original construction, we only constructed edges if both of the literals can be true and here we've chosen a whole set that can be simultaneously true (so they all must connect to all of each other).
- CLIQUE $\implies \phi$: Pick any size-k clique from the graph G. Set the corresponding literals to be true. This is not contradictory by our condition on simultaneous truthiness in the construction. Also, each vertex in the clique has to be in a different clause since we don't allow clause-internal edges. So, we end up with at least one true value per clause, which satisfies ϕ.

Our reduction ran in polytime too, thus CLIQUE is NP-hard and with the first proof we know it is NP-complete too. □