CS 218: Design and Analysis of Algorithms

Amit Rajaraman

Last updated April 19, 2021

Contents

§1. Different Types of Algorithms

1.1. Asymptotic Notation

1.1.1. Big-$\mathcal{O}$ Notation

Consider the following basic algorithm we use to check primality (checking if any $2 \leq i \leq \sqrt{n}$ divides n):

Algorithm 1: Algorithm to check if a number is prime

Input: A non-negative integer n
Output: If n is prime, output true else false

```
flag ← true
for i = 2 to √n do
    if i divides n then
        flag ← false
return flag
```

Is the above a polynomial time algorithm?
No! It is polynomial in n, but *not* polynomial in the input size $\log n$.[1]
While analyzing algorithms in general, it is important to look at the input size, the number of bits that constitute the input.

Definition 1.1 (Big-$\mathcal{O}$ notation)**.** $T(n)$ is said to be $\mathcal{O}(f(n))$ if there is some $c > 0$ and $N \in \mathbb{N}$ such that for all $n > N$, $T(n) \leq cf(n)$.

This notation is often abused to say, for example, that $\sqrt{n} = \mathcal{O}(n)$ (instead of the correct $\sqrt{n} \in \mathcal{O}(n)$).

Definition 1.2 (Big-Ω notation)**.** $T(n)$ is said to be $\Omega(f(n))$ if there is some $c > 0$ and $N \in \mathbb{N}$ such that for all $n > N$, $T(n) \geq cf(n)$.

Finally, we have

Definition 1.3 (Θ notation)**.** $T(n)$ is said to be $\Theta(f(n))$ if there are some $c_1, c_2 > 0$ and $N \in \mathbb{N}$ such that for all $n > N$, $c_1 f(n) \leq T(n) \leq c_2 f(n)$.

Equivalently, $T(n) \in \Theta(f(n))$ if and only if $T(n) \in \mathcal{O}(f(n))$ and $T(n) \in \Omega(f(n))$.

1.1.2. A Few Examples

We give a few examples with the aim of hopefully making the above notation more clear.

1. $\mathcal{O}(n)$. Given an array A containing n integers (0-indexed), find the value of the maximum element in the array. Consider Algorithm 2 that takes $\mathcal{O}(n)$ time.

   The bits used to represent each $\mathsf{A}[i]$ is $\log m$. We are assuming that comparison takes constant time. Technically the following algorithm is $\mathcal{O}(n \log m)$, but we often blur the details slightly. To be completely correct, we should say that the algorithm performs $\mathcal{O}(n)$ *comparisons*.

2. $\mathcal{O}(n \log n)$. Given an array A containing n elements (0-indexed), sort the array.
   The merge sort algorithm performs $\mathcal{O}(n \log n)$ comparisons. We do not explicitly write out the algorithm.

3. $\mathcal{O}(n^2)$. Given n points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ in the plane, output a pair i, j such that the distance between p_i and p_j is minimum.

   The algorithm is described in Algorithm 3.

   This problem can in fact be solved in $\mathcal{O}(n \log n)$ time, as we shall see in Section 1.3.2.

[1] A polynomial time algorithm was described in [AKS02].

Algorithm 2: Algorithm to find the maximum element in an array

Input: An array A containing n integers
Output: The maximum element in A

```
max ← A[0]
for i = 1 to n − 1 do
    if A[i] > max then
        max ← A[i]
return max
```

Algorithm 3: Algorithm to find a closest pair of points

Input: $n \geq 2$ points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$
Output: $i, j \in [n]$ such that the distance between p_i and p_j is minimum

```
i_1, i_2 ← 0, 1
min ← (x_1 − x_2)^2 + (y_1 − y_2)^2
for i = 1 to n do
    for j = i + 1 to n do
        d ← (x_i − x_j)^2 + (y_i − y_j)^2
        if d < min then
            min ← d
            i_1, i_2 ← i, j
return i_1, i_2
```

4. $\mathcal{O}(n^k)$. Given a graph $G = (V, E)$, find a $S \subseteq V$ such that $|S| = k$ and there is no edge between any two nodes in S.
   The above is known as the "independent set problem". The counterpart with an edge between any two nodes is known as the "clique problem".
   This is easily done by just checking every subset of size k, of which there are $\binom{n}{k}\mathcal{O}(k^2) = \mathcal{O}(n^k k^2)$ (there are $\mathcal{O}(k^2)$ comparisons for each subset). Since k is constant, this is just $\mathcal{O}(n^k)$.

   So for constant k, this *is* technically polynomial, but for reasonably large k, this is terrible. In fact, if we want to find the largest clique/independent set in a graph, then a polynomial time algorithm (in n) is not known at all. Indeed, this is an "NP-hard" problem, which we shall read more about later.

1.2. Greedy Algorithms

Greed is good, and sometimes, it's even optimal.

What is a greedy algorithm? It essentially builds a solution in tiny steps, performing some sort of *local* optimization, which ends up optimizing the problem requirement *globally*.
It is quite clear that greedy algorithms needn't always work, we must pick a local criterion that fits our requirements.

1.2.1. Interval Scheduling

Consider the "interval-scheduling problem". We have a supercomputer on which jobs need to be scheduled. We are given n jobs specified by their start and finish times. That is, for $i \in [n]$, we are given $J_i = (s(i), f(i))$. We wish to schedule as many jobs as possible on the computer. That is, given $J = \{J_i = (s(i), f(i)) : i \in [n]\}$, find a largest subset $S \subseteq J$ such that no two intervals in S overlap.
We want maximality in terms of *cardinality* of S (the number of jobs scheduled), not the total time covered.
Consider the following greedy algorithms.

1. Choose a job with the earliest starting time. This basically says that we never want to leave the computer idle. It is easily seen that this need not work, since the job that starts soonest can take a very long time, resulting in non-optimality.
   More concretely, let $J = \{(0, 3), (1, 2), (2, 3)\}$.

2. Choose the smallest available job. This needn't work either, since the smallest job could overshadow multiple (longer) jobs that intersect it.
   For example, let $J = \{(1, 5), (4, 6), (6, 10)\}$.

3. Choose the job with the earliest ending time. This *does* work. The idea behind this is that it tries to keep as many resources free as possible. One can try playing around with a few examples to convince oneself that it does work.
   But how would we prove that it works?

Let $\mathcal{A}$ be the set selected by the (last) greedy algorithm above and OPT denote an optimal solution. We wish to show that $|\mathcal{A}| = |\mathsf{OPT}|$.
Let $\mathcal{A} = \{a_1, \ldots, a_k\}$ and $\mathsf{OPT} = \{b_1, \ldots, b_m\}$.

Lemma. For $r \in [k]$, $f(a_r) \leq f(b_r)$.

Proof. This is easily shown via induction. For $r = 1$, it holds by the definition of $\mathcal{A}$. For $r > 1$, we have (by induction) $f(a_{r-1}) \leq f(b_{r-1}) \leq s(b_r)$. Then since b_r itself can be chosen by the algorithm, we must have that $f(a_r) \leq f(b_r)$. ■

This also implies that $\mathcal{A}$ is optimal since at each step, the job corresponding to OPT can be picked by $\mathcal{A}$. Try showing why this implies $|\mathcal{A}| = |\mathsf{OPT}|$.[2]
What is the running time of this algorithm? We first sort the jobs according to their finish times, which takes $\mathcal{O}(n \log n)$ comparisons. We then have to scan through all the jobs in this sorted list, which takes $\mathcal{O}(n)$ time. The total running time is thus $\mathcal{O}(n \log n)$.

1.2.2. Minimal Spanning Subgraph

Now, we consider the "Minimal Spanning Subgraph" problem. Given an undirected connected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{Z}^+$, find a subset $T \subseteq E$ such that T spans all the vertices, T is connected, and it is the set with the least cost among such sets.
It is easy to show that this T must be a tree. Indeed, suppose instead that it is connected, spanning and has a cycle C. If we delete the costliest edge e on C, then on removing e from T, T remains connected and spanning, but the weight goes strictly down (since c maps into $\mathbb{Z}^+$).
The brute force approach is to just iterate over all distinct spanning trees, but this is obviously quite terrible (exponential).
As it turns out, nearly any greedy strategy we come up with will work for this problem. We give four such algorithms.

(i) Choose an edge α such that $c(\alpha)$ is minimal. Each subsequent edge is chosen to be the cheapest among the remaining edges of G while ensuring that we do not form any cycles.

(ii) At each step, delete a costliest edge that does not destroy the connectedness of the graph.

(iii) Pick a vertex x_1 of G. Having found vertices $x_1, \ldots, x_k$ and an edge $x_i x_j$, $i < j$, for each vertex j with $j \leq k$, select a cheapest edge of the form $x_i x$, say $x_i x_{k+1}$, where $1 \leq i \leq k$ and $x_{k+1} \notin \{x_1, \ldots, x_k\}$. The process terminates after we have selected $n - 1$ edges.

(iv) This only works if all the edge costs are distinct. First, for each vertex, select the cheapest edge. After this, repeatedly select a cheapest edge between two distinct connected components until the graph becomes connected.

The first algorithm is also known as Kruskal's algorithm. More concretely, what it does is shown in Algorithm 4

[2] Attempt to prove that at any step of the algorithm, if S_i is the current set, then there is an optimal solution $\mathsf{OPT} \supseteq S_i$. Why does the required follow from this?

Algorithm 4: Kruskal's Algorithm

Input: A connected graph $G = (V, E)$ with $|V| = n$, $|E| = m$, and a cost function $c : E \to \mathbb{Z}^+$
Output: A minimal spanning subgraph of G

```
sort(E,c)                                   // sort the elements of E in non-decreasing cost
T ← ∅
for 1 ≤ i ≤ m do
    if T ∪ {e_i} doesn't have a cycle then
        T ← T ∪ {e_i}
return T
```

The set thus formed clearly has no cycles by construction. Spanningness follows due to its maximality. Proving that it is a minimal spanning tree is quite easy in the case where all costs are distinct using the following property of any minimal spanning tree.

Lemma (Cut Property)**.** Let $\varnothing \neq S \subsetneq V$. Let $e = vw$ be the minimal cost edge such that $v \in S$ and $w \in V \setminus S$. Then every minimal spanning tree of the graph must contain e.

To show that the required follows if we have the cut property, let T be a minimal spanning tree and T' the set output by the algorithm at some intermediate step.
Let $e = vw$ be the first edge added to T' in the subsequent steps and S be the neighbourhood of v in T'. Since the algorithm was able to add e to T', there is no edge in T' connecting any node in S to any node in $V \setminus S$ (we do not create cycles). We already know that e must be the lowest cost such edge. Then by the cut property, e must be present in T!
The only issue arises when v is not connected to any vertex in T', but this case is easily resolved.
Therefore, T must be minimum spanning.

Let us next show that T is connected. Suppose otherwise. There must then be a non-empty S such that no edge from S to $V \setminus S$ is in T. However, as G is connected, there is a minimum cost edge that connects S and $V \setminus S$ and by the cut-property, this edge must be in T.

Proof of the cut-property. Suppose we have set S and edge e as in the cut-property. let T be a spanning tree that does not contain e. Then adding e to T creates a cycle. Let P be a path in T that connects v to w. Let v' be the last vertex along this path in S and w' the first in $V \setminus S$. Let $e' = v'w'$.
Defining $T' = T \setminus \{e'\} \cup \{e\}$, we see that T' has lower cost than T, thus completing the proof. ■

Observe that the second algorithm given above is essentially Kruskal's algorithm in reverse.

The third algorithm given above arises quite naturally from the cut property – it is also known as *Prim's algorithm.*

Algorithm 5: Prim's Algorithm

Input: A connected graph $G = (V, E)$ with $|V| = n$, $|E| = m$, and a cost function $c : E \to \mathbb{Z}^+$
Output: A minimal spanning subgraph of G

```
T ← ∅, S ← {v}                              // v is an arbitrary node
while |S| < n do
    Compute E_s = {vw ∈ E : v ∈ S, w ∉ S}
    ẽ ← argmin_{e∈E_s} c(e)
    S ← S ∪ {w}                             // w is the vertex of ẽ that is in V \ S
    T ← T ∪ {ẽ}
return T
```

Proving optimality of the set output by the above is easily shown using the cut-property.

As a slight detour from what we have done thus far, how would one *implement* Prim's algorithm? We are given an undirected graph $G = (V, E)$ with edge costs given in an adjacency list and a source vertex $s \in V$.
We use a method similar to Djikstra's algorithm. Add an arbitary start vertex to the queue with key value 0 and let all other key values to be ∞. At each step, we use a priority queue to extract the node with the minimum key value from the queue. Explore the neighbourhood of the node, updating the key value. Here, the key value is the cost of the smallest cost edge leading to a node. The only change here is the update step, everything else is as in Djikstra's. The time complexity of this is the same as Djikstra's, namely $\mathcal{O}((m+n)\log n)$.

1.2.3. Huffman Coding

Given a file with data from an alphabet, convert it to a binary alphabet using as few bits as possible while keeping it uniquely decodable. For example, if we had $\Sigma = \{a, b, c, d, e\}$, using 3 bits would definitely get the job done since $\lceil \log_2(5) \rceil = 3$.
Suppose further that a, b, c, d, e occur with frequencies $100, 3, 5, 45, 20$ respectively. Using 3 bits for each would result in a total of 519 bits.
What if instead, we use fewer bits for letters that occur frequently? For example, what if we encode it as

$$\{a \mapsto 0), b \mapsto 100, c \mapsto 010, d \mapsto 1, e \mapsto 01\}?$$

This would only use 209 bits, which is obviously a huge improvement. However, this is *not* uniquely decodable (consider 0101).

Prefix-free encoding. It turns out that we must encode the letters such that each has a unique prefix (to ensure unique decoding). Consider

$$\{a \mapsto 1, b \mapsto 0000, c \mapsto 0001, d \mapsto 01, e \mapsto 001\}.$$

This enables unique decoding and uses 282 bits, which is obviously far better than 519. It uses fewer bits for low frequency letters, but a quite high number of bits for low frequency letters. Is it optimal? It is important to note that we desire optimality for this specific set of frequencies.

Greedy Strategy I. Sort the frequencies in non-increasing order ($f_1 \geq \cdots \geq f_n$). Use an i-length prefix-free string for f_i. It is quite easy to see that this fails spectacularly if all the frequencies are equal – in this case, the naïve encoding where each character takes $\log n$ bits is optimal.

Greedy Strategy II. This is a top-down strategy. Divide the letters into two sets of almost equal frequencies. Recursively code the two sets, giving a shorter prefix to the more frequent of the two sets. It turns out that this isn't optimal either.
This algorithm essentially creates a tree such that the nodes are subsets of the alphabet, leaves are single letters, and the depth of a leaf is the length of the code word associated with it. Specifying the tree specifies the code generated by this strategy (Why?).
The issue with this strategy is that we don't completely specify what "almost equal frequencies" mean, it is under-specified.

Greedy Strategy III. Since the top-down strategy doesn't work, perhaps we can try a bottom-up strategy. To construct the tree mentioned in the previous paragraph, start with the two least frequently occurring elements, say f, f'. Assign a string (suffix) 0 to one and 1 to the other. Combine the two frequencies to create a new letter with frequency $f + f'$. Repeat until all letters are assigned strings.
This strategy is optimal.
How do we prove correctness? For a tree T corresponding to a prefix-free encoding of Σ, we can associate it to a cost

$$c(T) = \sum_{1 \leq i \leq n} f_i d_T(a_i),$$

where $d_T(a_i)$ is the depth of a_i in T.
Observe that any optimal tree must be a full binary tree (any internal node has 2 children) – if not, we can shift

Algorithm 6: Huffman Coding

Input: An alphabet $\Sigma = \{a_1, \ldots, a_n\}$ and a frequency f_i for each a_i.
Output: A binary encoding of the letters for optimal compression

```
for 1 ≤ i ≤ n do
    Create a leaf node for a_i
    Insert it into the min-heap H with f_i as the key
while H has > 1 elements do
    Extract the two nodes with lowest key value from H, say u, v with keys f_u, f_v
    Create a new internal node w and f_w ← f_u + f_v
    Let u be w's left child and v its right child
    Insert w into H with key f_w.
return H
```

leaves lower down to a higher position and decrease the cost.

Lemma. Consider the two letters x and y with the smallest frequencies. There is an optimal code tree T^* in which these two letters are sibling leaves at the lowest level.

Proof. Let a and b be two letters at the lowest level of an optimal tree T that are siblings of each other. Let their frequencies be f_a and f_b. Assume $f_x \leq f_y$ and $f_a \leq f_b$. We may further assume that $f_x < f_a$.
Let T' be a tree formed from T by exchanging a and x. We know that $d_T(a) \geq d_T(x)$ and $d_T(b) \geq d_T(y)$. By definition, we have $c(T) \leq c(T')$. This implies that

$$c(T') \leq c(T) - (d_T(x)f_x + d_T(a)f_a - d_{T'}(x)f_x - d_{T'}(a)f_a) = c(T) - (d_T(x)(f_x - f_a) + d_T(a)(f_a - f_x)) < c(T),$$

thus yielding a contradiction and proving the claim.
Now, we have x and b as neighbours at the lowest level. We may assume $f_y < f_b$. Switching y and b as before, we get the required. ■

We shall next show that if a step in the algorithm is correct, the next step is also correct (the above lemma says that the first step is correct). That is,

Lemma. Let T be a tree corresponding to an optimal encoding of Σ. Let x, y be sibling leaves of T and z their parent in T. Let $f_z = f_x + f_y$, $T' = T \setminus \{x, y\}$, and $\Sigma' = \Sigma \setminus \{x, y\} \cup \{z\}$. Then T' corresponds to an optimal encoding of Σ'.

Proof. We have

$$c(T') = c(T) - d_T(x)(f_x + f_y) + (d_T(x) - 1)f_z = c(T) - f_z.$$

Non-optimality of T' is seen to result in non-optimality of T (if it is not optimal, then substituting z back with x and y in an optimal solution results in an encoding of Σ with cost strictly less than that of T), thus proving the claim. ■

Why does correctness follow? At each step, we are restricting ourselves to a smaller set while maintaining the presence of an optimal solution with the current (partial) mapping. At the end, the current mapping is a complete mapping, thus implying optimality.

1.3. Divide and Conquer Algorithms

The basic paradigm is as follows:

- A task needs to be solved on an instance of size n.
- Divide it into, say, k parts of size n/k each.

- Invoke recursion to solve each of these sub-problems.
- Combine the smaller answers to get the larger answer.

We *divide*, *delegate*, and *combine*. Once we have the answers, we get the time complexity as

$$T(n) = kT(n+k) + \text{time to combine.}$$

This can be unrolled using the Master Theorem:

Theorem 1.1 (Master Theorem)**.** Let $T(n) = aT(n/b) + \Theta(n^c)$, where $a, b, c \in \mathbb{N}$, $a \geq 1$, $b > 1$, and $c \geq 0$. Then

- $T(n) = \Theta(n^c)$ if $a < b^c$,
- $T(n) = \Theta(n^c \log n)$ if $a = b^c$, and
- $T(n) = \Theta(n^{\log_b a})$ if $a > b^c$.

More generally, if $T(n) = aT(n/b) + f(n)$, where $a, b \in \mathbb{N}$, $a \geq 1$, and $b > 1$, then

- $T(n) = \Theta(f(n))$ if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for all $\varepsilon > 0$,
- $T(n) = \Theta(n^{\log_b a} \log n)$ if $f(n) = \Theta(n^{\log_b a})$, and
- $T(n) = \Theta(n^{\log_b a})$ if $f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$ for all $\varepsilon > 0$.

A popular example of the divide and conquer paradigm is merge-sort. To sort an array, we divide it into two halves, sort the two halves separately, and merge the two (sorted) halves in sorted order. The overall time complexity is $\mathcal{O}(n \log n)$.

While in greedy algorithms the proof mainly comprised of showing that they return the correct answer, here the main issue is in showing that the algorithm actually runs – that the recombination is justified at each step.

1.3.1. Integer Multiplication

The input is two n-digit (non-negative) numbers x and y in decimal notation and we need to compute $x \times y$.

The basic algorithm studied in grade school is quite simple. What is the time complexity of this algorithm? It is *quadratic* in the number of digits ($\mathcal{O}(n^2)$). Here, the assumption is that adding two single digit numbers, multiplying two single digit numbers, or inserting a zero at the end of a number each take $\mathcal{O}(1)$ time. We stick with this assumption for the remainder of this section.

Is it possible to do better? The main algorithm we shall see in this section is known as *Karatsuba's Algorithm.*
Motivated by the divide-and-conquer paradigm, perhaps we could split each of the n-digit numbers into two numbers, each of $(n/2)$ digits. So for example, we split 1234 as 12 and 34.
In general, suppose we split x and y as a, b and c, d respectively. We compute $X = a \times c$ and $Y = b \times d$. We then compute $Z = (a+b) \cdot (c+d)$ and $W = Z - X - Y$. Finally, return $10^n \cdot X + 10^{n/2} \cdot W + Y$. Indeed,

$$\begin{aligned} x \times y &= (10^{n/2}a + b) \times (10^{n/2}c + d) \\ &= 10^n(a \times c) + 10^{n/2}(a \times d + b \times c) + b \times d \\ &= 10^n \cdot X + 10^{n/2} \cdot W + Y, \end{aligned}$$

so the algorithm is correct (we must use an inductive argument to conclude since calculating each of X, Z, Y use the same algorithm).
Is this an improvement over the naïve grade school algorithm? To perform the task for size n, we perform the task of size $(n/2)$ 3 times (for X, Y, Z) so

$$T(n) = 3T(n/2) + \mathcal{O}(n).$$

We have assumed that addition/subtraction take $\mathcal{O}(n)$.
Using the master theorem, it is easy to conclude that $T(n) = \mathcal{O}(n^{\log 3}) = \mathcal{O}(n^{1.584})$.
Note that if we instead calculate $a \times d$ and $b \times c$ separately (instead of calculating Z), there are 4 tasks of size $n/2$ at each step which results in an overall time of $\mathcal{O}(n^2)$.

Later, Andrei Toom broke the numbers into 3 parts and showed that 5 multiplications are enough. This gives $T(n) = 5T(n/3) + \mathcal{O}(n)$, resulting in a time complexity of $\mathcal{O}(n^{\log_3 5}) = \mathcal{O}(n^{1.465})$.

Stephen Cook attempted to generalize this idea even further. Breaking it into r parts reduces it to $\mathcal{O}(C(r)n^{\log_r(2r-1)})$, where $C(r)$ is some constant depending on r.
Later, Schönhage and Strassen managed to breach the $n^{1+\varepsilon}$ barrier and got it down to $\mathcal{O}(n \log n \log\log n)$. They also conjectured that $\mathcal{O}(n \log n)$ is the best possible running time.
Martin Fürer got it down to $\mathcal{O}(n2^{\log^* n} \log n)$ in 2007.
In 2019, Harvey and Hoeven managed to get the "perfect" method which takes $\mathcal{O}(n \log n)$. Soon after, a team at Aarhus University managed to prove that assuming the (unproven) Valiant's conjecture, this is indeed the best we can do.

1.3.2. Closest points in a plane

Suppose we have n points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$. We want to find i, j such that the distance between p_i and p_j is minimum.

There is obviously a naïve $\mathcal{O}(n^2)$ algorithm which does pairwise comparisons. Can we do better?

In the one-dimensional case, we can sort the points in $\mathcal{O}(n \log n)$ and then iterate through the points in $\mathcal{O}(n)$ to find the closest pair, which takes an overall time of $\mathcal{O}(n \log n)$.
How would we extend this to two dimensions? There is no ordering for the points in $\mathbb{R}^2$, so a similar idea will not work.

Divide the points into two halves. Recursively find the closest pair in each half. Finally, combine. The problematic part here is the recombination, which we want to take $\mathcal{O}(n)$.
If we just compare every pair across halves however, it takes $\Omega(n^2)$.

Suppose we split it into two halves based on the x-coordinates. Recursively compute the closest pair in each of the two halves. Let the minimum of these two distances be d. If the first division does not separate the closest pair, then d is the answer.
How many "cross-pairs" need to be checked though? Observe that we don't need to check anything for the points that are farther than d from the central separating line (Why?). But in the worst case, we might still need to check several points.

Lemma. Let $S_y = (q_1, \ldots, q_m)$ be the points in the distance d region sorted in decreasing order of their y coordinates. If the distance between q_i and q_j is less than d, then $j - i \leq 11$.

Proof. Consider the distance d region split into (closed) squares of side length $(d/2)$. In each row, there are 4 squares. Note that in each box, there is at most 1 point (if there was more than one, the minimum distance in one of the halves would be less than d). Suppose that there are more than 11 indices between i and j. Then at least 2 full rows separate the boxes containing q_i and q_j. However, this would imply that the distance between q_i and q_j is $\geq d$, thus yielding a contradiction and proving the claim. ∎

It is worth noting that the constant can be better than 11, but this is just a constant factor change and doesn't change the actual running time of the algorithm.
This then gives an $\mathcal{O}(n \log n)$ algorithm. Computing the first half of P_x (or P_y) takes $\mathcal{O}(n)$, and computing the answers by comparing the distances with the middle band takes $\mathcal{O}(n)$. Therefore,

$$t(n) \leq 2t(n/2) + \mathcal{O}(n) \text{ and the running time is } T(n) = t(n) + \mathcal{O}(n \log n),$$

where the extra $\mathcal{O}(n \log n)$ comes from having to sort the points based on x and y coordinate ahead of time.

1.3.3. Univariate polynomial multiplication

The input is two univariate polynomials A, B of degree $n-1$, which are described by coefficient vectors (a_i) and (b_i) of length n. The desired output is $C(x) = A(x)B(x)$ as a coefficient vector (c_i) of length $2n-1$.
This problem is also known as the "convolution problem".
It is seen that

$$c_k = \sum_{\substack{i,j<n \\ i+j=k}} a_i b_j,$$

so a naïve $\mathcal{O}(n^2)$ algorithm is easy to come up with. Note that here, we sweep the time taken to actually multiply integers under the rug, assuming that they can be done in constant time.
This problem feels quite similar in spirit to the integer polynomial multiplication algorithm that we have already seen. While something faintly similar to Karatsuba's would work, we need to make several significant changes to modify it into this (more general) context.
Inspired by what we got in Karatsuba's, can we do better than $\mathcal{O}(n^2)$?

Rather than multiplying A and B symbolically, consider evaluation of these polynomials. First, choose $2n$ values $\alpha_1, \ldots, \alpha_{2n}$, evaluate A and B at these points and then compute $C(\alpha_j) = A(\alpha_j)B(\alpha_j)$ for each j. Next, recover $C(x)$ from the evaluations by interpolating.
We have skimmed over most details here.
Since integer multiplication is assumed to take constant time, computing all the $C(\alpha_j)$ given $A(\alpha_j)$ and $B(\alpha_j)$ takes only $\mathcal{O}(n)$ time.
However, evaluating a polynomial at a single value takes $\Omega(n)$ time and we need to do so $\mathcal{O}(n)$ times! This gives a total time of $\mathcal{O}(n^2)$, which we still can't afford. We can't just separately evaluate each of the $A(\alpha_j)$ and $B(\alpha_j)$ hoping for it to work fast.

What is a more systematic and fast way of calculating $A(\alpha_i)$ and $B(\alpha_i)$? Perhaps we could somehow use values already calculated to speed up the current calculation.
Split the polynomial A into two parts:

$$A_0(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{(n-2)/2}$$

and

$$A_1(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{(n-2)/2}.$$

Observe that $A(x) = A_0(x^2) + xA_1(x^2)$ and the degree of each is around half the degree of A. Also observe that if we know A_0 and A_1, we can evaluate A in a constant number of operations.
The $2n$ values we choose should be related to each other to make the evaluations reusable. Consider the kth roots of unity given by $\omega_{j,k} = e^{(2\pi i)j/k}$ for $0 \leq j \leq k-1$. For our $2n$ evaluations, we shall use the $2n$th roots of unity.
The useful thing to note is that $\omega_{j,2n}^2 = \omega_{j,n}$.

Using this in our equation,

$$A(\omega_{j,2n}) = A_0(\omega_{j,n}) + \omega_{j,2n} A_1(\omega_{j,n}).$$

This is exactly what we desire from a divide and conquer algorithm! To calculate the value of A at the $2n$th roots of unity, we reduce it to evaluating the value of two polynomials of half the degree at the nth roots of unity.
Since it takes constant time to combine the evaluations, the recursion is just

$$T(n-1) = 2T\left(\frac{n-2}{2}\right) + \mathcal{O}(n) = \mathcal{O}(n \log n).$$

This rough idea is known as the *Fast Fourier Transform* and is extremely powerful (in fact, it is used in the faster algorithms in integer multiplication as well).
The final step in the algorithm, recovering C from its evaluations at $2n-1$ points, can be done in $\mathcal{O}(n \log n)$ using *Lagrange interpolation*.

We essentially have the following.

$$\begin{pmatrix} 1 & \alpha_1 & \cdots & \alpha_1^{2n-1} \\ 1 & \alpha_2 & \cdots & \alpha_2^{2n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_{2n-1} & \cdots & \alpha_{2n-1}^{2n-1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2n-1} \end{pmatrix} = \begin{pmatrix} C(\alpha_1) \\ C(\alpha_2) \\ \vdots \\ C(\alpha_n) \end{pmatrix}.$$

In general, solving this set of equations takes $\mathcal{O}(n^3)$ by inverting the Vandermonde matrix on the left. However, we do not have any arbitrary points. Can we do better?
We shall reduce the problem of *interpolation* of C to the *evaluation* of another polynomial D.

Lemma. Let $C(x) = \sum_{s=0}^{2n-1} c_s x^s$ and $D(x) = \sum_{s=0}^{2n-1} d_s x^s$, where for each s, $d_s = C(\omega_{s,2n})$. Then for each s,

$$c_s = \frac{1}{2n} D(\omega_{2n-s,2n}).$$

Proof. ■

This allows us to evaluate the (c_s) in $\mathcal{O}(n \log n)$ time by evaluating D at the required points (using the methods we have already seen).

1.4. Dynamic Programming

1.4.1. The Paradigm

Dynamic programming has a very standard template.

1. Figure out the types of sub-problems.
2. Define a recursive procedure.
3. Decide on the memoization strategy[3], which involves keeping a memo of things we've done so far.
4. Check that the sub-problem dependencies are acyclic (we don't get stuck in cycles in the recursion).
5. Analyze the time complexity using the recursion.

This is hopefully made more clear by the following several examples.

1.4.2. Fibonacci Numbers

Say we wish to find the kth Fibonacci number F_k.

The naïve algorithm where we recursively calculate F_{k-1} and F_{k-2} takes running time

$$T(k) = T(k-1) + T(k-2) + 1 = \mathcal{O}(2^n).$$

[3] No, there is no typo here.

This algorithm is obviously extremely inefficient. Observe that each F_r is calculated multiple times, resulting in a lot of redundancy. Since we've already calculated it, we don't really need to recurse. How do we implement this?

Algorithm 7: Calculating the kth Fibonacci number

```
1  Fibo(r)
2      if Table contains Fibo(r) then
3          return the table value
4      if r = 0 or r = 1 then
5          val ← k
6          Store val as the kth entry
7      else
8          val ← Fibo(r − 1) + Fibo(r − 2)
9          Store val as the rth entry in the table
10         return val
11 return Fibo (k)
```

1.4.3. Weighted Interval Scheduling

This problem is quite similar to the usual interval scheduling problem we studied, except that instead of maximizing the cardinality of the chosen set, we want to maximize the sum of the weights of the chosen set.
There is no nice greedy heuristic for this, can we design a recursive algorithm that works well?

Sort the jobs in decreasing order of finishing time. For each i, denote by $p(i)$ the last job j with $j < i$ that does not conflict with i.
Suppose we iterate backwards and we are currently at index j. Then if the jth job is in an optimal solution OPT, then no job i with $j > i > p(n)$ cannot belong to OPT. If $j \notin$ OPT, then we can recurse on $\{1, \ldots, j-1\}$.

Algorithm 8: Weighted Interval Scheduling Problem

Input: A set of n tasks, each with a start time $s(i)$, finish time $f(i)$, and weight $w(i)$ ordered in non-increasing finish time.
Output: An optimal solution to the weighted interval scheduling problem

```
1  WtIntSc(i)
2      if i = 0 then
3          return i
4      else
5          if Table(i) is non-empty then
6              return
7          else
8              Table(i)
9
10     Table(i) ← max{w(i) + WtIntSc(p(i)), WtIntSc(i − 1)}
11 return WtIntSc (n)
```

Correctness is easily quite shown using strong induction on i – that `WtIntSc(i)` omputes the optimal solution for the subproblem containing tasks $\{1, \ldots, i\}$, where the jobs are ordered according to their finish times.

The sorting takes $\mathcal{O}(n \log n)$ time. Computing the $p(i)$ takes $\mathcal{O}(n \log n)$ using binary search. The number of calls made to the subroutine `WtIntSc` is $\mathcal{O}(n)$. Therefore, the overall time is $\mathcal{O}(n \log n)$.

1.4.4. String-Related Problems

There are numerous interesting problems that are related to strings/sequences which can be solved using dynamic programming.

1. Figure out the *parenthesization* of an expression that minimizes the overall cost of evaluating the expression.
2. Given a string of positive numbers, find the *longest increasing subsequence*.
3. Given a long string of letters, find a way (if one exists) to *segment* the string into chunks such that the segmented string is a statement that makes sense (in a particular language).
4. Given two strings, find the *longest subsequence* common to both.
5. Given two string x and y, find the smallest number of *updates* (deletions, insertions, and swaps) needed to convert x into y.

In string-related problems, the sub-problem tends to be something of the form $\texttt{suff}[1, i]$, $\texttt{pre}[i, n]$, or sometimes $\texttt{substring}[i, j]$.
Let us look at the parenthesization problem for example. Suppose we have a $n \times 1$ matrix A, a $1 \times n$ matrix B, and a $n \times 1$ matrix C and we want to compute $A \times B \times C$. If we parenthesize it as $(A \times B) \times C$, it will cost $\mathcal{O}(n^2)$ operations. If we compute it as $A \times (B \times C)$ on the other hand, it will cost $\mathcal{O}(n)$ operations. In general, we want the intermediate steps to result in small matrices.
Suppose we have a string of matrices $A_1, \ldots, A_n$, where each A_i is of size $c_i \times r_i$.

1. Figure out the types of sub-problems. If we guess the topmost multiplication (the outermost one) of an optimal solution, then the sub-problems are the two smaller multiplications. So if we split it as $(A_1 \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_n)$, the two expressions are the sub-problems. Subdividing further, the sub-problems are intervals. Let us denote $A_i \times \cdots \times A_j$ as $A_{[i,j]}$.
2. Define a recursive procedure. Denote by $\texttt{para}(i, j)$ the problem of computing the minimum cost parenthesization of $A_{[i,j]}$. We see that

$$\texttt{para}(i, j) = \min_{i \le k < j} \left\{ \texttt{para}(i, k) + \texttt{para}(k + 1, j) + \texttt{cost}(i, k, j) \right\},$$

   where $\texttt{cost}(i, k, j)$ is the cost of computing the product of a $c_i \times r_k$ matrix ($A_{[i,k]}$) and a $c_{k+1} \times r_j$ matrix ($A_{[k+1,j]}$).
3. Decide on the memoization strategy. This can be done on the fly since $\texttt{cost}(i, j, k)$ can be computed in $\mathcal{O}(1)$ time. We recursively store the computed values of $\texttt{para}(i, j)$, starting from small intervals and building up to larger intervals.
4. Check that the problems are acyclic. There is nothing much to argue here, the `para` routine uses smaller intervals for larger intervals.
5. Analyze the time complexity using the recursion. The total number of sub-problems in $\mathcal{O}(n^2)$. For each sub-problem, the time required is the time needed to compute a minimum over a set of values. Interval $[i, j]$ takes $\mathcal{O}(j - i) = \mathcal{O}(n)$ time. Therefore, the total time required is $\mathcal{O}(n^3)$.

1.4.5. The Shortest Path Problem

Given a directed graph $G = (V, E)$ and a weight function $w : E \to \mathbb{Z}$ and two designated vertices $s, t \in V$, find the length of the shortest path from s to t.

This is slightly different from the usual problem that Djikstra's algorithm solves because weights can be negative.
What happens if the graph has cycles? It is possible to have a cycle with overall negative weight, so we could just loop in the cycle for an arbitrary amount of time, thus resulting in arbitrarily low (very negative) cost. Such a cycle is known as a "negative cycle".
Therefore, let us restrict ourselves to directed *acyclic* graphs for now (a more logical restriction would be to one only

without negative cycles, but let us stick with this).

Can we apply Djikstra's algorithm directly? No, we cannot. Consider the graph with $V = \{a, b, c\}$, $E = \{\overrightarrow{ac}, \overrightarrow{ab}, \overrightarrow{bc}\}$, $s = a$, $t = c$, and $w = \{\overrightarrow{ac} \mapsto 1, \overrightarrow{ab} \mapsto 2, \overrightarrow{bc} \mapsto -5\}$.
It is solved using *Bellman and Ford's Algorithm*.
Let $\mathsf{OPT}(v, t)$ be the minimum weight of v to t path. We want to compute $\mathsf{OPT}(s, t)$. Observe that

$$\mathsf{OPT}(s,t) = \min_{u:\overrightarrow{su}\in E} \{w(\overrightarrow{su}) + \mathsf{OPT}(u,t)\}.$$

This easily yields itself to a dynamic programming algorithm, with the above being the central recursion.
The memoization strategy involves remembering previously computed $\mathsf{OPT}(u, t)$. The problems are acyclic because the graph is acyclic.
What is the time complexity? There are at most $|V|$ sub-problems and each sub-problem takes time $\mathcal{O}(|V|)$ (we need to find the minimum of at most $|V| - 1$ elements). So the algorithm is overall $\mathcal{O}(|V| + |E|)$.

Now what happens if the algorithm has cycles, but no negative weight cycles? Is it possible to modify the above existing algorithm? Maybe we can use some sort of "time-stamp" idea.
For each i, denote by $\mathsf{OPT}(v, i)$ the minimum weight path from v to t that uses at most i edges. We want to compute $\mathsf{OPT}(s, n-1)$ (because all cycles are non-negative cycles).
The recursion is given by

$$\mathsf{OPT}(v,i) = \min\left\{\mathsf{OPT}(v,i-1), \min_{u:\overrightarrow{vu}\in E}\{w(\overrightarrow{su}) + \mathsf{OPT}(u,i-1)\}\right\}. \tag{1.1}$$

The memoization strategy involves creating a table M with n rows and n columns, where the (v, i)th entry contains the weight of $\mathsf{OPT}(v, i)$.
The sub-problem dependencies are acyclic because i decreases at every step.

Algorithm 9: Bellman and Ford's Algorithm

Input: A set of n tasks, each with a start time $s(i)$, finish time $f(i)$, and weight $w(i)$ ordered in non-increasing finish time.
Output: An optimal solution to the weighted interval scheduling problem

```
1 ShortestPath(G, s, t)
2     M[t, 0] ← 0
3     for v ∈ V \ {t} and i ∈ [n − 1] do
4         M[v, i] ← ∞
5     for i = 1 to n − 1 do
6         for v ∈ V do
7             Compute M[v, i] using Equation (1.1)
8     return M[s, n − 1]
```

The table has $\mathcal{O}(n^2)$ entries and each entry is filled in $\mathcal{O}(n)$. Therefore, the total running time is $\mathcal{O}(n^3)$.

A problem related to the one we have studied here is $\texttt{Cycle}(G, t)$, where given a directed graph $G = (V, E)$, a function $w : E \to \mathbb{Z}$, and a designated vertex $t \in V$, we output yes iff there is a negative cycle with a path reaching t.
Another is $\texttt{Cycle}(G)$, where we return yes iff there is a negative cycle in the graph.

$\texttt{Cycle}(G, t)$ and $\texttt{Cycle}(G)$ are closely interrelated. If we solve $\texttt{Cycle}(G, t)$, then we claim to be able to solve $\texttt{Cycle}(G)$. This is known as a *reduction*, which we shall look at in more detail later on.
Given a graph $G = (V, E)$, add a new vertex t_0 to it. Add directed edges from every $v \in V$ to t_0 of weight 0. Let this graph be G'. Then solving $\texttt{Cycle}(G', t_0)$ is equivalent to solving $\texttt{Cycle}(G)$! This reduction is symbolically represented as $\texttt{Cycle}(G) \leq \texttt{Cycle}(G', t_0)$.
It only remains to solve $\texttt{Cycle}(G, t)$.

Lemma. There is no negative cycle in G with a path to t iff $\mathsf{OPT}(v,i) = \mathsf{OPT}(v,n-1)$ for every $v \in V$ and $i \geq n$.

Above, OPT is the same as what we defined in Bellman and Ford's Algorithm. The proof for the above is quite clear. If a node v can reach t and is part of a negative cycle, then increasing i, $\mathsf{OPT}(v,i)$ can be made arbitrarily small.

Lemma. There is no negative cycle in G with a path to t iff $\mathsf{OPT}(v,n) = \mathsf{OPT}(v,n-1)$ for each $v \in V$.

1.5. Exercises

Exercise 1.1. Let A be an array of n distinct numbers. A number at location $1 < i < n$ is said to be a maxima in the array if $\mathsf{A}[i-1] < \mathsf{A}[i]$ and $\mathsf{A}[i] > \mathsf{A}[i+1]$. Also, $\mathsf{A}[1]$ is a maxima if $\mathsf{A}[2] < \mathsf{A}[1]$ and $\mathsf{A}[n]$ is a maxima if $\mathsf{A}[n] > \mathsf{A}[n-1]$. Find a maxima in the array in time $\mathcal{O}(\log n)$.

Solution

The basic idea behind this algorithm is to, at each step, greedily check the half of the array that contains the greater element among the two neighbours of the current element. It is described more precisely in Algorithm 10. We encourage the reader to prove the correctness of this algorithm.

Algorithm 10: Solution 1.1

Input: An array A containing n elements (1-indexed)
Output: A maxima in A

```
greedyStep(B)
    if size(B) = 1 then
        return B[0]
    mid ← size(B)/2
    if B[mid] > B[mid + 1] and B[mid] > B[mid − 1] then
        return B[mid]
    else if B[mid] ≤ B[mid − 1] then
        return greedyStep(B[1 : mid/2])
    else if B[mid] ≤ B[mid + 1] then
        return greedyStep(B[1 + mid/2 : size(B)])
return greedyStep(A)
```

Exercise 1.2. Let $G = (V,E)$ be an undirected graph. Consider the following greedy algorithm:

Algorithm 11: Exercise 1.2

Input: A connected graph $G = (V,E)$ with $|V| = n$, $|E| = m$, and a cost function $c : E \to \mathbb{Z}^+$

```
M ← ∅. Let V(M) be the set of vertices in M.
foreach uv ∈ E do
    if V(M) ∩ {u, v} = ∅ then
        M ← M ∪ {uv}
return M
```

(a) Give a graph G such that the above algorithm may not output a perfect matching in G even if it has a perfect matching.

(b) Give a graph that has a perfect matching M and a maximal matching M' such that $M' \neq M$.

(c) Prove that the above algorithm finds a maximal matching.

Solution

For parts (a) and (b), consider the graph $G = (V, E)$ with $V = \{1, 2, 3, 4\}$ and $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{1, 3\}\}$. Then $\{\{1, 3\}\}$ is a maximal matching but not a perfect matching and $\{\{1, 2\}, \{3, 4\}\}$ is a maximal matching. The former may be output by the algorithm depending on which edge is chosen.
Part (c) is easily shown. If the output is not a maximal matching, then there is some edge $e = vw$ such that $E \cap \{v, w\} = \varnothing$ (by definition). However, this contradicts the termination of the algorithm, thus proving the required.

Exercise 1.3. Consider the following modified interval scheduling problem. There are n jobs, each with a start and finish time $(s(i), f(i))$ and in addition, they have a (positive) weight $w(i)$. The problem is to maximize the total weight of the jobs scheduled on a (single) machine. For the sake of brevity, we denote the set of jobs J by a set of tuples with the ith tuple being $(s(i), f(i), w(i))$.

(a) Show that the algorithm for the usual interval scheduling need not give an optimal solution for this problem.

(b) Suppose a greedy algorithm picks the largest weight job with the earliest finishing time among the available jobs at each step. Prove/disprove that this strategy works.

(c) We say that jobs J and J' *overlap*, denoted $J \| J'$, if $J \cap J' \neq \varnothing$. For a job J_i, define $O_i = \sum_{i' : J_{i'} \cap J_i} w(i')$, called the *overlap-weight* of J_i. Consider a greedy algorithm that schedules a job with the smallest overlap-weight among the available jobs at each step. Prove/disprove that this algorithm gives the optimal solution.

Solution

(a) Consider the set of jobs $J = \{(1, 2, 5), (1, 3, 10)\}$. Then the algorithm chooses only the first job, while the optimal solution picks the second.

(b) Consider the set of jobs $J = \{(1, 2, 5), (2, 3, 5), (1, 3, 8)\}$. Then this algorithm chooses only the third job, whereas the optimal solution picks the first two.

(c) The algorithm is incorrect. Consider the counterexample

$$J = \{(0, 1, 1), (1, 2, 1), (0, 2, 3)\}.$$

Then the algorithm chooses jobs 1 and 2, whereas the (unique) optimal solution picks job 3.
The reader might be tempted to think that the algorithm would work if in the definition of O_i, we only consider those $i' \neq i$. However, this doesn't work either. Indeed, consider

$$J = \{(1, 3, 5), (1, 2, 1), (2, 4, 5), (3, 5, 5), (4, 5, 1)\}.$$

The algorithm chooses jobs 2, 3, and 5, whereas the (unique) optimal solution chooses jobs 1 and 4.
This idea is made more natural on rewriting the problem as finding $\mathcal{A} \subseteq [n]$ that minimizes $w\left(\bigcup_{i \in \mathcal{A}} O(i)\right)$, where $O(i) = \{i' \in [n] : i' \neq i \text{ and } J_{i'} \| J_i\}$ and $w(S) = \sum_{s \in S} w(s)$ for any $S \subseteq [n]$. The issue arises because a single j might appear in multiple $O(i)$. It is worth noting that there is a dynamic programming algorithm to solve this problem (which we shall study later).

Exercise 1.4. You are given a set S of n pairs of numbers $(\ell_1, c_1), (\ell_2, c_2), \ldots, (\ell_n, c_n)$ and some C (all positive). You are required to find a subset T of $[n]$ such that $\sum_{i \in T} c_i \leq C$ while maximizing the sum $\sum_{i \in T} \ell_i$. Consider the algorithm to do the same that picks jobs with the largest values of ℓ_i / c_i. This is described in Algorithm 12.

(a) Show that this algorithm need not find the optimal subset.

(b) Can you suggest any other greedy strategy to come up with the optimal subset in the above setting?

(c) Suppose that a "fractional" part of each of the n pairs can be taken. Choosing a fraction f_i of the item c_i adds cost $c_i f_i$. Give an optimal algorithm in this case. That is, we want to maximize $\sum_{i \in [n]} f_i \ell_i$ constrained by $\sum_{i \in [n]} f_i c_i \leq C$.

Algorithm 12: Exercise 1.4

Input: A set S of n tuples (ℓ_i, c_i) (each positive) and some $C > 0$
Output: $T \subseteq [n]$ that maximizes $\sum_{i \in S} \ell_i$ subject to the constraint $\sum_{i \in S} c_i \leq C$

```
merge-sort(S, ℓ_i/c_i)                    // Arrange in non-increasing order of (ℓ_i/c_i)
S ← ∅, cost ← 0
for 1 ≤ i ≤ n do
    if cost + c_i ≤ C then
        S ← S ∪ {i}
        cost ← cost + c_i
return S
```

The above problem is more often known as the "knapsack problem".

Solution

(a) Consider the set of tuples $\{(6,2),(9,3),(6,3),(8,4)\}$ with $C = 7$. The algorithm returns the set $\{(6,2),(9,3)\}$ with $\sum_i \ell_i = 15$ whereas the optimal solution is $\{(9,3),(8,4)\}$ with $\sum_i \ell_i = 17$.

(b) No, not yet.

(c) In the context of part (b), this just means that instead of choosing an $x_i \in \{0,1\}$ for each i, we choose $f_i \in [0,1]$ for each i. The algorithm is very similar in spirit to that given in the problem statement (of (a)). It is described explicitly in Algorithm 13. Correctness is easily proved.

Algorithm 13: Solution 1.4(c)

Input: An set S of n tuples (ℓ_i, c_i) (each positive) and some $C > 0$
Output: An $f_i \in [0,1]$ for each i that maximizes $\sum_{i \in S} f_i \ell_i$ subject to the constraint $\sum_{i \in S} f_i c_i \leq C$

```
merge-sort(S, ℓ_i/c_i)
S ← ∅, cost ← 0
f_i ← 0 for each i
for 1 ≤ i ≤ n do
    if cost + c_i ≤ C then
        f_i ← 1
        cost ← cost + c_i
    else
        f_i ← (C − cost)/c_i
        break
return (f_i)
```

Exercise 1.5. (Dis)prove that when all the edges in an undirected graph have distinct costs, the minimum spanning tree in the graph is unique.

Solution

Suppose instead there are two distinct MSTs T, T'. Let e be the least costly edge that is in exactly one of the trees. Suppose it is in T. $T' \cup \{e\}$ must contain an edge e' that is not in T which is in the newly formed cycle. Then by the MST nature of T' and the fact that all edges have distinct costs, $c(e) > c(e')$ (otherwise, $T' \cup \{e\} \setminus \{e'\}$ is a strictly cheaper tree). We similarly get $c(e') > c(e)$, thus resulting in a contradiction and proving the required.

Exercise 1.6. Suppose we have a computer and some n jobs. For the ith job, there are two parts with durations $t_1(i), t_2(i) > 0$. The part of duration $t_1(i)$ must be performed on the computer (only one such part can be scheduled

at a time) whereas the part of duration $t_2(i)$ can be performed at any point after the first part of the same job is done (multiple such parts can be scheduled simultaneously). Give an algorithm that designs an ordering for the jobs to be sent to the computer such that the overall time taken to complete all jobs is minimized.

Solution

The problem can be stated alternatively as: given n and two functions $t_1, t_2 : [n] \to \mathbb{R}^+$, find a permutation σ of $[n]$ such that

$$Q_\sigma = \max_{k \in [n]} \left(t_2(\sigma(k)) + \sum_{1 \le i \le k} t_1(\sigma(i)) \right)$$

is minimized. ($\sigma(i)$ denotes the position that is at the ith index after permuting)
Assume without loss of generality that the jobs are ordered in non-increasing order of t_2. We claim that then, the identity permutation suffices. Let Q be the value of Q_σ for the identity permutation and $k \in [n]$ attain the maximum involved. Let σ be any permutation of $[n]$.
Claim. If $Q_\sigma \le Q$, then for every $1 \le i \le k$, $1 \le \sigma(i) \le k$. This implies that the first k positions permute among themselves in any optimal solution.
Suppose otherwise and let $r = \max\{j \in [n] : \sigma(j) = i \text{ for some } 1 \le i \le k\} > k$. Then,

$$Q_\sigma \ge t_2(\sigma(r)) + \sum_{1 \le i \le r} t_1(\sigma(i)) > t_2(k) + \sum_{1 \le i \le k} t_1(i) = Q,$$

thus proving the claim.
Now, let σ be a permutation such that $Q_\sigma \le Q$. Then by the above claim and since the t_2 are in non-increasing order, $t_2(\sigma(k)) \ge t_2(k)$. Then

$$Q_\sigma \ge t_2(\sigma(k)) + \sum_{1 \le i \le k} t_1(\sigma(i)) = t_2(\sigma(k)) + \sum_{1 \le i \le k} t_1(i) \ge Q,$$

thus implying that the identity permutation is optimal.

Exercise 1.7. Given a set of n intervals, design a greedy algorithm to find the smallest subset of intervals such that every interval is contained in the union of the intervals of the subset.

Solution

First, we order the intervals as $\{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\}$ such that if $j > i$, either $a_j > a_i$ or $a_j = a_i$ and $b_j < b_i$ (not $>$). We denote (a_i, b_i) by A_i.
The algorithm performs a linear scan of the A_i. At the first step, choose A_1. At some step of the algorithm, let the last interval chosen be A_k. Among the intervals A_i with $a_i < b_k$ and $b_i > b_k$, choose that interval with the largest b_i. In case there is no such interval, then just choose the first interval encountered with $a_i \ge a_k$.
It is easy to see that any interval is contained in the union of these intervals.

To show correctness, it suffices to consider the first case of the algorithm alone (Why? It splits into smaller disjoint problems). Let S_i be the set of indices chosen by the algorithm so far when we are at index i. We shall show that for any i, there is an optimal solution OPT_i such that $S_i \subseteq \mathsf{OPT}_i$. For $i = 0$, it is trivial since $\varnothing$ is a subset of any optimal solution OPT. Suppose it is true for some $r - 1 \ge 0$. If the algorithm does not choose r, then $\mathsf{OPT}_r = \mathsf{OPT}_{r-1}$ will work. Now, let r be chosen and k be the most recently chosen index. It is easily shown that OPT_{r-1} must contain some j such that $a_j < b_k < b_j$ (How?). We can then set $\mathsf{OPT}_r = \mathsf{OPT}_{r-1} \setminus \{j\} \cup \{k\}$, thus proving the claim.
The required follows by the above claim (because the output of the algorithm is a valid solution).

Exercise 1.8. Given a permutation of the numbers 1 through n, count the number of pairs (i, j) such that all numbers that occur between i and j in the permutation have value between i and j.

Solution

It is relatively straightforward to come up with an $\mathcal{O}(n^2)$ algorithm for this. It is possible to come up with a $\mathcal{O}(n \log n)$ algorithm as well, which is what we describe here.
Denote the number at index i of the permutation by σ_i.
As a preprocessing step, for each index i, we compute the largest j such that $\sigma_j > \sigma_i$ and $j < i$. This can be accomplished in $\mathcal{O}(n)$, as can be seen in this For index i, denote this number as l_i.
We iterate upwards from 1 through n, and at each step, we increment our answer by the number of "blocks" that end at the current index. To do so, maintain an array of indices (which we update as we progress) such that for any index j in the array, we have not found an element that is less than j yet. That is, if we are at index i, let

$$\mathsf{A} = (j \in [i] : \text{there is no } j < k \leq i \text{ such that } \sigma_k < \sigma_j)\,.$$

At each step, we first update A. First of all, note that the elements of A are arranged in increasing order (the (σ_j) are in increasing order). Indeed, if not, then we have found an element less than j to the right, so it should not be in the array at all. To update A, find the rightmost (smallest) element in the array that is greater than σ_i and remove all the indices before it – this can be done in $\mathcal{O}(\log n)$ using binary search.
To update the answer at the current step, find the greater index j_i that is smaller than l_i. Increment the answer by $|\mathsf{A}| - |\mathsf{A} \cap [j_i]|$.

§2. Flow

2.1. Introduction

Where there are graphs, we tend to see flow networks quite a lot.

Definition 2.1 (Flow Network)**.** A directed graph $G = (V, E)$ together with a *capacity* function $c : E \to \mathbb{Z}^+$, a designated *source* s, and a designated *sink* t is known as a *flow network*.

We shall assume that there are no edges entering s and no edges leaving t.
We can think of s as a source of water, with the intermediate edges being pipes that have some limit to how much they can carry and the intermediate nodes being junctions that transmit water, and t as a reservoir.
The main question we wish to answer is:

> What is the maximum rate that can be sent from the source to the sink without violating any capacity constraints?

This is modelled more concretely using the following definition.

Definition 2.2 (Flow)**.** A *flow* in a flow network is a function $f : E \to \mathbb{R}^{\geq 0}$ that satisfies

- Capacity constraints: For each $e \in E$, $f(e) \leq c(e)$.
- Flow conservation: For each $v \in V \setminus \{s, t\}$,

$$\sum_{\overrightarrow{uv} \in E} f(\overrightarrow{uv}) = \sum_{\overrightarrow{vw} \in E} f(\overrightarrow{vw}).$$

The *value of the flow* f, denoted $|f|$, is given by

$$|f| = \sum_{\overrightarrow{sv} \in E} f(\overrightarrow{sv}).$$

The flow conservation rule written above is also known as *Kirchhoff's law*. For the sake of brevity, we denote

$$f^{\leftarrow}(v) = \sum_{\overrightarrow{uv} \in E} f(\overrightarrow{uv}) \text{ and}$$
$$f^{\rightarrow}(v) = \sum_{\overrightarrow{vw} \in E} f(\overrightarrow{vw}).$$

We then have $|f| = f^{\rightarrow}(s)$.
For any $U \subseteq V$, we use $f^{\leftarrow}(U)$ and $f^{\rightarrow}(U)$ to denote the flow from $V \setminus U$ to U and U to $V \setminus U$ respectively. That is,

$$f^{\rightarrow}(U) = \sum_{\substack{u \in U, v \in V \setminus U \\ \overrightarrow{uv} \in E}} f(\overrightarrow{uv}).$$

Lemma 2.1. For any flow f on a flow network with source s and sink t,

$$|f| = \sum_{\overrightarrow{sv} \in E} f(\overrightarrow{sv}) = f^{\rightarrow}(s) = \sum_{\overrightarrow{vt} \in E} f(\overrightarrow{vt}) = f^{\leftarrow}(t).$$

Proof. We have

$$\begin{aligned}
|f| &= f^{\rightarrow}(s) + \sum_{v \in V \setminus \{s,t\}} (f^{\rightarrow}(v) - f^{\leftarrow}(v)) \\
&= \sum_{v \in V \setminus \{t\}} (f^{\rightarrow}(v) - f^{\leftarrow}(v)) && (f^{\leftarrow}(s) = 0) \\
&= f^{\rightarrow}(V \setminus \{t\}) - f^{\leftarrow}(V \setminus \{t\}) \\
&= f^{\leftarrow}(t) - f^{\rightarrow}(t) = f^{\leftarrow}(t). && \text{(there are no outgoing edges from the sink)}
\end{aligned}$$

■

2.2. Max-Flow

In the *maximum flow problem* (shortened as max-flow), we are given a flow network $G = (V, E)$ along with a capacity function $c : E \to \mathbb{N}$. The output should be the maximum valued flow that can be transferred in the network.[4]

Definition 2.3. Given a directed graph $G = (V, E)$ with source s, sink t, and a capacity function $c : E \to \mathbb{N}$, an (s, t)-*cut* or just *cut* is given by $S, T \subseteq V$ such that

- $s \in S$, $t \in T$,
- $S \cup T = V$, and $S \cap T = \varnothing$.

Given an (s, t)-cut (S, T), we define its *capacity* by

$$\mathtt{cap}(S, T) = \sum_{\substack{u \in S, v \in T \\ \overrightarrow{uv} \in E}} c(\overrightarrow{uv}).$$

That is, the capacity of the cut is essentially the capacity of the edges across the cut.
In the *minimum cut problem* (shortened as min-cut), we are given a flow network $G = (V, E)$ along with a capacity function $c : E \to \mathbb{N}$. The output is the cut that has minimum capacity.
The max-flow and min-cut problems are in fact very closely related. When a minimization problem and maximization problem are related, we usually refer to it as a min-max relationship.
For a flow f and some edge e, we say that f *saturates* e if $f(e) = c(e)$ and *avoids* e if $f(e) = 0$.

Lemma 2.2. Let f be any flow in a flow network G. Let (S, T) be any (s, t)-cut in G. Then $|f| \leq \mathtt{cap}(S, T)$. In particular, the value of the maximum flow is at most the capacity of the minimum cut. Moreover, $|f| = \mathtt{cap}(S, T)$ if and only if f saturates every edge from S to T and avoids every edge from T to S.

[4] This is well-defined since the set of flow values is bounded above by the sum of capacities, so has a supremum. Further, this bound is attained, as can be shown by considering a sequence of flows that converge (in value) to the maximum flow value. We can then show that the flow in each edge must converge as well for some subsequence (using the Bolzano-Weierstrass Theorem), thus implying the required since the capacity constraint has a weak inequality and not a strong one.

Proof. We have

$$\begin{aligned}
|f| &= f^{\rightarrow}(s) \\
&= f^{\rightarrow}(S) - f^{\leftarrow}(S) \\
&\leq f^{\rightarrow}(S) \\
&= \sum_{\substack{u \in S \\ v \in T}} f(\overrightarrow{uv}) && \text{(here, } f(\overrightarrow{uv}) = 0 \text{ if } \overrightarrow{uv} \notin E) \\
&\leq \sum_{\substack{u \in S \\ v \in T}} c(\overrightarrow{uv}) = \mathsf{cap}(S,T) && \text{(here, } f(\overrightarrow{uv}) = 0 \text{ if } \overrightarrow{uv} \notin E).
\end{aligned}$$

The second part is easily seen from the above series of inequalities, where equality holds iff $f^{\leftarrow}(S) = 0$ and for each $u \in S$, $v \in T$, $f(\overrightarrow{uv}) = c(\overrightarrow{uv})$. ■

Let us try to come up with an algorithm to solve the max-flow problem.
Let us start with $f(e) = 0$ for all $e \in E$. We then try to "push" some flow along an s-t path allowed by the capacity constraints. We then ask if this is the maximum flow possible. We try to push more flow along another path. However, we might face a problem wherein a previous flow restricts the new flow we are trying to push. We then want to "undo" the previous flow. How do we formalize this notion?

Definition 2.4 (Residual Graph)**.** Given a graph G with capacity function $c : E \to \mathbb{N}$ and a flow f, a *residual graph* G with respect to f, denoted G_f is such that

- its vertex set is the same as that of G.
- if e is an edge in G such that $f(e) < c(e)$, then G_f has the edge e with capacity $c(e) - f(e)$.
- if $e = \overrightarrow{uv}$ is such that $f(e) > 0$, we add an edge $\overrightarrow{vu}$ in G_f with capacity $f(e)$ on it. This is called a *backward edge*.

In the above, "undoing" a flow just means sending flow along a backward edge and adding more flow corresponds to sending a flow along a normal edge (with a lowered capacity). This behaves as an intermediate step to finding the optimal flow.
For each edge of G, note that there at most two edges in G_f. So, the size of G_f is at most twice that of G.
Let π be any s-t path in G_f. Denote by $\theta(\pi, f)$ the smallest residual capacity (smallest capacity among the relevant

edges in the residual graph) along π in G_f.

Algorithm 14: Finding an Augmenting Path

```
Aug(π, f)
    b ← θ(π, f)
    foreach e = uv⃗ ∈ π do
        if e is a forward edge then
            increase f(uv⃗) in G by b
        else
            decrease f(vu⃗) in G by b
```

Algorithm 15: Ford and Fulkerson's Algorithm

```
for e ∈ E do
    f(e) ← 0
Compute G_f
while there is an s-t path π in G_f do
    f' ← Aug(π, f)
    f ← f'
    Compute G_f
return f
```

Algorithm 15 finds a flow with the maximum flow value in the network!
There are a couple of questions we must answer.

- Why does the algorithm terminate?
- Why is the algorithm even correct in the first place?
- What is the time complexity?
- Does it matter which s-t path we choose in the residual graph?
- Could we extend this to the case where capacities are not integers?

Also, what is the relationship between max-flow and min-cut?

Termination. Note that at each step of the algorithm, the new flow f' satisfies $|f'| = |f| + \theta(\pi, f)$. Since the capacities are always integral (even in the residual graph), the flow value increases by at least 1 at each step. It is also easy to see that the maximum flow is at most $C_{\max}$, which is the sum of the capacity of every edge.
Therefore, the algorithm terminates.

Lemma 2.3. The running time of the algorithm is bounded by $\mathcal{O}(C_{\max}|E|)$.

Proof. In each loop, we require $\mathcal{O}(m)$ time to maintain the residual graph. The time to find the s-t path π takes $\mathcal{O}(m+n)$ and augmenting takes $\mathcal{O}(n)$ (there are at most $n-1$ vertices on π). Since the loop runs for at most $C_{\max}$ iterations, the bound follows.
In fact, we can bound it better as $\mathcal{O}(f|E|)$, where f is the maximal flow value. ■

Lemma 2.4. The Ford and Fulkerson algorithm returns a flow with maximum value.

Correctness. Denote by $\mathfrak{f}$ be the flow returned by the Ford-Fulkerson algorithm. Let

$$A = \{v \in V : \text{there is a path to } v \text{ from } s \text{ in } G_{\mathfrak{f}}\}$$

and $B = V \setminus A$. Observe that $s \in A$ and $t \in B$. Further, when the algorithm terminates, there is no path from s to t in $G_{\mathfrak{f}}$. Since the two sets are disjoint, (A, B) is a cut on G (this makes sense because the two graphs have the same vertex set).

To show optimality, it suffices to show (using Lemma 2.2) that $\mathfrak{f}$ saturates every edge from A to B and avoids every edge from B to A.

- Suppose instead that there is an edge $\overrightarrow{uv} \in E$ such that $u \in A$, $v \in B$, and $\mathfrak{f}(\overrightarrow{uv}) < c(\overrightarrow{uv})$. However, then, $\overrightarrow{uv}$ is a forward edge in $G_{\mathfrak{f}}$, so $v \in A$. This is a contradiction and therefore, $\mathfrak{f}$ saturates every edge from A to B.
- On the other hand, suppose there is an edge $\overrightarrow{uv} \in E$ with $u \in B$ and $v \in A$ with $\mathfrak{f}(\overrightarrow{uv}) > 0$. Then $\overrightarrow{vu}$ will be a backward edge in $G_{\mathfrak{f}}$, so $u \in A$. This is again a contradiction, so $\mathfrak{f}$ saturates every edge from B to A.

Our analysis shows that the (A, B) cut has capacity equal to $|\mathfrak{f}|$. Therefore, $|\mathfrak{f}|$ is the maximal flow in G and further, $\texttt{cap}(A, B)$ is the minimum capacity cut in G (this follows from the fact that we have attained this flow value, so it cannot exceed the capacity of any cut).

Importance of chosen paths. The chosen path turns out to be quite important. An example to highlight this fact can be found here.

Non-integral costs. It is not too difficult to show that there is no problem if the costs are rational – the flow increases by at least $1/q$, where q is the maximum denominator of any cost when written as a fraction.
However, it need not terminate when the costs are irrational. Further, it need not even converge to the correct value.
A standard counterexample to show this can be found here.
However, there have been far more efficient algorithms for determining max-flow.
The Edmond-Karp algorithm augments along the path π with largest $\theta(\pi, f)$. This in fact runs in $\mathcal{O}(|E|^2 \log |E| \log |f|)$.
Augmenting along the shortest path (in terms of number of edges) gives rise to an algorithm that runs in $\mathcal{O}(|V||E|^2)$.
Following this, there were numerous algorithms. Most recently, the "compact networks" method proposed by Orlin in 2012 runs in $\mathcal{O}(|V||E|)$ using dynamic trees.
It is possible to get much faster algorithms if the capacities are all unit. For example, there exists an $\mathcal{O}(\min\{|V|^{2/3}, |E|^{1/2}\}|E|)$ algorithm.

2.3. Applications

Let us look at some applications of the max-flow min-cut theorem, which states that the value of the max-flow is equal to the capacity of the min-cut.

2.3.1. Edge Disjoint Paths

Let $G = (V, E)$ be a directed graph and $s, t \in V$ be designated vertices. Two s-t paths π and π' are said to be *edge disjoint* if they do not share any edges.
The edge disjoint paths problem asks to determine the maximum number of disjoint paths between the source and sink nodes s and t in a directed graph G.
We claim that this quantity is equal to the maximum flow in the graph if each edge is assigned a capacity of 1.

Lemma 2.5. The max flow value of G is at least k iff G has k edge disjoint paths.
It follows that the max flow value of G is k iff the maximum number of edge disjoint paths in G is k.

Proof. Suppose G has k edge disjoint paths. Consider the flow that sends a flow of 1 along each of these k disjoint paths from s to t ($f(e) = 1$ if e belongs to one of the k paths). This gives a flow of value k.

On the other hand, suppose the max flow value of G is k. Then, there is an integral flow f of value k in G (the flow output by the Ford and Fulkerson algorithm).
By tracing out the edges of flow 1, we can generate k edge disjoint paths. We prove this by induction on the number of edges that carry non-zero flow on them.[5]
If there are 0 edges, then there are no disjoint paths and there is nothing to prove.
Say there are $t > 0$ edges with non-zero flow. Then, let π be an acyclic path from s to t that walks along 1-edges (why does such a path exist?). Consider the flow f' that is the same as f except that $f(e) = 0$ for any $e \in \pi$. Now,

[5] More precisely, we are proving that given any flow in the graph, there is a decomposition of the edges with non-zero flow into a set of edge-disjoint paths.

since $|f'| < |f| = t$, we may consider the resulting edge disjoint paths. We wish to show that π and any of these paths are disjoint. This is trivial however, by definition, and the claim follows. ■

2.3.2. Network Connectivity

Let $G = (V, E)$ be a directed graph and s, t be the source and sink. We say that $F \subseteq E$ *disconnects* s from t if the removal of F from the graph disconnects t and s.
The network connectivity problem asks to determine what the smallest $F \subseteq E$ that disconnects s from t is.

Theorem 2.6 (Menger's Theorem)**.** The maximum number of edge-disjoint s-t paths in a graph is equal to the minimum number of edges whose removal disconnects t from s.

Proof. Suppose the graph has capacities as defined in Section 2.3.1. The first quantity, as we have seen, is equal to the max-flow. The quantity on the right on the other hand, is equal to the capacity of the min-cut (by definition). The required follows. ■

2.3.3. Maximum Bipartite Matching

Given an undirected bipartite graph $G = (V, E)$ with $V = X \cup Y$ such all edges are between X and Y, the problem asks to find a largest set $M \subseteq E$ such that for any $e, e' \in M$, e, e' do not share any common vertex.
This can be formulated as a max-flow problem. Create a directed graph $G' = (V', E')$ with vertex set $V' = V \cup \{s, t\}$ for some fresh vertices s, t and edge set

$$E' = \{\overrightarrow{xy} : x \in X, y \in Y, xy \in E\} \cup \{\overrightarrow{sx} : x \in X\} \cup \{\overrightarrow{yt} : y \in Y\}.$$

Further, assign a capacity of 1 to each edge.
The maximum bipartite matching is then just the value of a max-flow from s to t!
Before getting to the proof of the above, we give the following useful lemma.

Lemma 2.7. Let f be a flow in the flow network G with source s and sink t. Let (S, T) be a cut in G. Then the net flow across the cut, that is, the flow leaving S minus the flow entering S, equals $|f|$.

Proof. This is proved as

$$\begin{aligned} |f| &= f^{\rightarrow}(s) + \sum_{v \in S \setminus \{s\}} f^{\rightarrow}(v) - f^{\leftarrow}(v) \\ &= \sum_{v \in S} f^{\rightarrow}(v) - \sum_{v \in S \setminus \{s\}} f^{\leftarrow}(v) \\ &= f^{\rightarrow}(S) - f^{\leftarrow}(S). \end{aligned}$$

■

Lemma 2.8. Let G be a bipartite graph and G' be the flow network defined above. Then,

- if M is a matching in G, there is an integer valued flow f in G' such that $|f| = |M|$.
- if f is an integer valued flow in G', there is a matching M in G such that $|M| = |f|$.

In particular, the max-flow value of G' is equal to the size of a maximum matching of G.

Proof. The first part is direct, since we can construct a flow f such that for each edge $e = \overrightarrow{xy} \in M$ with $x \in X$ and $y \in Y$, $f(\overrightarrow{sx}) = f(\overrightarrow{xy}) = f(\overrightarrow{yt}) = 1$ and $f(e) = 0$ for the remaining edges e. The fact that f is a flow follows from the fact that M is a matching, so the conservation constraints are satisfied. The capacity constraints are trivially satisfied. Moreover, the value of f is equal to the cardinality of M, that is, $|f| = |M|$.

Let us now prove the other direction. Clearly, for each $e \in E'$, $f(e)$ is 0 or 1. Define

$$M = \{e \in E' \cap E : f(e) = 1\}.$$

This is just the set of edges from X to Y that carry non-zero flow.
Observe that because f is a flow, each vertex in X or Y is present in at most one edge in M. Indeed, any vertex

in X (resp. Y) has an incoming (resp. outgoing) flow of at most 1 and if it is present in more than one edge, the outgoing (resp. incoming) flow would be greater than 1, contradicting the conservation constraint.
Therefore, M is a matching. Using Lemma 2.7 on the cut $(X \cup \{s\}, Y \cup \{t\})$, $|f| = f^{\rightarrow}(S) - f^{\leftarrow}(S) = |M|$, proving the required. ■

The above could have been proved in a more direct way using Lemma 2.5.

2.3.4. Baseball Elimination

The last problem we discuss is the following (referred to as IPL Elimination in the course slides).

Suppose we have n teams. Each team has a certain number of wins w_i initially, and for each $\{x, y\}$, there is a quantity $r_{xy} \geq 0$ which represents the number of games teams x and y are left to play against each other ($r_{xy} = r_{yx}$ and $r_{xx} = 0$). Is it possible for a certain team z to get first position after all the games are done (possibly being tied with other teams)? Equivalently, if all teams except those which are at first position are eliminated, is z necessarily eliminated?

We shall formulate this as a max-flow min-cut problem, creating a flow network G_z. First, let $g_z^* = \sum_{x,y \in X \setminus \{z\}} g_{x,y}$. We show that G_z has a flow of value strictly less than g_z^* iff z is eliminated. We must also argue that G_z can be obtained from the input in polynomial time.

Let $X' = X \setminus \{z\}$. G_z has vertex set

$$V = \{s, t\} \cup \{u_{xy} : \{x, y\} \subseteq X'\} \cup \{u_x : x \in X'\},$$

where s and t are the source and sink respectively. G_z has edge set

$$E = \{\overrightarrow{su_{xy}} : \{x, y\} \in X'\} \cup \{\overrightarrow{u_x t} : x \in X'\} \cup \bigcup_{\{x,y\} \subseteq X'} \{\overrightarrow{u_{xy}u_x}, \overrightarrow{u_{xy}u_y}\}.$$

In the definition of the edge set, we refer to these three sets involved as E_{source}, E_{sink}, and E_{mid} respectively.
The capacity function c is defined as

$$c(e) = \begin{cases} g_{xy}, & e \in E_{\text{source}}, \\ \infty, & e \in E_{\text{mid}}, \\ m - w_x, & e \in E_{\text{sink}}, \end{cases}$$

where

$$m = w_z + \sum_{x \in X'} g_{zx}.$$

is the total number of games z can have won at the end.
In case any of the capacities are negative, then the problem is trivial anyway since $m_x > m$ for some x, so z cannot win.

Lemma 2.9. G_z has a flow strictly less than g_z^* iff z is necessarily eliminated.

Proof. Suppose G_z has a flow of value g_z^*. This means that it is possible to obtain outcomes where all the other teams win at most m games (totally). This means that z is not eliminated.
On the other hand, suppose z is not eliminated. Using the outcomes of the games, it is possible to set the flows in E_{sink} such that no flow exceeds $m - w_u$ for any u at the end of all the games. This achieves a flow of value g_z^*. ■

While not related directly to the algorithm given above, the following lemma, which can be proved from the max-flow min-cut theorem, is interesting.

Lemma 2.10. Suppose team z is necessarily eliminated. Then the following hold:

- z can finish with at most $m = w_z + \sum_{x \in X \setminus \{z\}} g_{zx}$ wins.
- There exists a set $Q \subseteq X$ such that

$$\sum_{x \in Q} w_x + \sum_{\{x,y\} \subseteq Q} g_{xy} \geq m|Q|.$$

§3. NP Hardness and Reductions

Suppose we have n jobs that take time $S_1, \ldots, S_n$ each and two processors P_1 and P_2. We wish to schedule the jobs on these two processors such that the overall time taken (the time when the last job completes) is minimized. How do we do this?

One idea is that we initially schedule S_1 and S_2 and when one of the jobs ends, we schedule the next job S_3 on the processor that is freed. We repeat this, scheduling jobs whenever processors become free.
This does not work – consider the set of times $\{1, 1, 100\}$. Our scheduling produces time 101 (we schedule the first two jobs first), but it is quite easy to see that it can be done in time 100.

The issue seems to be that the ordering of the jobs is problematic. So instead, schedule the jobs in non-increasing order and repeat the above algorithm. Does this work?
No, it does not. Consider the set of jobs $\{3, 3, 2, 2, 2\}$ – our algorithm gives a time of 7 but a time of 6 can be attained.

It in fact turns out that it is impossible (or at least extremely hard) to get an efficient (polynomial time) algorithm for this. It is quite surprising that we solved the interval-scheduling problem without too much difficulty, yet this is incredibly hard.
How do we show that this problem is "hard" though? What does "hard" even mean?

3.1. Overview and some Definitions

Definition 3.1. A problem Π is said to be in the class P if there exists an algorithm $\mathcal{A}$ such that for any input x, $\mathcal{A}$ finds the correct solution for x in ttime $\mathsf{poly}(|x|)$ time.

Searching, sorting, interval scheduling, primality testing, max-flow, finding a perfect matching in a bipartite graph are just a few examples of important problems in P.
Before we get to the definition of NP, let us start with a few examples.

The 3-colorability problem. Given a graph $G = (V, E)$, check if there exists a 3-coloring of G – a function $f : V \to \{1, 2, 3\}$ such that for any $e = uv \in E$, $f(u) \neq f(v)$.
One can think a bit and struggle to come up with an algorithm for the above. Now, suppose someone gives you a $c : V \to \{1, 2, 3\}$. Then in polynomial time, one can test whether or not this is a valid 3-colouring (by checking the colours of the two vertices on every edge). That is, while it is difficult to come up with a solution for the algorithm, it is easy to test whether a given input is a solution or not.

The k-clique problem. Given a graph $G = (V, E)$, does G have a clique of size k?
A brute-force algorithm would require $\mathcal{O}(n^k)$ time, which *is* polynomial for a fixed k. It is not polynomial if we have, say, $k = \sqrt{n}$ however.
As in the previous example, for a given subset $S \subseteq V$, we can easily test whether or not it is a k-clique (by checking that it has cardinality k and if there is an edge between any two vertices in S).

The satisfiability problem. Given a CNF formula φ over variables $x_1, \ldots, x_n$, does there exist an assignment $\tilde{a}$ such that $\tilde{a}$ satisfies φ?
If φ is satisfiable, then there exists an assignment $\tilde{a}$ that "witnesses" this. As in the previous examples, it is difficult to come up with an algorithm that finds a satisfying assignment (or determine satisfiability). It is easy to check if a given assignment is satisfying, however.
This sort of property where checking is easy is what leads to the definition of NP.

Definition 3.2. A problem Π is said to be in NP if there is a polynomial time algorithm $\mathcal{T}$ such that

- If input x is a positive instance of Π then there is a polynomial length proof y such that $\mathcal{T}$ on inputs x, y outputs `Yes`.
- If input x is a negative instance of Π then for any polynomial length proof y such that $\mathcal{T}(x, y)$ outputs `No`.

For example, in the 3-colorability problem, x would be the graph $G = (V, E)$, y would be a function $c : V \to \{1, 2, 3\}$, and $\mathcal{T}$ would be the algorithm that checks if c is a 3-coloring.
The "proof" just serves as the "witness" we mentioned earlier – an example that proves correctness.

Note that the problem Π itself is just a yes-no problem. The input x is a positive instance if the answer to Π for x is yes and a negative instance if the answer is no.

The first important observation to make is that any problem in P is a problem in NP. For any y, we can take $\mathcal{T}$ as the algorithm that just solves the problem and answers yes or no. We ignore the proof and merely determine if x is a positive or negative instance.
Therefore,

$$\mathsf{P} \subseteq \mathsf{NP}.$$

What if the problem itself is not a yes-no problem – perhaps a search or optimization problem?
A problem where the answer is just yes or no is referred to as a *decision problem*. For example, 3-colorability and primality testing.
In a *search problem*, we are asked to find a solution. For example, finding a path between a pair of vertices, sorting, and finding a 3-coloring.
In an *optimization problem*, we are asked to find a "best" solution with respect to some optimization parameter. For example, max-flow, min-cut, longest common subsequence.

Any search problem has a decision problem variant. If a search problem asks to find a specific solution, the decision problema asks whether there is a solution.
For example, finding a 3-colouring changes to asking whether a 3-coloring asks.
Observe that if we can solve a search problem, we can solve its decision variant as well.

Similarly, suppose we are given an optimization problem that asks to find a solution that maximizes/minimizes some parameter. Then the decision variant asks if there is a solution that has parameter at least/at most v?
For example, finding a max-flow changes to asking if there is a flow of value at least v.
If we can solve an optimization problem, we can solve its decision variant as well.

Now, consider the decision variant of the scheduling problem we discussed earlier – given $d_1, \ldots, d_n$ and T, does there exist a scheduling on two processors such that the total time is at most T? It is easy to see that this problem is in NP. Given a scheduling (a proof), we can just find the total time it takes and thus verify if this is at most T.

3.2. Reduction

Reduction essentially means using other algorithms for problems we wish to solve. For example, consider the problem of determining the existence of a perfect matching in a graph $G = (V, E)$.
On the other hand, consider the problem that finds a maximum matching in a graph $G = (V, E)$.
It is not too difficult to see that the first problem can be "reduced" to the second – if we find a maximum matching in the graph, we can check whether it has $|V|/2$ vertices and thus solve the first problem.
Can we come up with a reduction that goes the other way round? Before doing this in fact, can we reduce the problem of finding the *size* of a maximum matching in a graph to the first problem? Yes, we can.

- If G has a perfect matching, then return $|V|/2$.

- Let G_1 be a graph with vertex set $V \cup \{v\}$, where v is a fresh vertex that has an edge to every vertex in V. If G_1 has a perfect matching, then G has a matching of size $(|V| - 1)/2$ (Why?), which must be a maximum matching if this is the case.
- Continuing on, create G_{i+1} by adding a new vertex that is connected to every vertex of G_i. Then if k is the minimum index such that G_k has a perfect matching, G has a matching of size $(|V| - k)/2$ – there cannot be any edges within the new vertices in a perfect matching because this would imply the existence of a perfect matching of G_i for some $i < k$ (Why?). Further, this is a maximum matching. If there was a matching of size $(|V| - k + 2)/2$, then G_{k-2} must have a matching (Why?).

Since i can go up to n, we are calling the perfect matching algorithm only n times. This can be decreased to $\log n$ by using a binary search-like scheme.
Now that we have done this, the original problem is not too difficult. Suppose the size of a maximum matching is l. We can remove an edge and ask if the remaining graph has a matching of size l (that is, if the size of a maximum matching is still equal to l). If no, we discard the edge. The remaining graph will just be a matching, which gives us a maximum matching in the original graph.

3.3. Coping with NP-hardness

How do we deal with hard problems?

1. Come up with some heuristics. There are no provable guarantees, but they work somewhat well in practice.
2. There may be efficient algorithms for some special cases.
3. Better than brute force algorithms, where we come up with algorithms that are not polynomial, but better than a naïve brute-force algorithm.
4. Approximation algorithms for optimization problems, where we merely want to approximate the answer. For example, suppose we have a graph such that the size of the largest independent set is OPT. Then in an approximation algorithm, we should be able to find an independent set that is of size not less that OPT/c for some $c \geq 1$ (this is known as a c-approximation algorithm). Similarly, for a minimization problem, we would find a solution of size not more than $c \cdot \mathsf{OPT}$.

First, let us look at an approximation problem for the scheduling problem.
Given m processors $p_1, \ldots, p_m$ and n jobs $j_1, \ldots, j_n$ with durations $d_1, \ldots, d_n$, find a scheduling for these jobs that minimises the total completion time.
Let $\mathcal{S}$ be a schedule such that

- A_i is the set of jobs scheduled on processor $i \in [m]$.
- $T_i = \sum_{j \in A_i} d_j$.
- The *completion time* is then $\max_{i \in [m]} T_i$.

We want to minimize the completion time. Let $\mathcal{S}$ be a scheduling that does so, and let its completion time be T^*.
While the optimization problem itself is NP-complete, the approximation version has a polynomial time algorithm.
Consider the algorithm that schedules the jth job on the machine with the smallest load so far (with the jobs initially arranged in any arbitrary order), and T be the completion time achieved by this schedule.

Lemma 3.1. With the above notation, $T \leq 2 \cdot T^*$, that is, the above described algorithm gives a 2-approximation.

Proof. Let $T' = \frac{1}{2}\left(\max_j d_j + \frac{1}{m}\sum_{j \in [n]} d_j\right)$. We trivially have $T' \leq T^*$, since T^* is greater than each of the two quantities involved in the average.
Choose $i \in [m]$ such that $T = T_i$, and let $j \in [n]$ be the last job scheduled on it. Then for any $i' \neq i$, when j was scheduled on i, $T_{i'} \geq T_i - d_j$.
Therefore, $\sum_{k \in [n]} T_k \geq m(T_i - d_j)$. That is,

$$T = T_i \leq d_j + \frac{1}{m}\sum_{k \in [n]} T_k \leq 2T' \leq 2T^*,$$

proving the lemma. ■

If we arrange the jobs in non-increasing order of durations, then the above algorithm gives a (3/2)-approximation. With this line of thought, one might ask whether there exist questions that are not only hard to solve (exactly), but also hard to approximate? There is extensive literature on this side of things as well, but we do not look at these in this course.

References

[AKS02] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Annals of Mathematics*, 160, 09 2002.