CS 218: Design and Analysis of Algorithms

Amit Rajaraman

Last updated February 3, 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 $c \geq 0$ 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 $c \geq 0$ 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 $c_1, c_2 \geq 0$ 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, which we shall see later (you can try thinking about it now).

[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

```
1 max ← A[0]
2 for i = 1 to n − 1 do
3     if A[i] > max then
4         max ← A[i]
5 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

```
1 i1, i2 ← 0, 1
2 min ← (x1 − x2)^2 + (y1 − y2)^2
3 for i = 1 to n do
4     for j = i + 1 to n do
5         d ← (xi − xj)^2 + (yi − yj)^2
6         if d < min then
7             min ← d
8             i1, j1 ← i, j
9 return i1, i2
```

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, then a polynomial time algorithm (in n) is not known. 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's quite clear that greedy algorithms needn't always work, we have to 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 the 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, obviously 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 see 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 1.1. 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 $|\mathcal{A}| = |\mathsf{OPT}|$.
What is the running time of this algorithm? We first sort the jobs according to their finish times, which takes $\mathcal{O}(n \log n)$ time. We then have to scan through all the jobs in this sorted list, which takes $\mathcal{O}(n)$ times. The total running time is $\mathcal{O}(n \log n)$.

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 such cost.
It is easy to show that this T must be a tree. Suppose 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 no matter what greedy strategy we choose, the optimal solution is attained. We give four such algorithms.

(i) Choose an edge α such that $c(\alpha)$ is minimal. Each subsequent edge is chosen from the cheapest remaining edges of G ensuring that we never 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 $x_1, \ldots, x_k$ an 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
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.

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
```

Lemma 1.2 (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.

Also note that the third algorithm given above aises 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 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 a, b, c, d, e occur with frequencies $100, 3, 5, 45, 20$ respectively. Then using 3 bits for each, it would result in 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 leaves lower down to a higher position and decrease the cost.

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
```

Lemma 1.3. Consider the two letters x and y with the smallest frequencies. There is an optimal code tree T^* in which these tw 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 $f_a \leq f_b$.
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

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

We may further assume that there is a strict inequality in the first step (if there was not, we are done), thus resulting in a contradiction. ■

We shall next show that if the first step is optimal, the next step is optimal. That is,

Lemma 1.4. 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'.

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 1.5 (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,

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

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 $\mathcal{O}(C(r)n^{\log_r(2r-1)})$.
Later, Schönhage and Strassen managed to breach the $n^{1+\varepsilon}$ barrier and got it down to $\mathcal{O}(n \log n \log\log n)$.
Martin Fürer got it down to $\mathcal{O}(n \log n 2^{\log^* n})$ in 2007.
In 2019, Harvey and Hoeven managed to get the "perfect" method which takes $\mathcal{O}(n \log n)$, thus attaining the bound in a result that says that this is optimal.

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

Suppose we split it into two halves based on their x-coordinates. Recursively compute the closest pair in each of the two halves. Let the minimum of these two distances is 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 1.6. Let S_y be the points in the distance d region sorted in decreasing order of their y coordinates. Let these points be $(q_1, \ldots, q_m)$. We claim that if the distance between q_i and q_j is less than d, then $j - i \leq 15$.

1.4. 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 7. We encourage the reader to prove the correctness of this algorithm.

Algorithm 7: Solution 1.1

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

```
1  greedyStep(B)
2      if size(B) = 1 then
3          return B[0]
4      mid ← size(B)/2
5      if B[mid] > B[mid + 1] and B[mid] > B[mid − 1] then
6          return B[mid]
7      else if B[mid] ≤ B[mid − 1] then
8          return greedyStep(B[1 : mid/2])
9      else if B[mid] ≤ B[mid + 1] then
10         return greedyStep(B[1 + mid/2 : size(B)])
11 return greedyStep(A)
```

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

Algorithm 8: Exercise 1.2

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

$M \leftarrow \varnothing$. Let $V(M)$ be the set of vertices in M.
foreach $uv \in E$ **do**
  if $V(M) \cap \{u, v\} = \varnothing$ **then**
    $M \leftarrow M \cup \{uv\}$
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 9.

(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 $\sum_{i\in[n]} f_i c_i \leq C$.

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

Algorithm 9: 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
```

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 10. Correctness is easily proved.

Algorithm 10: 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

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

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 $Q_\sigma \leq Q$, then for every $1 \leq i \leq k$, $1 \leq \sigma(i) \leq k$. This implies that the first k positions permute among themselves in any optimal solution.
Suppose otherwise and let $r = \max\{j \in [n] : \sigma(j) = i \text{ for some } 1 \leq i \leq k\} > k$. Then,

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

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

order, $t_2(\sigma(k)) \geq t_2(k)$. Then

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

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 $a_i \geq a_k$.
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 $r - 1 \geq 0$. 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).

References

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