CS 218: Design and Analysis of Algorithms

Amit Rajaraman

Last updated January 18, 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
1 flag ← true
2 for $i = 2$ to $\sqrt{n}$ do
3     if $i$ divides $n$ then
4         flag ← false
5 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 abused to say, for example, that $\sqrt{n} = \mathcal{O}(n)$ (instead of the correct $\sqrt{n} \in \mathcal{O}(n)$. We also write $\sqrt{n} = 2^{\mathcal{O}(\log n)}$ sometimes, which means that there is some $f(n) \in \mathcal{O}(\log n)$ such that .

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
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
$i_1, i_2 \leftarrow 0, 1$
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
            $i_1, j_1 \leftarrow i, j$
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, then a polynomial time algorithm (in n) is not known. Indeed, this is an "NP-hard" problem, which we shall read more about later.

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 1.1

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 4. We encourage the reader to prove the correctness of this algorithm.

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.

Algorithm 4: Algorithm to find a maxima in an array

```
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 $\leftarrow$ size(B)$/2$
    if B[mid] $>$ B[mid $+ 1$] and B[mid] $>$ B[mid $- 1$] then
        return B[mid]
    else if B[mid] $\leq$ B[mid $- 1$] then
        return greedyStep(B[$1$ : mid$/2$])
    else if B[mid] $\leq$ B[mid $+ 1$] then
        return greedyStep(B[$1 +$ mid$/2$ : size(B)])
return greedyStep(A)
```

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

Algorithm 5: Algorithm to find a minimal spanning subgraph

```
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 $G$
1 sort($E$,$c$)
                                        // sort the elements of E in non-decreasing cost
2 $T \leftarrow \varnothing$
3 for $1 \leq i \leq m$ do
4     if $T \cup \{e_i\}$ doesn't have a cycle then
5         $T \leftarrow T \cup \{e_i\}$
6 return $T$
```

The set thus formed clearly has no cycles by construction. Connectedness 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 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. Let 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 already know that e must be lowest cost such edge. But by the cut property, e must be present in T!

Proof. ■

References

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