Name: Alimulla Shaik
Email: shal5122@colorado.edu
Algorithms Problem Set 1
Collaborators: Shubham Mudgal, Sunil Bn, Praveen D

1) Sol.
To prove that we can design algorithm which runs with running time O(min(f(n), g(n))), let's take f(n) has running time of n^5 and g(n) has running time of n^3
Now, we have to design an algorithm with running time $O(\min(n^5, n^3)) = O(n^3)$
Which is running time of g(n). To generalize this, an algorithm with O(min(f(n), g(n))) can be designed using f(n) or g(n) functions based on which one is minimum.

Algorithm:
1. Consider functions f(n), g(n) and say h(n) is desired function.
2. For each $\{n_1 \varepsilon$ f(n), g(n)$\}$, compute $\min(f(n_1), g(n_1))$ and $h(n) = \min(f(n_1), g(n_1))$.

Yes, we can achieve running time exactly min(f(n), g(n)) which is explained with example above. But in some case it depends on functions f(n), g(n).

2) Show (prove) that for any real constants a and b, where b > 0, the asymptotic relation $(n + a)^b = (n)^b$ is true.

Sol. From the Θ() definition, for any real constant a, there exist
Θ(n) = n + a (constants don't matter).
Therefore, Ω(n) = n + a and O(n) = n + a.

From this we can get,
$k_1 n \le (n + a)$, for $n \ge n_1$ and $(n + a) \le k_2 n$, for $n \ge n_2$
Here k_1, k_2, n_1 and n_2 are constants.

Therefore,
$k_1 n \le (n + a) \le k_2 n$, $n \ge n_0$ Here n_0 is constant.

In the problem, it is given that b is real constant and b>0, so we can take all to power b without affecting equation.
we will get:
$(k_1 n)^b \le (n + a)^b \le (k_2 n)^b$, $n \ge n_0$
$(k_1)^b (n)^b \le (n + a)^b \le (k_2)^b (n)^b$, $n \ge n_0$
$k_3 (n)^b \le (n + a)^b \le k_4 (n)^b$, $n \ge n_0$ Here k_3, k_4 are some constants.

If we take $k_3 (n)^b \le (n + a)^b$, $n \ge n_0$ then we will get $(n + a)^b = \Omega(n^b)$
If we take $(n + a)^b \le k_4 (n)^b$, $n \ge n_0$ then we will get $(n + a)^b = O(n^b)$

Hence, $(n + a)^b = \Theta(n^b)$.

3) Sort the following functions by order of asymptotic growth such that the nal arrangement of functions g1, g2,....g12 satisfies the ordering constraint g1 = Ω(g2), g2 = Ω(g3),... , g11 = Ω(g12). Give the nal sorted list and identify which pair(s) of functions f(n), g(n), if any, are in the same equivalence class, i.e., f(n) = Θ(g(n)).

Sol. We are asked to list functions to satisfy g1 = Ω(g2),g2 = Ω(g3),.... g11 = Ω(g12) which means that we need to sort and list functions g1, g2, g3… g12 in decreasing order.

I have considered some values of n and observed below order of growth.
factorial functions > exponential functions > polynomial functions > poly logarithmic functions.

Please find the order below which satisfies required constraint and the functions in the same class are listed together.

n! > e^n > $(3/2)^n$ > (lg n)! > n^2 > n lg n, lg(n!) > n > $(\sqrt{2})^{\lg n}$ > $2^{lg^* n}$ > $n^{1/\lg n}$, 1

Below functions are in the same equivalence class.
n lg n, lg(n!)
$n^{1/\lg n}$, 1

4) Solve the following recurrence relations. Show all your work.
a) T(n) = T(n-1) + n, T(1) = 1

Sol. We are given relation, T(n) = T(n-1) + n,
we can find T(n-1) by replacing n with n-1 in given relation.
⇒ $T(n-1) = T(n-2) + (n-1)$
Substitute T(n-1) value in given relation ⇒ T(n) = T(n-2)+(n-1)+n
Substitute T(n-2) value in given relation ⇒ T (n) = T(n-3)+(n-2)+(n-1)+n
It goes on…
let's consider relation after k^{th} iteration (k is some constant)
⇒ $T(n) = T(n-k) + (n-k+1) + (n-k+2) + ... + (n-2) + (n-1) + n$
This relation goes till T(0)
so n-k = 0 ⇒ $n = k$ substitute this in T(n),
then T(n) = T(0)+1+2+...+(k-2)+(k-1)+k
from the relation, T(0) = 0 ⇒ $T(n) = 1 + 2$ +...+(k-2)+(k-1)+k
⇒ $T(n) = n(n+1)/2$
⇒ T(n) = (½)(n^2+n)
⇒ $T(n) =$ O(n^2)

b) T(n) = 2T(n/2) + n^3, T(1) = 1 (assuming n is a power of 2)

Sol. We are given the relation, T(n) = 2T(n/2) + n^3
We can find T(n/2) by replacing n with n/2 in given relation.

$\Rightarrow T(n/2) = 2T(n/2^2) + (n/2)^3$

Substitute T(n/2) value in given relation $\Rightarrow T(n) = n^3 + 2[2T(n/2^2) + (n/2)^3]$

$\Rightarrow T(n) = n^3 + (n^3)/2^2 + 2^2T(n/2^2)$

Substitute $T(n/2^2)$ value in given relation $\Rightarrow T(n) = n^3 + (n^3)/2^2 + 2^2[2T(n/2^3)+(n/2^2)^3]$

$\Rightarrow T(n) = n^3 + (n^3)/2^2 + (n^3)/2^4 + 2^3T(n/2^3)$

Substitute $T(n/2^3)$ value in given relation $\Rightarrow T(n) = n^3 + (n^3)/2^2 + (n^3)/2^4 + 2^3[2T(n/2^4) + (n/2^3)^3]$

$\Rightarrow T(n) = n^3 + (n^3)/2^2 + (n^3)/2^4 + (n^3)/2^6 + 2^4T(n/2^4)$

It goes on…

for k iteration $\Rightarrow T(n) = n^3 + (n^3)/2^2 + (n^3)/2^4 + (n^3)/2^6 + 2^4T(n/2^4) + ... + 2^kT(n/2^k)$

This relation goes on till T(1) so making $n/2^k = 1 \Rightarrow k = \log n$

$\Rightarrow T(n) = n^3 + (n^3)/2^2 + (n^3)/2^4 + (n^3)/2^6 + 2^4T(n/2^4) + ... + (n^3)(1/2^2)^{\log n-1} + 2^{\log n}$

$\Rightarrow T(n) = (n^3) \sum_{k=0}^{\log n-1} (1/2^2)^k + n$

$\Rightarrow T(n) = (n^3)\ \Theta(1) + n$

$\Rightarrow T(n) = \Theta(n^3)$

5) a) Show that if an array is nearly sorted, in the sense that only k elements array are out of order for some constant k, then insertion sort runs in O(n) time.

Sol. In insertion sort on n elements, we will scan through all n elements, and will move each element an average of n/2 positions. Which means we will perform approximately n*n/2 operations.

So, complexity of Insertion Sort is $T(n) = O(N^2)$

If an array is nearly sorted and only k elements of array are out of order then the required number of operations are approximately n*k/2.

But here k is a constant, so k/2 term will be ignored in a big-O complexity.

Hence, Complexity of insertion sort in this case is T(n) = O(N).

b) Suppose you are given a guarantee that every entry in an array is a positive integer, and at most some constant k. Give an O(n) algorithm to sort such arrays.

Sol.

Algorithm:

1. Consider HashTable HT(0..k) with at most ‘k’ elements and initialize it to 0.
2. For each element i $\varepsilon\{0,...k\}$ store 1 in hash table in corresponding i^{th} position HT(i) =1.
3. Traverse all elements of hash table and only list the elements with value 1.

Correctness:

It is given that every element in array is a positive integers and array has at most k elements. So we can make use of hash table to sort the elements of array.

Let's consider a hash table with at most 'k' elements. We will initialize the hash table with 0. We will traverse all the elements of the array and for each i $\varepsilon\{0,...k\}$ keep 1 in hash table in corresponding ith position. This will ensure that all the elements in hash table have either 0 or 1 value. If we start listing elements with value 1 in hash table, it produces elements in sorted order.

Running Time:
For this algorithm we traversed all elements of the array twice, one to store value 1 in hash table for element i $\varepsilon\{0,...k\}$ in corresponding ith position and to list elements with value 1 in hash table.
So complexity of the algorithm is T(n) = n + n
$\Rightarrow T(n)$ = 2n
$\Rightarrow T(n)$ = O(n)

c) Wait a minute, why doesn't (b) contradict the Ω(n log n) lower bound given on page 59 of the textbook?

Sol. Ω(n log n) lower bound deals with comparisons while sorting array. There is an argument in the textbook, that any comparison tree that sorts n elements must make, in the worst case, Ω(n log n) comparisons. Well, this argument applies only to algorithms that use comparisons.
Here we are using hash table to sort the elements in the array without any comparisons. And we are achieving O(n) running time. So this will not contradict the argument made in the text book.

6) a)
Sol. Let's assume G' be the number of good minions then N-G' will be the number of bad minions. And also, assume N-G' >= G'.
From this assumption it is clear that we can find set G of good and set B of bad minions of equal size G'.
Now, assume that the set B of bad minions conspire to fool the Gru, then
For any test made by the Gru, they report that they are "good" and the minions in G are "bad". Here the minions in G always tell the truth. So they report that they are "good" and the minions in B are "bad".
From this, whatever is the strategy based on the kind of test used by Gru, the lab results will be the same and there won't be any difference in the behaviour of minions in G in comparison with minions in B. This does not allow the Gru to determine which minions are good.

b)
Sol. Assume n is even. We can consider all minions in pairs and test all these pairs. We need to ignore all pairs of minions that do not say 'good, good' and finally we take a minion from each pair. All the pairs that we have discarded contain at least a bad minion. So we are still keeping more good minions than bad ones and the final set contains at most n/2 minions.

Now, assume that n is odd. Then, we test [n/2] pairs like before.This time, we have a minion that is left alone and we don't test it, let's say c_n. Our final set will contain one minion for every pair tested that reported 'good, good'. Such minions has form 4n + 3 or 4n + 1 and based on that, we either discard minion c_n or put it into our final set.

Now, assume that we have ignored the pairs of minions that do not report 'good, good', and are left with 4n+ 3 minions, for some constant n. In this case there are at least 2n + 2 good minions. From this, we can say that there are at least n + 1 pairs of 'good, good' minions, and at most k pairs of 'bad, bad' minions. So it does not matter about minion c_n and can also be ignored. There is only a minion in each of these 2n + 2 pairs which is not discarded.

Finally, assume that we have ignored the pairs of minions that do not report 'good,good', and we are left with 4n + 1 minions, for some constant n. In this case there are at least 2n + 1 good minions.
Let's say c_n is bad, then there will be at least 2n + 2 good minions, means, n + 1 pairs of 'good, good' minions, and at most (n -1) pairs of bad minions. So, taking a single minion from every pair, plus c_n gives us at least n + 1 good minions and at most (n -1) + 1 = k bad minions. Means, a majority of minions are good and our final set is at most of size [n/2].
Let's say c_n is good, then we have at least k pairs of 'good, good' minions. Taking a minion from each pair plus c_n gives us a majority of good minions and our final set is at most of size [n/2].

c) If at least n/2 minions are good, then using the previous answer to question b), we can compute a single good minion. Now, we can just test the good minion with all the others to find all good minions. From previous solution b), For each pair of this type, it is enough to know that at least one of the two minions in the pair is good to decide the other minion is good or not. This last stage takes (n -1) more tests.

From previous solution b),
number of tests requires to find good minion, $T(n) <= T([n/2]) + [n/2]$ and $T(1) = 0$.
We can solve this relation by master's theorem. After applying master's theorem, we get
$\Rightarrow T(n) = \Theta(n)$
Hence, good minions can be found with $\Theta(n)$ pairwise tests.

7)
Sol. It is given that we can perform any of the below actions.

A. Flip a single tile of your choice, taking one time step.
B. Gather k tiles of your choice in time n/k and spill them onto the table, flipping each with probability 1/2.

Algorithm:
FlipUsingBag B() →

1. if reached at end of the tiles(N) then return.
2. For each (0, (logN)/2) gather k tiles in time N/k and spill in them onto table. And continue FlipUsingBag() with tiles faced up.

FlipOneAtaTime A()→

3. Flip a single tile one time at a time for the remaining flips which are face up.

Running Time:
When we flip N tiles using a bag approximately N/2 tiles are flipped. So time taken in each iteration is N/(N/2) = 2.
Therefore, Runtime T(n) = $2+2^2+..+2^i$

$T(n) = \sum_{i=0}^{logn} (2^i)^{1/2}$

$T(n) = (2)^{logn/2}$

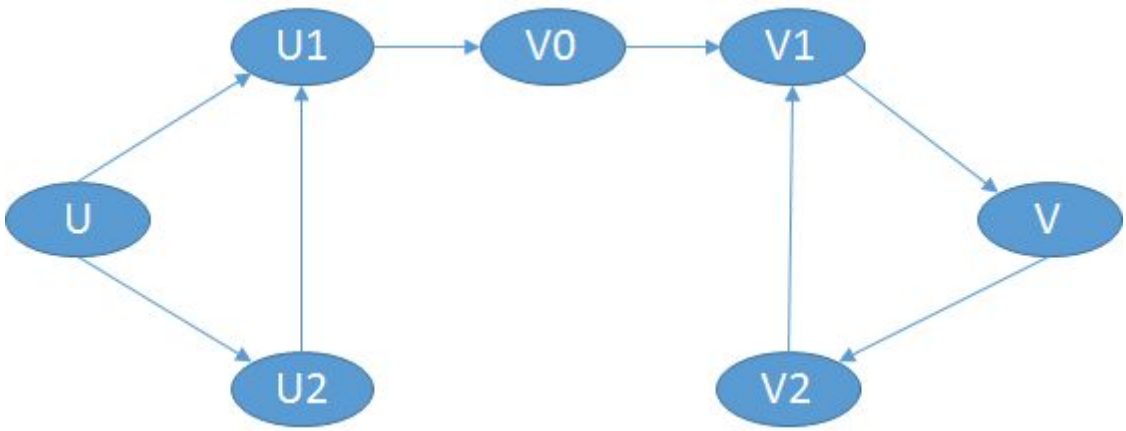
$T(n) = \sqrt{n}$

Runtime for comparisons
T(n) = O(1).(1+2+3+...+log(N/2)) a constant.

Total Run time is ($\sqrt{n}$) and clearly we can say that T(n) = o(n).
Hence, this algorithm has fipped all tiles face-down in expected run time o(n)

8) You are given a strongly connected directed graph G = (V,E) with positive edge weights along with a particular node $v_0 \varepsilon V$. Give an efficient algorithm for finding shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through v_0.

Sol.
We are given a strongly connected directed graph G = (V, E) with positive edge weights along with a particular node $v_0 \varepsilon V$.



As we can see, shortest path between two nodes u and v that passes through v_0 will be the shortest path from [u to v_0] plus from [v_0 to v].
We can calculate the shortest path from [v_0 to v] using Dijkstra's algorithm. To compute the shortest path from [u to v_0], let's look at the nodes first. As we can see, we need to calculate the shortest paths from every vertex to v_0. This is exactly the opposite of finding shortest path using Dijkstra's algorithm, so we can achieve it easily by first reversing the graph (G^R) and then applying Dijkstra's algorithm to find the shortest paths from v_0 to each vertex in G^R.

Algorithm:

1. For each node $u,v,v_0 \varepsilon V$ in Graph G, Run Dijkstra's algorithm from [v_0 to v].
2. Reverse the graph of (u,v_0), G^R = ([v_0 to u],E)
3. Run Dijkstra's algorithm from [v_0 to u].
4. Return Dijkstra(v_0 to v) + Dijkstra(v_0 to u)

Running Time:
Here Dijkstra's algorithm has complexity O ((|V| + |E|) lg |V|). Reversing the graph takes linear time so Dijkstra's algorithm with revere graph also have same complexity. Storing all of the shortest path lengths takes O ($|V|^2$) time.
Hence, complexity of this algorithm is T(n) = O ((|V| + |E|) lg |V|)+O ($|V|^2$).

9)
Sol. We are given a directed graph G = (V, E) with positive edge lengths l_e and positive vertex costs c_v.
Now, we can treat the cost of a path to be its length plus the costs of all vertices on the path. We will find paths of minimum cost using shortest paths problem.
Consider modifying the edge lengths as below.
$l'_{u,v} = l_{u,v} + c_v \rightarrow (1)$
Like this we need to increase the length of each edge by the cost.
Then the cost of any path $s \rightarrow u1 \rightarrow u2 \rightarrow \cdots \rightarrow uk$ is
$c_s + l_{s,u1} + c_{u1} + l_{u1,u2} + c_{u2} + \cdots + l_{uk-1,uk} + c_{uk} = c_s + (l_{s,u1} + c_{u1}) + (l_{u1,u2} + c_{u2}) + \cdots + (l_{uk-1,uk} + c_{uk})$
$= c_s + l'_{s,u1} + l'_{u1,u2} + \cdots + l'_{uk-1,uk}$ (After using equation (1))
Means, the cost of a path is its l'-length, plus c_s. So we can find paths of minimum cost by running Dijkstra's algorithm on G, with edge lengths l'.

Algorithm:

1. For each $(u, v) \in E$, consider edge length as $l'_{u,v} = l_{u,v} + c_v$
2. Run Dijkstra's algorithm with edge length l'.
3. Add cost c_s to Dijkstra(G, l') computed in step 2.

Running Time:
The running time of this algorithm is same as Dijkstra's algorithm, O ((|V| + |E|) lg |V|)