# Assignment 1

Mehul Jain - 112072812
CSE 548 - Analysis of Algorithm

September 16, 2018

## Problem-1

### Part A

For solving these kind of problems, I have a unique way of approach. I feel that world is built around symmetry and we can take benefit of that in CS as well.
For this problem, when k=2,
The complexity of our solution would be f(n) = $O(f(j_1))$ + $O(f(j_2))$, where $j_1$ and $j_2$ are two jars. So here we can see that whichever function is asymptotically bigger will decide the complexity of f(n).If we can decide a strategy in which we utilize both the jars equally we will get an optimal solution.

Let's say that in worst case $j_2$ has to be dropped x times for finding the highest safe rung. Now to come up with an optimal solution( $O(f(j_1)) = O(f(j_2)) = O(x)$ ) we need to ensure that $j_1$ is dropped no more than x times. So here is the strategy for $f(n) = O(f(j_1)) + O(f(j_2)) = 2*O(x) = O(x)$:
Let us take $j_1$ and drop it from the start position at every multiple of $x^{th}$ position(x,2x,3x and so on). If $j_1$ gets broken when it falls from (c*x)th rung (c is any natural number), we need to check $j_2$ for every rung starting from $((c-1)*x+1)^{th}$ rung up to $(c*x)^{th}$ rung. Now how do we decide the value of x? According to the above strategy $j_1$ skips x-1 rungs in one step, so for $j_1$ to have f($j_1$) = $O(x)$, $j_1$ should take maximum of x steps.
$\therefore (x-1)+x\times x \approx$ n. If n $\to \infty$, we can also assume that $x\times x \approx$ n. So from here we can say that f(n) = $O(x) = O(n^{1/2})$. Hence, we can prove that our strategy is

better than linear time complexity.
$\lim_{n\to\infty} f(n)/n = \lim_{n\to\infty}\sqrt{n}/n = \lim_{n\to\infty} 1/\sqrt{n} = 0$

## Part B

We can extend the solution of Part A to obtain solution for part B. For example when k=3,let us take $j_1$ and drop it from the start position at every multiple of $x^{th}$ position(x,2x,3x and so on). If $j_1$ gets broken when it falls from (c*x)th rung (c is any natural number), we need to apply the same logic we used in previous solution with k=2 but this time it would be between $((c-1)*x+1)^{th}$ rung up to $(c*x)^{th}$ rung.For us to have an optimal solution, $j_1$ should not take more than x steps, and since the rungs that can be checked by 2 remaining jugs (k=2) in $O(x)$ is $x^2$, we can say that $x * x^2 \approx$ n. So from here we can say that f(n) = $O(x) = O(n^{1/3})$. Similarly, we can find solution for any f(n) with any $k > 2$ in $O(k * n^{1/k})$.
Following is the explanation of about why I have multiplied by k:
Our initial hypothesis for k=2 was $f(n) = O(f(j_1)) + O(f(j_2))$ = 2*$O(x)$.
Similarly for any k it will be

$$f(n) = \sum_{i=1}^{k} O(f(j_i)) = k * O(x) = O(k * n^{1/k})$$

So lets see how our complexity is in comparison to linear time complexity.

$\lim_{n\to\infty} f(n)/n = \lim_{n\to\infty}(k * \sqrt[k]{n})/n$

From the above statement if k is anywhere comparable to $n^{(k-1)/k}$ we need to device a new strategy. So here is my modified strategy for optimization.
If $k >$ log$(n)$, we can directly apply binary search and solve the problem. So we will use my strategy only when k<=$log(n)$.
$\lim_{n\to\infty} f(n)/n$
= $\lim_{n\to\infty}(k * \sqrt[k]{n})/n$
= $\lim_{n\to\infty}(\log(n) * n^{1/\log(n)})/n$
= $\lim_{n\to\infty}(\log(n) * 2)/n$ [ because $\lim_{x\to\infty} n^{1/\log(n)} = 2$]
= 0

# Problem-2

Assuming that we already have an adjacency list for each specimen, where the list for each specimen would be storing the relation it has with other specimen.

Also, I am assigning color A and B to indicate different specimens where Mark(s) will indicate type assigned to that specimen.Let n be the number of edges and m be the number of relationships we have among n edges. In the following algorithm we randomly assign color to each specimen but at the same time we ensure that relationship between those edges is taken care.

Pseudocode:

```
Main Function(Specimens):
    ∀x ∈ Specimens, Mark(x) = ∅
    For each s ∈ Specimens where Mark(s) = ∅
        Mark(s) = A
        For each neighbor v of s
            If pair(s,v) belongs to same specimen
                DFS(v,A)
            else
                DFS(v,B)
            End if
        End for
    End for


DFS (s, color):
    Mark(s) = color
    For each neighbor v of s
        If pair(s,v) belongs to same specimen
            If Mark(v) = ∅
                DFS(v,color)
            else if Marks(v) != color
                ”Report Inconsistency”
            End if
        Else
            If Mark(v) = ∅
                DFS(v,opposite(color))
            else if Marks(v) = color
                ”Report Inconsistency”
            End if
        End if
    End for
```

Analysis of this Algorithm:
Creating an adjacency list would take $O(m + n)$ time complexity. Also, total time complexity for applying DFS to each component would be $O(m + n)$, where m is number of edges and n is number of nodes. So overall time complexity of our solution would be $O(m + n)$. This algorithm will terminate in 2 cases:
1. All nodes are traversed, in which case the judgment would be Consistent.
2. Inconsistency is Reported.

# Problem-3

Let us create an Adjacency List where each List for a particular Computer C stores the list of pairs - (Cp,T) where (C,Cp,T) is a triple. We will store these pairs in the list in descending order of time (i.e edge(communication) that has maximum time should be accessed first). We will ignore all communications happening before time x and after time y.This Adjacency List can be created in $O$(m+n)
Bool Main Function() :
$\forall x \in$ Computers, VisitTime(x) = $\infty$
return DFS($C_a$,x)

```
Bool DFS(C, time):
        If time >= VisitTime(C)
                return false
        End If
        If C = C_b
                return true
        End If
        VisitTime(C) = time
        For all neighbor-pairs (Cp,T) of C where T >= time
                Remove pair(Cp,T) from the list
                If DFS(Cp,T) = true
                        return true
                else
                        continue mn
                End If
        End For
        return false
End
```

Proof of Complexity: Since I am storing the pairs in descending order of time, each edge of any Node will be visited only once. So the worst case complexity of this single component DFS Algorithm would be $O(m)$.

Proof of Correctness: We are traversing using DFS algorithm through every edge which has a communication time greater than the currently infected Computer's least communicated time. So if we reach our destination $C_b$ then we return true indicating that the computer will be infected.
Hence, Complexity of our Algorithm is $O(m+n)$

# Problem-4

$\sum_{i=1}^{n} \frac{(i^2+5i)}{(6i^4+7)} \equiv \sum_{i=1}^{n} \frac{(1)}{(i^2)} \prec \lg(\lg(n)) \prec \sqrt{\lg(n)} \prec \ln(n) \equiv \lg(\sqrt{n}) \equiv \sum_{i=1}^{n} \frac{(1)}{(i)} \prec 2^{\sqrt{\lg n}} \prec (\lg n)^{\sqrt{\lg n}} \prec min\{n^2, 1045n\} \prec \ln(n!) \prec n^{\ln 4} \prec \lceil \frac{n^2}{45} \rceil \equiv \frac{n^2}{45} \equiv \lfloor \frac{n^2}{45} \rfloor \prec 5n^3 + \log n \prec \sum_{i=1}^{n} i^{77} \prec 2^{\frac{n}{3}} \prec 3^{\frac{n}{2}} \prec 2^n$

# Problem-5

## Part A

In this problem we have to prove that in a connected component if we have a Euler path it implies that every vertex has even degree. Also, if we have all vertex with even degree in a connected graph, it implies we can have at least one Euler Path.
First Part : If we have a Euler path it indicates that every vertex has even degree.
Proof:
We can have a Euler path $V_0, V_1, V_2, V_3 ..... V_n, where V_i \in Vertex$. Since this is a Euler Path it implies $V_0 and V_n$ are same vertex(Implies it has even number of edges till now). For every other vertex $V_i$ in this path we will have an incoming as well as outgoing edge from that vertex. So it implies that every vertex will always have even number of edges.

Second Part : If we have all vertex with even degree in a connected graph, it implies we can have at least one Euler Path.

Proof:
We can always find a closed path in a connected graph with all vertex having even degree, because if we don't then vertex at each end of the path will have an odd degree. Now in this path if have vertex whose edges are not yet visited , we can find a circular path starting from the same vertex and ending at the same vertex and merge it into the existing path. If we do this recursively we will have a point where we have visited all the edges. And the resulting path obtained will be Euler path.

## Part B

Create an adjacency list and check if every vertex has even number of edges. If this is not true, Report "No Euler Path exist". Let us create a PathQueue which will store the Euler Path. Now lets apply DFS from any node v.DFS(∅, v)

```
DFS(p,v):
        Mark(V) as "Visited"
        Remove edge(v,p) from the Adjacency list
        if Number of neighbors = 0
                Add v to PathQueue
                return
        End If
        For s ∈ neighbor
                remove edge (v,s)
                DFS(v,s)
        End For
        Add v to PathQueue
```

Check if all nodes are visited. If this is true return PathQueue, else Report "No Euler Path exist". Proof of Complexity: Complexity of creating an Adjacency List is $O(m+n)$ and Complexity of applying this DFS is $O(m)$. Although we may visit a vertex more than one time, we are ensuring that we are traversing each edge no more than 1 time. We should use a HashMap to store where exactly in the list an edge is located, this will ensure that removal of an edge can be done in $O(1)$ time.Complexity of our Algorithm is $O(m+n)$

# Problem-6

Let us create a connected acyclic graph with 2 vertex. The only way to do so is by having an edge between the two vertex. So when n=2, we can see that we will have exactly 2 vertex with degree 1.
Now for n=3, we should add a new vertex to the existing graph with n=2 by connecting it to only one vertex. If we connect the new vertex to more than 1 vertex we will create a cycle because the existing graph was already connected and we are adding another path between the existing 2 vertex which will pass through this new vertex. Also, even if we are connecting an edge to a vertex with degree = 1, we are providing the graph this new node which will have a degree of 1
From this we can derive 3 properties:
1. If we have a connected acyclic graph with n vertex, we can only add a new vertex IFF it is connected to the existing graph using only one edge.
2. We cannot add edges between existing vertex in a connected acyclic graph since there already exist a path between any two vertex and adding an edge would make this graph cyclic.
3. When we add a new vertex to the existing connected acyclic graph, we get a new graph that will always have #(vertex with degree 1) greater than or equal to the former graph.

Now lets create a graph with n vertex. We can only do so recursively such as: G(n) : G(n-1) + Edge(v,u) , where u is an existing vertex and v is a new vertex added to the graph. Since this graph will somehow be made from G(2) which has 2 vertices with degree 1 we can therefore say that this graph with n vertex will always have at least 2 vertex with degree 1 based on the properties derived above.