AOA Assignment 1

Sunny Shah - 112044068

February 13, 2019

Problem 1

(a)

Problem Summary

Given a budget of k=2 jars, design a strategy that grows slower than linearly

Solution

Since we have 2 jars, we can devise the solution as follows:

- Divide the n rungs(numbered $1, 2, 3..n$) of the ladder into m blocks such that $m = \frac{n}{\lfloor\sqrt{n}\rfloor}$, where each block includes $k = \lfloor\sqrt{n}\rfloor$ rungs, except the last block, which might have less than k rungs
- Test the jar from rung k, if it does not break, test it again from the rung $2k$, $3k$, ... , mk, until it breaks
- If the jar breaks during testing at some rung $a * k$, then it is true that the lowest unsafe rung is between $[(a-1) * k]$(exclusive) and $[(a) * k]$(inclusive)
- Since, we now have only 1 jar with us to test, we test it one by one from $[(a-1) * k + 1]$ to $[(a) * k]$ and check if the jar breaks at any of this positions. If the jar breaks at any position i between $[(a-1) * k + 1]$ to $[(a) * k]$, then we know that the highest safe rung is i-1

Time Complexity

The solution discussed above has 2 parts:

- Find the correct block, which in worst case might require $\frac{n}{\lfloor\sqrt{n}\rfloor} (\approxeq \sqrt{n})$ operations
- Find the lowest unsafe rung from within the $\sqrt{n}$ rungs within the detected block. Since, all blocks have $\sqrt{n}$ rungs within it, we will need at max $\sqrt{n}$ operations to find our solution.

In worst case, we will need to test the first jar m times, i.e. test the safety of jar from end of each block to find it only breaking in the last block and then test each rung in the block.
Thus, the overall complexity of the solution is $((\sqrt{n}) + \sqrt{n})$ making $f(n) \in O(\sqrt{n})$
Therefore,

$$\lim_{n\to\infty} \frac{f(n)}{n} = \lim_{n\to\infty} \frac{\sqrt{n}}{n} = \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

(b)

Problem Summary

Given a budget of $k > 2$ jars, describe a strategy for finding the highest safe rung using at most k jars.

Solution

Taking the same idea as above:

- For k=2, we divided the n rungs into $\frac{n}{\lfloor\sqrt{n}\rfloor}(\approxeq \sqrt{n})$ blocks, similarly for k≥2, we should divide the n rungs into $n^{\frac{1}{k}}$ i.e. $m = \frac{n}{\lfloor n^{\frac{k-1}{k}}\rfloor}(\approxeq n^{\frac{1}{k}})$ blocks
- Thus, if we have k jars, we will have a total of $n^{\frac{1}{k}}$ blocks and each block will contain $L = \lfloor n^{\frac{k-1}{k}}\rfloor$ rungs
- Now, test the first jar with the rungs at position L, 2L, 3L ..., mL(each denoting the end of an individual block of rungs), if we found the block, where the first jar broke, we again recursively apply:

  F_{k-1}(no of rungs in each block) = $F_{k-1}(n^{\frac{k-1}{k}})$

Time Complexity

For n rungs with budget of k jars, we require $n^{\frac{1}{k}}$ operations to select the block and then recursively call the same for the rungs within the block($n^{\frac{k-1}{k}}$) with $k-1$ jars

Where, $F_{k-1}(n^{\frac{k-1}{k}}) = \frac{n^{\frac{k-1}{k}}}{(n^{\frac{k-1}{k}})^{\frac{k-2}{k-1}}} = \frac{n^{\frac{k-1}{k}}}{(n^{\frac{k-2}{k}})} = n^{\frac{1}{k}}$
Similarly, $F_{k-2}(n^{\frac{k-2}{k}}) = n^{\frac{1}{k}}$

Thus, we call the above operations recursively till $k = 1$, which makes our total cost of $F_k(n) = k * n^{\frac{1}{k}} \in 0(kn^{\frac{1}{k}})$
Also,
$F_k(n) \in O(kn^{\frac{1}{k}})$
$F_{k-1}(n) \in O((k-1)n^{\frac{1}{k-1}})$
$F_{k-2}(n) \in O((k-2)n^{\frac{1}{k-2}})$
$F_{k-3}(n) \in O((k-3)n^{\frac{1}{k-3}})$

Therefore,

$$\lim_{n\to\infty} \frac{f_k(n)}{f_{k-1}(n)} = \lim_{n\to\infty} \frac{k * n^{\frac{1}{k}}}{(k-1) * n^{\frac{1}{k-1}}} = \frac{k}{(k-1)}(\lim_{n\to\infty} \frac{1}{n^{\frac{1}{k*(k-1)}}}) = 0$$

What if $k = n$, shouldn't our solution become equal to $O(n * n^{\frac{1}{n}})$?

In case of $k \geq \log n$, we can directly make use of binary search technique and solve the problem in $O(\log n)$

Problem 2

Problem Summary

Textbook Kleinberg & Tardos Chapter 3, page 107, problem #4

Solution

- Let,
  m = no. of judgements
  n = no. of butterflies
- Let G be a graph consisting of N nodes(1,2,3...N), each denoting the i^{th} butterfly
- Let the graph be described using adjacency lists (Adj[1...N]), as per each (u,v) pair in M, such that:

```
for (u,v) in M:
    Add u to Adj[ v ]
    Add u to Adj[ v ]
```

- Now consider the following pseudo code to check if the m judgements are consistent:

```
label[1...N] <- to store label for each node

visited[1...N] <- to check if a node was visited, initially all false

for i <- 1 ... N:

    if visited[i] is False:
        initialize Stack S as empty:
        visited[i] = True
        label[0] = B
        label[i] = A
        push (0,i) to Stack S

        while S is non-empty:
            (u,v) = s.pop()

            for each node w in Adj[v]:
                if visited[w]:
                    if (v,w) is labelled "same" in M:
                        if label[v] != label[w]
                            return False
                    if (v,w) is labelled "different" in M:
                        if label[v] == label[w]
                            return False
                else:
                    if (v,w) is labelled "same" in M:
                        label[w] = label[v]
                    if (v,w) is labelled "different" in M:
                        label[w] = (label[v] == 'A')?B:A
                visited[w] = True
                push (v,w) to Stack S

// If the judgements were really consistent, we can now confidently return true

return True
```

Time Complexity

The above solution visits each edge once and will do the same for multiple disconnected graph(forest). Hence the overall complexity of the above algorithm is $O(m + n)$

Problem 3

Problem Summary

Textbook Kleinberg & Tardos Chapter 3, page 107, problem #11

Solution

- Given:
  C_a = id of computer where virus was first inserted
  x = time at which computer C_a was infected
  C_b = id of computer which we need to verify whether it was infected before or at time y
  y = time threshold of verifying if virus infected C_b

- First, remove all triplets (C_i, C_j, t) where $t < x$ and $t > y$

- Create a Graph G with remaining list of edges (E')

- Let graph be represented by adjacency list as follows:

  Adj $[C_i]$ = [(C_{j_1}, t_{j_1}) , (C_{j_2}, t_{j_2}) , ...]
  where, $t_{j_2} \geq t_{j_2} \geq .... \geq t_{j_n}$

  The above adjacency list generation will only take $O(n)$ operations, as the edge list is already sorted.

- Now consider the following pseudo code to check if C_b will be infected by time y: //

```
Queue queue:
currTime = X
queue.offer( (Ca,x) )

while queue is not Empty():
    currOffer = queue.poll()
    node = currOffer[0]
    currTime = currOffer[1]

    for edge (u,t) in Adj[node]:
        if t < currTime:
            break
        if t >= currTime AND t <= y:
            if( u == Cb )
                return True
            queue.offer( (u,t) )
            Adj[node].delete((u,t))

return False;
```

- The above method runs in $O(n + E)$

- In the above method, we apply BFS to nodes connected to C_a between $time \geq x$ and $time \leq y$ and apply the BFS till we reach the goal state i.e. C_b

- In the solution, we remove the edge once we had traversed over it, since if we have traversed the edge, then it means that we have explored all possible states after the edge as we only look for edges with timings greater than or equal to last edge.

- Also, while applying BFS, we traverse the child nodes in decreasing order of their time i.e. if A is connected to (B,5), (C,4), (D,3), then during the BFS traversal, we traverse the edge (B,5) prior to (C,4), this will ensure that we don't unnecessarily compare the edges. eg. consider there are n edges incident on node B, with each edge having a time/weight of at least k, and there are n other nodes leaving from B, with each edge having time/weight of less than k. In these case, for every n edges incident on B, we will unnecessarily compare the n outgoing edges only to find that there is no eligible edge available. Thus, we traverse from higher weight to lower weight.

Problem 4

Problem Summary

List the functions in increasing asymptotic order

Solution

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

Problem 5

(a)

Problem Summary

Prove that a connected graph G has an Euler tour if and only if every vertex has even degree

Proof:

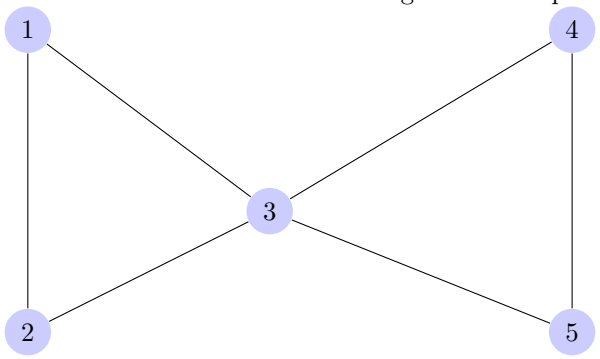
An Euler tour of a graph G is a closed walk through G that traverses every edge of G exactly once

Consider the following Euler's tour:

$a_0, a_1, a_2, ...a_1, ..., a_2, ..a_0$

where, a_0 is part of both start and end of the tour

- For every vertex in the graph, there must be 2 edges incident to it (one for entering that node and other for leaving).
- Thus, for every vertex in the path, the number of edges incident to it will be a multiple of 2.
- This, proves that Euler's tour can only be applied to a connected graph where each vertex have a even degree.
- Lets consider that we are searching the euler's path in the following graph:



- In the above graph, we might end up with the following circuit: 1,2,3,1
- However, since the node 3 have degree more than 2, we can easily replace 3 in the above path with the circuit starting and ending with 3 i.e. 3,4,5,3. This is only possible as long as node 3 have a even degree as we already proved that a circuit requires nodes with even degree.
- We can easily see that if all the nodes in the graph has a even degree, we can merge all the independent closed circuit to form a euler's tour.
- Any odd degree node will never to able to generate a cycle starting and ending from it.

(b)

Problem Summary

Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such tour exists

Solution

- We can easily find the Euler's path using a recursive DFS approach.
- Let the graph be described using adjacency lists (Adj[1...N]), as per each (u,v) edge, such that:
- Pseudo-code:

```
circuit = []   #will be used to store the actual euler circuit
totalNoOfedges = M   #will be used to check if graph is connected

function DFS ( node )
    for u in Adj[ node ]:
        delete u from Adj [node]
        delete node from Adj[ u ]
        totalNoOfedges--
        DFS(u)
    circuit.append( node )

for i in (1...N):
    if len( Adj[i] ) is not even:
        return "Euler circuit do not exist"

# Apply DFS to find the Euler's Path
dfs(1)

# All edges should be traversed in a connected graph
if totalNoOfEdges > 0:
    return "Euler circuit do not exist in non-connected graph"

# Euler's circuit should have M+1 nodes in it
if circuit.length != M+1:
    return "Euler circuit not found"

# Euler's circuit found
return circuit
```

Time complexity

The above algorithm runs in $O(M)$ times, where M is number of Edges in the graph.

Problem 6

Problem Summary

Prove that any connected acyclic graph with n $\geq$ 2 vertices have at least two vertices with degree 1. Notice that you should not use any known properties of trees and your proof should follow from the definitions directly.

Proof

- Let us consider a connected acyclic graph G

```
if n==2
    then, there can only be 1 edge between them
    thus there will be 2 nodes with degree 1
```

For n > 2:

- Let us assume, that there are 2 vertices in G with degree 1 i.e. A_i and A_j
- Since, G is a connected graph, there will be a path between A_i and A_j
- Let us try to add to the degree of node A_i i.e. let us make an edge from A_i to some node A_k.
- Since, this is a connected graph, A_k must have a path to A_i for adding this new edge. So, adding the new edge from A_i to A_k will create 2 paths from A_i to A_k.
- This in turn is creating a cycle between A_i, A_j and A_k, which contradicts the fact that the graph is acyclic