CSE 548 Homework 1

Amit Dharmadhikari (SBU ID: 112044244)

September 2018

1 Question 1

1.1 (a)

Given a budget of k=2 jars, the task of finding the highest safe rung can be completed with a time complexity of $O(\sqrt{n})$. $f(n) = O(\sqrt{n})$ is a function which grows slower than linearly. The strategy to achieve this is as follows:

We drop the first jar from rungs that are multiples of $\sqrt{n}$. If the first jar doesn't break at all, then the highest rung is the highest safe rung. If the jar breaks at, say, $j\sqrt{n}$, then we know that the highest safe rung is somewhere between $(j-1)\sqrt{n}$ and $j\sqrt{n}-1$, both inclusive. So, we drop the second jar from every rung from $(j-1)\sqrt{n}+1$ to $j\sqrt{n}-1$. The highest safe rung is the one just below the rung from which the second jar breaks. If the second jar doesn't break at any point, then $j\sqrt{n}-1$ is the highest safe rung.

Thus, we drop both the jars $\sqrt{n}$ times each. Hence, the time complexity of this algorithm is $O(\sqrt{n})$. In the event that n is not a perfect square, we will consider the floor of $\sqrt{n}$. In this case, the first jar will be dropped a maximum of $2\sqrt{n}$ times. However, this won't impact the time complexity.

1.2 (b)

This question can be thought of as a generalization of the previous question. First of all, we drop the first jar for a maximum of $2n^{1/k}$ times at intervals of $n^{(k-1)/k}$ each. We take the floor of $n^{(k-1)/k}$ if need be. Now, we have an interval of $n^{(k-1)/k}$ rungs in which we know that the highest safe rung exists and we have $k-1$ jars left. We apply the same procedure recursively. Thus, we drop the second jar $2n^{(k-1)/k^{1/(k-1)}}$ times, i.e. $2n^{1/k}$ times. This process is repeated k times, thus giving us $f_k(n) = 2kn^{1/k}$. As k increases, the exponent of n decreases, and thus, it is clear that $f_k(n)$ with higher k grows asymptotically slower than $f_k(n)$ with lower k.

2 Question 2

We can use the BFS algorithm taught in class for solving this problem. We can think of the specimens of butterflies as nodes of a graph and judgments as edges. First of all, we will construct a graph $G = (V, E)$ considering V, i.e. vertices, as the set of specimens of butterflies and E, i.e. edges, as the set of judgments. Right now, for the purpose of constructing a graph, we will not consider whether the judgment is "same" or "different"; we will consider all judgments as edges. Then, we perform labelling of the nodes using BFS. We start off by labelling "S", i.e. the start node as "A". Then, every time we add a node to the BFS tree, we check whether the judgment for this node and its parent node is "same" or "different" and accordingly assign a label to it. eg. If the parent node's label is "A" and the judgment is "different", then the new node will be assigned the label "B". Whether the starting node is assigned the label "A" or "B" makes no difference.

Now, once the BFS tree is constructed, we will run through our list of judgments and check whether they hold true. If we spot any inconsistency, we will declare that the set of judgments is inconsistent and return from the algorithm. Otherwise, if we don't spot any inconsistency, we will declare that the set of judgments is consistent.

We have n nodes(specimens) and m edges(judgments). The time complexity of constructing the graph is $O(m + n)$. The time complexity of the BFS algorithm is $O(m + n)$. The time complexity of the running through the set of judgments to check for inconsistencies is $O(m)$. Thus, the time complexity of the entire algorithm is $O(m + n)$.

3 Question 3

We can solve this question using the concepts of directed graph and breadth first search, both of which have been taught in class. First of all, we need to construct a directed graph from the given triplets. We will maintain a separate linked list for each computer, in which we will store pointers to the nodes corresponding to that computer. This step proves useful later when we need to find the node from which to start our breadth first search. For every triplet (C_a, C_b, t_k), we will make the nodes (C_a, t_k) and (C_b, t_k). We add the pointers to these nodes to their respective computers' linked lists. Then, we add two directed edges between these two nodes, one in each direction. If there already exists a node corresponding to a computer, then we need to add an edge to connect the new node to it. For example, if there exists a node (C_a, t_j) and $t_j \leq t_k$, then we need to connect (C_a, t_j) and (C_a, t_k) with a single directed edge going from (C_a, t_j) to (C_a, t_k).

Now, once the graph is constructed, we need to find the node at which the

computer in question got infected. We can do this by traversing through the linked list for that particular computer. Let's consider computer C_a got infected at time x. We need to find the first node (C_a, t) in the linked list of C_a such that $t \geq x$. Then, we need to start our directed breadth first search algorithm from this node. Note that if there is a directed edge from node A to node B, then the time-stamp of node B is greater than or equal to A, there is no possibility of mistakenly stepping backward in time while performing breadth first search. Our goal is to answer the question of whether the virus has infected computer C_b by time y. Hence, every time we visit a node, we check if it corresponds to the computer C_b, and if it does, we check if the time-stamp of that node is less than or equal to y. If yes, we return the answer as "Yes". If we don't find any such node, then we return the answer as "No". The complexity of constructing the graph and of BFS is $O(m+n)$ each, hence our algorithm runs in $O(m+n)$ time.

4 Question 4

$\sum_{i=1}^{n}(i^2+5i)/(6i^4+7)$

$\sum_{i=1}^{n}\frac{1}{i^2}$

$\lg\lg n$

$\sqrt{\lg n}$

$\lg\sqrt{n}$

$\sum_{i=1}^{n}\frac{1}{i}$

$\ln n$

$2^{\sqrt{\lg n}}$

$(\lg n)^{\sqrt{\lg n}}$

$\min\{\mathrm{n}^2, 1045n\}$

$\ln(n!)$

$\mathrm{n}^{\ln 4}$

$\left\lfloor\frac{n^2}{45}\right\rfloor$

$\frac{n^2}{45}$

$\left\lceil \frac{n^2}{45} \right\rceil$

$5n^3 + \log n$

$\sum_{i=1}^{n} i^{77}$

$2^{n/3}$

$3^{n/2}$

2^n

$\sum_{i=1}^{n}(i^2+5i)/(6i^4+7) \equiv \sum_{i=1}^{n} \frac{1}{i^2}$

$\sum_{i=1}^{n} \frac{1}{i^2} \prec \lg \lg n$

$\lg \lg n \prec \sqrt{\lg n}$

$\sqrt{\lg n} \prec \lg \sqrt{n}$

$\lg \sqrt{n} \equiv \sum_{i=1}^{n} \frac{1}{i}$

$\sum_{i=1}^{n} \frac{1}{i} \equiv \ln n$

$\ln n \prec 2^{\sqrt{\lg n}}$

$2^{\sqrt{\lg n}} \prec (\lg n)^{\sqrt{\lg n}}$

$(\lg n)^{\sqrt{\lg n}} \prec min\{n^2, 1045n\}$

$min\{n^2, 1045n\} \prec \ln(n!)$

$\ln(n!) \prec \mathrm{n}^{\ln 4}$

$n^{\ln 4} \prec \left\lfloor \frac{n^2}{45} \right\rfloor$

$\left\lfloor \frac{n^2}{45} \right\rfloor \equiv \mathrm{n}^2/45$

$\frac{n^2}{45} \equiv \left\lceil \frac{n^2}{45} \right\rceil$

$\left\lceil \frac{n^2}{45} \right\rceil \prec 5\mathrm{n}^3 + \log n$

$5n^3 + \log n \prec \sum_{i=1}^{n} i^{77}$

$\sum_{i=1}^{n} i^{77} \prec 2^{n/3}$

$2^{n/3} \prec 3^{n/2}$

$3^{n/2} \prec 2^{n}$

5 Question 5

5.1 (a)

We need to prove this in two parts. First, we will prove that if a connected graph G has a Euler tour, then every vertex has even degree. Then, we will prove its converse.

5.1.1 To prove: If a connected graph has a Euler tour, then every vertex has even degree

Every time the Euler tour visits a vertex (apart from the starting vertex), it leaves immediately after entering. Thus, it uses two edges adjacent to a vertex every time it visits a vertex. Even if a Euler tour revisits a vertex, it will use two unused edges, so basically it uses an even number of edges adjacent to each vertex. Also, the Euler tour ends where it started, so it uses an even number of edges for the starting node as well. Furthermore, we know that a Euler tour uses every edge of the graph exactly once; there are no edges in the graph that are not included in the Euler tour. Hence, every vertex has even degree.

5.1.2 To prove: If a graph is connected and every vertex has even degree, then the graph has a Euler tour

Let G be a connected graph in which every vertex has an even degree. Let v be the starting vertex. We will start from v and select any edge adjacent to v arbitrarily. Since every edge has an even degree, if there is a suitable edge to enter a vertex, there will also be a suitable edge to leave a vertex. In other words, we will not get stuck at any vertex; we are bound to return back to v. Let T be the tour we make in this manner. T may or may not contain all edges in graph G. If T does contain all edges in graph G, then we have proved that the graph G has a Euler tour. Otherwise, we need to consider the parts of G that have not been covered by T. $G - T$ may have several components, let's name them as $C_0, C_1, ...$ Every component is a connected graph with every vertex having even degree (as the tour T used an even number of edges from each vertex). We will apply the same procedure recursively to each component. Thus, we can find a Euler tour for each of the components. (If we don't find a tour covering all vertices for a component, then it will give rise to more components and so on). Now, for each component $C_0, C_1, ...$ we need to keep the vertex which is common between the component and tour T as the starting vertex. Thus, we

can join all these Euler tours to the original tour T, thus proving that a Euler tour exists for the entire graph G.

5.2 (b)

Let G be a graph with n vertices and m edges. First of all, we need to check whether the graph satisfies the conditions which are required in order to have a Euler tour, i.e. it needs to be connected and every vertex must have even degree. We will run through all the vertices to check their degrees. If any vertex does not have even degree, then we will report that a Euler tour does not exist for this graph. Next, we need to check whether the graph is connected or not. For this, we will perform breadth first search on the graph. If every vertex of the graph is covered under breadth first search, then it means that the graph is connected, otherwise it is not connected. If it isn't connected, then we will report that a Euler tour doesn't exist for this graph.

Now that we have confirmed that a Euler tour exists for graph G, we need to find the Euler tour. The algorithm for finding a Euler tour is fairly obvious from the second part of the proof in the previous question. We start off with a vertex arbitrarily selecting an edge and when we come back to the starting vertex, we follow the same procedure for the remaining components of the graph, and finally join all these tours together to form a complete Euler tour. This process of recursion can be best modelled with a stack as follows.

G is a graph, V is the set of vertices, E is the set of edges and s is the starting node Initialize stack as null Initialize an array of booleans of the size of the set of edges E, in order to store whether the edges are marked or unmarked.

```
Function Visit(v):
        For each unmarked edge (v,w) adjacent to v:
                Mark edge (v,w)
                Visit(w)
        Push vertex v on stack

Visit(s)

While stack is not empty:
        Pop a vertex from the stack and print it
```

Initially, checking the degree of each vertex costs $O(n)$ time, as we run through each vertex. Later, BFS costs $O(m + n)$ time. Finally, our main algorithm runs through all the edges once and the number of elements stored on the stack corresponds to the number of edges, hence the time complexity as well as space complexity is $O(m)$. Thus, we can say that the overall time complexity of our algorithm is $O(m + n)$.

6 Question 6

Let's consider that graph G is connected and acyclic, and P is the path with maximum length in graph G.

$$P = \{v_0, v_1, v_2, ..., v_{m-1}, v_m\}$$

v_0 is the starting vertex of path P and v_m is the ending vertex of path P. Since v_1 is adjacent to v_0 and v_{m-1} is adjacent to v_m, the degree of v_0 and v_m is at least 1. However, v_0 and v_m can't be adjacent to any other vertices in G. It is proved as follows:

Let's assume that there exists a vertex v adjacent to v_0. There are two cases:

1. v doesn't exist in P:

In this case, the path P can be extended to the vertex v. Thus, the length of the path P increases by 1. However, this contradicts our earlier assumption that path P is the path with the longest length in graph G. Thus, the vertex v can't exist.

2. v exists in P:

In this case, the path P will form a cycle, which contradicts our assumption that graph G is an acyclic graph. Thus, vertex v can't exist.

Thus, we have proved that vertex v_0 is not adjacent to any vertex apart from v_1, thus making its degree 1. Same applies to v_m. Thus, we have proved that any connected acyclic graph with $n >= 2$ has at least two vertices with degree 1.