Bo Yang (Aaron) USCID: 7526922531

USC CSCI 570 Homework1 Date: 2021-01-25

1. Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if *f(n) = O(g(n))*

$$2^{log(n)}, \quad (\sqrt{2})^{log(n)}, \quad n \cdot (log(n))^3, \quad 2^{\sqrt{2log(n)}}, \quad 2^{2^n}, \quad n \cdot log(n), \quad 2^{n^2}$$

Solution:

> Basically, I follow this order,
> $O(1) <= O(log(n)) <= O((log(n))^C) <= O(C^{log(n)}) <= O(n) <= O(n \cdot log(n)) <= O(n^C) <= O((log(n))!) <= O(n^{log(n)}) <= O(C^n)$
> ,
> which can help me solve this kind of problem very fast.
> $O(1) \Rightarrow None$
> $O(log(n)) \Rightarrow None$
> $O((log(n))^C) \Rightarrow None$
> $O(C^{log(n)}) \Rightarrow$ ①②④
> $O(n) \Rightarrow None$
> $O(n \cdot log(n)) \Rightarrow$ ③⑥
> $O(n^C) \Rightarrow None$
> $O((log(n))!) \Rightarrow None$
> $O(n^{log(n)}) \Rightarrow None$
> $O(C^n) \Rightarrow$ ⑤⑦
>
> Answer : ④ < ② < ① < ⑥ < ③ < ⑤ < ⑦
> Or : $(2^{\sqrt{2log(n)}}, \quad \sqrt{2})^{log(n)}, \quad 2^{log(n)}, \quad n \cdot log(n), \quad n \cdot (log(n))^3, \quad 2^{n^2}, \quad 2^{2^n}$

2. Give a linear time algorithm based on BFS to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. It should not output all cycles in the graph, just one of them. You are NOT allowed to modify BFS, but rather use it as a black box. Explain why your algorithm has a linear time runtime complexity.

Solution:

> When we use BFS traverse the given graph **G**, we will get a tree **T**. if all edges in the tree **T** also existed in the graph **G**, implying that **G = T**. As we all know, tree does not has a circle. so dose this graph G. and the time complexity is $O(V+E)$. (pure BFS)
> Otherwise, there is at least one circle in **G**. Based on that difference, the specific edge = **(a, b)**, we can find **a** and **b**'s least common ancestor repeatedly and save them into a list variable, those edges stored in the list should form a circle.

> Pseudocode

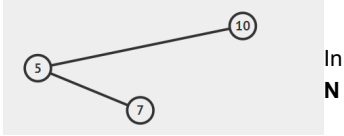
```
Tree T = BFS(G)
if T.edges == G.edges:
    print("no circle")
else:
    path = list()
    edge = (a, b)
    path.add(edge)
```

```
        path.add(edge)

        while a != b:
            # find node's ancestor

            v1  = find_ancestor(a)
            v2  = find_ancestor(b)

            path.add((v1, a))
            path.add((v2, b))

            a, b = v1, v2

        print(path)

# func 'find_ancestor' could take advantage of T.edges, take O(1) time hopefully
# (edges just store key-value pair, and use hash key to find it)
```

> Thus, the total time complexity of my algorithm is also $O(E+V)$

3. A binary tree is a rooted tree in which each node has at most two children. Show by *induction* that in any nonempty binary tree the number of nodes with two children is exactly one less than the number of leaves.

Solution:

> Based on the question, it just ask us to prove the # of node which have two children nodes equals to the # of leaves.
> So I let **N** denote the number of node which have two children nodes in tree T.
> And **L** is the number of leave in tree T.
>
> > Base case: a tree only contain one leaf ⇉ N = 0, L = 1, Thus N = L - 1.
> >
> > 
> >
> >
> > Inductive hypothesis: Assume **N** = **L** - 1 for every tree with **N** < n nodes.
> > Inductive step: Prove **N** = **L** - 1 for every graph with **N** = n nodes.
> >
> > Follow this thought, we add one node on that previous tree (only a root and a leaf). **N < 3**
> > We will get tree A or tree B. **N = 3**
> >
> > 
> >
> > In tree A, we add node on an existing node which has a node already.
> > In this way, **N** = 1, **L** = 2.
> >
> > 
> >
> > In tree B, we add node on an existing node which dose not has child.
> > **N** = 0, **L** = 1.
> >
> > We keep adding nodes on tree T, we can say if **N** increase by 1, **L** will add 1 as well. or they will keep the same.
> > In binary tree, you won't meet other situation except for these two cases.

4. Prove by contradiction that a complete graph K_5 is not a planar graph. You can use facts regarding planar graphs proven in the textbook.

Solution:

> Based on the theorem, **"In any simple connected planar graph with at least 3 vertices, E < 3V - 6"**, we can say a complete graph K_5 cannot be a planar graph.
> That is because in this question, the mentioned complete graph has vertices *V = 5*, edges *E = V(V - 1) / 2 = 10*.
> According to the previous theorem, Only the graph meets the number of edge *E < 3 · 5 - 6 = 9* could the graph be a connected planar graph.

5. Suppose we perform a sequence of *n* operations on a data structure in which the i^{th} operation costs *i* if *i* is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

Solution:

> Assume $n = 2^k$ operations, and these operation can be split into two parts.
>
> 1. if *i* is an exact power of 2, $cost(i) = 2^i$, which $i \in k$.
>
>   > part 1 cost = $\Sigma_{i=0}^{k} 2^i$
>
> 2. if not, the remaining operations will cost 1.
>
>   > part 2 cost = $n - k - 1$
>
> So all cost is:
> $$\begin{aligned} Cost &= \Sigma_{i=0}^{k} 2^i + n - k - 1 \\ &= 2^{k+1} - 1 + n - k - 1 \\ &= 2 \cdot n + n - k - 2 \\ &= 3 \cdot n - k - 2 \\ &= 3 \cdot n - 2 - log_2 n \end{aligned}$$

So cost per operation = $\frac{3 \cdot n - 2 - log_2 n}{n} = O(3)$

6. We have argued in the lecture that if the table size is doubled when it's full, then the amortized cost per insert is acceptable. Fred Hacker claims that this consumes too much space. He wants to try to increase the size with every insert by just two over the previous size. What is the amortized cost per insertion in Fred's table?

Solution:

> Assume there is an array whose size is 1. And we will insert one element each time. For now, our array of size one is empty, and it has one available space.
>
> 1. array is empty, just do insertion. *(size = 1, available = 0)*
> 2. array is full, we allocate an new array which has three (1 + 2) element spaces and do insertion one time and one copy operation. *(size = 3, available = 1, copy = 1)*
> 3. just do insertion. *(size = 3, available = 0, copy = 0)*
> 4. array is full, we allocate an new array which has five (3 + 2) element spaces and do insertion one time and three copy operation. *(size = 5, available = 1, copy = 3)*
> 5. just do insertion. *(size = 5, available = 0, copy = 0)*
>
> .... (keep doing until finish all data)
>
> So let's say, there are N elements need to store. Definitely, we need to do N times insertion. And when $i_{th} (i \in N)$ is even, we still need to do copy work.
> Thus, we denote $N = 2 \cdot k$. When doing $2 \cdot k_{th}$ insertion, we also need to do $2 \cdot k_{th} - 1$ times copy.
> Finally, $Cost = N + \frac{(1 + (2 \cdot k - 1)) \cdot k}{2}$
> $$\begin{aligned} &= N + k^2 \\ &= N + N^2 / 4 \end{aligned}$$

$N + N^2/4$

The amortized cost per insertion is $\frac{N+N^2/4}{N} = \frac{N}{4} + 1 = O(N)$

7. You are given a weighted graph G, two designated vertices s and t. Your goal is to find a path from *s* to *t* in which the minimum edge weight is maximized i.e. if there are two paths with weights 10→1→5 and 2→7→3 then the second path is considered better since the minimum weight (2) is greater than the minimum weight of the first (1). Describe an efficient algorithm to solve this problem and show its complexity.

Solution:

Assume graph G = (V, E), and we can solve this problem by the following steps:

1. Sort all edges by their weight. This will take $O(E \cdot log(E))$ time.
2. And then we enumerate all edges. In every specific loop, we can run try to find a path from s and t by using BFS or DFS. The running time complexity of BFS is $O(V + E)$.

So the time complexity of whole algorithm is $O((V + E) \cdot E)$.