CSCI 570 Spring 2020 Xinrui Ying (USCID 1092350440)
Homework 1 Problem

Solution:

1. The increasing order is as following:
   $2^{\sqrt{2\log n}}, (\sqrt{2})^{\log n}, 2^{\log n}, n\log n, n(\log n)^3, 2^{n^2}, 2^{2^n}$
   Proof (Mainly based on L'Hopital's Rule and constant are ignored in the process):

$$\lim_{x\to\infty} \frac{2^{\sqrt{2\log n}}}{(\sqrt{2})^{\log n}} = \lim_{x\to\infty} \frac{2^{\sqrt{2\log n}}}{2^{\frac{1}{2}\log n}} = \lim_{x\to\infty} \frac{\frac{1}{\sqrt{\log n}}\frac{1}{n}}{\frac{1}{n}} = 0 \implies 2^{\sqrt{2\log n}} = O((\sqrt{2})^{\log n})$$

$$(\sqrt{2})^{\log n} = 2^{\frac{1}{2}\log n} = 2^{\log\sqrt{n}} = O(\sqrt{n})$$

$$2^{\log n} = O(n)$$

$$\lim_{x\to\infty} \frac{n(\log n)^3}{n\log n} = \infty \implies n\log n = O(n(\log n)^3)$$

$$\lim_{x\to\infty} \frac{2^{2^n}}{2^{n^2}} = \lim_{x\to\infty} \frac{2^n}{n^2} = \infty \implies 2^{n^2} = O(2^{2^n})$$

2. Choosing one node in the graph, run BFS on the node and retrieve one spinning tree T. We delete the edges in original graph that are also in T.

   We loop through all other nodes in the original graph and do BFS, if the BFS returns just one node, we continue; if BFS returns a spinning tree U, we choose any one edge from node a of U to node b of U. We traverse the graph T, to find the path from a to b. Concatenate two paths to be the result and end the whole loop.

   If after looping through all nodes, every time just one node is returned, we can output that there is no cycle in the graph.

   Let n be the size of $|V|$, and m be size of E. We use BFS black box to get the first spinning tree T takes $O(n+m)$. We delete all edges in T takes $O(n-1) = O(n)$ as there is only $n-1$ edges in a n—node spinning tree. Then, in the loop, if the BFS returns a node, that means this node is isolated, the BFS for this process will take $O(1)$. If the BFS returns another spinning tree, we are going to choose any arbitrary edge from this new spinning tree. This takes $O(1)$. Finding the path between these two node in the first spinning tree T will take at most $O(n)$ time as the edges in T are $O(n)$.

   Hence, in the worst case, the total time complexity will be $O(n+m)+O(n)+n*O(1)+$ $*O(n) = O(n+m) \implies$ linear.

3. BFS find a path using fewest number of edges. That is, for each node a in BFS tree U, the path from root v to a in U should be the minimum among all paths from v to s in original graph. Suppose the height of U is h, then there is a set of vertices V' that each vertex in V', the path in U from v to it will be h. Now, for each vertex in V', we check them in DFS tree T, the path from v to it will be at least h in T because BFS find a path using fewest number of edges. Thus, the height of T will be at least h as the height is the longest path in a tree. Therefore, the height of T will be at least the height of U.

4. Let n be number of nodes in G

   Base case: When $n = 1$, the number of leaves is 1 and the number of nodes with exactly 2 children is 0, the statement that the number of nodes with two children is exactly one less than the number of leaves holds.

   Suppose the statements holds for $n = 1, 2, 3...k$, we want to show that $n = k+1$ also holds.

   Let G' be the graph with k node, let a be number of nodes with two children and b be number of leaves, by assumption, $b - a = 1$

   Now adding one node v to the graph G', if v is connected to v', there will be three different cases:

   - If v' is a leave, then connecting v to v' makes v a children of v', v is leaf but v' is no longer a leaf. The number of leaves b doesn't change. v' will not be a two children node thus a doesn't change $\implies b - a = 1$
   - If v' is a parent node with 1 children, then number of leaves will add 1 due to v will be a leave node. Also, the number of nodes with two children will also add one due to v' will have two children after v become its second children. Thus, $b = b + 1$ and $a = a + 1 \implies b - a = 1$
   - If v is the root and has two children, v' becomes the parent node of the root. Nothing changes as no additional leaves no additional nodes has two children $\implies b - a = 1$

   Combine all cases, we proved that in any nonempty binary tree the number of nodes with two children is exactly one less than the number of leaves.

5. By Euler formula, let v be number of vertices, e be number of edges and f be number of faces, $v - e + f = 2$. Given k_5, $v = 5$ and $e = 1 \implies f = 7$. For any graph with $v \geq 3$, from theorem, $f \leq 2v - 4 \implies 7 \leq 6$, hence, k_5 is not a planar graph.

6. Total cost of n operation is

$$\sum_{i=0}^{\log n} 2^i + (n - \log n) = 2^{\log n + 1} + n - \log n - 1$$

   amortized cost per operation takes $(2^{\log n+1} + n - \log n - 1)/n = 2 + 1 - \frac{\log n}{n} - \frac{1}{n} = O(1)$

7. Total cost of $2n$ insert is

$$\sum_{i=1}^{i\leq n}(2i-1)+2n=\frac{(1+2n-1)(n)}{2}+2n=n^2+2n$$

amortized cost per insert takes $(n^2+2n)/(2n)=\frac{n}{2}+1=O(n)$

8. First sort all edges once, put edges in ascending order in a list L and initialize result outputPath as None.

Iterations:
We take the median of the list L, if L is of size 0, we return outputPath. If L is not empty, delete the set of edges S that weight are less than the median in the graph. We run BFS on the graph to see if a path exist from s to t. If a path exist, we store the path in outputPath and delete all edges in L that's less or equal to the above median and iterate the above process. If such path doesn't exist, we add back the set of edges S we just deleted to the graph and we then delete all edges in L that's greater or equal to the above median and iterate the above process.

We only have to sort the edges once, which takes $O(E\log E)=O(n^2\log n)$.

For the iteration process, because we are using binary search, it will iterate $O(\log E)=O(\log n)$ times. In each iteration, deleting edges and possibly adding back edges will cost $O(E)=O(n^2)$, running BFS to find the path between s and t takes $O(E+V)=O(n^2)$. Thus, whole iteration process will be $O(n^2\log n)$

The total time complexity will be $O(n^2\log n)$ or $O(|E|\log|V|)$.

9. For k—way merging, we first store lists as value in a min binary heap with each list's first element as key. Initialize result as an empty list named resultList.
Iteration:
While the heap isn't empty, we append the root's key to the resultList. We delete the first element in root's value, if the list is now empty, we delete the root node in the heap; else, we update the root's key to the new first element in the root's value. We then need to do heapify to re-sort the heap. Continue the loop.

Time complexity:
Build the heap for k list takes $k\log k$, we only do it once. Then, because there are n elements in total, we are going to do iteration n times. Each time, changing the keys takes $O(1)$. Heapify takes $O(1)$ time and we need to do it $O(\log k)$ times because the height is $O(\log k)$. The iteration process in total takes $O(n\log k)$.

The total time complexity is $O(n\log k+k\log k)=O(n\log k)$.

■