CSCI 570 Homework 1 Rubrics

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

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

Solution:

It is easy to find that $2^{\log n} = n$ and $(\sqrt{2})^{\log n} = 2^{\log n/2} = n^{1/2} = \sqrt{n}$. To compare $2^{\sqrt{2\log n}}$ with $2^{\log n}$. By applying logarithm on both functions, we have $\sqrt{2\log n}$ and $\log n$. Since $\sqrt{2\log n} \ll \log n$, we have $2^{\sqrt{2\log n}} = O(2^{\log n})$

The final result is

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

Rubric: (10 points)

The grader will try to find the longest common subsequence between the student's solution and our solution. The partial credit will be given according to the length of the longest common subsequence, L, with the following rule:

- 10 points if $L = 7$, which means the student's answer is exactly the same as our solution.
- $10 - 2 \times (7 - L)$ points if $3 \leq L \leq 6$.
- 1 point if $L = 2$, which means the grader can only find one pair of functions in the student's solution is correct.
- 0 point if $L = 1$.

Notice that if the grader notice that the student's solution is arranging the functions by the decreasing order of growth rate, the grader should reverse the student's answer, and give points by the following:

- 5 points if the reversed sequence is exactly the same as our solution.

- 2 points if the longest common subsequence between the reversed sequence and our solution is 6.
- 0 points for other cases.

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:

After running the BFS on the graph G from an arbitrary vertex r, we have its BFS search tree, T. (Here we assume that G is connected, where every vertex appears in T. Otherwise, we can repeat the following algorithm on all disjoint components of the graph)

Then, for each edge of the graph, we check whether it appears in the BFS search tree. If all edges of G is on T, we know that G has no cycle. If one edge, $e = (u, v)$, does not belong to the search tree, we know that there is a cycle. We can find the least common ancestor of u and v, denoted by a, by comparing the the path $r \to u$ and $r \to v$. Now we have the cycle in G: $a \to \ldots \to u \to v \to \ldots \to a$

Analysis of time complexity Let the number of the edge of the graph G be E and the number of the vertices be V. After generating the BFS search tree T, we need to construct a vector p where for each vertex v, $p[v]$ is the parent of v in the BFS search tree. The time complexity of the construction is $O(V)$.

To check whether an arbitrary edge $e' = (u', v')$ exists in T, we may just check whether $p[u'] = v'$ or $p[v'] = u'$. This will take constant time for each edge and the total time complexity will be $O(E)$.

If one of them is true, we know that e' is in T. If both of them are false, then there is a cycle. We can find that cycle by finding the least common ancestor. First, we traverse from u' to the root and from v' to the root. Then we can compare the two path to find the common vertices, also the least common ancestor.

The time complexity of finding the least common ancestor of u and v is also $O(V)$ since both traversing and comparing will take $O(V)$ time.

Thus, the total time complexity is $O(E + V)$

Another solution we can also use adjacency matrix to check whether e' exists in T in constant time. The total time complexity of enumerating edges will be also $O(E)$. If we find an edge that is not in T, then we can find the cycle by finding the least common ancestor.

Rubric: (15 points)

- (5 points) The algorithm should be able to find a cycle in linear time.
- (5 points) The algorithm should be able to output a cycle in linear time.
- (5 points) Analysis of the time complexity is correct.

Notice that if the student's algorithm is using a modified BFS, 10 points will be deducted.

3. Let $G = (V, E)$ be a connected undirected graph and let v be a vertex in G. Let T be the depth-first search tree of G starting from v, and let U be the breadth-first search tree of G starting from v. Prove that the height of T is at least as great as the height of U.

Solution:

We can prove it by contradiction. Suppose that the distance of each edge in G is 1. Then for every vertex u, the path from v to u on the BFS search tree is the shortest path from v to u in G.

Let h_T the depth of the DFS search tree T and h_U the depth of the BFS search tree U. If $h_T < h_U$, then there exists a vertex u in U such that the depth of u, the number of edges consisting the path $v \to u$ in U, is $d_u = h_T + 1$.

However, we can find a path from v to u in T, whose length is at most h_T. Using the statement at the beginning, we have $d_u < h_T$. Now we have $h_T + 1 < h_T$, which is a contradiction.

Rubric: (10 points)

- (5 points) The negation of the statement is correct, which is the height of T is less than the height of U.
- (5 points) The proof why the negation of the statement will lead to a contradiction is correct.

4. 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:

We can prove the statement by the induction on the number of nodes, V. Let L the number of the leaves and S the number of nodes having two children. The statement we try to prove is

$$S = L - 1 \tag{1}$$

Base Case If $V = 1$, equation (1) is obviously true.

Induction Hypothesis Assume that equation (1) holds for all trees of $V - 1$ nodes.

Induction Step Now we try to prove that it also holds for an arbitrary tree, T, of V nodes. We can choose an arbitrary leaf node, u, whose parent is p. If we remove u, the remaining tree, T', will contains $V - 1$ nodes. By the induction Hypothesis, we know that the number of the nodes with two children in the new tree, S', is exactly one less than the number of leaves, L'. That is, $S' = L' - 1$.

Consider the number of children p has before removing u.

- If p has two children, we have $S = S' + 1$ and $L = L' + 1$, which leads to $S = L + 1$.
- If p has only one child, we have $S = S'$ and $L = L'$, which also leads to $S = L + 1$

This completes our proof.

Notice that we can also prove the induction step by inserting a new node on a tree of $V - 1$ nodes. The proof should be similar.

Rubric: (10 points)

1. (2 points) Base case should be stated. And since it is a nonempty binary tree, the base case must be $V = 1$ rather than $V = 0$.
2. (2 points) Induction hypothesis should be stated.
3. (6 points) Induction step should be stated. Each case worth 3 points.

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

Solution:

Suppose that K_5 is a planar graph. Since K_5 is also a simple and connected graph with 5 vertices. Thus, we can use the following theorem from the book:

In any simple connected planar graph with at least 3 vertices, $E \leq 3V - 6$.

However, in the graph K_5, we have $E = 10$ and $V = 5$. This leads to $3V - 6 = 9 < 10 = E$, which is a contradiction. This leads to the conclusion that K_5 is not a planar graph

Rubric: (10 points)

- (5 points) *Proof by contradiction* is used in the proof.
- (5 points) The proof is correct.

6. 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:

Consider the case when $n = 2^k$, where k is a positive integer. It is obvious that $k = \log n$. The 1st, 2nd, 4th, ..., 2^kth operations will cost $1, 2, 4, \ldots, 2^k$ respectively, while all the remaining $n - (k + 1)$ operations cost 1. The total cost is

$$\begin{aligned} & n - (k+1) + (1 + 2 + 4 + \ldots + 2^k) \\ = \; & n - (k+1) + 2^{k+1} - 1 = n - (k+1) + 2n - 1 \\ = \; & 3n - (\log n + 2) \end{aligned}$$

Thus, the average cost per operation when n tends to the infinity is

$$\lim_{n \to \infty} \frac{3n - (\log n + 2)}{n} = 3$$

This leads to the result that the amortized cost of the operation is three, i.e. a constant value.

Rubric: (10 points)

- (5 points) The total cost of n operations is correct.
- (5 points) The amortized cost when n tends to the infinity is correct.

7. 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:

Notice that we only increase the size when the table is full.

We will calculate the amortized cost by using aggregate analysis. Consider that we are starting with an array of size 1 and we are inserting $n = 2k$ elements.

- In the 1st insertion, we can directly do the insertion.
- In the 2nd insertion, since the array is full, we need to allocate a new array of size $1 + 2 = 3$ and require one copy operation.
- In the 3rd insertion, we can also directly do the insertion.
- In the 4th insertion, the array is full again. The new array will be of size 5 the we need 3 copy operations.

After the observation, we can find that in the $2i$th insertion, where $i \leq k$, we will require $2i - 1$ copy operations. Suppose that both the insertion and the copy operation cost 1. Thus, the total cost will be

$$\begin{aligned} & n + [1 + 3 + 5 + \ldots + (2k-1)] \\ = \quad & n + \frac{[(2k-1)+1]k}{2} = n + k^2 \\ = \quad & \frac{n^2}{4} + n \end{aligned}$$

The amortized cost per insertion is

$$\frac{n^2/4 + n}{n} = n/4 + 1 = O(n)$$

Rubric: (10 points)

- (3 points) Analysis of each insertion's cost is correct.
- (4 points) The calculation of the total cost for n insertions is correct.
- (4 points) The calculation of the cost for n insertions is correct.

8. 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 $10 \to 1 \to 5$ and $2 \to 7 \to 3$ 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:

Let W^* the maximum value of all $s \to v$ paths' minimum edge weight. Consider the following routine: Given a threshold W, we construct a new graph, G', by removing all edges in G whose weight is less than W. Then we try to find a path from s and t by using BFS or DFS.

We can observe that if $W > W^*$, then in the new graph, we cannot find a path from s to t. If we can find that path, then this path should also a path from s to t in G, but its minimum edge weight is greater than W^*, which leads to a contradiction.

If $W \leq W^*$, then we know that there is a path from s to t, which is exactly the one in G whose edges' weights are at least W^*, which is also at least W.

Using this (monotones) property, we can solve the original problem by the following algorithm: Firstly, we sort all edges' weights by the non-decreasing order. The time complexity of this step is $O(E \log E)$ Then, we can use enumerate all weights to run the routine in the beginning to find a path with at least W. The maximum W is W^*. Since the time complexity of the routine is $O(V + E)$, the time complexity of the whole algorithm is $O((V + E)E)$

Improvement We can also use a modified binary-search to find W^*. The time complexity will be reduced to $O((V + E) \log E)$

Rubric: (15 points)

- (10 points) The algorithm should be efficient.
  - 10 points if the time complexity of the algorithm is less than $O((V + E) \log E)$.
  - 7 points if the time complexity of the algorithm is polynomial to V and E
  - 2 points if the time complexity of the algorithm is higher than the polynomial (e.g. exponential) but is clearly stated.
- (5 points) The proof of the time complexity is correct, no matter the algorithm is correct or not.

9. When we have two sorted lists of numbers in non-descending order, and we need to merge them into one sorted list, we can simply compare the first two elements of the lists, extract the smaller one and attach it to the end of the new list, and repeat until one of the two original lists become empty, then we attach the remaining numbers to the end of the new list and it's done. This takes linear time. Now, try to give an algorithm using $O(n \log k)$ time to merge k sorted lists (you can also assume that they contain numbers in non-descending order) into one sorted list, where n is the total number of elements in all the input lists. Use a binary heap for k-way merging.

Solution:

Algorithm Description To support k-way merging, we can construct a min-heap containing all the smallest elements of the k lists. Then, we remove the smallest element in the min-heap by `deleteMin` and attach it to the end of the output list. Suppose that we know the removed element comes from the ith list. (This could be implemented by attaching the list information to the element when they will be added to the min-heap) Then, if the ith list is not empty, we will move its smallest element to the min-heap.

Time complexity The time complexity of the min-heap construction is $O(k)$ since we can build it by `Heapify`. The number of the elements in the heap is at most k. Thus, the time complexity of `deleteMin` and the one of `insert` are both $O(\log k)$. And for all elements of all the k lists, we will can at most once `deleteMin` and `insert`. The total time complexity will be $O(n \log k)$

Rubric: (10 points)

- (5 points) The algorithm is stated clearly.
- (5 points) The time complexity of the algorithm is proved correctly.