CSE 431/531: Algorithm Analysis and Design — Fall 2021

Homework 1

Instructor: Shi Li — **Deadline: 9/28/2021**

Your Name: ______________________ Your Student ID: ________________

Problems	1	2	3	Total
Max. Score	20	30	30	80
Your Score				

Problem 1. For each pair of functions f and g in the following table, indicate whether $f = O(g)$, $f = \Omega(g)$ and $f = \Theta(g)$ respectively. Then prove $\lceil 10n^{1.9} \rceil = O(n^2)$.

$f(n)$	$g(n)$	O	Ω	Θ
$n^2 - 3n + 10$	n	no	yes	no
$\log_3 n$	$10\log_2(n^3)$	yes	yes	yes
$10n^2 - n$	$n^2 \log n$	yes	no	no

For every $n \geq 1$, we have

$$\lceil 10n^{1.9} \rceil \leq 10n^{1.9} + 1 \leq 10n^2 + n^2 \leq 11n^2.$$

So, for every $n \geq 1$, we have $\lceil 10n^{1.9} \rceil \leq 11n^2$. So, $\lceil 10n^{1.9} \rceil = O(n^2)$.

Problem 2. Given a *sorted* array A of size n, design an algorithm to check if there are two numbers in the array whose sum is 0. That is, decide whether there are two indices $i, j \in \{1, 2, 3 \cdots, n\}$ such that $A[i] + A[j] = 0$. (The two indices can be the same; thus if the array contains the number 0, we should output “yes”.)

Example: if the input is $(-8, -5, -2, 1, 4, 6, 8, 9)$, then the output is “yes” since $(-8) + 8 = 0$. If the input is $(-8, -5, -2, 1, 4, 6, 7, 9)$, then the output is “no”. Design an $O(n)$-time algorithm for the problem.

The algorithm is as follows:

```
1: $i \leftarrow 1, j \leftarrow n$
2: while $i \leq j$ do
3:     if $A[i] + A[j] = 0$ then return yes
4:     if $A[i] + A[j] < 0$ then $i \leftarrow i + 1$ else $j \leftarrow j - 1$
5: return no
```

Running time: in every iteration of Loop 2 except for the last one, either i is increased by 1, or j is decreased by 1. In either case, $j - i$ is decrease by 1. Initially, $j - i = n - 1$ and the while loop will break if $j - i$ becomes -1. So, the while loop will break in at most n iterations. Each iteration of the while loop takes $O(1)$ time and so the algorithm runs in $O(n)$ time.

Correctness: first, if the algorithm returns yes, then there is indeed some (i^*, j^*) pair such that $A[i^*] + A[j^*] = 0$. So, it remains to prove that if the algorithm returns no, then there is no such (i^*, j^*) pair. The contra-positive statement is the following: if there is a (i^*, j^*) pair with $A[i^*] + A[j^*] = 0$, then the algorithm will return yes.

We prove that we always have $i \leq i^*$ and $j^* \leq j$ and so the algorithm will return yes. The is clearly true at the beginning of the algorithm as $i = 1$ and $j = n$ at the moment. Focus on some iteration of the while loop; initially we have $i \leq i^*$ and $j^* \leq j$. If $A[i] + A[j] = 0$, then the algorithm terminates in the iteration. If $A[i] + A[j] < 0$, then we must have $A[i] < -A[j] \leq -A[j^*] = A[i^*]$. So $i < i^*$ and after we increase i by 1 we still have $i \leq i^*$. If $A[i] + A[j] > 0$, then we have $A[j] > -A[i] \geq -A[i^*] = A[j^*]$. Therefore we have $j > j^*$. After we decrease j by 1, will still have $j \geq j^*$.

Problem 3. A cycle in a *directed* graph $G = (V, E)$ is a sequence of $t \geq 2$ *different* vertices $v_1, v_2, \cdots, v_t$ such that $(v_i, v_{i+1}) \in E$ for every $i = 1, 2, \cdots, t - 1$ and $(v_t, v_1) \in E$. See Figure 1 for an example. Given the linked-list representation of a directed graph $G = (V, E)$ with $n = |V|$ and $m = |E|$, design an $O(n+m)$-time algorithm to decide if G contains a cycle or not; if it contains a cycle, output one (you only need to output one cycle).
You can use topological ordering as a sub-procedure. If you do, it is not necessary to write down the whole pseudo-code for the procedure.

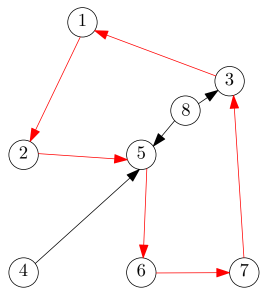

Figure 1: A cycle (denoted by red edges) in a directed graph.

For (3b), we use topological ordering to detect if a cycle exists in a directed graph. If we can topologically sort all the n vertices, then there is no directed cycle, otherwise there is a directed cycle.

Solution 1: We can assume we are given both the array of linked lists for outgoing edges, and that for incoming edges. (Given one of them, one can construct the other in $O(n + m)$ time).

Algorithm 1 Find-Directed-Cycle-via-Topological-Ordering

```
1: use the topological order algorithm learned in class on the graph $G$, to obtain an
   in-degrees $d[v]$ for vertices $v$ in the remaining graph. Return "no cycle" if all vertices
   are topologically sorted.          $\triangleright$ If $d[v] = 0$ for some $v$, then $v$ is removed.
2: for every $v \in V$ do: if $d[v] > 0$ then break
3: $seq \leftarrow$ queue of size $n$, $i \leftarrow 1, seq[1] \leftarrow v$
4: $pos \leftarrow$ array over $V$, $pos[v] \leftarrow 1$, $pos[u] \leftarrow 0$ for every $u \in V \setminus \{v\}$
5: while true do
6:     for every incoming edge $(u, v)$ of $v$ do: if $d(u) > 0$ then break
7:     if $pos[u] \neq 0$ then
8:         return $(seq[i], seq[i-1], seq[i-2], \cdots, seq[pos[u]])$
9:     else
10:        $i \leftarrow i + 1, seq[i] \leftarrow u, pos[u] \leftarrow i, v \leftarrow u$
```

Running time: Every vertex v is handled at most once in the while loop. When we handle the vertex v, the running time of step 6 in the iteration is $O(1 + d^{\text{original}}[v])$, where $d^{\text{original}}[v]$ is the incoming degree of v in the original graph. So the total running time for Step 6 is at most $\sum_{v \in V} O(1 + d^{\text{original}}[v]) = O(n + m)$. Step 1 takes time $O(n + m)$. All the other steps take time $O(n)$.

Correctness: In the remaining graph after we remove all vertices that are topologically ordered, every vertex has incoming degree at least 1. So, we can start from any vertex in the remaining graph, and walk backwards along edges in the remaining graph, until we visited some vertex for the second time. This will eventually happen since every vertex in the remaining graph has in-degree at least 1 and the graph is finite. The vertices between the two visits of the same vertex form a cycle.

Solution 2: The algorithm uses DFS. Using this algorithm to solve the problem dis-encouraged, as it is hard to prove its correctness without prior knowledge about DFS on directed graphs.

Algorithm 2 Find-Directed-Cycle-via-DFS

```
1: $visited \leftarrow$ boolean array over $V$, with $visited[v] = false$ for every $v \in V$
2: $pos \leftarrow$ array over $V$, with $pos[v] = 0$ for every $v \in V$
3: $stack \leftarrow$ array of size $n$, $top \leftarrow 0$
4: for every $v \in V$ do
5:     if $visited[v] = false$ then DFS($v$)
6: return "no cycle"
```

Algorithm 3 DFS(v)

```
1: $visited[v] \leftarrow true, top \leftarrow top + 1, stack[top] \leftarrow v, pos[v] \leftarrow top$
2: for every outgoing edge $(v, u)$ of $v$ do
3:     if $pos[u] \neq 0$ then
4:         exit the whole algorithm, by returning the cycle $stack\,[pos[u] \,..\, top]$
5:     else if $visited[u] = false$ then
6:         DFS($u$)
7: $pos[v] \leftarrow 0, top \leftarrow top - 1$
```

Running time: the recursion of DFS for a vertex v has running time $O(1 + d^{\text{out}}(v))$, where $d^{\text{out}}(v)$ is the out-degree of v. Every v will be handled in at most one recursion of DFS. So, the algorithm runs in time $O(n + m)$.

Correctness: Consider the moment before we call DFS(v) in Step 5 of Algorithm 2. Let V' be the set of unvisited vertices in V, and G' be the sub-graph of G induced by V'. Let $U \subseteq V'$ be the set of vertices that can be reached from v in G'. Then we need to prove the following two things:

(a) If U does not contain a cycle, then the call of DFS(v) will visit all vertices in U.

(b) If U contains a cycle, then the call of DFS(v) will exit the whole algorithm by returning a cycle in U.

(a) is by the nature of the DFS procedure. An ordinary DFS from a vertex v in a graph G' will visit all the vertices that can be reached from v. Now consider (b). Assume otherwise. Then DFS(v) will visit all vertices in U. Focus on the DFS tree T for the call of DFS(v). It is known that an edge $(u', v'), u', v' \in U$ can be one of the following 3 types with respect to the relationship to T:

1. (u', v') is a tree edge, i.e, an edge in T, where u' is the parent of v'.
2. (u', v') is an ancestor edge: v' is an ancestor of u' in T.
3. (u', v') a horizontal edge: neither u' is an ancestor of v', nor v' is an ancestor of u' and v' is visited before u' in the DFS procedure.

If there is an ancestor edge (u', v'), then the call to DFS will return a cycle formed by the edge (u', v') and the tree path from v' down to u'. By our assumption, there is no ancestor edge. So, all the edges are tree edges or horizontal edges. For a tree edge (u', v'), $DFS(u')$ finishes later than $DFS(v')$ does. For a horizontal edge (u', v'), $DFS(u')$ also finishes later than $DFS(v')$ does. This contradicts the fact that U contains a cycle. So we proved (b).

Finally, if U does not contain a cycle, no cycle will use vertices in U. Otherwise, some cycle will use some vertices in U and some vertices outside U. But in this case, the vertices outside U can also be reached from v and this is a contradiction with the definition of U. Therefore, in case (a) happens, we can just remove all visited vertices from consideration.