CIT 596 Online **Summer 2020**

Module 5–6

Collaborators: Han Yang Lim/ Newman Ilgenfritz

All questions require written answers. For any questions asking for an algorithm, you much include the proof of correctness and run time analysis as well. Furthermore, while we prefer algorithms written in English, you may write out pseudo code as well. However, any pseudo code must be complemented with sufficient comments so we can understand your code.

1. You are given an array $A[1..n]$ of n distinct elements such that each element in A is at most k positions away from its position in sorted order ($2 \leq k < n$). Give an $O(n \log k)$ time algorithm to sort the array. Prove your algorithm correct and analyze its running time.

   W.l.o.g. assume the array is sorted in increasing order.

   Algorithm:

   We iterate through each element of A from $1 \ldots k$, and insert each element into a min-heap data structure of size k. Once the min-heap is full (i.e. filled with k elements), we loop through the following steps for a given round i until the min-heap is empty.
   1. We perform extract_min function on the min heap, replacing the value returned at $A[i]$.
   2. With $k - 1$ elements left in the min-heap, we insert into the min-heap, the next element $A[k + i]$.
   3. We skip step 2 if there are no more elements in A that haven't been inserted in the min-heap (i.e. if $A[k + i] > A[n]$). Once the above loop finishes, we return A, now sorted in increasing order.

   Proof of correctness: The min-heap data structure gives us the minimum value of all k elements at the root of the tree every time we perform the extract_min function. When we insert elements $A[1...k]$ into the min-heap, the elements are automatically sorted into their correct position within the tree. Because each element in A is at most k positions away from its sorted order position, starting from the first element $A[1]$, we have 4 cases to consider.

   For each case with element i in position $A[j]$, at most k positions away from its sorted order position $A[l]$:

   Case 1: i in $A[1] \leq A[j] < A[l]$
   Then after the first iterations of elements inserted into the min-heap of size k, i is inside the min-heap and will be placed in its correct sorted order when it eventually percolates up to the root node.

   Case 2: i where $A[l]$ could be in range $A[j-k] \ldots A[j+k]$, and $A[l]$ is after the first k elements
   Since the min-heap is of size k, and each element $A[k + i]$ is inserted from A after extract_min is performed on the min-heap to maintain k elements within the structure, we always have the state where when it is time to insert i into $A[l]$.

   Case 3: i where $A[j] < A[l] \leq A[n]$
   By symmetry with case 1, $A[j]$ will be inserted into the min-heap before it is time to extract_min from the min-heap to add to $A[l]$. At this point the min-heap may be of size ¡ k.

   Case 4: i in $A[j] = A[l]$
   Then when $A[j]$ is inserted into the min-heap, it would immediately bubble up to the root, to be extracted and inserted into $A[l]$

   In all 4 cases, i in $A[j]$ is guaranteed to be in the min-heap when it is time to perform extract_min on i to be inserted into $A[l]$. $\therefore$ proving the algorithm returns the sorted array.

   Runtime complexity:

We first iterate through each element of A from $1 \dots k$, performing the $O(1)$ function of getting an element in the array by its index and subesquently $O(logk)$ functions to insert elements into the min-heap in each iteration.
We then continue iterating from element $A[k+1]$ until $A[n]$, this time performing the extract_min function (worst case $O(logk)$ operation), the $O(1)$ getting array by index function and again the $O(logk)$ insert function into the min-heap. The entire algorithm loops through every element in $A[1 \dots n]$ which is an $O(n)$ operation. Combining the operations above, and discarding any $O(1)$ operations, we have:

$$\begin{aligned}
&O(k) \cdot (O(1) + O(logk)) + O(n-k) \cdot (O(logk) + O(1) + O(logk)) \\
&= O(k) \cdot O(logk) + O(n-k) \cdot O(2logk) \\
&= O(logk) \cdot (O(k) + O(n-k)) \\
&= O(logk) \cdot O(n) \\
&= O(nlogk)
\end{aligned}$$

$\therefore$ the algorithm's worst-case runtime complexity is $O(nlogk)$.

2. You are given an array $A[1..n]$ of n integers. We say the array is *well-positioned* with respect to a parameter $k \in [1..n]$ iff for any k consecutive indices in the array, there are no two indices such that the value of one index is at least twice the value as the other. Give an $O(n \log k)$ time algorithm to determine if A is well-positioned. Prove your algorithm correct and analyze its running time.

Algorithm:
Iterate through the first k elements in A and insert each element into 2 different heaps, a min-heap and a max-heap. Once the heaps are filled with k elements, compare the root of both heaps. If the root of the max heap is at least twice the value of the min heap, return false.

Now let i be 1. While $A[k+i] \leq A[n]$, search and remove element $A[i]$ from both heaps and insert element $A[k+i]$. Compare the root of both heaps again.

Repeat the above, incrementing i by 1 in each round, until either the root of both heaps fail the condition or until the last element inserted into both heaps is $A[n]$.

Proof of correctness: By iterating through the first k elements in A and inserting them into separate min and max heaps, we get the min and max value of the first k consecutive indices.

By checking whether the max value is at least twice the min value, we are implicitly doing so for every value of the first k elements because the remaining $k-2$ elements are in between the max and min values, maintaining both heaps throughout the algorithm execution at size k. So if the max value is less than twice the min value, this condition holds true for every value in the first k consecutive indices.

The algorithm then searches and removes element $A[i]$ where $i = 1$ and inserts $A[k+i]$ into both min and max heaps before checking the max/min value difference again. This loop shifts the start $A[i]$ and end $A[k+i]$ indices of the array until $A[k+i] = A[n]$, covering every set of k consecutive indices in the array in an incremental fashion from $A[1 \dots n]$.

$\therefore$ the algorithm checks for the well-positioned condition for any k consecutive indices in the array.

Runtime complexity:

Iterating through the first k elements in A is an $O(k)$ operation. Max and min heap insertions where the heap is size k are a $O(logk)$ operations. Checking if the root of the max heap is at least twice the root of the min heap is a $O(1)$ operation. Obtaining the values of $A[i]$ and $A[k+i]$ from an array by index is an $O(1)$ operation. Removal of $A[i]$ and insertion of $A[k+i]$ in both heaps are again $O(logk)$ operations. Repeating the well-positioned condition check with each max/min heap pair of values continues to be a $O(1)$ operation. Iterating from $A[k+i]$ where i++ until $A[k+i] = A[n]$ is a $O(n-k)$ operation. Considering the above operations, we get:

$$\begin{aligned}
&O(k) \cdot (O(logk) + O(1)) + O(n-k) \cdot (O(1) + O(logk) + O(1)) \\
&= O(k) \cdot O(logk) + O(n-k) \cdot O(logk) \\
&= O(logk) \cdot (O(k) + O(n-k)) \\
&= O(logk) \cdot (O(n)) \\
&= O(nlogk)
\end{aligned}$$

$\therefore$ the algorithm's worst-case runtime complexity is $O(nlogk)$.

3. You are given three arrays of integers A, B, and C with n elements each as well as a target integer t. Design an algorithm to determine if there exists three numbers $a \in A, b \in B$, and $c \in C$ such that $a + b + c = t$. Prove your algorithm correct and analyze its running time. Your algorithm should run in worst-case $O(n^2)$.

Algorithm:

We iterate through all elements $a \in A$ and store them in a hash table H.

We then iterate through all possible combinations of $b + c$. In each iteration, we take the value from $t - (b + c)$ and check in H if there exists an a where $a = t - (b + c)$. If there is, we return true.

If after iterating through all possible combinations of $b+c$, an a in H isn't found where $a = t-(b+c)$, we return false.

Proof of correctness:

$$\begin{aligned}
a + b + c &= t \\
a &= t - b - c \\
a &= t - (b + c)
\end{aligned}$$

With all possible values of $b + c$, we check if there exists an a in H where $a = t - (b + c)$. If there is, then there exists a combination of a, b, c where $a + b + c = t$. $\therefore$ the algorithm will return the correct value.

Runtime complexity:

Iterating through all elements $a \in A$ is a $O(n)$ operation while storing them in a hash table H in each iteration is an $O(1)$ operation.

Iterating through all combinations of $b + c$ in B, C which are n size arrays yields a running time of $O(n^2)$. In each iteration, we search the hashtable ($O(1)$ operation) for an a that satisfies $a = t-(b+c)$ ($O(1)$ equation).

In the worst case, combining the big O notation of all operations performed in this algorithm, we have

$$O(n) + O(1) + O(n^2) + O(1) + O(1) = O(n^2)$$

4. You are given a directed graph $G = (V, E)$ with n vertices labeled 1 through n. For each vertex u, let $H(u)$ be the highest-valued vertex that is reachable from u. Design an efficient algorithm to compute $H(u)$ for each vertex in the graph.

   Algorithm:

   We use an array A of size $n+1$, with the 0-index empty, leaving the remaining array elements in indices $1 \dots n$ to store the $H(u)$ of each vertex u in its corresponding index position.

   Given the set of edges E, set a variable $i = 0$ and start from the highest valued vertex $n - i$ and get all edges that are reachable to n. Insert n into A at index positions corresponding to the vertices that have a directed edge to n (including itself).

   While A is not completely filled (excluding $A[0]$), increment i by 1. Then iterate on the next highest-valued vertex $n - i = n - 1$. Get all edges reachable to $n - 1$. Check $A[n-1]$ for the current $H(u)$ value. If $A[n-1]$ is empty, insert $n - 1$ into A at all corresponding vertices with a directed edge to $n-1$ (including at $A[n-1]$). If $A[n-1]$ is not empty, insert the current value of $A[n-1]$ instead, into all corresponding A index positions, vertices with a directed edge with $n - 1$.

   Repeat the above iteration until A is filled and return A.

   Proof of correctness:

   In order to return $H(u)$ for each u, we start with the highest-valued u at n. By getting the edges from E that have a direct edge with n, we have found the $H(u)$ of those vertices and will insert n as the $H(u)$ in their corresponding A index positions accordingly.

   Now we repeat the above process for the next highest-valued vertex $n - 1$, $n - 2$ and so on, using i as a counter in decrementing from n. Only this time, after getting all vertices with a direct edge to the current highest-valued vertex $n - i$, we check if $A[n-i]$ is empty. This gives us 2 cases.

   Case 1: $A[n-i]$ is empty.
   Then $n - i$ is the $H(u)$ of itself, and of all other vertices with a direct edge to it. We then populate A accordingly.

   Case 2: $A[n-i]$ is not empty.
   Then the vertex $n - i$ has a direct edge with some vertex $n - j$ where $0 < j < i$. If $n - j$ is reachable from $n - i$, then all vertices that reach $n - i$ will also reach $n - j$, making $n - j$ the $H(u)$ value of these vertex. We then populate A accordingly.

   This algorithm stops when $A[1 \dots n]$ is filled because at this point all values in A are the $H(u)$ of their respective vertices in G. To continue the iteration would only lower the value of the next highest vertex to be examined.

   Runtime complexity:

   Initializing A is a $O(1)$ operation. For each iteration, getting all edges reachable to a given vertex u is an $O(deg(u))$ operation assuming an adjacency list representation; so is inserting the current $H(u)$ value in question into the respective index positions in A. The check for whether a given value $A[n-i]$ is empty is a $O(1)$ operation. However the loop runs in $O(n)$ time in the worst case.

   Taking the above together, the most asymptotically significant operations are $O(n)$ and $O(deg(u))$, leading to a runtime complexity of $O(n \cdot deg(u))$.

   Initialize H(u) to 0 for all nodes. Start a BFS at the highest valued node n and update h(u) to that value. While BFS checks the adjacent nodes to n, keep track of the H(u) for all visited nodes. After all nodes that are reachable by n have been traversed, run BFS on node $n-1$. If H(U) is already computed for that node, then that node is already adjacent to some higher node than $n - 1$. If $H(u) \neq 0$, then return $H(U)$ = value of $n - 1$. Repeat BFS on $n - 2, n - i$ where i = smallest value. Output each vertex's H(u) once BFS has traversed the entire Graph.

5. Given a *tree* there is a unique simple path between any two vertices. Give an efficient algorithm to find the length of the longest such path.

   Algorithm:

   Pick a random node u in the tree and perform a modified BFS traversal that inserts a null into the queue at the end of each level. At the end of the traversal, the vertex v before the last null at the end of queue is the vertex furthest away from u.

   From u, perform the modified BFS traversal again to find w, the vertex before the last null at the end of the queue after the traversal and thus the furthest vertex away from v. While doing so, keep a count of the number of nulls being inserted in the queue after every level during the traversal.

   Return the count, which is the length of the longest path between any 2 vertices in the tree.

   Proof of correctness:

   The modified BFS traverses through every vertex in the tree from random vertex u to find v, the vertex at the end of the longest path from u. We now have the path of some pair of vertices (u, v) in the tree, which is the longest path from u. u, being randomly chosen, is part of the longest path in the tree, but not necessarily the vertex to start at to reach v. However v is one vertex of both the longest path from u and the longest path in the tree.

   With v known as one of the pair of vertices (v, w) that will result in the length of the longest path in the tree, doing another BFS search while keeping a count of the length will return us both w and the length of the longest such path in the tree, to be returned from the algorithm.

   Runtime complexity:

   Since 2 BFS are performed with other operations (such as randomly choosing a vertex u and keeping count of the length in the 2nd BFS traversal) running at O(1) time, the asymptotic complexity of the algorithm is $O(2(n + m))$, simplified to $\mathbf{O(m)}$ since we are dealing with a tree $(m \geq n - 1)$.

6. You are given a directed graph $G = (V, E)$. Let S be the subset of vertices in G that are able to reach some cycle. Design an efficient algorithm to compute S.

   Algorithm:

   We perform DFS on a random vertex u, adding vertices to a temporary set T to track visited vertexes for a given DFS tree traversal. During the DFS traversal, if a back edge is identified, we add all vertices currently in T into S. This continues until the traversal ends. After a traversal, we empty T and see if there are still unvisited vertices in G. If there were no back edges identified in the first traversal, then no vertex in T was added into S.

   If there are still unvisited vertices in this directed graph, we perform the above DFS traversal on them as well, tracking traversed vertices in T and adding them into S when a back edge is found. This continues until all vertices in G are visited.

   S is then returned.

   Proof of correctness:

   As we perform DFS traversal through the vertices, we are indirectly creating the DFS tree. When a back edge is identified, this creates a cycle c where not only are the vertices in c added to S, but also the ancestor vertices further up the DFS tree that are able to reach cycle c as well.

   If G requires multiple DFS traversals until all vertices are visited, the algorithm covers this edge case to ensure all vertices that should be in S are added.

   Runtime analysis:

   With operations such as initializing and adding to a set taking $O(1)$ time. The DFS traversals contribute most significantly to the run time of the algorithm and so the runtime complexity is $O(n + m)$ if G is not connected and simplified to $O(m)$ if G is connected.

7. You are given a directed acyclic graph $G = (V, E)$, where each vertex v that has indegree 0 has a value $value(v)$ associated with it. For each other vertex u, define $Pred(u)$ to be the set of vertices that have incoming edges to u. We now define $value(u) = \sum_{v \in Pred(u)} value(v)$. Design an efficient algorithm to compute $value(u)$ for all vertices u. Analyze the running time of your algorithm and prove it correct

Algorithm:

Initialize a dictionary D with all vertices $u \in V$ as the key and a null $value(u)$ as the value in this key-value pair.

Perform a DFS traversal on each v. For all reachable nodes u of each v, look up the relevant vertex u's key and add $value(v)$ to its current value before marking the vertex u as visited. Repeat this as the DFS traversal proceeds with each subsequent vertex u and adding $value(v)$ to the current $value(u)$.

Return D once DFS traversal performed on all v in G.

Proof of correctness:

D first starts with only a null $value(u)$ for all $u \in V$. The DFS traversal of the first source vertex v will cover all vertices u reachable from v, adding the respective $value(v)$ to the current $value(u)$ sum held in the key-value pair of each u in D. This also covers the case where we have multiple u vertices. For example we have a v reaching a u_1 that reaches a u_2. When we add $value(v)$ to u_1, due to the edge from u_1 to u_2, and because $value(u_2)$ is the sum of all $value(u)$ that has edges to u_2. $value(v)$ will rightly be added to the current value of $value(u_2)$ as well.

At this point, for any u, $value(u)$ would be $\sum_{v \in Pred(u)} value(v)$ unless vertices u are missing because they were not covered by the DFS traversal from first source vertex v. Subsequent traversals (with $value(v)$ added to the $value(u)$ of adjacent u nodes) will cover all reachable vertices for every u and add the respective $value(v)$ to the $value(u)$ of each u traversed, returning $value(u)$ for all vertices u in the form of a completed D.

Runtime complexity:

Operations pertaining to the creation, and updating of values in D are performed in O(1) time.

The most significant contributor to the runtime complexity is thus the DFS traversals of all source vertices in G. Which is $O(k(n+m))$ where k is a constant representing total number of source vertices in G. $\therefore$ the worst case runtime complexity is $O(n+m)$ if the graph is not connected and $O(m)$ if G is connected.