COMPSCI 311: Introduction to Algorithms **Spring 2020**

Homework 3

Released 2/25/2019 Due 3/11/2019 11:59pm in Gradescope

Instructions. You may work in groups, but you must write solutions yourself. List collaborators on your submission. Also list any sources of help (including online sources) other than the textbook and course staff.

If you are asked to design an algorithm, please provide: (a) the pseudocode or precise description in words of the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1. **(20 points) Three Friends and a Car**

Algorithm: Take a set of vertices V as input, for each contains the connected edges.

```
input V
input start, end as start and target node index as integer

function trackPath(prev, start, end):
  path = list()
  v = end
  while v != start or v[1] != 0:
    path.add(v)
    v = prev[v]
    if v is null:
      return null
    end if
  end while
  path.add(v)
  return reversed(path)
end function

function dijkstraMod3(V, start, end):
  input start, end as index of the start/end node
  set A as Priority Queue
  foreach v in V:
    for m = 0 to 2:
      d'[(v, m)] = INF // (index, modular)
      prev[(v, m)] = null
    end for
  end for
  d'[(start, 0)] = 0
  while A not empty:
    extract node v in A with smallest d'[v] value
    set d = d'[v]
    for all edges (v[0], w) where w in A:
      if d + l(v, w) < d'[w]:
        node = (w, modular_cycle[v[1]])
        d'[node] = d + l(v, w)
        prev[node] = v
      end if
```

```
      end for
    end while
    path = trackPath(prev, start, end)
    if path is null:
      exception 'no solution'
    length = d'[(end, 0)]
    return path, length
  end function
```

Simply expand each nodes into 3 nodes to express the three modular $0, 1, 2$, and the new connection should be based on the segment modular, like $p_0 \to q_1$, then do a Dijkstra's Algorithm on the new graph.

intuition: It is quick to come up with Dijkstra's Algorithm as it is about shortest path. For each stop, it has three possibilities: mod 3 = (0, 1, 2) to the target, so during the search, I expand all these three possibilities. Because they should drive equal segments, the final modular when reach the end stop should be 0.

Proof: Ground truth: Dijkstra's Algorithm is correct.

Suppose that there is a shortest path that num of edges divisible by 3 but not found by the algorithm. The expansion of this graph expand each node p into three nodes p_0, p_1, p_2 where $0, 1, 2$ are modular. Consider two nodes p, q which are connected directly by an edge, this edge is changed into directed edges that connects $(p_0, q_1), (p_1, q_2), (p_2, q_0), (q_0, p_1), (q_1, p_2), (q_2, p_0)$ which makes a cycle for $p_0, q_1, p_2, q_0, p_1, q_2, p_0$, means that these two nodes are still a strongly connected, means any path found in the original graph will not be broken by the change, so if there exists some paths that fulfills the requirement, this path can also be found in the changed graph, which is contradictory to the assumption.

Suppose that there is not such a path but the algorithm gives such a path. If the start and end u, v are connected, there must be a path which has number of edges divisible by 3, because there must be an arbitrary path $P(u, v)$ which passes a node q that there is an edge $E(q, v)$. Consider the length of $P(u, v) mod 3 = m, m > 0$, we have $P(u, v) + 2 * E(q, v) = P_1(u, v), m_1 = (m+2) mod 3$, $P(u, v) + 4 * E(q, v) = (m + 4) mod 3 = (m + 1) mod 3$. So, if v cannot be reached by u by a path has number of edges divisible by 3, u, v must not be in the same connected component. The new graph does not change the connectivity of the original graph, so there must be no path between u, v in the new graph, which is contradictory to the assumption.

Because the algorithm will find the required shortest path if it exists, and it will not find if it does not exist, this algorithm is correct.

Complexity: Same as the original Dijkstra's Algorithm, the main operations are the extract-min and update-key. Because there are $3n$ elements in the matrix, the extract and update should be $O(3n \log 3n), O(m \log 3n)$. As a result, the total complexity is $O((3n + m) \log 3n) = O((m + n) \log n)$. The path track function should be $O(3n)$, because there are maximum $3n$ elements in the list, and there is no cycle.

2. **(20 points) Bottleneck Edge**

a) Consider MST (Maximum Spanning Tree) and MCST (Maximum Capacity Spanning Tree) Consider $w(u, v) = a$ as the smallest weight in MST, $w'(u', v') = b$ as the smallest weight in MCST.

If $a > b$, we can argue that the there are some MCST that have larger smallest weight, give larger capacity, conflict to the definition of MCST.

If $a < b$, we consider a cut on (u, v), which can create two sub-trees. Consider in the MCST, there is another edge connects (p, q) where $w'(p, q) = c$. As (u', v') has smallest weight, we have $c \geq b$. Because (u, v) is in MST while (p, q) not, we have $c \leq a$, so we have $a \geq c \geq b$, which is conflict with $a < b$.

As a result, the MST on G is a MCST, means it contains a path of maximum capacity between any two nodes.

b) **Algorithm:** Apply a BFS search from s to t, only select edges with $weight > C$, to find if there is a path. If there is not, then the capacity is at most C.

Intuition: It is easy to think that to form a path with capacity at most C means there should not be a path with all weights $> C$. So, if we can find such a path, means the capacity is greater than C, otherwise the capacity is less than C.

Proof: Consider the definition of the capacity between s, t, it is the maximum capacity of any path between s, t. To find whether the capacity is at most C, we can find its negation which is the capacity is at least C (not include), and the solution is its negation. Because the algorithm only considers the edges that $weight > C$, any path found by the BFS must have capacity $> C$. Consider that there is a path has capacity $> C$ not found by the algorithm. Then, there is a path that all edges have $weight > C$, but not included in the algorithm, which is impossible because any edge $weight > C$ and its end points can be reached by a path from s will be considered by the algorithm.

Complexity: $O(m + n)$, because it is simply a BFS search. The comparison is $O(1)$, so in loose bound, it is $O(m + n)$.

c) **Algorithm:** Use Dijkstra's Algorithm. Modify it so that $d'(v)$ initialized by -INF and track the minimum weight in the path from s to v, and update it if some other path find a larger minimum weight. To be more specific, consider $W(u, v)$ as the weight of the edge that directly connects u, v, $d'(v) = max(d'(v), min(d'(u), W(u, v)))$. The priority should take the capacity in decreasing order.

Intuition: It is easy to come up with Dijkstra's Algorithm, but change the shortest path to the maximum min-weight path.

Proof: Consider Q as the priority queue kept in the algorithm, A is the set of nodes still to explore, $S = V \setminus A$ Set $C(path)$ be the capacity of the path.

Claim: 1. For $v \in S, d(v)$ is the maximum capacity from s to v.

2. For $v \in A$, $d'(v)$ is the maximum capacity $s \to v$ with all nodes in S except v

Base Case: Initially $S = \emptyset$, 1. is true, 2. is empty path, $d'(s) = \infty$

Induction: Let v be the next node added to S, $d'(v) = \max_{w \in A} d'(w)$, so we have $d(v) = d'(v)$ is the maximum capacity as claim 1., with all prior nodes in S.

Consider that a path with some prior nodes not in S, let $y \in A$ be the first such node in $s \to y \to v$. $C(s \to y \to v) \leq C(s \to y) \leq d'(y) \leq d'(v)$, because the capacity can only go smaller on extending the path, as if the new edge is larger or equal than prior ones, the capacity will not change, and if the new edge is smaller than prior ones, it will become the capacity, because capacity is determined by the smallest weight. So, we cannot have a larger capacity.

Consider v in S, we get new similar paths $s \to u$ for all (v, u), update $d'(v) = max(d'(v), min(d'(u), W(u, v)))$ maintains the invariant.

Complexity: $O((m + n) \log n)$, because the priority queue will act the same compare to the original one. The min operation should be m time, so is $O(m)$, and the complexity is still $O((m + n) \log n)$.

3. **(20 points) Paintball**

(a) **Algorithm:**

```
input array H as the heights
left_target = array(size=n)
right_target = array(size=n)
for i = 1 to n:
  for j = i - 1 to 1 step -1:
    if H[j] > H[i]:
      left_target[i] = j
      break
    end if
  end for
  for j = i + 1 to n:
    if H[j] > H[i]:
      right_target[i] = j
      break
```

```
    end if
  end for
end for
return left_target, right_target
```

Intuition: Just do a brute force search on left and right for all players.

Proof: This algorithm first pick one player, then search from it to the two endings of the array, stop at the first element that is larger than it, so it must be the closest left/right larger element.

Complexity: $O(n^2)$, because it is just two nested loops with range n, obviously $O(n^2)$

(b) **Algorithm:**

```
function left_right_larger(array H) -> array, array:
  input array H as the heights
  array left_target = array(size=n, init=-1)
  array right_target = array(size=n, init=-1)

  function left_right_sub(arr, start, end):
    if arr.length < 2:
      return arr

    integer mid = (1 + arr.length) / 2
    array left_arr = arr[1, 2, ..., mid]
    array right_arr = arr[mid + 1, mid + 2, ..., end]
    array left_arr = left_right_sub(left_arr, start, start + mid - 1)
    array right_arr = left_right_sub(right_arr, start + mid, end)

    integer i = 1, j = start + mid, max_j = H[start + mid]

    while i <= arr.length and j <= end:
      max_j = H[j]
      li = left_arr[i]
      if H[li] < max_j:
        if right_target[li] == -1:
          right_target[li] = j
        end if
        i++
      else:
        j++
      end if
    end while

    i = 1, j = start + mid - 1, max_j = H[start + mid - 1]

    while i <= arr.length and j >= start:
      max_j = H[j]
      ri = right_arr[i]
      if H[ri] < max_j:
        if left_target[ri] == -1:
          left_target[ri] = j
        end if
        i++
      else:
        j--
      end if
    end while
```

```
    i = 1, j = 1
    integer k = 1
    array sort_arr = array(size=arr.length)

    while i <= left_arr.length and j <= right_arr.length:
      li = left_arr[i], ri = right_arr[j]
      if H[li] < H[ri]:
        sort_arr[k] = left_arr[i]
        i++
      else:
        sort_arr[k] = right_arr[j]
        j++
      end if
      k++
    end while

    for i = i to left_arr.length:
      sort_arr[k] = left_arr[i]
      k++
    end for

    for j = j to right_arr.length:
      sort_arr[k] = right_arr[j]
      k++
    end for
    return sort_arr
  end function

  return left_target, right_target
end function
```

For each sorted subarray by weight ascending order, check its left/right array by the original index. The sorted is done by merge sort. The index is recorded in the recursion.

Intuition: The intuition of this algorithm is that the taller player will always search equal or further than a shorter player if they stand on the same position. In the algorithm, I divide the array by half recursively, and when combine them, the left array, in the order from shorter to taller, will search on the original order of the right array (if the current element in the right array is smaller than the current element in the left array, shift the right mark), and same for the right array except the search will search from the end to the begin of the left array. For example, when combing [4, 1], [2, 5], for left array, it will first take 1 and take 2 as the right target of 1, then it goes to 4, but find 2 is smaller than 4, so right mark will shift to right to 5, and assign as 4's right target. For right array, it will be $2 \to 5, 1 \to 4$.

-1 means no target.

Proof: Base case: n = 1, only 1 player, no shot, correct

Hypothesis: the left and right targets are correct in the array

Proof: For n = k, there two sorted arrays (order by the weights) which are select from 1 to $\lfloor k/2 \rfloor$ which is a and selected from $\lceil n/2 \rceil$ to k which is b. We know that every element in a is on the left of b.

If the target is assigned in the sub array, it must also be correct in the combined array, because for a, the right targets are chosen from a, must be closer than the elements in b, and the left targets are not affected because there is no array append on the left of it, so no new elements to consider. The proof for b is the same, just exchange the left and right.

We iterate through a by weight increasing order, set the weight as w_{a_i}. For each a_i, we take the index from b by index order as right target if $w_{b_j} > w_{a_i}$ and a_i has not assigned the right target. If this condition not satisfied, we shift the b_j to b_{j+1}.

In this process, the new right targets must be correct, because $w_{a_i} > w_{a_j} \Rightarrow rtarget_{a_i} \geq rtarget_{a_j}$ if a_i, a_j have no right targets, and a_i always chooses the proper right target from b as closer as possible. The proof for b is the same, just exchange the left and right.

As a result, as the new left/right target for sub arrays a, b are correct, the combined array has the correct left/right targets. The hypothesis is correct for $n = k$, so does the algorithm.

Complexity: $T(n) = O(n \log n)$. The algorithm will always divide the array into half, so that is $T(n/2)$ In each sub node, it will iterate 2 times through the array, so the calculation out of recursion is $O(n)$. Then, we have $T(n) = T(n/2) + O(n) = O(n \log n)$

(c) Consider the array is divided into $\lceil n/2 \rceil$ with sub arrays all length 2 except 1 if n is odd, then, there are $\lfloor n/2 \rfloor$ sub arrays with length 2. For each sub array, because the weights are distinct, one of the element must be larger than another, so it will be shot at the first round. Because there are $\lfloor n/2 \rfloor$ such sub arrays, at least $\lfloor n/2 \rfloor$ elements are shot at the first round.

(d) **Algorithm:**

```
function get_rounds(array H):
  input H as the array of heights
  if H.length < 2:
    return 0
  end if
  list local_min = []
  if H[1] < H[2]:
    local_min.add(H[0])
  end if
  for i in 2 to H.length - 1:
    if H[i] < H[i - 1] and H[i] < H[i + 1]:
      local_min.add(H[i])
    end if
  end for
  if H[H.length] < H[H.length - 1]:
    local_min.add(H[H.length])
  end if
  return get_rounds(local_min) + 1
end function
```

Get the local minimum in each round, and keep them for the next round.

Intuition: The intuition is that at each round, the players that are not targeted are the local minimums of the heights. So, for each round, we get the local minimum and keep simulate this process recursively, until there are only one element in the array.

Proof: Base Case: 1 player, no target, so 0 rounds, correct.

Because this is a simulation, we only need to prove that for each round, a player is not shot iff the height h_i is local minimum. If it is a local minimum, means $h_{i+1} > h_i, h_{i-1} > h_i$, so h_i will not be chose by the players next to it. For an arbitrary elements h_j that are not h_{i+1}, h_i, h_{i-1}, even when it is smaller than h_i, they will not choose h_i as target because $h_{i+1}, h_{i-1} > h_i > h_j$, and either h_{i+1} or h_{i-1} is closer to h_j than h_i. If it is not a local minimum, means $h_{i-1} > h_i > h_{i+1}$ or $h_{i-1} < h_i < h_{i+1}$, it will be chosen as a target by either h_{i-1} or h_{i+1}, and will be shot in this round.

So, we can prove that local minimum height players will be kept in each round. Keep doing this until reach base case, because each step is correct and the base case is correct, the algorithm is correct.

Complexity: $O(n)$. $T(n) = T(\lceil n/2 \rceil) + n$ because for each round we iter through the whole input array, and we have proved that at least $\lfloor n/2 \rfloor$ players will be shot each round. Because $\log_2 1 < 1$, $T(n) = O(n)$

4. **(20 points) Recurrences**

a) $T(n) = 2T(n/3) + 2T(n/9) + n \Rightarrow 2 * (n/3) + 2 * (n/9) = \frac{6}{9} + \frac{2}{9} = \frac{8}{9}$

$T(n) = 2(2T(n/9)+2T(n/27)+n)+2(2T(n/27)+2T(n/81)+n)+n = 4T(n/9)+8T(n/27)+4T(n/81)+5n$

$\Rightarrow \frac{36n}{81} + \frac{24n}{81} + \frac{4n}{81} = \frac{64n}{81}$

With this recurrences tree, we can say that $T(n) = n + \left(\frac{8}{9}\right) n + \left(\frac{8}{9}\right)^2 n + \cdots$ $T(n) \leq \frac{1}{1-8/9} n = 9n = \Theta(n)$

b) $T(n) = \frac{1}{3}(T(n-1) + T(n-2) + T(n-3)) + cn \Rightarrow \frac{cn}{3} + \frac{cn}{3} + \frac{cn}{3} = cn$

$T(n) = \frac{1}{3}\left(\frac{1}{3}(T(n-2) + T(n-3) + T(n-4)) + \frac{1}{3}(T(n-3) + T(n-4) + T(n-5))\frac{1}{3}(T(n-4) + T(n-5) + T(n-6))\right)$

$\Rightarrow \frac{1}{3}\left(\frac{1}{3}(cn + cn + cn) + \frac{1}{3}(cn + cn + cn) + \frac{1}{3}(cn + cn + cn)\right) = cn$

With this recurrences tree, we can say $T(n) = cn + cn + \cdot + cn = ncn = cn^2 = \Theta(n^2)$, because that is $n - 1$, so the tree depth should be n

c) $T(n) = T(n/2) + T(n/3) + T(n/6) + n \Rightarrow \frac{3n}{6} + \frac{2n}{6} + \frac{n}{6} = n$

$T(n) = (T(n/4)+T(n/6)+T(n/12)+n)+(T(n/6)+T(n/9)+T(n/18)+n)+(T(n/12)+T(n/18)+T(n/36)+n)$

$\Rightarrow \frac{n}{4} + 2\frac{n}{6} + 2\frac{n}{12} + \frac{n}{9} + 2\frac{n}{18} + \frac{n}{36} = n$

With this recurrences tree, we can say $T(n) = n + n + \cdots = \Theta(n \log n)$, because the deepest branch would have the depth of $\log_6 n$, while the shallowest branch would have the depth of $\log_2 n$, so we can conclude that $n \log_6 n \leq T(n) \leq n \log_2 n$.

d) Consider $m = \log_2 n - 1, n = 2^{m+1}, \sqrt{2n} = 2^{\frac{m}{2}+1}, \sqrt{n} = 2^{\frac{m+1}{2}}$

Consider $S(m) = \frac{T(2^{m+1})}{2^{m+1}} = \frac{T(n)}{n} = 2\frac{T(\sqrt{2n})}{\sqrt{2n}} + \frac{1}{\sqrt{n}} = 2S(\frac{m}{2}) + \frac{1}{2^{\frac{m+1}{2}}}$, master theorem form, in which $a = 2, b = 2, d = 0$, because $\frac{1}{2^{\frac{m+1}{2}}}$ decreases as m increases. So, $S(m) = \Theta(m^{\log_2 2}) = \Theta(m)$. As a result, we have $T(n) = nS(m) = n\Theta(\log n - 1) = \Theta(n \log n)$

5. **(20 points) Largest Perfect Tree**

Algorithm:

```
function max_perfect_tree(Node T):
  input T as the Tree

  integer max_perfect_depth = 1
  integer node_depth = 1
  Node max_perfect_node = T

  function max_tree(Node tree, integer depth):
    if tree.left == null:
      return 1
    end if
    integer left_perfect = max_tree(tree.left, depth + 1)
    integer right_perfect = max_tree(tree.right, depth + 1)
    integer perfect_depth = min(left_perfect, right_perfect) + 1
    if perfect_depth >= max_perfect_depth:
      max_perfect_depth = perfect_depth
      node_depth = depth
      max_perfect_node = tree
    end if
    return perfect_depth
  end function

max_tree(T)
return node_depth

end function
```

Get the left/right sub perfect trees and form a new perfect tree for the current node.

Intuition: The intuition is that if the subtree of a node are perfect trees, then this tree can also be a perfect by cutting all the nodes that is deeper than the max depth of the smaller subtree.

Proof: Set $D(T)$ as the depth of the tree

Base case: Single node tree, perfect with depth 1, correct

Hypothesis: Attach two perfect trees to a node will create a perfect tree if remove all nodes that is lower than the smaller perfect tree, and it is the largest perfect under this node.

Proof: Consider node V with two sub perfect trees V_l, V_r. Consider $D(V_l) \leq D(V_r)$. Because V_l, V_r are perfect, we know that for depth $\leq D(V_l)$, $D(V_l)$ and $D(V_r)$ have the same structure, because for an arbitrary depth, there is only one perfect binary tree structure. Consider V_r' as V_r remove all the nodes with depth $> D(V_l)$. In V_r' and V_l, all the nodes except the leaves have two children because they are perfect binary trees. Attach the root of V_r', V_l to V will create a new perfect tree with root V, because in this process V have two children V_r', V_l, so all nodes except the leaves have two children, is a perfect binary tree. If we add any subset of nodes from $V_r \setminus V_r'$, it will create unbalance and break the perfect tree because any nodes in V_l has less depth than those nodes. So, this tree rooted in V is the largest perfect tree can be formed on root V. As a result, this process can get the largest perfect tree for each node as root.

Out of the recursion, we get the max of the perfect subtrees rooted on each node, and this must be the largest perfect subtree on the whole tree, because it is a brute force search on all nodes.

Complexity: $O(n)$. Because this algorithm simply reaches all the nodes in the tree one time each. We can see that the iteration part is just go into the left/right subtree of a node, and the non-recursion part is just $O(1)$ because there are just some comparison and variable assign. So, $T(n) = 2T(n/2) + O(1) = O(n)$

6. **(0 points).** How long did it take you to complete this assignment?