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

Homework 4

Released 3/15/2020 Due 4/1/2020 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.

Efficiency. The algorithms you design should be polynomial-time, unless something else is stated.

1. **(20 points) Longest Weighted Paths**

(a) **Algorithm:**

```
input G as the graph in which G[u, v] means the weight of the edge u->v
output distances and the previous vertex of nodes as an array
function largest_weight(G):
  R = topological_sort(G) // get the topological order from the standard topological sort
  dist = array(size=N, init=0)
  from_node = array(size=N)

  for u in R:
    foreach v in get_neighbors(u):
      if dist[v] < dist[u] + G[u, v]:
        dist[v] = dist[u] + G[u, v]
        from_node[v] = u
      end if
    end for
  end for
  return dist, from_node
end function

output maximum path
function maximum_weight_path(G):
  dist, from_node = largest_weight(G)
  node = argmax(dist) // get one node with the maximum distance
  path = []
  while node != null:
    path.add(node)
    node = from_node[node]
  end while
  return path.reverse()
end function
```

Simply search from the topological order, and record the maximum distance and the node it comes from in each node.

Intuition: Because it is a DAG, then the path must be from the nodes that has no in-edges. So, it naturally comes to iterate through the topological order of DAG, and for each node, check the vertices it goes to and check if the path length (maximum path length that reaches there) is longer than the other path that has went through that vertex or there is no has went there. Therefore, we can find the maximum weight to each node, and we can retrieve the maximum path by select the max distance node and get its from_node reversely.

Proof: In a DAG, a largest weighted path always starts from a source node and ends at a sink node. Consider a path $u \to v$ that is not starts from a source node or not ends at a sink node. Because we have positive integer weight, we can always find some path $u' \to u \to v$ or $u \to v \to v'$ for this path which has larger weight. However, when this path starts from source vertex and ends at sink vertex, we cannot attach more edges to it, so we cannot find a longer path that contains this path.

Based on this principle, we just need to calculate the largest weighted path from every source vertices to every sink vertices.

Consider a start node s, $d(x)$ is the distance from s to x, $\sigma(x)$ is the longest distance from s to x

Base case: $|G| = 1$, $G = s, d(s) = \sigma(s) = 0$, correct Hypothesis: Consider u be the last vertex get from R. Consider R' contains all previous vertices and u. $\forall x \in R' : d(x) = \sigma(x)$. Also, they all record their previous nodes of the largest weight path reach them. Proof: Because the hypothesis, we have $forall x \in R' \wedge x! = u : d(x) = \sigma(x)$. So, we only need to show $d(u) = \sigma(u)$.

u must be a sink node because of the nature of DAG and topological order.

In the algorithm, we are expanding the nodes by topological order, so $d(u)$ will be fixed only after all nodes that can be reached to it has been expanded. It means, to get correct $d(u)$, we only need to get the $d(u) = max(d(x) + l(x, u))$ where x is the set of nodes adjacent to u. $d(u) = \sigma(u)$, because we exhaust all the paths to u and accept the path with the maximum distance.

Also, during this process, we record that node x' as the previous node of the u in the maximum weight path. $x' \to u$ is the last edge of the largest weight path $s \to u$.

So, in this algorithm, all path to each node is the maximum path to that node. Because all the path starts from source nodes, as proved before, they have the potential to become the maximum weight path (consider removing all lower level nodes for each nodes, these nodes will become sink nodes). Therefore, we can retrieve this path by reversely get the recorded previous node of the node with the largest recorded weight.

Complexity: $O(m + n)$. Topological sort is $O(m + n)$. For the nested loop in largest_weight, the outer one iterates through the vertices, and the second one iterates through the edges, so their asymptotic complexity is $O(m + n)$. For the loop in maximum_weight_path function, it cannot be more than $O(n)$ because it is a DAG with n nodes, so the path cannot retrieve more than n nodes. So, the total complexity is $O(m + n)$

(b) **Algorithm:**

```
input G as the graph in which G[u, v] means the weight of the edge u->v
    If no edge, the weight is 0.
output distances and the previous vertex of nodes as an array
function num_max_path(G):
  R = topological_sort(G) // get the topological order from the standard topological sort
  dist = array(size=N, init=0)
  max_path = array(size=N)

  for u in R:
    if max_path[u] == 0:
      max_path[u] = 1
    foreach v in get_neighbors(u):
      if dist[v] < dist[u] + G[u, v]:
        dist[v] = dist[u] + G[u, v]
        max_path[v] = max_path[u]
      else if dist[v] == dist[u] + G[u, v]:
        max_path[v] += max_path[u]
      end if
    end for
  end for

  // get the number of the maximum from the memory
  result = max_path[1]
  max_dist = dist[1]
  for u = 2 to N:
    if max_dist < dist[u]:
      result = max_path[u]
      max_dist = dist[u]
    else if max_dist == dist[u]:
      result += max_path[u]
    end if
  end for
  return result
end function
```

Intuition: If the previous node has n maximum weight paths end there, and there is m edges with same weights connects the current node which is not visited, then there are $m * n$ maximum weight paths end in current node. Based on this idea, I used an array to record the number of maximum weight path of each node. If there is a path have larger weight, then re-initialize the current number of maximum path.

Proof: In a), we proved that the algorithm gives the largest weight path from G. So, in the process, we can record how many largest weight paths go to each node. The algorithm is to sum the paths that can reach this node at maximum distance. For node u, if it has C_u paths, and there is $E_{u,v}$, then $C_v = C_v + C_u$ because there is no cycle.

Complexity: $O(m + n)$. The structure of this algorithm is the same as the algorithm in part(a) for the first part which is $O(m + n)$. The second part is a loop of $O(n)$, so the total complexity is $O(m + n)$.

2. **(20 points) Maximum Parenthesized Sum**

Algorithm:

```
input V as the value [1,..,N] // I accidentally used the wrong index system
input OP as the operators [1,..,N-1]
function max_parenthesized_sum(V, OP):

  N = V.length
  memo_p_max = matrix(size=(N, N), init=set())
  memo_p_min = matrix(size=(N, N), init=set())
  memo_v_max = matrix(size=(N, N), init=-INF)
  memo_v_min = matrix(size=(N, N), init=INF)

  function calc(a, op, b):
    if op == '+':
      return a + b
    else if op == '-':
      return a - b
    end if
  end function

  input u as sub array of V
  input p as sub array of OP
  input start as the start of u in V
  function helper(u, p, start):
    if u.length == 1:
```

```
    memo_v_max[start, start] = u[0]
    memo_v_min[start, start] = u[0]
    return v[0]
  end if

  max_v, min_v = -INF, INF
  N = v.length
  max_p, min_p = -1, -1
  left_max_p, right_max_p = set(), set()
  left_min_p, right_min_p = set(), set()

  for i = 1 to N:
    l, r = u.subarray(1, i), u.subarray(i + 1, N) // here subarray only affects the indexing behavi
                                                  // instead of copy the array
    lo, ro = p.subarray(1, i - 1), p.subarray(i + 1, N)
    ls, le = start, start + i - 1
    rs, re = le + 1, start + N

    if memo_v_max[ls, le] > -INF:
      max_l = memo_v_max[ls, le]
    else:
      max_l = helper(l, lo, ls)
    end if

    if memo_v_max[rs, re] > -INF:
      max_r = memo_v_max[rs, re]
    else:
      max_r = helper(r, ro, rs)
    end if

    if memo_v_min[ls, le] < INF:
      min_l = memo_v_min[ls, le]
    else:
      min_l = helper(l, lo, ls)
    end if

    if memo_v_min[rs, re] < INF:
      min_r = memo_v_min[rs, re]
    else:
      min_r = helper(r, ro, rs)
    end if

    o = op[i - 1]

    res1 = calc(max_l, o, max_r)
    res2 = calc(max_l, o, min_r)
    res3 = calc(min_l, o, max_r)
    res4 = calc(min_l, o, min_r)

    result_max = max(res1, res2, res3, res4)
    result_min = min(res1, res2, res3, res4)

    if result_max > max_v:
      max_p = i
      max_v = result_max
      if chose res1 or res2:
        left_max_p = memo_p_max[ls, le]
      else:
        left_max_p = memo_p_min[ls, le]
      end if

      if chose res1 or res3:
        right_max_p = memo_p_max[rs, re]
      else:
        right_max_p = memo_p_min[rs, re]
      end if
    end if

    if result_min < min_v:
      min_p = i
      min_v = result_v
      if chose res1 or res2:
        left_min_p = memo_p_max[ls, le]
      else:
        left_min_p = memo_p_min[ls, le]
      end if

      if chose res1 or res3:
        right_min_p = memo_p_max[rs, re]
      else:
        right_min_p = memo_p_min[rs, re]
      end if
    end if
  end for

  s, e = start, start + N - 1
  left_top_p = [start, start + max_p - 1]
  right_top_p = [start + max_p, start + N]

  if left_top_p[1] != left_top_p[2]: // no sense to add on a single number
    memo_p_max[s, e].add(left_top_p)
  end if
  if right_top_p[1] != right_top_p[2]:
    memo_p_max[s, e].add(right_top_p)
  end if

  left_top_p = [start, start + min_p - 1]
  right_top_p = [start + min_p, start + N]

  if left_top_p[1] != left_top_p[2]:
    memo_p_min[s, e].add(left_top_p)
  end if
  if right_top_p[1] != right_top_p[2]:
    memo_p_min[s, e].add(right_top_p)
  end if



  memo_p_max[s, e].add_all(left_max_p) // consider it just links a pointer
                                       // from memo_p_max[s, e] to left_max_p
```

```
                                             // which should be O(1)
      memo_p_max[s, e].add_all(right_max_p)
      memo_p_min[s, e].add_all(left_min_p)
      memo_p_min[s, e].add_all(right_min_p)

      memo_v_max[s, e] = max_v
      memo_v_min[s, e] = min_v

      return max_v
    end function

    max_value = helper(V, OP, 1)
    parenthesis = memo_p_max[1, N]
    return max_value
  end function
```

Intuition: For a array of numbers a, it can be separated into two parts l, r, with between an operator o either + or -. When the operator is +, l, r should be the largest, however when the operator is -, l should be the largest and r should be the smallest. So, in the algorithm, I keep two memory space to keep track on the smallest and largest value of a segment. Also, for a, it should have parentheses of all parentheses in l, r and the parentheses that separate l, r, for example, $(l) - (r)$.

Proof: Base case: single value, no parentheses required, max min is this value, correct.

Subproblem: For a array of values and ops, split the value many times at all possible indexes, and take the operator between the two set of values, feed them to the sub-subproblem, and should get the max and min of these set of values by placing parentheses.

Proof: Consider an array of values A is splitted into B, C with $A = BopC$ where op is the operator between B, C Because we know $min(B), max(B), min(C), max(C)$, we know that if the operator is +, then $max(A) = max(B) + max(C), min(A) = min(B) + min(C)$. When $op = -$, we have $max(A) = max(B) - min(C), min(A) = min(B) - max(C)$. Because we take the split in any possible indexes in A and take the max, min of the intermediate results, we can have the $max(A), min(A)$, so the subproblem is correct.

Complexity: The complexity should be $O(n^3)$. There are $O(n^2)$ subproblems in this question. Consider a array of number of size n, it can add parentheses at $(1,1), (1,2), .., (1,n), (2,2), (2,3), .., (2,n), .., (n,n)$. So, the total number of subproblems is $n + (n-1) + (n-2) + \cdots + 1 = O(n^2)$. For each subproblem, becuase there is a loop of $O(n)$ while all other calculations are $O(1)$, the total complexity is $O(n^2 * n) = O(n^3)$

3. **(20 points) Splitting into Words**

a) U = str.substring(i, j) means take the substring of S as $U = S[i..j]$ **Algorithm:**

```
  input d as the dictionary of the words
  function splitting_into_words(d, S):

    memo = array(size=N, init=0)

    input w as a substring of S
    input start as the start index of the substring
    function helper(w, start)
      if w.length == 0:
        return 1
      end if
      if memo[start] != 0:
        return memo[start]
      end if
      for i = 1 to w.length:
        prefix = w.substring(1, i)
        if prefix in d:
          memo[start] += helper(w.substring(i + 1, w.length), start + i)
        end if
      end for
      return memo[start]
    end function

    return helper(S)
  end function
```

Intuition: The main idea is try to list the words sequentially, if the last remains is a word then it is a valid split. To find the word, scan through the string from the beginning to the end, and check if the substring s from the beginning to the current index is a word. If it is a word, then recursively do this to the string $S - s$.

Proof: Base Case: 0 length string, get 1 way of split, correct

Subproblem: For each valid prefix p in S, take the $S - p$ as the new subproblem with result r. Therefore we have $r[1..n]$. The result is the sum of r

Proof: the subproblem can get the optimal of the problem, because clearly, for one prefix, if the suffix can have u different ways of splits, then for this prefix, it can have u different ways of splits, because the overall split is just append the prefix at the beginning for all the different splits of the suffix. So, get all valid prefix and sum the result together will get all possible splits based on all possible prefixes, so is generates the maximum splits of the string, optimal.

Complexity: $O(n^3)$. For a string with length n, there are n^2 different substrings, that is because for substring with length $1, 2, ..., n$, there are $n, n-1, n-2, ..., 1$ different substrings, so the total is $n + n - 1 + n - 2 + \cdots + 1 = O(n^2)$. There are n substrings with length 1, $n-1$ substrings with length 2, ..., 1 substring with length n. That is because the substrings be get sequentially by start index and length, and if the length is x, the last start index is $n - x$ because the substring cannot exceeds the original string. So, the total length of all substrings is $n + 2 * (n-1) + 3 * (n-2) + \cdots + n = O(n^3)$. Because in the function, we copy each of the substring, the total complexity is $O(n^3)$, because here use the memory to prevent duplicates, so only one calculate for each substring.

b) **Algorithm:**

```
  function has_same_split(d, S, T):
    memo = array(size=N, init=false)

    input u as a substring of S
    input v as a substring of T
    input start as the start index of the substring
    function helper(u, v, start):
      if u.length == 0 and v.length == 0:
        return true
      end if
      if u.length != v.length:
        return false
      if memo[start]:
        return memo[start]
      end if
      for i = 1 to u.length:
        prefixU = u.substring(1, i)
        prefixV = v.substring(1, i)
        if prefixU in d and prefixV in d:
          if helper(u.substring(i + 1, u.length), v.substring(i + 1, v.length), start + i):
            memo[start] = true
            break
          end if
        end if
      end for
      return memo[start]
    end function

    return helper(S, T, 1)
  end function
```

Intuition: One single cut can be made if this cut can separate each of S, T into a prefix which is in the dictionary, and the rest of the string. Do this recursively, and if there is no more substring to separate (length is 0), then it must be a valid split, because all the previous substring segments are in the dictionary.

Proof: Base Case: 0 length S, T, have same split, correct. For some S, T with different length, no same split clearly, correct.

Subproblem: For each split on same index that gives valid prefixes u, v for S, T, take $S - u, T - v$ as the new subproblem. If some of the new subproblems is valid, then the this subproblem is valid.

Proof: For each valid split index for the prefixes, we check that if their suffixes can also be split to valid words at same index. It is clear that if the suffixes can be splitted correctly, the split is still valid if valid prefixes are appended to them. So, the subproblem is correct.

Complexity: $O(n^3)$. It is just a modification of the algorithm in part a), with just a copy on the substrings of T, so it is $O(n^3) + O(n^3) = O(n^3)$.

4. **(20 points) 2D Sheet Cutting**

Algorithm:

```
  structure Recorder:
    integer height, integer width
    string type
    integer k
    Recorder l = null, Recorder r = null

    // get the cut instructions tree
    function get_record(depth=0):
      tabs = '\t' * depth
      content = 'cut (%i, %i) at %s=%i'.format(height, width, type, k)
      result = tabs + content + '\n'
      if type == 'x':
        type_l = 'left'
        type_r = 'right'
      else:
        type_l = 'upper'
        type_r = 'lower'
      end if
      if l != null:
        result = result + tabs + type_l + ':\n' + l.get_record(depth + 1)
      end if
      if r != null:
        result = result + tabs + type_r + ':\n' + r.get_record(depth + 1)
      end if
      return result
    end function
  end structure
```

```
input P as matrix with mxn as the prices of the sheet
function cut_sheet(P):
  memo = matrix(size=(m, n, m, n), init=-INF)
  memo_cut = matrix(size=(H, W, H, W), init=null)

  input p as a submatrix of P of size uxv
  input h, w as the lower right point of p in P
  function helper(p, h, w):
    oh = h - u + 1
    ow = w - v + 1 // get the upper left point of p in P
    cut_by = 'y'
    cut_index = -1
    record_left = null
    record_right = null
    if u == v == 1:
      memo[oh, ow, h, w] = p[0, 0] // for single element
    end if

    if memo[oh, ow, h, w] > -INF:
      return memo[oh, ow, h, w]
    end if

    price = p[u, v] // the price of current split
    for i = 2 to u:
      // submatrix function means take the submatrix with index [a,b,c,d]
      // for a,b as the upper left coordinate of the submatrix
      // and c,d as the lower right coordinate of the submatrix
      upper = helper(p.submatrix(1, 1, i - 1, v), oh + i - 1, w)
      lower = helper(p.submatrix(i, 1, u, v), h, w) // split the matrix to upper and lower

      if upper + lower > price:
        price = upper + lower
        cut_index = i
        cut1 = memo_cut[oh, ow, oh + i - 1, w]
        cut2 = memo_cut[oh + i, ow, h, w]
      end if
    end for

    for i = 2 to v:
      left = helper(p.submatrix(1, 1, u, i - 1), h, ow + i - 1)
      right = helper(p.submatrix(1, i, u, v), h, w)

      if left + right > price:
        price = left + right
        cut_index = i
        cut1 = memo_cut[oh, ow, h, ow + i - 1]
        cut2 = memo_cut[oh, ow + i, h, w]
      end if
    end for

    memo[oh, ow, h, w] = price

    if cut_index != -1: // if has cut
      memo_cut[oh, ow, h, w] = Recorder(height=h - oh + 1, width=w - ow + 1,
                                        type=cut_by, k=cut_index
```

```
                                                   l=cut1, r=cut2)
      end if

      return memo[oh, ow, h, w]
    end function

    return memo[1, 1, m, n], memo_cut[1, 1, m, n].get_record()
  end function
```

Intuition: For each submatrix, it will have the max price if it upper/lower or left/right splits have the max price. In the problem, we cannot know intuitively which way of split can get max value, so take 2 loops to check each. For each subproblem, record the way of cut as a tree in which root is the current split, and the leaves are the ways of cut of its submatrix.

Proof: Base case: 1 element matrix, the maximum is the only price, correct.

Subproblem: For each submatrix given, split it either by x or y in all possibilities, and feed these new sub-submatrix to the sub-subproblem. The result of each subproblem should be the maximum among the sum of two side of each split.

Proof: It is obvious that it generates the maximum for each subproblem, Consider A and $B \subset A, C \subset A, B \cup C = A$, where B, C have the maximum price. Because there is a simple cut across the A to generate B, C, union B, C will not change the price of B, C, and it will also not generate new submatrix S that $S \subset A, S \not\subseteq B, S \not\subseteq C$. Therefore, no matter what the value B, C is, the price $p(A)$ can only be $p(B) + p(C)$. So, if we guarantee that $p(B), p(C)$ are maximum, then $p(A)$ is the maximum, therefore this algorithm is optimal.

Complexity: $O((m+n)m^2n^2) = O(m^3n^2 + m^2n^3)$. There are total $O()$ subproblems, because the number of subproblem is $mn + (m-1)(n-1) + (m-2)(n-2) + \cdots + 1 = O(m^2n^2)$. In each subproblem, we have two parallel loops with complexity $O(m), O(n)$, so the complexity of each subproblem is $O(m+n)$. Therefore, the total complexity is $O((m+n)m^2n^2)$

5. **(20 points) Longest Fluctuating Sequence**

Algorithm:

```
  input A as the array with size N
  function LFS(A):
    if A.length < 4:
      return A.length
    start, end = 1, 3
    ts, te = 1, 3
    for i = end + 1 to N:
      if A[i] < A[i - 1] < A[i - 2] < A[i - 3] or A[i] > A[i - 1] > A[i - 2] > A[i - 3]:
        if te - ts > end - start:
          start, end = ts, te
        end if
        ts = i - 2
        te = i
      else:
        te++
      end if
    end for
    if te - ts > end - start:
      start, end = ts, te
    end if
    return start, end
  end function
```

Intuition: Because we are going to find a continuous sequence with no consecutive values in three, the basic idea is to scan the array, keep the local max length subarray with no consecutive values. Because we know that the minimum length must be larger than 3 to have 4 values, the array with length less than 4 must be fluctuating. Also, it gives that for i where it forms a consecutive with its previous 3 values, the subarray $A[i-2,..,i]$ will not be consecutive.

Proof: For this algorithm, it selects a start point at the beginning of the array, and expand the end point until the array has 4 consecutive values. Because we want the subsequence to be continuous, the fluctuating subsequences are separated by consecutive segments. The algorithm expands the sequence mark from the beginning of such sequence till the end of this sequence, because it scans through the array, and for some element i that is consecutive to 3 of the previous values, the new sequence will start from $i-2$. As said in the intuition, $R = A[i-2,..,i]$ is not consecutive, and if we keep expand R to some index j that $A[j-2,..,j+1]$ is consecutive, $R = A[i-2,..,j]$ will be the longest fluctuating continuous sequence between $i-3, j+1$. Therefore, we keep scanning, and finally keep the longest fluctuating sequence, it will be that sequence.

Complexity: $O(n)$. There is only one $O(n)$ loop, while all other operations are $O(1)$, so the total complexity is still $O(n)$

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