Homework 4

Liwen Ouyang

Mar 25 2020

Problem 1

(a) Because the longest path in a DAG has optimal substructure, i.e. the longest path from u to v consists of the longest path from u to x and the longest path from x to v, where x is an intermediate node. Therefore we can use dynamic programming to construct an optimal solution:

procedure LONGESTPATH($G = \{V, E\}$)
  Initialize an array A to store the solution, where $A[v]$ represent the cost of longest path to v
  set every entry of A to be 0
  Get a topological ordering of all the vertices of G ▷ $O(n + m)$
  for all $v \in V$ following topological ordering **do**
    for all $u \in V$ such that $(v, u) \in E$ **do**
      if $A[u] < A[v] + w(v, u)$ **then** ▷ $w(v, u)$ is the cost of edge from v to u
        $A[u] \leftarrow A[v] + w(v, u)$
        $P[u] \leftarrow v$ ▷ set the predecessor of u to be v
      end if
    end for
  end for
  return $\max(A)$ and the corresponding path by following the predecessor ▷ $O(n)$
end procedure

The algorithm terminates because we are just iterating over all the edges and nodes of G following the topological order, which gives us a running time of $O(n + m)$. The algorithm gives the correct answer because of the fact that we are following a topological ordering of the nodes, which means every time we visit a node, we have already taken into consideration all the possible paths that can reach that node: It follows from the definition of topological ordering that if we visit v before u, there cannot be a path from u to v. Hence, it can be shown by induction that, every time after we visit a node u from v, the maximum weight is in $A[u]$, assuming $A[v]$ is the maximum weight for node v, since essentially we have $A[u] = \max(A[u], A[v] + w(v, u))$. For the base case we know the first node we visit is going to have a maximum weight of 0, because there is no edge into that node, which means it is indeed the maximum weight of that node.

(b) For this problem we can use a modified version of the algorithm in part a, where besides A, we maintain an additional array C to keep track of the number of the longest path into every node. The key observation here is that if we are currently visiting v from u, following a topological order, then

$$A[u] + w(u, v) > A[v] \rightarrow C[v] = C[u]$$

This is true because if we update $A[v]$ when visiting v from u, this means the longest path into v must go through u, and the number of it $C[v]$ must be the same as $C[u]$, the number of longest path into u. And similarly:

$$A[u] + w(u, v) = A[v] \rightarrow C[v] = C[v] + C[u]$$

This means if the longest path we found into v from u, $A[u] + w(u, v)$ is the same as the longest path currently stored $A[v]$, which is longest path into v without passing u, then we find another $C[u]$ longest paths into v, therefore we need to increment $C[v]$ by $C[u]$. With this observation we can devise a correct algorithm:

```
procedure NLongestPath(G = {V, E})
    Initialize an array A to store the solution, where A[v] represent the cost of longest path to v
    set every entry of A to be 0
    Get a topological ordering of all the vertices of G                          ▷ O(n + m)
    Initialize an array C, set C[i] to be 1 if i has no incoming edges, and 0 otherwise
    for all v ∈ V following topological ordering do
        for all u ∈ V such that (v, u) ∈ E do
            if A[u] < A[v] + w(v, u) then          ▷ w(v, u) is the cost of edge from v to u
                A[u] ← A[v] + w(v, u)
                C[u] ← C[v]
            end if
            if A[u] = A[v] + w(v, u) then
                C[v] += C[u]
            end if
        end for
    end for
    n ← 0                                                  ▷ to store the final result
    for all v ∈ v such that A[v] = max(A) do                              ▷ O(n)
        n += C[v]
    end for
    return n
end procedure
```

The algorithm has the sam runtime as the algorithm in (a), which is $O(m+n)$, because we are just doing one extra $O(1)$ step for every iteration. Note that the initialization of C can be done in $O(n+m)$ so it doesn't affect the runtime. Similarly the last loop is just to sum up all counts of the longest path, which is $O(n)$, and it doesn't affect runtime either. We set $C[i]$ to be 1 if there's no edge into i in the initialization because if there's no edge into i, the only path into i would be the trivial path from i to itself, and hence the number of the longest path into i is 1. The algorithm correctly calculates the result as shown at the beginning.

Problem 2

Our intuition comes from the fact that, if we denote the maximum value we can get from $v[i]$ to $v[j]$ as $M(i,j)$ and the minimum as $m(i,j)$, and if we split the expression at kth operator where $i < k \leq j$, then:

$$(op[k] = -) \rightarrow M(i,j) = M(i,k) - m(k+1,j) \wedge m(i,j) = m(i,k) - M(k+1,j)$$

$$(op[k] = +) \rightarrow M(i,j) = M(i,k) + M(k+1,j) \wedge m(i,j) = m(i,k) + m(k+1,j)$$

These are easy to prove, as $a+b$ evaluates to maximum if a, b are both maximum and minimum if both a, b are minimum. Similarly, $a-b$ evaluates to maximum if a is maximum and b is minimum and minimum if a is minimum and b is maximum.

With the recursive relation, we can come up with a solution using dynamic programing, where we create two matrices M and m to store $M(i,j), m(i,j)$ for all pairs of (i,j). It is trivial to see that $M(i,i) = m(i,i) = v[i]$. Therefore we can initialize two matrices' diagonals to be filled by $v[0 \ldots n]$, and constantly update the entries of M and m using the recurrence we have until we have $M(0,n)$, which is what we want:

```
procedure MaxParen(v, op)
    initialize n × n matrices M and m with their diagonals set to be v[i]
    set all other entries in M to be 0 and all other entries in m to be ∞
    initialize another two n × n matrices S_M, S_m to store the tuples indicating where to put parenthesis
    for 2 ≤ l ≤ n do                                        ▷ the length of sub-expression
        for 0 ≤ i ≤ n − l do                          ▷ starting index of the sub-expression
            j ← i + l                                   ▷ ending index of the sub-expression
```

```
            for i ≤ k < j do                    ▷ the index of the operator where we split the expression
                if op[k] = − then
                    if M[i][j] < M[i][k] − m[k + 1][j] then
                        M[i][j] ← M[i][k] − m[k + 1][j]
                        S_M[i][j] ← (i, k), (k + 1, j), S_M[i][k], S_m[k + 1][j]
                    end if
                    if m[i][j] > m[i][k] − M[k + 1][j] then
                        m[i][j] ← m[i][k] − M[k + 1][j]
                        S_m[i][j] ← (i, k), (k + 1, j), S_m[i][k], S_M[k + 1][j]
                    end if
                end if
                if op[k] = + then
                    if M[i][j] < M[i][k] + M[k + 1][j] then
                        M[i][j] ← M[i][k] + M[k + 1][j]
                        S_M[i][j] ← (i, k), (k + 1, j), S_M[i][k], S_M[k + 1][j]
                    end if
                    if m[i][j] > m[i][k] + m[k + 1][j] then
                        m[i][j] ← m[i][k] + m[k + 1][j]
                        S_m[i][j] ← (i, k), (k + 1, j), S_m[i][k], S_m[k + 1][j]
                    end if
                end if
            end for
        end for
    end for
    return S_M[0][n − 1]
end procedure
```

The algorithm has a triple loop, which gives a runtime of $O(n^3)$. Now in order to prove it is correct let's assume it correctly stores the parenthesis in S_M, S_m for all the subproblems. In our algorithm, we will find a k to split the expression into E_1 and E_2 such that $E_1\, op[k]\, E_2$ is maximum and put parenthesis $(0, k), (k + 1, n)$. If $op[k] = -$, we will maximize E_1 by putting parenthesis at the places stored in $S_M[0][k]$, which by our assumption is correct. And we will minimize E_2 by putting parenthesis at the places stored in $S_m[k+1][n]$, which by assumption also correct. We know $E = E_1 - E_2$ is maximized if E_1 is maximized and E_2 is minimized, therefore we proved this case. The argument for $op[k] = +$ is similar. For the base case, if $E = a\, op\, b$ where a, b are numbers, our algorithm will put parenthesis as:$(a)\, op\, (b)$, which is correct for both cases where op is plus or minus.

Problem 3

(a) For this problem it is useful for us to define some functions to begin with. Let $D(s)$ be the function that returns true if s is in the dictionary and false otherwise in $O(1)$. Let $f(a, b)$ be a function that returns 1 if $D(S[a \dots b])$ is true and 0 otherwise, i.e. $f(a, b)$ returns 1 if the substring from $S[a]$ to $S[b]$ is a word, and 0 otherwise. The implementation of f will result in an iteration over S therefore has a runtime of $O(n)$.

Now we can make the observation that if we denote $N(a, b)$ as the number of ways to split $S[a \dots b]$ into words, then we have the number of ways to split described by the recurrence:

$$N(a, b) = (\sum_{i=a}^{b-1} f(a, i)N(i + 1, b)) + f(a, b)$$

The reasoning of the above recurrence is that if we take the first split at i where $a \le i \le b$, we are left with two substrings $S_1 = S[a \dots i]$ and $S_2 = S[i + 1 \dots b]$. If S_1 is a word, then in this case (take the first cut at i), $N(S) = N(S_2) + 1$ if $S[a, b]$ is a word, which itself counts as 1 way to split, and

$N(S) = N(S_2)$ if $S[a, b]$ is not a word. If S_1 is not a word, that means this cut is invalid. Therefore $N(S) = f(a, b)$ in this case. If we take a sum over all possible first cut from a to b, we have the total number of ways to split $S[a \dots b]$. With this recurrence, we can design a correct algorithm computing the recurrence bottom up using an array C for memoization, where $C[i]$ stores the number of ways to split $S[i \dots n]$:

procedure NSPLIT($S[1 \dots n]$)
  Initialize an array C of length n with all entries set to 0
  for j from n to 1 **do**
    for i from j to n **do** ▷ Nested loop: $O(n^2)$
      if $i = n$ **then**
        $C[j] += f(j, n)$ ▷ $O(n)$
      else
        $C[j] += f(j, i)C[i + 1]$
      end if
    end for
  end for
  return $C[1]$
end procedure

As shown by comments, we are doing a $O(n)$ operation in a nested $O(n^2)$ loop. Hence, our algorithm has a runtime of $O(n^3)$. It is correct as shown by the proof of the recurrence in the beginning, and we are just updating every entry based on the derivation from the recurrence:

$$C[j] = N(j, n) = (\sum_{i=j}^{n-1} f(j, i)N(i + 1, n)) + f(j, n) = (\sum_{i=j}^{n-1} f(j, i)C[i + 1]) + f(j, n)$$

(b) We will be using an algorithm that is similar to part a. For our convenience, let's refine $f(a, b)$ to be a function that returns boolean value true if both $S[a \dots b]$ and $T[a \dots b]$ are words and false otherwise. And another function $Same(a, b)$ that returns true if both $S[a \dots b]$ and $T[a \dots b]$ can be cut into words at same places. With this we can have a similar boolean algebraic expression similar to what we have in a:

$$Same(a, b) = (\bigvee_{i=a}^{b-1} f(a, i) \wedge Same(i + 1, b)) \vee f(a, b)$$

This is true because

$$Same(a, b) \iff \exists i : f(a, i) \wedge Same(i + 1, b) \vee f(a, b)$$

That is, we can split $T[a \dots b], S[a \dots b]$ at same place into words if and only if there's a number i such that both $T[a \dots i]$ and $S[a \dots i]$ are words, and at the same time we can split $S[i + 1 \dots b]$ and $T[i + 1 \dots b]$ into words at the same place. Another case is that if both T and S are themselves words, then it is trivially true that we can split them at the same places to words.

Now that we have the recurrence and proved its correctness, we can devise an iterative algorithm:

procedure SAME($S[1 \dots n], T[1 \dots n]$)
  Initialize an array C of length n with all entries set to F
  for j from n to 1 **do**
    for i from j to n **do** ▷ Nested loop: $O(n^2)$
      if $i = n$ **then**
        $C[j] = C[j] \vee f(j, n)$ ▷ $O(n)$
      else
        $C[j] = C[j] \vee (f(j, i) \wedge C[i + 1])$
      end if
    end for
  end for

return $C[1]$
end procedure

This algorithm is very similar to the one we have in a, and it has the same runtime of $O(n^3)$. The correctness of this algorithm was discussed above.

Problem 4

Let's denote the maximum price we can get from the sheet $M(m, n)$ and consider the first cut on the $m \times n$ sheet. We can either not cut the sheet and be left with the whole sheet with total price $P(m, n)$, or we can split the sheet into two sheets. If we decide to cut the sheet into two smaller sheets, obviously we need to maximize the price of the two sub-sheets. Hence, we get a recurrence in this case, where we need to take all the possible ways to split the sheet into two smaller sheets into consideration and get the maximum out of them.

Let's first consider horizontal cuts. If we cut at $y = i$, we will be left with $i \times n$ and $(m - i) \times n$ sheets. Therefore, we want to have $M(i, n) + M(m - i, n)$ as a potential maximum cut. Similarly if we cut vertically at $x = j$, we want $M(m, j) + M(m, n - j)$ as a potential maximum price. Taking all possible values for i, j into consideration we can come up with a recurrence by finings the maximum case among all cases considered:

$$M(m, n) = \max_{1 \le i \le m/2, 1 \le j \le n/2} \{M(i, n) + M(m - i, n), M(m, j) + M(m, n - j), P(m, n)\}$$

Note that we are only considering $1 \le i \le m/2, 1 \le j \le n/2$, because of symmetry. Since we are just computing the maximum recursively by enumerating all the possible cases, the recurrence is true by induction, assuming M returns the correct result for the subproblems with a correct base case:$M(1, 1) = P(1, 1)$. With this in mind, we can devise an algorithm:

procedure MaxCut(m, n, P)
  Initialize a $m \times n$ 2-d array M, where $M[i][j]$ is the maximum price obtainable for $i \times j$ sheet
  for all entries $M[i][j]$ of M **do** ▷ we default the maximum price to be obtained by not cutting.
    $M[i][j] \leftarrow P(i, j)$
  end for
  Initialize another $m \times n$ 2-d array C to store the instructions.
  for i from 1 to m **do**
    for j from 1 to n **do**
      for a from 1 to $\lfloor \frac{i}{2} \rfloor$ **do** ▷ All horizontal splits
        if $M[i][j] < M[a][j] + M[i - a][j]$ **then**
          $M[i][j] \leftarrow M[a][j] + M[i - a][j]$
          $C[i][j] \leftarrow$ "cut at $y = a$ to get $a \times j$ and $(i - a) \times j$" $\cup C[a][j] \cup C[i - a][j]$
        end if
      end for
      for b from 1 to $\lfloor \frac{j}{2} \rfloor$ **do** ▷ All vertical splits
        if $M[i][j] < M[i][b] + M[i][j - b]$ **then**
          $M[i][j] \leftarrow M[i][b] + M[i][j - b]$
          $C[i][j] \leftarrow$ "cut at $x = b$ to get $i \times b$ and $i \times (j - b)$" $\cup C[i][b] \cup C[i][j - b]$
        end if
      end for ▷ Get the maximum among all cases
    end for
  end for
  return $C[m][n]$
end procedure

The runtime of the algorithm is $O(nm(m + n))$, as we have triple nested for-loops, and inside which we are performing $O(1)$ work. The algorithm returns the desired result because we are just computing the recurrence defined above bottom up, by iterating over all possible cuts (including the case where we don't make a cut, which is set to the default value for all entries of M) and updating the maximum whenever we

find a larger value, which is going to result in the maximum value among all cases at the end of one whole iteration.

Problem 5

To begin with it is useful for us to define a helper function that, given an index and a sequence, returns the end of the longest fluctuating sequence starting at the index.

procedure LONGESTAT(S, i) ▷ Given sequence $S[1 \dots n]$, index $i \leq n$
  if $n - i \leq 2$ **then**
    return n ▷ any sequence of length less than 4 is not fluctuating.
  end if
  $x \leftarrow i$
  while $x \leq n - 3$ **do**
    if $S[x] \leq S[x+1] \leq S[x+2] \leq S[x+3] \vee S[x] \geq S[x+1] \geq S[x+2] \geq S[x+3]$ **then**
      return $x + 2$ ▷ found non-fluctuating sub-sequence
    else
      $x+ = 1$
    end if
  end while
  return n ▷ no non-fluctuating sequence found.
end procedure

This algorithm correctly returns the desired output because we are just iterating over the input trying to find non-fluctuating part. If there is one, we return the sequence from the beginning i up to the second last element $S[x+2]$ in the non-fluctuating subsequence we found first as the longest fluctuating sequence starting at i. There cannot be longer one because we cannot possibly include $S[x+3]$, otherwise it will result in a non-fluctuating sequence. If no non-fluctuating sequence is found, we simply return S. This algorithm has a runtime of $O(n)$

Now we can define a recursive algorithm to find the longest sequence in S:

procedure L(S, m, n) ▷ Given $S[m \dots n]$
  if $n - m \leq 2$ **then** ▷ Base case
    return (m, n)
  else
    $L_1 \leftarrow LongestAt(S, m) - m + 1$ ▷ L_1: length of the longest fluctuating sequence starting at m
    $s, t \leftarrow L(S, m+1, n)$
    $L_2 \leftarrow s - t + 1$
    if $L1 > L_2$ **then** ▷ L_2: length of the longest fluctuating sequence in $S[m+1 \dots n]$
      return $(m, LongestAt(S, m))$
    else
      return $L(S, m+1, n)$
    end if
  end if
  call $L(S, 1, n)$ to get the starting and ending index of the longest fluctuating sequence in S
end procedure

The algorithm's runtime is described by the recurrence: $T(n) = T(n-1) + O(n)$, because we reduce the size of S by one at every step of the recursion, while calling a $O(n)$ function defined above to find the longest fluctuating sequence starting from certain index. It can be shown that the recurrence solves to $T(n) = \Theta(n^2)$ by induction. Now we need to prove it is correct. For the base case it is trivial to see that if $|m - n| \leq 2$, S itself is fluctuating, as no sequence of length less than 4 can be non-fluctuating as defined. Now let's assume the algorithm correctly return the result for $S[m+1 \dots n]$, i.e. the longest fluctuating sequence in $S[m+1 \dots n]$. That is to say, for $S[m \dots n]$, we only need to calculate the longest fluctuating sequence starting at m and compare it with the longest obtainable from $S[m+1 \dots n]$ to get the maximum among all fluctuating sequence. By assumption $L(S, m+1, n)$ returns it, and therefore by comparison between L_1, L_2,

the algorithm will return the correct result.