Abraham Cardenas

CMPS 102 — Winter 2019 – Homework 5

1. (10 pts)

a. Given array of words P of length n and Q of length m, we need to find the length of the longest common subsequence. To find the longest subsequence, we'll need to find the max subsequence of each smaller subsequence. So, the recurrence would be given by the following:

$$T(n,m) = \begin{cases} 0, & \text{if } n = 0 \text{ or } m = 0 \\ 1 + T(n-1, m-1), & \text{if } P_{n-1} = Q_{m-1} \\ \max\{\, T(n, m-1), T(n-1, m) \,\}, & \text{otherwise} \end{cases}$$

The recurrence above describes a recursive solution that computes each possible subsequence. It does this by first checking to see if P and Q have a length > 0. If not, it returns 0 since no subsequence exists with either length being 0. If both lengths are non-zero, then it checks if the current ending words of P and Q are the same. If they are the same, then we add 1 to a recursive call where we reduce the size of P and Q by 1. If they're not the same, then we return the maximum length between 2 recursive calls. The first recursive call reduces the size of Q by 1 and leaves P the same. Conversely, the second call reduces the size of P by 1 and leaves Q the same.

b. To implement a bottom-up algorithm, we must loop through both P and Q. We'll define a table of numbers which will use $O(n \cdot m)$ extra space.

```
1:  function LONGEST-SUBSEQUENCE-LENGTH(P, Q, n, m)
2:      table[n][m]
3:      for i ← 0 to i ≤ n do
4:          for j ← 0 to j ≤ m do
5:              if i = 0 or j = 0 then
6:                  table[i][j] ← 0
7:              else if P[i − 1] = Q[j − 1] then
8:                  table[i][j] ← table[i − 1][j − 1] + 1
9:              else
10:                 if table[i − 1][j] > table[i][j − 1] then
11:                     table[i][j] ← table[i − 1][j]
12:                 else
13:                     table[i][j] ← table[i][j − 1]
14:                 end if
15:             end if
16:         end for
17:     end for
18:     return table[n][m]
19: end function
```

c. Since the inner loop and outer loop loops through all of P and Q respectively, the algorithm would have an upper bound of $\boxed{O(n \cdot m)}$.

d. To "traceback" our result so we can get the actual words instead of the length of the longest common subsequence, we need to traverse $table$ from part b.

```
1:  function TRACE-BACK(table, P, Q, n, m)
2:      trace[n + m]                          ▷ Declare trace of size n + m.
3:      i ← n − 1
4:      j ← m − 1
5:      k ← (n + m) − 1
6:      while i > 0 and j > 0 do
7:          if P[i] = Q[i] then
8:              trace[k] ← P[i]
9:              i ← i − 1
10:             j ← j − 1
11:         else if table[i − 1][j] > table[i][j − 1] then
12:             i ← i − 1
13:         else
14:             j ← j − 1
15:         end if
16:         k ← k − 1
17:     end while
18:     r ← 0
19:     for s ← k to (n + m) − 1 do        ▷ Reverse the traced sequence of word of words.
20:         if r = k then
21:             break
22:         end if
23:         swap(trace[s], trace[((n + m) − 1) − r])
24:         r ← r + 1
25:     end for
26:     return trace[k ... ((n + m) − 1)]
27: end function
```

Since we traversed through the all values in P and/or Q and swapped all of $trace$, our running time will be $O(2 \cdot (n+m)) = \boxed{O(n+m)}$. [Could use a stack to avoid swapping at the end]

a. Given an array d of size n, we need to find the minimum number of operations that is needed to multiply the sets of matrices in $d[0 \ldots n-1]$. We can derive a simple recursive solution to this problem by using the following recurrence:

$$\boxed{T(i,j)_{1 \le (i,j) \le n} = \begin{cases} 0, & \text{if } i = j \\ \min_{i \le k < j}\big(T(i,k) + T(k+1,j) + d_{i-1} * d_k * d_j\big), & \text{otherwise} \end{cases}}$$

b. The subproblems in the recursive solution in part a computes all possible permutations of parenthesis. It does this by checking which permutation contains the minimum amount of $d_{i-1} * d_k * d_j$ operations. This recursive solution is not optimal since we have some repeating subproblems in our recursion calls.

c. To implement a bottom-up algorithm, we'll need to use an $n \cdot n$ table that keeps track of previous calculated amount of operations. The total amount of extra space that we'll use is $O(n^2)$.

```
1:  function MIN-MATRIX-MULT(d, n)
2:      table[n][n]
3:      for i ← 1 to n − 1 do
4:          table[i][i] ← 0
5:      end for
6:      for ℓ ← 2 to n − 1 do
7:          for i ← 1 to n − ℓ do
8:              j ← i + ℓ − 1
9:              table[i][j] ← ∞
10:             for k ← i to j − 1 do
11:                 costMult ← table[i][k] + table[k + 1][j] + d[i − 1] ∗ d[k] ∗ d[j]
12:                 if costMult < table[i][j] then
13:                     table[i][j] ← costMult
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return table[1][n − 1]
19: end function
```

Since we have 3 nested loops where each loop iterates at most n times, our upper bound would be $\boxed{O(n^3)}$.

d. To find the "last" multiplication, all we need to do is check the left and the bottom cells of $table[1][n-1]$ (where we got the minimum on line 18 in part c). These cells are $table[1][n-2]$ and $table[2][n-1]$ respectively. We then check which cells hold the smaller value, and we use that cell to determine which multiplication happened last. This can either be $d[1 \ldots n-2] * d[n-1]$ or $d[1] * d[2 \ldots n-1]$.