ECE608 CHAPTER 15 PROBLEMS

1) 15.2-2
2) 15.2-5
3) 15.2-6
4) 15.3-1
5) 15.3-2
6) 15.3-3
7) 15.4-2
8) 15.4-3
9) 15.4-5
10) 15.5-3
11) 15-5

ECE608 - Chapter 15 answers

(1) CLR 15.2-2

```
MATRIX CHAIN MULTIPLY($A, s, i, j$)
1. if $i = j$
2. then return $A_i$
3. else X = MATRIX CHAIN MULTIPLY($A, s, i, s[i, j]$)
4.      Y = MATRIX CHAIN MULTIPLY($A, s, s[i, j] + 1, j$)
6.      return MATRIX-MULTIPLY($X, Y$)
```

(2) CLR 15.2-5

Each time the l-loop executes, the i-loop executes $n-l+1$ times. Each time the i-loop executes, the k-loop executes $j-i = l-1$ times, each time referencing m twice to create an entry. Thus the total number of times m is referenced is $\sum_{l=2}^{n}(n-l+1)(l-1)2$.

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) &= \sum_{l=2}^{n}(n-l+1)(l-1)2 \qquad \text{let } k = l-1 \\
&= 2\sum_{k=1}^{n-1}(n-k)k \\
&= 2\sum_{k=1}^{n-1} n\,k - 2\sum_{k=1}^{n-1} k^2 \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6} \\
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\
&= \frac{n^3 - n}{3}
\end{aligned}$$

(3) CLR 15.2-6

Given that parentheses group two matrices at a time, there must be exactly $n-1$ pairs of parentheses in a full parenthesization of an n-element expression. To see this, consider arbitrarily parenthesizing two matrices in a chain. This effectively replaces the two grouped matrices with one multiplied matrix. This process of grouping two elements can occur $n-1$ times before we are left with a single item.

(4) CLR 15.3-1

Running Recursive-Matrix-Chain is asymptotically more efficient than enumerating all the ways of parenthesizing the product and computing the number of multiplications for each.

Consider the treatment of subproblems by each approach:

(i) For each possible place to split the matrix chain, the enumeration approach finds all ways to parenthesize the left half, finds all ways to parenthesize the right half, and looks at all possible combinations of the left half with the right half. The amount of work to look at each combination of left and right half subproblem results is thus the product of the number of ways to parenthesize the left half and the number of ways to parenthesize the right half.

(ii) For each possible place to split the matrix chain, Recursive-Matrix-Chain finds the best way to parenthesize the left half, finds the best way to parenthesize the right half, and combines just those two results. Thus the amount of work to combine the left and right half subproblem results is $O(1)$.

Section 15.2 showed that the running time for enumeration is $\Omega(\frac{4^n}{n^{3/2}})$. We will show that the running time for Recursive-Matrix-Chain is $O(n\,3^{n-1})$.

To obtain an upper bound on the running time of Recursive-Matrix-Chain, we shall use a similar approach as was used in Section 15.3 to obtain a lower bound. First, we derive an upper bound recurrence and then use substitution to prove the O bound. For the lower-bound recurrence, the book assumed that the execution of lines 1-2 and 6-7 take at least unit time each. For the upper-bound recurrence, we'll assume those pairs of lines each take at most constant time c. This yields the recurrence $T(1) \le c$ and $T(n) \le c + \sum_{k=1}^{n-1} (T(k) + T(n-k) + c)$, for $n > 1$.

Due to the fact that for $k = 1, 2, \ldots, n-1$, each $T(i)$ appears twice, the recurrence can be rewritten as $T(n) \le 2\sum_{i=1}^{n-1} T(i) + cn$.

We shall prove that $T(n) = \mathrm{O}(n\,3^{n-1})$ by using the substitution method. (Note: any upper bound on $T(n)$ that is in $\mathrm{o}(\frac{4^n}{n^{3/2}})$ will suffice. You might prefer to prove one something simpler, such as $T(n) = O(3.5^n)$). Specifically, we shall show that $T(n) \le cn\,3^{n-1}$ for all $n \ge 1$. The basis is easy, since $T(1) = c \cdot 1 \cdot 3^0 \le c$. Inductively, for $n \ge 2$ we have:

$$\begin{aligned} T(n) &\le 2\sum_{i=1}^{n-1} T(i) + cn \\ &\le 2\sum_{i=0}^{n-1} T(i) + cn \\ &\le 2\sum_{i=0}^{n-1} ci3^{i-1} + cn \end{aligned}$$

$$\begin{aligned}
&= c(2\sum_{i=0}^{n-1} i3^{i-1} + n) \\
&= c(2(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{(3-1)^2}) + n) \\
&= cn3^{n-1} + c(\frac{1-3^n}{2} + n) \\
&= cn3^{n-1} + \frac{c}{2}(2n+1-3^n) \\
&\leq cn3^{n-1} \quad \text{for all } c > 0, n \geq 1
\end{aligned}$$

Note: the above substitution uses the fact that $\sum_{i=0}^{n-1} ix^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2}$. This can be derived from Equation (A.5) by taking the derivative:

$$\begin{aligned}
f(x) &= \sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x-1} \\
\sum_{i=0}^{n-1} ix^{i-1} &= f'(x) \\
&= \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2}
\end{aligned}$$

Running Recursive-Matrix-Chain takes $O(n3^{n-1})$ time, and enumerating all parenthesizations takes $\Omega(\frac{4^n}{n^{3/2}})$ time, so Recursive-Matrix-Chain is more efficient than enumeration.

(5) CLR 15.3-2

In a divide-and-conquer algorithm such as MERGE-SORT there are no overlapping subproblems. Algorithms suited to the divide-and-conquer problem solving approach have this character. Because the solution to a particular subproblem is needed only once memoization will not speed up this kind of algorithm.

(6) CLR 15.3-3

Yes, this problem does exhibit optimal substructure property. Say the highest level parenthesization splits the matrix chain into two sub chains. The parenthesization within each sub-chain must be such that they maximize the number of scalar multiplications involved for each sub chain. Let us suppose that such is not the case i.e., say that the first sub-chain could be parenthesized in another way that increases the multiplications for that sub-chain. Then we could obtain a higher number of total multiplications for the entire chain by parenthesizing the first chain in this other way. This is so because the parenthesization within one sub-chain does not affect the other chain, neither does it affect the cost of the eventual multiplication of the two subchains (the final multplication cost itself depends on the matrix dimensions which are constant for a particular split point regardless of how each sub-chain is internally parenthesized). So it must be the case that when choosing from the various split points possible, the best split point can be obtained by considering, for every split point, the cost of multiplying two sub-chains and the cost of each sub-chain optimally parenthesized internally.

(7) CLR 15.4-2

Here is the pseudocode for an algorithm that uses the c table to guide the process of marking an LCS in the X and Y arrays.

```
MARK-LCS($c, X, Y$)
1.  //Mark the positions in $X$ and in $Y$ that constitute an LCS. Returns nothing.
2.  $i \leftarrow m \leftarrow length[X]$
3.  $j \leftarrow n \leftarrow length[Y]$
4.  while $c[i,j] > 0$
5.        do if $X[i] = Y[j]$
6.              then Mark $X[i]$ and $Y[j]$.
7.                    $i \leftarrow i - 1$   // Move diagonally.
8.                    $j \leftarrow j - 1$
9.              else if $c[i-1,j] = c[i,j]$
10.                   then $i \leftarrow i - 1$
11.                   else $j \leftarrow j - 1$   // Must have $c[i,j-1] = c[i,j]$.
```

The indices i and j always point to a table entry $c[i,j]$ and initially point to $c[m,n]$. The algorithm moves to an entry in c with a value one smaller only when it marks an entry of X and Y as being in the LCS and otherwise moves to a c entry with the same value. Since $c[m,n]$ is the size of a largest LCS, the algorithm marks an optimal LCS.

(8) CLR 15.4-3

```
MEMOIZED-LCS-LENGTH($X, Y$)
1. $m \leftarrow length[X]$
2. $n \leftarrow length[Y]$
3. for $i \leftarrow 1$ to $m$
4.       do $c[i, 0] \leftarrow 0$
5. for $j \leftarrow 0$ to $n$
6.       do $c[0, j] \leftarrow 0$
7. for $i \leftarrow 1$ to $m$
8.       do for $j \leftarrow 1$ to $n$
9.             do $c[i, j] = -1$
10. return LOOKUP-LCS($X, Y, m, n$)
```

```
LOOKUP-LCS($X, Y, i, j$)
1. if $c[i, j] \geq 0$
2.    then return $c[i, j]$
3. if $x_i = y_j$
4.    then $c[i, j] \leftarrow$ LOOKUP-LCS($i - 1, j - 1$) + 1
5.          $b[i, j] \leftarrow$ "$\nwarrow$"
6.    else $q_1 \leftarrow$ LOOKUP-LCS($i - 1, j$)
7.          $q_2 \leftarrow$ LOOKUP-LCS($i, j - 1$)
8.          if $q_1 \geq q_2$
7.             then $c[i, j] \leftarrow q_1$
8.                   $b[i, j] \leftarrow$ "$\uparrow$"
9.          else $c[i, j] \leftarrow q_2$
10.                $b[i, j] \leftarrow$ "$\leftarrow$"
11. return $c[i, j]$
```

(9) CLR 15.4-5

LCS can be viewed as a filter that extracts from a given sequence the largest instance of something that it has in common with another sequence. If a sequence is processed by LCS-LENGTH with a sequence of the same or greater length that consists of only a single, repeated character, then the result is a count of the number of occurrences of that character in the given sequence.

To extract the longest monotonically increasing subsequence the given sequence must be "filtered" by a sequence for which all subsequences are monotonically increasing and which contains exactly the same mix of sequence components. This comparision sequence is the original sequence in increasing sorted order.

By calling the LCS-LENGTH procedure to find the longest common sequence of the original sequence and its sorted version, we can get the longest monotonically increasing subsequence of the original sequence. Sorting will take $\Theta(nlgn)$. LCS-LENGTH will take $\Theta(n^2)$.

The algorithm is as follows:

Figure 1: (a) $e[i,j]$ table, (b) $w[i,j]$ table, (c) $root[i,j]$ table, and (d) the optimal binary search tree for 15.5-2

```
LMIS($A, n$)
1:  for $i \leftarrow 1$ to $n$
2:      $B[i] \leftarrow A[i]$
3:  MERGE-SORT ($B, 1, n$)
4:  LCS-LENGTH($A, B$)
5:  PRINT-LCS($b, A, n, n$)
```

(10) CLR 15.5-3

If we use Eq. 15.17 to compute $w[i,j]$ everytime we need it in line 10 of OPTIMAL-BST then we would need to run a loop for $j-i$ iterations. This loop could be inserted between line 8 and 9 of the original code. Effectively, the code now looks like three nested loops. The inner most loop actually consists of two loops executed sequentially , each running for $r \leftarrow i$ to j. And the work done within both these inner loops is a constant for each iteration. The complexity of the algorithm could written as:

$$T(n) = \sum_{l=1}^{n} \sum_{i=1}^{n-l+1} \sum_{r=i}^{j} constant_1 + constant_2$$

The analysis for the above summation is the same as for the OPTIMAL-BST algorithm given in the text. Intiutively we can see tyhat the for loops are nested three deep and each loop index takes on the order of n values. The running time is $\Theta(n^3)$.

(11) CLR 15-5

(a)

```
VITERBI-1($G, \sigma_1 ... \sigma_k$)
1. for $\leftarrow k$ downto 1
2.     Push($S, \sigma_i$)
3. for all $v \in V$
4. $\pi(v) \leftarrow$ NIL
5. color($v$) = WHITE
```

```
6. ENQUEUE($Q$,sentinel)
7. ENQUEUE($Q, v_0$)
8. while !empty(S)
9.       do $u \leftarrow$ DEQUEUE($Q$)
10.         if $u =$ sentinel
11.            then $\sigma \leftarrow$ Pop($S$)
12.                 $u \leftarrow$ DEQUEUE($Q$)
13.                 ENQUEUE($Q$,sentinel)
14.         flag $\leftarrow 0$
15.         for each $v \in$ Adj[$u$]
16.         do if $(u, v) = \sigma$ AND color($v$) = WHITE
17.               then if $\sigma = \sigma_k$
18.                       then PRINT-PATH($G, v_0, v$)
19.                            exit
20.                    color($v$) = BLACK
21.                    ENQUEUE ($v, Q$)
22.                    $\pi(v) \leftarrow u$
23.                    flag $\leftarrow 1$
24.      if flag = 0
25.         then print NO-SUCH-PATH
```

The queue operations used above are as those used in the BFS algorithm on pg 532 of CLR. The stack operations are as the ones described on pg 406 of CLR. PRINT-PATH is given on pg 538 of CLR. The algorithm presented performs a sort of breadth first search of the graph, and discards the nodes that do not satisfy the sequence s provided. The sentinal "vertex" is used within the queue for marking when to start comparing edges to the next σ in the sequence. The running time is the same as that of BFS which is $O(V + E)$.

(b)

```
VITERBI-2($G, \sigma_1 ... \sigma_k$)
1. for $\leftarrow k$ downto 1
2.       Push($S, \sigma_i$)
3. for all $v \in V$
4. $\pi(v) \leftarrow$ NIL
5. color($v$) = WHITE
6. ENQUEUE($Q$,sentinel)
7. ENQUEUE($Q, v_0$)
8. $maxprob \leftarrow 0$
9. while !empty(S)
10.      do $u \leftarrow$ DEQUEUE($Q$)
11.         if $u =$ sentinel
12.            then $\sigma \leftarrow$ Pop($S$)
```

```
13.                 $u \leftarrow$ DEQUEUE($Q$)
14.                 ENQUEUE($Q$,sentinel)
15.         flag $\leftarrow 0$
16.         **for** each $v \in$ Adj[$u$]
17.         **do if** $(u, v) = \sigma$ AND color($v$) = WHITE
18.               **then if** $\sigma = \sigma_k$
19.                       **then** $prob$ = PATH-PROB($G, v_0, v$)
20.                             **if** $prob > maxprob$
21.                               **then** $v_{max} \leftarrow v$
22.                                     $maxprob = prob$
22.                   color($v$) = BLACK
23.                   ENQUEUE ($v, Q$)
24.                   $\pi(v) \leftarrow u$
24.                   flag $\leftarrow 1$
25.     **if** flag = 0
26.       **then** print NO-SUCH-PATH
27. PRINT-PATH($G, v_0, v_{max}$)
```

```
PATH-PROB($G, s, v$)
1. $pathprob \leftarrow 1$
2. **while** $v! = s$
3.     **do** $pathprob \leftarrow pathprob*$ Edge-Prob($\pi[v], v$)
4.         $v \leftarrow \pi[v]$
```

This algorithm is similar to the one in part (a) except that when matching edges against the last character σ_k we do not terminate on the first match. Instead we compute the path probability for the path that just matched and compare it against a global maximum. We update the global maximum with the new path if it has a higher probability. At the end of the algorithm we print out the path corresponding to the maximum probability.

Due to the breadth first tree being constructed we know that the tree contains all paths that match so far (i.e., upto k-1 characters) and the last vertex on each such path is currently enqueued because when that vertex matched for the k-1'th characeter it was enqueued in line 23. Hence we are guaranteed that all matching paths are found when the k'th character is being matched. Upon each matching path that is found we compute its path probability and remember the path with maximum probability seen so far. So we are guaranteed to find the matching path with maximum probability.

We call PATH-PROB for a maximum of V nodes, and the running time for PATH-PROB can be at most $O(V)$. SO in addition to the running time of BFS we incurr an $O(V^2)$ cost. The running time for the algorithm is then $O(E + V^2)$.