CS 610: Assignment 3

Submitted By: Chanda Grover
Roll No: 17310004

November 12, 2017

1. a) The "heaviest-first algorithm "does not find an independent set of maximum weight on this graph. Consider the sequence of weights {5,7,5}. The given algorithm will choose the node with highest weight {7} in the independent set and eliminate both of its neighbors. But the optimal solution for this sequence is found to be having node {5,5} in the independent set. Therefore algorithm is not optimal.

   b) The "either odds or evens "algorithm does not find an independent set of maximum weight on this graph. Consider the the grpah having sequence {7,4,3,8 }. The given algorithm will output the independent set {4,8}. But the optimal solution for this problem is found to be having {7,8}. Therefore the algorithm is not optimal.

   c) We will use Dynamic Programming approach to solve this problem.

   Claim: S(k) $=max(S(k-1), S(k-2)+w_k)$. Request k belongs to an optimal solution on the set {1,2,....., n} only if

$$w_k + S(k-2) \geq S(k-1)$$

   Correctness of the Claim: If S is a maximal independent set for n nodes and it included the last node, then S$-e_n$ is a maximal independent set of the first n-2 nodes. Otherwise (if e_n was not in S), then S is a maximal independent set of the first n-1 nodes. These facts form the crucial component on which dynamic programming solution is based. A recurrence equation that expresses the optimal solution in terms of optimal solution to its subproblems.

   Algorithm:M-MAxIndSet(j)

```
 I  if j=0 then
 II         Return 0
 III Else If M[j] is not empty
 IV         Return M[j]
 V  Else
```

VI Define $M[j] = max(M-MAxIndSet(k-1), M-MAxIndSet(k-2) + w_k)$

Analysis of Time Complexity: The time complexity for this algorithm is $O(n)$. Since we are computing the value for every node. So, it runs for n number of iterations.

2. a) Below is the counter Example representing the given algorithm is not optimal for the problem.

Case	Week1	Week2	Week3
l	3	3	3
h	1	7	12

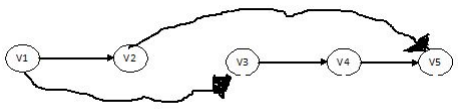
The given algorithm will choose {7} as the high stress job in week2 and nothing in week1 and week3 but the actual optimal solution will take 3(low stress job) in week1, nothing in week2 and 12(high stress job in week3). So in total giving optimal solution 15.

b) **Claim:**Let Opt(i) denote the maximum value revenue achievable in the input instance restricted to weeks 1 through i.
$Opt(i) = max(l_i + opt(i-1), h_i + opt(i-2))$
Correctness of Claim: If it selects a low stress job, it can behave optimally uptoweek i-1, followed by this job, while if it slects a high stress job, it can behave optimally upto week i-2, followed by this job. Thus , we can compute all opt values by invoking this recurrence for i=1,2,...,n.
Algorithm:M-HighLow(j):

I if j=0 then
II Return 0
III Else If M[j] is not empty
IV Return M[j]
V Else
VI Define $M[j] = max(l_i + M - HighLow(i-1), h_i + M - HighLow(i-2))$

Analysis of Time Complexity: We Compute optimum values for all the weeks from week 1 to n which takes constant time. Therefore, Time taken by this algorithm is O(n).

3. a) The given algorithm is not optimal. Below is the graph representing



Given Algorithm will output the longest path as {v1, v2, v5} but the actual longest path is {v1,v3,v4,v5}

b) **Claim:** The longest path through the first node in ordered path is one edge longer than the longest path through any node to which the first node is directly connected.

$$L(k) = max_{c \in children(c)}(1 + L(c))$$

Correctness of Claim: First, it is clear that the second vertex on any path from v of non-zero length must be a neighbor vertex u such that (v; u) 2 E. Let u0 be a vertex which maximizes the above expression (i.e. u0 is the destination f some edge from v such that lenpath[u] is maximum among all such u's). Then the length of a longest path from v is exactly lenpath[u0] + 1 for the edge (v; u0). We will show this by a simple contradiction: assume there is a path from v of length greater than lenpath[u0] + 1. Then the next vertex on this path must be some other neighbor u0 of v, and the length of the path would be the length of a longest path from u0 plus 1 for the edge (v; u0). But this is exactly lenpath[u0] + 1. Therefore it must be that lenpath[u0] ¿ lenpath[u0], which contradicts u0 being the vertex that maximizes lenpaths. Now, since computing lenpath[v] involves the values for vertices that are later in a topological ordering of the DAG, we will

fill in the array in a reverse topological order. The base cases will be vertices with no out-going edges, for whom the longest path is necessarily of length 0. Below is the full algorithm:

Algorithm: Long-path(n)

I Array M[1,..., n]
II M[1]=0
III for i=1,2, ...n
IV M=-1
V for all edges (j,i) then
VI if ($M[j] \neq -1$) and $M < M[j] + 1$, then
VII $M = M[j] + 1$
VIII M[i]=M
IX Return M[n] as the longest path

Time Complexity: This algorithm takes $O(n^2)$ time.Since we have two for loops here which are running on each vertex and then on all edges. So this algorithm is quadratic in complexity time.

4. **Claim:** The optimal solution for the given problem is given by :

$$Opt(i) = min_{j \leq i}\{Opt(j-1) + Qualityj....n\},$$

where Opt(i) is defined as the score of the best segmentation of the prefix consisting of the first i characters of y and Quality(a, ..., b) means the quality of the word that is formed by the characters starting from position a and ending in position b.

Correctness of the claim: We can prove its correctness with induction at index i. Base case is trivial,since there is only one word with one letter. For the inductive step, we consider the last word in the optimal segmentation of this prefix. Let's assume it starts at index j¡=i. Then, prefix containing only first j-1 characters must also be optimal. But according to our induction hypothesis, opt(j) will yield us the value of aforementioned optimal segmentation. Therefore optimal cost opt(i) would be equal to opt(j) plus the cost of last word.

Algorithm: M-SegmentedWords

```
I    M = Array of length n + 1
II   M[0] = 0
III  for i=1 to n
IV       M[i] = 0
V        for j=0 to n-1
VI           if (quality(y_{j+1}...y_i) + M[j] ≥ M[i])
VII              M[i] = quality(y_{j+1}...y_i) + M[j]
VIII         Return M[n]
```

Analysis of Time Complexity: The time complexity for this algorithm is $O(n^2)$. Since it is just two nested for loops, each of which runs (at most) from 1 to n , and inside the two for loops the time is constant. Correctness is straightforward by induction. We claim that $M[i] = Q(i)$ for all i . This is sufficient since we will return M [n], which will be equal to Q (n), which is by definition the optimal value of the instance. For the base case, note that we set $M[0] = 0 = Q(0)$ by definition. For the inductive step, the inner for loop is calculating the maximum of a final block together with its prefix.

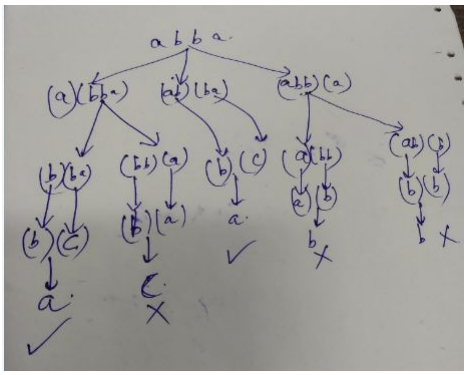
5. Given is the table

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

We have to give an efficient algorithm that examines a string of these symbols and decides whether or not it is possible to parenthesize in a way such that the value of the resulting expression equals a. Since there are three symbols, let us consider a matrix of size 3*3, say M[3][3].

Claim: $Res = isPresent_{k=1}^{k-1}(Mult(i,k) * Mult(k+1,j))$,

where IsPresent is defined as checking the product of every literal with others and verifying if it results in a. **Proof of Correctness:** We are verifying recursively every literal with other literals in the given string. So, result of any product contains a will return as true, otherwise false. Below is the example proving this claim diagrammatically. It not only results in true or false of the claim, but will also explain proper parenthesis where this algorithm will get pass.



Algorithm: MMulti(i,j,n) //i=start of string , j= end of string, n= number of symbols

```
 I  Initialize M[m][n]=0, for all m,n. define globally.
 II  Initialize isPresent[][][]= -1, define globally.
 III  if (j-i) ¡=0, return 0
 IV  if isPresent[i][j]['a']!=-1, then
 V        return isPresent[start][end]['a']
 VI  for k=i= to j
 VII       for x=1 to n
 VIII            for y=1 to n
 IX                   if M[x][y]== 'a'
 X                         if(MMulti(i, k, x)==1 and MMulti(k+1, j, y)==1 )
 XI                              {
 XII                             isPossible[i][j]['a'] = 1;
 XIII                            return 1;
 XIV                             }
 XV  Return 0
```

Time Complexity Analysis: The given algorithm will take $O(n^3)$time. Since three for loops are execiting in O(n).

6. **Claim:** We define function Denom(v) as true if it makes change using available denominations {x1, x2, ..., xn}. i.e

$$Denom(v) = \forall_{1<=i<=n} \{Denom(v - x_i), if x_i \leq v$$

false, otherwise

Correctness of Claim:
If it is possible to make change for v using given denominations x1, x2,...,xn, then it is possible to make change for v xi using the same denominations with one coin of xi being chosen. But since we dont know which i will finally give us true value (which indicates change of v is possible) we will do a logical or over all i.
Algorithm: MakeChange(x1, x2,... xn, V)

```
 I  Declare an array D of size V+1
 II D[0]=true
 III for i=1 to V
 IV      D[i]= false
 V  for v=1 to V
 VI      for j=1 to n
 VII           if x_j <= v
 VIII                D[v] = D[v] ∨ D[v - x_j]
 IX            else
 X                   D[v] = false
 XI Return D[v]
```

Analysis of Time Complexity: There are two for loops in the algorithm where one loop runs $\mid V \mid$ times while the other loops run $\mid n \mid$ times. Therefore the complexity of the algorithm is $O(\mid n \mid\mid V \mid)$.

References:

1. Jon KleinBerg, Eva Tardos. *Algorithm Design.* Pearson Addison Wesley

2. Thomas H. Coremen, Charles E.Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts London, England.

Collaborators: