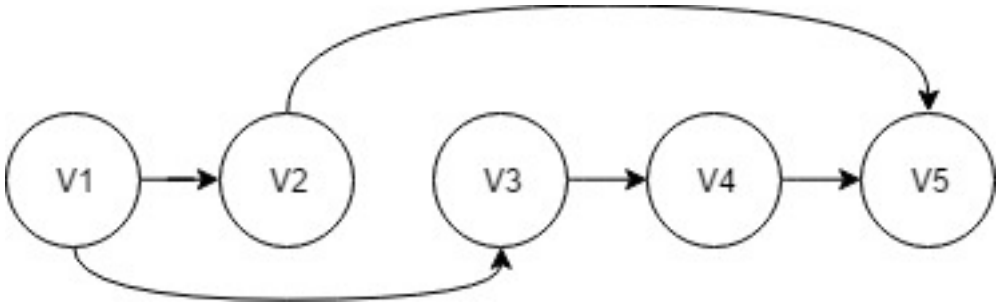
Cody Hubbard 004843389

CS180 HW5

CH8 #3
Solution:

(a)



Consider the above ordered graph. The algorithm supplied by part (a), which simply follows the path leading to the next connected node with the smallest possible j value returns the path $V1 \rightarrow V5$ which is of length 2. However the longest path is actually $V1 \rightarrow V3 \rightarrow V4 \rightarrow V5$ which is of length 3.

(b)
Find Longest Path - FLP()
create array of size j, $Len[j]$
$Len[0] = 0$, $Len[1] = 0$
for each node $i = 2$ to j
$Len[i] = max\{Len[k]\} + 1$ where k is all nodes connected to node$i$
return $Len[j]$

This algorithm will compute the logest path by going though all paths that lead to every node and placing all partial maximum lengths into an array, and subsequently placing the longest possible path to our destination into its position in said array as well. This will give the correct answer for any node in our graph as well as any nodes which are possibly added to our graph (or the case where our graph was the sub-problem). Lastly this algorithm should run in time complexity of $O(n^2)$ becasue it will go though all nodes in our graph n and each node could have at the most $n-1$ edges connected to it $O(n(n-1) = O(n^2)$

CH8 #4
Solution:

(a)

	Month 1	**Month 2**	**Month 3**	**Month 4**
NY	1	2	1	2
SF	2	1	2	1

M=10

Consider the above table, the algorithm given in (a) would give the solution $[NY, SF, NY, SF]$ which is $1+10+1+10+1+10+1=34$ however one of the the two optimal solutions is simply $[NY, NY, NY, NY]$ which gives a total cost of $1+2+1+2=6$.

(b)

	Month 1	**Month 2**	**Month 3**	**Month 4**	**Month 5**	**Month 6**	**Month 7**	**Month 8**
NY	1	1	1000	1	1	1	1	1000
SF	1000	1	1	1	1000	1	1	1

M=10

This example causes every optimal plan to change locations at least three times becasue the cost of staying in the alternative city is very high in specific months, especially due to the relatively low moving costs.

(c)

Create a matrix[row][column] of size $2xn$ $Costs[2][n]$ where n is the total number of months, row one represents NY, and row two represents SF as in the above examples.

$Costs[NY][1] = NY1$
$Costs[SF][1] = SF1$
for each month i to total months n
$Costs[NY][i] = NYi + minimum\{Costs[NY][i-1], Costs[SF][i-1] + M\}$
$Costs[SF][i] = SFi + minimum\{Costs[SF][i-1], Costs[NY][i-1] + M\}$
return lesser of $Costs[NY][n]$ and $Costs[SF][n]$

This algorithm will systematically go though each month and check if the cumulative cost of operating optimally and having been in the current city or the alternative city (plus moving cost) in the previous month will lead to a lower total cost of operation. It will contine this behavior up to the last month n as well as any months which could be added to our set (or the case where our set was the sub-problem). Lastly this algorithm should run in time complexity of $O(n)$ becasue it simply goes though every possible moth for its calculations which are all $O(1)$ as they are doing mathematical operations, accessing an array's values, and storing a value in an array.

CH8 #6
Solution:

First create some sort of array to hold precomputed Slack of the line values which can be looked up in some constant time, call is $SL[]$. $SL[]$ will be an array of word legnths to line slacks.

Line slacks should be computed using $\left(L - \sum wordlengths(i \to j) + (j - i)\right)^2$ as mentioned in the problem

Create an array $Costs[n]$ for n words, and a Last Word array $LW[n]$ to hold the last word of each line.

Let $L[n]$ be an array, of the $Length$ of each word, and $words[n]$ be an array of the words themselves

$Costs[1] = 0$

$k = 0$ (a counter for the last word array)

for $i = 1$to n

  $bc = i$ (a counter for going backwards though our words)

  $Costs[i + 1] = \infty$

  $tempCost = Costs[i] + SL[L[i]]$

  if $tempCost < Costs[i + 1]$

    $Costs[i + 1] = tempCost$ and $LW[k] = words[i]$

  until $tempcost >= \infty$or $bc < 2$

    $bc - -$

    $tempCost = Costs[bc] + SL[L[bc]]$

    if $tempCost < Costs[i + 1]$

      $Costs[i + 1] = tempCost$ and $LW[k] = words[i]$

This algorithm should go though all of our words and there relative slack costs on the line and compute the arrangemnt with the minimal slack cost and store the last words of each line in an array which can later be used to wite out the exact arrangement of optimal words found. This algorithm is correct becasue when adding a new word to our list it will only ever change the last line, that is we can never change any of the previous lines to achieve a smaller slack cost as those lines were already computed optimally with our current global line length. The running time of this algorithm is porportional to global line length divided by the average length of the words, as well as well as how many words we have n. Thus the complexity will be something like $O(length \cdot n)$. this is due to the fact that once a word can no longer fit on a line we no loger need to consider the words that come after it which saves many computations as well as the fact that we precomputed the slack of the line array.

CH8 #12
Solution:

Create an array of total costs $TC[n]$ where n is the number of servers, and where the placement costs array $PC[n]$ and access costs array $AC[n]$ are already given

for any file (f) server (j) combo

for server $i = 0$ to j

for server $k = 0$ to i

$TC[i] = PC[i] + minimum_{k \to i}\Big(TC[i-1] + \sum_{z=k+1}^{i} AC[z]\Big)$

return minimum $TC[]$ from i to j

This algorithm should systematically go though and create an array of optimal total costs from servers 0 to j by searching for the best possible places to place any copies of the data on servers between$i$ and j. It will contine this behavior up to the last server n as well as for any servers which could be added to our set (or the case where our set was the sub-problem). These caluclations include the cost of placement into server j which has a required placement condition. Since this algorithm has a nested for lop of maximum size n, the number of server the running time would look something like $O(n(n(1))) = O(n^2)$.