Jason Lai
204995126
Dis 1B

CS 180
May 16, 2019

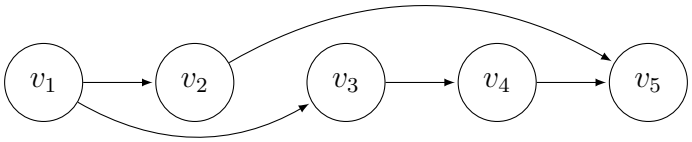
Homework 5

1 Problem 3

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

a) To show that the given algorithm is incorrect, this is an example of an ordered graph G, where the correct longest path is of length 3 and follows the edge order (v_1, v_3), (v_3, v_4), (v_4, v_5). However, the path that the given algorithm provides is of length 2 and follows the edge order (v_1, v_2), (v_2, v_5), which is incorrect; this is because the given algorithm fails to account for longer paths that the node nearest to the current one in the order of the graph could skip.

Graph G



b) An efficient algorithm that takes an ordered graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n is as follows:

```
getLongestPathLength(n) {
   Array DP[1 to n]
   Set DP[1]=0
   For i=1 to n {
      DP[i]=-∞
      For all edges j out of node i {
         if DP[j] + 1 > DP[i] {
            DP[i] = DP[j] + 1
         }
      }
   }
   Return DP[n]
}
```

Here, the algorithm checks all edges out of each node i, takes the longest path length out of the node i, and stores that path length into the DP array; at the end of the algorithm, the function returns the last element of the DP array, which stores the longest path length of the ordered graph G. The running time for this algorithm is $O(n^2)$, as it has a nested loop that iterates through n nodes in the outer loop and iterates through possibly n nodes in the inner loop.

2 Problem 4

Given a value for the moving cost M, and sequences of operating costs N_1,..., N_n and S_1,..., S_n, find a plan of minimum cost.(Such a plan will be called optimal.)

a) To show that the given algorithm is incorrect, this is an example where the number of months $n = 4$ and the fixed moving cost $M = 10$. This table shows the possible operating costs for each month in New York (NY) and San Francisco (SF).

	Month 1	**Month 2**	**Month 3**	**Month 4**
NY	5	10	5	10
SF	10	5	10	15

From the example, we can see that the correct plan of minimum cost would be to stay in New York for all four months [NY, NY, NY, NY], where the cost of the plan is $5+10+5+10 = 25$. However, the plan that the given algorithm provides is [NY, SF, NY, NY], where the cost of the plan would then be $5 + 5 + 5 + 10 + 10 + 10 = 45$; this is due to the fact that the given algorithm doesn't account for the cost of switching from one city to the other, naively taking whichever city has the lower cost.

b) An example of an instance in which every optimal plan must move at least three times is as follows: suppose the number of months $n = 4$ and the fixed moving cost $M = 10$. This table shows the possible operating costs for each month in New York (NY) and San Francisco (SF).

	Month 1	**Month 2**	**Month 3**	**Month 4**
NY	5	100	5	100
SF	100	5	100	5

From this example, we see that the optimal plan of minimum cost is [NY, SF, NY, SF] where the cost is $5+5+5+5+10+10+10 = 50$, since the plan moves three times. Any other plan would have to include at least one operation cost of 100, which would make those plans >50, so this means all other plans are suboptimal; the only optimal plan is the one that starts in New York and moves three times.

c) An efficient algorithm that takes values for n, M, and sequences of operating costs N_1,..., N_n and S_1,..., S_n, and returns the cost of an optimal plan is as follows:

```
getMinCostPlan(M, N, S, n) {
  Array DP_N[1 to n]
  Array DP_S[1 to n]
  DP_N[1] = 0
  DP_S[1] = 0
  For i=1 to n {
    DP_N[i] = min(DP_N[i-1], M + DP_S[i-1]) + N_i
    DP_S[i] = min(DP_S[i-1], M + DP_N[i-1]) + S_i
  }
  Return min(DP_N[n], DP_S[n])
}
```

Here, since there are two sequences of operating costs, the algorithm uses two arrays to keep track of previously calculated values. For n months, an optimal plan up to the ith month will end in either San Francisco or New York, so at each iteration of a month, we take the current N_i and S_i, add them to the optimal plan up to the previous (i-1)th month, and store them in their respective arrays. The minimum of the last element of either array is then the minimum cost of the optimal plan up the nth month. The running time for this algorithm is $O(n)$, since there is a single loop iterating through the operating costs of the N and S sequences for n months.

3 Problem 6

Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the squares of the slacks of all lines (including the last line) is minimized.

The objective is to find the minimum total of the squares of the slacks for a set W of n words w_1, w_2, ..., w_n to produce "even" right margins for a pretty print format. An efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the squares of the slacks of all lines is minimized is as follows:

```
getLeastSquaresSlack(W) {
  Array DP[1 to n]
  Set DP[1]=L-w_1
  For i=2 to n {
    Set slack=∞
    For j=1 to i {
      Set temp_slack = DP[j] + (((i-j-1) + ∑_{k=j}^{i} w_k)%L)^2
      if temp_slack < slack {
        slack = temp_slack
      }
    }
    DP[i] = slack
  }
  Return DP[n]
}
```

Here, the algorithm works by calculating the minimum total sum of the square of slacks for every additional word w_i added. Each element of the DP array stores the minimum total square slack sum for up to that word represented by that element of the DP array. To compute this, the algorithm assumes that all previous slacks in the DP array were found optimally, and adds the square slack sum for a word w_j DP[j] to the square slack found for words w_j to w_i. It then compares these sums and takes the minimum value, thereby using a recurrence relation to get the sums to obtain a minimum square slack total. At the end, the minimum square slack total is stored in the element DP[n] and returned; to find the optimal partition for the set of n words W, we backtrack from DP[n] to see all lines that contain a subset of the n words W. These lines are the partition of the optimal solution. The running time of this algorithm is $O(n^2)$ as we iterate through all n words, and for each word w_i, we must find the minimum square slack total for the previous i-1 words, which iterates through another i-1 elements of the DP array.

4 Problem 12

Give a polynomial-time algorithm to find a configuration of minimum total cost.

The objective is to find a configuration of minimum total cost, wherein the total cost is the sum of placement costs for a file replicated over n servers S_1, S_2, ..., S_n, and the sum of the access costs over said n servers. A polynomial-time algorithm to find a configuration of minimum total cost is as follows:

```
getMinCostConfig(S) {
  Array DP[1 to n]
  Set DP[1]=c_1
  For i=2 to n {
    Set optimal_cost=∞
    For j=1 to i-1 {
      Set temp_optimal = DP[j] + Σ_{k=j}^{i-1} k
      if temp_optimal < optimal_cost {
        optimal_cost = temp_optimal
      }
    }
    DP[i] = c_i + optimal_cost
  }
  Return DP[n]
}
```

Here, the algorithm proceeds as follows: we operate under the assumption that every server S_i is going to have the file copied to it and compute the minimum total cost configuration up that server S_i and store it in the element DP[i] of the array. To compute the optimal configuration, this is the same as the sum of the cost c_i to copy the file to server S_i and the minimum of the set of sums of the elements in the DP array and their access cost sums when trying to reach the file at S_i. In other words, the minimum cost of the optimal configuration up to the server S_i is $c_i + \min(\text{DP}[j] + \sum_{k=j}^{i-1} k)$, where j is any server from 1 to i-1. Note, we initialize the first server S_1 to equal the cost c_1, since there are no servers before it. By the end of the algorithm, DP[n] contains the total cost of the minimum total cost configuration for all n servers; to get the configuration itself, we backtrack from DP[n] to get all previous configurations in the DP array used to create the final configuration. The running time of this algorithm is $O(n^2)$, as we iterate through all n servers and for each server S_i, we must iterate through the DP array for all elements 1 to *i-1* to get the minimum of the previous configurations, hence $O(n^2)$.