ADA Assignment 2 Theory

Ayushi Jain 2019031

Kesar Shrivastava 2019051

Problem 1.

Subproblems: Let for all i = 0, 1, 2,..., n and candy = c1, c2 and c3, dp[i][candy] store the optimal solution of the subproblem. It is an nX3 matrix. MaxScore(C[1...n], i, candy) denotes the maximum score at the (n-i)-th position in the game with the given candy in hand. Note that i > n denotes that the player is not at any valid position and hence the score is 0. Also, C[1..n] is the array of the candies owned by the i-th animal.

Recurrence:

1. Base case: For any candy, MaxScore(C[1...n], 0, candy) = 0
2. For all i > n, MaxScore(C[1...n], i, candy) = 0
3. For all 1 <= i <= n, C[i] = candy

   MaxScore(C[1...n], i, candy) = MaxScore(C[1...n], i+1, candy)+1
4. For all 1 <= i <= n, C[i] ≠ candy

   MaxScore(C[1...n], i, candy) = max((MaxScore, C[1...n], i+1, C[i])-1, (MaxScore, C[1...n], i+1, candy))

Final Solution: We want to compute MaxScore(C[1...n], 0, candy). dp[0][candy] will give the answer.

Correctness of Recurrence: We observe that the optimal solution MaxScore(C[ ], i, candy) can have three cases with respect to the candy that the player has. We will prove the correctness for all the three cases.

Case 1: The candy that the player has at the i-th position is the same as the candy the animal of i-th position has. In this case, the player exchanges the candy with the animal and gets the score increased by 1.

We claim that MaxScore(C[ ], i+1, candy) = MaxScore(C[ ], i, candy)-1, this is the optimal solution for the n-(i+1)-the position. Suppose this is not true. Then MaxScore(C[ ], i+1, candy) is a different optimal solution for the subproblem as here we fill the dp array from right to left such that MaxScore(C[ ], i+1,

candy) > MaxScore(C[ ], i, candy)-1. But then we can create another feasible solution for the given positions by taking this solution and subtracting 1. Clearly this is the feasible solution contradicting the optimality of the latter.

MaxScore(C[ ], i+1, candy) = opt[i+1] is not the optimal solution. Let opt'[i+1] be the optimal solution. Thus, opt'[i+1]+1 is the optimal solution for MaxScore(C[ ], i, candy). However, this is a contradiction as the optimal for MaxScore(C[ ], i, candy) is opt[i+1]+1. This implies that opt[i+1] is the optimal for MaxScore(C[ ], i+1, candy)

Case 2: The candy that the player has at the i-th position is different from the candy that the animal of i-th position has. In this case, the player may or may not exchange the candy. This gives rise to further two cases:

a. The player does not exchange the candy and there is no change in the score.

We claim that MaxScore(C[ ], i+1, candy) = MaxScore(C[ ], i, candy), this is the optimal solution for the n-(1+i) -the position. Suppose this is not true. Then MaxScore(C[ ], i+1, candy) is a different optimal solution for the subproblem as here we fill the dp array from right to left such that MaxScore(C[ ], i+1, candy) > MaxScore(C[ ], i, candy). But then we can create another feasible solution for the given positions by taking this solution. Clearly this is the feasible solution contradicting the optimality of the latter.

MaxScore(C[ ], i+1, candy) = opt[i+1] is not the optimal solution. Let opt'[i+1] be the optimal solution. Thus, opt'[i+1] is the optimal solution for MaxScore(C[ ], i, candy). However, this is a contradiction as the optimal for MaxScore(C[ ], i, candy) is opt[i+1]. This implies that opt[i+1] is the optimal for MaxScore(C[ ], i+1, candy)

b. The player exchanges the candy and gets the score decreased by 1.

We claim that (MaxScore, C[1...n], i+1, C[i]) = MaxScore(C[1...n], i, candy) + 1 , this is the optimal solution for the n-(1+i) -the position. Suppose this is not true. Then MaxScore(C[ ], i+1, candy) is a different optimal solution for the subproblem as here we fill the dp array from right to left such that MaxScore(C[ ], i+1, candy) > MaxScore(C[ ], i, C[i])+1. But then we can create another feasible solution for the given positions by taking this solution and adding 1. Clearly this is the feasible solution contradicting the optimality of the latter.

MaxScore(C[ ], i+1, candy) = opt[i+1] is not the optimal solution. Let opt'[i+1] be the optimal solution. Thus, opt'[i+1] is the optimal solution for MaxScore(C[ ], i, C[i]). However, this is a contradiction as the optimal for MaxScore(C[ ], i, C[i]) is opt[i+1]-1. This implies that opt[i+1] is the optimal for MaxScore(C[ ], i+1, candy)

Pseudocode:

```
MaxScore(C[1...n], i, candy){
      if ( i>n ){
            return 0;
      }
      if (dp[i][candy] != null) return dp[i][candy];
      if ( dp[i][candy] == 0 ){
            if ( C[i] == candy ){
                  dp[i][candy] = MaxScore(C[1...n], i+1, candy)+1;
            }
            else{
                  dp[i][candy] = max((MaxScore(C[1...n], i+1, C[i])-1),
            MaxScore(C[ ], i+1, candy));
            }
      }
      return dp[i][candy];
}
```

Runtime: The runtime of the algorithm is O(n) because we are calling the function n number of times for the n positions.

Problem 2.

Subproblems: Let for all i = 0, 1, 2, · · · n-1 and j = 1, 2 · · · k, minScore(i, j) denote the optimal solution for placing j bakeries till i-th house, where the i-th house certainly has j-th bakery placed.

k = 0 means that the solution is not possible and we return null**.** Here, the grid or the dp is the matrix which stores the optimal solution. Its size is (n+1)X(k+1).

If the number of houses is less than k, then also such an answer is not possible and we return null assuming that one house cannot have more than one bakery.

Recurrence:

1. Base Case: grid[0][1] = 0 (if there is only one bakery to be placed in one house)
2. For 0 <= i < n and 1 <= j <= k
   grid[i][j] = grid[t][j - 1] + (summation of absolute distances of houses from (t + 1 to i inclusive) to their nearest bakeries (that is, position t or position i). Where t is from 0 to i - 1).

Final Solution:

ans = Math.min(ans, grid[i][k] + Sum of distances of houses of i + 1th position to n-th position from i-th server)) for i from .

Correctness of Recurrence:

Proof by induction:

P(i, j) = For i houses on a line, j bakeries are optimally placed and thus it gives the minimum distance.

Base Case: P(0, 1): One house one bakery because the indexing of houses is done from 0-th position and that of bakeries is done from first position. So P(0, 1) is true as there is only one way of keeping the bakery and that one way is to keep the bakery at that particular house. Thus, the distance is 0.

Inductive Hypothesis: P(i, j-1) is true -> All the j-1 bakeries are correctly placed for the given i houses.We use strong induction here. That is all subproblems P(a, b) are true where a < i and b <= j. Also, all subProblems P(i, u) are true where u is less than j.

Inductive Step: We prove that the recurrence is true for P(i, j).

From the recurrence we know that grid[i][j] = grid[t][j - 1] + (summation of absolute distances of houses from (t + 1 to i inclusive) to their nearest bakeries (that is, position t or position i) where t is from 0 to i - 1).

From the inductive hypothesis we know that grid [t][j-1] gives the optimal solution. To get the optimal solution for placing j bakeries in i houses we only need to add the all the absolute distances of houses from t+1 to i to their nearest bakeries. Hence, grid[i][j] gives the minimum distance.

Thus by induction it is proved that the recurrence is correct.

Proof by contradiction:

We claim that the minimum distance for the homes to their nearest bakeries is given by grid[i][j] = grid[t][j - 1] + (summation of absolute distances of houses from (t + 1 to i inclusive) to their nearest bakeries (that is, position t or position i) where t is from 0 to i - 1).

Let's assume that grid[t][j-1] is not the optimal answer for any t.
Then grid[t][j-1] will be some answer that is different from the above claim and which is less than the initial solution. Let opt'[t][j-1] be its optimal solution instead of opt[t][j-1] Hence we can create a different feasible solution for the homes till position t in the array.
This implies that the position of the bakeries is now different. Now to get the solution of positioning j bakeries till i-th position we can get the same solution but now append one more position of bakery to the original solution.
-> grid[i][j] = opt'[t][j-1] + (summation of absolute distances of houses from (t + 1 to i inclusive) to their nearest bakeries (that is, position t or position i) where t is from 0 to i - 1).
According to our solution, the optimality of grid[i][j] depends on the optimality of the subproblems. So the optimal solution for the larger problem was grid[i][j] = opt[t][j-1] + (summation of absolute distances of houses from (t + 1 to i inclusive) to their nearest bakeries (that is, position t or position i) where t is from 0 to i - 1). Hence, each subproblem has to be optimal because if it is not the case then this is a contradiction as it contradicts the optimality of grid[i][j] hence our assumption is wrong and the claimed recurrence is correct.

Pseudocode:

```
minScore(A[1...n], n, k){
      dp = new int[n+1][k+1];
      dp[0][1] = 0;
      for(int i = 0; i<n; i++){
            for(int j = 1; j <= k; j++){
                  U = Integer.MAX_VALUE;
                  if(i+1>=j){
                        for(int k1 = 0; k < i; k++){
                              T = 0;
                              if(dp[k1][j-1]!=null){
                                    T = dp[k1][j-1];
                                    for(int i1 = k1; i1 <= i; i1++){
                                          T += Math.min(Math.abs(arr[i1] -
                              arr[i]), Math.abs(arr[i1] - arr[k1]));
                                          U = Math.min(u, t);
                                    }
```

```
                        }
                        else{
                              for(int i1 = 0; i1 <= i; i1++){
                                    T += Math.abs(arr[i1] - arr[i]);

                              }
                              U = Math.min(U, T);
                        }
                  }
            }
            if( U!=Integer.MAX_VALUE)
                  dp[i][j] = U;
      }
   }
   ans = Integer.MAX_VALUE;
   for (int i = 0; i < dp.length; i++){
         int g = Integer.MAX_VALUE;
         if (dp[i][k] != null){
               g = dp[i][k];
               for (int o = i + 1; o < arr.length; o++){
                     g += Math.abs(arr[o] - arr[i]);
          }
      }
      ans = Math.min(g, ans);
   }
   System.out.println(ans);
}
```

Runtime

O((n^3)k) as we need to fill in n*k cells and for each cell grid[i][j] we need to iterate from 0 to i - 1 and for each iteration we need to calculate the sum of distances of the houses to their nearest bakery.

Problem 3.

Subproblems: Let for any i = 1,2,...n and j = 1,2,...n, mergeTwoNumbers(A[1...n], i, j) denote the optimal solution for the problem defined for the droplets d_i, d_(i+1), ..., d_j. In other words, it returns an array of two elements where the 0th position is the volume of the resultant droplet while the 1st position is the total energy released to form the resultant droplet.

Recurrence:

1. Base case:
    a. 1 drop, n = 1, there is nothing to merge this droplet with so we return {0, 0}
    b. 2 drops, n=2, mergeTwoNumbers(A[1...n], 1, 2) = { A[1]+A[2], A[1]^2+A[2]^2 }
    c. 3 drops, n = 3,

```
mergeTwoNumbers(A[1...n], 1, 3) = int[ ] answer;
int a = (A[1]^2+A[2]^2)+(A[1]+A[2])^2+A[3]^2;
int b = (A[2]^2+A[3]^2)+(A[2]+A[3])^2+A[1]^2;
answer[1] = max(a, b);
return answer;
```

2. For all 1 <= i <=n, 3 <= j <= n: int[ ] ans = new int[2]; m varies from i to j

```
int[ ] k1 = mergeTwoNumbers(A[1...n], i, m-1);
int[ ] k2 = mergeTwoNumbers(A[1...n], m+1, j);
int[ ] k3 = new int[2];
k3[0] = k1[0] + k2[0];
k3[1] = k1[1] + k2[1] + k1[0]^2 + k2[0]^2 + A[m]^2 + Math.max((k1[0]+A[m])^2,
(k2[0]+A[m])^2);

if (k3[1] >= ans[1]){ans[1] = k3[1]; ans[0] = k3[0];}
```

The final answer array for grid[i][j] is the array with maximum value at 2nd position.

Final Solution: We want to compute mergeTwoNumbers(A[1...n], 1, n). The dp matrix will contain the answer. dp[1][n] will give the final answer.

Correctness of recurrence:

Proof by Induction:

P(i, j) = We observe that each droplet has two possibilities of getting combined, either with its left droplet or right except for the first and the last droplet which has been taken in consideration with the base cases. The droplets at the end of the array will only be combined with its right or the left droplet if it is the

leftmost or the rightmost droplet respectively. Thus, the predicate is that the function returns the maximum energy between the indices i and j.

Base Case: P(1, 1) = there is nothing to merge this droplet with so we return {0, 0}.

P(1, 2) = there are two drops and the maximum energy is square of two drops.

P(1, 3) = three drops and we take maximum of the two possibilities.

Inductive Hypothesis: P(i, j) is true for the droplets d_i, d_(i+1), …, d_j and for any subinterval in between. Thus, we use strong induction here.

Inductive Step: We prove that P(i, j+1) is true.

In the for loop there are two function calls when m varies from i to j+1: mergeTwoNumbers(A[1...n], i, m-1) and mergeTwoNumbers(A[1...n], m+1, j)

By the inductive hypothesis these subproblems return the correct answer as the length of the interval in less than j-i for any subproblem

1. So , for an interval [i, j+1] for all i <= t <= j+1 there will be two cases that either the drop is combined with the left drop first and then the resultant drop is connected with the right drop. Or vice versa.
2. So, the resulting answer for the interval [i, j+1] is the maximum energy returned by all possible combinations, thus considering all possible cases.

Hence proved that the recurrence is correct by strong induction.

Proof by contradiction:

There are two cases:

Case 1: The droplet is combined with its left droplet. We claim that mergeTwoNumbers(A[1...n], i, m) = k1[1] + k2[1] + k1[0]^2 + k2[0]^2 + (k1[0]+A[m])^2. This gives the optimal solution assuming that the other subproblem returns the correct answer. Suppose this is not true, for the sake of contradiction for the droplets from i-th position to the m-th position. Here, the m-th position is in consideration and hence all the droplets getting combined here lies to its left. Then mergeTwoNumbers(A[1...n], i, m) is a different optimal solution for the subproblem containing droplets from i-th position to the m-th position such that mergeTwoNumbers(A[1...n], i, m) > k1[1] + k2[1] + k1[0]^2 + k2[0]^2 + (k1[0]+A[m])^2. But then we can create another feasible solution for the droplets d_i, d_(i+1), …., d_m by taking this solution and combining the rest of the droplets in some different fashion. Thus, the order in which the droplets will be

combined will have some difference from the initial solution and we will get a higher energy but this contradicts the optimality of the problems that are required to get the optimal solution, i.e., k1 and k2. Hence, the recurrence is correct.

Case 2: The droplet is combined with its right droplet. We claim that mergeTwoNumbers(A[1...n], m, j) = k1[1] + k2[1] + k1[0]^2 + k2[0]^2 + (k2[0]+A[m])^2. This gives the optimal solution assuming that the other subproblem returns the correct answer. Suppose this is not true, for the sake of contradiction for the droplets from m + 1-th position to the j-th position. Here, the m-th position is in consideration and hence all the droplets getting combined here lies to its right. Then mergeTwoNumbers(A[1...n], m + 1, j) is a different optimal solution for the subproblem containing droplets from m + 1-th position to the j-th position such that mergeTwoNumbers(A[1...n], m + 1, j) > k1[1] + k2[1] + k1[0]^2 + k2[0]^2 + (k2[0]+A[m])^2. But then we can create another feasible solution for the droplets d_i, d_(i+1), …., d_m by taking this solution and combining the rest of the droplets in some different fashion. Thus, the order in which the droplets will be combined will have some difference from the initial solution and we will get a higher energy but this contradicts the optimality of the problems that are required to get the optimal solution, i.e., k1 and k2. Hence, the recurrence is correct.

Pseudocode:

```
mergeTwoNumbers(A[1...n], s, e){
      if ( s > e ){arr = {0, 0}; return arr;}
      if ( dp[s][e] != null ) return dp[s][e];
      if ( s == e ) {arr = {A[s], 0}; dp[s][e] = arr; return arr;}
      if ( e - s == 1 ){
            arr = {A[e] + A[s], A[e]^2+A[s]^2};
            dp[s][e] = arr;
            return arr;
      }
      ans = new int[2];
      for (m = s; m <= e; m ++){
            k1 = mergeTwoNumbers( A[1...n], s, m-1 );
            k2 = mergeTwoNumbers( A[1...n], m+1, e );
            k3 = new int[2];
            k3[0] = k1[0]+k2[0]+A[m];
            k3[1] = k1[1]+k2[1]+k1[0]^2+k2[0]^2+A[m]^2+
      Math.max((k1[0]+A[m])^2),(k2[0]+A[m])^2));

            if ((k1[0] == 0&&k1[1] == 0)||(k2[0] == 0 && k2[1] == 0)){
            k3[1] -= Math.max((k1[0]+numbers[m])^2, (k2[0]+A[m]^2);
```

```
        }
            if (k3[1] >= ans[1]){ans[1] = k3[1]; ans[0] = k3[0];}
        }
        dp[s][e] = ans;
        return ans;
    }
```

Runtime:

O(n^3) as we need to fill n^2 cells and for each interval s to e, we need to iterate from s to e and find the minimum answer.