Abhimanyu Gupta - 2019226
Abhinav Gudipati - 2019227

Analysis and Design of Algorithms Homework 2

Question -1:

Let the three candies types “circus peanuts”, “Heath bars”, and “Cioccolateria Gardini chocolate truffles” be represented by integers 0, 1 and 2 respectively.

Sub-problems:

i - denotes the i^{th} of the animal
j - denotes the candy type
Let the animals be $a_0, a_1, \dots a_i, \dots a_{n-1}$ (n is the number of animals)
Let $\forall\ i\ \in [0, n)$ and $\forall\ j\ \in [0, 2]$, scores[i][j] denote the optimal answer for the game with animals $a_i, a_{i+1}, \dots a_{n-1}$ and the candy in the hand of the player is ‘j’

Recurrence:

1. Base Case: i == n (no animal remaining to swap candy)
   scores[i][j] = 0, $\forall\ j\ \in [0, 2]$
2. The player and animal have the same candy
   scores[i][j] = 1 + scores[i + 1][j]
3. The player and animal do not have the same candy
   scores[i][j] = max(-1 + scores[i+1][candies[i]] , scores[i+1][j]) {here candies[i] is the candy present with the i^{th} animal}

Final Solution:

scores[0][0] is our final answer
The player tries to swap his candy with all the animals from a_0 to a_{n-1} ; with the initial candy present with him to be of type 0 i.e. “circus peanuts”.

Correctness of Recurrence:

For the i^{th} animal, one of the following cases, are possible,

1. Player and the Animal have the same candy
2. Player and the Animal do not have the same candy

Case 1: Player and the Animal have the same candy
For this case the player is free to either swap or not swap the candy with the animal.
If the player wishes to swap then score[i][j] would become 1 + score[i+1][j].
If the player doesn't want to swap then the score[i][j] would remain the same.
Clearly the best choice for the player is to swap the candy and increase the score.
That is if candies[i] == j then score[i][j] = 1 + score[i+1][j]
Case 2: Player and the Animal do not have the same candy
For this case the player is again free to either swap or not swap the candy with the animal.
If the player wishes to swap then score[i][j] would become -1 + score[i+1][candies[i]].
If the player doesn't want to swap then the score[i][j] would remain the same.
Here the score is not just dependent on the effect of the swap process on the score, it is also dependent on the score the player gains after he receives the i^{th} animal candy in swapping.
So, the optimal answer in this case is to take the maximum score of both the cases
That is if candies[i] !=j then score[i][j] = max(score[i+1][j], -1 + score[i+1][candies])

Pseudocode:

```
FUNCTION CandyScoreRoutine(candies, n, i, candy, scores):
   IF i = n:
       RETURN 0
   ENDIF

   IF scores[i][candy] != -inf:
       RETURN scores[i][candy]
   ENDIF

   IF candies[i] = candy:
       scores[i][candy] <- 1 + CandyScoreRoutine(candies, n, i + 1, candy, scores)
   ELSE:
       scores[i][candy] <- max(-1 + CandyScoreRoutine(candies, n, i+1, candies[i], scores),
CandyScoreRoutine(candies, n, i+1, candy, scores))
   ENDIF

   RETURN scores[i][candy]

ENDFUNCTION

FUNCTION CandyScore(candies):
   types <- 3
   startCandy <- 0
   n <- len(candies)

   scores <- [[-inf for _ in range(types)] for _ in range(n)]
```

```
   RETURN CandyScoreRoutine(candies, n, 0, startCandy, scores)

ENDFUNCTION
```

Runtime :

O(n) :- We have 3 * n states in our DP, each of which is visited only once, hence the total run time of our algorithm is O(n).

Question-2:

Sub-problems:

i - denotes the i^{th} of the house

j - denotes the number of bakeries

Let the houses be $h_0, h_1, \ldots h_i, \ldots h_{n-1}$ (n is the number of houses)

Let $\forall\ i \in [0, n)$ and $\forall\ j \in (0, k]$, sumDistances[i][j] denote the optimal answer for the houses $h_0, h_1, \ldots h_i$ having j spots for the bakeries distributed among themselves.

RecurrenceMarks:

1. Base Cases:
    a. i=-1 and j=0, i.e. all k spots have been picked.
    sumDistances[i][j] = 0
    b. i=-1 and j!=0, i.e j spots for bakeries have not been picked.
    sumDistances[i][j] = inf
    c. i!=-1 and j==0, i.e i houses are not considered while picking spots for the bakeries
    sumDistances[i][j] = inf
2. $\forall\ i \in [0, n)\ \ and\ \ j > 0$
    sumDistances[i][j] = $min_{k=0\ \ldots\ i}$ (sumDistances[k-1][j-1] + sumMedianDistance[k][i]),
    where sumMedianDistance[k][i] is the sum of distances walked by people of houses k to i, each day to visit bakery present in a house h_b with $k \le b \le i$.

Final Solution :

sumDistances[n-1][k] ,

(n-1, k) means we are picking k bakeries from houses 0 to n-1, for positioning our bakeries.

Correctness of Recurrence :

For a range of houses, $h_0, h_1, \ldots. h_i$ with k spots to be picked up, we can break the problem into two parts, picking k-1 spots in the range $h_0, h_1 \ldots. h_{j-1}$ and picking 1 spot in the range $h_j, h_{j+1} \ldots h_i$ (for some j such that $0 \le j \le i$).

Then the sum of distances to the nearest bakery in the range (0,i) with k bakeries, i.e sumDistance[i][k] would become distance[j][i] + sumDistance[j-1][k-1], where distance[j][i] is the minimum sum of all the distances to the nearest bakery for the range of houses $h_j, h_{j+1} \ldots h_i$.

Therefore the sumDistance[i][k] = $\min_{j=0.....i}$ (distance[j][i] + sumDistance[j-1][k-1])

Similarly, sumDistance[j-1][k-1] can be computed optimally.

So, we can deduce that for an appropriate unique value of j , the quantity { distance[j][i] + sumDistance[j-1][k-1] } would be minimum and hence, sumDistance[i][k] would be computed optimally as the least minimum sum of distances.

Pseudocode:

```
FUNCTION getMinimumDistanceRoutine(distance, sumDistances, n, k):
   IF n = -1 AND k = 0:
       RETURN 0
   ELSEIF n = -1 OR k = 0:
       RETURN inf
   ENDIF
   IF sumDistances[n][k] != inf:
       RETURN sumDistances[n][k]
   ENDIF

   for i in range(n, -1, -1):
       sumDistances[n][k] <- min(sumDistances[n][k], distance[i][n] +
getMinimumDistanceRoutine(distance, sumDistances, i-1, k-1))
   ENDFOR

   RETURN sumDistances[n][k]

ENDFUNCTION

FUNCTION getMinimumDistance(houses, n, k):
   sumDistances <- [[inf for _ in range(k+1)] for _ in range(n)]

   distance <- [[0 for _ in range(n)] for _ in range(n)]

   for i in range(n):
       for j in range(i+1, n):
           medianHouse <- houses[(i+j)//2]
           for l in range(i, j+1):
               distance[i][j] += abs(medianHouse - houses[l])
           ENDFOR
       ENDFOR
   ENDFOR

   RETURN getMinimumDistanceRoutine(distance, sumDistances, n-1, k)

ENDFUNCTION
```

Runtime :

$O(n^3)$ -

The matrix sumMedianDistance is filled in $O(n^3)$ computations.

The matrix sumDistances has n*k states, each of which requires n computations to be filled.

Hence the whole sumDistances is filled in $O(n^2 * k)$ computations.

Therefore total Complexity is $O(n^3 + n^2 * k) = O(n^3)$ [$k \leq n$]

Question-3:

Sub-problems:

Let n denote the number of drops.

Let $\forall\ i\ \in [0, n)$ and $\forall\ j\ \in [i, n)$,energy[i][j] denote the maximum energy gained when drops d_i, d_{i+1}, … d_j are merged to form a single drop

Recurrence:

1. Base Case: i=j
   energy[i][i] = 0, energy gained by just a single drop is 0 i.e the initial energy a drop possesses is 0.
2. $\forall\ j\ >\ i$
   energy[i][j] = max(energy[i][j-1] + combinedDrops[i][j-1]^2 + drops[j]^2 , energy[i+1][j] + drops[i]^2 + combinedDrops[i+1][j]^2)
3. $\forall\ j\ <\ i$
   energy[i][j] = 0

Final Solution:

energy[0][n-1] is the required maximum energy, when drops d_0, d_1, … d_{n-1} are merged.

Correctness of Recurrence :

Let energy[i][j] denotes the energy gained when all the drops from d_i to d_j are combined.

We can combine the drops i to j in primarily two ways:

1. Combine the drops i to j-1 and then combine this combined drop with the j^{th} drop.
   In this the total energy gained till this point would be energy[i][j-1] (energy gained while combing drops i to j-1) and combinedDrops[i][j-1]^2 (volume of the combined drop from i to j-1) + drops[j]^2, which is-
   energy[i][j-1] + (combinedDrops[i][j-1]^2 + drops[j]^2)
2. Combine the drops i+1 to j and then combine this combined drop with the i^{th} drop
   Drawing on the lines of the method followed in the 1st point , energy gained in this case would be (drops[i]^2 + combinedDrops[i+1][j]^2) + energy[i+1][j]

Now, the maximum energy gained when combing drops from d_i to d_j would be
energy[i][j] = max(energy[i][j-1] + (combinedDrops[i][j-1]^2 + drops[j]^2) , (drops[i]^2 + combinedDrops[i+1][j]^2) + energy[i+1][j])

Recursively applying the same logic on smaller parts, we can have all the possible ways to merge drops d_i to d_j, the maximum desired energy would be the max of all possible energies at each level of merging.

Pseudocode :

```
FUNCTION getMaxEnergy(drops):
   n <- len(drops)

   combinedDrops <- [[0 for _ in range(n)] for _ in range(n)]
   energy <- [[0 for _ in range(n)] for _ in range(n)]

   for i in range(n):
       combinedDrops[i][i] <- drops[i]
   ENDFOR

   for i in range(1, n):
       for j in range(n-i):
           r <- j
           c <- j+i

           combinedDrops[r][c] <- drops[r] + combinedDrops[r+1][c]

           energy1 <- ( drops[r]^2 + combinedDrops[r+1][c]^2 ) + energy[r+1][c]
           energy2 <- energy[r][c-1] + ( drops[c]^2 + combinedDrops[r][c-1]^2 )

           energy[r][c] <- max(energy1, energy2)

       ENDFOR
   ENDFOR

   maxEnergy <- energy[0][n-1]

   RETURN maxEnerg

ENDFUNCTION
```

Runtime :

$O(n^2)$:

There are n^2 entries in our energy matrix, each of which gets filled only once, which is $O(n^2)$ computations to fill the energy matrix.