CSE222: Assignment 2

Ananya Lohani (2019018) | Mihir Chaturvedi (2019061)

Problem 1: Q14, page 131

Subproblems

Let for all 1 <= i <= n, 1 <= j <= 3, OPT[i][j] denote the optimal score at the end of interacting with the ith animal and ending up with the jth candy. Maximum of OPT[n][1], OPT[n][2] and OPT[n][3] would give us the final solution to the problem.
Note the following numbering convention for candies:
Circus Peanuts: j = 1
Health Bar: j = 2
Cioccolateria Gardini chocolate truffles: j = 3

Recurrence

Base Case:
For i = 1, we are at the stage after we've interacted with the first animal. Since we know that we had candy 1(circus peanuts) at the beginning of the first turn, the following cases are possible:

Case 1: If C[1] = 1

OPT[1][1] = 1 (swapped with C[1] and got +1)
OPT[1][2] = -inf (not possible)
OPT[1][3] = -inf (not possible)

Case 2: If C[1] = 2

OPT[1][1] = 0 (did not swap with C[1])
OPT[1][2] = -1 (swapped with C[1] and got -1)
OPT[1][1] = -inf (not possible)

Case 3: If C[1] = 3

OPT[1][1] = 0 (did not swap with C[1])
OPT[1][2] = -inf (not possible)
OPT[1][3] = -1 (swapped with C[1] and got -1)

We can end up with OPT[i][j] at the end of the ith turn in the following ways:

Case 1: If C[i] = j:

- **Case 1.1:** We had candy j' != j at the end of the $(i-1)^{th}$ animal and we swapped it with C[i], thus: OPT[i][j] = OPT[i-1][j'] - 1
- **Case 1.2:** We had candy j' == j at the end of $(i-1)^{th}$ animal and we swapped it with C[i], thus: OPT[i][j] = OPT[i-1][j] + 1

Case 2: If C[i] != j:

- This is only possible if we did not swap our candy with C[i], meaning that we had candy j at the end of the $(i-1)^{th}$ turn as well. Thus: OPT[i][j] = OPT[i-1][j]

Correctness

For the base cases, the reasoning and correctness of the optimal solutions have been mentioned alongside their assignments. It is also worth noting that if the candy we have right now is the same as the candy the animal whom we are going to interact with is holding, it is always optimal to swap candies to gain a point, rather than not swapping and having no change. This is because, in each scenario, we end up with the same candy for the next interaction.

For i > 1:

- **Case 1:** C[i] = j
  - **Case 1.1:** A decrement of 1 is necessary if the swapped candies are different, and so we claim that OPT[i][j] = OPT[i-1][j'] - 1 is our optimal solution. Suppose OPT[i-1][j'] is not an optimal solution for the subproblem for i-1 and j. Then there exists OPT'[i-1][j'] > OPT[i-1][j'] that is optimal. This implies that OPT'[i-1][j'] - 1 > OPT[i-1][j'] - 1. This contradicts the optimality of OPT[i][j].
  - **Case 1.2:** An increment of 1 is necessary, and ideal (as mentioned above), if the swapped candies are the same, and so we claim that OPT[i][j] = OPT[i-1][j'] + 1 is our optimal solution. Suppose OPT[i-1][j'] is not an optimal solution for the subproblem for i-1 and j. Then there exists OPT'[i-1][j'] > OPT[i-1][j'] that is optimal. This implies that OPT'[i-1][j'] + 1 > OPT[i-1][j'] + 1. This contradicts the optimality of OPT[i][j].
- **Case 2:** C[i] != j:
  - In this case, the correctness of OPT[i][j] depends on the correctness of OPT[i-1][j] since no swapping takes place, and thus there is no change in the score. Suppose OPT[i-1][j] is not an optimal solution for the subproblem for

i-1 and j. Then there exists OPT'[i-1][j] > OPT[i-1][j] that is optimal. This contradicts the optimality of OPT[i][j].

Algorithm

```
OPT[n][3]: 2D Array, all elements initialised to 0
MaximiseScore(n, j):
  // initialise the base cases
  if C[1] == 1:
    OPT[1][1] = 1
    OPT[1][2] = -inf
    OPT[1][3] = -inf
  else if C[1] == 2:
    OPT[1][1] = 0
    OPT[1][2] = -1
    OPT[1][3] = -inf
  else if C[1] == 3:
    OPT[1][1] = 0
    OPT[1][2] = -inf
    OPT[1][3] = -1
  end if

  for i = 1 to n:
    for j = 1 to 3:
      if C[i] == j:
        OPT[i][j] = -inf
        for j' = 1 to 3:
          if j == j':
            OPT[i][j] = max(OPT[i][j], OPT[i-1][j'] + 1)
          else:
            OPT[i][j] = max(OPT[i][j], OPT[i-1][j'] - 1)
          end if
        end for
      else:
        OPT[i][j] = OPT[i-1][j]
      end if
    end for
  end for

  return max(OPT[n][1], OPT[n][2], OPT[n][3])
```

Time Complexity

The time complexity of the algorithm is *O(n)*.

Problem 2

Subproblems

For the sake of simplicity, let us sort the array X. Let for all 1 <= i <= n, 1<= j <= k, OPT[i][j] denotes the optimal cost of building j bakeries in houses X[1...i]. OPT[n][k] would give us the final solution to the problem.

Recurrence

1. **Base Case:** for j = 1, the bakery will be placed in the median of the houses X[1...i]. Thus,

$$OPT[i][1] = \sum_{1 \le h \le i} (\,|X[h] - X[floor((i+1)/2)]|\,) \;\forall\; 1 \le i \le n$$

2. For all $1 \le i \le n,\ 1 < j < k$:

$$OPT[i][j] = min_{1 \le h \le i}(OPT[h][j-1] + \sum_{h < p \le i} (\,|X[p] - X[floor((i+p)/2)]|\,))$$

Correctness

Case 1(Base Case): If there is only 1 bakery to serve houses on coordinates between i and j, then the optimal placement of the bakery is at the median house between the houses i and j. This is equivalent to proving that the median of points in set S minimizes the sum of absolute deviations in set S w.r.t median.

A simple proof goes along the lines of taking two points $a,\ b\ (a \le b)$, and a third point x with which the sum of their absolute deviations need to be taken: $|a - x| + |b - x|$, and showing that this sum is minimal if $a \le x \le b$. This can then be generalized for all points in the interval $[i, j]$, by going inwards from each end, one-by-one, taking pairwise the points (i and j, $i+1$ and $j-1$, ...) and showing that for each pair of points, the median x minimizes the sum of absolute deviation. Further proofs can be seen here.

Case 2: If there are $j > 1$ bakeries available, we consider serving:

- the first $h\ (\forall\ h,\ i \le h \le j)$ houses, $X[i..j]$, with $j-1$ bakeries;
- the last houses $X[h+1..j]$ with the j th bakery (Case 1 is used in this case to calculate distance since the number of bakeries is 1).

...and adding up their respective distances.

Suppose for the subarray X[i..j] the optimal solution OPT[i][j] is given by index h = p. Meaning, the last few houses X[p+1...j] are served optimally by the jth bakery. Thus the remaining houses X[i...p] will be served optimally by j-1 remaining bakeries. Thus, we get:

OPT[i][j] = OPT[p][j-1] + Σ. (where $\Sigma = \sum_{p < p' \leq i} (|X[p'] - X[floor((i+p')/2)]|)$)

Suppose we assume that OPT[p][j-1] is not the optimal solution for the subproblem with houses X[1...p]. Thus there exists some other value OPT'[p][j-1] which is optimal for the subproblem X[1...p], i.e. OPT'[p][j-1] < OPT[p][j-1]. But this means, OPT'[p][j-1] + Σ < OPT[p][j-1] + Σ. This contradicts the optimality of OPT[i][j].

Also to note is that if there are k bakeries, n houses and $k \geq n$, then the optimal distance is 0 since we can assign a bakery to every house, and no one has to walk any distance.

The optimal/minimal sum of distances of our subproblem(i,j) is produced by the h that gave the least sum of distances.

Algorithm

```
OPT[n][k]: 2D Array, all elements initialised to 0
MinimiseTotalDistance(X[1 ... n], k):
  sort(X) // sort array X in O(nlogn) time
  for i = 1 to n:
    OPT[i][1] = sumOfDistances(1, i)
  end for

  if k ≥ n:
     return 0
  end if

  for i = 1 to n:
    for j = 2 to k:
      OPT[i][j] = inf
      for h = 1 to i:
        OPT[i][j] = min(OPT[i][j], OPT[h][j-1] + sumOfDistances(h+1, i))
      end for
    end for
  end for

  return OPT[n][k]

sumOfDistances(i, j):
  summ = 0
  for h = i to j:
    summ += abs(X[h] - X[floor((i+j)/2)])
  end for
  return summ
```

Time Complexity

The worst-case runtime for the outermost for loop is *O(n)*, the middle loop is *O(k)*, the innermost loop is *O(n)*, and for the *sumOfDistances* subroutine in *O(n).* Thus the total time complexity of the algorithm is $O(kn^3)$.

Problem 3

Subproblems

Let for all 0 <= i <= j < n, OPT[i][j] denote the maximum energy gained for the subarray V[i...j]. Note that for i = j, OPT[i][i] represents an array with a single element, for which the energy gained is simply 0. OPT[0][n-1] would give us the final solution to the problem.

Recurrence

1. **Base Case:** The array contains a single element. Thus,
   $$OPT[i][i] = 0 \ for \ 0 \leq i < n$$
2. For all $0 \leq i < j < n$:
   $$OPT[i][j] = sum^2(V[i...h_{max}]) + sum^2(V[h_{max}+1...j]) + max_{i \leq h < j}(OPT[i][h] + OPT[h+1][j])$$

Where sum(V[i...j]) denotes the sum of all elements of V from index i to j (both inclusive).

Correctness

Case 1(Base Case): The energy gained by a single element is simply 0.
Case 2: Suppose for the subarray V[i...j] the optimal solution OPT[i][j] is given by the index h = p. Then OPT[i][p] is the optimal solution for the subarray V[i...p] and OPT[p+1][j] is the optimal solution for the subarray V[p+1...j]. Thus we get:

$$OPT[i][j] = sum^2(V[i...p]) + sum^2(V[p+1...j]) + (OPT[i][p] + OPT[p+1][j])$$

Suppose OPT[i][p] is not optimal and instead OPT'[i][p] is optimal for the subarray V[i...p], i.e OPT'[i][p] > OPT[i][p]. This means

$$sum^2(V[i...p]) + sum^2(V[p+1...j]) + (OPT'[i][p] + OPT[p+1][j]) > sum^2(V[i...p]) + sum^2(V[p+1...j]) + (OPT[i][p] + OPT[p+1][j])$$

Which contradicts the optimality of OPT[i][j].

Similarly, suppose OPT[p+1][j] is not optimal and instead OPT'[p+1][j] is optimal for the subarray V[p+1...j], i.e OPT'[p+1][j] > OPT[p+1][j]. This means

$$sum^2(V[i...p]) + sum^2(V[p+1...j]) + (OPT[i][p] + OPT'[p+1][j]) > sum^2(V[i...p]) + sum^2(V[p+1...j]) + (OPT[i][p] + OPT[p+1][j])$$

Which contradicts the optimality of OPT[i][j].

Algorithm

```
OPT[n][n]: 2D Array, all elements initialised to 0
MaximiseEnergy(V[0 ... n-1]):
  if n = 0:
    return 0
  end if

  for l = 1 to n-1:
    for i = 0 to n-l:
      j = i + l
      OPT[i][j] = -inf
      hMax = -1
      for h = i to j-1:
        if OPT[i][j] < OPT[i][h] + OPT[h+1][j]:
          hMax = h
          OPT[i][j] = OPT[i][h] + OPT[h+1][j]
        end if
      end for

      leftSum = sum(V[i .. hMax])^2
      rightSum = sum(V[hMax + 1 ... j])^2

      OPT[i][j] += leftSum + rightSum
    end for
  end for

  return OPT[0][n-1]
```

Time Complexity

The time complexity of each for loop above is $O(n)$ in the worst case, thus the total time complexity of the algorithm is $O(n^3)$.