CSE 222 (ADA) Homework Assignment 2 (Theory)

Deadline : Feb 19 (Friday) 10am.

The theory assignment has to be done in a team of at most two members, as already selected by you. The solutions are to be typed either as a word document or latex-ed and uploaded as pdf on GC. We shall strictly not accept solutions written in any other form. Remember that both team members need to upload the HW solution on GC. Collaboration across teams or seeking help from any sources other than the lectures, notes and texts mentioned on the homepage will be considered an act of plagiarism

General instructions:

Each question is worth 15 points. For each question we need you to provide the following:

1. (3 points) A precise description of the subproblems you want to solve in the dynamic program. *Notice this is just the subproblem, not how to solve it, or how it was obtained.*
2. (3 points) A recurrence which relates a subproblem to "smaller" (Whatever you define "smaller" to mean) subproblems. *Notice this is just the recurrence, not the algorithm or why the recurrence is correct.*
3. (1 points) Identify the subproblem that solves the original problem *Often but NOT always, this is the answer of the"largest" subproblem.*
4. (3 points) A clear proof for why the recurrence is correct. Specifically, prove that an optimal solution for your subproblem can be obtained using optimal solutions for "smaller" subproblems via your recurrence. *Notice this is the proof of why the recurrence is correct. Often this involves contradiction arguments involving the optimal solution. This is NOT(!) to prove correctness of the final algorithm.*
5. (3 points) A pseudo-code for a dynamic program which solves the recurrence efficiently *You do not need to prove correctness of the pseudocode itself*
6. (2 points) An argument for the running time of your dynamic program

We are only interested in the *value* of the optimal solution, not the optimal solution itself. So you do not need to give the reconstruction algorithm.

If your solution does not have the structure above, you will be awarded *a zero* (yes, you read that right). There will be *no* concessions on this.

The first question below is from the text by Jeff Erickson. You can find an online copy here.

Problem 1. Solve Question 14 from the above text. **Solution.**

Subproblem. Let the candy types be $1, 2, 3$ and $a_i \in \{1, 2, 3\}$ denote the candy of animal i, where 1 is the circus peanut. Let us call a sequence of decisions for any input, of whether to swap the candy or not as a *swap sequence*. Then, we have the subproblem for $1 \leq j \leq 3, 1 \leq i \leq n+1$:

$OPT(i, j)$: maximum score of a swap sequence for animals $i, i+1, \ldots, n$ assuming you have candy j in hand when comparing with animal i

Recurrence.

$$OPT(i, j) = \begin{cases} 1 + OPT(i+1, j) & \text{if } a_i = j \\ \max(-1 + OPT(i+1, a_i), OPT(i+1, j)) & \text{otherwise} \end{cases}$$

Base case: $OPT(n+1, j) = 0$ for $j = 1, 2, 3$

Solution to original problem.

$$OPT(1, 1)$$

Correctness of recurrence.

Consider the optimal solution for a subproblem for $OPT(i, j)$. Then, there are two possibilities for whether $OPT(i, j)$ swaps the candy in hand with that of animal i:

1. $OPT(i, j)$ *does not swap the candy:* In this case, it gets a score of 0 when comparing with animal i. When comparing with animal $i+1$, it has candy j in hand. Now, the swap sequence of $OPT(i, j)$ from animal $i+1$ onwards must be $OPT(i+1, j)$. Why? Suppose for the sake of contradiction that it is not. Then, we can just replace the swap sequence of $OPT(i, j)$ from $i+1$ onwards, with $OPT(i+1, j)$ (because $OPT(i+1, j)$ is always a valid swap sequence from $i+1$ onwards if we had candy j in hand when comparing with animal $i+1$). This will increase the total score, which contradicts the optimality of $OPT(i, j)$. Hence, in this case, $OPT(i, j)$ is equal to $OPT(i+1, j)$.
2. $OPT(i, j)$ *swaps the candy:* In this case, it gets a score of 1 if $a_i = j$ and -1 otherwise. By a similar argument as above, the swap sequence of $OPT(i, j)$ from $i+1$ onwards must be $OPT(i+1, a_j)$. Hence, in this case, $OPT(i, j)$ is equal to $OPT(i+1, a_j) + 1$ if $a_i = j$ and $OPT(i+1, a_j) - 1$ if $a_i \neq j$..

The optimal solution must be the best of all these possibilities. Hence, if $a_i = j$, it must be $1 + OPT(i+1, j)$. And if $a_i \neq j$, it must be the larger of $-1 + OPT(i+1, a_i)$ and $OPT(i+1, j)$. This proves the recurrence.

Pseudocode. Read algorithm 1. Here, an array A stores the candies of the animals, $A[i] \in \{1, 2, 3\}$ is the candy of animal i and the circus peanut candy is denoted by 1.

Time Complexity.

There are $3n$ subproblems, each of which takes $O(1)$ time to solve. Hence, the time complexity is $O(n)$. Another way to see this is that the outer loop has n iterations and the inner loop has 3 iterations.

Algorithm 1 Pseudo-code for candy swap

```
procedure CANDYSWAP(A)
    Construct a 2-D table D[i][j] of size n + 1 × 3. Initialize D[n + 1][j] = 0 for all j.
    for all i from n to 1 do
        for all j ∈ {1, 2, 3} do
            if A[i] = j then
                D[i][j] ← 1 + D[i + 1][j]
            else
                D[i][j] ← max(−1 + D[i + 1, A[i]], D[i + 1, j])
            end if
        end for
    end for
    return D[1][1].
end procedure
```

Problem 2.

Solution.

Subproblem. for each $i \geq 1$

$P(i) =$ max score from songs $i, i+1, \ldots, n$ so that you don't dance to $wait[i]$ songs after dancing to song i

Recurrence.

$$P(i) = \max\left(P(i+1), score[i] + P(i + wait[i] + 1)\right) \tag{1}$$

with base case $P(i) = 0$ if $i > n$.

Subproblem solving original problem $P(1)$

Correctness of recurrence.

Enough to prove two claims

Claim 1 *The RHS of* (1) *is the value of a feasible solution for* $P(i)$.

Proof:

1. Any optimal feasible solution of value $P(i+1)$ for the songs $i+1, i+2, \ldots, n$ is clearly a feasible solution for $P(i)$ as $i+1, i+2, \ldots, n$ is a subset of the songs $i, i+1, i+2, \ldots, n$
2. if S is an optimal feasible solution for $P(i + wait[i] + 1)$, then $i \cup S$ is feasible for $P(i)$. This is because the wait between i and the next song in $i \cup S$ is at least $wait[i]$ (since $P(i + wait[i] + 1)$ only has songs from $i + wait[i] + 1$ onwards), and S is feasible for $i + wait[i] + 1$ onwards. Also, the total score of $i \cup S$ is $P(i + wait[i] + 1) + score[i]$.

□

Claim 2 *The score of any feasible solution for* $P(i)$ *is at most the RHS of* (1).

Proof: Consider any feasible solution S for $P(i)$. Then, there are only 2 possibilities:

1. $i \in S$: Then, the next song in S after i must have index atleast $i + wait[i] + 1$. Hence, $S \setminus i$ is feasible for $P(i + wait[i] + 1)$. Therefore, the total score of $S \setminus i$ is at most $P(i + wait[i] + 1)$. This gives that the total score of S is at most $score[i] + P(i + wait[i] + 1)$, which is at most the RHS of (1).
2. $i \notin S$: Then, $S \subset \{i + 1, i + 2, \ldots, n\}$. Hence, the total score of S is at most $P(i + 1)$, which is at most the RHS of (1).

□

Pseudocode

See algorithm 2.

Algorithm 2 Algorithm for song and dance

```
procedure DANCE(score,wait)                    ▷ score and wait are arrays of length n
    D ← new array of length n + 1
    D[i] = 0 for i > n
    for all i ∈ n down to 1 do
        D[i] ← max(D[i + 1], D[min(i + 1 + wait[i], n + 1)] + score[i])
    end for
    return D[1]
end procedure
```

Time complexity There are n subproblems for which the recurrence needs to be solved, each takes $O(1)$ time since its a max over two cases. Hence the time complexity is $O(n)$.

Problem 3. Upon time traveling to an ancient laboratory, you find n drops of a *mystical* liquid arranged in an order. The *volume* of each drop is given to you. You find a scroll with the following instructions.

You can do the following operation as many times as you like until just one drop remains:

Combine any two *adjacent* drops and place the combined drop at the same place in the order

When you combine two drops of volumes x and y respectively, you get a new drop whose volume is the sum $x + y$ of the two original volumes (i.e. volume is conserved). Also, you gain an amount of energy for your time traveling device equal to $x^2 + y^2$.

For example,

1. Start with volumes
   $$1 \quad 4 \quad 2 \quad 3 \quad 5$$
2. Combine the second and the third drop to get volumes
   $$1 \quad 6 \quad 3 \quad 5$$
   and gain energy $4^2 + 2^2 = 20$
3. Combine the third and the fourth drop to get volumes
   $$1 \quad 6 \quad 8$$
   and gain energy $3^2 + 5^2 = 34$

4. Combine the first and the second drop to get volumes

$$7 \quad 8$$

and gain energy $1^2 + 6^2 = 37$

5. Combine the remaining two drops to get volumes

$$15$$

and gain energy $7^2 + 8^2 = 113$

Hence, you gain a total energy of $20 + 34 + 37 + 113 = 204$.

Eager to time travel back to 2021, you'd like to maximize the energy gained for your time traveling device. Give an efficient algorithm to find the sequence of operations to achieve this.

Solution

Subproblems. Let, for all $i, j = 1, 2, \cdots n$ and $i \leq j$, $\text{ENERGY}(i, j)$ denote the optimal solution for the problem defined with liquids $v_i, v_{i+1}, \cdots v_j$.

Recurrence.

1. *Base cases:* For all $i = 1, 2, \cdots n$, $\text{ENERGY}(i, i) = v_i$
2. For all $1 \leq i \leq n, i < j \leq n$

$$\text{ENERGY}(i, j) = \max_{i \leq k < j} \{\text{ENERGY}(i, k) + \text{ENERGY}(k+1, j) + \text{SUM}(v_i, v_{i+1}, \cdots v_k)^2 + \text{SUM}(v_{k+1}, v_{k+2}, \cdots v_j)^2\}$$

Final Solution. We want to compute $\text{ENERGY}(1, n)$.

Correctness of Recurrence. We first observe that in the optimal solution of $\text{ENERGY}(i, j)$ there are $j - i$ possibilities to choose the last operation i.e. the last operation could be to combine the mixture of liquid drops $v_i, v_{i+1}, \cdots, v_k$ with the mixture of the drops $v_{k+1}, v_{k+2}, \cdots, v_j$, where k lies in $i, i+1, \cdots, j-1$.

Suppose in the optimal solution of $\text{ENERGY}(i, j)$, the last operation is to combine the mixture of drops $v_i, v_{i+1}, \cdots, v_x$ with the mixture of drops $v_x, v_{x+1}, \cdots, v_j$. We claim that in the optimal solution, the mixture of drops $v_i, v_{i+1}, \ldots, v_x$ was obtained

$$\text{ENERGY}(i, x) + \text{ENERGY}(x+1, j) = \text{ENERGY}(i, j) - \left(\text{SUM}(v_i, v_{i+1}, \cdots v_k)^2 + \text{SUM}(v_{k+1}, v_{k+2}, \cdots v_j)^2\right)$$

For the sake of contradiction, suppose this is not true i.e. $\text{ENERGY}(i, x)$ and $\text{ENERGY}(x+1, j)$ are not. But then we can create another feasible solution for liquid $v_i, v_{i+1} \cdots, v_j$ by taking this solution and then mixing both the segment. Clearly this is a solution to our sub-problem with liquid $v_i, v_{i+1}, \cdots, v_j$ which has a higher energy than $\text{ENERGY}(i, j)$ contradicting the optimality of the latter.

Pseudocode. (On last page)

Runtime. The running time is clearly dominated by the nested for-loops. Hence it in $\mathcal{O}(n^3)$.

Algorithm 3 Energy

```
1: procedure ENERGY((1, n))
2:     ENERGY : 2-D Array of size (n) × (n)
3:     for i = 1 to n do
4:         ENERGY[i][i] = v_i
5:     end for
6:     for l = 1 to n − 1 do
7:         for i = 1 to n − l do
8:             j = i + l
9:             for k = i to j − 1 do
10:                ENERGY[i][j] = max{ENERGY[i][k] + ENERGY[k + 1][j] + (SUM(v_i, v_{i+1}, ··· v_k)^2 + SUM(v_{k+1}, v_{k+2}, ··· v_j)^2), ENERGY[i][j]}
11:            end for
12:        end for
13:    end for
14:    Return ENERGY[1][n]
15: end procedure
```

Old Problem 2. In the city of Sidhapur, there is a very long straight street on which n people live in houses built next to each other (not necessarily at the same wait) - house i is identified by a coordinate x_i. The citizens currently need to get their bread from the neighboring town of RotiGarh, which they find annoying. Hence the local councillor decides to open k bakeries on the long street. The residents are happy to lease out one floor of their houses for opening a bakery. Your task would be to pick k out of the n houses. Remember that each citizen needs to travel everyday to their closest bakery. Hence, you better come up with a solution such that the total distance travelled by all citizens each day is as small as possible. Design a polynomial (in n and k) time algorithm to pick the k spots.

Solution

Subproblems. Let, for all $i = 0, 1, 2, \cdots n$ and $k' = 0, 1, 2 \cdots k$,
$bake(i, k')$ denote the optimal solution for the bakery problem when the last (or rightmost) bakery (i.e. k') is placed at location i and we use exactly k' bakeries.

Recurrence.
Let l_i = sum of distances of the coordinates x_j from x_i (such that $x_j < x_i$)
Let r_i = sum of distances of the coordinates x_j from x_i (such that $x_j > x_i$)
Let $b_{ii'}$ = sum of distances of all coordinates x_j (such that $x_{i'} \leq x_j \leq x_i$) from their closest bakery, when $x_{i'}, x_i$ are the only two bakeries for the range $[x_{i'}...x_i]$. So, the closest bakery will be either $x_{i'}$ or x_i

1. *Base cases:*

   For k' = 1: bake(i,k') = $l_i + r_i$

   For all i<k': bake(i,k') = ∞

2. For all $1 \leq i \leq n, 2 \leq k' \leq k$,

$$bake(i,k') = \min\{bake(i',k'-1) - r_{i'} + r_i + b_{ii'}\}\ \{where, k'-1 \leq i' \leq i-1\}$$

Final Solution. We want to compute $\min\{bake(i,k)\}$ $where, 1 \leq i \leq n$.

Correctness of Recurrence.

1. Case I (k' = 1 and $1 \leq i \leq n$): In this case, we have to place only one bakery and that is at i, so bake(i,k') = sum of distances of all coordinates from this bakery (as this is the only bakery and thus the closest one) = sum of distances of all $x_j < x_i$ from x_i + sum of distances of all $x_j > x_i$ from $x_i = l_i + r_i$

2. Case II ($i < k'$): In this case, as we have already placed k' - 1 bakeries in i' - 1 positions and $i - 1 < k - 1$ (because $i < k$) which implies we have placed two bakeries at the same location, which makes no sense. Thus, this case should be discarded and bake(i,k') = ∞ is justified.

3. Case III ($k' > 1$ and $x_j >= x_i$):
   In the optimal solution of bake(i,k')(say it is S') there are i - k' + 1 possible answers(as the k' - 1 would be at one of the positions between k'-1 to i-1) when we are placing the last bakery(k') at i.

   Suppose in the optimal solution of bake(i,k'), we have placed k' - 1 bakery at i' (where $k' - 1 \leq i' \leq i - 1$).

   We claim that: bake(i',k'-1) = $bake(i,k') - r_i - bii' + r_{i'}$

   By Contradiction, let's say it is not true: then it means we have better optimal solution for bake(i',k'-1) (i.e. $bake(i',k'-1) = S'' < S' - r_i - bii' + r_{i'}$).....(1)

   But if this is the case then we could have used S'' for constructing a better solution for bake(i,k') as follows: Since, we know the better optimal solution for bake(i',k'-1)(which is S'') and we have to place the k' bakery at i, so for that we have to recompute the cost of only houses that are between x_i' and x_i and those that are to the right of x_i. The second category will all go to x_i now while the first category will go to x_i' or x_i depending on whether they lie 'closer' to x_i' or x_i. So, the new solution for bake(i,k') will be S'' $+ r_i + bii' - r_{i'}$. But if this is the case then we know $S'' + r_i + bii' - r_{i'} < S'$(from (1)) which means we got more optimal solution for bake(i,k') which contradicts our assumption that S' is the optimal solution for bake(i,k').

Pseudocode. (On last page)

Runtime. The running time is clearly dominated by the nested for-loops and if l,r,b are not precomputed and dynamically computed then time complexity will be O(n^3k).

Algorithm 4 Bakery

procedure BAKERY((n, k))
  $bake$: 2-D Array of size $(n) \times (k)$
  for $i = 1$ to n **do**
    $bake[i][1] = l[i] + r[i]$ ▷ Array l, r, b are trivial to compute
  end for
  for $i = 1$ to n **do**
    for $k' = 1$ to k **do**
      if $(i < k')$ **then**
        $bake[i][k'] = \infty$
      else
        $temp = \infty$
        for $i' = k' - 1$ to $i - 1$ **do**
          $temp = \min\{temp, bake[i'][k' - 1] - r[i'] + r[i] + b[i][i']\}$
        end for
        $bake[i][k'] = temp$
      end if
    end for
  end for
  $ans \leftarrow \infty$
  for $i = 1$ to n **do**
    $ans = \min\{ans, bake[i][k]\}$
  end for
  Return ans
end procedure