CS218 Assignment 2

190050004 - Adarsh Raj
190050041 - Gudipaty Aniket
190050102 - Sahasra Ranjan
190050104 - Sambit Behera

May 11, 2021

Exercise 1

Looking at the time complexity, one can easily infer that we should try to decompose the given problem into finding 6 products dealing with numbers of size $\frac{n}{3}$, and combining them to get the final result. So, we use the following divide and conquer approach :

- Let the number of digits n be a multiple of 3. Then, we can divide the decimal representation of a into 3 equal parts of sizes $\frac{n}{3}$ each (If n is not a multiple of 3, we can make the necessary change by taking first 2 parts from the left (a_0 and a_1) to be of $\lfloor \frac{n}{3} \rfloor$ digits each and remaining digits to be in a_2, which will all again be approximately $\frac{n}{3}$). In other words, we can write :

$$a = a_0 + a_1 10^{\frac{n}{3}} + a_2 10^{\frac{2n}{3}}, \text{ where } a_0, a_1, \text{ and } a_2 \text{ are } \frac{n}{3} \text{ digit numbers.}$$

- Now, we can write a^2 as :

$$a^2 = a_0^2 + (2a_0a_1)10^{\frac{n}{3}} + (a_1^2 + 2a_0a_2)10^{\frac{2n}{3}} + (2a_1a_2)10^n + a_2^2 10^{\frac{4n}{3}}$$

$$\implies a^2 = a_0^2 + ((a_0+a_1)^2 - a_0^2 - a_1^2)10^{\frac{n}{3}} + (a_1^2 + (a_0+a_2)^2 - a_0^2 - a_2^2)10^{\frac{2n}{3}} + ((a_1+a_2)^2 - a_1^2 - a_2^2)10^n + a_2^2 10^{\frac{4n}{3}}$$

- So, to find a^2, we need the squares of 6 numbers with around $\frac{n}{3}$ digits :
  $a_0^2, \quad a_1^2, \quad a_2^2, \quad (a_0 + a_1)^2, \quad (a_0 + a_2)^2, \quad (a_1 + a_2)^2$
- Hence, we can find these smaller squares and combine them as shown above (the combining step takes $O(n)$ time, assuming that adding 2 or more numbers takes $O(n)$ time) to get a^2.
- The time complexity analysis will be :

$$T(n) = 6T\left(\frac{n}{3}\right) + O(n)$$

  Comparing this with the general expression :

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^c)$$

  Using the master's theorem, we have $a = 6, b = 3, c = 1$. So, $a > b^c$. Hence, the time complexity will be $\Theta(n^{log_b a}) = \Theta(n^{log_3 6})$

  We can also get the time complexity analytically, by setting $n = 3^k$ and $T(3^r) = F(r)$. So, we get :

$$F(k) = 6F(k-1) + 3^k = 6^2F(k-2) + 6^1 3^{k-1} + 3^k$$

  Expanding the recursion, we get :

$$F(k) = 6^k + 6^{k-1}3^1 + 6^{k-2}3^2 + \ldots + 6^1 3^{k-1} + 3^k + O(1)$$

Now, using the expression for the sum of a geometric progression and using the fact that $a6^k + b3^k \in \Theta(6^k)$ (a and b are constants), we get:

$$T(n) = T(3^k) = F(k) \in \Theta(6^k) = \Theta(6^{log_3 n}) = \Theta(n^{log_3 6})$$

Following is a more structured and formal definition of the algorithm :

1. We need to find $Square(a)$, which returns the square of an n digit number a.
2. Divide a into 3 parts of $\frac{n}{3}$ digits each and express it as (If n is not a multiple of 3, we can make the necessary change by taking first 2 parts from the left (a_0 and a_1) to be of $\lfloor \frac{n}{3} \rfloor$ digits each and remaining digits to be in a_2, which will all again be approximately $\frac{n}{3}$):

$$a = a_0 + a_1 10^{\frac{n}{3}} + a_2 10^{\frac{2n}{3}}, \text{ where } a_0, a_1, \text{ and } a_2 \text{ are } \frac{n}{3} \text{ digit numbers.}$$

3. Find $Square(a_0)$, $Square(a_1)$, $Square(a_2)$, $Square(a_0+a_1)$, $Square(a_1+a_2)$ and $Square(a_2 + a_0)$ recursively. Store them in variables s_0, s_1, s_2, s_{01}, s_{12} and s_{20} respectively.
4. Define the variable $result$ corresponding to a^2 as:

$$result = s_0 + (s_{01} - s_0 - s_1)10^{\frac{n}{3}} + (s_1 + s_{20} - s_0 - s_2)10^{\frac{2n}{3}} + (s_{12} - s_1 - s_2)10^n + s_2 10^{\frac{4n}{3}}$$

5. Return the value of $result$.

Correctness of Algorithm

The correctness of this algorithm is very trivial, which can be seen in the derivation of the algorithm itself. We divide the given number a into a_0, a_1, and a_2, which have $\frac{n}{3}$ digits each, and hence we can also say that $(a_0 + a_1)$, $(a_1 + a_2)$, and $(a_2 + a_0)$ also have approximately $\frac{n}{3}$ digits. So, we calculate their squares first, and express a^2 as their linear combination, as derived before:

$$a^2 = a_0^2 + ((a_0 + a_1)^2 - a_0^2 - a_1^2)10^{\frac{n}{3}} + (a_1^2 + (a_0 + a_2)^2 - a_0^2 - a_2^2)10^{\frac{2n}{3}} + ((a_1 + a_2)^2 - a_1^2 - a_2^2)10^n + a_2^2 10^{\frac{4n}{3}}$$

The correctness of the above expression is proved by its derivation itself, which has been done before.
We can reuse the values of smaller squares which repeatedly occur in the above expression after computing them only once.

Combining them according to the given linear combination takes $O(n)$ time, assuming that adding 2 numbers takes $O(1)$ time.

Exercise 2

(a) Divide and Conquer algorithm for WhoTargetsWhom

We have the problem given to compute the left and right targets for each hero. An array $Ht[1 \dots n]$ is given with height of each of the heroes. So our problem is to find closest left and right hero who is taller than the hero.

We can take a Divide and Conquer approach for this problem. We can divide the Ht array in two halves and calculate these values:

- $L[i]$ which stores the left target for i-th hero. It is $-\infty$ if no one is taller than the hero on its left
- $R[i]$ which stores the right target for i-th hero. It is ∞ if no one is taller than the hero on its right

Suppose we know these values for Left and Right sub-arrays. We will now try to compute these values for the merged (conquer) array Ht.

Lemma 2.1 — *In the left(right) array, those element i whose $R[i] = \infty$ ($L[i] = -\infty$) are in decreasing(increasing) order of $Ht[i]$.*

We will prove this lemma for the left sub-array and similarly we can show for the right sub-array.

Proof. If for an element i, $R[i] = \infty$, this mean there is no other element j ($R[j] = \infty$) and $j > i$ in that array which has larger height than the i_{th} element (since i_{th} hero cannot hit anyone else).
So we have $\forall i, j(((j > i) \wedge (R[i] = \infty) \wedge (R[j] = \infty)) \implies (Ht[i] > Ht[j]))$ □

Now we will construct an algorithm to merge these arrays. A few trivial observations before we state the algorithm:

- In left sub-array, all the elements left of the largest element will have no effect on merge. Because the L array will be same and R values cannot be an element on the right of the largest element.
- In right sub-array, all the elements right of the largest element will have no effect on merge. Because the R array will be same and L values cannot be an element on the left of the largest element.
- We have to change $R[i]$ only for those elements in the left array who have $R[i] = \infty$. Other will remain same (closest taller).
- We have to change $L[i]$ only for those elements in the right array who have $L[i] = -\infty$. Other will remain same (closest taller).
- These (last two points) values are already in decreasing (increasing) order in left (right) sub-array.

Overview of the algorithm:

- We will check the largest element in each side and will assign the R(or L) value according to their height.

- We will keep two pointers one at each left and right sub-array (starting from largest element in each)
- We will iterate all those values which we have to compute (described earlier) and check if it is smaller than the the element in the other sub-array. If yes, it mean it can shoot the element in the other sub-array
- Since both the sub-arrays have the element (we need to evaluate) in monotonic fashion, we will be able to compute the L, R values for each of them.

First of all, we will introduce two functions which will return the next element in each sub-array to be evaluated. Since we argued we will check elements with targets as ∞ and $-\infty$ for left and right sub-array respectively.

Algorithm 1: Function findLeft(i,L,n)

Result: Next element to compute in the left sub-array
for *j = i+1; j<n* **do**
  if *R[j] is* ∞ **then**
    return *j*
  j $\longleftarrow$ *j+1*
end
return n

Algorithm 2: Function findRight(i,R,n)

Result: Next element to compute in the right sub-array
for *j = i-1; j>=n* **do**
  if *R[j] is* $-\infty$ **then**
    return *j*
  j $\longleftarrow$ *j-1*
end
return n

Correctness of the algorithm

This is a standard divide and conquer algorithm, and our choice of sub-arrays are left half and right half. We have to show the correctness of the conquer step.

- As already mentioned we need to compute the left and right targets of those which are to the right(left) of largest element in the left(right) sub-array. As shown above the targets of the other values won't change in the merge step.
- For the left sub-array, we have to find the the right target of an element i (say). Now $L[i]$ won't change. But for $R[i]$ we'll have to check the right sub-array.
- $R[i]$ will be one of those elements in the right sub array which has $L[i] = -\infty$ if not, then we will get an another element j in the right sub-array for which $Ht[L[j]] > Ht[j]$ and $L[j] < j$. But in this case the target for i will $L[j]$ and not j. So our statement about searching only those element in right sub-array whose left target is $-\infty$ is correct.
- Now we have to show that our choice of target for an element i is correct. For this we can say we will have to take the first element which is larger that $Ht[i]$. And since the height

is in increasing order in the right sub-array (argued above). Our algorithm (two-pointer method to check the values) will chose the right target for each of the element.

- Similarly we can show for the right sub-array

So correctness of each of the step is algorithm is argued.

Algorithm 3: Merge the sub-arrays and compute L, R values

Result: Merged array L, R, and position of the largest element
Ht ⟵ Left sub-array with the height of our heroes
L ⟵ left target of our heroes
R ⟵ right target of our heroes
i ⟵ position of the largest element in left sub-array
j ⟵ position of the largest element in right sub-array
a ⟵ ∞(Store the value which is just largest in the right sub-array and L value −∞)
b ⟵ −∞(Store the value which is just largest in the left sub-array and R value ∞)

```
if Ht[i]>Ht[j] then
    maxElem ⟵ i
else
    maxElem ⟵ j
while i<j do
    if Ht[i]>Ht[j] then
        R[i] ⟵ a
        i ⟵ findLeft(i,L,n/2)
        b ⟵ i
    else if Ht[i]>H[j] then
        L[j] ⟵ b
        j ⟵ findRight(j,R,n/2)
        a ⟵ j
end
return L, R, maxElem
```

Time complexity analysis: Our algorithm is a standard Divide and Conquer type algorithm with the conquer step in $\mathcal{O}(n)$ time since we are using two pointer method to compute the values and going to each element at most once.

So time complexity for our recurrence algorithm will be:

$$T[n] = 2 \cdot T[n/2] + \mathcal{O}(n)$$

This is a standard recurrence relation and will evaluate to $\mathcal{O}(n \log n)$ using the master's theorem (a=2, b=2, k=1).

(b) The bound of $\lfloor n/2 \rfloor$

We have to show that at-least $\lfloor n/2 \rfloor$ of the n heroes are targeted.

Lemma 2.2 — *For each adjacent pairs, at-least one will hit the other.*

This lemma is trivial to prove since each of the heroes have different height, at least one of them will be smaller in height (and will hit the other)

Now, we can divide n heroes in $\lfloor n/2 \rfloor$ pairs and in each of the pairs, one of them is targeted. So in total, $\lfloor n/2 \rfloor$ of the n heroes are targeted. Hence, the bound of $\lfloor n/2 \rfloor$ is proved

(c) Algorithm to compute the no. of rounds

A simple observation is that, only those heroes will survive in a run if both the its left and right neighbours are taller than him. Otherwise one of them will kill the hero. So in each round all those heroes will die who are not smaller than both of his neighbour. And for the heroes at the first and last position we have to check his right and left neighbour respectively.

So we can compute who will survive in in $\mathcal{O}(n)$ time with a linear search.

Algorithm 4: Compute number of steps in the process before everyone but one dies

```
Result: Number of rounds before a single hero is left
h ← Array with the height of our heroes
cnt ← 0 (Number of rounds)
while More than one hero is alive do
    for each hero i (index) in H do
        if i==0 and h[0]>h[1] then
            Remove hero at 0th position
        else if i==n-1 and h[n-1]>h[n-2] then
            Remove hero at (n-1)th position
        else if h[i] > h[i+1] or h[i] > h[i-1] then
            Remove hero i from the array
    end
    cnt++
    // We have removed all the heroes who will die in this round
end
return cnt
```

Time Complexity analysis: Each round of the process takes $\mathcal{O}(n)$ time if size of the array is n. Also at each step we are removing those heroes who died. We proved in the (b) part that at-least $\lfloor n/2 \rfloor$ heroes will die at each step. So at each step our array size will reduce by at-least $\lfloor n/2 \rfloor$. So our time complexity will be:

$$T_n \leq \mathcal{O}(n) + T_{\lceil n/2 \rceil} \qquad T_1 = 1$$

From this relation, assuming the worst case scenario, we will get the relation:

$$T_n = \mathcal{O}(n) + T_{\lceil n/2 \rceil}$$

Using the master's theorem (a=1,b=2,k=1), this recurrence relation evaluates to $\mathcal{O}(n)$.

Correctness

This is a very trivial algorithm to find the those elements whose neighbours are shorter than them or not. And each iteration represents a round of the game. So, as described above, in each iteration (or game) we will remove all those elements which are taller than their neighbour. So the no of iterations will give us the number of rounds before only a single element is left. So our above mentioned algorithm is correct.

Exercise 3

Fill $3 \times 2n$ tile using 2×1 tiles

We have to fill a. $3 \times 2n$ tile using 2×1 tiles. We can using a standard Dynamic Programming approach for this problem.

Sub-problems

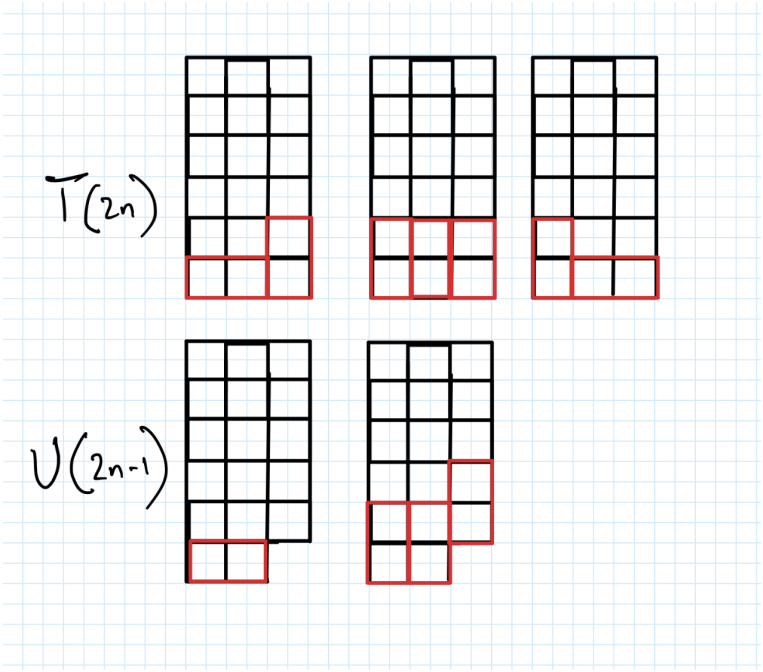

Figure 1: Sub-cases

We divide our subproblems into two parts, and can maintain two DP arrays, $T(2n)$ being the solution to the main problem and $U(2n-1)$ being the solution to the case when we have to tile a floor with one corner of the $3 \times 2n-1$ being absent. We can start tiling from the bottom left and form a recursion by dividing the problem into smaller problems.

Recursion

Base Case: $T(0) = 1$
In $3 \times 2n$ floor, if we place first tile horizontally, then the next tile is fixed and hence we get a $U(2n-1)$ subproblem. In 2nd case, if first and second tiles are vertical, then the problem reduces to $T(2n-2)$. And if first tile is vertical and second is horizontal, we have only one possibility, and the subproblem is the flipped version of $U(2n-1)$ which has the same value. Hence,

$$T(2n) = 2U(2n-1) + T(2n-2)$$

Similarly, for the second part of solving $U(2n-1)$, we take cases.
Base Case: $U(1) = 1$
If first tile is horizontal then problem reduces to $T(2n-2)$. If first tile is vertical then the placement of next 2 tiles is fixed and the problem reduces to $U(2n-3)$. Hence,

$$U(2n-1) = T(2n-2) + U(2n-3)$$

Correctness

Base cases for T and U are trivially true. For the recursive step, the image above describes all the possible case of generating a $3 \times 2n$ sized tile by adding 2×1 tiles to the smaller tiles. Any other combination of tiles will either get covered in $T(2n-2)$ or $U(2n-1)$. We already described above how we can represent $T(2n)$ as subproblems $T(2n-2)$ and $U(2n-1)$. Similarly for $U(n)$, any combination of tiling will be covered in $U(2n-3)$ or $T(2n-2)$.It is also easy to see that our decomposition of the given problems is into non-redundant subproblems. So our recursive step is correct for this problem. Now, this will terminate because we are recursively calling strictly smaller subproblems. So our algorithm to find number of 2×1 tiles is correct.

Memoization

Note that, as the T values are needed only for even numbers and U values for odd numbers we can store them in a single array of length 2n+1 at the corresponding parity positions and the value of each T or U can be calculated by using the values of previously computed subproblems T and U.

Sub-problem dependencies

By the recurrence relations it can be clearly stated that the value of T depends only on U and T values with smaller indices, and this holds true for U as well. Hence the recursion is acyclic and hence can be computed easily.

Time Complexity Analysis

For filling up the array of length 2n+1, we need to compute for each value of T and U. This can be done in constant time as the subproblem values are already stored.
Hence, time complexity of solution = $\mathcal{O}(1) \cdot \mathcal{O}(n) = \mathcal{O}(n)$

Fill $4 \times n$ tile using 2×1 tiles

We have to fill a. $4 \times n$ tile using 2×1 tiles. We can using a standard Dynamic Programming approach for this problem.

Sub-problems

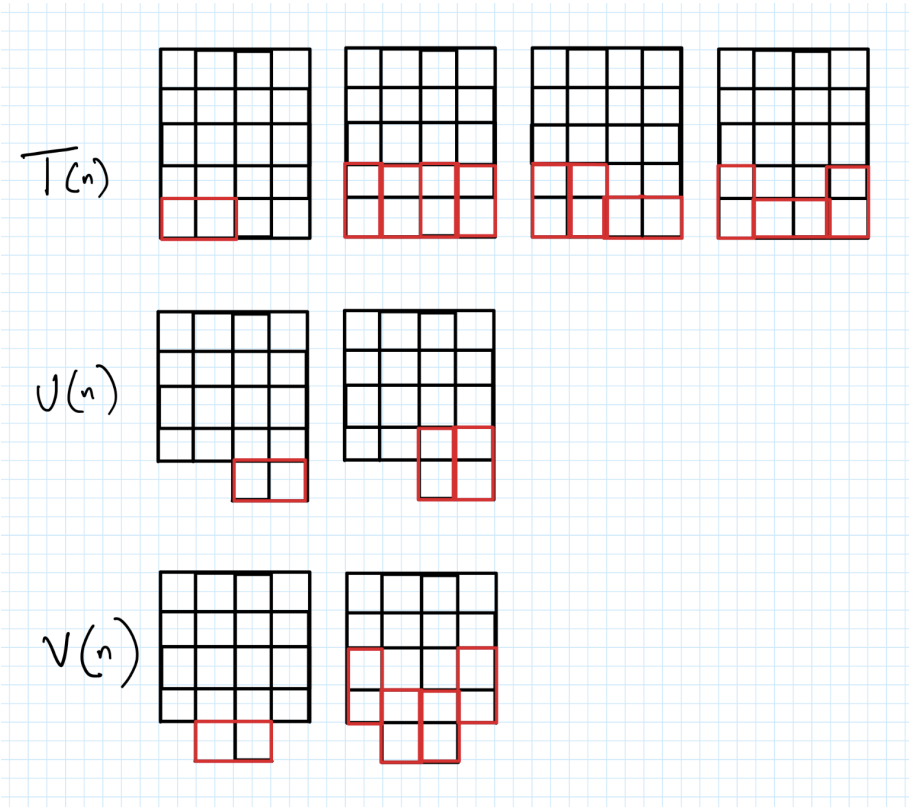

Figure 2: Sub-cases

Here, we divide the main problem into three kinds of subproblems, and maintain 3 DP arrays that compute values of $T(n), U(n), V(n)$. $T(n)$ represents the value of total ways to fit tiles in a $4 \times n$ floor. $U(n)$ is the total number of ways to fit tiles in a $4 \times n$ floor with two consecutive horizontal squares missing, one being the corner and $V(n)$ is the number of ways to fit tiles in a $4 \times n$ floor with two horizontally level corner tiles missing. Also, the case with two central tile missing will be counted in $V(n)$ since it is only possible when we add two vertical tiles at the corner to tiling in $V(n)$. We can start tiling from the bottom left and form a recursion by dividing the problem into smaller problems.

Recursion

T(n):
Base Case: $T(0) = 1, T(1) = 1$.
If we put a horizontal tile first, the problem simply reduces to a case of $U(n)$. If we put 4 vertical tiles, the problem reduces to a subproblem of $T(n-2)$. If first two tiles placed are vertical and the next tile is placed horizontally the problem is reduced to a case of $U(n-1)$. If first tile is vertical and next is horizontal, the third tile is forced to be vertical and towards the corner and hence the problem reduces to $V(n-1)$. Hence,

$$T(n) = U(n) + T(n-2) + U(n-1) + V(n-1) \quad \forall n \geq 2$$

.

U(n):
Base Case: $U(0) = 1$.
If we fit a horizontal tile, the problem reduces to a case of $T(n-1)$ and if we put a vertical tile, next tile has to be vertical and hence we get a flipped case of $U(n-1)$, which has the same value as its mirror image. Hence,

$$U(n) = T(n-1) + U(n-1) \quad \forall n \geq 1$$

V(n):
Base Case: $V(0) = 1, V(1) = 1$.
If we fit a horizontal tile at the bottom end, we get a subproblem of $T(n-1)$. If we fit a vertical tile first, then we are forced to fit the next 3 tiles as shown in the 2nd structure in the lowest row in Figure 2. Hence, after fitting the four tiles, the problem reduces to $V(n-2)$. Hence,

$$V(n) = T(n-1) + V(n-2) \quad \forall n \geq 2$$

At the end the required value is $T(n)$.

Correctness

Base case for T, U and V are trivially true. For the recursive step, the image above describes all the possible case of generating a $4 \times n$ sized tile by adding 2×1 tiles to the smaller tiles. Any other combination of tiles will either get covered in $T(n-2)$, $U(n)$, $V(n-1)$ or $U(n-1)$. We already described above how we can represent $T(n)$ as subproblems $T(n-2)$, $U(n)$, $V(n-1)$ and $U(n-1)$. Similarly for $U(n)$ and $V(n)$, any combination of tiling will be covered in $U(n-1)$, $V(n-1)$ or $T(n-1)$. It is also easy to see that our decomposition of the given problems is into non-redundant subproblems. So our recursive step is correct for this problem. Now, this will terminate because we are recursively calling strictly smaller subproblems (Note that while $T(n)$ calls $U(n)$, $U(n)$ itself depends on $T(n-1)$ and $U(n-1)$ which are values at strictly smaller indices. Hence, $T(n)$ also effectively depends on the values at strictly smaller indices). So our algorithm to find number of 2×1 tiles is correct.

Memoization

We can have 3 DP arrays, to store the values of T,U and V and can be referred to in $\mathcal{O}(1)$ time.

Sub-problem dependencies

$T(k)$ depends on values $U(k)$, $U(k-1)$, $V(k-1)$ and $T(k-2)$. $U(k)$ depends on values $U(k-1)$, $T(k-1)$. $V(k)$ depends on values $T(k-1)$, $V(k-2)$. We can compute for each k, the values in the order $U(k)$, $V(k)$, $T(k)$ so that we already have $U(k)$ pre computed while computing $T(k)$. Hence, the recurrence relations are acyclic.

Time Complexity Analysis

We can compute each value of each array in $\mathcal{O}(1)$ time and we need to calculate a total of $3n$ values. Hence, the time complexity of the recursion is $\mathcal{O}(1) \cdot \mathcal{O}(n) = \mathcal{O}(n)$

Exercise 4

Maximum Weight Subtree

Given an undirected tree with arbitrarily weighted (positive or negative) edges, we need to find a subtree having maximum total weight. The idea of the approach is using Dynamic Programming. We can do this by choosing an arbitrary node v, and apply **Depth First Search** with v as the root node and find the maximum.

Sub-problems

For any given node u and the root node of the tree being arbitrarily fixed to v, we can define the subproblems as:-

- $MWSyes(u)$ - Maximum weight subtree of tree with u as the root node and also included in the subtree
- $MWSno(u)$ - Maximum weight subtree of a tree with u as root node but not included in the subtree

In this way, we can ensure that we are covering both the cases for u. If we have the values of $MWSyes$ and $MWSno$ for all the children of u, then we can calculate it for u.

Recursion

For any tree with root node u, we first compute $MWSyes(u)$. If the node u is being included in the maximum subtree of u then if one of the children w belongs to this subtree, we must include $MWSyes(w)$ and the weighted edge $wt(u, w)$. Note that $MWSyes(w)$ can only attain minimum value 0 as we can choose to include only the node w if the sum falls to less than 0.
Base case: $MWSyes(u) = 0$ if u is a leaf node.
Hence, we have expression for $MWSyes(u)$ as

$$MWSyes(u) = \sum_{w \in S} \max\left(MWSyes(w) + wt(u, w), 0\right)$$

where S is the set of all the children of u, while keeping *randomly chosen* v as the root node.

For computing $MWSno(u)$ we have to consider which of the children has the subtree. It is clear that we can't include subtrees from more than one children, as we are not including u, this would result in more than one mutually disconnected subtrees. Hence we can only consider subtree to belong to one of the children. For each child w, we consider the maximum of the values $MWSyes(w)$ and $MWSno(w)$.
Base case: $MWSno(u) = 0$ if u is a leaf node.
Hence expression for $MWSno(u)$ becomes-

$$MWSno(u) = \max_{w \in S}\left[\max\left(MWSyes(w), MWSno(w)\right)\right]$$

where S is the set of all the children of u, while keeping *randomly chosen* v as the root node.

Finally, we can obtain the global maximum as $max(MWSyes(v), MWSno(v))$.

Memoization Strategy

We maintain the values of 2 DP arrays for $MWSyes$ and $MWSno$ for each node, hence both being sizes of n. The order can be kept in post order traversal, so we compute the values of the children, before we reach the parent.

Sub-problem dependencies

From the recursion strategy, it is clear that each node is only dependent on the values of $MWSyes$ and $MWSno$ of its children, and hence calculation of values in a post order traversal way will ensure that we have the calculated values of the children before we compute for the parent. In this way, we can ensure that the recursion strategy is acyclic.

Recursion Time Complexity Analysis

Computing the value of each element in the array depends on only some previous values in the array (post-order) and total number of children in the tree is n-1. Hence, we can't conclude that filling both the arrays is of time complexity $\mathcal{O}(n)$. And hence the overall time complexity of the algorithm is $\mathcal{O}(n)$.

Correctness of Recursion

It is not difficult to see that our algorithm covers every subtree of a given tree and only the subtrees, through $MWSyes$ and $MWSno$ values, which are propagated to the arbitrarily assigned root node.

There always will be a maximum weight subtree in a given tree (can be a single node iff all edge weights are negative). Let an optimal subtree be T_{opt}. Now, we need to show our recursion returns exactly the optimum value, which is the weight of this subtree. Our algorithm picks an arbitrary v in the tree and assumes this node to be root node.

- If this node v belongs to an optimal subtree T_{opt} then our algorithm will return its weight in the value $MWSyes(v)$. Now clearly, v is the root node for the optimal subtree as well. Let subset S_{opt} be the complete set of children of v that are included in the optimal subtree. Our claim is that the recursion in our algorithm for $MWSyes(v)$ picks exactly that subset of children.
  Proof: Let the set included by our algorithm be S'. Let a node $w \in S'$, w is not in S_{opt}. Then the value $wt(v, w) + MWSyes(w)$ is positive (by recursion argument) but not been included in S_{opt}. Hence, we will have a contradiction on maximum weight of the optimal subtree. And in case $w \in S_{opt}$ but not in S', then $wt(v, w) + MWSyes(w)$ must have been positive (or else it would reduce the value of T_{opt}) and would violate the recursion argument. Base case for $MWSyes$ is trivial with no edge, and in our argument we have assumed $MWSyes(w)$ for children of v, and shown that recursion value for $MWSyes(v)$ is correct. Also, $MWSno(v)$ cannot be higher than $MWSyes(v)$, as that would be a contradiction to the optimality of the weight (because it means that a subtree with a higher weight than the maximum weight exists).
- If this node v doesn't belong to tree then we do a Depth First Search to first reach a node u of an optimal subtree, which will hence become the root of that subtree. Suppose we reach some descendant of u before we reach u. This is not possible as we are always going down in depth and as the whole graph is a tree, we cannot reach u's

descendant without encountering u. Hence, now we will have the value of optimal subtree as $MWSyes(u)$ which we have proved in the earlier part to be correct. Now, we can see that, if $MWSyes(u)$ is maximum for any subtree in the given graph, then $MWSno(w) = MWSyes(u)$, w being parent of u, by our algorithm. In similar way our algorithm will forward this maximum value till $MWSno(v)$ and hence will return this as the answer (as it has the maximum value overall). Again, $MWSno(v)$ will not be smaller than $MWSyes(v)$, because of the similar reason as stated in the previous point.

Maximum Weight Path

Given an undirected tree with arbitrarily weighted (positive or negative) edges, we need to find a path with the maximum total weight. The idea of this approach is quite similar to the previous subsection and we have to modify the recursion to achieve the required task.

Sub-problems

For any given node u and the root node of the tree being arbitrarily fixed to v, we can define the subproblems as:-

- $MWPone(u)$ - Maximum weight path of tree with u as the root node and also the ending (corner) node of the path in that tree
- $MWPtwo(u)$ - Maximum weight path of tree with u as the root node and a middle node of the path in that tree
- $MWPzero(u)$ - Maximum weight path of a tree with u as root node but not included in the path

Hence, this way we are covering all three cases for each node u.

Recursion

For node u, we first compute $MWPone(u)$.
Base Case: $MWPone(u) = 0$ for leaf node.
To construct such a maximum path, we must take the maximum of all the $MWPone$ paths of children and add the corresponding edge to sum. This holds because we can only extend the paths that have only one edge at the children of u. Hence, the expression holds-

$$MWPone(u) = \max_{w \in S} \left[\max(MWPone(w) + wt(u, w), 0) \right]$$

where S is the set of all the children of u, while keeping *randomly chosen* v as the root node.

For computing $MWPtwo(u)$ we take the two maximum $MWPone(w) + wt(w, u)$ values of children and join them via u to get u as a middle node in the path.
Base Case: $MWPtwo(u) = 0$ for leaf node.
The expression hence, becomes

$$MWPtwo(u) = \max_{w,x \in S, w \neq x} \left[(MWPones(w) + wt(u, w)) + (MWPones(x) + wt(u, x)), 0 \right]$$

where w,x are the children with top two maximum values of $MWPones(k) + wt(u, k)$, $k \in S$. Note that this can be found in $\mathcal{O}(|S|)$ time by keeping track of top two max values. Also, note that the min possible values for all the elements in all the DP arrays is zero.

For computing $MWPzero(u)$, we have to consider the max out of all the DP array values in the children of u and not include any edge from u to its children.
Base Case: $MWPzero(u) = 0$ for leaf node.
The expression hence is-

$$MWPzero(u) = \max_{w \in S} \left[max(MWPzero(w), MWPone(w), MWPtwo(w)) \right]$$

where S is the set of all the children of u, while keeping *randomly chosen* v as the root node.

Finally we obtain the total maximum path of the whole tree by computing

$$max(MWPzero(v), MWPone(v), MWPtwo(v))$$

Memoization

We store the values of $MWPzero, MWPone, MWPtwo$ for all nodes in **post order**. This needs 3 arrays of n size and hence a space complexity of $\mathcal{O}(n)$.

Sub-problem dependencies

From the recursion strategy, it is clear that each node is only dependent on the values of $MWPzero,\ MWPone$ and $MWPtwo$ for its children, and hence our recursion is acyclic.

Time Complexity Analysis

For each node, the 3 values are dependent only on the values of their children which are already computed. Hence, we can argue that, as there are a total of n-1 children in the whole tree, the filling up of the 3 DP arrays will be linear in n ie. $\mathcal{O}(n)$. Hence the overall time complexity is $\mathcal{O}(n)$

Correctness of Recursion

Again, it is not difficult to see that our algorithm covers every path in a given tree and only the paths, through $MWSyes$ and $MWSno$ values, which are propagated to the arbitrarily assigned root node.

This can be proved in a quite similar way as in first part, by taking cases on root node's inclusion in maximum weight path and arguing our recursion to output the same optimum value. Let an optimum path be P_{opt} and root node chosen arbitrarily by algorithm at the start be v.

- If v is an ending node in P_{opt}, then our algorithm will return $MWPone(v)$. We can prove by induction that $MWPone(v)$ returns the maximum value given the condition for v. Base case is a single node k with no edges hence $MWPone(k) = 0$ holds true. Assuming the values of $MWPone$ are correct till children of some node l, we need to show it is true for l. We need to connect on of the children to l that would give the maximum value, hence we need to choose $MWPone(l') + wt(l, l')$ to be maximum and that is what our recursion does at each step. Hence our recursion for $MWPone$ is correct till root node v by induction.
- If v is a middle node in P_{opt}, then it must be connected to exactly two of its children and these children are in the path. So we would be needing the values of $MWPone$ for

the children so as to add an edge to connect them to parent v. Also we have shown the correctness for $MWPone$ in the previous part. Now, our algorithm picks the top 2 nodes among the children with maximum values for $wt(u, v) + MWPone(u)$, where u a child of v. This must be equal to the weight of the path P_{opt} or else we would have a contradiction on our assumption of maximum weight of P_{opt}. Hence, $MWPtwo(v)$ would return the weight equal to that of P_{opt}.

- If v is not included in P_{opt}, then we can apply DFS to reach first a node u in an optimal path, which will become a root node in that context during the DFS for its subtree, which contains all the remaining nodes in the path. Argument for this is similar to the one done in previous section. Now, this root node u can be an ending node or a middle node in P_{opt}. Hence, one of $MWPone(u)$ or $MWPtwo(u)$ will correspond to this value. This is the actual optimum for the whole graph. Now this value will be propagated to the parent of u by our algorithm as $MWPzero(w)$, where w is parent of u, will pick the maximum value from its children (given by the recursion). And hence, this value gets propagated to the root node v as $MWPzero(v)$ and is the true maximum weight any path in the whole tree.
- Again, like in the previous part, we can easily show that if one of $MWPone(u)$, $MWPtwo(u)$ and $MWPzero(u)$ corresponds to a maximum weight path, then neither of the other two can be larger than it, because that would lead to a contradiction.

Exercise 5

5a. Maximum Weight Increasing Subsequence

For a given array of elements, with arbitrary weights assigned to them, we have to find the increasing subsequence with maximum weight sum. This problem is similar to the longest increasing subsequence problem, hence, we will be using Dynamic Programming Approach to find an algorithm.

Sub-problems

For a given array of elements of size n, $arr = [a_0, a_1, a_2....., a_{n-1}]$ with weights assigned $W = [w_0, w_1, w_2, ..., w_{n-1}]$.

We can define our sub problem as $MWIS[k]$, which will calculate the maximum weight sum of increasing sub sequence ending at index k. Ending at index k means, the function calculates the sum considering a_k as the last element of sub sequence and upto k^{th} index.

For index 0, $MWIS[0] = w_0$

Similarly, upto index 1,
$MWIS[1] = w_0 + w_1$, if $a_0 < a_1$
$MWIS[1] = w_1$, else if $a_0 \geq a_1$

Recursion

The recursion strategy for the problem will be similar to that of LIS problem. For any index k, the MWIS weight will be greater than the maximum MWIS weight of all subsequences, upto index j such that $arr[j] < arr[k]$ $(j < k)$. And, the maximum weight sum upto that index will be sum of it's weight and the maximum subsequence weight so far. This statement can be expressed as a recursion strategy.

Base Case Initialization: $MWIS[0] = w_0$
For all j such that $j < i$ and if $a_j < a_i$, $MWIS[i] = \max_j(MWIS[j]) + w_i$
If there is no such j, $MWIS[i] = w_i$

The maximum weight increasing subsequence will have the **maximum value among** $MWIS$ **values**. To find the maximum weight increasing subsequence, we can iterate over the values of $MWIS$ and the maximum value will correspond to the weight sum of the required subsequence.

Memoization Strategy

We can store the values of maximum increasing weight sum, $MWIS[k]$, in an array of size n, which will correspond to the space complexity of $\mathcal{O}(n)$.

Sub-problem dependencies

Since our recursion strategy is calling for the values stored in smaller indices to calculate the values for larger index, i.e, from the start in linear order, and each value after starting index, uses the previously calculated values in DP table/arrays linearly and stores it in the table. Hence, our sub-problem dependencies are acyclic.

Recursion time complexity analysis

The total number of sub-problems is $O(n)$ for each element, and for each i^{th} element, iteration is done from $j = 0$ to $j = i - 1$, and maximum among them is calculated.

The time for each sub problem is $\mathcal{O}(i)$, which will be bounded by $\mathcal{O}(n)$ upon summation.

Therefore, Total time = $\mathcal{O}(n) \times \mathcal{O}(n) = \mathcal{O}(n^2)$.

Note that, time complexity can be further reduced to $\mathcal{O}(n \log n)$, by implementing binary search instead of a loop for each sub problem.

Correctness

Let $Opt(i)$ denote the optimal solution for the sub problem upto index i, i.e, $\{0, 1, 2, 3, \dots, i\}$.

Induction: To prove, that Opt and $MWIS$ corresponds to same value for each i.

Trivially, $Opt(0) = w_0$ as the maximum weight increasing subsequence till index 0 will be $[a_0]$ and weight sum corresponding to it equals w_0.
Hence, $Opt(0) = MWIS(0) = w_0$

Assume induction hypothesis holds $\forall j < i$:

- Due to induction hypothesis, we know that $\forall j < i, Opt(j) = MWIS[j]$
- From our previous reasoning in `Recursion` section, the correct answer for the sub problem is $w_i + \max_j(Opt(j))$, because if the subsequence's last element is a_i, then the maximum weight increasing subsequence weight sum will include w_i and the maximum weight sum of a subsequence which end at other element a_j,such that, $j < i$. Hence, all possible j are considered and maximum among them is considered.
- By our recursion strategy, $MWIS(j) = w_i + \max_j(MWIS(j))$. Hence, $Opt(i) = MWIS(i)$.
- Therefore, by induction, $\forall i, Opt(i) = MWIS(i)$. And thus, the maximum value among this array, will give the final result.

Hence, the algorithm calculates each sub problem correctly and therefore, overall correctness of algorithm is derived.

5b. Maximum Weight Subsequence (No three consecutive elements)

For a given array of elements, with arbitrary weights assigned to them, we have to find the subsequence with maximum weight sum such that no three consecutive elements from original array occur in sub sequence. Approach used here is Dynamic Programming.

Sub-problems

For a given array of elements of size n, $arr = [a_0, a_1, a_2....., a_{n-1}]$ with weights assigned $W = [w_0, w_1, w_2, ..., w_{n-1}]$.

We can define our sub problem as $sum[k]$, which will calculate the maximum weight sum of sub sequence upto index k, such that no three consecutive elements occur.Upto index k, means the problem is reduced for array of length n to array of length $k + 1$, i.e from index 0 to index k.

It is trivial that $sum[0] = w_0$
Also, $sum[1] = w_0 + w_1$

Now, for further indexes, the values can be calculated by taking cases on the three consecutive elements whether to be included or not. This is explained in Recursion strategy section.

Recursion

Recursion Strategy can be determined by taking cases on indices:

For i = 2, we have three cases, as follows:

- $sum[2] = w_0 + w_1 = sum[1]$, if a_2 is not included.
- $sum[2] = w_0 + w_2 = sum[0] + w_2$, if a_1 is not included.
- $sum[2] = w_1 + w_2$, if a_0 is not included.

Therefore, $sum[2] = \max(sum[1], sum[0] + w_2, w_1 + w_2)$

Similarly, for i = 3, we have three cases:

- $sum[3] = sum[2]$, if a_3 is not included.
- $sum[3] = sum[1] + w_3$, if a_2 is not included.
- $sum[3] = w_0 + w_2 + w_3 = sum[0] + w_2 + w_3$, if a_1 is not included.

Hence, $sum[3] = \max(sum[2], sum[1] + w_3, sum[0] + w_2 + w_3)$

Therefore, in general, for index i:

- $sum[i] = sum[i - 1]$, if a_i is not included.
- $sum[i] = sum[i - 2] + w_i$, if a_{i-1} is not included.
- $sum[i] = sum[i - 3] + w_{i-1} + w_i$, if a_{i-2} is not included.

Therefore, $sum[i] = \max(sum[i - 1], sum[i - 2] + w_i, sum[i - 3] + w_{i-1} + w_i)$

Using this recursion to find the final answer is explained in memoization section.

Memoization Strategy

We can store the values of maximum weight sums, $sum[k]$, in an array of size n, which accounts for space complexity $\mathcal{O}(n)$.

Now, for each i, there are three cases, each of which corresponds to not including a certain element. For each sub problem, the index of that element is known after knowing the maximum value and to which case it corresponds. That index is stored in an array S (i.e indices which are not included in thee subsequence) upon linear traversal of the array. This array accounts for space $\mathcal{O}(n)$.

The final result, i.e the subsequence required can be found by a linear traversal of array S, including indices which are not in S, from 0 to n.

$MWS = [a_i | i \notin S]$ (MWS such that no three consecutive elements occur)

Sub-problem dependencies

Since our recursion strategy is calling for the elements from index 0, i.e, from the start in linear order, and each value after index 0, uses the previously calculated values in DP table/array linearly, i.e $sum[i-1]$, $sum[i-2]$ and $sum[i-3]$. Hence, our sub-problem dependencies are acyclic.

Recursion time complexity analysis

The total number of sub-problems is $O(n)$, accounting each element as a sub problem.

The time for each sub problem is $\mathcal{O}(3) = \mathcal{O}(1)$, because, for each sub problem, 3 expressions are calculated and maximum among them and which element is to be excluded is stored, which accounts for constant time, as all the elements are known from sum array and given weight array.

Therefore, Total time = $\mathcal{O}(n) \times \mathcal{O}(1) = \mathcal{O}(n)$.

Correctness

Let $Opt(i)$ denote the optimal solution for the sub problem upto index i, i.e, $\{0, 1, 2, 3...., i\}$.

Induction: To prove, that $Opt(i)$ and $sum(i)$ corresponds to same value for each i.

Trivially, $Opt(0) = w_0$ as the maximum weight subsequence till index 0 will be $[a_0]$ and weight sum corresponding to it equals w_0.
Hence, $Opt(0) = sum(0) = w_0$

Also, $Opt(1) = w_0 + w_1$, as the required subsequence will be $[a_0, a_1]$, as no three consecutive elements are possible. Also, by base case of recursion strategy, $sum(1) = w_0 + w_1$. Therefore, $Opt(1) = sum(1)$

For $i = 2$, the correct answer can be reasoned by taking cases, that among three elements, which element is to be excluded. Therefore:

- $Opt(2) = w_0 + w_1$, if a_2 is not included.
- $Opt(2) = w_0 + w_2$, if a_1 is not included.

- $Opt(2) = w_1 + w_2$, if a_0 is not included.

Correct answer will be maximum weighted case, hence $Opt(2) = \max(w_0 + w_1, w_0 + w_2, w_1 + w_2)$
Now, $Opt(2) = sum(2)$ (Can be confirmed with $sum(2)$ value in Recursion section)

Assume induction hypothesis holds $\forall 2 < j < i$:

- Due to induction hypothesis, we know that $\forall 2 < j < i, Opt(j) = sum(j)$
- From our previous reasoning by taking cases, the correct answer for the sub problem will be:
  - $Opt(i) = Opt(i-1)$, if a_i is not included.
  - $Opt(i) = Opt(i-2) + w_i$, if a_{i-1} is not included.
  - $sum(i) = Opt(i-3) + w_{i-1} + w_i$, if a_{i-2} is not included.

  Correct answer will be maximum weighted case, hence $Opt(i) = \max(Opt(i-1), Opt(i-2) + w_i, Opt(i-3) + w_{i-1} + w_i)$
- By our recursion strategy, $sum(i) = \max(sum(i-1), sum(i-2) + w_i, sum(i-3) + w_{i-1} + w_i)$. And, by induction hypothesis, $sum(i) = \max(Opt(i-1), Opt(i-2) + w_i, Opt(i-3) + w_{i-1} + w_i)$. Hence, $Opt(i) = sum(i)$
- Therefore, by induction, $\forall i, Opt(i) = sum(i)$. And thus, the final answer will be $Opt(n-1) = sum(n-1)$.

During the calculation of each sub problem, the index of element to be excluded is stored in an array MWS, and after all iterations, the required sub sequence will be the array containing the indices not in MWS. This is correct due the case reasoning above and in `Memoization` section.

Hence, the algorithm calculates each sub problem correctly and therefore, overall correctness of algorithm is derived.