CS 218: Assignment 2

Ankit Kumar Misra	Devansh Jain	Harshit Varma	Richeek Das
190050020	190100044	190100055	190260036

March 10, 2021

Contents

Question 1

Problem:
Given an n-digit number a, give an algorithm to compute a^2 that runs in time $n^{\log_3 6}$.
Give the correctness of your algorithm and a detailed justification for its time complexity.

Principle:
Inspired by Karatsuba's Multiplication Algorithm, we use the following identity to our advantage.

$$\begin{aligned}
(10^{2m}a_2 + 10^m a_1 + a_0)^2 &= 10^{4m}(a_2^2) + 10^{3m}(2a_2a_1) + 10^{2m}(a_1^2 + 2a_2a_0) + 10^m(2a_1a_0) + a_0^2 \\
(2a_2a_1) &= (a_2)^2 + (a_1)^2 - (a_2 - a_1)^2 \\
(2a_2a_0) &= (a_2)^2 + (a_0)^2 - (a_2 - a_0)^2 \\
(2a_1a_0) &= (a_1)^2 + (a_0)^2 - (a_1 - a_0)^2 \\
(10^{2m}a_2 + 10^m a_1 + a_0)^2 &= 10^{4m}[(a_2)^2] + 10^{3m}[(a_2)^2 + (a_1)^2 - (a_2 - a_1)^2] \\
&\quad + 10^{2m}[(a_2)^2 + (a_1)^2 + (a_0)^2 - (a_2 - a_0)^2] \\
&\quad + 10^m[(a_1)^2 + (a_0)^2 - (a_1 - a_0)^2] + [(a_0)^2]
\end{aligned} \tag{1}$$

Pseudo Implementation:

```
Algorithm 1: FSQ($a, n$)
  Input: Given $a$ and $n$, number of digits in $a$
  Output: $a^2$, square of $a$
  if $n = 1$ then
      /* If $a$ is single digit then compute square in O(1)                     */
      return $(a * a)$
  else
      /* Divide: Splitting $a$ as $10^{2m}a_2 + 10^m a_1 + a_0$                     */
      $m \leftarrow \lceil n/3 \rceil$
      $a_2 \leftarrow \lfloor a/10^{2m} \rfloor$
      $a_1 \leftarrow \lfloor (a \mod 10^{2m})/10^m \rfloor$
      $a_0 \leftarrow (a \mod 10^m)$
      /* Recurse: Calculating squares of six $\lceil n/3 \rceil$ digits numbers      */
      $s_{22} \leftarrow \text{FSQ}(a_2, m)$
      $s_{11} \leftarrow \text{FSQ}(a_1, m)$
      $s_{00} \leftarrow \text{FSQ}(a_0, m)$
      $s_{21} \leftarrow \text{FSQ}(|a_2 - a_1|, m)$
      $s_{20} \leftarrow \text{FSQ}(|a_2 - a_0|, m)$
      $s_{10} \leftarrow \text{FSQ}(|a_1 - a_0|, m)$
      /* Combine: Use the calculated squares to obtain square of $a$               */
      $s \leftarrow 10^{4m} * (s_{22})$
      $s \leftarrow s + 10^{3m} * (s_{22} + s_{11} - s_{21})$
      $s \leftarrow s + 10^{2m} * (s_{22} + s_{11} + s_{00} - s_{20})$
      $s \leftarrow s + 10^m * (s_{11} + s_{00} - s_{10})$
      $s \leftarrow s + s_{00}$
      return $s$
  end
```

Correctness:
Follows from identity (Eq 1) mentioned in the Principle.

Time Analysis:
First, let's state the primitive operations:

1. Adding two single digit numbers in $\Theta(1)$ time
2. Multiplication of two single digit numbers in $\Theta(1)$ time
3. Integer division of two single digit numbers in $\Theta(1)$ time
4. Adding a zero at the end in $\Theta(1)$ time
5. Removing a digit from the end in $\Theta(1)$ time

From these we can get the following operations in $\Theta(n)$ time:

1. Adding two n digit numbers
2. Subtracting two n digit numbers (or finding difference)
3. Multiplication of any number with 10^n
4. Integer division of a n digit number by a single digit number
5. Integer division of any number by 10^n
6. Calculating remainder of any number upon integer division by 10^n

Let $T(n) :=$ Maximum running time of FSQ(a, n) for any n digit number a

$T(1) = \Theta(1)$ (From Line 2)

Time taken for Line 8 thru 13 $\leq 6T(\lceil n/3 \rceil)$
Operations involved in Line 4 thru 7 and Line 14 thru 18 take $\Theta(n)$ time
$T(n) \leq 6T(\lceil n/3 \rceil) + \Theta(n)$

The recursion tree method transforms this recurrence into an increasing geometric series,
Or directly applying Master Theorem on $T(n) = 6T(n/3) + \Theta(n)$,
We get $T(n) = \Theta(n^{\log_3 6})$ as desired.

Comments:

1. The reason for converting $T(\lceil n/3 \rceil)$ to $T(n/3)$ is stated in Page 34 of Jeff Erickson's book "Algorithms".
2. Instead of using the identity $2ab = (a)^2 + (b)^2 - (a-b)^2$, we could have used $2ab = (a+b)^2 - (a)^2 - (b)^2$, but $(a + b)$ would have been a $m + 1$ digit number and the recursion would have been more complex though the running time would still be $\Theta(n^{\log_3 6})$,
   Reference: Footnote on Page 41 of Jeff Erickson's book "Algorithms".

Question 2

Problem:
Suppose we are given the heights of all **n** heroes, in order from left to right, in an array **Ht[1 .. n]**. (No two heroes have the same height.) Then we can compute the *Left* and *Right* targets of each hero in $\mathcal{O}(n^2)$ time using the following brute-force algorithm.

Algorithm 2: WhoTargetsWhom($Ht[1 \dots n]$)

```
  Input: Ht array containing heights of the n heroes
  Output: Left and Right targets of each hero
1 for $j = 1$ to $n$ do
      /* Find the left target L[j] for hero j                    */
2     $L[j] \leftarrow None$
3     for $i = 1$ to $j - 1$ do
4         if $Ht[i] > Ht[j]$ then
5             $L[j] \leftarrow i$
6         end
7     end
      /* Find the right target R[j] for hero j                   */
8     $R[j] \leftarrow None$
9     for $k = n$ to $j + 1$ do
10        if $Ht[k] > Ht[j]$ then
11            $R[k] \leftarrow k$
12        end
13    end
14 end
15 return $L[1 \cdots n], R[1 \cdots n]$
```

1. **Problem:** Describe a divide-and-conquer algorithm that computes the output of **WhoTargetsWhom** in $\mathcal{O}(nlogn)$ time.

   Approach: Since we see the key points "divide and conquer" and a linearithmic time complexity, lets aim for the recurrence relation $\longrightarrow T(n) = 2 \times T(n/2) + \mathcal{O}(n)$.

   - **Subproblems**: We break the main problem with an array of n elements from the middle into two arrays of $n/2$ elements each. We solve the subproblems of size $n/2$ and get the *left* and *right* targets of each hero in the sub arrays. Lets see how we combine the results from the $n/2$ sized arrays into the result for n elements.
   - $\mathcal{O}(n)$ **Combination step**: We will use a two-pointer approach for combining the results from the two sub-arrays. **Few things to notice in order to understand the solution:**

     Claim 1: Any hero who has been assigned a left or right target in the subproblems, will retain the same target after the combination.

     Proof: If a hero could find any of its targets in the subproblem then the distance from the hero to the target is $< n/2$. Let's **assume** that the target obtained from the subproblem is wrong and it gets replaced by some other target in the combination step. If that happens then new distance between hero and target is $> n/2$. We already had a target which was closer! Contradiction. Our assumption is wrong. Therefore, claim is true.

Due to the last claim, we can ignore the heroes in the left subarray who already have a right target.

Claim 2: Let $a[1\cdots k]$ be the left sub array with the heroes without any right target. Array $a[1\cdots k]$ is sorted in descending order.

Proof: **Assume** $a[1\cdots k]$ is not sorted in descending order. This means there exists some index z such that, $a[z-1] < a[z]$. Therefore, the hero at $z-1$ can have a potential right target at z. But by definition, $a[1\cdots k]$ doesn't have any heroes with right targets. Contradiction. Therefore, our assumption is wrong. Claim is correct.

Second claim is **important** for using the two-pointers strategy.

The actual combining step: **We will solve the problem for finding the right target.** Say, we have solved the subproblems for $a1 = a[1\cdots m]$ and $a2 = a[m+1\cdots n]$, where a is the original array and $m = \lfloor \frac{n}{2} \rfloor$.

(a) We keep two pointers i and j. Initially $i = m$ and $j = m+1$.

(b) `while` $a[j] < a[i]$ we do $j++$.

(c) `If` $a[j] > a[i]$ we set j as the right target of i.

(d) `while` $a[j] > a[i]$ we do $i--$ (Because of **claim 2**, the right target of $i' < i$ is definitely $j' \geq j$). We keep a check and skip i which already have a right target(because of **claim 1**).

(e) We do all of these steps till, $i > 0$ **and** $j < n+1$.

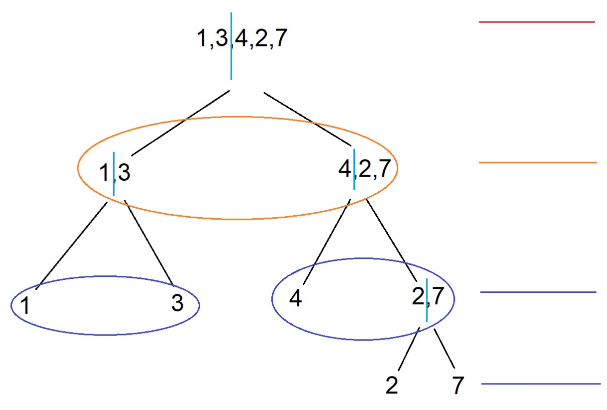
In every step we either incremented j or decremented i. Therefore, we can have at max n iterations. Therefore the **combining step** algorithm is $\mathcal{O}(n)$!

- **Base case:** We have two subproblems each of size 1(we can't break further). We can check if the element in the right subarray is the right target of the element in the left subarray using a single comparison. Therefore base case solved in $\mathcal{O}(1)$ time! $\mathbf{T(1)=c}$.
- **Proof of Correctness:** All the combination steps and the claims used are proved above. Still, lets propose a proof of correctness which shows that **step (c)** of our combination step indeed chooses the correct right targets.

  Proof: Step **(b)** increments the right pointer **j** keeping **i** fixed until $a[j] > a[i]$. We choose the first such **j** for which this happens.Therefore, we have the smallest $j - i$ distance for every i! Rest of it follows trivially from **Claim 2**.
- **Summing up:** From the combination step and the base case, we see that the recurrence relation we aimed for before, is indeed possible. We have a time complexity of $\mathcal{O}(nlogn)$. If the above conclusion seems difficult to follow, see the example that follows.

  The exact same idea can be used to find the **left targets** in $\mathcal{O}(nlogn)$ time!

Example: Let $\mathbf{Ht[1 \cdots n]} = \{1, 3, 4, 2, 7\}$. The recurrence tree and combinations:



R = {1,2,4,4,N} are the final right targets

R = {1,2,4,4,N} // As mentioned in the algorithm, we keep two pointers. One starting on index 1, other at index 2 and proceed as the described algo.

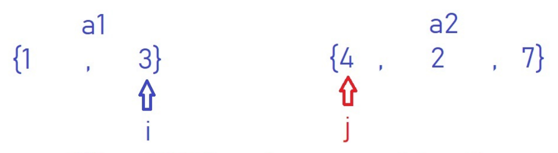
R = {1,N,4,4,N} // We solved the [0 .. 1] and [2 .. 4] sub problems. Now we will combine the results.

R = {N,N,N,4,N} // We are using 0th-index

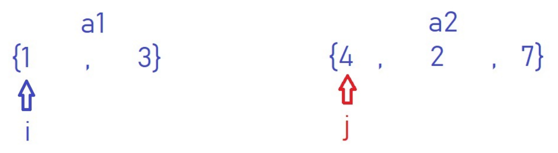
Initially R={N,N,N,N,N} // N stands for None

Say, we solved the problems a1{1,3} and a2{4,2,7} :
R[0 ... 1] = {1,N}
R[2 .. 4] = {4,4,N}

Let's visualize, how we merge these results together for the problem: {1,3,4,2,7} Initially, i=1 and j=2



we see, a1[i] < a2[j] therefore we set j as the right target of i



We decrement i (i=0 now). But, R[i] != None. Therefore, we decrement i further. i = -1. We are done.

2. **Problem:** Prove that at least $\lfloor \frac{n}{2} \rfloor$ of the n heroes are targets. That is, prove that the output arrays $R[0 \cdots n-1]$ and $L[0 \cdots n-1]$ contain at least $\lfloor \frac{n}{2} \rfloor$ distinct values (other than None).

Proof: It is easy to see that, **for a hero to be not a target** in a particular round $\longrightarrow$ "It must be a local minimum". Which means, **if** hero **i** is not a target, then $a[i-1] > a[i]$ and $a[i] < a[i+1]$. If any of these inequalities do not hold, then $a[i]$ will be the right target of $a[i-1]$ or the left target of $a[i+1]$ respectively.

If we want to find the **minimum number of targets** we need to **maximise the number of non-targets**. Therefore we want to maximise the number of **local minima**!

Notice that **we cannot have two local minima** adjacent to each other. This sets the maximum possible number of local minima to $\longrightarrow \lceil \frac{n}{2} \rceil$. Example of such a case: $[lm, t, lm, t, lm, t, lm]$ where lm is local minima and t is the target.

Therefore there are **atleast** $n - \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$ **heroes who are targets**.

3. **Problem:** Describe and analyze an algorithm to compute the **number of rounds** before Dr. Metaphor's deadly process finally ends. Algorithm should run in $\mathcal{O}(n)$ time.

Proof: As we saw in the last problem, heroes which are **non-targets in a particular round**, must be **local minima**. So our idea would be to find all the local minimas in every round. We can easily do that in $\mathcal{O}(n)$ time complexity (we will see the pseudo-code shortly). Also, from the last proof we see that, we will find atleast $\lfloor \frac{n}{2} \rfloor$ targets in each round having n heroes.

Therefore, we find and **retain** the **local minimas in every round** and remove the rest. So, in each step our problem is getting reduced to a new sub-problem with a worst case of $\lceil \frac{n}{2} \rceil$ heroes.

We can write this in the form of a **recurrence relation:**

$$T(n) = T(\left\lceil \frac{n}{2} \right\rceil) + \mathcal{O}(n)$$

Let's solve this recurrence. Let $n = 2^m$. Then, the last equation turns into $T(n) = T(n/2) + \mathcal{O}(n)$. We can directly apply masters theorem on this new equation. We get, $\longrightarrow T(n) = \Theta(n)$

If $n \neq 2^m$ then we can say, there exists m such that $2^m < n < 2^{m+1}$. Therefore,

$$T(2^m) < T(n) < T(2^{m+1})$$
$$\implies 2^m < T(n) < 2^{m+1}$$

So, as $m \longrightarrow \infty$, $T(n) \longrightarrow 2^m = n$. Therefore, $T(n)$ is $\Theta(n)$. Check the pseudo-code in the next page.

Algorithm 3: NumberOfRounds($\{Ht\}_{i=1}^{n}$)

```
Input: The Heights of all the heroes ordered as given in the array $\{Ht\}_{i=1}^{n}$
Output: An integer storing the number of rounds need to be taken to end the game
if $Ht$ has less than 2 elements then
    return 0
end
/* localMinimas will store the indexes of the local minimas in Ht */
if $Ht[1] < Ht[2]$ then
    $localMinimas \leftarrow 1$
end
if $Ht[n-1] > Ht[n]$ then
    $localMinimas \leftarrow n$
end
for $k = 2$ to $n-1$ do
    if $Ht[k-1] > Ht[k] < Ht[k+1]$ then
        $localMinimas \leftarrow k$
    end
end
Remove the indices not in localMinimas from Ht
return $1 + NumberOfRounds(Ht_{i=1}^{z})$
```

Question 3

Problem (3x2n):

Suppose you are given rectangular tiles with height 1 and width 2. You have to use these to tile a rectangle of height 3 and width 2n.
Describe an $\mathcal{O}(n)$ time algorithm to find the number of ways in which this can be done.
Note that tiling a rectangle means covering it by tiles whose interiors are disjoint. A tile may be placed vertically or horizontally.

Solution:

The required complexity of $\mathcal{O}(n)$ suggests a Dynamic Programming approach.

Step 1 - Types of Sub-Problems:
Let $f(k)$ denote number of ways to tile a $3 \times 2k$ rectangle using 2×1 tiles.
Let $g(k)$ denote number of ways to tile a $3 \times (2k+1)$ rectangle with a corner (top-right or bottom-right) missing using 2×1 tiles.
We need to compute $f(n)$.

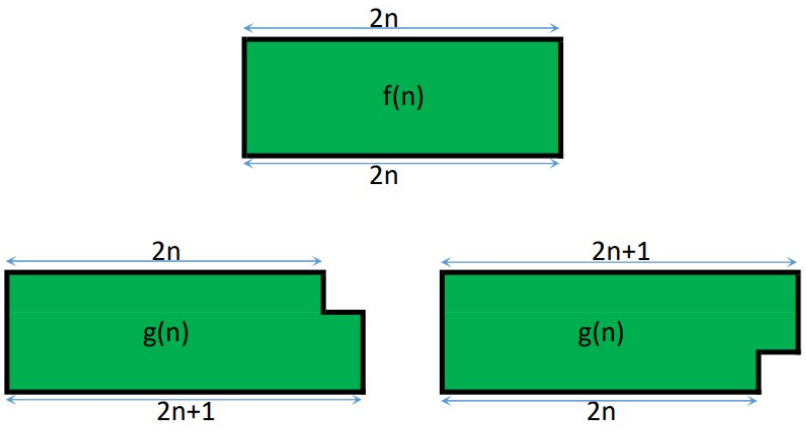


Figure 1: Defining $f(n)$ and $g(n)$

Step 2 - Recursive Relation:
We have a mutual recursion relationship between $f(.)$ and $g(.)$
Base Case:
$f(0) = 1$
$g(0) = 1$
Recursive definition:
$f(n) = f(n-1) + 2g(n-1) \; \forall \; n \geq 1$
$g(n) = g(n-1) + f(n) \; \forall \; n \geq 1$

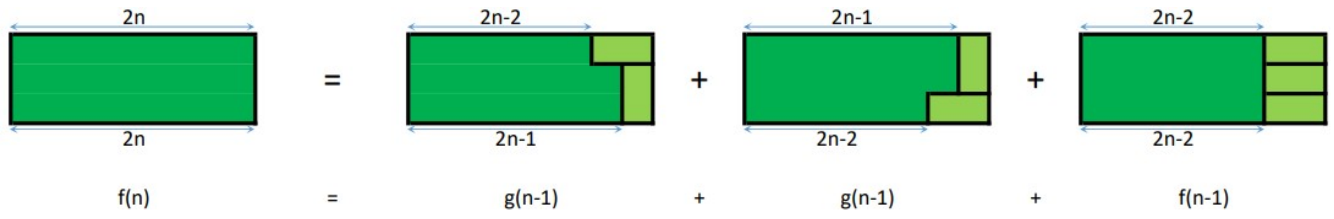


Figure 2: Proof of recursive definition of $f(n)$, for $n \geq 1$

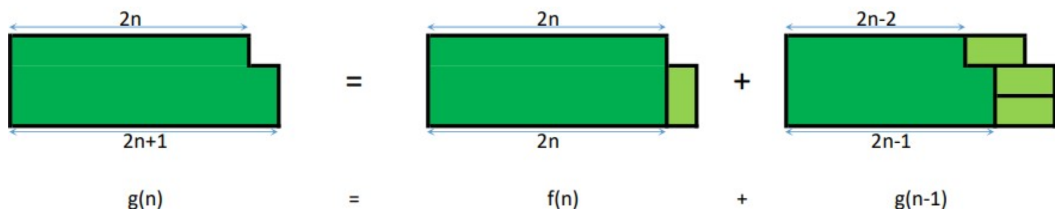


Figure 3: Proof of recursive definition of $g(n)$, for $n \geq 1$

Step 3 - Memoization Strategy:
Two arrays, one each for storing values of $f(i)$ and $g(i)$, can be used for memoization.
Let us call these arrays `F` and `G`, each of size n.

Step 4 - Acyclicity of Dependencies:
$f(i)$ calls $g(.)$ with index $\leq i-1$ which calls $f(.)$ with index $\leq i-1$.
$f(i)$ calls $f(.)$ with index $\leq i-1$.
Thus, $f(i)$ would never call itself.
$g(i)$ calls $f(.)$ with index $\leq i$ which calls $g(.)$ with index $\leq i-1$.
$g(i)$ calls $g(.)$ with index $\leq i-1$.
Thus, $g(i)$ would never call itself.

Step 5 - Time Complexity Analysis:
The total number of sub-problems is $\mathcal{O}(n)$ (including both $f(.)$ and $g(.)$).
The time taken per sub-problem is $\mathcal{O}(1)$ (basic arithmetic operations on integers).
Total time = $\mathcal{O}(n) \times \mathcal{O}(1) = \mathcal{O}(n)$ as required.

Psuedo Implementation:

Algorithm 4: CountWays3x2n(n)

```
Input: $n$, integer
Output: Number of ways of tiling a $3 \times 2n$ rectangle using $2 \times 1$ tiles
Def $f(i, F, G)$:
    if F[$i$] is defined then
        return F[$i$]
    end
    F[$i$] $\leftarrow f(i-1, F, G) + 2 * g(i-1, F, G)$
    return F[$i$]

Def $g(i, F, G)$:
    if G[$i$] is defined then
        return G[$i$]
    end
    G[$i$] $\leftarrow g(i-1, F, G) + f(i, F, G)$
    return G[$i$]

F, G empty array initialized
F[0] $\leftarrow 1$
G[0] $\leftarrow 1$

return $f(n, F, G)$
```

Problem (4xn):

Suppose you are given rectangular tiles with height 1 and width 2. You have to use these to tile a rectangle of height 4 and width n.
Describe an $\mathcal{O}(n)$ time algorithm to find the number of ways in which this can be done.
Note that tiling a rectangle means covering it by tiles whose interiors are disjoint. A tile may be placed vertically or horizontally.

Solution:

The required complexity of $\mathcal{O}(n)$ suggests a Dynamic Programming approach.

Step 1 - Types of Sub-Problems:
Let $f(k)$ denote number of ways to tile a $4 \times k$ rectangle using 2×1 tiles.
Let $g(k)$ denote number of ways to tile a $4 \times (k+1)$ rectangle with a corner (top-right or bottom-right) and its vertically neighbouring box missing using 2×1 tiles.
Let $h(k)$ denote number of ways to tile a $4 \times (k+1)$ rectangle with two right edge's center boxes missing using 2×1 tiles.
We need to compute $f(n)$.

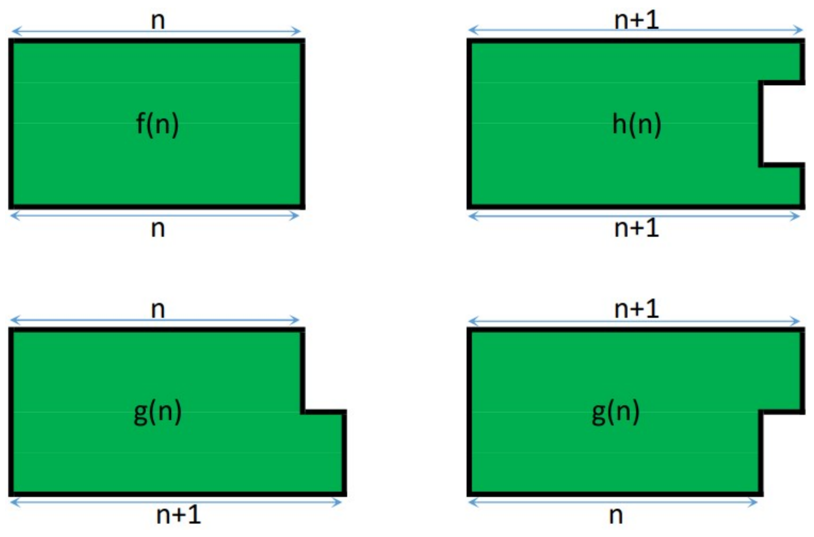


Figure 4: Defining $f(n)$, $g(n)$ and $h(n)$

Step 2 - Recursive Relation:
We have a mutual recursion relationship between $f(.)$, $g(.)$ and $h(.)$
Base Case:
$f(0) = 1,\ f(1) = 1$
$g(0) = 1,\ g(1) = 2$
$h(0) = 0,\ h(1) = 1$
Recursive definition:
$f(n) = f(n-1) + f(n-2) + 2g(n-2) + h(n-1)\ \forall\ n \geq 2$
$g(n) = g(n-1) + f(n)\ \forall\ n \geq 2$
$h(n) = h(n-2) + f(n-1)\ \forall\ n \geq 2$

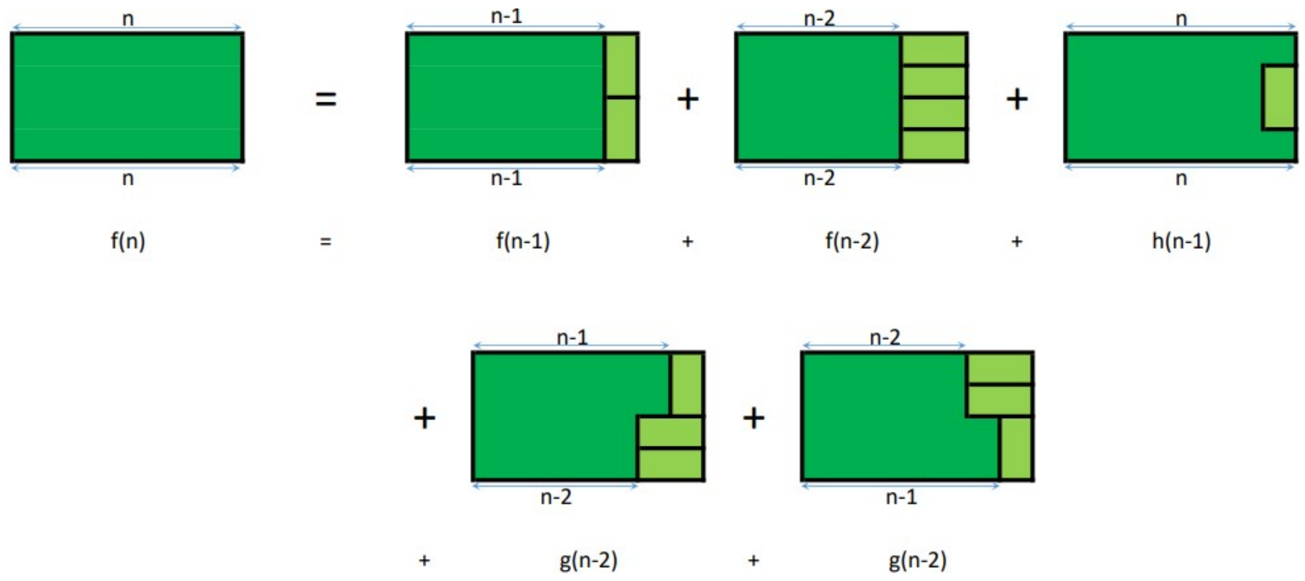


Figure 5: Proof of recursive definition of $f(n)$, for $n \geq 2$

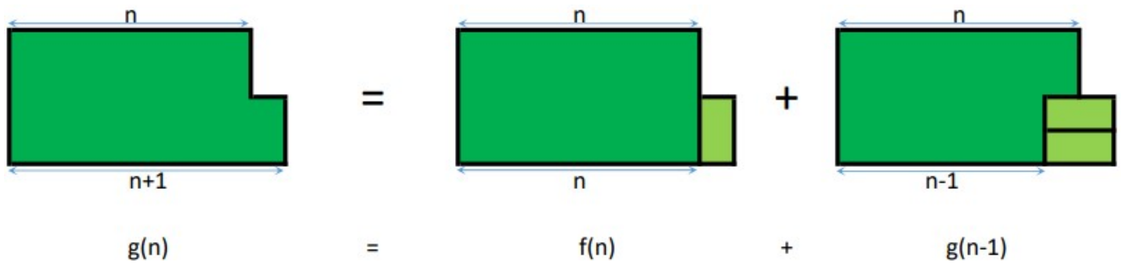


Figure 6: Proof of recursive definition of $g(n)$, for $n \geq 2$

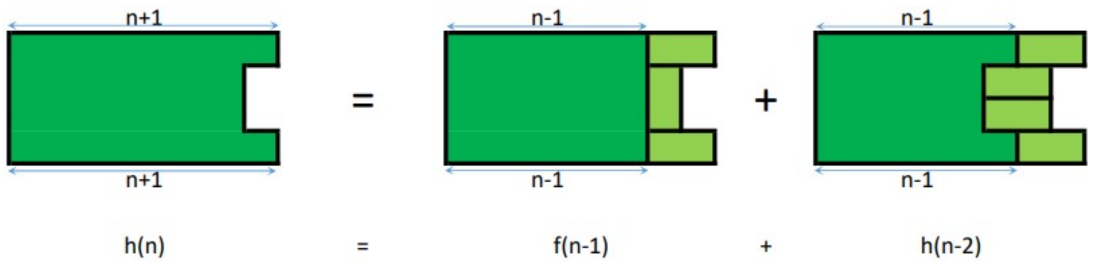


Figure 7: Proof of recursive definition of $h(n)$, for $n \geq 2$

Step 3 - Memoization Strategy:
Three arrays, one each for storing values of $f(i)$, $g(i)$ and $h(i)$, can be used for memoization. Let us call these arrays `F`, `G` and `H`, each of size n.

Step 4 - Acyclicity of Dependencies:
$f(i)$ calls $g(.)$ with index $\leq i-2$ which calls $f(.)$ with index $\leq i-2$.
$f(i)$ calls $h(.)$ with index $\leq i-1$ which calls $f(.)$ with index $\leq i-2$.
$f(i)$ calls $f(.)$ with index $\leq i-1$.
Thus, $f(i)$ would never call itself.
$g(i)$ calls $f(.)$ with index $\leq i$ which calls $g(.)$ with index $\leq i-2$.

$g(i)$ calls $g(.)$ with index $\leq i-1$.
Thus, $g(i)$ would never call itself.
$h(i)$ calls $f(.)$ with index $\leq i-1$ which calls $h(.)$ with index $\leq i-2$.
$h(i)$ calls $h(.)$ with index $\leq i-2$.
Thus, $h(i)$ would never call itself.

Step 5 - Time Complexity Analysis:
The total number of sub-problems is $\mathcal{O}(n)$ (including both $f(.)$, $g(.)$ and $h(.)$.
The time taken per sub-problem is $\mathcal{O}(1)$ (basic arithmetic operations on integers).
Total time = $\mathcal{O}(n) \times \mathcal{O}(1) = \mathcal{O}(n)$ as required.

Psuedo Implementation:

Algorithm 5: CountWays4xn(n)

```
Input: $n$, integer
Output: Number of ways of tiling a $4 \times n$ rectangle using $2 \times 1$ tiles
Def $f$(i, F, G, H):
    if F[i] is defined then
        return F[i]
    end
    F[i] ← $f(i-1,F,G,H) + f(i-2,F,G,H) + 2 * g(i-2,F,G,H) + h(i-1,F,G,H)$
    return F[i]

Def $g$(i, F, G, H):
    if G[i] is defined then
        return G[i]
    end
    G[i] ← $g(i-1,F,G,H) + f(i,F,G,H)$
    return G[i]

Def $h$(i, F, G, H):
    if H[i] is defined then
        return H[i]
    end
    H[i] ← $h(i-2,F,G,H) + f(i-1,F,G,H)$
    return H[i]

F, G, H empty array initialized
F[0] ← 1
G[0] ← 1
H[0] ← 0
F[1] ← 1
G[1] ← 2
H[1] ← 1

return $f(n,F,G,H)$
```

Question 4

Problem:

Suppose you are given an undirected tree with arbitrary (positive or negative) weights assigned to the edges. Describe an $O(n)$ time algorithm to find a subtree of the tree having maximum total weight. Modify the algorithm in the case the subtree is required to be a path.

Part A: Subtree

Subproblems and Recurrences

Without loss of generality, we root the tree T at an arbitrary vertex [1]
Let MWST denote Maximum Weight SubTree
Then we can either include u in the MWST or exclude it.
Let $inc(u)$ denote the MWST if we include u
Let $exc(u)$ denote the MWST if we exclude u
If we include it, we get the following recurrence

$$inc(u) = \sum_{v \in children(u)} \max(inc(v) + weight(u, v), 0)$$

This means that we include a child v of u in the final MWST iff $inc(v) + weight(u, v)$ is positive.
If we exclude it, we get

$$exc(u) = \max_{v \in children(u)} (\max(inc(v), exc(v)))$$

This means that if we exclude u, then we consider all cases of including and excluding it's children and take the one which yields the maximum value.

Memoization Strategy

For every node, we need to have the values of it's children pre-computed. This naturally suggest a post-order traversal order. While computing the post order traversal, we compute inc, exc from the recurrence relations.

Acyclicity of Dependencies

While computing $inc(u)$ and $exc(u)$, we only use the values for it's children, which occur before u in the post-order traversal. Thus no cyclic dependencies.

[1] Page 121, Algorithms by Jeff Erickson

Algorithm/Pseudocode

```
T_root = root(T) // T_root roots the tree T at an arbitrary vertex
initialize memo array to false for all nodes
MWST(u)
    u.inc, u.exc = 0
    for v in children(u)
        if not memo[v]
            MWST(v)
        u.inc += max(v.inc + weight(u, v), 0)
        u.exc =  max(u.exc, v.inc, v.exc)
    return
MWST(root)
ans = max(root.inc, root.exc)
```

Time Complexity

The algorithm is essentially a post-order traversal of the tree, where each node is processed exactly once. Thus the complexity is $O(n)$

Part B: Subtree

Subproblems and Recurrences

Without loss of generality, we root the tree T at an arbitrary vertex
Let MWP denote Maximum Weight Path
Then we can either include u in the MWP or exclude it.
Let $inc(u)$ denote the MWP if we include u
Let $exc(u)$ denote the MWP if we exclude u
If we include it, we get the following recurrence

$$m_1, m_2 = maxTopTwo_{v \in children(u)} \max(inc(v) + weight(u, v), 0)$$
$$inc(u) = m_1 + m_2$$

This means we find the 2 max weight paths that start at u and join them at u iff including them increases the weight of the MWP that includes u
If we exclude it, we get

$$exc(u) = \max_{v \in children(u)} (\max(inc(v), exc(v)))$$

This means that if we exclude u, then we consider all cases of including and excluding it's children and take the one which yields the maximum value.

Memoization Strategy

For every node, we need to have the values of it's children pre-computed. This naturally suggest a post-order traversal order. While computing the post order traversal, we compute inc, exc from the recurrence relations.

Acyclicity of Dependencies

While computing $inc(u)$ and $exc(u)$, we only use the values for it's children, which occur before u in the post-order traversal. Thus no cyclic dependencies.

Algorithm/Pseudocode

```
T_root = root(T) // T_root roots the tree T at an arbitrary vertex
initialize memo array to false for all nodes
MWP(u)
    u.inc, u.exc, m1, m2 = 0
    for v in children(u)
        if not memo[v]
            MWP(v)
        temp = v.inc + weight(u, v)
        if temp >= m_1
            m_2 = m_1
            m_1 = temp
        else
            if temp > m_2
                m_2 = temp
        u.exc = max(exc[u], inc[v], exc[v])
    u.inc = m_1 + m_2
    return
MWP(root)
ans = max(root.inc, root.exc)
```

Time Complexity

The algorithm is essentially a post-order traversal of the tree, where each node is processed exactly once. Thus the complexity is $O(n)$

Question 5

(a)

Problem: Given a sequence of distinct numbers $a_1, a_2, \ldots, a_n$ with arbitrary weights $w_1, w_2, \ldots, w_n$ assigned to them respectively, design an algorithm to find a maximum weight increasing subsequence.

Solution: We use dynamic programming.

1. **Types of Sub-Problems**: For all i from 1 to n, let $M(i)$ be defined as a maximum weight increasing subsequence that has a_i as its last element. Our goal is to compute $\max_{j \in [n]} M(j)$. Note that the *max* function here compares subsequences based on total weight; the larger weight subsequence is said to be greater than the smaller weight subsequence.

2. **Recursive Relation**: The recursive relation among $M(i)$ values can be expressed as follows:

   $$M(i) = \max_{j \in [i-1],\ a_j < a_i} M(j) \cdot a_i$$

   where $\cdot$ represents concatenation. This is quite an obvious relation, since if we remove a_i from any maximum weight increasing subsequence S ending at a_i, the resultant would have to be a maximum weight increasing subsequence ending at the second last element of S. If not, then we could choose a higher weight increasing subsequence ending at the previous element and thus a higher weight increasing subsequence ending at a_i, resulting in a clear contradiction.

   Note: If there is no j with $a_j < a_i$, then the max weight $M(j)$ is assumed to be the empty sequence.

3. **Memoization Strategy**: A simple array of `pair(vector,int)` can be used for memoization. Let us call this array `A`. The elements of this array would be ordered pairs such that `A[i]` is the ordered pair ($M(i)$, weight($M(i)$)). Initially, none of the $M(i)$ values is known, so the array `A` is empty. Whenever a recursive call requires us to find $M(i)$ for some i, we first check if there is a subsequence already stored in `A[i]`. If so, we use that directly. Otherwise, we compute this subsequence using the recursive relation defined above, by doing a linear search over all $M(j)$ subsequences for j smaller than i, and we finally store the computed subsequence (as a vector) and its weight in `A[i]`.

4. **Acyclicity of Dependencies**: Each call to an $M(i)$ uses only the values of $M(j)$ for j smaller than i. So, the recursive calls could never lead back to calling $M(i)$ itself, and thus we can be sure that the dependencies are acyclic.

5. **Time Complexity Analysis**: If the values of $M(j)$ and their weights for all j from 1 to $i-1$ are known, then computing $M(i)$ comprises of iterating over j from 1 to $i-1$, and whenever $a_j < a_i$, updating the maximum weight subsequence encountered so far if it has a larger weight than the current maximum. Assuming that accessing elements in an array takes constant time, this process clearly takes constant time for each j, and thus $O(i)$ time for a fixed i. Since we must calculate $M(i)$ for all i from 1 to n, the total time complexity for computing all $M(i)$'s is:

   $$\sum_{i=1}^{n} O(i) = O(n^2)$$

   Once this is done, all we have to do is iterate over i from 1 to n and find the $M(i)$ which has maximum weight. The weights are already stored in the array `A`, making this easily achievable in $O(n)$ time.

   So, the overall time complexity of the algorithm is $O(n^2)$.

The pseudocode to compute $M(i)$ for any given i is shown below:

Algorithm 6: LIS($\{a_j\}_{j=1}^n, \{w_j\}_{j=1}^n, i, A$)

```
Input: distinct numbers $\{a_j\}_{j=1}^n$, corresponding weights $\{w_j\}_{j=1}^n$, index $i$, memoization array $A$
Output: largest weight increasing subsequence of $\{a_j\}_{j=1}^n$ ending at index $i$, and its weight
if $A[i]$ is not null then
    $(S, w) \leftarrow A[i]$
else
    $S \leftarrow \{a_i\}$
    $w \leftarrow w_i$
    for $k = 1$ to $i - 1$ do
        if $a_k < a_i$ then
            $(S', w') \leftarrow$ LIS($\{a_j\}_{j=1}^n, \{w_j\}_{j=1}^n, k, A$)
            if $w' + w_i > w$ then
                $S \leftarrow S' \cdot a_i$
                $w \leftarrow w' + w_i$
            end
        end
    end
    $A[i] \leftarrow (S, w)$
end
return $(S, w)$
```

We use the above algorithm to first compute $M(i)$ values for all i, and then finding the $M(i)$ having the largest weight simply takes a linear search over the array A.

(b)

Problem: Given a sequence of distinct numbers $a_1, a_2, \ldots, a_n$ with arbitrary weights $w_1, w_2, \ldots, w_n$ assigned to them respectively, design an algorithm to find a maximum weight subsequence such that no three consecutive numbers in the original sequence belong to the subsequence.

Solution: Once again, we use dynamic programming.

1. **Types of Sub-Problems**: For all i from 1 to n, let $N(i)$ denote the maximum weight subsequence of $(a_1, a_2, \ldots, a_i)$, including no three consecutive elements into the subsequence. Our goal is to compute $N(n)$, that is, the maximum weight subsequence of the entire sequence that does not contain any three consecutive elements from the original sequence $\{a_j\}_{j=1}^n$. Also, let $z(i)$ be defined as the weight of $N(i)$.

2. **Recursive Relation**: For any $i > 3$, we form our recursive relation based on the three elements a_i, a_{i-1}, and a_{i-2}. Clearly, we cannot include all three of them in our subsequence, as they're consecutive elements. We must exclude at least one, which leads us to the following cases:

   (a) a_{i-2} is excluded. In this case, $z(i) = z(i-3) + w_{i-1} + w_i$.
   (b) a_{i-1} is excluded. In this case, $z(i) = z(i-2) + w_i$.
   (c) a_i is excluded. In this case, $z(i) = z(i-1)$.

   Since we want the subsequence having maximum weight, we reach the following recursive relation:

$$z(i) = \max\{z(i-1), z(i-2) + w_i, z(i-3) + w_{i-1} + w_i\}$$

Thus, we have:

$$N(i) = \max\{N(i-1), N(i-2) \cdot a_i, N(i-3) \cdot a_{i-1} \cdot a_i\}$$

where max is taken on the basis of weights of the subsequences, and $\cdot$ represents concatenation.

The base cases for this recursion are:

- $N(1) = (a_1)$
- $N(2) = (a_1, a_2)$
- $N(3) = \max\{(a_1, a_2), (a_2, a_3), (a_1, a_3)\}$

3. **Memoization Strategy**: Just as in part (a), we can use a simple array of `pair(vector,int)` for memoization. Suppose we call this array `B`. Then, for all i from 1 to n, `B[i]` is used to store the pair $(N(i), z(i))$. Whenever a recursive call requires the value of $N(i)$ for some i, we check if there is a pair already present in `B[i]`. If there is, then we use $N(i)$ directly from the array. If not, then $N(i)$ is computed recursively, and then stored in the array in a pair, as a vector along with its weight $z(i)$, for use in the future if required.

4. **Acyclicity of Dependencies**: Any call to $N(i)$ depends only on the values of $N(i-1), N(i-2), N(i-3)$, that is, it depends on $N(j)$ for $j < i$. Thus, a recursive call can never reach back to $N(i)$ itself, as the argument must always decrease. This guarantees acyclicity of dependencies.

5. **Time Complexity Analysis**: If the values of $N(i-1), N(i-2), N(i-3)$ are known, along with their weights, then computing $N(i)$ consists of comparing the weights of $N(i-1), N(i-2) \cdot a_i$, and $N(i-3) \cdot a_{i-1} \cdot a_i$. Assuming that accessing array elements is a constant time operation, it is easy to see that computing $N(i)$ from $N(i-1), N(i-2), N(i-3)$ takes constant time, since it essentially requires us to just find the maximum of three numbers. Since the same constant time operation has to be performed for all i from 1 to n, the time complexity for calculating all $N(i)$ values is:

   $$\sum_{i=1}^{n} O(1) = O(n)$$

   Once we've done this, $N(n)$ gives us the required solution to the problem, and can be accessed in constant time.

   So, the overall time complexity of the algorithm is $O(n)$.

The pseudocode to compute $N(i)$ for any given i is shown on the next page. To solve the given problem, we simply call $N(n)$.

Algorithm 7: NCS($\{a_j\}_{j=1}^n, \{w_j\}_{j=1}^n, i, B$)

Input: distinct numbers $\{a_j\}_{j=1}^n$, corresponding weights $\{w_j\}_{j=1}^n$, index i, memoization array B

Output: largest weight subsequence of $\{a_j\}_{j=1}^i$ with no 3 consecutive elements of original seq

```
if $B[i]$ is not null then
    $(S, w) \leftarrow B[i]$
else if i = 1 then
    $S \leftarrow \{a_1\}$
    $w \leftarrow w_1$
    $B[i] \leftarrow (S, w)$
else if i = 2 then
    $S \leftarrow \{a_1, a_2\}$
    $w \leftarrow w_1 + w_2$
    $B[i] \leftarrow (S, w)$
else if i = 3 then
    $k \leftarrow argmax\{w_1 + w_2, w_2 + w_3, w_1 + w_3\}$
    if k = 1 then
        $S \leftarrow \{a_1, a_2\}$
        $w \leftarrow w_1 + w_2$
    else if k = 2 then
        $S \leftarrow \{a_2, a_3\}$
        $w \leftarrow w_2 + w_3$
    else
        $S \leftarrow \{a_1, a_3\}$
        $w \leftarrow w_1 + w_3$
    end
    $B[i] \leftarrow (S, w)$
else
    $(S'_1, w'_1) \leftarrow$ NCS($\{a_j\}_{j=1}^n, \{w_j\}_{j=1}^n, i-1, B$)
    $(S'_2, w'_2) \leftarrow$ NCS($\{a_j\}_{j=1}^n, \{w_j\}_{j=1}^n, i-2, B$)
    $(S'_3, w'_3) \leftarrow$ NCS($\{a_j\}_{j=1}^n, \{w_j\}_{j=1}^n, i-3, B$)
    $k \leftarrow argmax\{w'_1, w'_2 + w_i, w'_3 + w_{i-1} + w_i\}$
    if k = 1 then
        $S \leftarrow S'_1$
        $w \leftarrow w'_1$
    else if k = 2 then
        $S \leftarrow S'_2 \cdot a_i$
        $w \leftarrow w'_2 + w_i$
    else
        $S \leftarrow S'_3 \cdot a_{i-1} \cdot a_i$
        $w \leftarrow w'_3 + w_{i-1} + w_i$
    end
    $B[i] \leftarrow (S, w)$
end
return $(S, w)$
```