2 Saturday
FEBRUARY

033-332 Week 5

DYNAMIC PROGRAMMING

Two ways:

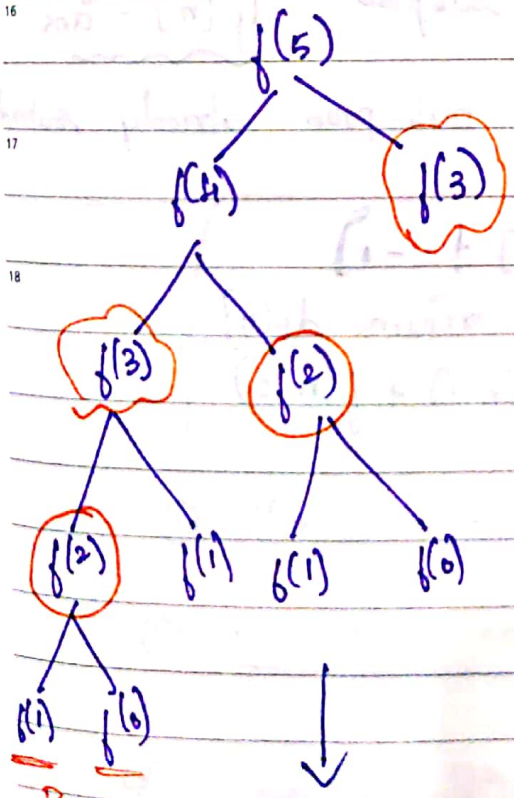
- ✓ Tabulation → Bottom up approach (usually from start to end 0 - n)
- ✓ Memoization → Top-down approach (often from end to start)

(1) → Fibonacci 0, 1, 1, 2, 3, 5, 8, 15, 21 . . .

(addition of previous 2 nos)

$$f(n) = f(n-1) + f(n-2)$$

Recursion Tree (n=5)



Overlapping sub-problems happen here

↓

How to avoid? => MEMOIZATION

(store values of sub-prob in some map/table)

3 Sunday

034-331 Week 5

=> If your sub-problem returns a single value, then store in 1D-Array

The sub-problems here are f(0), f(1), f(2), f(3), f(4), f(5)

(1) (2) (3) (4) (5) (6)

Create table of (n+1) size

2019

Base

MARCH 2019

W	M	T	W	T	F	S	S
9					1	2	3
10	4	5	6	7	8	9	10
11	11	12	13	14	15	16	17
12	18	19	20	21	22	23	24
13	25	26	27	28	29	30	31

Create 1D array of size (n+1) ⇒ dp[] = | -1 | -1 | -1 | -1 | -1 | -1 |
0 1 2 3 4 5

Initially store with -1 & update the values post each recursion call

~~f(0)~~ → returns 0 (store in dp)
~~f(1)~~ → returns 1 (store in dp)
✓ f(2) → returns 1 (store in dp)
✓ f(3) → returns 2 (store in dp)

Recursive Code

```
f(n) {
  if (n<=1)
    return n;
  return f(n-1) + f(n-2);
}
```

f(n-1) + f(n-2); → sub-problem (return to dp array)

MEMOIZATION

① declare dp[n+1]

② f(n-1) + f(n-2) → store in dp
value of subprob
dp[n] = ans

③ check if sub-prob already solved

check ←

```
if (dp[n] != -1)
    return dp[n];
return f(n-1) + f(n-2)
```

2019

MEMOIZATION CODE

Tc: O(N)
Sc: O(N) + O(N)
recursion stack ↓ Array

```
main() {
    int[] dp = new int[n+1];
    Arrays.fill(dp, -1);   ↳ declare Arr & fill (-1)
    fib(n, dp);
}
int fib(int n, int[] dp) {
    if (n<=1)
        return n;          → Base condn

    if (dp[n] != -1)       } → check if val is already calculated;
        return dp[n];
    return dp[n] = fib(n-1, dp) + fib(n-2, dp);
}
```

↓ store in dp array & then return

×

Recursion → Tabulation (Bottom-up)

Base Condition to the Required

Tc: O(N)
Sc: O(N)
↓ only Array

① Declare dp[n+1]

② Fill Base Case

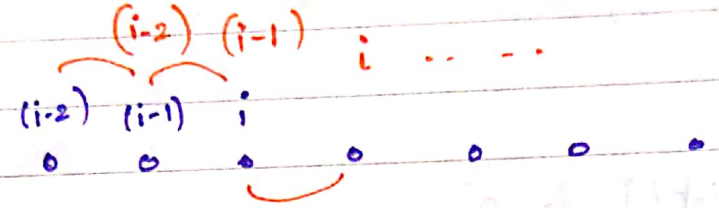
```
dp[0] = 0
dp[1] = 1
```

③ Implement Recursion Equation (if n>1; from 2)

```
for(i=2; i<=n; i++) {
    dp[i] = dp[i-1] + dp[i-2];
}
```

Space optimization in Tabulation

What we require to solve i ? } => i-1 & i-2



Each time, 2 variables will be required for calculating i

(i-2) (i-1)

① prev2 = 0 prev = 1

② for (i=2 ; i<=n ; i++) {

cur = prev + prev2 ;

prev2 = prev

prev = curi

}

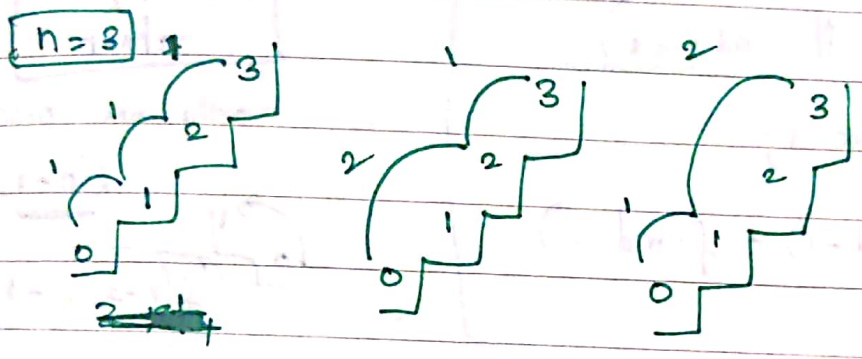
print prev → end of for loop at (n+1), so cur points at n+1 hence return prev

TC: O(N)
SC: O(1)

L2 → CLIMBING STAIRS

Gn: n (no. of stairs), initially at 0th step & have to reach nth step

Each time → climb 1/2 steps, return no. of distinct ways to climb steps



o/p => 3 ways

① Count total no. of ways // all possible ways
② Multiple ways possible: return min // max

→ Think as Recursion followed by DP

Soln Recursion using DP

i) Try to represent in terms of index.

ii) Do all possible stuff on that index acc to pbm statement

iii) Count all ways → (sum of all stuffs)

find min → (min (of all stuffs))

⇒ This prob is same as fibonacci

little change in base case ; `n == 0; return 1`

only one possibility

```
f(ind) {
    if (ind == 0 || ind == 1)
        return 1;
    return f(ind-1) + f(ind-2)
}
```

`n == 1; return 1`

only one possibility

f(n-1)

f(n-2) = -1 X

X

* Represent in terms of index f(3)

+ Try all possible ways : can jump 1 step / 2 step

```
lh = f(ind-1)
rh = f(ind-2)
```

* All possible shifts

⇒ return (lh + rh)

9 Saturday
FEBRUARY

(13) Frog Jump

Gn → Frog on 1st step of floor, want to reach the nth stair

Height [i] is the height of (i+1)th stair,
energy lost if jump from i to j → ht(i-1) - ht(j-1)
Can jump either +1 / +2 steps. Find min total energy to reach from 1st to nth stair

Eg ⇒ 10 20 30 10 to reach from (0 → 3)
idx 0 1 2 3 idx

Possibilities

① 10 → 10 → 20 → 10 = 40
10 20 30 10

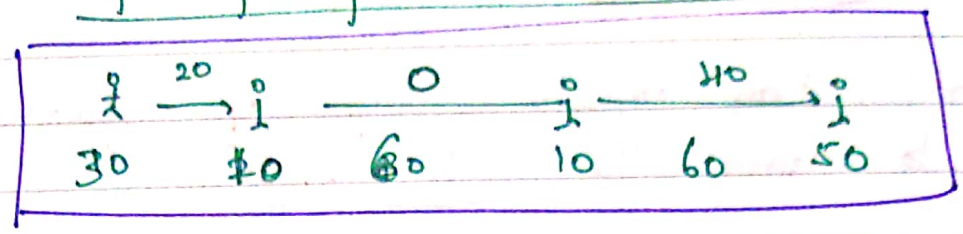
② 10 → 10 → 10 = 20 → Min.
10 20 30 10

③ 20 → 20 → 10 = 40
10 20 30 10

Here, we are trying all possible ways → Use Recursion

Why greedy doesn't work here

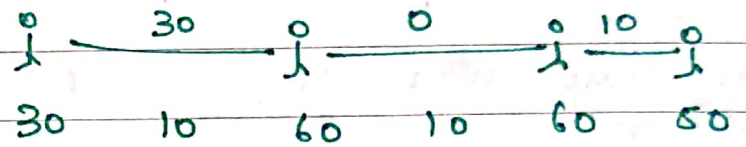
min in taking each step, doesn't work



= (60) greedy ans

⇒ Moving in min energy loss in each step.

BUT

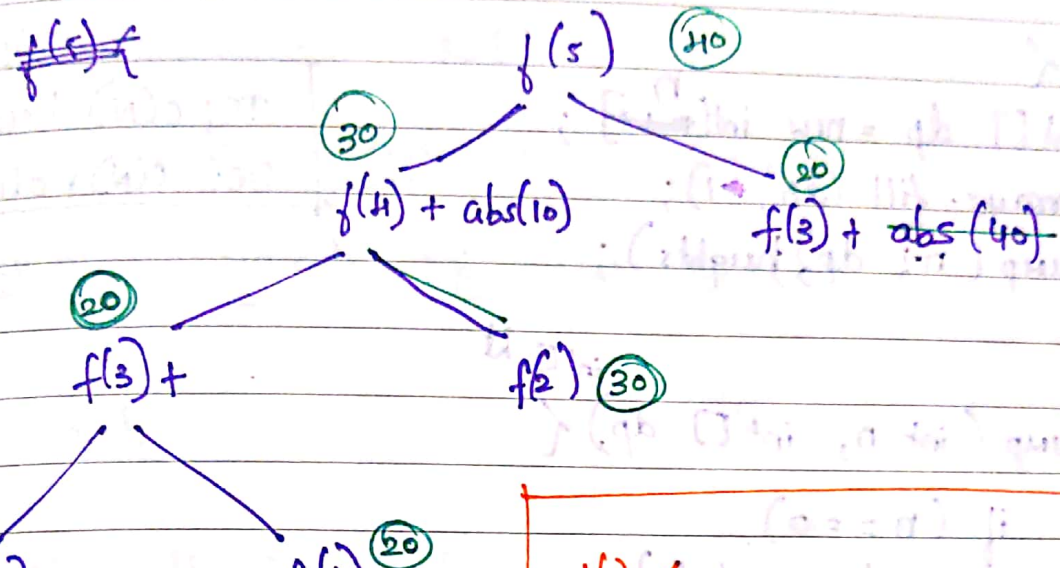


= (40) ↓ Best Ans

✓ Use Recursion ⇒ try all possible ways
✓ Among poss ways ⇒ find min

```
f(ind) {
    if (ind == 0)
        return 0;
    int lh = f(ind-1) + abs(arr[ind] - arr[ind-1]);
    if (ind > 1)
    int rh = f(ind-2) + abs(arr[ind] - arr[ind-2]);
    return Math.min(lh, rh);
}
```

[30, 10, 60, 10, 60, 50] n=6



f(1) { 0+20
left = f(0) + abs(30-10)
rh = max
return min(20, max);
}

f(1) = 20

f(2) { (20+50 = 70)
left = f(1) + abs(50)
right = f(0) + abs(30) (0+30)
return min(70, 30)
}

f(2) = 30

f(3) { (30+50 = 80)
left = f(2) + abs(50)
right = f(1) + abs(0) (20+0 = 20)
return min(80, 20)
}

f(3) = 20

f(4) {
left = f(3) + abs(50) 20
rig = f(2) + abs(0) 30
return min(70, 30)
}

f(4) = 30

f(5) { 30
left = f(4) + abs(
rig = f(3) + abs(
20
return min(
}

f(5) = 40

MEMOIZATION

```
main() {
    int[] dp = new int[n+1];
    Arrays.fill(dp, -1);
    jump(n-1, dp, heights);
}

int jump(int n, int[] dp, int[] ht) {
    if (n == 0)
        return dp[n] = 0;
    if (dp[n] != -1) return dp[n];
    int left = jump(n-1, dp) + abs(ht[n] - ht[n-1]);

    int right = INT_MAX;
    if (n > 1)
        right = jump(n-2, dp) + abs(ht[n] - ht[n-2]);

    return dp[n] = Math.min(left, right);
}
```

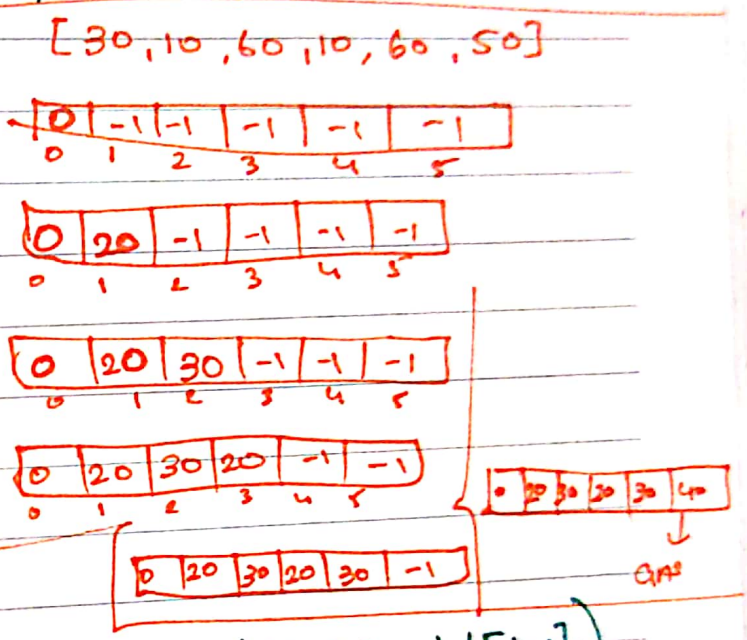
TC: O(N) linear
SC: O(N) + O(N)

TABULATION — Bottom-up

Step 1: Declare dp[] array

Step 2: Base Case → dp[0] = 0

Step 3: Iterate for next steps



Code

```
int[] dp = new int[n];
Arrays.fill(dp, -1);
dp[0] = 0;
for (int i = 1; i<n; i++) {
    int l = dp[i-1] + Math.abs(ht[i] - ht[i-1]);
    int r = Int_MAX;
    if (i>1)
        r = dp[i-2] + Math.abs(ht[i], ht[i-2])
    dp[i] = Math.min(l, r);
}
SOP(dp[n-1]);
}
```

Further you can space optimize using 2 Var concept

18 Monday FEBRUARY

Week 8 049-316

(15) Max Sum of Non-adjacent elements

Gn: Array of n integers → Return the max sum of subsequence,

In subsequence, ele's can't be adj.

i/p ⇒ 2 1 4 9

o/p ⇒ 2 9 (2, 9)

Approach → Try all sub-sequences with gn Constraint & pick the one with Max sum

(RECURSION)

pick | not pick

Recursion Approach

* Every index ⇒ two options
* pick & move to (i+2) element (because no adj)
  If picked idx 0 → move to idx 2
* not pick & move to (i+1) element
  If not picked idx 0 → pick idx 1

050-315 ○ Week 8

Recurrence Relation:

i) Represent in terms of index:

$f(idx)$ = max sum of subsequence from idx 0 to idx last

ii) Try all choices .. pick / non-pick (keeping in mind non-adj elements)

- ✓ pick → $arr[idx] + f(idx-2)$
- ✓ non-pick → $0 + f(idx-1)$

→ $f(idx-2)$ ↓ because no adj eles

iii) Return max of all choices (return max of 2 choices: pick/non-pick)

BASE CONDN

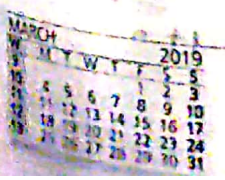
```
if (idx == 0)
    return arr[idx]
```

When will we reach ind 0? → only if we ignored ind 1

So Consider idx 0

```
if (idx < 0)
    return 0
```

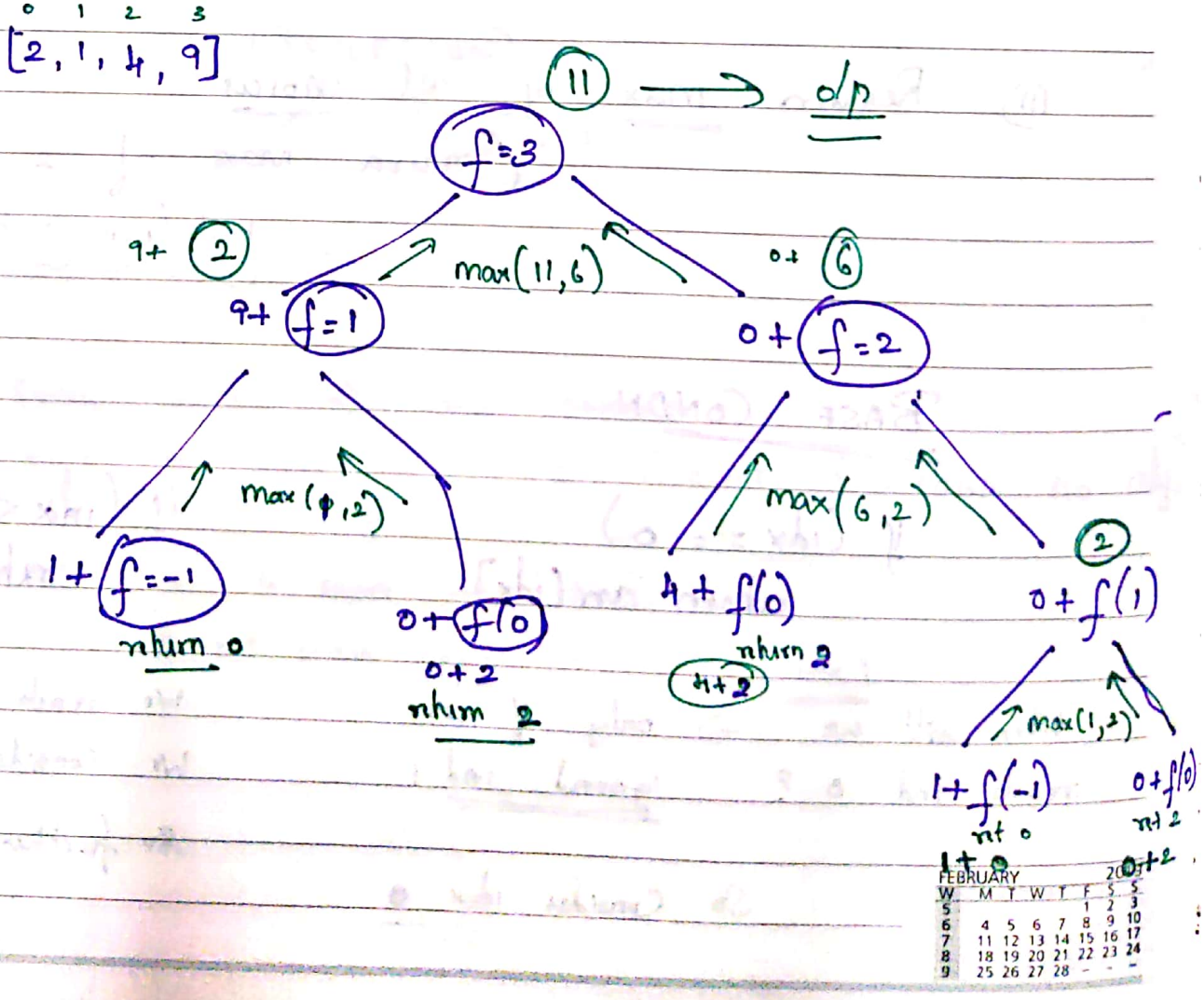
We reach here when we considered idx 1, so further can be ignored



PSEUDO - CODE

```
f(ind, arr[]) {
    if (ind == 0)
        return arr[ind];
    if (ind < 0)
        return 0;

    int pick = arr[ind] + f(ind-2, arr);
    int notpick = 0 + f(ind-1, arr);
    return Math.max(pick, non-pick);
```



MEMOIZATION → (overlapping problems)

```
int main (int n, int[] arr) {
    int[] dp = new int[n];
    Arrays.fill(dp, -1);
    Solve(n-1, arr, dp);
}

public int Solve(int ind, int[] arr, int[] dp) {
    if (ind == 0)
        return arr[ind];
    if (ind < 0)
        return 0;
    if (dp[ind] != -1) → return dp[ind];
    int pick = arr[ind] + Solve(ind-2, arr, dp);
    int notpick = 0 + Solve(ind-1, arr, dp);
    return dp[ind] = Math.max(pick, notpick);
}
```

① Represent in index

② Base cond

③ check if already calculated

④ Before return, store it

TC: O(N)
SC: O(N) + O(N)

TABULATION

```
int[] dp = new int[n];
dp[0] = arr[0];
int neg = 0;

for (int i=1; i<n; i++) {
    take = arr[ind];
    if (i>1)
        take += dp[i-2];

    notTake = 0 + dp[i-1];

    dp[i] = max(take, notTake);
}
```

Base ← dp[0] = arr[0]; int neg = 0;

only if i>1 do (i-2) ← if (i>1) take += dp[i-2];

for max of both ← dp[i] = max(take, notTake);