Introduction to Dynamic Programming 1

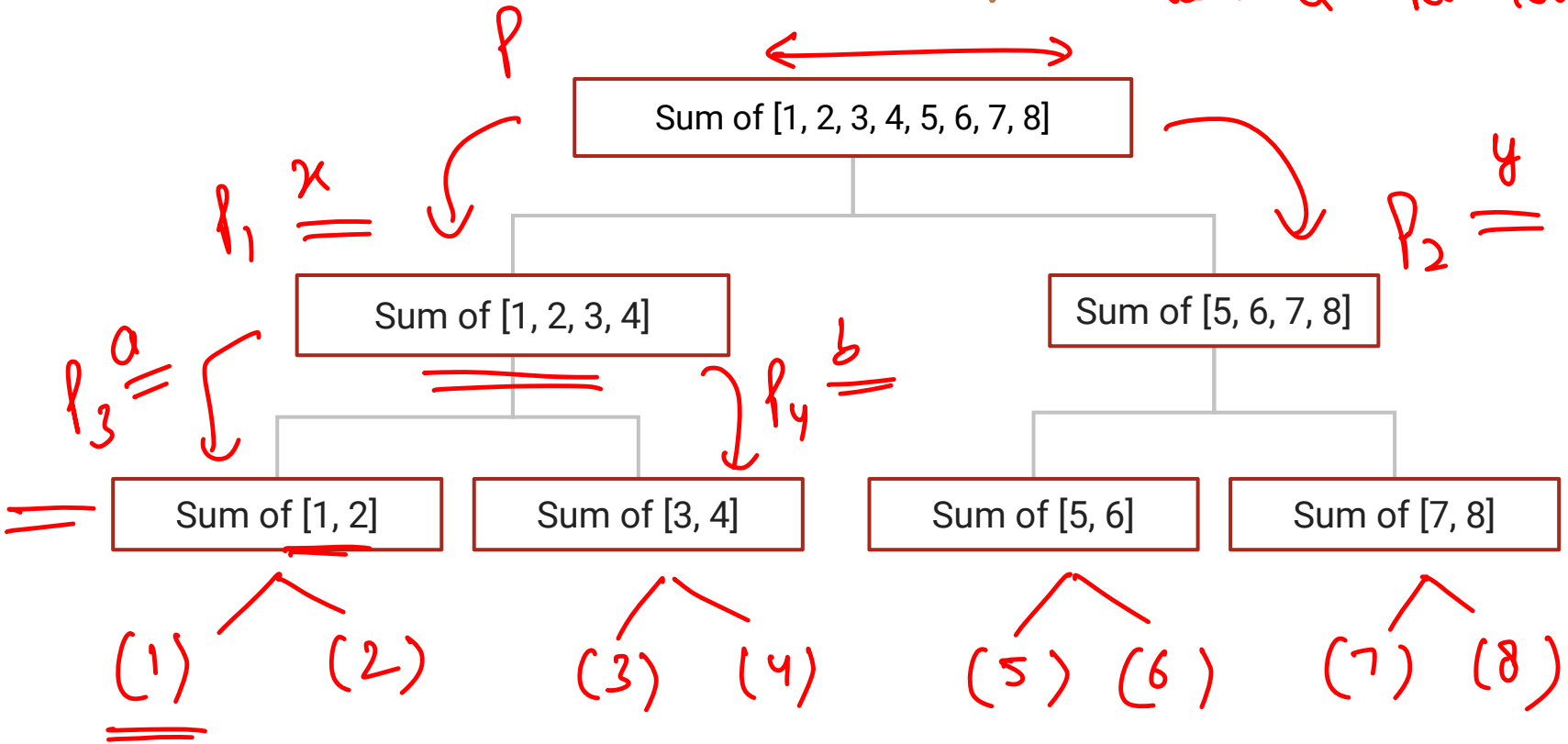
- Priyansh Agarwal

Divide & Conquer

- Given a big or hard problem, break it down into smaller or easier subproblems.
  - There must be a way to use the answers of the smaller subproblems to construct the answer of the bigger subproblem.
- Continue breaking down the subsequent subproblems into smaller subproblems until you end up at a trivial subproblem.

Sum of elements in array

not allowed to use a for loop



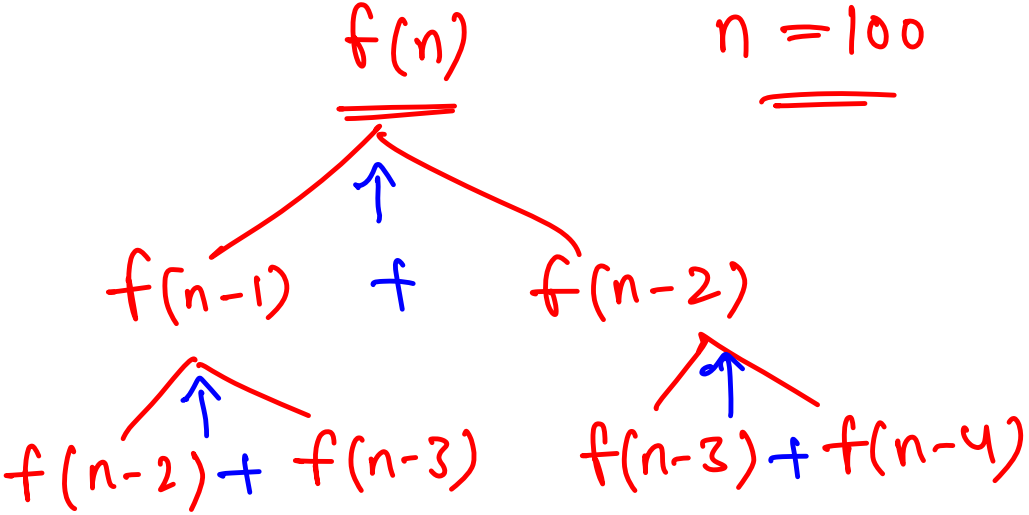
Fibonacci Problem

[1, 1, 2, 3, 5, 8, 13 ....]

- Find n^{th} fibonacci number
  - F(n) = F(n - 1) + F(n - 2)
  - F(1) = F(2) = 1

Example:

- F(5) = F(4) + F(3)
- F(4) = F(3) + F(2)
- F(3) = F(2) + F(1)

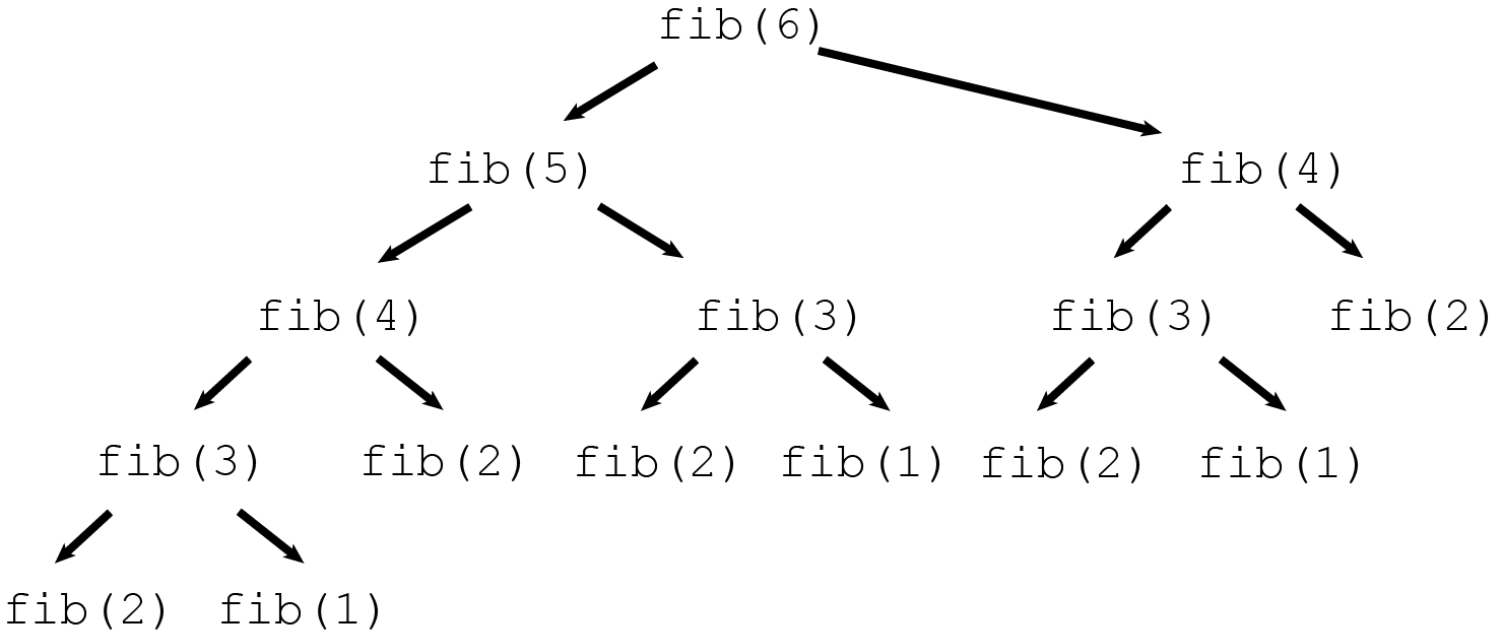


Code

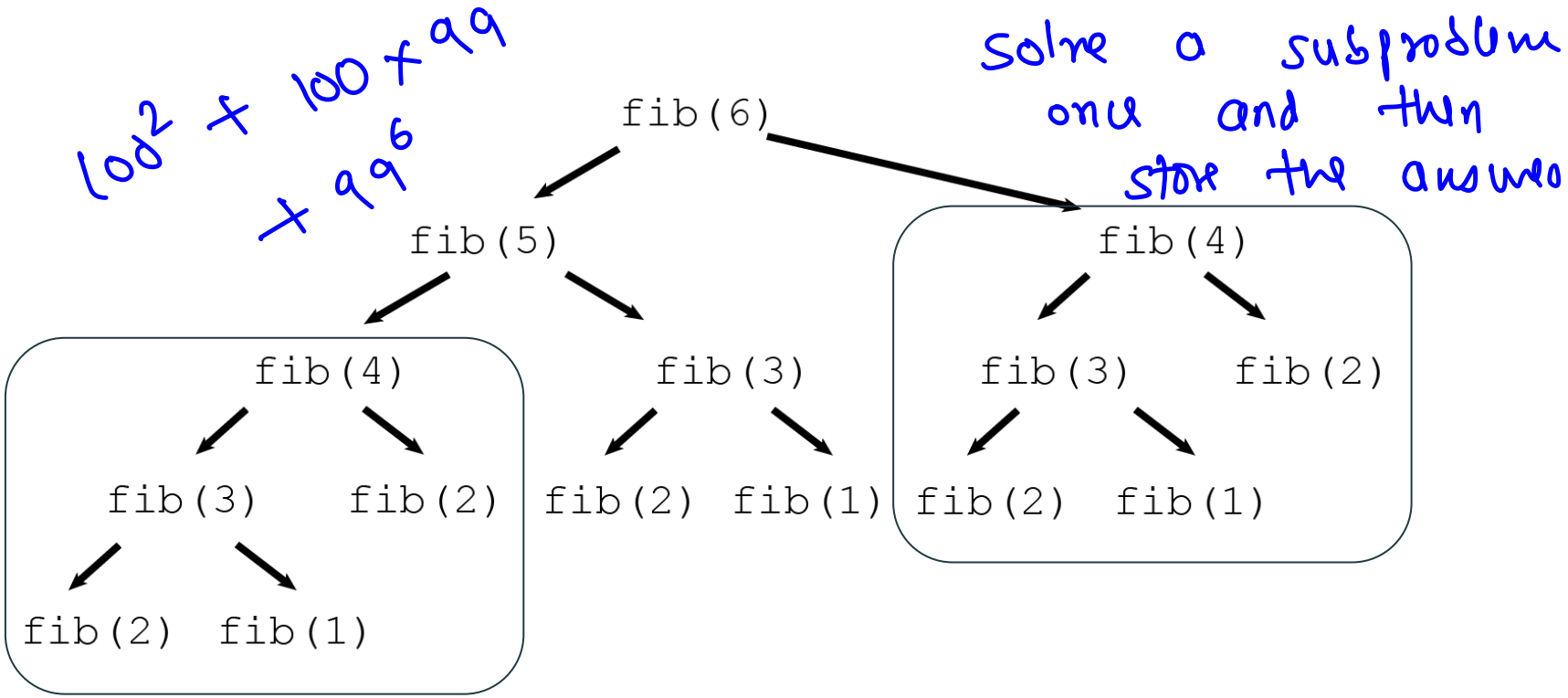
```
int functionEntered = 0;
int helper(int n){
    functionEntered++;
    if(n == 1 || n == 2){
        return 1;
    }
    return helper(n - 1) + helper(n - 2);
}
void solve(){
    int n;
    cin >> n;
    cout << helper(n) << nline;
    cout << functionEntered << nline;
}
```

work happening in 1 function call = O(1)

functionEntered = 1664079
with n = 30



Something wrong here?



Solve a subproblem once and then store the answer

Overlapping subproblems !!!

Why are we calculating fib(4) twice?

What is Dynamic Programming?

- Calculating the answers of overlapping subproblems once and using them every time they are needed.
  - It is called Memoization when writing a recursive solution
    - Similar to caching
- You must ask yourself these questions every time you do divide and conquer
  - Am I solving some subproblem again and again?
  - Can I calculate it once and store it somewhere for later use?

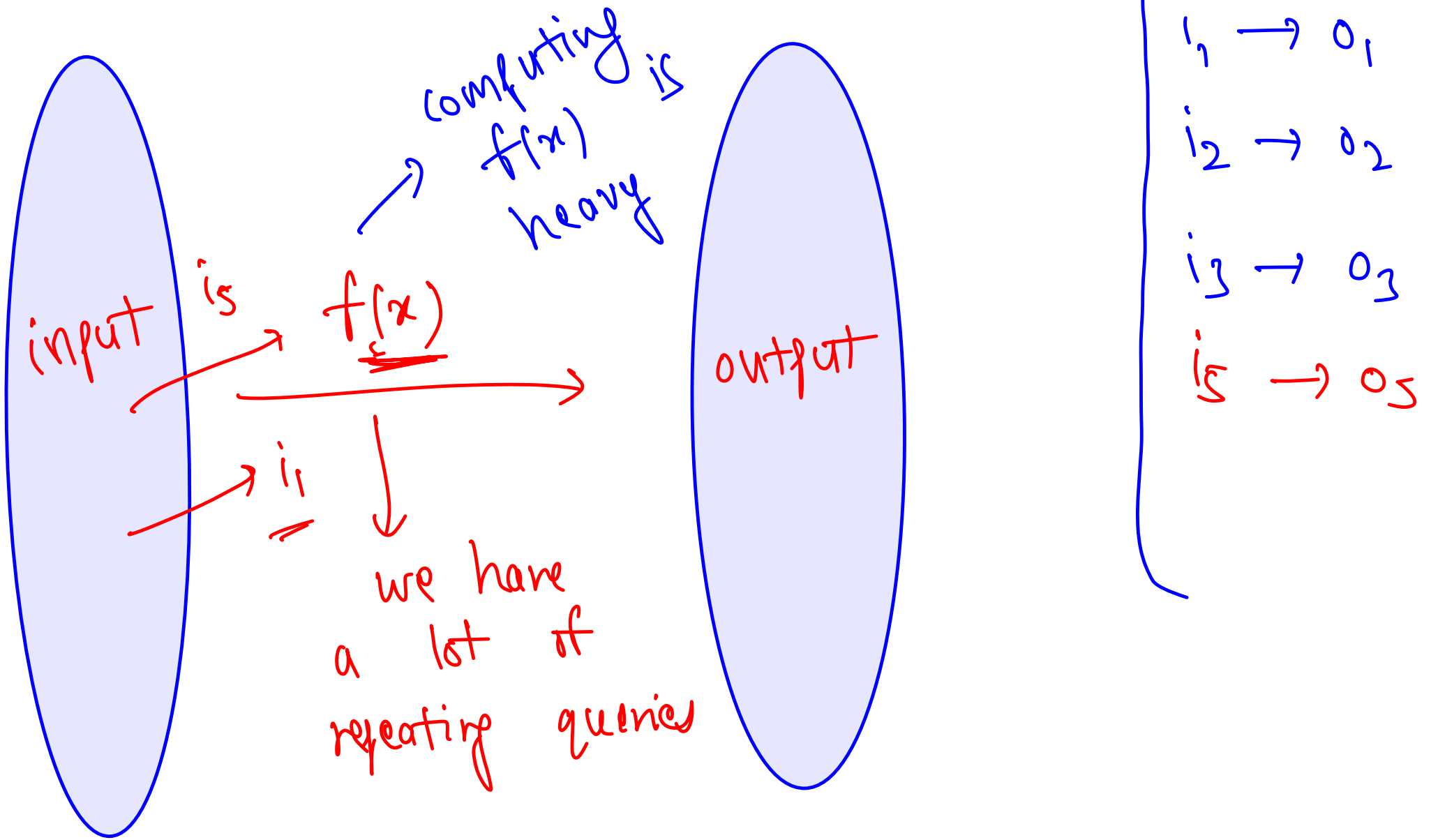
How to store the data?

map
- hash map — O(1)
- tree map — O(logn)

- You need a key, value mapping of subproblem with its answer
  - For this you need to identify every subproblem with a unique key
- You need to also find out whether you've previously solved a subproblem or not

	hash map Avg	hash map Worst
Search	O(1)	O(n)
update	O(1)	O(n)
delete	O(1)	O(n)

	tree map Avg	tree map Worst
	O(logn)	O(logn)
	"	"
	"	"



$$\begin{cases} i_1 \to o_1 \\ i_2 \to o_2 \\ i_3 \to o_3 \\ i_5 \to o_5 \end{cases}$$

map (tree map)

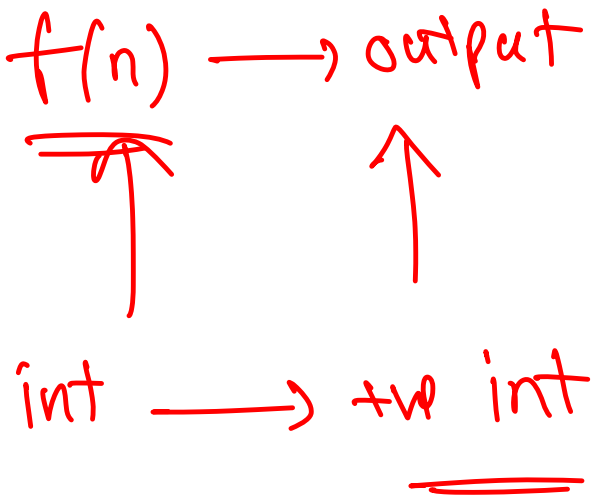
key ⟷ value

string
int
vector
set
map
double

O(log n)

fibonacci no. problem

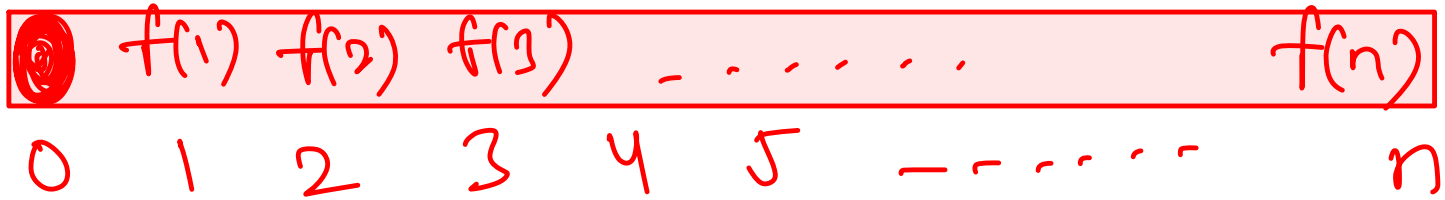
nth fibonacci



map ⟶ { 1 → f(1) , 2 → f(2) ---- n → f(n) }

O(log n)

arr ⟶



O(1)

nth fibonacci no. ⟶ n = 10^{16}

input ⟶ +ve int

⟶ input must not be v. large

map → { }

arr → | DV | DV | DV | DV | | | |

keys

f(input) → output

output ≥ 0

| -2 | -2 | -2 | -2 | -2 | -2 | -2 |

DV should be any value outside the range of the function

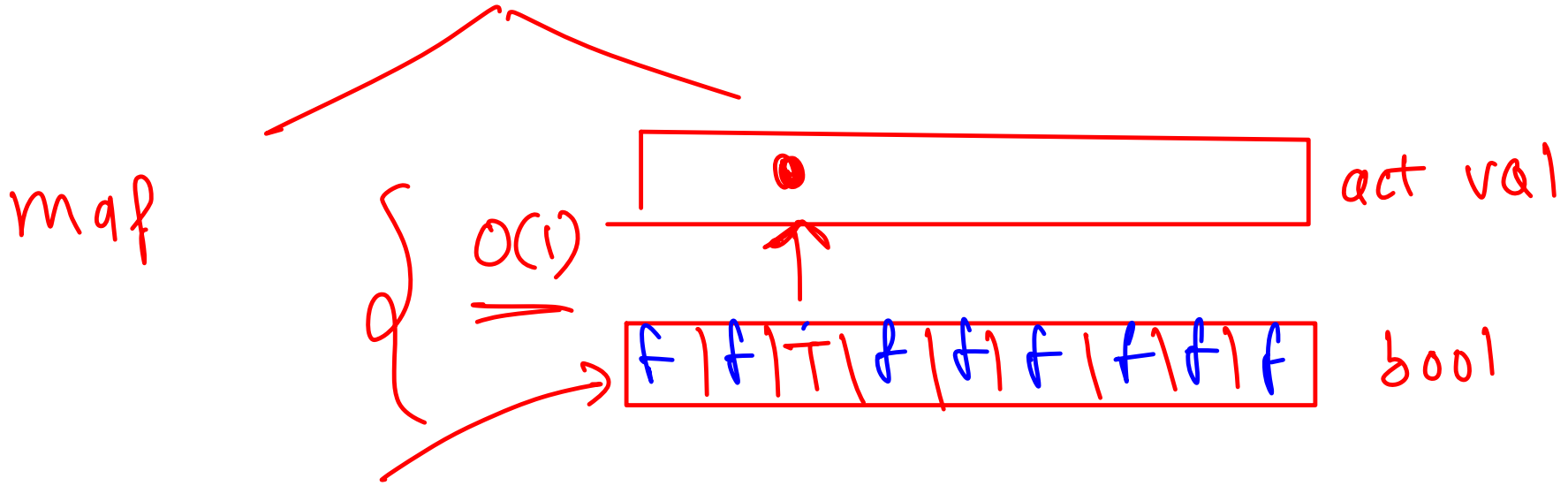
Same code with Dynamic Programming

```
int functionEntered = 0;
int dp[40];
int helper(int n){
    functionEntered++;
    if(n == 1 || n == 2){
        return 1;
    }
    if(dp[n] != -1)
        return dp[n];
    return dp[n] = helper(n - 1) + helper(n - 2);
}
void solve(){
    int n;
    cin >> n;
    for(int i = 0; i <= n; i++)
        dp[i] = -1;
    cout << helper(n) << nline;
    cout << functionEntered << nline;
}
```

functionEntered = 57
with n = 30

input → { 0 to 10^7 }

output → { -inf to inf }

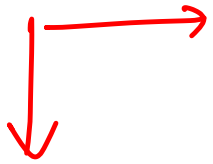


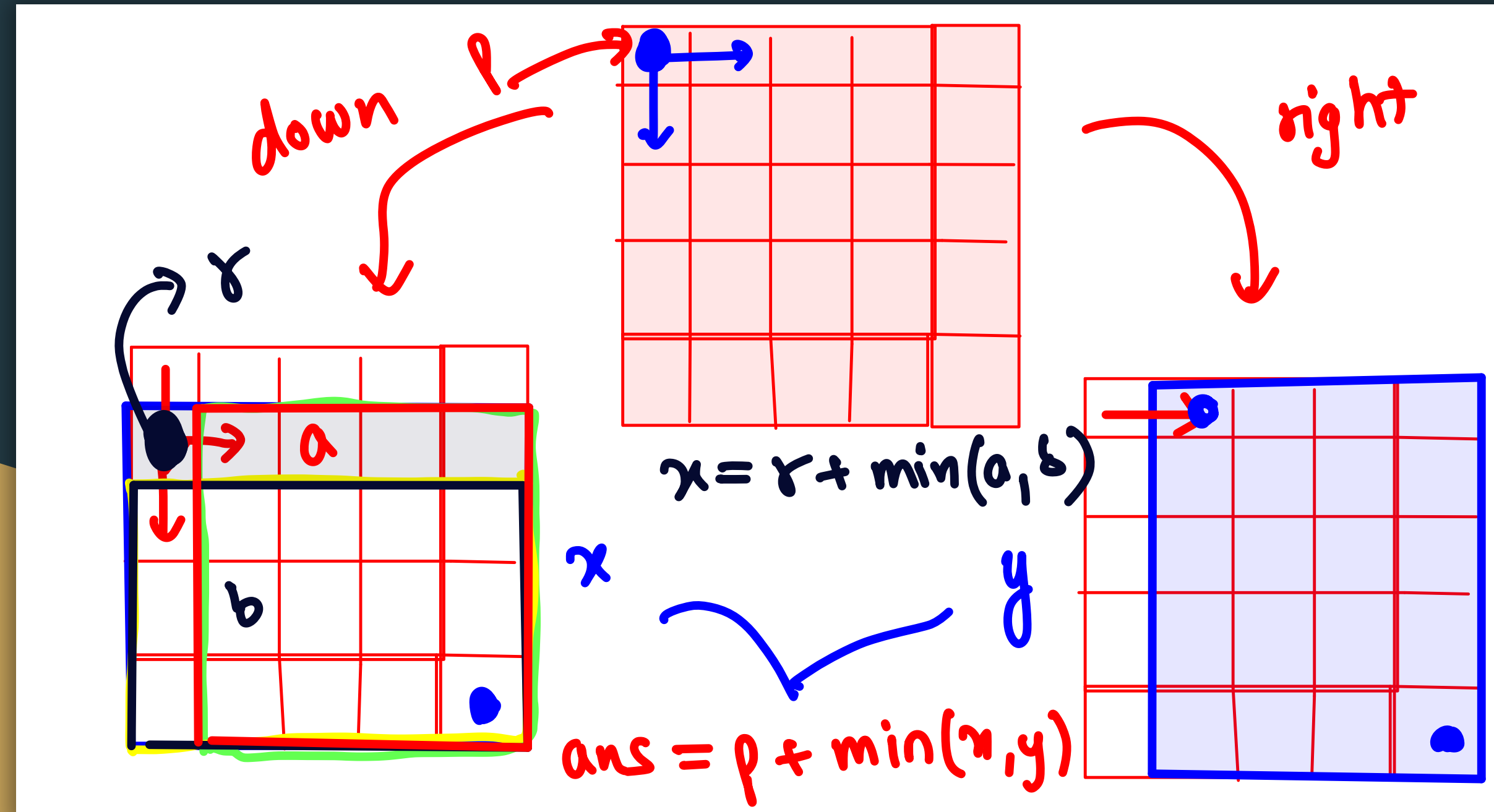
Grid Problem

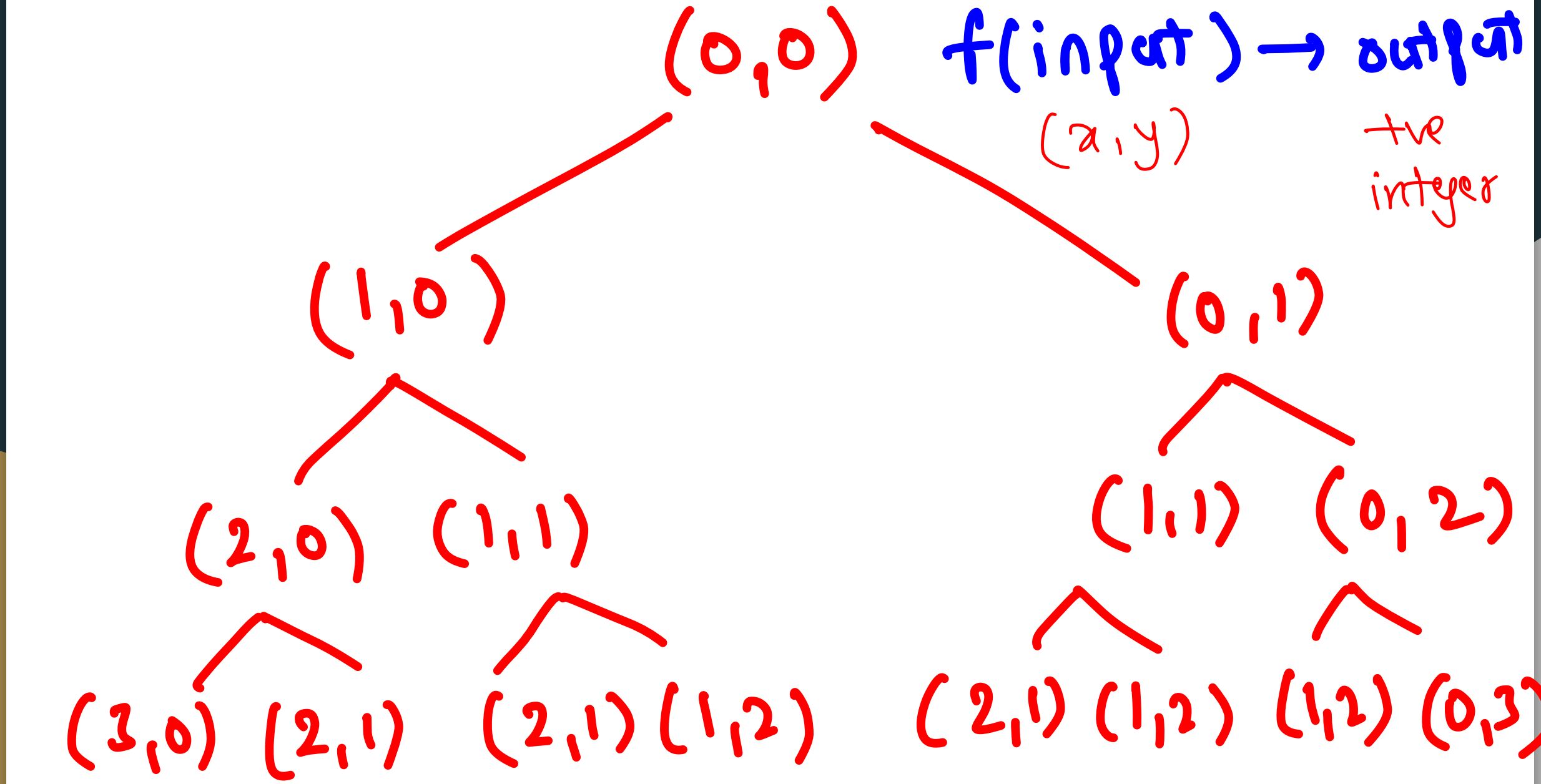
Given a 2D grid (N X N) with numbers written in each cell, find the path from top left (0, 0) to bottom right (n - 1, n - 1) with minimum sum of values on the path such that you can either go down or right from every cell.

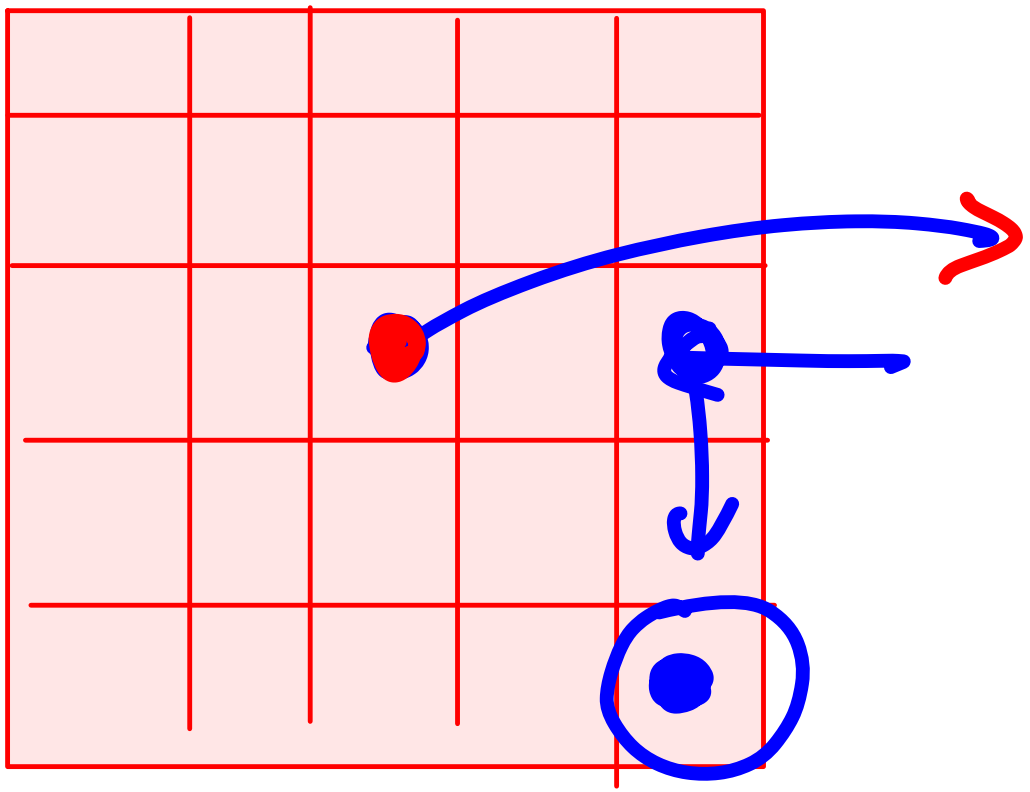
all values in grid are +ve

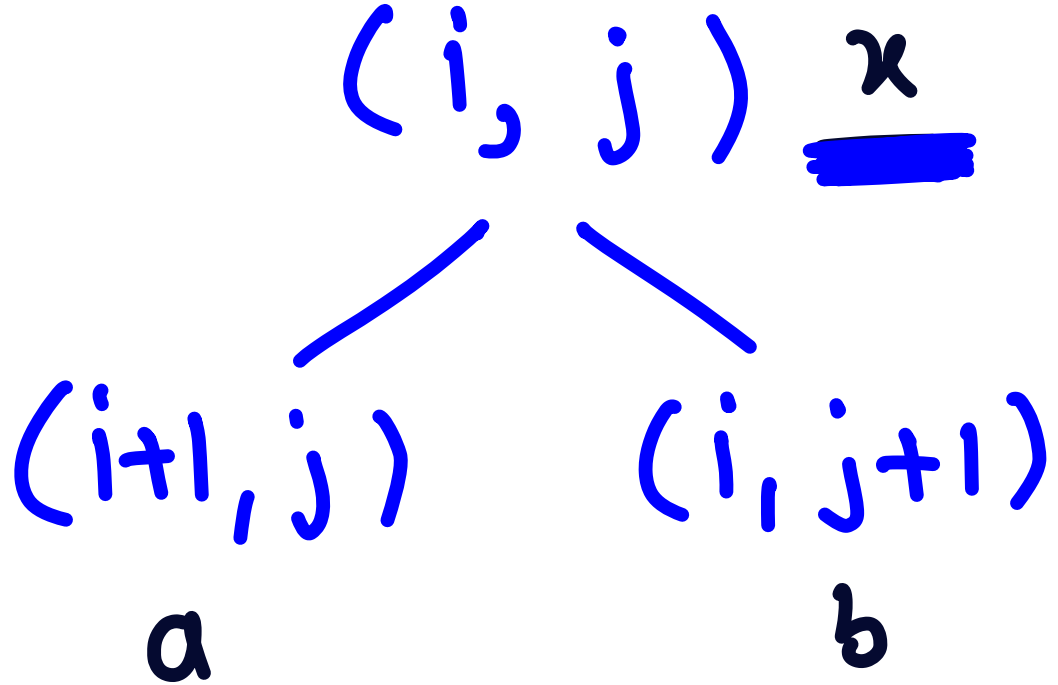
1	5	8
6	2	7
9	3	4




down
p
right
r
a
b
x
y
x = r + min(a, b)
ans = p + min(x, y)


(0,0)
f(input) → output
(x,y)
+ve integer
(1,0)
(0,1)
(2,0)
(1,1)
(1,1)
(0,2)
(3,0)
(2,1)
(2,1)
(1,2)
(2,1)
(1,2)
(1,2)
(0,3)





$x = \min(a, b) + grid[i][j]$

any when you reach dest $= grid[n-1][n-1]$

Naive Way

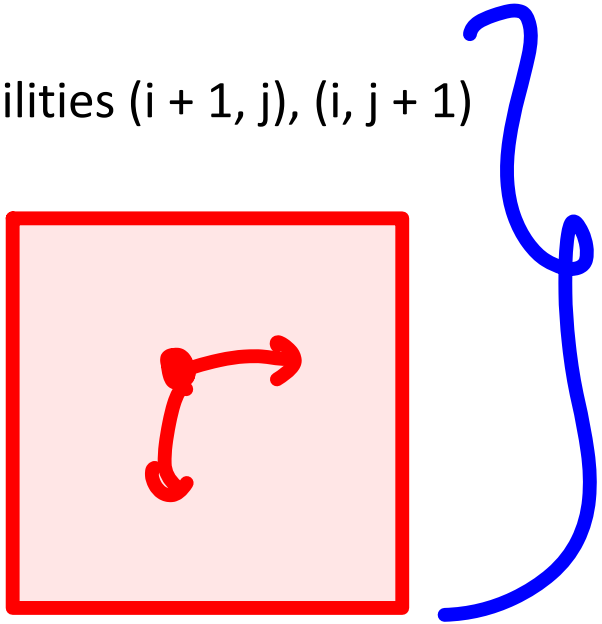
Explore all paths. Standing at (i, j) try both possibilities (i + 1, j), (i, j + 1)

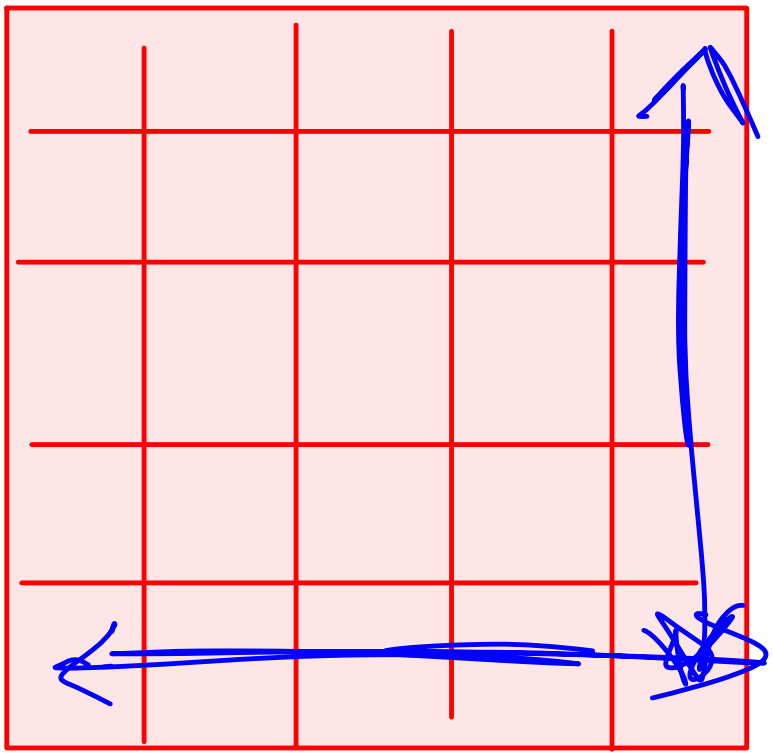
Every cell has two choices

Time complexity: $O(2^{n*n})$?

Actual Time complexity: O(C(2n - 2, n - 1))

$$\frac{(2n-2)!}{(n-1)!\ (n-1)!}$$

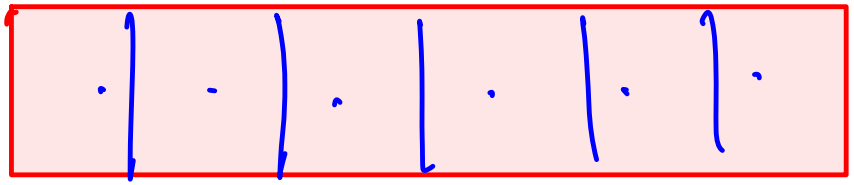




$n = 5$

$n \times n = 25$

2^{25}

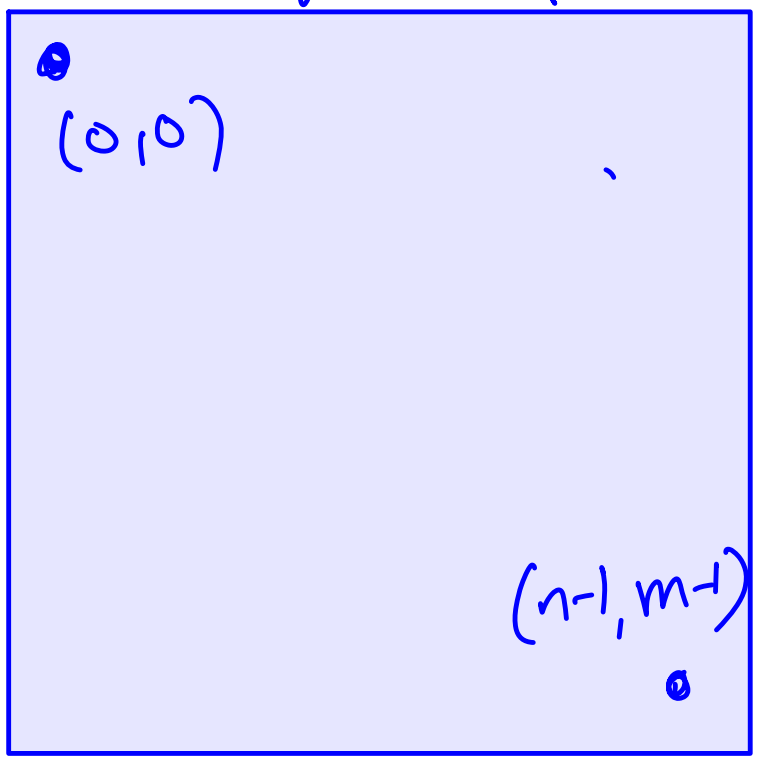


rows = 1

cols = 6

2^6

no. of unique paths



D ↓ n-1 times

R → m-1 times

path → DDRRRDRDR ....

n-1 Ds

m-1 Rs

n unique characters in a string

↓ no. of unique strings

$n!$

A B C → $\frac{(20+30+10)!}{10!\ 20!\ 30!}$

20 30 10

D R → $\frac{(n-1+m-1)!}{(n-1)!\ (m-1)!}$

n−1 m−1

Efficient Way

Overlapping subproblems -> Memoization

Time complexity: O(n * m)

Space complexity: O(n * m)

```cpp
int grid[n][m]; // input matrix

int dp[n][m]; // every value here is -1

// subproblem: f(i, j) represents minimum sum path from (i, j) to (n - 1, m - 1)
int f(int i, int j){ //
    if(i >= n || j >= m){ // moving outside the grid // not allowed
        return INF;
    }
    if(i == n - 1 && j == m - 1) // reached the destination
        return grid[n - 1][m - 1];

    if(dp[i][j] != -1) // this state has been calculated before
        return dp[i][j];

    // state never calculated before
    dp[i][j] = grid[i][j] + min(f(i, j + 1), f(i + 1, j));
    return dp[i][j];
}

void solve(){
    cout << f(0, 0) << nline;
}
```

O(1)

O(1)

O(1)

O(1)

O(1)

Important Terminology

State

- Definition (meaning) of a subproblem that we want to solve along with parameters which are used to differentiate b/w 2 states.

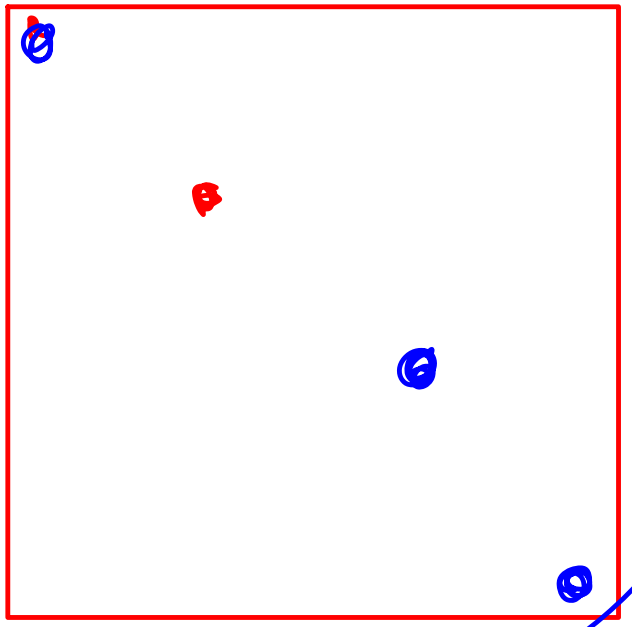
State in Fibonacci Problem:

- dp[i] or f(i) = meaning i^{th} fibonacci number

f(3) f(5) f(n)

State in Grid Problem:

- dp[i][j] or f(i, j) = minimum sum path from (i, j) to (n - 1, n - 1)



① go from (i,j) to (n-1, n-1) and find out the min sum path

② go from (0,0) to (i,j) and find out the min sum path

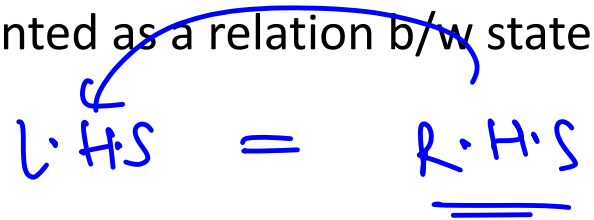
① $f(i,j) = grid[i][j] + \min(f(i+1,j), f[i,j+1])$

② $f(i,j) = grid[i][j] + \min(f(i-1,j), f(i,j-1))$

Important Terminology

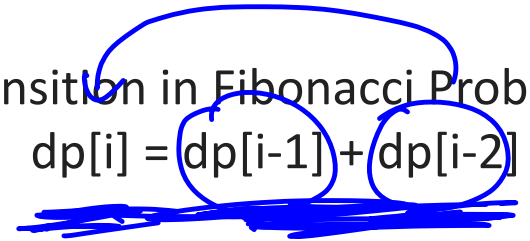
Transition

- Calculating the answer for a state by using the answers of other smaller states (subproblems). Represented as a relation b/w states.



Transition in Fibonacci Problem:

- dp[i] = dp[i-1] + dp[i-2]



Transition in Grid Problem:

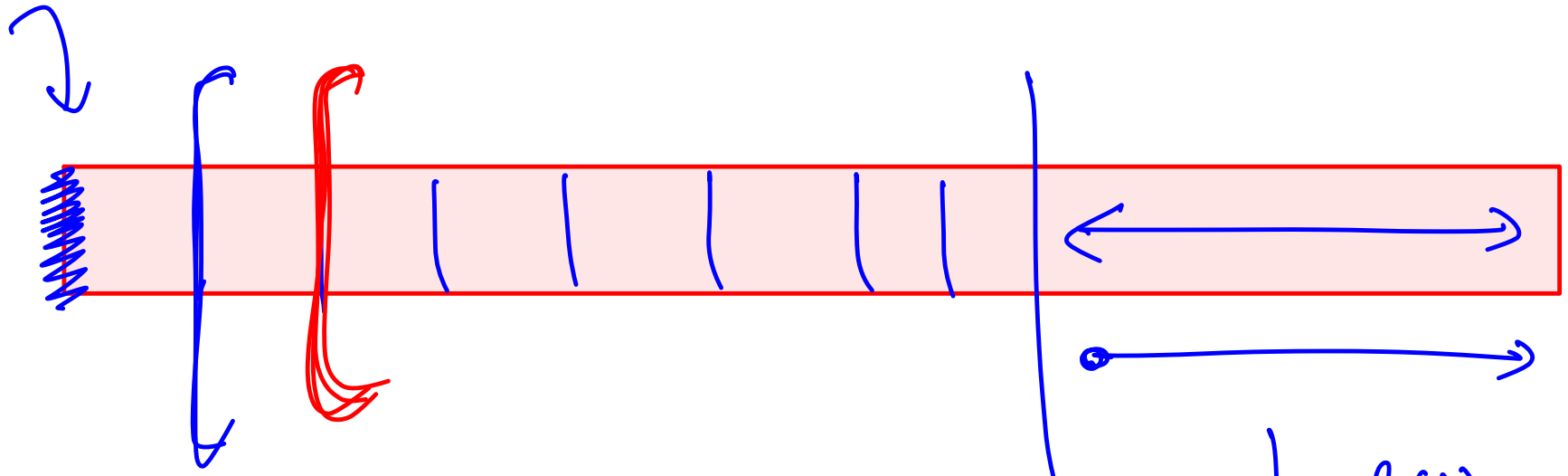
- dp[i][j] = grid[i][j] + min(dp[i+1][j] + dp[i][j+1])

Problem Link

Given an array of integers (both positive and negative). Pick a subsequence of elements from it such that no 2 adjacent elements are picked and the sum of picked elements is maximized.

Sum = 14

Sum = 13



(0) → n-1

pick skip

max

a[0] + (2) 0 + (1)

DnC

f(i) = max sum from i to n-1

f(i) = max(a[i] + f(i+2), f(i+1))

max of both

f(n-1) = max(0, a[n-1])

① $dp[i]$ = max sum from i to $n-1$

$dp[i] = \max(a[i] + dp[i+2], dp[i+1])$

$dp[n-1] = \max(0, a[n-1])$

$dp[n-2] = \max(a[n-2], dp[n-1])$

final ans = $dp[0]$

② $dp[i]$ = max sum from 0 to i

$dp[i] = \max(a[i] + dp[i-2], dp[i-1])$

$dp[0] = \max(0, a[0])$

$dp[1] = \max(a[1], dp[0])$

final ans = $dp[n-1]$

n-1

state

dp[i][0] — don't pick ith element

dp[i][1] — pick the ith element

dp[i][0] = dp[i+1][0]
dp[i][0] = dp[i+1][1]

transition

dp[i][1]
↳ a[i] + dp[i+1][0]

trivial case

dp[n-1][0] = 0
dp[n-1][1] = a[n-1]

final ans

max(dp[0][0], dp[0][1])

Some ways to solve the problem

1. Having 2 parameters to represent the state

State:

dp[i][0] = maximum sum in (0 to i) if we don't pick i^{th} element

dp[i][1] = maximum sum in (0 to i) if we pick i^{th} element

Transition:

dp[i][0] = max(dp[i - 1][1], dp[i - 1][0])

dp[i][1] = arr[i] + dp[i - 1][0]

Final Answer:

max(dp[n - 1][0], dp[n - 1][1])

trivial case :

dp[0][0] = 0

dp[0][1] = a[0]

Some ways to solve the problem

2. Having only 1 parameter to represent the state

State:

dp[i] = max sum in (0 to i) not caring if we picked i^{th} element or not

Transition: 2 cases

- pick i^{th} element: cannot pick the last element : arr[i] + dp[i - 2]
- leave i^{th} element: can pick the last element : dp[i - 1]

dp[i] = max(arr[i] + dp[i - 2], dp[i - 1])

Final Answer:

dp[n - 1]

```cpp
int a[n]; // input array

int dp[n]; // filled with -INF to represent uncalculated state

// f(i) = max sum till index i
int f(int index){
    if(index < 0) // reached outside the array
        return 0;
    if(dp[index] != -INF) // state already calculated
        return dp[index];

    // try both cases and store the answer
    dp[index] = max(a[index] + f(index - 2), f(index - 1));
    return dp[index];
}

void solve(){
    cout << f(n - 1) << nline;
}
```

-inf = -10^8

dp(i) = max sum from 0 to i inspection if whether ith element was piked or not