Introduction to Dynamic Programming 2

- Priyansh Agarwal

$p_4 - A_4$

$p_5 - A_5 =$

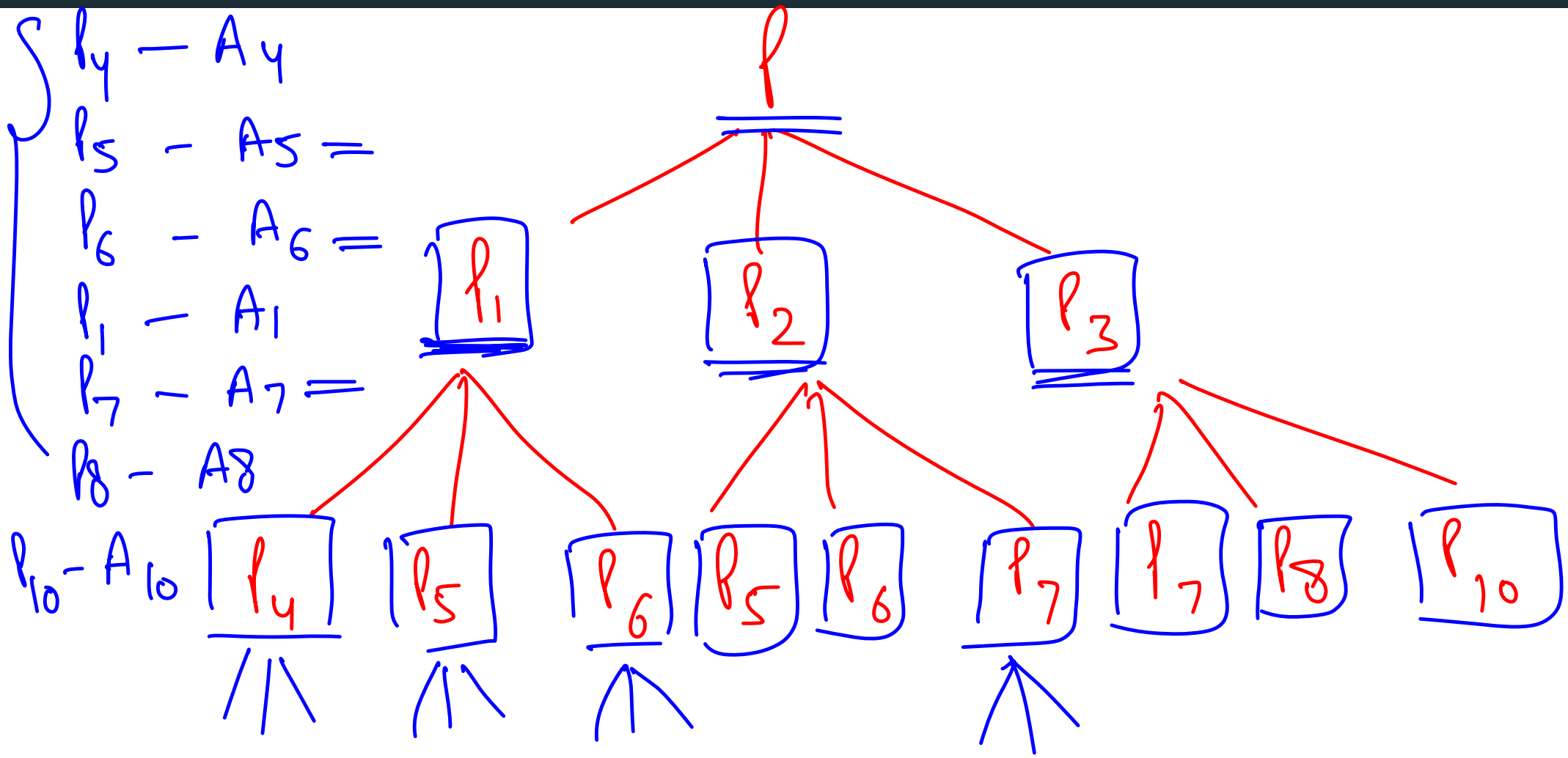
$p_6 - A_6 =$

$p_1 - A_1$

$p_7 - A_7 =$

$p_8 - A_8$

$p_{10} - A_{10}$



We are ensuring that every subproblem is solved just on

Time complexity for fibonacci

$T(n) \longrightarrow O(1) \times 1$

$T(n-1) \longrightarrow O(1) \times 1$

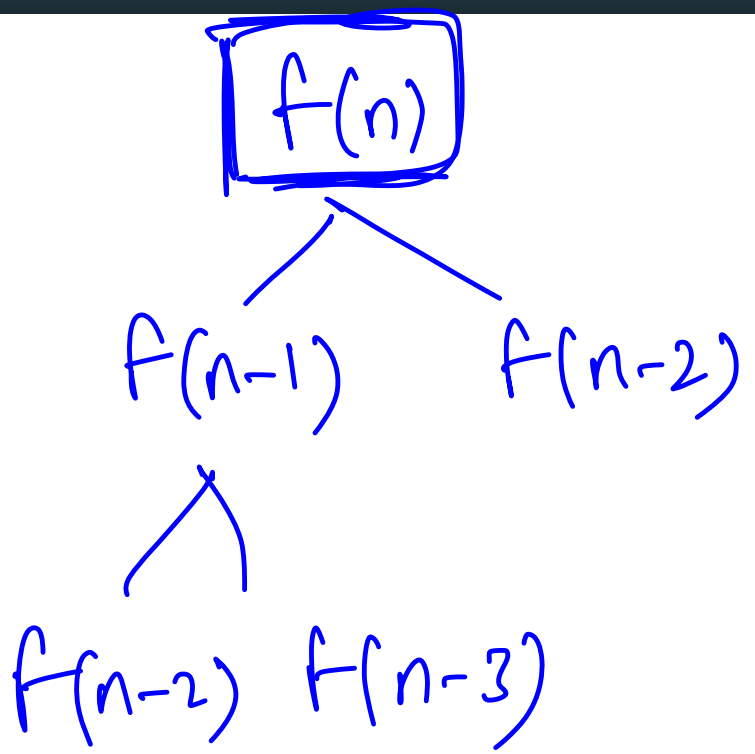
$T(n-2) \longrightarrow O(1) \times 1$

$\vdots \qquad\qquad \vdots$

$T(1) \longrightarrow O(1) \times 1$

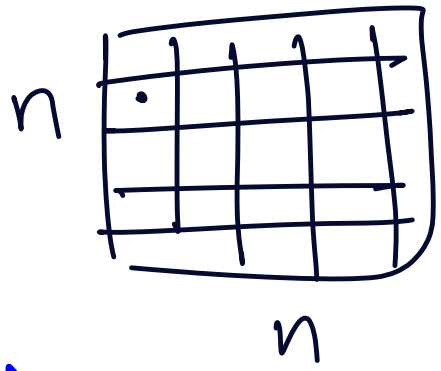
$= O(1) \cdot n$

$= O(n)$



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

Time complexity for Grid problem



$T(0,0) \rightarrow O(1) \times 1$

$T(0,1) \rightarrow O(1) \times 1$

$T(i,j) \rightarrow O(1) \times 1$

$T(n-1, n-1) \rightarrow O(1) \times 1$

$= O(1) \cdot n \times n$

$= O(n \cdot n)$

$$f(i,j) = grid[i][j] + \min \begin{cases} f(i+1, j) \\ f(i, j+1) \end{cases}$$

$O(1)$ $O(1)$

Time and Space Complexity in DP

Time Complexity:

unique subproblems

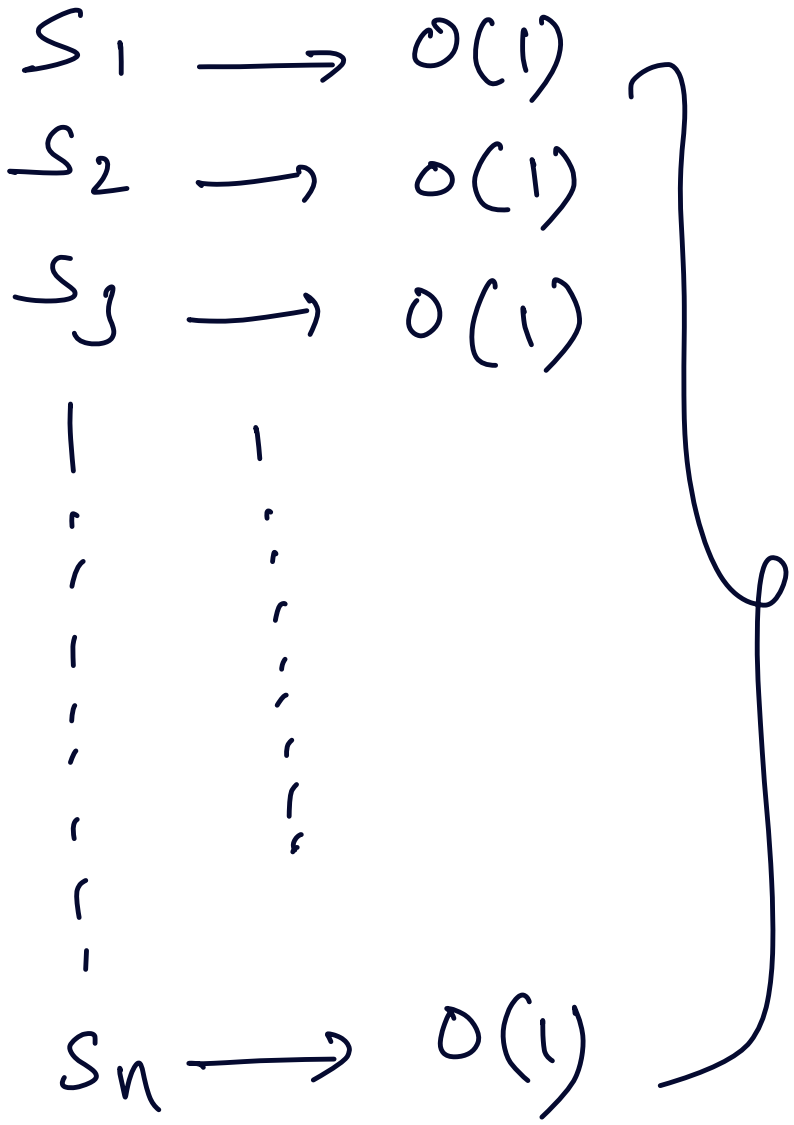
Estimate: Number of States * Transition time for each state

Exact: Total transition time for all states

Space Complexity:

Number of States * Space required for each state

→ O(1) in 99% problems



T.C = # states x T.T ~~for~~ 1 state

$= n \times O(1) = O(n)$

$S_1 \rightarrow O(1)$

$S_2 \rightarrow O(n)$

$S_3 \rightarrow O(\log n)$

$S_4 \rightarrow O(1)$

$\vdots$

$S_n \rightarrow O(1)$

T.C = # states x T.T for 1state

T.C = $\sum_{i \in states}$ T.T for state$_i$

Examples (fine with estimate)

Example 1

- State: dp[i][j] (1 <= i <= n), (1 <= j <= m)
- Transition: dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

states = n.m
T.T/state = O(1)

Example 2

- State: dp[i][j][k] (1 <= i <= n), (1 <= j <= m), (1 <= k <= 10)
- Transition: dp[i][j][k] = dp[i - 1][j][k] + dp[i - 2][j][k].... Dp[1][j][k]

O(m.n.n/2.10) = O(m.n^2)

states = n.m.10 , T.T/state = n/2

Example 3:

- State: dp[i] (1 <= i <= n)
- Transition: dp[i] = dp[i / 2] + dp[i / 4] + dp[i / 8]....

states = n

T.T/state = log(i)

< n.log n

$dp[i][j][k] = dp[i-1][j][k] + dp[i-2][j][k]$

$+ \cdots\cdots + dp[1][j][k]$

$\rightarrow O(i)$

$(1, j, k) \rightarrow 1$

$(2, j, k) \rightarrow 2$

$(3, j, k) \rightarrow 3 \quad \cdots\cdots \quad (n, j, k) \rightarrow n$

$(1, j, k) \longrightarrow 1 \quad (m \times 10)$

$(2, j, k) \longrightarrow 2 \quad (m \times 10)$

$\vdots$

$(n, j, k) \longrightarrow n \quad (m \times 10)$

$m \times 10 \,(1 + 2 + 3 + \cdots + n)$

$= m \times 10 \times \frac{n(n+1)}{2}$

$= O(m \cdot n^2)$

Exact

Estimate = #states $\times$ TT/state $= n \times m \times 10 \cdot \frac{n}{2} = O(m \cdot n^2)$

$dp(1) \rightarrow \log(1)$

$dp(2) \rightarrow \log(2)$

$\vdots$

$dp(n) \rightarrow \log(n)$

$\log_2(1) + \log_2(2) \cdots \log_2(n)$

$= \log_2(1 \times 2 \times 3 \cdots n)$

$= \boxed{\log_2(n!)}$

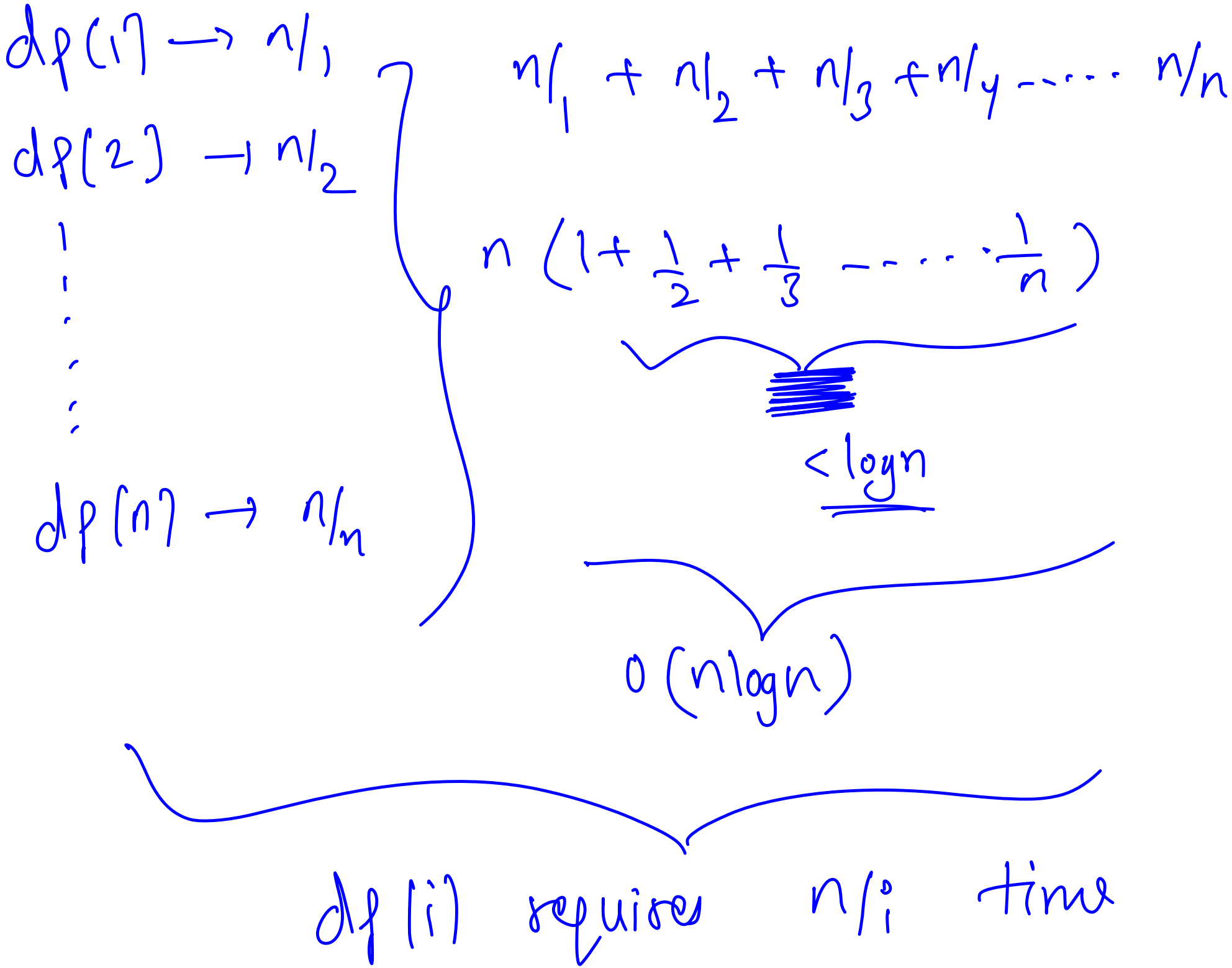
$\log_2(n!) \ll \log_2(n^n) = < n \log n$

More Example

Example 1: Link for studying further
- State: dp[i] (1 <= i <= n)
- Transition: dp[i] requires O(n / i) time to get evaluated

Example 2: Link for studying further
- State: dp[i] (1 <= i <= n)
- Transition: dp[i] requires O(number of primes <= i) time

Example 3: Link for studying further (tip: translate to English)
- State: dp[i] (1 <= i <= n)
- Transition: dp[i] requires O(number of factors of i) time


dp(1) → n/1
dp(2) → n/2
dp(n) → n/n
n/1 + n/2 + n/3 + n/4 ..... n/n
n (1 + 1/2 + 1/3 ..... 1/n)
< log n
O(n log n)
dp(i) requires n/i time

dp[i] requires O(no. of primes < i) time

dp[i] < O(i) time

dp[1] → O(1)
dp[2] → O(2)
⋮
dp[n] → O(n)

$1 + 2 + 3 \cdots n$

$= \frac{n(n+1)}{2} = O(n^2)$

---

dp[i] = exact no. of primes < i

No. of primes upto $n \approx \frac{n}{\log_e n}$

$df(1) \rightarrow \frac{1}{\log 1}$

$df(2) \rightarrow \frac{2}{\log 2}$

$\vdots$

$df(n) \rightarrow \frac{n}{\log n}$

$\left(\sum_{i=1}^{n} \frac{i}{\log i} \right) <<<< O(n^2)$

dp[1] -> O(f(1))

dp[2] -> O(f(2))

⋮

dp[n] -> O(f(n))

$f(i)$ = # of factors of i

$f(i) \leq f(n)$

$f(n) \approx \text{sqrt}(n)$

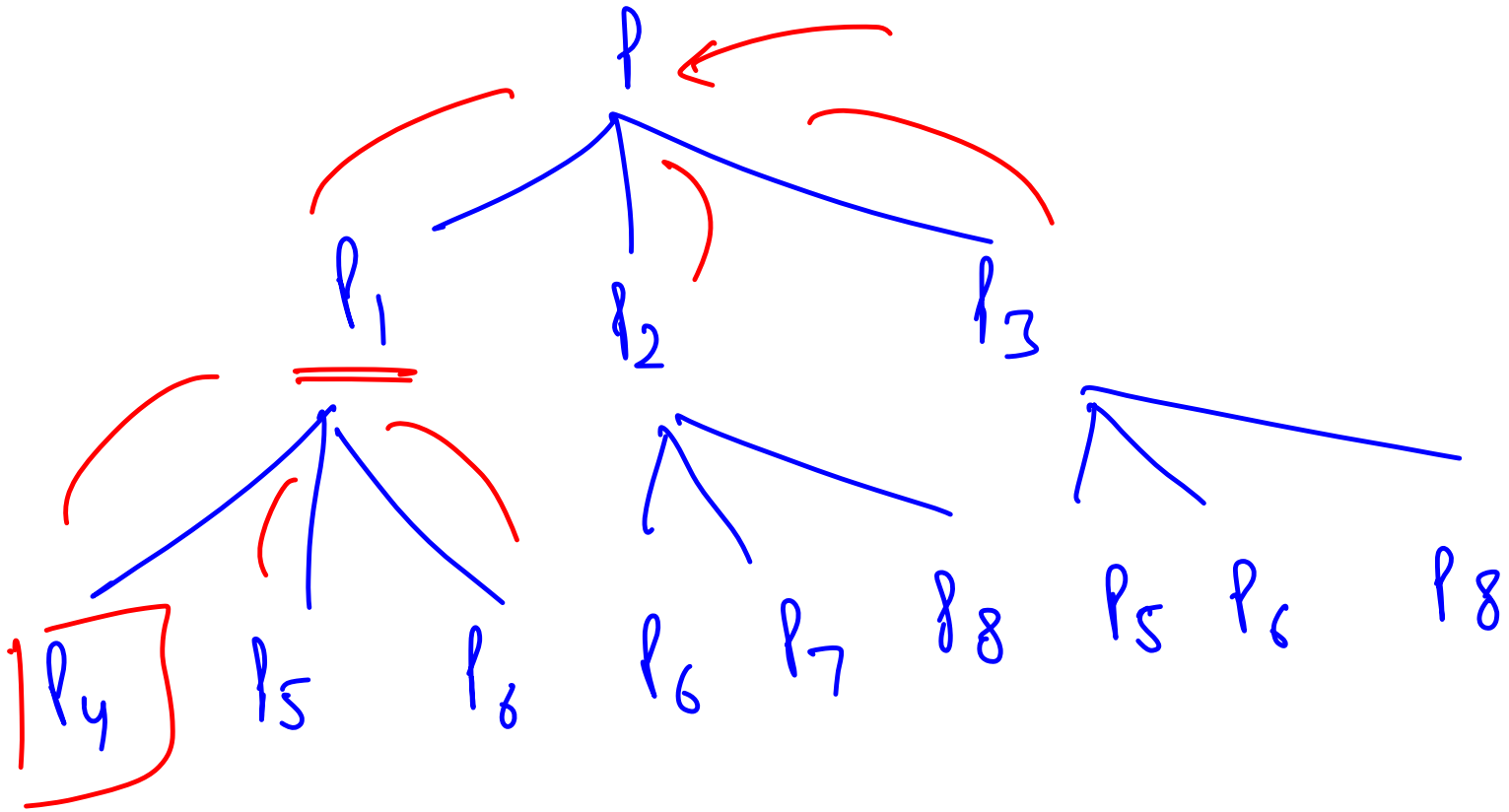
$n \cdot \text{sqrt}(n)$ → Estimate

Exact → $\sum_{i=1}^{n} f(i) = n \cdot n^{1/2}$

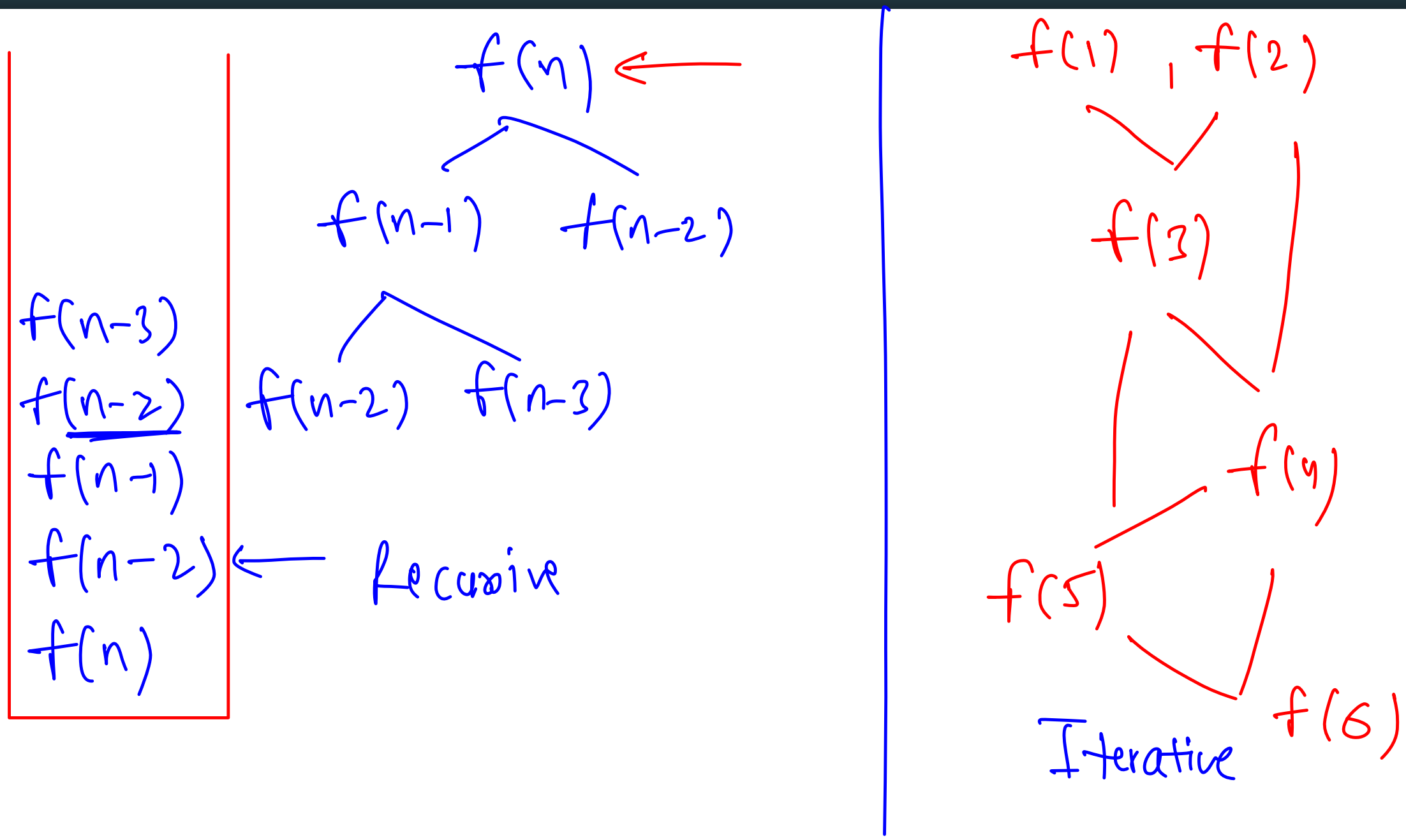
Recursive vs Iterative DP

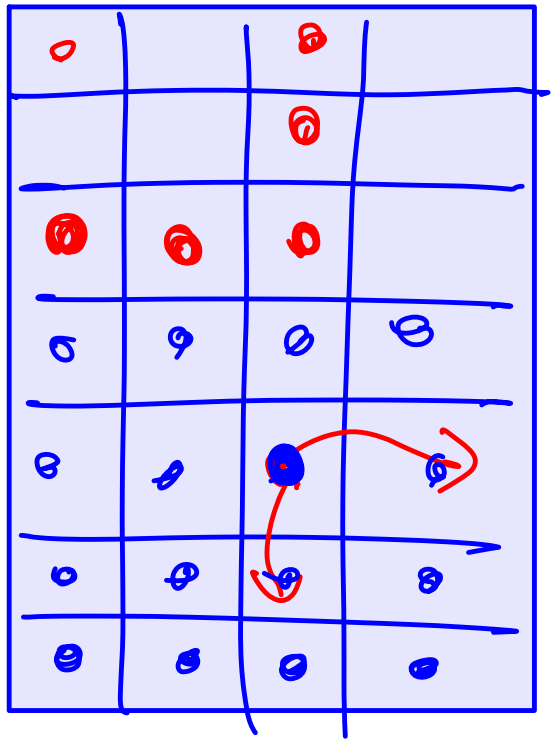
Recursive	Iterative
Slower (runtime)	Faster (runtime)
No need to care about the flow	Important to calculate states in a way that current state can be derived from previously calculated states
Does not evaluate unnecessary states	All states are evaluated
Cannot apply many optimizations	Can apply optimizations

only advantage of recursive dp



Recursive Implementation


f(n)
f(n-1)
f(n-2)
f(n-2)
f(n-3)
f(n-3)
f(n-2)
f(n-1)
f(n-2) ← Recursive
f(n)
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
Iterative



no. of paths to go from top left to bottom right

$f(i,j)$ = no. of paths from (i,j) to $(n-1, m-1)$

$f(i,j) = f(i+1, j) + f(i, j+1)$

$f(i+1, j)$: only when bottom call is empty

$f(i, j+1)$: only when right call is empty

Converting Recursive to Iterative

Rule 1:

All the states that a particular state depends on must be evaluated before that state

Note:

You don’t have to convert Recursive to Iterative if it is not intuitive at this point.

fibonacci

f(n) < f(n-1), f(n-2)

f(n) < f(n-2), f(n-3)

⋮

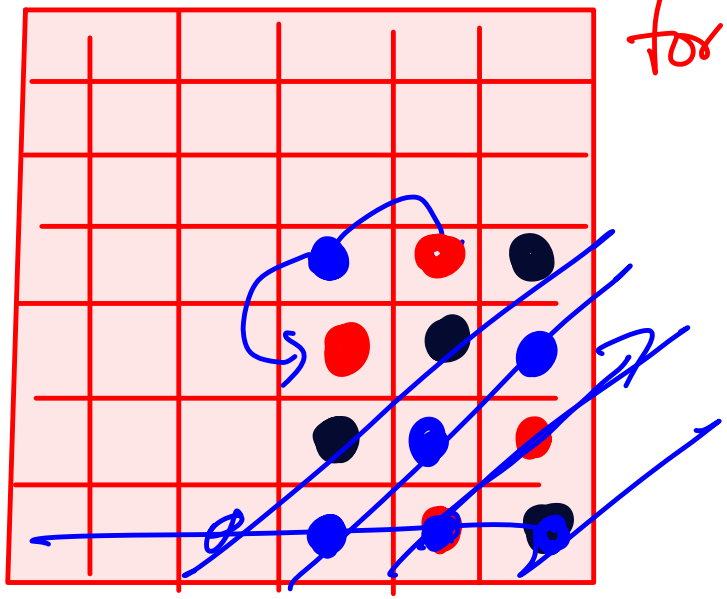
f(3) < f(2), f(1)

```
dp[1] = 1 , dp[2] = 1
for (int i = 3 ; i <= n ; i++) {
    dp[i] = dp[i-1] + dp[i-2]
}
cout << dp[n] ;
```

(i, j) → (i-1, j+1)
→ (i+1, j-1)

Grid problem

f(i, j) < f(i, j+1), f(i+1, j)



```
for ( i → (n-1 too)
for (j → (m-1 too)
dp[i][j]
```

dp[i+1][j] dp[i][j+1]

General Technique to solve any DP problem

Divide n Conquer

1. State
   Clearly define the subproblem. Clearly understand when you are saying dp[i][j][k], what does it represent exactly
2. Transition:
   Define a relation b/w states. Assume that states on the right side of the equation have been calculated. Don't worry about them.
3. Base Case
   When does your transition fail? Call them base cases answer before hand. Basically handle them separately.
4. Final Subproblem
   What is the problem demanding you to find?

Problem 1: Link (Next class)