Algorithm Design and Analysis

Dynamic Programming I

8/2/2022

The First Example

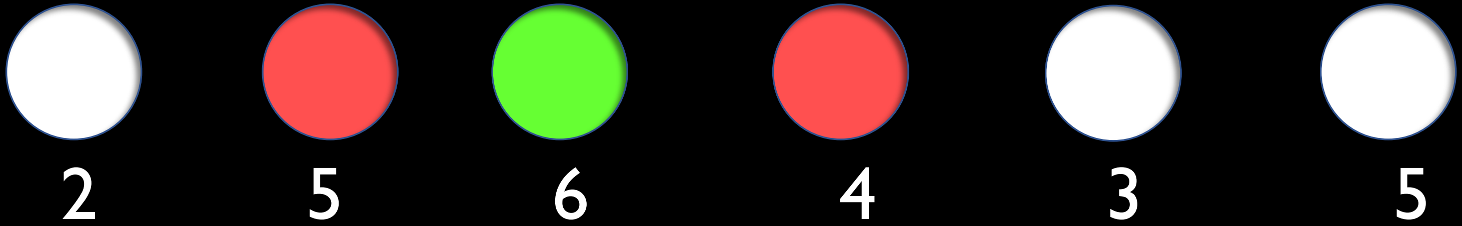
Input : A set of balls arranged in a row ; each ball has a weight

Output : Pick balls of maximum possible total weight ; No two adjacent balls can be picked

The First Example : An Intuitive ‘Greedy’ Approach

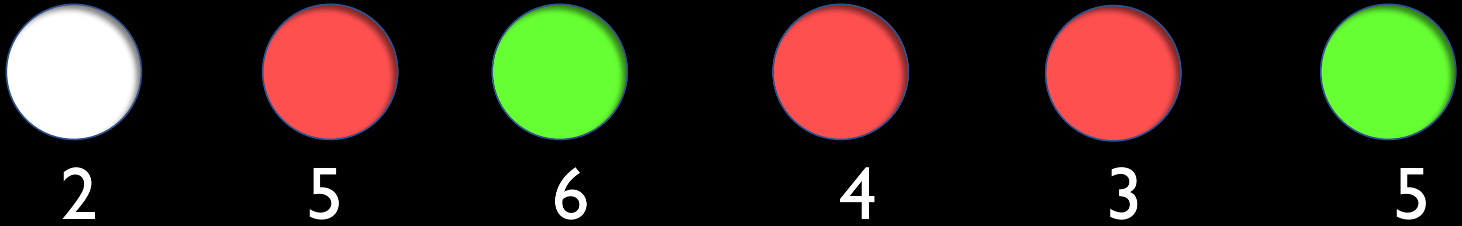
Attempt I : Pick the ball of maximum weight unless you have picked its neighbors and continue…

The First Example : An Intuitive 'Greedy' Approach



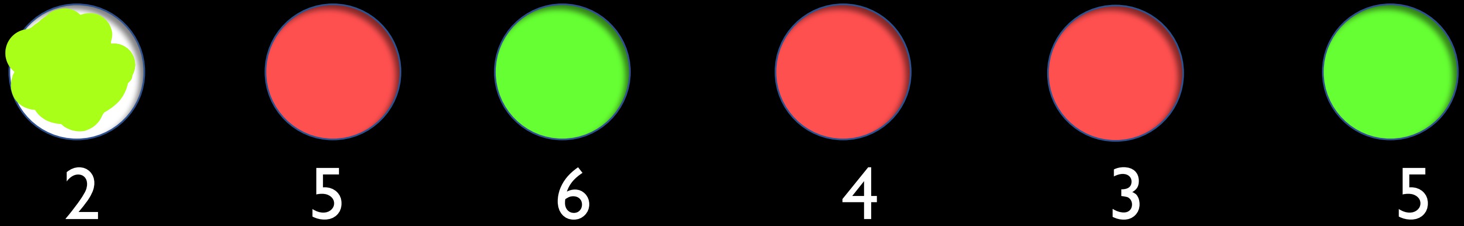
Attempt I : Pick the ball of maximum weight unless you have picked its neighbors and continue…

The First Example : An Intuitive ‘Greedy’ Approach



Attempt 1 : Pick the ball of maximum weight unless you have picked its neighbors and continue…

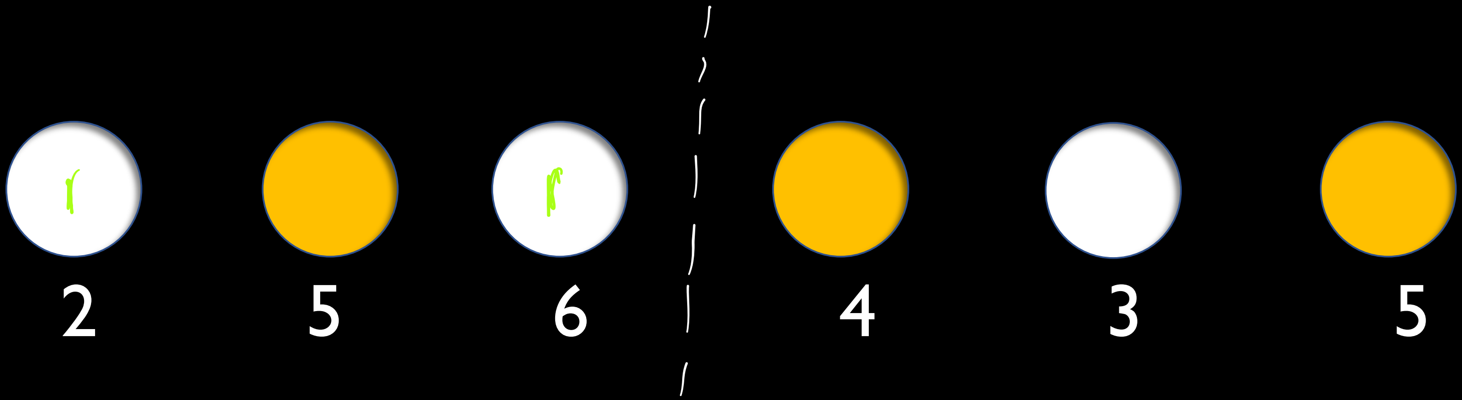
The First Example : An Intuitive ‘Greedy’ Approach



Attempt 1 : Pick the ball of maximum weight unless you have picked its neighbors and continue

Total weight of greedy solution = 11 + 2 = 13 .

The First Example : An Intuitive 'Greedy' Approach



Attempt 1 : Pick the ball of maximum weight unless you have picked its neighbors and continue

Total weight of greedy solution = ~~14~~ 13

But optimal solution is = 14

$T(n) = 4T(n/2)$

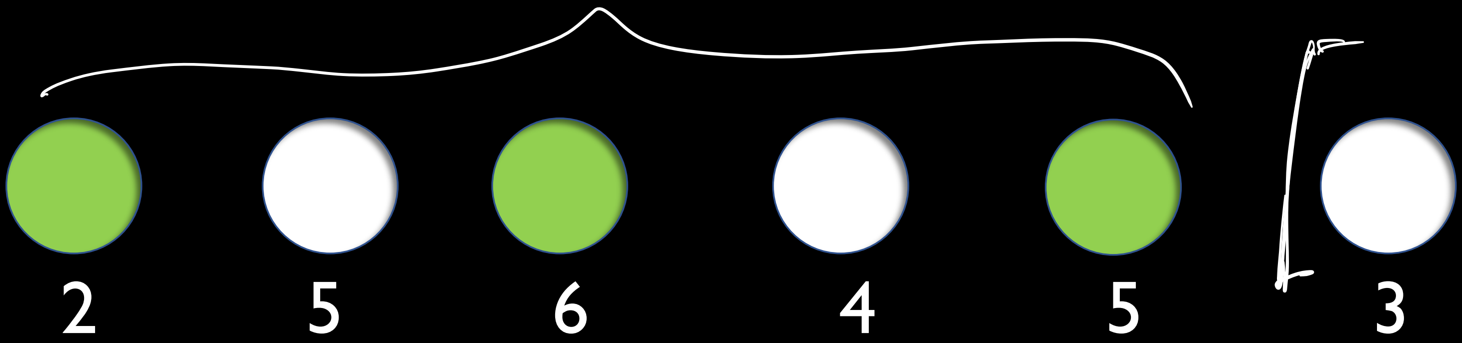
$O(n^2)$

A Recursive approach

Recap:

A recursive algorithm is one which solves a problem by invoking itself repeatedly on inputs of strictly smaller sizes until the size is so small that one can solve it trivially

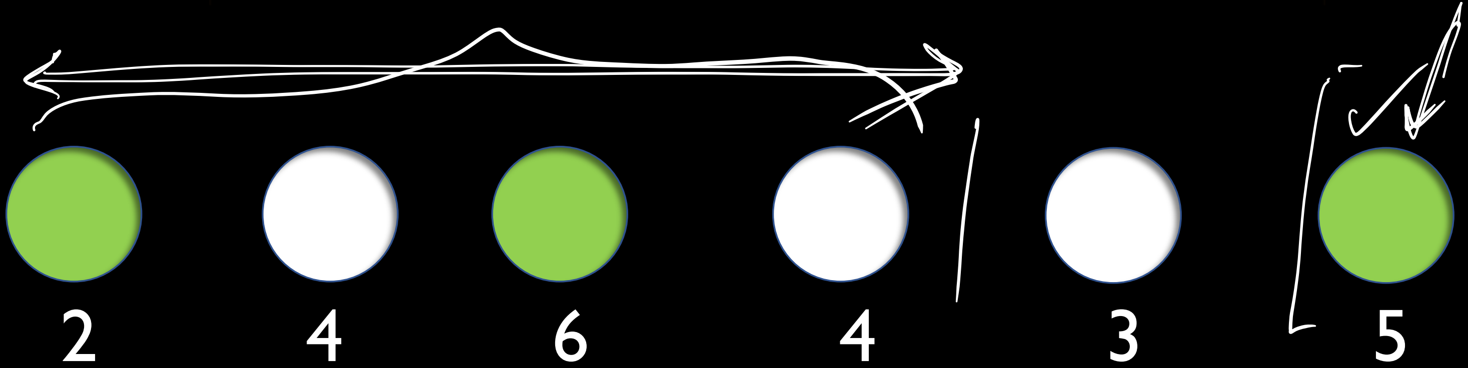
Understanding the Optimal Solution



Observation 1:

If Last ball is not part of the optimal solution, then
Optimal solution = Optimal solution with the last ball removed from input
(a strictly smaller input !!)

Understanding the Optimal Solution



If not true $\Rightarrow \exists$ another optimal solution with value $>$ opt[i] - weight(i)

Observation 2:

If Last ball is part of the optimal solution

So, second last ball cannot be !

Optimal solution to the original problem = Optimal solution to the problem without last two balls (a strictly smaller solution) + weight of last ball

Formal Recurrence

$opt[i]$: The optimal solution with balls $\{b_1, b_2, \dots, b_i\}$ [Subproblem defn]

$\forall\, i = 0, 1, 2, \dots, n$

$opt[0] = 0$; $opt[1] = weight(1)$.

$$opt[i] = \max\begin{cases} opt[i-1] & \text{Case I} \\ opt[i-2] + weight(i) & \text{Case II} \end{cases}$$

Claim (Case II):-

Proof:- Optimal solution for $b_1, b_2, \dots, b_i$ with b_i removed IS an optimal sol^n. for $b_1, b_2, \dots, b_{i-2}$

S'pose this is not true. [$opt[i-2] = opt[i] - weight(i)$

Towards an Algorithm

Upshot : The optimal solution for balls $b_1, b_2, \cdots b_n$ can look **only two different ways** –

Towards an Algorithm

Upshot : The optimal solution for balls $b_1, b_2, \cdots b_n$ can look **only two different ways** –

1. b_n not in the solution : Then the overall solution is just the solution with balls $b_1, b_2, \cdots b_{n-1}$

Towards an Algorithm

Upshot : The optimal solution for balls $b_1, b_2, \cdots b_n$ can look **only two different ways** –

1. b_n not in the solution : Then the overall solution is just the solution with balls $b_1, b_2, \cdots b_{n-1}$

2. b_n is in the solution : Then overall solution is solution with balls $b_1, b_2, \cdots b_{n-2}$ plus b_n

Towards an Algorithm

Upshot : The optimal solution for balls $b_1, b_2, \cdots b_n$ can look **only two different ways** –

1. b_n not in the solution : Then the overall solution is just the solution with balls $b_1, b_2, \cdots b_{n-1}$

2. b_n is in the solution : Then overall solution is solution with balls $b_1, b_2, \cdots b_{n-2}$ plus b_n

Trouble : We do not know the optimal solution So, we do not know which option to take !

Towards an Algorithm

Upshot : The optimal solution for balls $b_1, b_2, \cdots b_n$ can look **only two different ways** –

1. b_n not in the solution : Then the overall solution is just the solution with balls $b_1, b_2, \cdots b_{n-1}$

2. b_n is in the solution : Then overall solution is solution with balls $b_1, b_2, \cdots b_{n-2}$ plus b_n

Trouble : We do not know the optimal solution So, we do not know which option to take !

Way out : Try both and take the best

A Recursive Algorithm

SelectBalls $(b_1, b_2, \cdots b_n, n)$

If n==1, return b_1 Else If If n==0, return 0 Else

$w_1 = \text{SelectBalls}(\underline{b_1, b_2, \cdots b_{n-1}}, n-1)$

$w_2 = \text{SelectBalls}(\underline{b_1, b_2, \cdots b_{n-2}}, n-2)$
$\underline{+ \text{ weight of } b_n}$

Return $\underline{\max\{w_1, w_2\}}$

Runtime:

$$\left.\begin{aligned} T(n) &= T(n-1) \\ &\quad + T(n-2) + c. \end{aligned}\right\}$$

$T(0) = 1;\ T(1) = 1.$

A Recursive Algorithm : Runtime

SelectBalls $(b_1, b_2, \cdots b_n, n)$

If n==1, return b_1 Else

$w_1 = \text{SelectBalls}(b_1, b_2, \cdots b_{n-1}, n-1)$

$w_2 = \text{SelectBalls}(b_1, b_2, \cdots b_{n-2}, n-2)$
+ weight of b_n

Return $\max\{w_1, w_2\}$

$\mathrm{T(n) = T(n-1) + T(n-2) + c}$

A Recursive Algorithm : Runtime

SelectBalls $(b_1, b_2, \cdots b_n, n)$

If n==1, return b_1 Else

$$w_1 = \text{SelectBalls}(b_1, b_2, \cdots b_{n-1}, n-1)$$

$$w_2 = \text{SelectBalls}(b_1, b_2, \cdots b_{n-2}, n-2) + \text{weight of } b_n$$

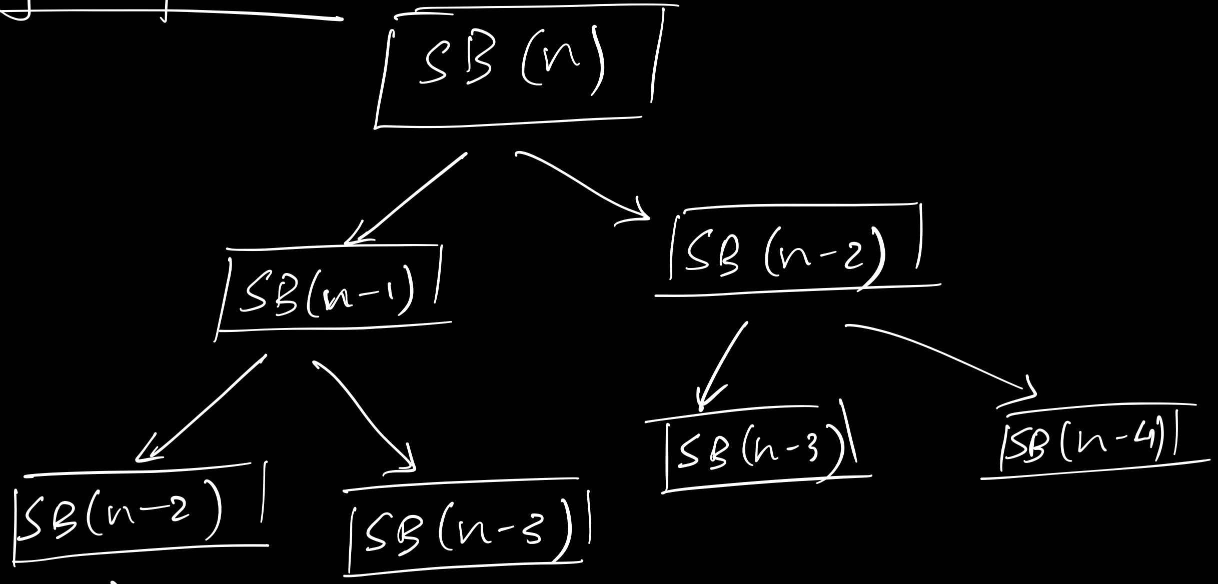
Return $\max\{w_1, w_2\}$

$$\mathrm{T(n) = T(n-1) + T(n-2) + c}$$

$$T(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Why is the Runtime So Horrible ?

Overlapping Subproblems



The MOST important insight

Question : How many distinct recursive subproblems is this algorithm really solving ?

Answer : n

The MOST important insight

Question : How many distinct recursive subproblems is this algorithm really solving ?

Answer : n

Obvious Fix : Cache already computed subproblem values in an array and look them up in $O(1)$ time if available ; otherwise recurse [memo(r?!)ization]

Eliminating Redundancy

Opt[i]

Tab : Array of size n → Memoization Table

SelectBalls $(b_1, b_2, \cdots b_n)$

Tab[i]

If Tab[n] is valid return Tab[n] else

If n==0, Tab[n]= 0 Else If

If n==1, Tab[n]= weight of b_1 Else

$$\left.\begin{aligned} w_1 &= \text{SelectBalls}(b_1, b_2, \cdots b_{n-1}) \\ w_2 &= \text{SelectBalls}(b_1, b_2, \cdots b_{n-2}) + \text{weight of } b_n \end{aligned}\right\}$$

Tab[n] = $\max\{w_1, w_2\}$

A Linear Time Iterative Solution

Tab : Array of size $n + 1$

SelectBalls $(b_1, b_2, \cdots b_n)$

$\text{Tab}[0] = 0; \ \text{Tab}[1] = b_1$

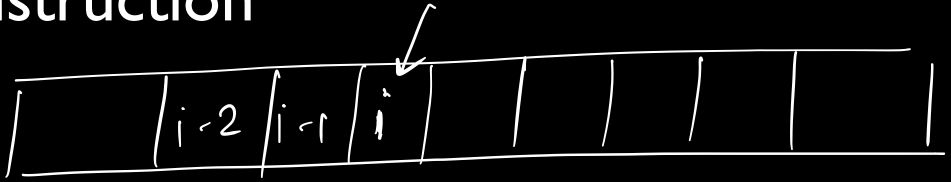
for $i = 2, 3, \ldots n$

$$\text{Tab}[i] = \max \begin{cases} \text{Tab}[i-1] \\ \text{Tab}[i-2] + \text{weight}(b_i) \end{cases}$$

Return $\text{Tab}[n]$

Reconstruction

Tab



Obvious:- Maintain the 'solution' for each entry.

Better:- Use the already computed Tab.

Key Point:-

Ball i is selected $\Longleftrightarrow$ $Tab[i] = Tab[i-2] + weight(i)$

Reconstruction

Reconstruct (Tab)

S : ϕ [Store the balls]

$i = n$

while $i \geq 1$

if $Tab[i] = Tab[i-2] + weight(b_i)$ ✓

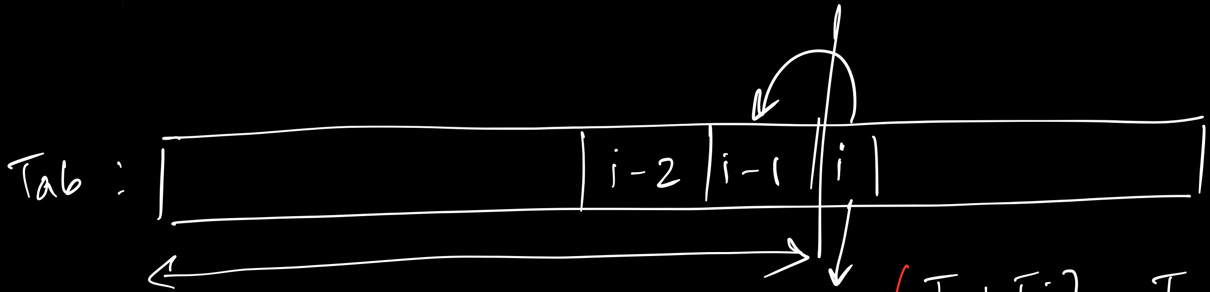
add i to S

decrement i by 2.

else dec. ~~ϕ~~ i by 1.

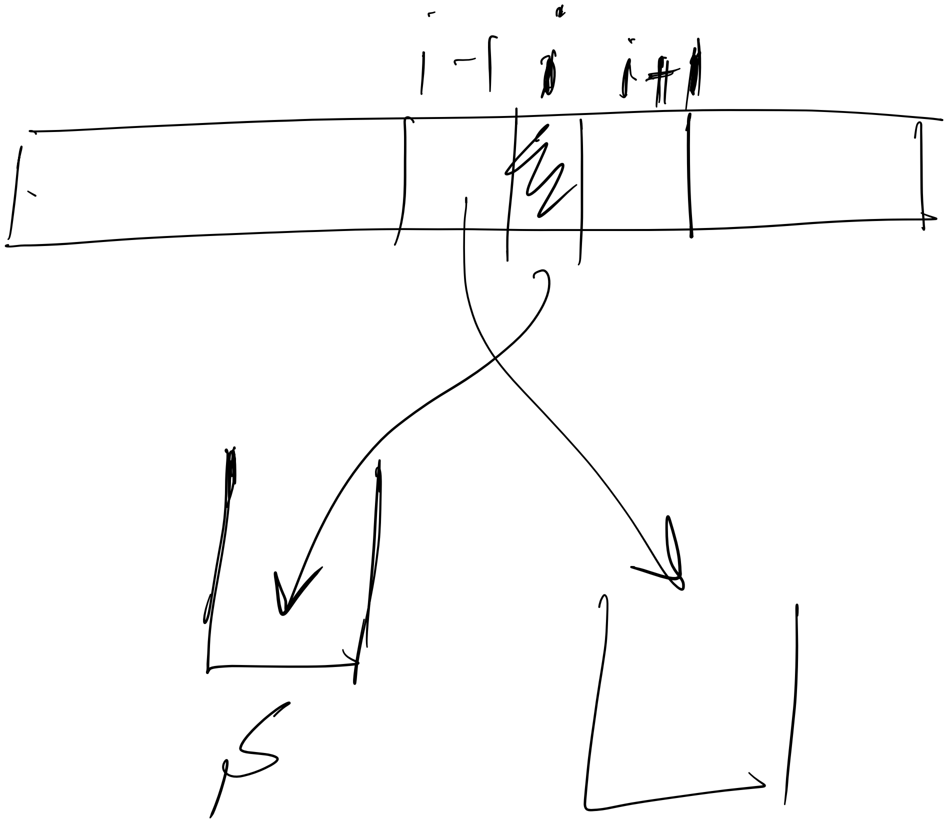
Return S.

Reconstruction



$$\checkmark\ \text{Tab}[i] = \text{Tab}[i-2] + \text{weight}(b_i)$$

$$\checkmark\ \text{Tab}[i] \stackrel{?}{=} \text{Tab}[i-1]$$


i-1
i
i+1
S