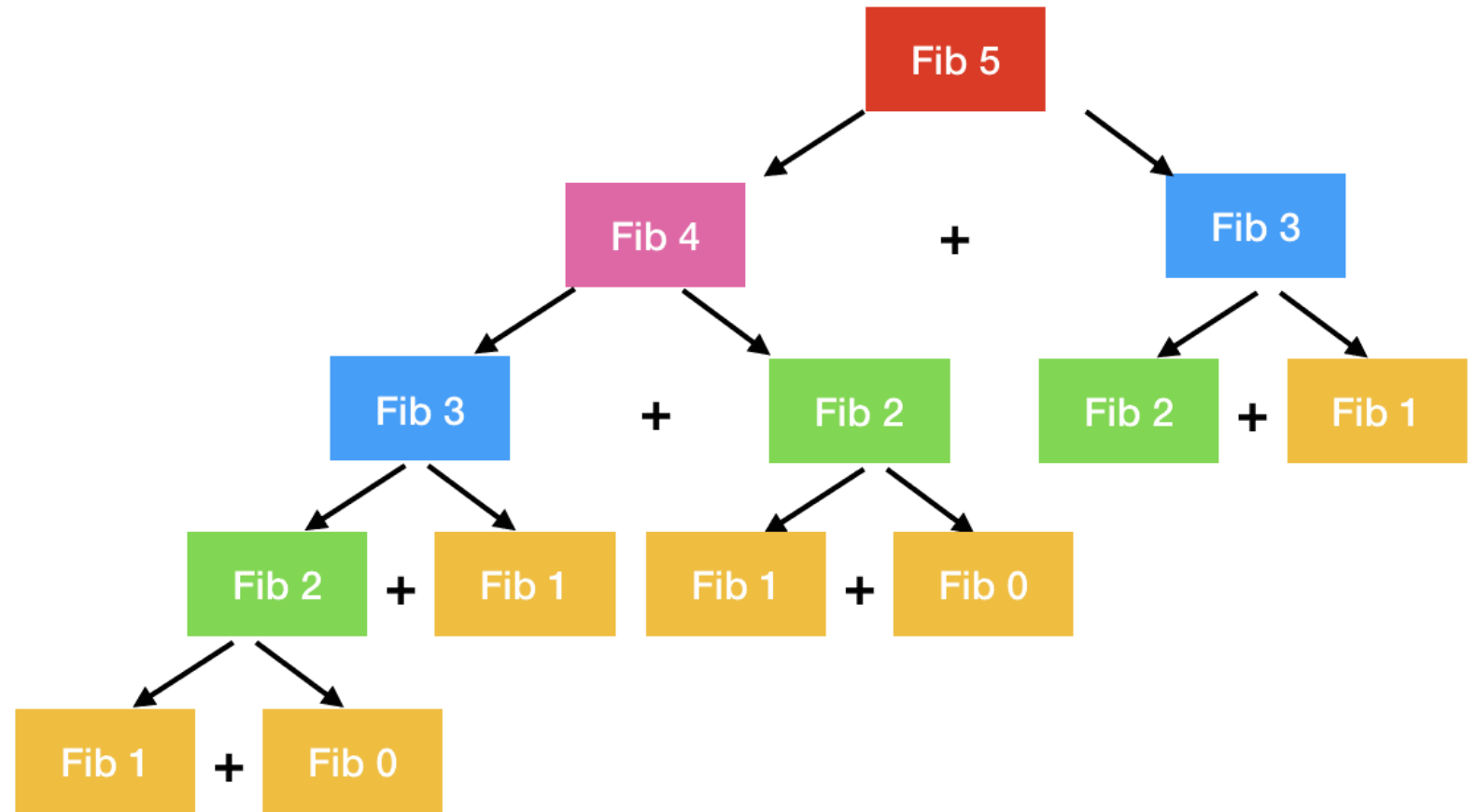


Dynamic Programming

-
- The basic idea of dynamic programming is to store the result of a problem after solving it. So you don't have to solve the problem again just use the stored solution.
 - Cons: Dynamic programming basically trades time with memory.

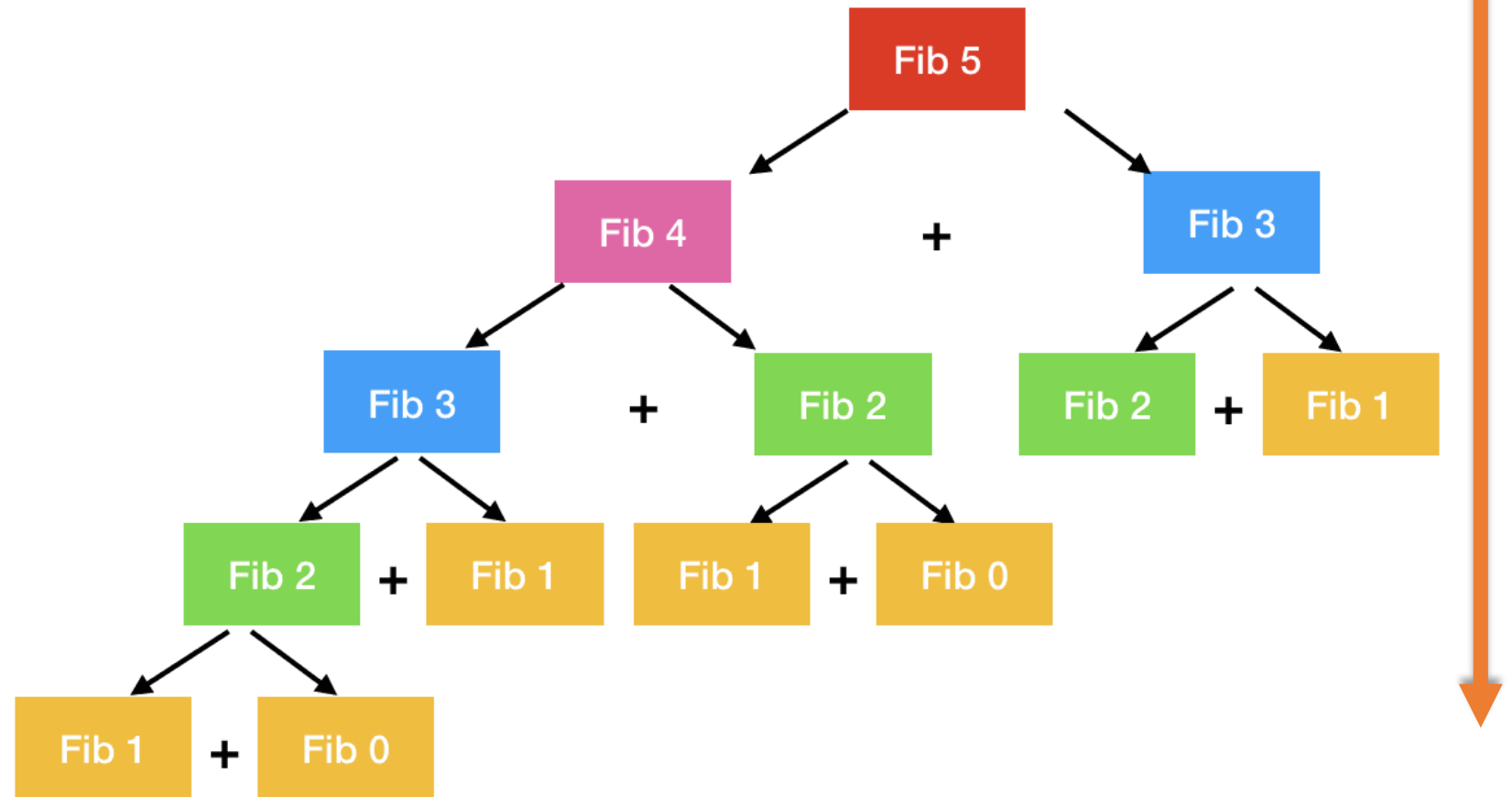
Revisit Fibonacci numbers

- Divide and conquer Approach
- Recompute subproblems



Revisit Fibonacci numbers

- Top down approach
- **memoization**



Fibonacci Formulation

- Define subproblem → the original problem on a prefix of the input.
- Recurrence relation → Express the relation $T(n)$ in terms of $T(n-1), \dots, T(1)$
- update

Revisit Fibonacci numbers

- Bottom up Approach
- Dynamic programming
- Store in an array

Fib(n)

```
//Create an array of length n
A[0]=0
A[1]=1
for i =2 to n
    A[i]=A[i-1]+A[i-2]
return A[i]
```

Complexity = $\theta(n)$

1
2
3
4
:
:
:
n

Dynamic Programming

- Notes:
 - There is no recursion(only recursive formula)
 - Memoization
 - Dynamic programming is faster. Less overhead from avoiding recursion.
 - Dynamic programming is easier to write
 - Simpler to analyze running time of DP algorithms.

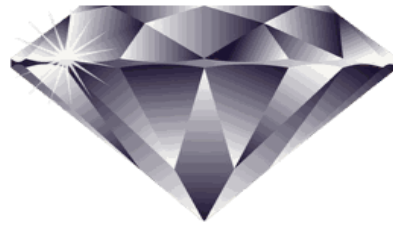
“Programming” in “Dynamic Programming”

- Has Nothing to Do with Programming!
- Richard Bellman developed this idea in 1950s working on an Air Force project. At that time, his approach seemed completely impractical. He wanted to hide that he is really doing math from the Secretary of Defense.
- **“What name could I choose? I was interested in planning but planning, is not a good word for various reasons. I decided therefore to use the word, “programming” and I wanted to get across the idea that this was dynamic. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.”**

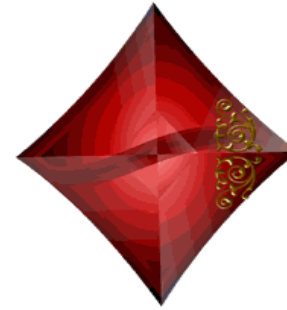
Knapsack Problem



Wt. = 5
Value = 10



Wt. = 3
Value = 20



Wt. = 8
Value = 25



Wt. = 4
Value = 8



Maximum wt. = 13

Knapsack Problem

Example:

Objects	1	2	3	4
Value	15	10	8	1
Weights	15	12	10	5



Optimal Solution?

Greedy Solution?

Optimal Value = 18 item 2&3

Greedy Value = 16 item 1&4

Knapsack Problem

- Input: n objects with
 - integer weights w_1, \dots, w_n
 - integer values v_1, \dots, v_n
 - a total capacity W .
- Output: The set S of objects that Maximize Objective Function and satisfy constraints
- Objective Function : $\sum_{i=1}^n v_i x_i$ (summation of the *number of items taken * its value*)
- Constraint: : $\sum_{i=1}^n w_i x_i \leq W$ where W is the maximum Capacity of Knapsack
the weight of all the items should be less than the maximum weight

Knapsack Problem

Example:

Objects	1	2	3	4
Value	15	10	8	1
Weights	15	12	10	5



Optimal Solution?

Greedy Solution?

Optimal Value = 18 item 2&3

Greedy Value = 16 item 1&4

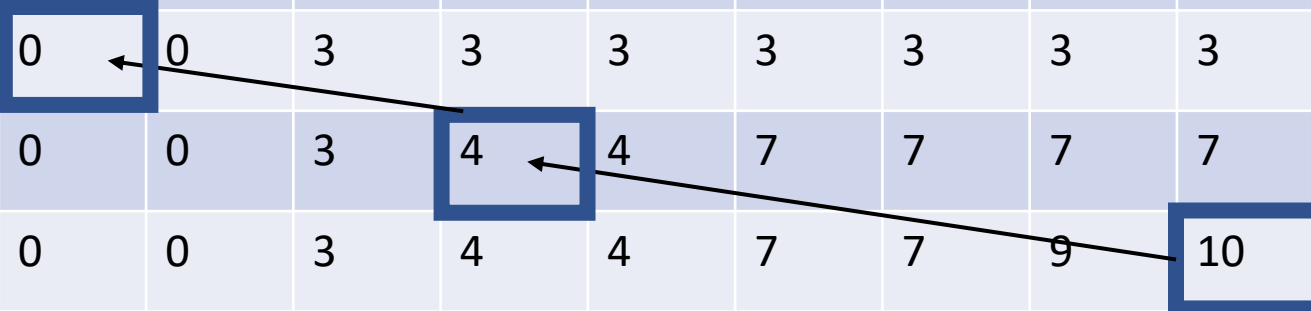
Knapsack 0-1 using DP

- Example

item	1	2	3
w	2	3	5
v	3	4	6

W=8

	0	1	2	3	4	5	6	7	8
No items	0	0	0	0	0	0	0	0	0
Item 1	0	0	3	3	3	3	3	3	3
Item 1,2	0	0	3	4	4	7	7	7	7
item 1,2,3	0	0	3	4	4	7	7	9	10



Algorithm

```
//w array of weights  
//w array of values  
//n number of items  
// W capacity
```

DP_0-1knapsack(w,v,n,W):

Create 2D array T[n+1][W+1]

for j=0 to W : //Capacity Counter

 T[0][j]=0

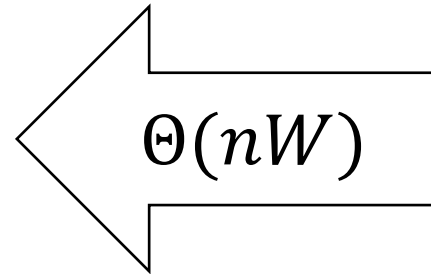
for i=1 to n: // i item counter

 if w[i]>j: //item doesnot fit

 t[i][j]=t[i-1][j]

 else

$$t[i][j] = \max \begin{cases} v[i] + t[i-1][j-w[i]] & \text{take the item} \\ t[i-1][j] & \text{donot take the item} \end{cases}$$



Algorithm

Find items(t, n, W)

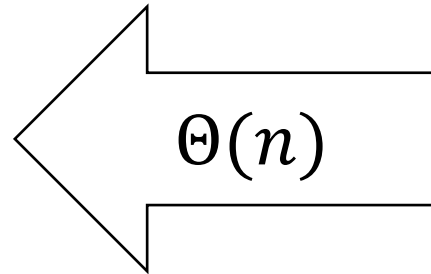
$j = W$

for $i = n$ to 1:

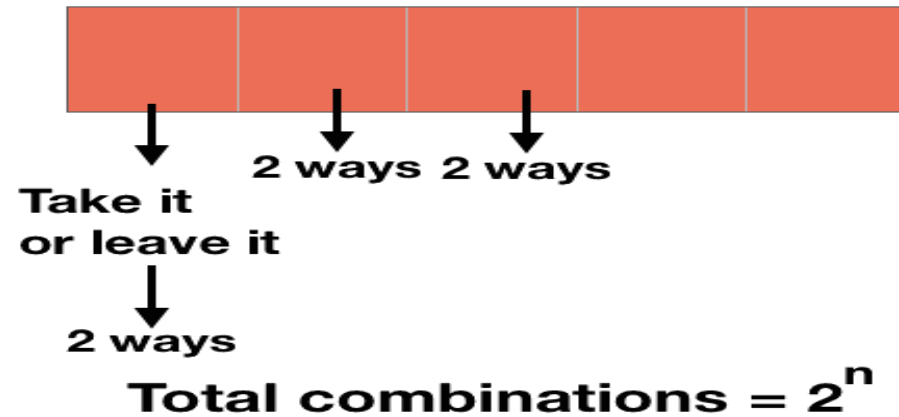
 if $t[i][j] > t[i-1][j]$:

 take item $[i]$

$j = j - w[i]$



- 1) using brute force i.e., checking each possibility (two options for each item either we can take it or leave it)



Longest Increasing subsequences(LIS)

- Input: n number a_1, a_2, \dots, a_n

- Output : **length of LIS**

- Example

5	7	4	-3	9	1	10	4	5	8	9	3
---	---	---	----	---	---	----	---	---	---	---	---

- Substring: a set of consecutive elements

- Sequences: subset of elements in order (can skip)

- 5,7,3

- 4,9,10

- 4,4,8,9

- LIS=6 \rightarrow -3,1,4,5,8,9 **Len=6**

LIS Quiz

- Example

7	11	-5	-2	15	1	16	6	7	11	8	9	0
---	----	----	----	----	---	----	---	---	----	---	---	---

- LIS=-5,-2,1,6,7,8,9

LIS Formulation(Attempt 1)

- Let $L(i)$ length of a_1, a_2, \dots, a_i
- Express $L(i)$ in terms of $L(1), \dots, L(i-1)$

5	7	4	-3	9	1	10	4	5	8	9	3
---	---	---	----	---	---	----	---	---	---	---	---

1	2	2	2	3	3	4	4	4	??		
---	---	---	---	---	---	---	---	---	----	--	--

- $L(i)=4$ 5,7,9,10
- Another
- $L(i)=4$ -3,1,4,5,8
- Need: Length of LIS for every ending character

LIS – updated Formulation

- Let $L(i)$ length of a_1, a_2, \dots, a_i which include a_i
- Express $L(i)$ in terms of $L(1), \dots, L(i-1)$

5	7	4	-3	9	1	10	4	5	8	9	3
1	2	1	1	3	2	4	3	4	5		

- $L(i) = 1 + \max_{1 \leq j \leq i-1} L(j) \mid a_j < a_i$

LIS- Algorithm

LIS(a_1, a_2, \dots, a_n)

for i=1 to n:

 L(i)=1

 for j =1 to i-1

 if $a[j] < a[i] \& L[i] < 1 + L[j]$

 L[i]=1+L[j]

max=1

for i=2 to n

 if $L[i] > L[\text{max}]$

 max=i

return (L[max])

LIS- Running Time

- What is the running time for the LIS algorithm?
- $O(n^2)$

Longest Increasing Subsequence Algorithm

```
LIS( $a_1, a_2, \dots, a_n$ ) :  
  for  $i = 1 \rightarrow n$  :  
     $L(i) = 1$   
    for  $j = 1 \rightarrow i - 1$  :  
      if  $a_j < a_i$  &  $L(i) < 1 + L(j)$  :  
        then  $L(i) = 1 + L(j)$   
 $max = 1$   
for  $i = 2 \rightarrow n$  :  
  if  $L(i) > L(max)$  then  $max = i$   
return  $L(max)$ 
```