Dynamic Programming

Tanjina Helaly

Dynamic programming (DP)

- Like the divide-and-conquer method, it solves problems by combining the solutions of subproblems.
- What is the difference the?
  - divide-and-conquer algorithms partition the problem into **disjoint** subproblems
  - In contrast, dynamic programming applies when the subproblems **overlap**—that is, when subproblems share subsubproblems.

Dynamic programming (DP)

- It solves each subsubproblem just once (just the first time) and then saves its answer in a table,
- And at any subsequent time if it needs to solves the same subsubproblem – just use it from the table.
  - this simple idea can sometimes transform **exponential-time** algorithms into **polynomial-time** algorithms.
  - Otherwise it will be normal brute force technique.
- So, we can call DP a **smart/clever Brute force** technique.

Dynamic programming (DP)

- DP typically applies to **optimization** problems in which we make a set of choices in order to arrive at an optimal solution.
  - Either maximize or minimize something

- Dynamic programming is effective when a given subproblem **may arise from more than one** partial set of choices;

- So, DP can be think of as
  - **Overlapped** subproblems that can be reused
  - **Exhaustive** search but in a **clever** way
    - As it will consider all possibilities in come to a solution not just one greedy choice.

STEPS OF DP

1. Characterize the structure of an optimal solution.
   - Define subproblem
   - Guess part of the solution
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
   - **Memoize** or **bottom-up** fashion
4. Construct an optimal solution from computed information.

Steps of DP

- Steps 1–3 form the basis of a dynamic-programming solution to a problem.
- If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.
  - When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

Fibonacci Number

Let's think about Fibonacci number

$$\underline{fib(n):}$$

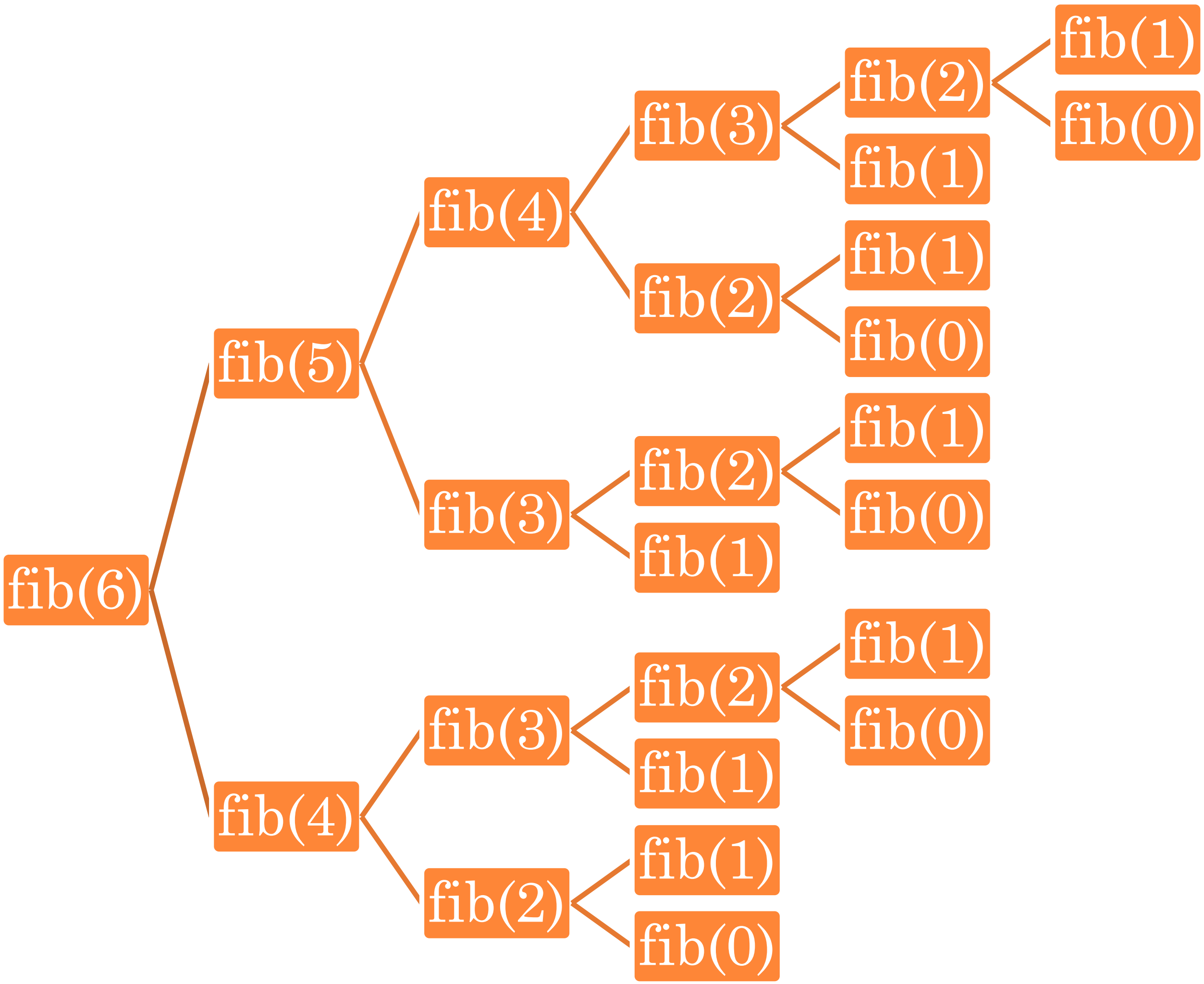
$$if\ (n<2)\ \ f=n;$$

$$else\ \ f\ =\ fib(n-1)+fib(n-2)$$

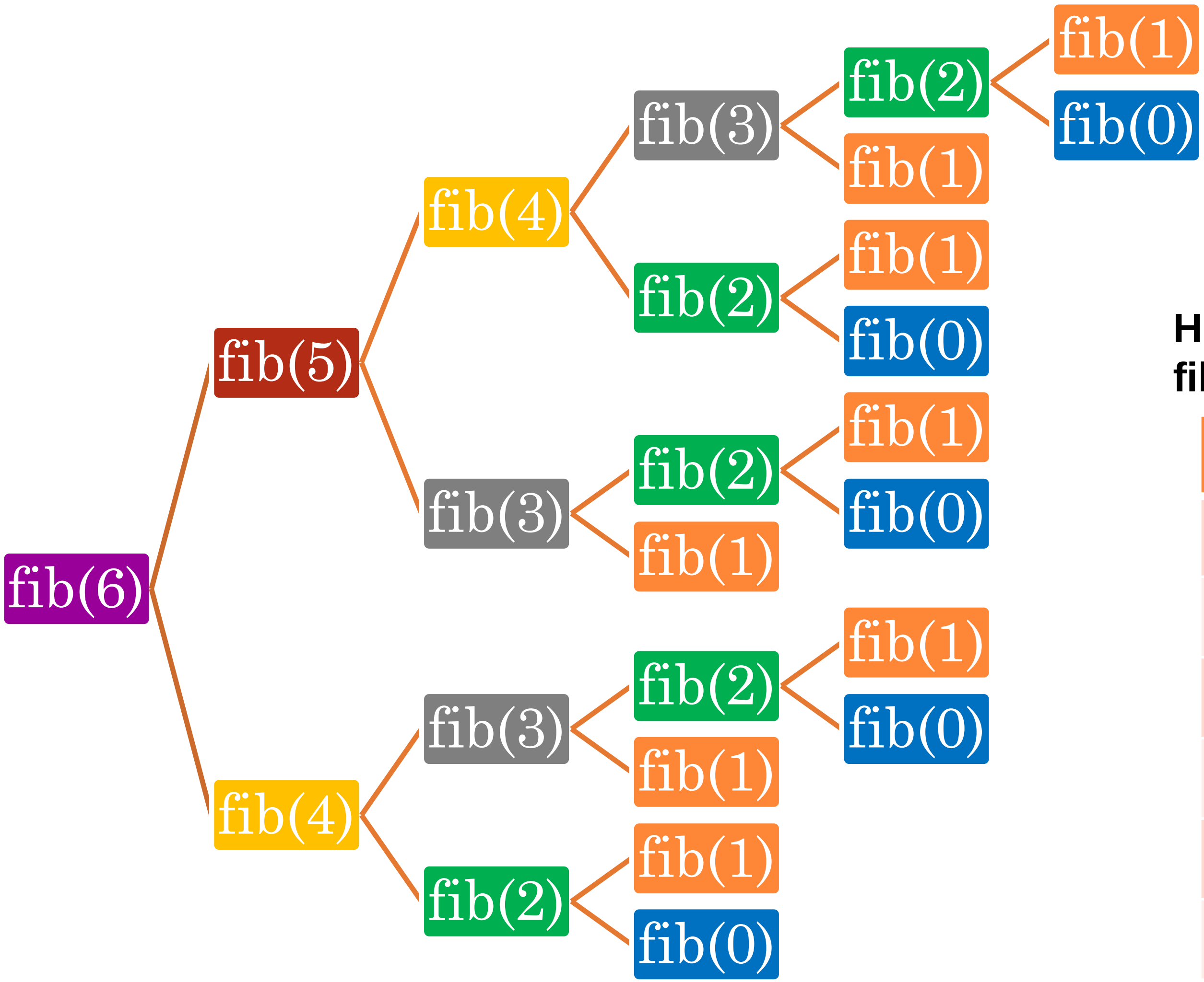
$$return\ \ f;$$

- Time Complexity: $\Theta(2^{n/2})$
- Is it a good algorithm?
- Is there any way to improve?

Fibonacci number – recursion tree for n=6



Fibonacci number – recursion tree for n=6



How many times each fib(n) is called?

Function	Count
fib(5)	1
fib(4)	2
fib(3)	3
fib(2)	5
fib(1)	7
fib(0)	5

Improvement – memoization (remembering)

- Calculate once
- Store it
- And reuse it

Fibonacci with memoization

$Let\ F[n]\ be\ an\ array\ or\ dictionary$

$\underline{fib(n):}$

$if\ F[n]\ has\ a\ value\ return\ F(n)$

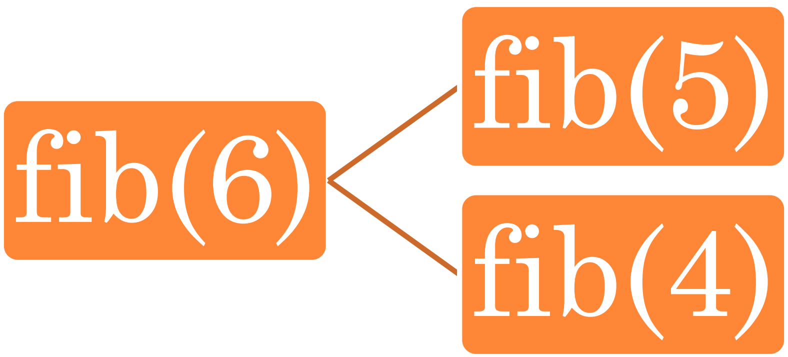
$if\ (n<2)\ f=n;$

$else\ f=fib(n-1)+fib(n-2)$

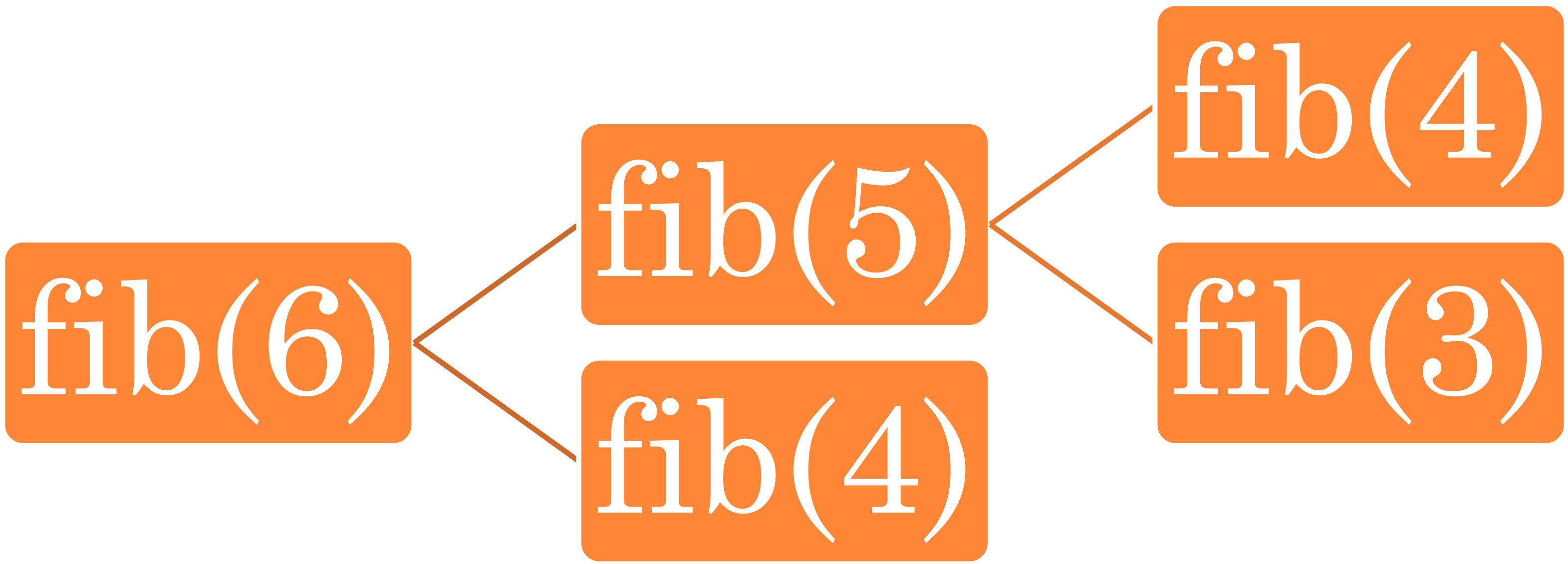
$F[n]=f$

$return\ f;$

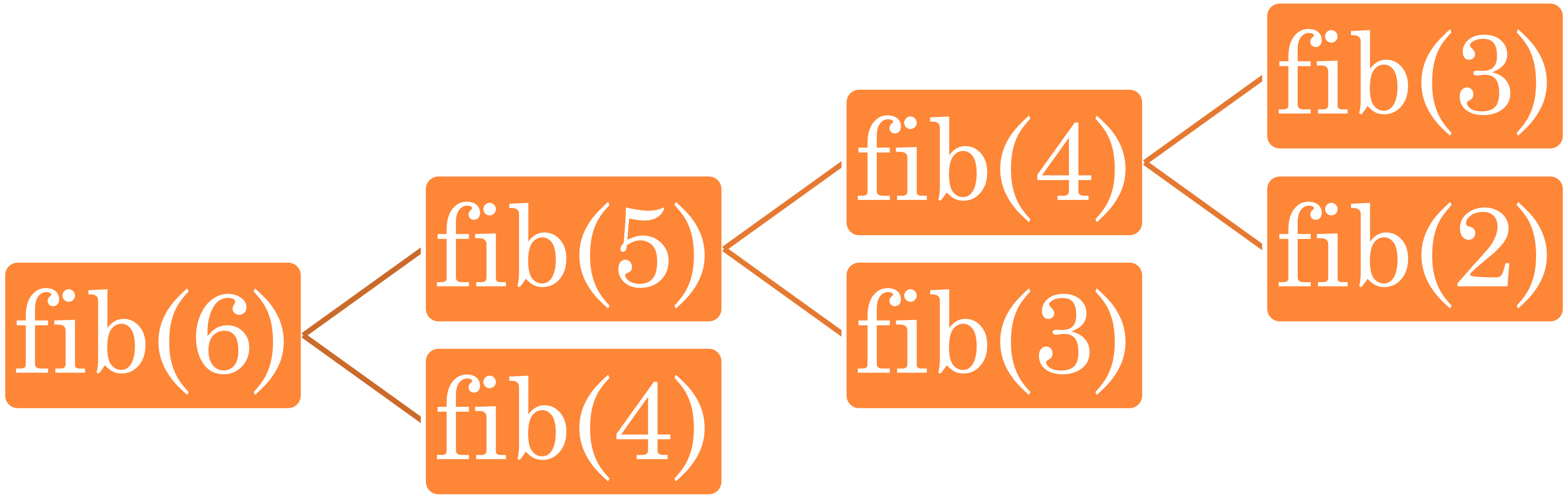
Fibonacci number – divide step



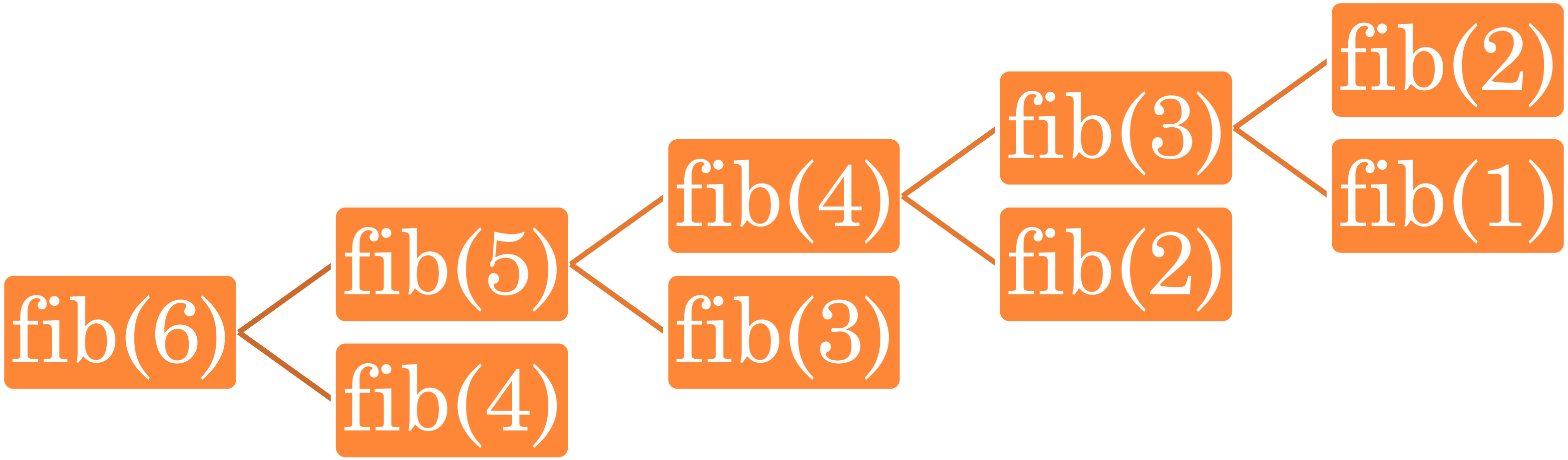
Fibonacci number – divide step Cont..



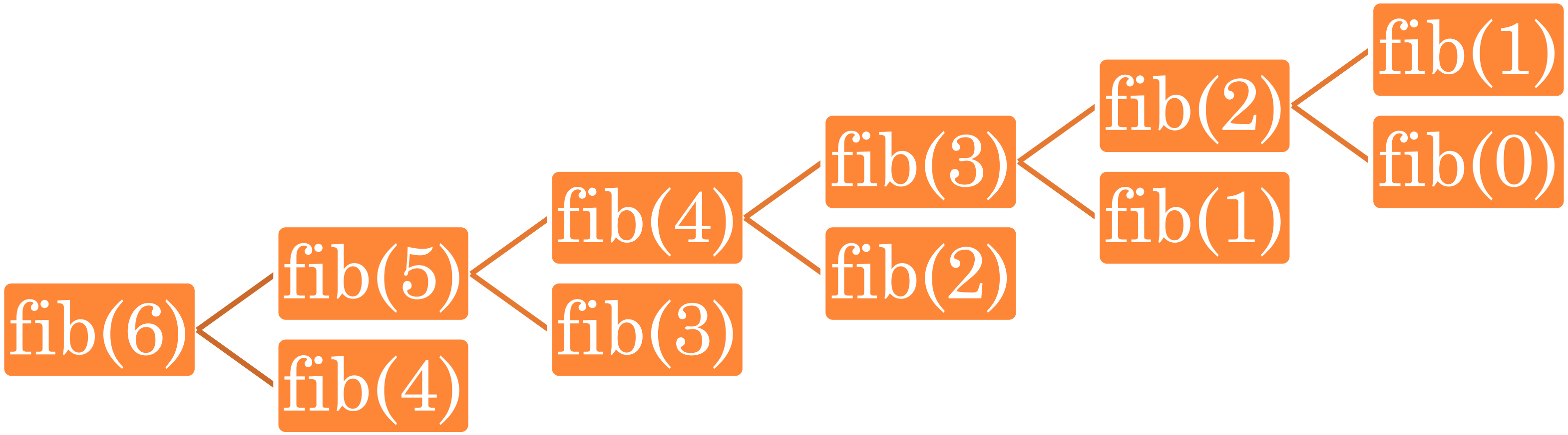
Fibonacci number – divide step Cont..



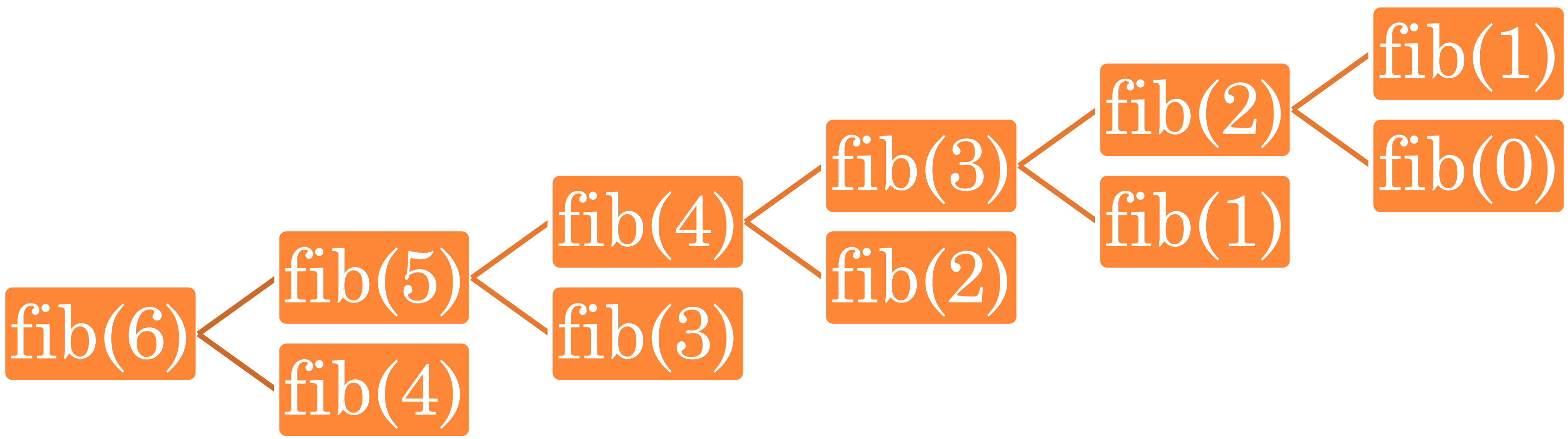
Fibonacci number – divide step Cont..



Fibonacci number – divide step Cont..

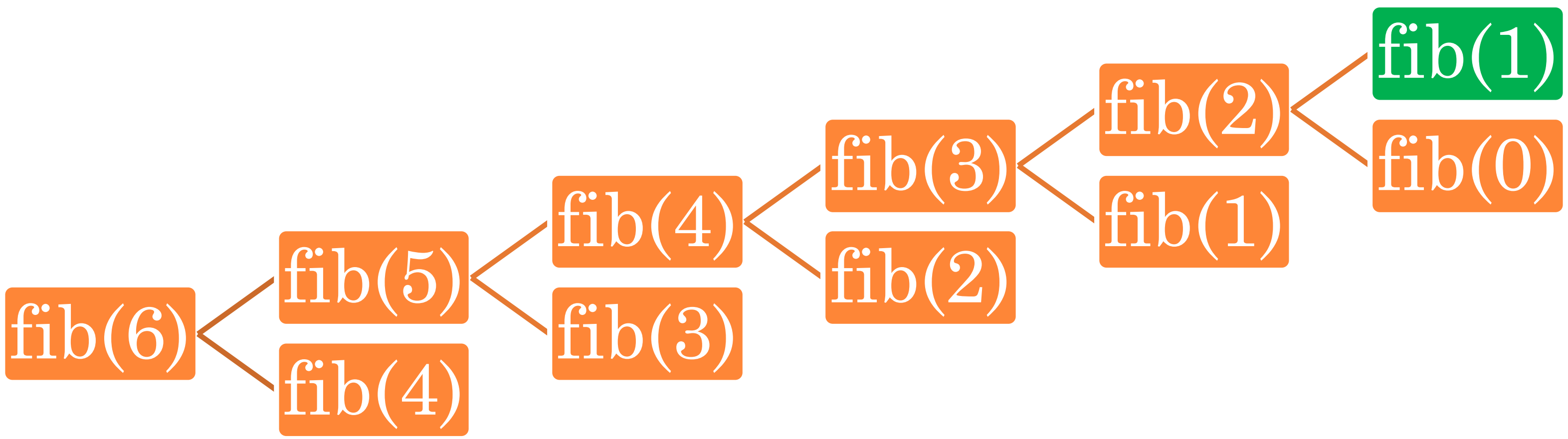


Fibonacci number – divide step Cont..



Function	value
fib(6)	
fib(5)	
fib(4)	
fib(3)	
fib(2)	
fib(1)	
fib(0)	

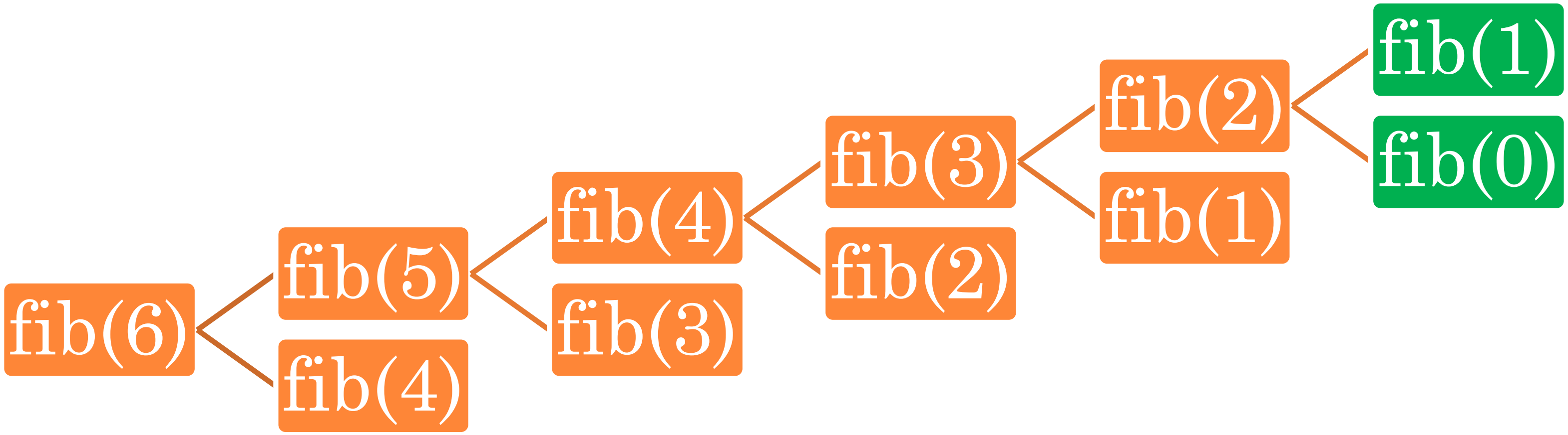
Fibonacci number – conquer step



Function	value
fib(6)	
fib(5)	
fib(4)	
fib(3)	
fib(2)	
fib(1)	1
fib(0)	

fib(n) Indicates calculated and saved to table

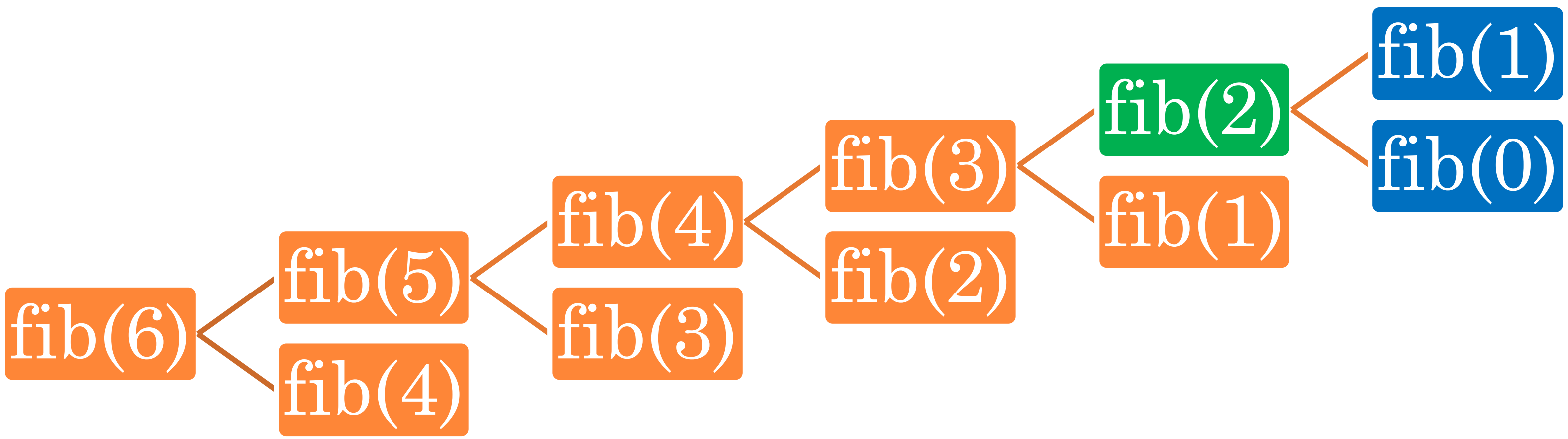
CONQUER STEP CONT..



Indicates calculated and saved to table

Function	value
fib(6)	
fib(5)	
fib(4)	
fib(3)	
fib(2)	
fib(1)	1
fib(0)	0

Combine & conquer step Cont..

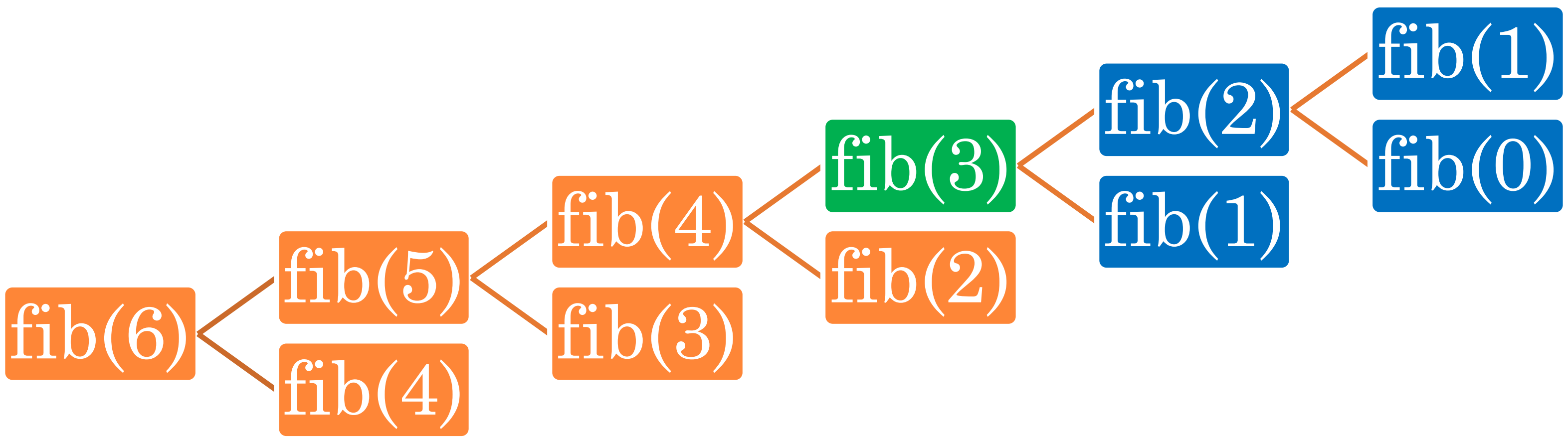


Function	value
fib(6)	
fib(5)	
fib(4)	
fib(3)	
fib(2)	1
fib(1)	1
fib(0)	0

fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

Combine & conquer step Cont..

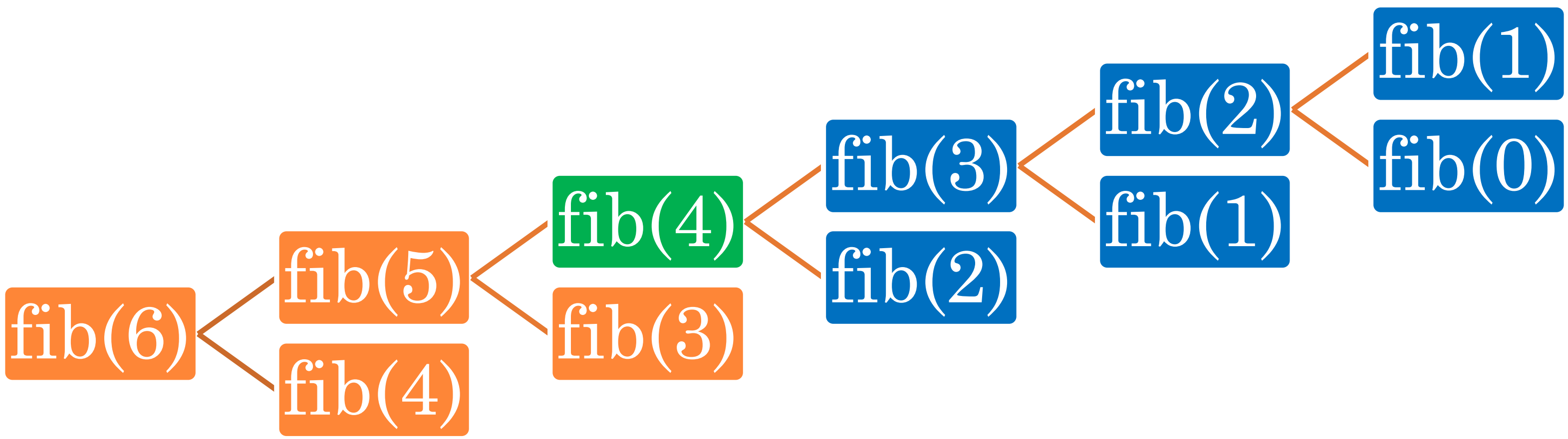


fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

Function	value
fib(6)	
fib(5)	
fib(4)	
fib(3)	2
fib(2)	1
fib(1)	1
fib(0)	0

Combine & conquer step Cont..

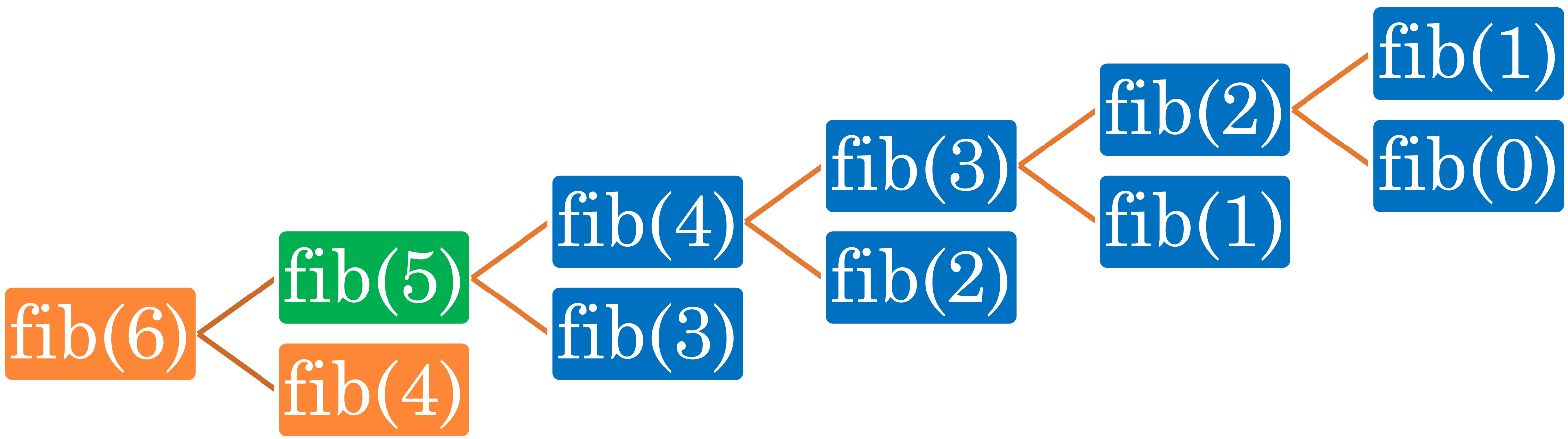


fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

Function	value
fib(6)	
fib(5)	
fib(4)	3
fib(3)	2
fib(2)	1
fib(1)	1
fib(0)	0

Combine & conquer step Cont..

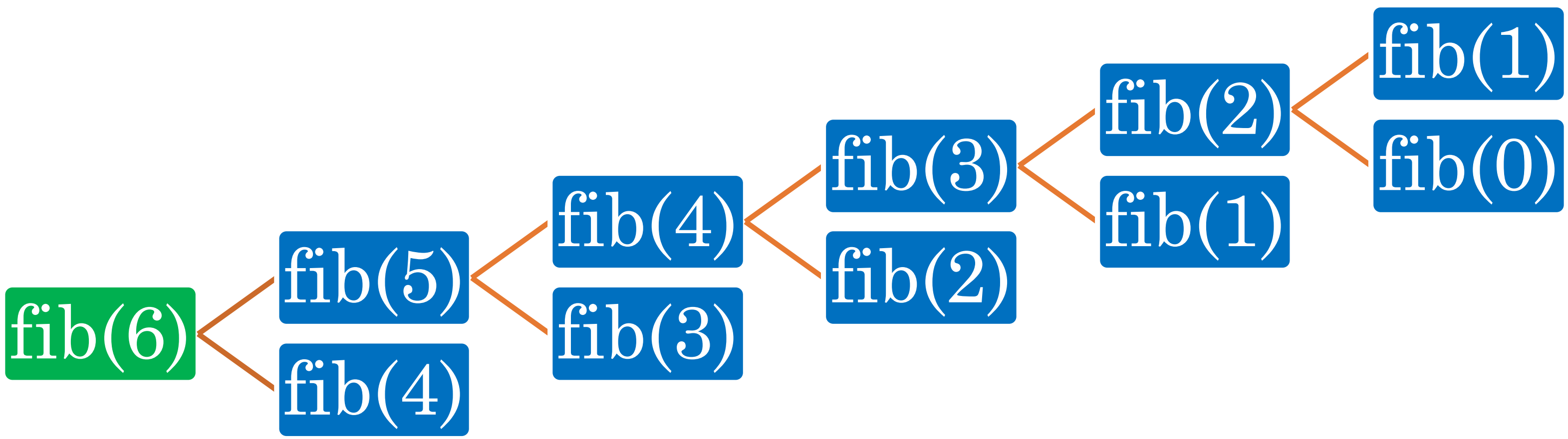


Function	value
fib(6)	
fib(5)	5
fib(4)	3
fib(3)	2
fib(2)	1
fib(1)	1
fib(0)	0

fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

Combine & conquer step Cont..

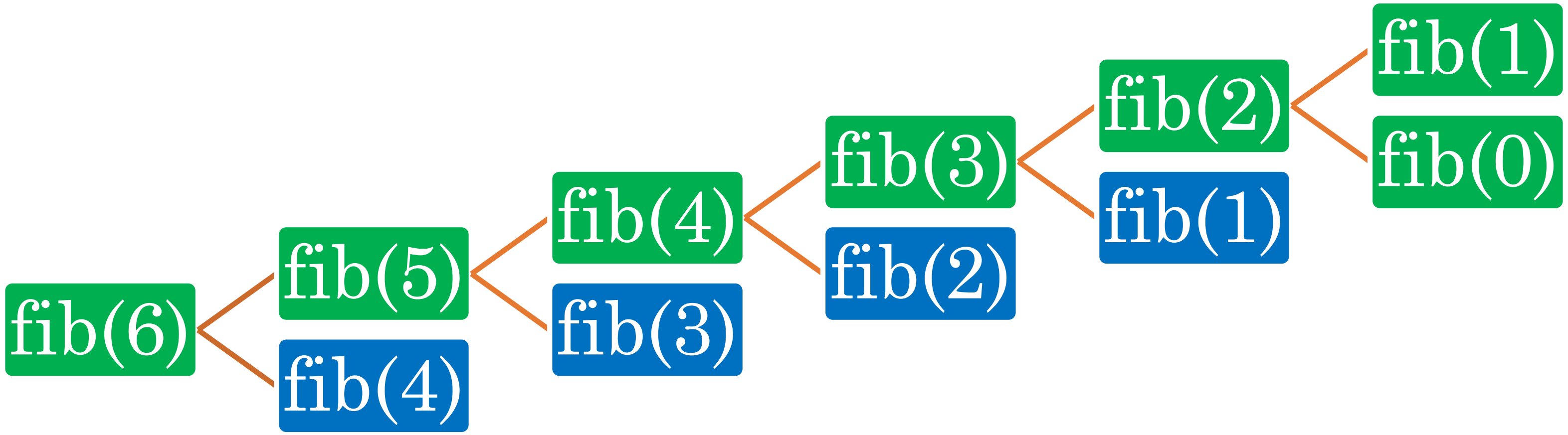


fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

Function	value
fib(6)	8
fib(5)	5
fib(4)	3
fib(3)	2
fib(2)	1
fib(1)	1
fib(0)	0

Combine & conquer step Cont..



Indicates the steps where we calculated.

- Time Complexity= $\Theta(n)$

When do we use memoization

- When a problem has following 2 properties:
  - **Optimal Substructure**: A problem depends on the solution of the sub-problems.
  - **Overlapping Subtructure**: Sub-problems are called several times.

Fibonacci: Bottom-up approach(Tabular approach)

$$Let\ F[n]\ be\ an\ array\ or\ dictionary$$

$\underline{fib(n):}$

$for\ k = 0\ to\ n$

$\quad if(k < 2)$

$\quad\quad F[k] = k;$

$\quad else$

$\quad\quad F[k] = F[k-1] + F[k-2]$

$return\ F[n];$

- Time Complexity= Θ(n)

0-1 Knapsack

0-1 KNAPSACK

- **Definition:** Given items of different values and volumes, find the most valuable set of items that fit in a knapsack of fixed volume.
- **Formal Definition:** There is a knapsack of capacity $c > 0$ and N items. Each item has value $v_i > 0$ and weight $w_i > 0$. Find the selection of items ($\delta_i = 1$ if selected, 0 if not) that fit, $\sum_{i=1}^{N} \delta_i w_i \leq c$, and the total value, $\sum_{i=1}^{N} \delta_i v_i$, is maximized.

0-1 KNAPSACK

- Assume
  - the knapsack can hold 10 lb. So, **c = 10**.
  - And the following items are available.
  - We need to find the most valuable items that will fit into our knapsack.

Item#	1	2	3	4	5
Value	7	2	1	6	12
Weight	3	1	2	4	6

0-1 KNAPSACK-HOW TO SOLVE?

Item#	1	2	3	4	5
Value	7	2	1	6	12
Weight	3	1	2	4	6

- Brute Force:
  - Start from the beginning and check it we can maximize the value either by including or excluding the item
- DP
  - GUESS: either an item will be included or excluded
  - SubProblem: remaining items and remaining capacity

0-1 KNAPSACK-BRUTE FORCE

$// \ c-> weight \ needs \ to \ be \ filled \ in.$

$// \ i-> item \ that \ we \ are \ working \ on$

$// \ n-> number \ of \ the \ item$

$\underline{KS(c,i):}$

$if \ (i > n) \ return \ 0;$

$if \ (c < w[i]) \quad // \ item's \ weight \ is \ more \ than \ the \ capacity \ So, exclude \ item$

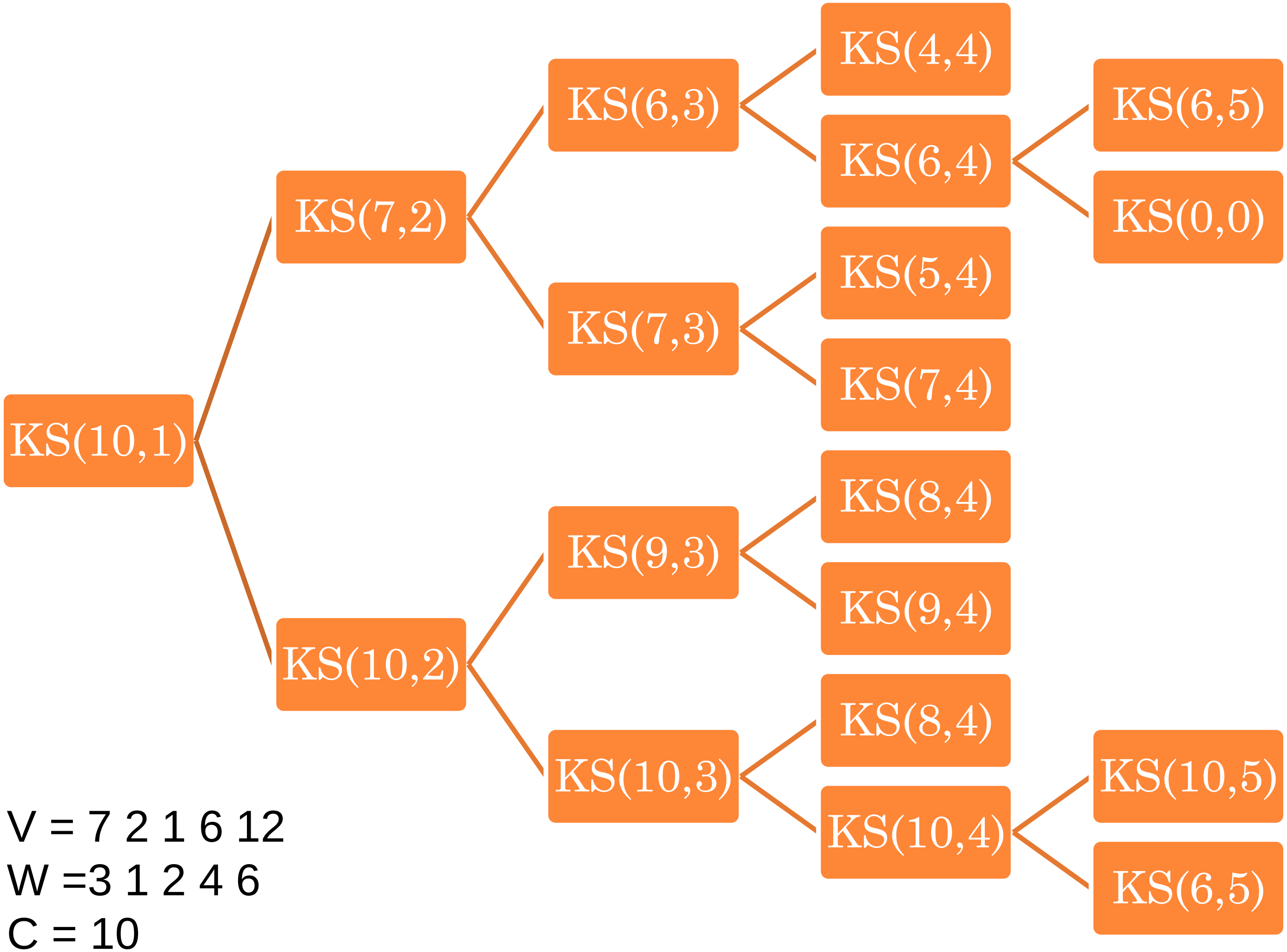
$return \ KS(c, i+1)$

$else \quad // \ item \ will \ fit \ in \ the \ sack$

$// \ try \ by \ both \ including \ and \ excluding \ the \ item \ and \ take \ the \max \ of \ those \ 2$

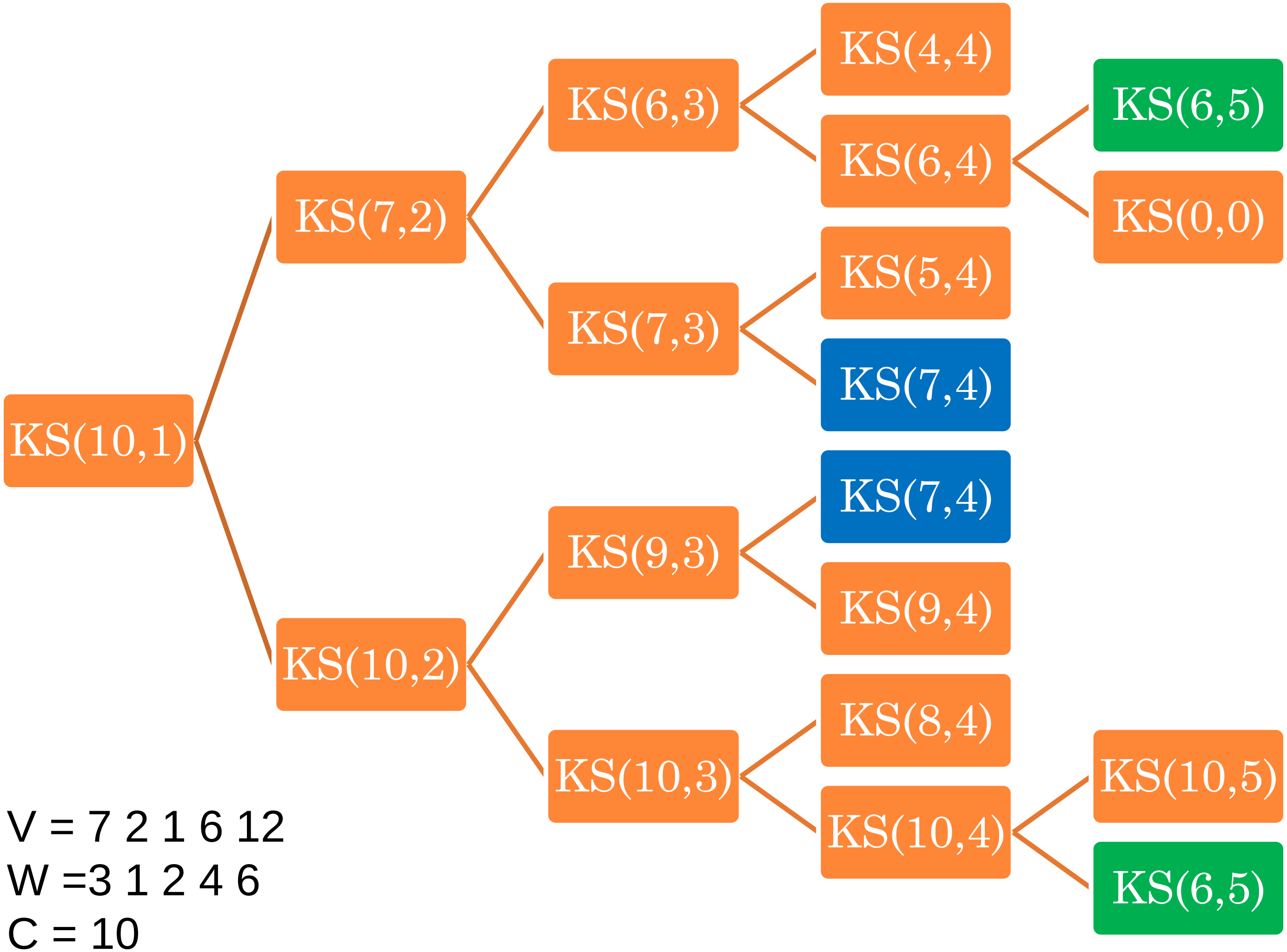
$return \quad \max(KS(c, i+1), v[i] + KS(c - w[i], i+1))$

0-1 KNAPSACK



V = 7 2 1 6 12
W =3 1 2 4 6
C = 10

KNAPSACK–OVERLAPPING SUBPROBLEM ?



V = 7 2 1 6 12
W =3 1 2 4 6
C = 10

0-1 KNAPSACK – WITH MEMOIZATION (TOP-DOWN)

$Assume \quad T[c,i] \rightarrow table\ to\ store \quad KS's\ value$

$\underline{KS(c,i)}$:

$if\ T[c,i]\ has\ a\ value\ return\ T[c,i];$

$if(i > n)\ k = 0;$

$else$

$if(c < w[i])$

$//item's\ weight\ is\ more\ than\ the\ capacity\ So, exclude\ item$

$k = KS(c, i+1)$

$else\ //item\ will\ fit\ in\ the\ sack. try\ by\ both\ including$

$//\ and\ excluding\ the\ item\ and\ take\ the\ \max\ of\ those\ 2$

$k = \max(KS(c, i+1), \quad v[i] + KS(c - w[i], i+1))$

$T[c,i] = k;$

$return\ k;$

0-1 KNAPSACK (BOTTOM-UP)

Assume $T[i,c]$ −> *table to store KS's value. T[0,c] =0 and T[i,0]=0 as no item can be added to the bag.*

$\underline{KS(c,i)}$:

for i = 1 to n

if (w[i]>c) //item's weight>capacity. So, exclude the item,

T[i, c] = T[i-1,c]

else //item will fit in the sack, try both exclude and including the item,

T[i, c] = $\max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0											
1	2	1											
2	1	2											
3	7	3								A			
4	6	4											
5	12	6											

Capacity

Any cell in the table represents the maximum value attained by choosing items from i items (not i^{th}) in a sack of capacity listed in the header. For example, the cell with value "A" represents that we can add items of total value "A" from 3 items and with a sack capacity=7 which is represented as T[3,7] = A

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(\ T(i-1, c),\ v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	**v_i**	**w_i**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0											
1	2	1											
2	1	2											
3	7	3											
4	6	4											
5	12	6											

Capacity

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(\ T(i-1, c),\ \ v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	**v_i**	**w_i**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1											
2	1	2											
3	7	3											
4	6	4											
5	12	6											

Capacity

If i=0, no items are available, to put to the sack, the maximum value we can attain is 0.

KNAPSACK SIMULATION

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(\ T(i-1, c),\ \ v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

If bag capacity is 0, we can't
add anything into the sack.
So, attained value is 0.

Knapsack simulation

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(\ T(i-1,c),\ v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

T[1,1] = Max(T[1-1,1], v_1 + T[1-1,1-1])
= Max(T[0,1], 2 + T[0,0])
= Max(0, 2+0) = 2

KNAPSACK SIMULATION

if (w[i]>c)
T[i, c] = T[i-1,c]
else
$T[i,c] = \max(\ T(i-1,c),\ v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

$T[1,1] = \text{Max}(T[1\text{-}1,1],\ v_1 + T[1\text{-}1,1\text{-}1])$
$= \text{Max}(\ T[0,1],\ 2 + T[0,0])$
$= \text{Max}(0,\ 2+0) = 2$

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(\ T(i-1, c),\ v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2									
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

$$\begin{aligned} T[1,1] &= \text{Max}(T[1-1,1],\ v_1 + T[1-1,1-1]) \\ &= \text{Max}(\ T[0,1],\ 2 + T[0,0]) \\ &= \text{Max}(0,\ 2+0) = 2 \end{aligned}$$

KNAPSACK SIMULATION

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(T(i-1, c),\ v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2								
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

Going w_1 cell back

$$T[1,2] = \text{Max}(T[1-1,1],\ v_1 + T[1-1,2-1])$$
$$= \text{Max}(\ T[0,1],\ 2 + T[0,1])$$
$$= \text{Max}(0,\ 2+0) = 2$$

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	$\mathbf{v_i}$	$\mathbf{w_i}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2								
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

T[1,2] = Max(T[1-1,1], v_1 + T[1-1,2-1])
= Max(T[0,1], 2 + T[0,1])
= Max(0, 2+0) = 2

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

For any c >= 1,
T[1,c] = Max(T[1-1,1], v_1 + T[1-1,c-1])
= Max(T[0,1], 2 + T[0,c-1])
= Max(0, 2+0) = 2

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	**v_i**	**w_i**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

T(2,1)= T[2-1,1] as $w_2 > c$
= T[1,1] = 2

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2									
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

T(2,1)= T[2-1,1] as $w_2 > c$
= T[1,1] = 2

KNAPSACK SIMULATION

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2									
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

Going w_2 cell back

T[2,2] = Max(T[2-1,2], v_2 + T[1-1, 2-2])
= Max(T[1,2], 1 + T[1,0])
= Max(2, 1) = 2

Knapsack simulation

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2								
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

T[2,2] = 2

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2								
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

Going w_2 cell back

T[2,3] = Max(T[2-1, 3], v_2 + T[2-1, 3-2])
= Max(T[1,3], 1+ T[1,1])
= Max(2, 3) = 3

Knapsack simulation

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

So, simplest version is compare 1) the cell above the current cell and 2) v_i + value of w_i cell backward in previous row. Populate the current cell with whichever value is bigger.

Populate the table with this logic.

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

How to find the items that are in the sack?

- 1. Start from the bottom right corner where i=n and sack has the full capacity.
- 2. Compare the value with the cell above it
- 3. If the values are equal
  - Then the item "i" is not included in the sack as its value do not have any impact in total value.
  - Pick the cell above the current cell as next cell.
- 4. Else
  - The item is included in the sack.
  - Go one row up (i-1) and go w_i cell backward. This is the next cell to check.
- Repeat Step#2-4 until you reached the 0^{th} row (i=0)

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

Sack = {}

1. Start with 21 (Green cell) and compare with the one above it (16).
2. As 21 and 16 are not equal item# 5 is included in the sack.
3. Go 6(weight if item) units back in previous row which is the next cell to check.

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	19

Capacity

included

Sack = {5}

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

Not Included

Included

Sack = {5}

1. Compare T[4,4] 9 (Green cell) with the one above it (9).
2. As both cell has same value item# 4 is not included in the sack.

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	1	0	2	2	2	2	2	2	2	2	2	2	
2	1	2	0	2	2	3	3	3	3	3	3	3	3	
3	7	3	0	2	2	7	9	9	10	10	10	10	10	Included
4	6	4	0	2	2	7	9	9	10	13	15	15	16	Not Included
5	12	6	0	2	2	7	9	9	12	14	15	19	21	Included

← Capacity

Sack = {5, 3}

1. Compare T[3,4] 9 (Green cell) with the one above it (3).
2. As the cells have different values item# 3 is included in the sack.
3. Go 3 units back in previous row which is the next cell to check.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	← Capacity
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	1	0	2	2	2	2	2	2	2	2	2	2	
2	1	2	0	2	2	3	3	3	3	3	3	3	3	Not Included
3	7	3	0	2	2	7	9	9	10	10	10	10	10	Included
4	6	4	0	2	2	7	9	9	10	13	15	15	16	Not Included
5	12	6	0	2	2	7	9	9	12	14	15	19	21	Included

Sack = {5, 3}

1. Compare T[2,1] 2 (Green cell) with the one above it (2).
2. As both cells have same values item# 2 is not included in the sack.

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	← Capacity
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	1	0	2	2	2	2	2	2	2	2	2	2	Included
2	1	2	0	2	2	3	3	3	3	3	3	3	3	Not Included
3	7	3	0	2	2	7	9	9	10	10	10	10	10	Included
4	6	4	0	2	2	7	9	9	10	13	15	15	16	Not Included
5	12	6	0	2	2	7	9	9	12	14	15	19	21	Included

Sack = {5, 3, 1}

1. Compare T[1,1] 2 (Green cell) with the one above it (0).
2. As the cells have different values item# 1 is included in the sack.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	← Capacity
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	1	0	2	2	2	2	2	2	2	2	2	2	**Included**
2	1	2	0	2	2	3	3	3	3	3	3	3	3	Not Included
3	7	3	0	2	2	7	9	9	10	10	10	10	10	**Included**
4	6	4	0	2	2	7	9	9	10	13	15	15	16	Not Included
5	12	6	0	2	2	7	9	9	12	14	15	19	21	**Included**

Sack = {5, 3, 1}
As we have reached the 0^{th} row, we are done with item selection. So, the sack contains **1, 3 and 5** item with value = 2+7+12 = 21

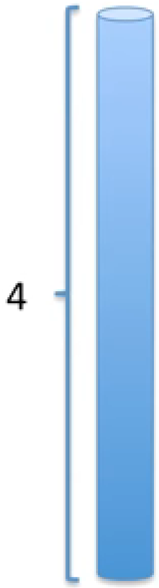
Rod-cutting problem

Rod cutting problem

- Given
  - a rod of length n inches and
  - an array of prices that contains prices of all pieces of size smaller than n.
- Determine
  - the **maximum** value obtainable by **cutting** up the rod and selling the pieces.

EXAMPLE

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30

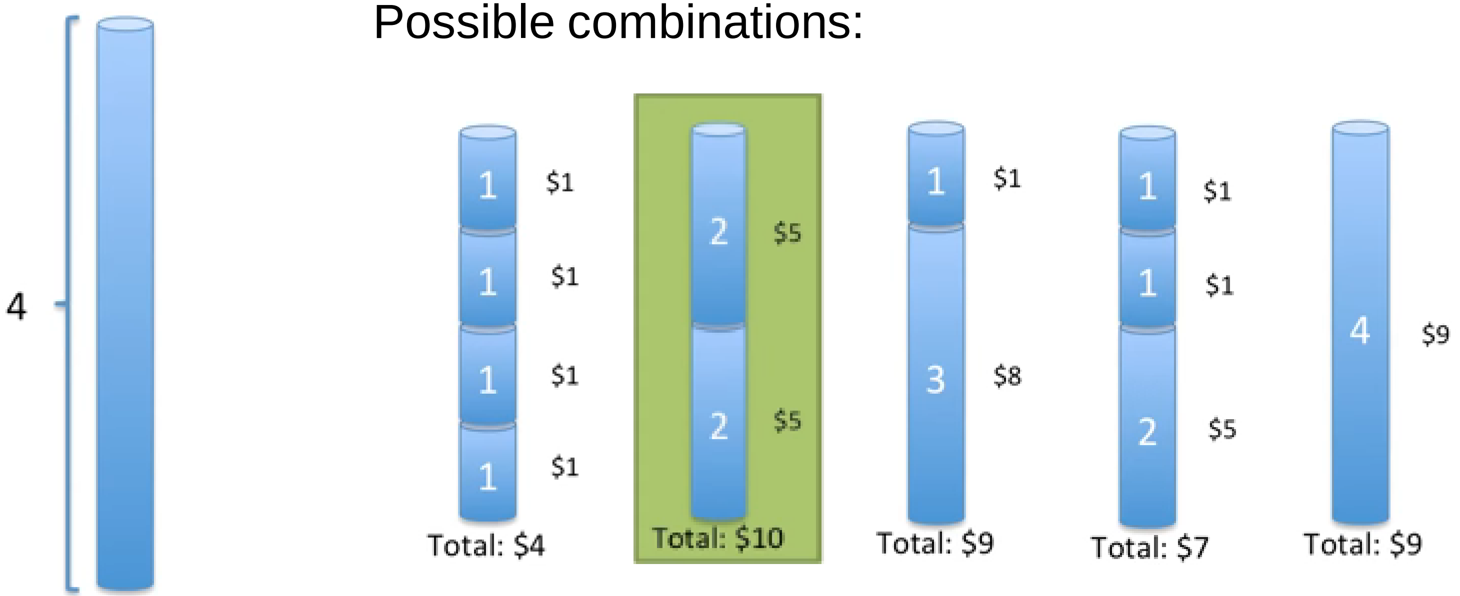


- Assume you have a rod of length 4 inch.
- A table that shows the price of all different length of rod.
- How can we cut the rod so that we can maximize the profit?

Example cont..

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30

Possible combinations:



How to solve

- Brute force
  - Try all possible combination- 2^{n-1} combination
- Dynamic programming
  - Time complexity – n^2

Brute Force solution

- Assume for optimal solution we cut the rod as below
  - $n = i_1 + i_2 + ... + i_k$ for $1 \le k \le n$ and i_k is the rod of k inch
- So, optimum price
  - $r_n = p_{i1} + p_{i2} + ... + p_{ik}$ where p_{ik} is the optimum price of a k size rod
- We start by cuttin into 2 halves that will give optimum revenue

Brute Force solution cont..

- We assume the rod is cut into 2 halves of size *i* & *n-i* which will give optimum solution of the problem.
- Then find the optimum price of each of those halves by cutting into smaller pieces.
- Suppose
  - r_n -> optimum price/revenue achieved by cutting (or not cutting) a rod of length n.
  - p_n -> price of a rod of length n,
  - Then we can write

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \ldots\ldots, r_{n-1} + r_1)$$

Brute Force solution- 1 inch rod

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price										

$$r_1 = p_1 = 1 \quad (\textit{base case. No cut possible})$$

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \ldots\ldots, r_{n-1} + r_1)$$

Brute Force solution- 2 inch rod

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1									

$$r_2 = \max\begin{cases} p_2 = 5 \\ r_1 + r_1 = 1+1 = 2 \\ r_1 + r_1 = 1+1 = 2 \end{cases}$$

$$= 5$$

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \ldots\ldots, r_{n-1} + r_1)$$

Brute Force solution- 3 inch rod

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1	5								

$$r_3 = \max\begin{cases} p_3 = 8 \\ r_1 + r_2 = 1 + 5 = 6 \\ r_2 + r_1 = 5 + 1 = 6 \end{cases}$$

$$= 8$$

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \ldots\ldots, r_{n-1} + r_1)$$

Brute Force solution- 4 inch rod

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1	5	8							

$$r_4 = \max\begin{cases} p_4 = 9 \\ r_1 + r_3 = 1 + 8 = 9 \\ r_2 + r_2 = 5 + 5 = 10 \\ r_3 + r_1 = 8 + 1 = 9 \end{cases}$$

$$= 10$$

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \ldots.., r_{n-1} + r_1)$$

Brute Force solution- 5 inch rod

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1	5	8	10						

$$r_5 = \max\begin{cases} p_5 = 10 \\ r_1 + r_4 = 1 + 10 = 11 \\ r_2 + r_3 = 5 + 8 = 13 \\ r_3 + r_2 = 8 + 5 = 13 \\ r_4 + r_1 = 1 + 10 = 11 \end{cases}$$

$$= 13$$

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \ldots.., r_{n-1} + r_1)$$

Brute Force solution- 6 to10 inch rod

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1	5	8	10	13	17	18	22	25	30

- With similar calculation we can fill out the remaining table
  - r_6 -> 17 (no cut)
  - r_7 -> 18 (1+6 or 2+2+3)
  - r_8 -> 22 (2+6)
  - r_9 -> 25 (3+6)
  - r_{10} -> 30 (no cut)

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \ldots\ldots, r_{n-1} + r_1)$$

Another version

- We view a decomposition as consisting of
  - a first piece of length i cut off the left-hand end,
  - and then a right-hand remainder of length n - i.
- Only the remainder, and not the first piece, may be further divided.
- Thus obtain the following simpler version of equation:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Brute Force

$\underline{Cut-Rod(p,n):}$

$if\ \ n == 0\ return\ 0;$

$q = -\infty;$

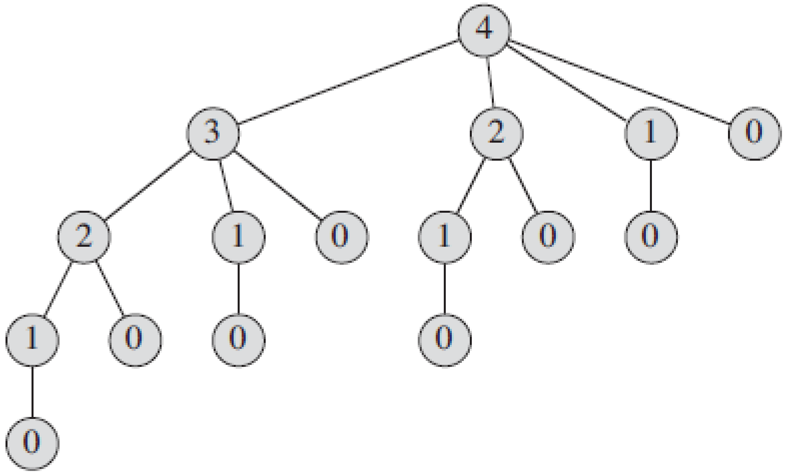
$for\ i = 1\ to\ n$

$\quad q = \max(q, p[i] + Cut-Rod(p, n-i));$

$return\quad q;$

- Complexity: $\Theta(2^n)$

BRUTE FORCE



The recursion tree showing recursive calls resulting from a call **CUT-ROD(p, n)** for n=4. Each node label gives the size **n** of the corresponding subproblem, so that an **edge** from a **parent with label s to a child with label t** corresponds to cutting off an **initial piece of size (s – t)** and leaving a **remaining subproblem of size t** .

- Complexity: $\Theta(2^n)$

Memoized version

$\underline{Memoized-Cut-Rod(p,n):}$

$let \quad r[0...n]\, be\ a\ new\ array$

$for \quad i = 0\ to\ n$

$\quad r[i] = -\infty$

$return \quad Memoized-Cut-RodAux(p,n,r);$

$\underline{Memoized-Cut-RodAux(p,n,r)}$

$if\ \ r[n] \geq 0 \quad return\ r[n];$

$if\ \ n == 0 \quad q = 0;$

$else \quad q = -\infty;$

$\quad for\ i = 1\ to\ n$

$\qquad q = \max(q, p[i] + Memoized-Cut-Rod(p, n-i, r);$

$r[n] = q;$

$return \quad q;$

- **Complexity: $\Theta(n^2)$**

BOTTOM-UP VERSION

$\underline{Bottom-Up-Cut-Rod(p,n):}$

$let \quad r[0...n]\, be\ a\ new\ array$

$r[0]=0$

$for\ \ j=1\, to\ n$

$\quad q=-\infty;$

$\quad for\ i=1\, to\ \ j$

$\quad\quad q=\max(q, p[i]+r[j-i]);$

$\quad r[j]=q;$

$return \quad r[n];$

- **Complexity: $\Theta(n^2)$**

Coin Change Problem

Coin Change Problem

- Make change of n coin with denomination $\{d_1, d_2, d_3, \ldots, d_m\}$

- Greedy works for most of the coin system.
  - {1,5,10,25}
  - {1,2,5,10,20,50}

Coin Change Problem

- But think about the following
  - Change 12 or 16 cent with {1,4,8,10} denomination
    - For 12, Greedy 10+1+1 whereas optimal is 8+4
    - For 16, Greedy 10+4+1+1 whereas optimal is 8+8
  - Change 30 with {1,10,25,50}
    - For 30, Greedy 25+1+1+1+1+1 whereas optimal is 10+10+10
  - Change 16 cent with {1,5,12,25}
    - For 16, Greedy 12+1+1+1+1 whereas optimal is 5+5+5+1

What is the solution?

- **Solution-** try all coin combinations equal or less the change we are asking and give the minimum one.
- Check if we can optimize the solution by applying DP.

Step 1: Characterize the structure of the problem

- Lets define a function that we wish to minimize, we call it objective function.
- c(n) = number of coins needed for giving change of amount n.
- We need to minimize c(n).

Step 2: Recursively define the value of an optimal solution

- The solution to the problem with an amount n must start with one of the available coins! (50,25,10,1)
- Suppose we chose to take a 25 cent. Then we get a sub-problem with an amount of n - 25
- **Each choice of coin leads** us to a sub-problem. We have to solve the subproblems in order to know which one is the best (minimum)!

$$c(n) = \begin{cases} 0 \quad if \;\; n = 0 \\ else \quad \min \begin{cases} 1 + c(n-1) \quad if \;\; n > 1 \\ 1 + c(n-10) \quad if \;\; n > 10 \\ 1 + c(n-25) \quad if \;\; n > 25 \\ 1 + c(n-50) \quad if \;\; n > 50 \end{cases} \end{cases}$$

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

```
CoinChange(n)
  if n == 0
          return 0
  else
          minValue= 1 + CoinChange(n - 1)
  if n >10
          tempVal = 1 + CoinChange(n - 10)
          if tempVal < minVal
          minVal = tempVal
  if n > 25
          tempVal = 1 + CoinChange(n -25)
          if tempVal < minVal
          minVal = tempVal
  if n >50
          tempVal = 1 + CoinChange(n-50)
          if tempVal < minVal
          minVal = tempVal
  return minVal
```

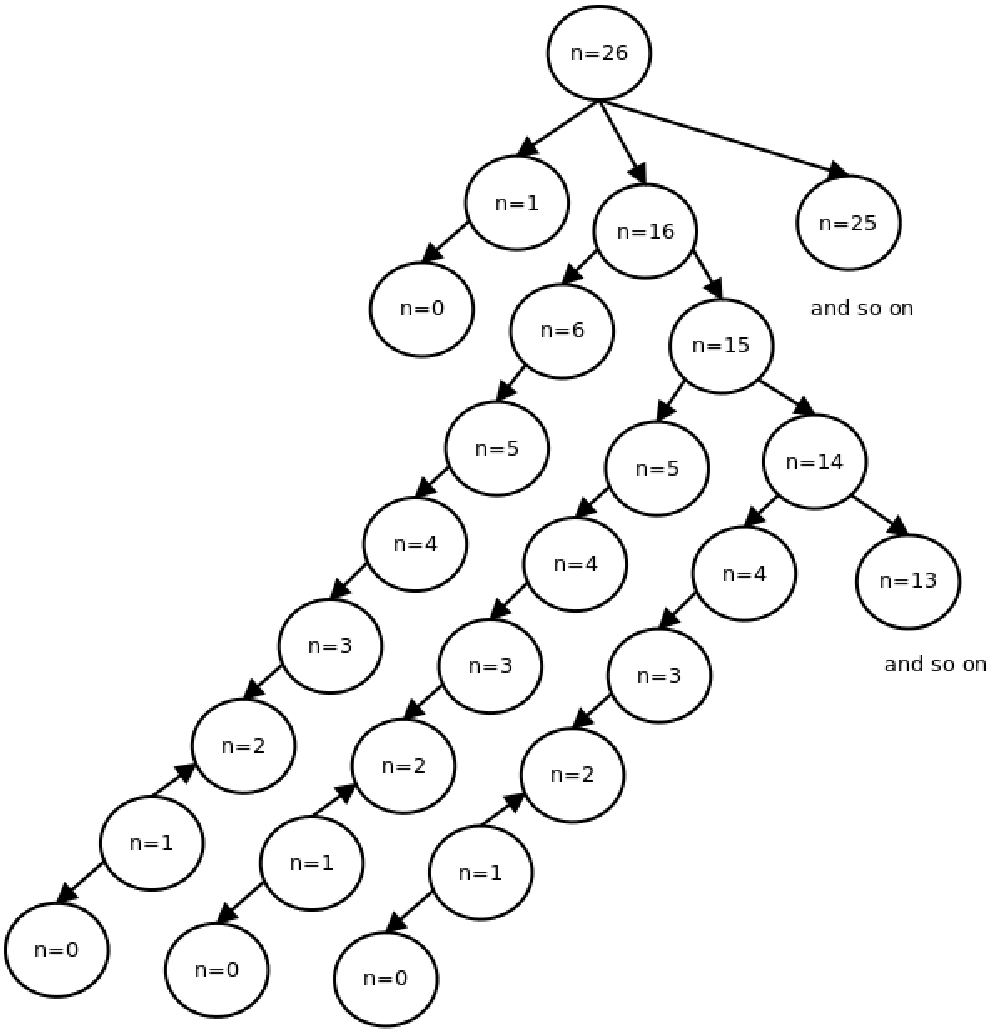
Step 3: Compute the value of an optimal solution in a bottom-up fashion.

Assume the coin array v{v_1,v_2, … v_k}

```
CoinChange(n)
  if n == 0
        return 0
  else
        minCoin = min(v); // minimum valued coin
        minValue = 1 + CoinChange(n – minCoin)
        for each coin v[i] in the list excluding the minCoin
              If n>v[i]
                    tempVal = 1 + CoinChange(n – v[i])
                    if tempVal < minVal
                          minVal = tempVal

  return minVal
```

LETS DRAW THE RECURSION TREE



Lots of overlapping substructure.

Can we use DP?

- Lots of overlapped substructure.
- Also optimal substructure .
- How to optimize?
  - Use DP
  - Memoization or tabular approach

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

```
Let c[0…n] be an array
c[0] = 0;
For i:1 to n
    c[i]=-1

CoinChange(n)
  if c[n] ≠ -1 return c[n]
  else
        minCoin = min(v); // minimum valued coin
        minValue = 1 + CoinChange(n – minCoin)
        for each coin v[i] in the list excluding the minCoin
              If n>v[i]
                    tempVal = 1 + CoinChange(n – v[i])
                    if tempVal < minVal
                    minVal = tempVal
        c[n] = minVal
  return minVal
```

Step 4: Construct an optimal solution from computed information.

```
Let c[0…n] and s[0…n] be two arrays
c[0] = 0;
For i:1 to n
    c[i]=-1

CoinChange(n)
  if c[n] ≠ -1 return c[n]
  else
        minCoin = min(v); // minimum valued coin
        minValue = 1 + CoinChange(n – minCoin)
        for each coin v[i] in the list excluding the minCoin
                If n>v[i]
                        tempVal = 1 + CoinChange(n – v[i])
                        if tempVal < minVal
                                minVal = tempVal
                                coinVal =v[i]
        c[n] = minVal ;    s[n] = coinVal
return minVal
```

Lets trace

$$c[n] = \begin{cases} 0 & if \;\; n = 0 \\ 1 + \min\limits_{v_1 \le v_i \le n}(c[n - v_i]\; if \;\; n > 0 \end{cases}$$

$$here\; v_i \rightarrow each\; coin\; in\; the\; array$$

Assume we have to make change for 12 cents with denomination {1,4,8,10}

Let s[0…n] be an array to store the last coin used. Now we need to fill up the table below at bottom-up approach.

i									
C[i]									
S[i]									

Trace cont…

- c[0]=0, s[0]=0 -> base case.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]													
S[i]													

Available coins: {1,4,8,10}

Trace cont…

- $c[0]=0, s[0]=0$ -> base case.
- $c[1]=\min\{1+c[1-1]\}=1+c[0]=1$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0												
S[i]	0												

Available coins: {1,4,8,10}

Trace cont…

- c[0]=0, s[0]=0 -> base case.
- c[1]=min{1+c[1-1]}=1+c[0]=1

Similarly

- c[2]=min{1+c[2-1]}=1+c[1]=2
- c[3]=min{1+c[3-1]}=1+c[2]=3

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1											
S[i]	0	1											

Available coins: {1,4,8,10}

Trace cont…

- c[0]=0, s[0]=0 -> base case.
- c[1]=min{1+c[1-1]}=1+c[0]=1

Similarly

- c[2]=min{1+c[2-1]}=1+c[1]=2
- c[3]=min{1+c[3-1]}=1+c[2]=3

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3									
S[i]	0	1	1	1									

Available coins: {1,4,8,10}

Trace cont…

- Now c[4]
- c[4] = min{1+c[4-**4**], 1+c[4-1]}=min{1,4}=1
- c[5] = min{1+c[5-**4**], 1+c[5-1]}=min{2,2}=2
- c[6] = min{1+c[6-**4**], 1+c[6-1]}=min{3,3}=3
- c[7] = min{1+c[7-**4**], 1+c[7-1]}=min{4,4}=4

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3									
S[i]	0	1	1	1									

Available coins: {1,4,8,10}

Trace cont…

- Now c[8]
- c[8]=min{1+c[8-**8**], 1+C[8-4], 1+c[8-1] } =min{1,2,4}=1
- C[9] = min{1+c[9-**8**], 1+C[9-4], 1+c[9-1] } =min{2,3,2}=2
- c[10] = min{1+c[10-10], 1+c[10-**8**], 1+C[10-4], 1+c[10-1] } =min{1,3,4,3}=1

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3	1	2	3	4					
S[i]	0	1	1	1	4	1	1	1					

Available coins: {1,4,8,10}

Trace cont…

- $c[11] = \min\{1+c[11-10], 1+c[11-\mathbf{8}], 1+C[11-4], 1+c[11-1]\} = \min\{2,4,5,2\} = 2$

- Now c[12]
- $c[12] = \min\{1+c[12-10], 1+c[12-\mathbf{8}], 1+C[12-4], 1+c[12-1]\} = \min\{3,2,2,3\} = 2$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3	1	2	3	4	1	2	1		
S[i]	0	1	1	1	4	1	1	1	8	1	10		

Available coins: {1,4,8,10}

Trace cont…

- c[11] = min{1+c[11-10], 1+c[11-**8**], 1+C[11-4], 1+c[11-1] } =min{2,4,5,2}=2

- Now c[12]
- c[12] = min{1+c[12-10], 1+c[12-**8**], 1+C[12-4], 1+c[12-1] } =min{3,2,2,3}=2

- **Result- minimum 2 coins needed**

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3	1	2	3	4	1	2	1	2	2
S[i]	0	1	1	1	4	1	1	1	8	1	10	1	4

Available coins: {1,4,8,10}

TRACE CONT…

- To build the solution
  1. **Pick the number of s[i] assume v_i**
  2. **Trace backward v_i times**
  3. **If you land to different s[i] other than s[0], repeat 1 and 2.**
  4. **Solution would be the s[i] values we landed on excluding the s[0].**

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3	1	2	3	4	1	2	1	2	2
S[i]	0	1	1	1	4	1	1	1	8	1	10	1	4

TRACE CONT…

- Example: want to find coins for 11 cent
  - s[11] -> 1
  - Trace backward 1 time. So, you will land to s[10]
  - S[10] has value 10. So, trace 10 backward and you will land to s[0].
  - So, solution(minimum coins needed) is the s[i] value we landed on
    - for 11 cents = {1,10}

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3	1	2	3	4	1	2	1	2	2
S[i]	0	1	1	1	4	1	1	1	8	1	10	1	4

Trace 10 position

Trace 1 position

TRACE CONT…

- Example: want to find coins for 6 cent
  - s[6] -> 1
  - Trace backward 1 time. So, you will land to s[5] which has value 1.
  - Trace backward 1 time. So, you will land to s[4] which has value 4.
  - So, trace 4backward and you will land to s[0].
  - So, solution for 6 cents = {1, 1, 4}

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3	1	2	3	4	1	2	1	2	2
S[i]	0	1	1	1	4	1	1	1	8	1	10	1	4

4 1 1

TRACE CONT…

- Example: want to find coins for 8 cent
  - s[8] -> 4
  - Trace backward 4 time. So, you will land to s[8]
  - S[8] has value 8. So, trace 8 backward and you will land to s[0].
  - So, solution for 12 cents = {4,8}

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C[i]	0	1	2	3	1	2	3	4	1	2	1	2	2
S[i]	0	1	1	1	4	1	1	1	8	1	10	1	4

Trace 8 position

Trace 4 position

Elements of Dynamic Programming (DP)

- Optimal substructure
  - an optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping subproblem
  - a recursive algorithm revisits the same problem repeatedly

Reference

- Chapter 15 (15.1 and 15.3) (Cormen)