Dynamic Programming

Job Scheduling

Knapsack

Fibonacci

Stair Climbing

CSE 3318 – Algorithms and Data Structures
University of Texas at Arlington

Alexandra Stefan
(Includes images, formulas and examples from CLRS, Dr. Bob Weems, wikipedia)

Steps for iterative (bottom up) solution

1. Identify trivial problems
    1. typically where the size is 0
2. Look at the last step/choice in an optimal solution:
    1. Assuming an optimal solution, what is the last action in completing it?
    2. Are there more than one options for that last action?
    3. If you consider each action, what is the smaller problem that you would combine with that last action?
        1. Assume that you have the optimal answer to that smaller problem.
    4. Generate all these solutions
    5. Compute the value (gain or cost) for each of these solutions.
    6. Keep the optimal one (max or min based on problem)
3. Make a 1D or 2D array and start feeling in answers from smallest to largest problems.

Other types of solutions:

1. Brute force solution
2. Recursive solution (most likely exponential and inefficient)
3. Memoized solution

The 0-1 Knapsack Problem

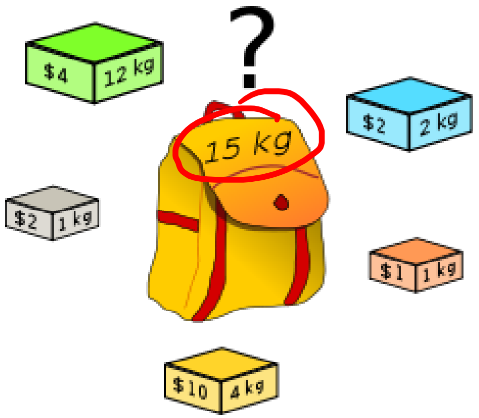


Image from Wikipedia:
https://en.wikipedia.org/wiki/Knapsack_problem

Problem:

- A thief breaks into a store.
- The maximum total weight that he can carry is W.
- There are N items at the store.
- Each item has a value v_i and a weight w_i.
- **There is only one of each item.**
- What is the maximum total **value** that he can take without exceeding capacity W?
- What items should he pick to obtain this maximum value?

Problem variations based number of items:

- Unlimited amounts – *Unbounded* Knapsack
- Limited amounts – *Bounded* Knapsack

What is a smaller problem?

What problem is trivial?

$sol \quad =$

$sol \quad =$

$sol \quad =$

$k = current\ weight$

Brute force approach

Max capacity: W=8		
item	Weight (Kg)	Value ($)
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

- See problem presented in table.

1. What are all possible combinations?
2. How many combinations are there in total?
3. What do I want to avoid?
4. Does the order in which I make my choices matter?

Developing the solution

Max capacity: W=8		
item	weight	Value
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

1. Find *stepping stones* toward a solution
   1. what do I make choices on? ____________
   2. does their order matter? ______________
2. Trivial, smallest problem(s): (think 0)
   1. _______________________
   2. _____________________
3. What is my last choice? ___________________
4. Write the answer for the original problem in terms of smaller problems

   ____
5. (Check that this is not a brute-force approach)
6. Formulate problem description

   ____

Max capacity: W=8

item	weight	Value
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

Worksheet: 0-1 Knapsack Example 1

		0	1	2	3	4	5	6	7	8
0 No item , kg, $	0									
A, 4, *5*	1									
B, 3, *4*	2									
C, 2, *3*	3									
D, 1, *2*	4									

Let *Knap(n,{….},W)*-Knapsack pb for n items, and max capacity W
E.g.
Knap(n=4,{A,B,C,D},W=8)

Smaller problems:

____

____

____

____

____

____

Meaning of
cell at [0][5] = ______________________________

cell at [3][0] = ______________________________

cell at [4][8] = ______________________________

cell at [3][8] = ______________________________

cell at [3][7] = ______________________________

Knap(n=4,{A,B,C,D},W=8)
What choices do we have? (especially related to consecutive problems in size)

For one item :

E.g. for item D:

$\Rightarrow$ **sol is**

E.g. for choice at item D **sol is**

Worksheet: 0-1 Knapsack Example 1

Max capacity: W=8

item	weight	Value
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

		0	1	2	3	4	5	6	7	8
0 No item , kg, $	0									
A, 4, *5*	1									
B, 3, *4*	2									
C, 2, *3*	3									
D, 1, *2*	4									

Let *Knap(n,{....},W)*-Knapsack pb for n items, and max capacity W
E.g.
Knap(n=4,{A,B,C,D},W=8)

Smaller problems:

____

____

____

____

____

____

Meaning of
cell at [0][5] = ______________________________

cell at [3][0] = ______________________________

cell at [4][8] = ______________________________

cell at [3][8] = ______________________________

cell at [3][7] = ______________________________

Knap(n=4,{A,B,C,D},W=8)
What choices do we have? (especially related to consecutive problems in size)

For one item :

E.g. for item D:

$\Rightarrow$ **sol is**

E.g. for choice at item D **sol is**

Worksheet: **0-1 Knapsack** Example 1

Max capacity: W=8		
item	weight	Value
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

		0	1	2	3	4	5	6	7	8
0 No item , kg, $	0									
A, 4, *5*	1									
B, 3, *4*	2									
C, 2, *3*	3									
D, 1, *2*	4									

Let *Knap(n,{….},W)*-Knapsack pb for n items, and max capacity W
E.g.
Knap(n=4,{A,B,C,D},W=8)

Smaller problems:
_Knap (n=4,{A,B,C,D}, W=7)___
_Knap (n=4,{A,B,C,D}, W=6)___
_Knap (n=4,{A,B,C,D}, W=0)___
_Knap (n=3,{A,B,C}, W=8)___
_Knap (n=2,{A,B}, W=8)___
_Knap (n=0,{ }, W=8)___

Note that we use the same order for the items: A,B,C,D.
=> for n=3 only {A,B,C} (no {A,B,D})

Meaning of
cell at [0][5] = Sol for Knap(n=**0**,{}, W=**5**)

cell at [3][0] = Sol for Knap(n=**3**,{A,B,C}, W=**0**)

cell at [4][8] = Sol for Knap(n=**4**,{A,B,C,D},W=**8**)

cell at [3][8] = Sol for Knap(n=**3**,{A,B,C}, W=**8**)

cell at [3][7] = Sol for Knap(n=**3**,{A,B,C}, W=**7**)

Knap(n=4,{A,B,C,D},W=8)
What choices do we have? (especially related to consecutive problems in size)

For one item : - do not take it
- take it

E.g. for item D: - do not take D
- take D

$\Rightarrow$ **sol is max of solutions for each choice**

E.g. for choice at item D **sol is max of**:
- Knap(n=3,{A,B,C},W=8) (do not take D)
- value(D)+Knap(n=3,{A,B,C},W=8-weight(D)) (take D if it fits)

Worksheet: **0-1 Knapsack** Example 1

Max capacity: W=8		
item	weight	Value
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

Take out paper and draw the table below: 5 rows, 9 columns

- rows = number of items+1
- columns = W+1

									7	8
No item , kg, $	0									
A, 4, *5*	1									
B, 3, *4*	2									
C, 2, *3*	3									
D, 1, *2*	4									

Max capacity: W=8		
item	weight	Value
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

Worksheet: **0-1 Knapsack** Example 1

Fill in the table. After that, write the solution formula.

		0	1	2	3	4	5	6	7	8
No item , kg, $	0									
A, 4, *5*	1									
B, 3, *4*	2									
C, 2, *3*	3									
D, 1, *2*	4									

Worksheet: **0-1 Knapsack** Example 1 - Answers

sol[i][k] – optimal solution for 0-1 Knapsack of max capacity k with only the first i items, 1,2,…,i,.

At row (i-1) we have optimal solutions WITHOUT item i.

$$sol[0][k] = 0, \forall k$$
$$sol[i][0] = 0, \forall i$$
$$sol[i][k] = \begin{cases} sol[i-1][k] & if\ k < w[i] \\ \max\{ sol[i-1][k],\ v[i] + sol[i-1][k-w[i]] \} & if\ k \geq w[i] \end{cases}$$
$$k = current\ weight$$

Value using **first i items:**
sol[i] [k] = **max**{sol[i-1] [k] , sol[i-1] [k – w[i]] + v[i]}

Max capacity: W=8		
item	Weight (Kg)	Value ($)
A	4	5
B	3	4
C	2	3
D	1	2

		0	1	2	3	4	5	6	7	8
0	No item , kg, $	0 .	0 .	0 .	0 .	0 .	0 .	0 .	0 .	0 .
1	A, 4, 5	0 .	0 .	0 .	0 .	5*	5*	5*	5*	5*
2	B, 3, 4	0 .	0 .	0 .	4*	5 .	5 .	5 .	9*	9*
3	C, 2, 3	0 .	0 .	3*	4 .	5 .	7*	8*	9 .	9 .
4	D, 1, 2	0 .	2*	3 .	5*	6*	7 .	9*	10*	11*

Where is the final answer to the original problem? ________

What items give that money? ___________________

Worksheet: 0-1 Knapsack Example 1 - Backtrace

sol[i][k] – optimal solution for 0-1 Knapsack of max capacity k with only the first i items, 1,2,…,i,.

At row (i-1) we have optimal solutions WITHOUT item i.

$$sol[0][k] = 0, \forall k$$
$$sol[i][0] = 0, \forall i$$
$$sol[i][k] = \begin{cases} sol[i-1][k] & if\ k < w[i] \\ \max\{ sol[i-1][k],\ v[i] + sol[i-1][k - w[i]] \} & if\ k \geq w[i] \end{cases}$$
$$k = current\ weight$$

Value using **first i items:**
sol**[i]** [k] = **max**{sol**[i-1]** [k] , sol**[i-1]** **[** k – w[i]**]** + v[i]}

Max capacity: W=8		
item	Weight (Kg)	Value ($)
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

		0	1	2	3	4	5	6	7	8
0	No item, kg, $	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*
1	A, 4, *5*	*0 .*	*0 .*	*0 .*	*0 .*	*5**	*5**	*5**	*5**	*5**
2	B, 3, *4*	*0 .*	*0 .*	*0 .*	*4**	*5 .*	*5 .*	*5 .*	*9**	*9**
3	C, 2, *3*	*0 .*	*0 .*	*3**	*4 .*	*5 .*	*7**	*8**	*9 .*	*9 .*
4	D, 1, *2*	*0 .*	*2**	*3 .*	*5**	*6**	*7 .*	*9**	*10**	*11**

Where is the final answer to the original problem? **11**

What items give that money? Backtrace from cell[4][8] gives: **A,B,D**

Backtrace:

choice[4][8] -> * -> **D**
row--, column = 8-weight(D)
choice[3][7] -> . ->
row--,
choice[2][7] -> * -> **B**
row--, column = 7-weight(B)
choice[1][4] -> * -> **A**
row--, column = 4-weight(A)
choice[0][0] stop
(either row or column is 0)

Redo this problem with order: ~~A,B,D,C~~

B, A, D, C

Worksheet: **0-1 Knapsack** Example 2

sol $=$
sol $=$
sol $=$
$k = current\ weight$

solve & check your answers on next page

Max capacity: W=16		
item	weight	Value
A	3	*4*
B	4	*6*
C	7	*11*
D	8	*13*
E	9	*15*

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	No item kg, *$*																		
1	A, 3, *4*																		
2	B, 4, *6*																		
3	C, 7,*11*																		
4	D, 8,*13*																		
5	E, 9,*15*																		

Final answer: ______ Items that give this value: ______________________

Answer: 0-1 Knapsack –Example 2

sol[i][k] – optimal solution for 0-1 Knapsack of max capacity k with only the first i items, 1,2,…,i,.

At row (i-1) we have optimal solutions WITHOUT item i.

$$sol[0][\mathrm{k}] = 0, \forall k$$
$$sol[\mathrm{i}][0] = 0, \forall i$$
$$sol[i][k] = \begin{cases} sol[i-1][k] & if\ k < w[i] \\ \max\{\ sol[i-1][k],\ v[i] + sol[i-1][k-w[i]]\ \} & if\ k \geq w[i] \end{cases}$$
$$k = current\ weight$$

Value using **first i items:**
sol**[i]** [k] = **max**{sol**[i-1]** [k] , sol**[i-1]** **[** k – w[i]**]** + v[i]}

E.g.: Solution using **first 3 items(A,B,C)** for max capacity 15: sol**[3]** [15] = **max**{sol**[2]** [8], sol**[2]** **[15 - 7]** +11} = max{ 10, 10+11 } = 21

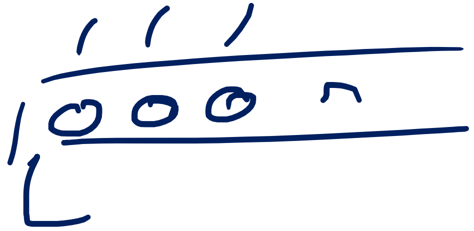
Max capacity: W=16		
item	weight	Value
A	3	*4*
B	4	*6*
C	7	*11*
D	8	*13*
E	9	*15*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0 No item kg, $	*0.*	*0.*	*0.*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*	*0 .*
1 A, 3, *4*	*0.*	*0.*	*0.*	*4**	*4**	*4**	*4**	*4**	*4**	*4**	*4**	*4**	*4**	*4**	*4**	*4**	*4**
2 B, 4, *6*	*0.*	*0.*	*0.*	*4 .*	*6**	*6**	*6**	*10**	*10**	*10**	*10**	*10**	*10**	*10**	*10**	*10**	*10**
3 C, 7,*11*	*0.*	*0.*	*0.*	*4 .*	*6 .*	*6 .*	*6 .*	*11**	*11**	*11**	*15**	*17**	*17**	*17**	*21**	*21**	*21**
4 D, 8,*13*	*0.*	*0.*	*0.*	*4 .*	*6 .*	*6 .*	*6 .*	*11 .*	*13**	*13**	*15 .*	*17**	*19**	*19**	*21 .*	*24**	*24**
5 E, 9,*15*	*0.*	*0.*	*0.*	*4 .*	*6 .*	*6 .*	*6 .*	*11 .*	*13 .*	*15**	*15**	*17 .*	*19**	*21**	*21**	*24 .*	*26**

10+11 10

Final answer: ______ Items that give this value: ____________________

Iterative Solution for 0-1 Knapsack



```
/* Assume arrays v and w store the item info starting at index 1:
first item has value v[1] and weight w[1]    */

int knapsack01(int W, int n, int * v, int * w){
    int sol[n+1][W+1];
    for(k=0; k<=W; k++) { sol[0][k] = 0;}
    for(i=1; i<=n; i++) {
        for(k=0;k<=W;k++) {
            sol[i][k] = sol[i-1][k]; // solution without item i
            if (k>w[i]) {
                with_i = v[i]+sol[i-1][k-w[i]];
                if (sol[i][k] < with_i) {    // better choice
                    sol[i][k] = with_i;       // keep it
                }
            }
        }// for k
    }// for i
    return sol[n][W];
}
```

// Time: Θ(nW) Space: Θ(nW) pseudo polynomial in W

// need Θ(n) bits to store n items (values and weights) , but only log_2(W) bits to store W

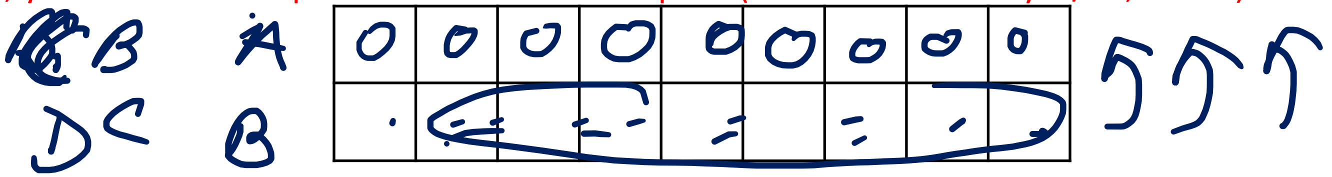
8 4 // n // w 7 7 7 < 7 4 8

linear

$(\log_2 w)$ bits to store w

Improving memory usage: Θ(W)

- Optimize the memory usage: store only smaller problems that are needed.
  - Store either 2 rows or 2 columns
  - the choices cannot be recovered anymore (i.e. cannot recover what items to pick to achieve the computed optimal value).
    - if you need to recover the items, you cannot save space. Youmust use nW space (for the table with yes/no, * no *)



Θ(W)

- Space complexity: Θ(W)

- Practice:
  - Can you implement this solution?

Hint for DP problems

- For a DP problem you can typically write a MATH function that gives the solution for problem of size N in terms of smaller problems.
- It is straightforward to go from this math function to code:
  - Iterative: The math function 'maps' to the sol array
  - Recursive: The math function 'maps' to recursive calls
- Typically the math function will be a
  - Min/max (over itself applied to smaller N)
  - Sum (over itself applied to smaller N)

max

max/min

Weighted Interval Scheduling

(Job Scheduling)

Weighted Interval Scheduling
(a.k.a. Job Scheduling)

Problem:

Given n jobs where each job has a start time, finish time and value, (s_j, f_j, v_j) select a subset of them that do not overlap and give the largest total value.

E.g.:
(start, end, value)
(6, 8, \$2)
(2, 5, \$6)
(3, 11, \$5)
(5, 6, \$3)
(1, 4, \$5)
(4, 7, \$2)

Weighted Interval Scheduling

(a.k.a. Job Scheduling)

E.g.:

(start, end, value)
(6, 8, $2)
(2, 5, $6)
(3, 11, $5)
(5, 6, $3)
(1, 4, $5)
(4, 7, $2)

Add job 0
Sort by
finish time

JobId (start, **end**, value)
0 (0, 0, $0)
1 (1, 4, $5)
2 (2, 5, $6)
3 (5, 6, $3)
4 (4, 7, $2)
5 (6, 8, $2)
6 (3, 11, $5)

p(j) - last job prior to *j*, that does not overlap with *j*

JobId (start, end, value, **p(j)**)
0 (
1 (
2 (
3 (
4 (
5 (
6 (

Preprocessing:

- Add job 0 : (0,0,0)
- Sort jobs by finish time (increasing).
- For each job *j*, compute *p(j)*, the last job prior to *j*, that does not overlap with *j*.
  - p(4) is _ (last job that does not overlap with job 4)
  - p(5) is _

Weighted Interval Scheduling

(a.k.a. Job Scheduling)

E.g.:

(start, end, value)
(6, 8, \$2)
(2, 5, \$6)
(3, 11, \$5)
(5, 6, \$3)
(1, 4, \$5)
(4, 7, \$2)

Add job 0
Sort by
finish time

$N \lg N$

JobId	(start,	end,	value)
0	(0,	**0**,	\$0)
1	(1,	**4**,	\$5)
2	(2,	**5**,	\$6)
3	(5,	**6**,	\$3)
4	(4,	**7**,	\$2)
5	(6,	**8**,	\$2)
6	(3,	**11**,	\$5)

p(j) - last job prior to *j*, that does not overlap with *j*

N^2 linear search

$N \lg N$ binary search

JobId	(start,	end,	value,	p(j))
0	(0,	0,	\$0,	-1)
1	(1,	4,	\$5,	0)
2	(2,	5,	\$6,	0)
3	(5,	6,	\$3,	2)
4	(4,	7,	\$2,	1)
5	(6,	8,	\$2,	3)
6	(3,	11,	\$5,	0)

Preprocessing:

- Sort jobs by finish time (increasing).
- For each job *j*, compute *p(j)*, the last job prior to *j*, that does not overlap with *j*.
  - p(4) is 1 (last job that does not overlap with job 4)
  - p(5) is 3

Weighted Interval Scheduling

(a.k.a. Job Scheduling)

E.g.:

(start, end, value)
(6, 8, \$2)
(2, 5, \$6)
(3, 11, \$5)
(5, 6, \$3)
(1, 4, \$5)
(4, 7, \$2)

Add job 0
Sort by
finish time

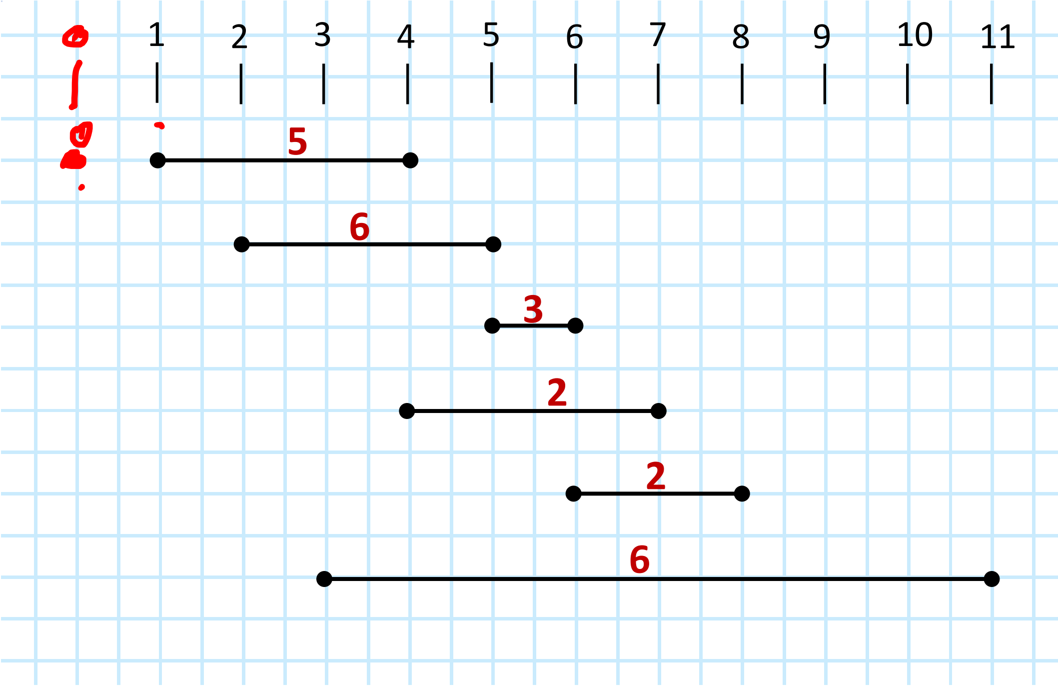
JobId	(start,	end,	value)
0	(0,	**0**,	\$0)
1	(1,	**4**,	\$5)
2	(2,	**5**,	\$6)
3	(5,	**6**,	\$3)
4	(4,	**7**,	\$2)
5	(6,	**8**,	\$2)
6	(3,	**11**,	\$5)

p(j) - last job prior to *j*, that does not overlap with *j*

JobId	(start,	end,	value,	p(j))
0	(0,	0,	\$0,	**-1**)
1	(1,	4,	\$5,	**0**)
2	(2,	5,	\$6,	**0**)
3	(5,	6,	\$3,	**2**)
4	(4,	7,	\$2,	**1**)
5	(6,	8,	\$2,	**3**)
6	(3, 11,		\$5,	**0**)

Preprocessing:

- Sort jobs by finish time (increasing).
- For each job *j*, compute *p(j)*, the last job prior to *j*, that does not overlap with *j*.
  - p(4) is 1 (last job that does not overlap with job 4)
  - p(5) is 3



Weighted Interval Scheduling
(a.k.a. Job Scheduling)

Problem:

- Given n jobs where each job has a start time, finish time and value, (s_j,f_j,v_j) select a subset of them that do not overlap and give the largest total value.

Preprocessing:

- Sort jobs in increasing order of their finish time. –already done here
- For each job ,*j*, compute the last job prior to *j*, *p(j)*, that does not overlap with *j*.
- TC: O(nlgn) (nlgn sorting and binary search for finding p(j))

Solve the problem:

Steps: one step for each job.

Choice: pick job or not

Smaller problems: 2:

pb1 = jobs 1 to j-1, => sol(j-1)

pb2 = jobs 1 to p(j) (where p(j) is the last job before j that does not overlap with j. => sol(p(j))

Solution function (gives the money value: sol(j) = the most money we can make using jobs 1,2,..,j):

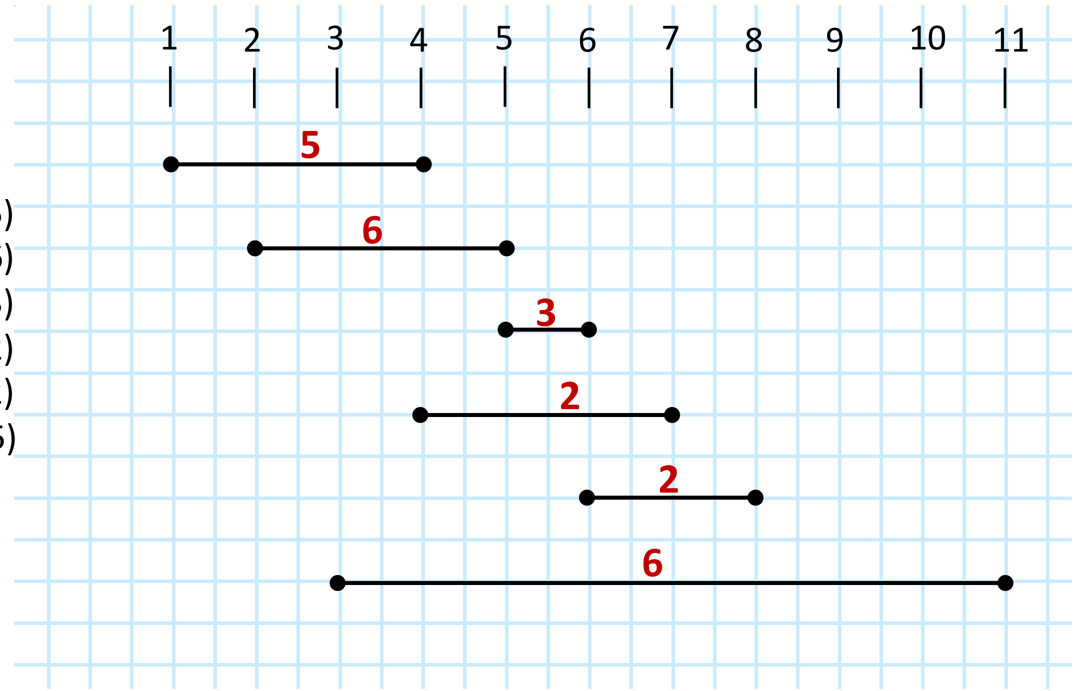
$$sol(0) = 0$$

$$sol(j) = \max\{sol(j-1), v(j) + sol(p(j))\}$$

Time complexity: O(n) *(if data is already preprocessed)* Fill out *sol(j) in* constant time for each j)

O(nlgn) *(with preprocessing)*

(s, f, $)
(1, 4, $5)
(2, 5, $6)
(5, 6, $3)
(4, 7, $2)
(6, 8, $2)
(3, 11, $5)



Solve the problem:

Steps: one step for each job.

Option: pick it or not (pick job j or not pick it)

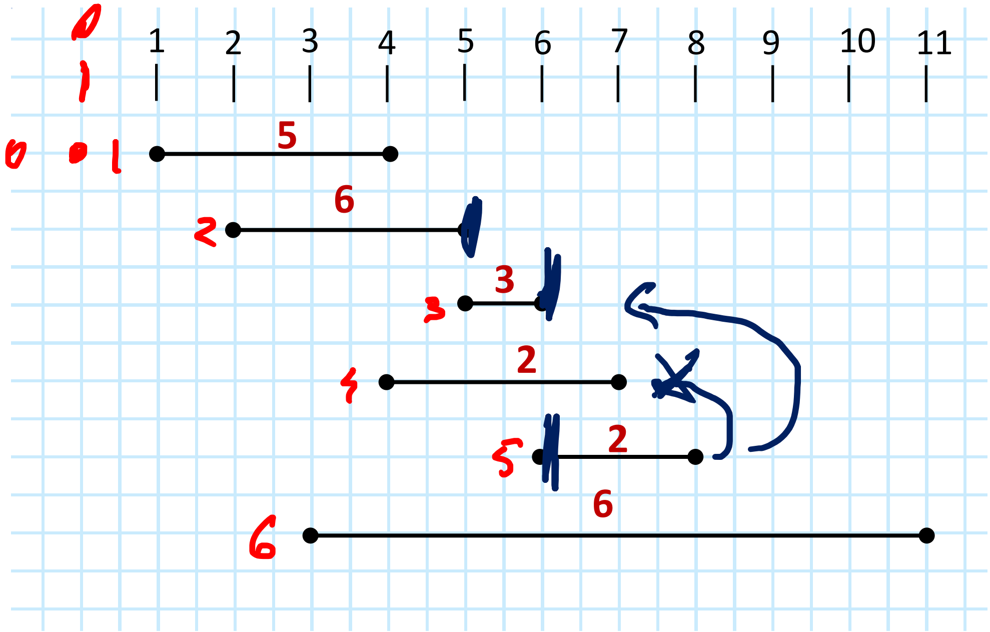
Smaller problems: 2:

pb1 = jobs 1 to j-1, => sol(j-1)

pb2 = jobs 1 to p(j) (where p(j) is the last job before j that does not overlap with j. => sol(p(j))

Solution function:

$$sol(0) = 0$$
$$sol(j) = \max\{sol(j-1), v(j) + sol(p(j))\}$$



Time complexity: ________ (without preprocessing)

________ (with preprocessing)

Solution function:

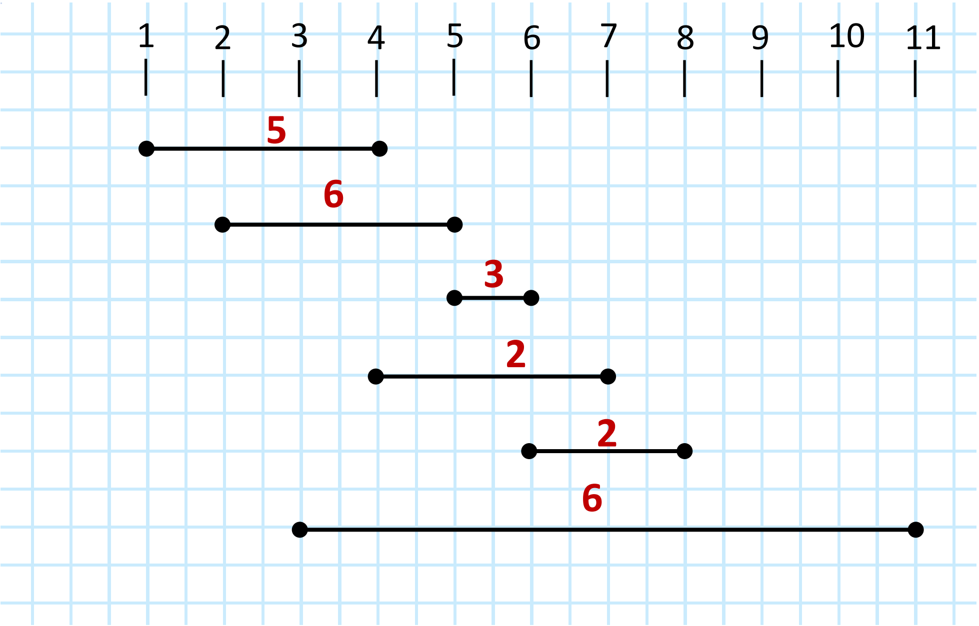
$sol(0) = 0$

$sol(j) = \max\{sol(j-1), v(j) + sol(p(j))\}$

No Yes No Yes

j	v_j	p_j	j	Y/N	sol(j)	Show work
0	0	-1	0	-1	0	
1	5	0	1	Y	5	= max{sol(1-1), v(1)+sol(0)} = max{sol(0), 5+0} = max{0, 5} = 5
2	6	0	2	Y	6	= max{sol(2-1), v(2)+sol(0)} = max{sol(1), 6+0} = max{5, 6} = 6
3	3	2	3	Y	9	= max{sol(2), v(3)+sol(2)} = max{6, 3+6} = max{6, 9} = 9
4	2	1	4	N	9	= max{sol(3), v(4)+sol(1)} = max{9, 2+5} = max{9, 7} = 9
5	2	3	5	Y	11	= max{sol(4), v(5)+sol(3)} = max{9, 2+9} = max{9, 11} = 11
6	5	0	6	N	11	= max{sol(5), v(6)+sol(0)} = max{11, 5+0} = max{11, 5} = 11

Optimal value: 11, jobs picked to get this value: 5, 3, 2



Time complexity: ________ (without preprocessing)

________ (with preprocessing)

Solution function:

$sol(0) = 0$

$sol(j) = \max\{sol(j-1), v(j) + sol(p(j))\}$

j	v_j	p_j
0	*0*	**-1**
1	*5*	**0**
2	*6*	**0**
3	*3*	**2**
4	*2*	**1**
5	*2*	**3**
6	*5*	**0**

j	Y/N	sol(j)	Show work
0			
1			
2			
3			
4			
5			
6			

Optimal value: ___, jobs picked to get this value: ______

Solve the problem:

Steps: one step for each job.

Option: pick it or not (pick job j or not pick it)

Smaller problems: 2:

pb1 = jobs 1 to j-1, => sol(j-1)

pb2 = jobs 1 to p(j) (where p(j) is the last job before j that does not overlap with j. => sol(p(j))

Solution function:

$$sol(0) = 0$$
$$sol(j) = \max\{sol(j-1), v(j) + sol(p(j))\}$$

After preprocessing
(sorted by END time):
JobId (start, end, value, p(j))
1 (1, 4, $5, _0_)
2 (2, 5, $6, _0_)
3 (5, 6, $3, _2_)
4 (4, 7, $2, _1_)
5 (6, 8, $2, _3_)
6 (3, 11, $5, _0_)

Time complexity: O(n) (if data is preprocessed)

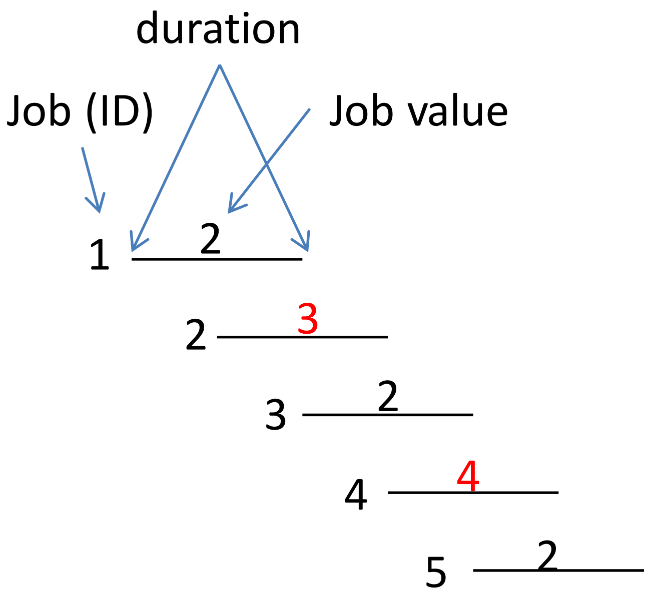
O(nlgn) (if jobs need to be sorted first and nlgn sorting algorithm and nlgn for p(j) binary search for finding p(i))

j	v_j	p_j	j	sol(j)	j	sol(j) used j	In optimal solution
0	*0*	**-1**	0	0	0	N	
1	*5*	**0**	1	5 = max{0, 5+0}	1	Y	
2	*6*	**0**	2	6 = max{5, 6+0}	2	Y	Y
3	*3*	**2**	3	9 = max{6, 3+6}	3	Y	Y
4	*2*	**1**	4	9 = max{9, 2+5}	4	N	
5	*2*	**3**	5	11 = max{9, 2+9}	5	Y	Y
6	*5*	**0**	6	11 = max{11, 5+0}	6	N	

Optimal value: **11**, jobs picked to get this value: **2,3,5**

Another example

- Notations conventions:
  - Jobs are already sorted by end time (no preprocessing needed)
  - Horizontal alignment is based on time. *In this example, only consecutive jobs overlap*, (e.g. jobs 1 and 3 do not overlap).

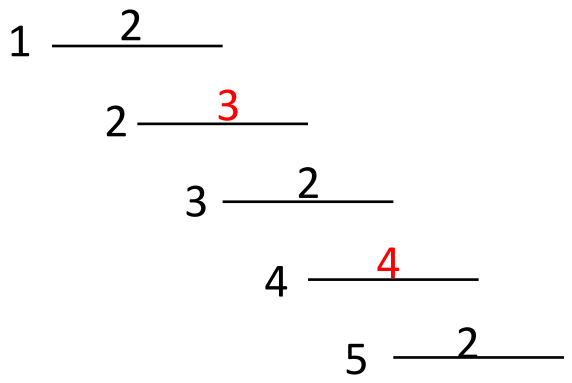


E.g.:
(Job, start, end, value)
(1, 3pm, 5pm, 2$)
(2, 4pm, 6pm, 3$)
(3, 5pm, 7pm, 2$)
(4, 6pm, 8pm, 4$)
(5, 7pm, 9pm, 2$)

Time complexity (excluding the preprocessing part): O( )

Recovering the Solution

- Example showing that when computing the optimal gain, we *cannot decide which jobs will be part of the solution and which will not*. We can only recover the jobs picked <u>AFTER</u> we computed the optimum gain and by going from <u>end to start</u>.



j	v_j	p_j	sol(j)	sol(j) used j	In optimal solution
0	0	0	0	N	
1	2	0	2	Y	
2	3	0	3	Y	Y
3	2	1	4	Y	
4	4	2	7	Y	Y
5	2	3	7	N	

Time complexity (excluding the preprocessing part): O( )

Bottom-up (BEST)

Math function:
$sol(0) = 0$
$sol(j) = \max\{sol(j-1), v(j) + sol(p(j))\}$

The program will create an populate an array, `sol`, corresponding to the *sol* function from the math definition.

```
// Bottom-up (the most efficient solution)
int js_iter(int* v, int*p, int n){
   int j, with_j, without_j;
   int sol[n+1];
   // optionally, may initialize it to -1 for safety
   sol[0] = 0;
   for(j = 1; j <= n; j++){        // O(N) rep
      with_j = v[j] + sol[p[j]];    // Θ(1)
      without_j = sol[j-1];         // O(1)
      if ( with_j >= without_j)     O(1)
         sol[j] = with_j;
      else
         sol[j] = without_j;
   }
   return sol[n];
}
```

O(N)

O(1)

v[1] = value of job,
p[1] = index of last job that does not overlap with 1

Here N=6
(6 jobs)

The `sol` array must have size n+1 b.c. we must access indexes from 0 to n.

j	v_j	p_j
0	*0*	**-1**
1	*5*	**0**
2	*6*	**0**
3	*3*	**2**
4	*2*	**1**
5	*2*	**3**
6	*5*	**0**

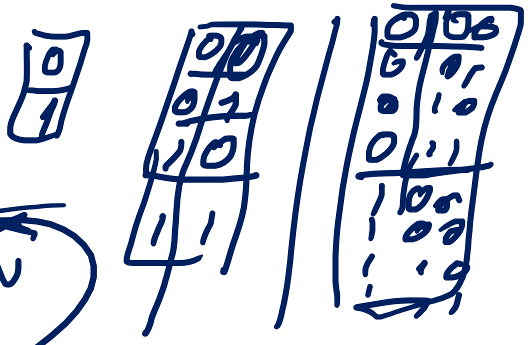
	sol[j]
0	0
1	5 = max{0, 5+0}
2	6 = max{5, 6+0}
3	9 = max{6, 3+6}
4	9 = max{9, 2+5}
5	11 = max{9, 2+9}
6	11 = max{11, 5+0}

with preprocessing
TC: O(N lg N)
or O(N²)

Time complexity: Θ(N), Space complexity: Θ(N)

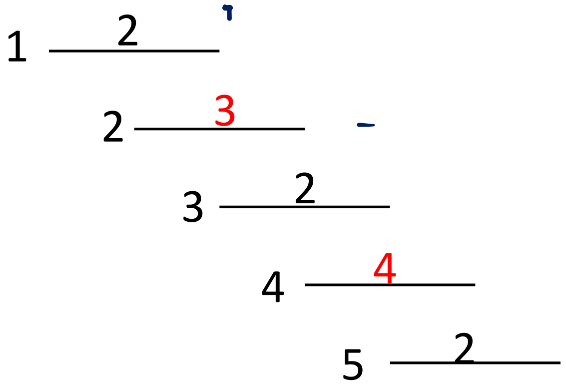
Job Scheduling – Brute Force Solution

$2^5 \to 2^N$



2^N

- For each job we have the option to include it (1) or not(0). Gives:
  - The power set for a set of 5 elements, or
  - All possible permutations with repetitions over n positions with values 0 or 1=> O(__)
  - Note: exclude sets with overlapping jobs.
- Time complexity: O(___)

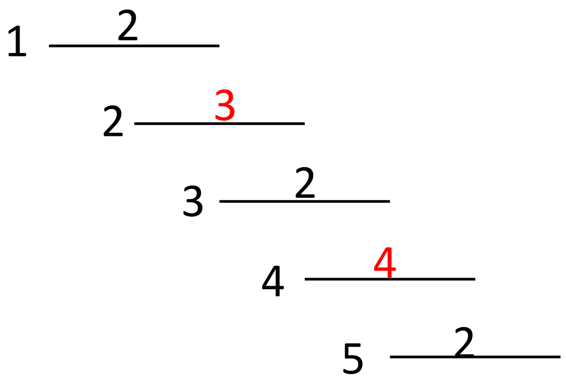


NO

1	2	3	4	5	Valid	Total value
0	0	0	0	0	yes	0
0	0	0	0	1 Yes	yes	2
0	0	0	1	0	yes	4
0	0	0	1	1	no	
0	0	1	0	0	yes	2
0	0	1	0	1	yes	4 (=2+2)
0	0	1	1	1	no	
…	…	…	…	…	…	…
1	1	1	1	1	no	

Job Scheduling –
Brute Force Solution

- For each job we have the option to include it (1) or not(0). Gives:
  - The power set for a set of 5 elements, or
  - All possible permutations with repetitions over n positions with values 0 or 1=> O(**2^n**)
  - Note: exclude sets with overlapping jobs.
- Time complexity: O(**2^n**)



1	2	3	4	5	Valid	Total value
0	0	0	0	0	yes	0
0	0	0	0	1	yes	2
0	0	0	1	0	yes	4
0	0	0	1	1	no	
0	0	1	0	0	yes	2
0	0	1	0	1	yes	4 (=2+2)
0	0	1	1	1	no	
…	…	…	…	…	…	…
1	1	1	1	1	no	

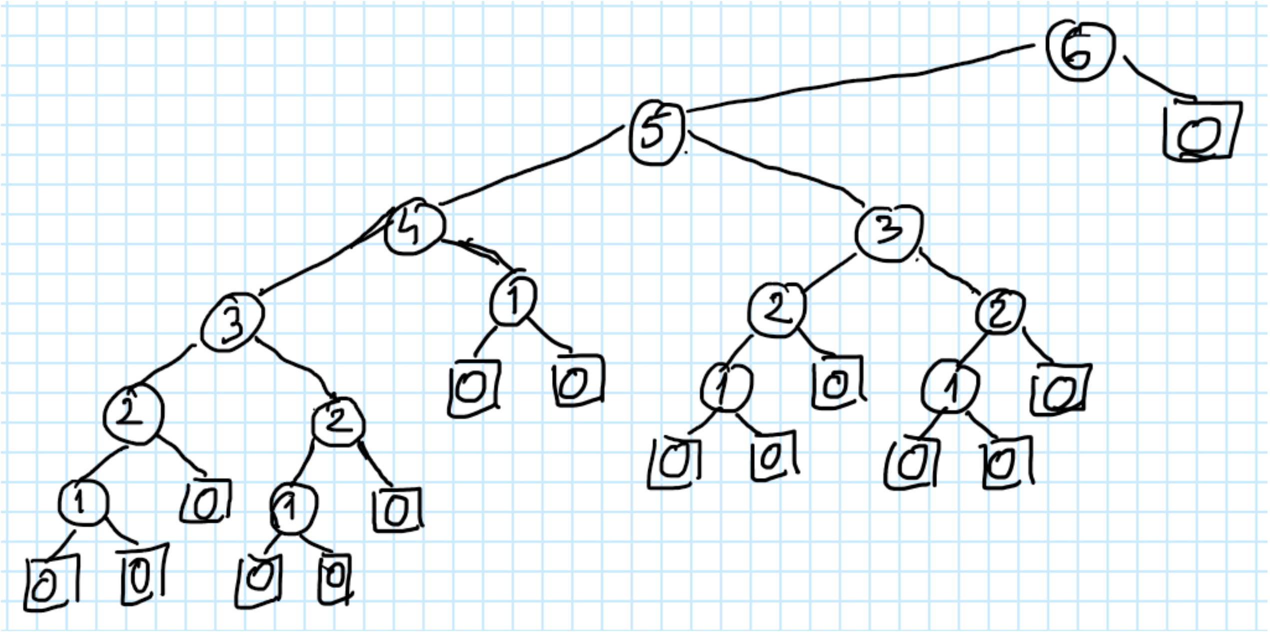
Recursive (inefficient)

- Write the solution for problem size n
- Make a recursive call for the smaller problem size (instead of array look-up).
- Recomputes multiple times the answer for the same problem (e.g. pb size 2 is computed 4 times) . That makes it inefficient.

Math function:
$sol(0) = 0$
$sol(j) = \max\{sol(j-1), v(j) + sol(p(j))\}$

```
// Inefficient recursive solution:
int jsr(int* v, int*p, int n){
    if (n == 0) return 0;
    int res;
    int with_n = v[n] + jsr(v,p,p[n]);
    int without_n = jsr(v,p,n-1);
    if ( with_n >= without_n)
        res = with_n;
    else
        res = without_n;
    return res;
}
```

j	v_j	p_j
0	*0*	**-1**
1	*5*	**0**
2	*6*	**0**
3	*3*	**2**
4	*2*	**1**
5	*2*	**3**
6	*5*	**0**



Memoization (Recursion combined with saving)

Math function:

$sol(0) = 0$

$sol(j) = \max\{sol(j-1), v(j) + sol(p(j))\}$

```
// Memoization efficient recursive solution:
int jsm(int* v, int*p, int n, int* sol){
    if (sol[n] != -1) // already computed.
        return sol[n]; // Used when rec call for a smaller problem.
    int res;
    int with_n = v[n] + jsm(v,p,p[n],sol);
    int without_n = jsm(v,p,n-1,sol);
    if ( with_n >= without_n)      res = with_n;
    else                       res = without_n;
    sol[n] = res;
    return res;
}
int jsr_out(int* v, int*p, int n){
    int sol[n+1];
    int j;
    sol [0] = 0;
    for (j = 1; j<= n; j++)  sol [j] = -1; //not computed
    jsm(v,p,n,sol);
    return sol[n];
}
```

$\Theta(N)$

sol(2)

rec(2)

rec(3)

sol

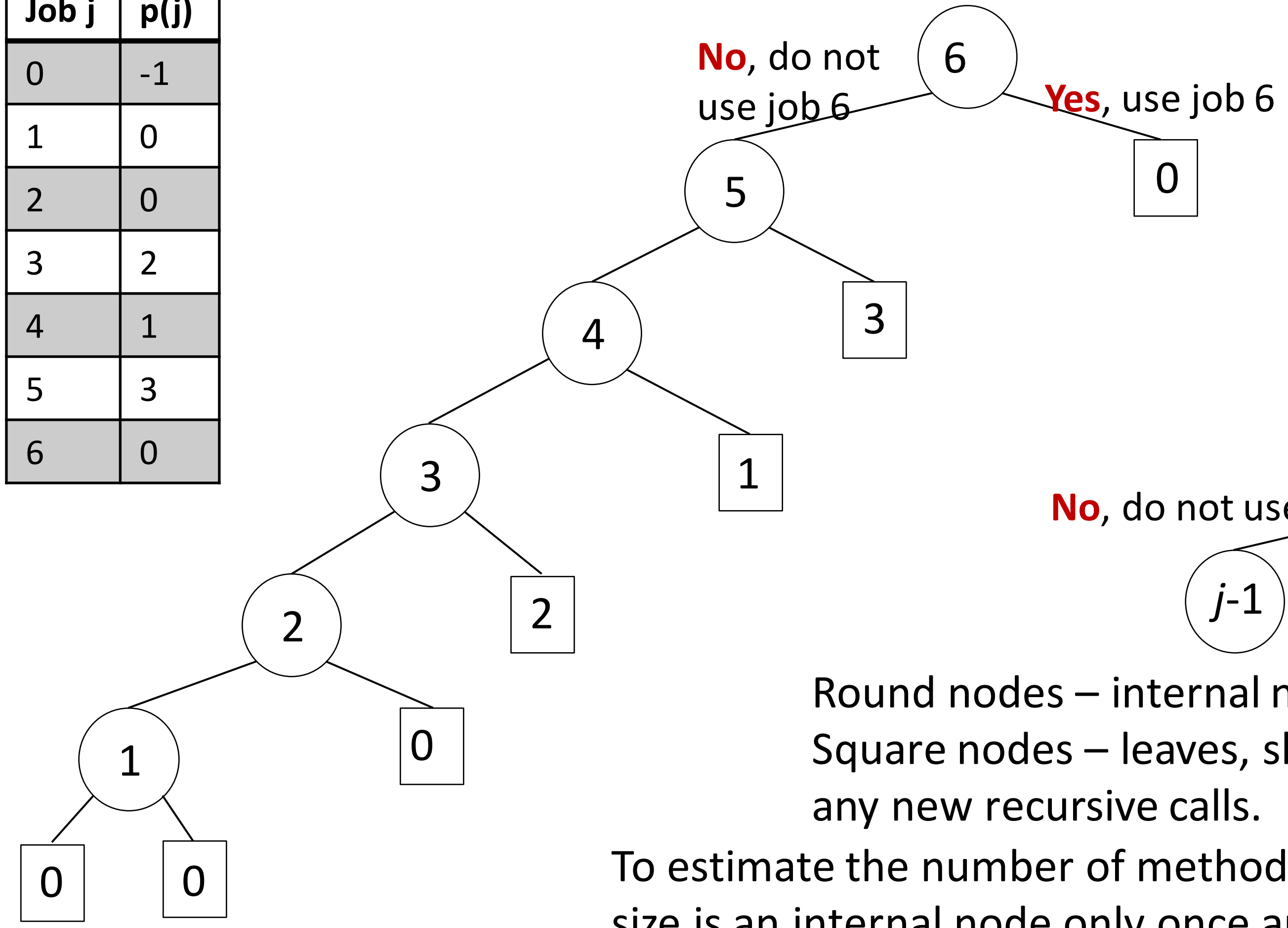
jsm(v,p,6,sol)

jsm(0)

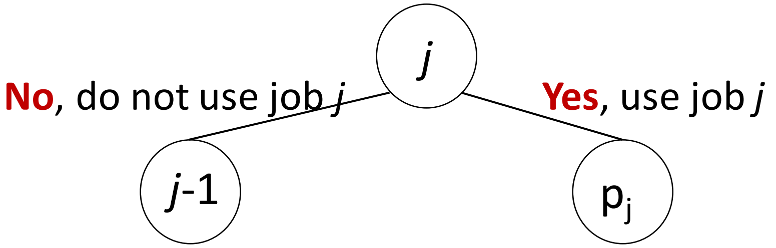
jsm(v,p,5,sol)

Function call tree for the memoized version

Job j	p(j)
0	-1
1	0
2	0
3	2
4	1
5	3
6	0



	0	1	2	3	4	5	6
sol	0	-1	-1	-1	-1	-1	-1



Round nodes – internal nodes. Require recursive calls.
Square nodes – leaves, show calls that return without any new recursive calls.

To estimate the number of method calls note that every problem size is an internal node only once and that every node has exactly 0 or 2 children. A property of such trees states that the number of leaves is one more than the number of internal nodes => there are at most (1+2N) calls. Here: N = 6 jobs to schedule.

Fibonacci Numbers

Fibonacci Numbers

- Generate Fibonacci numbers
  - 3 solutions: inefficient recursive, memoization (top-down dynamic programming (DP)), bottom-up DP.
  - Not an optimization problem but it has overlapping subproblems => DP eliminates recomputing the same problem over and over again.

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2:

  Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- E.g.

0,	**1,**	**1,**	**2,**	**3,**	**5,**	**8,**	**13,**	**21,**	**34,**	**55,**	**89,**
0	1	2	3	4	5	6	7	8	9	10	11

- Write a function

  `int FibFct(int n)`

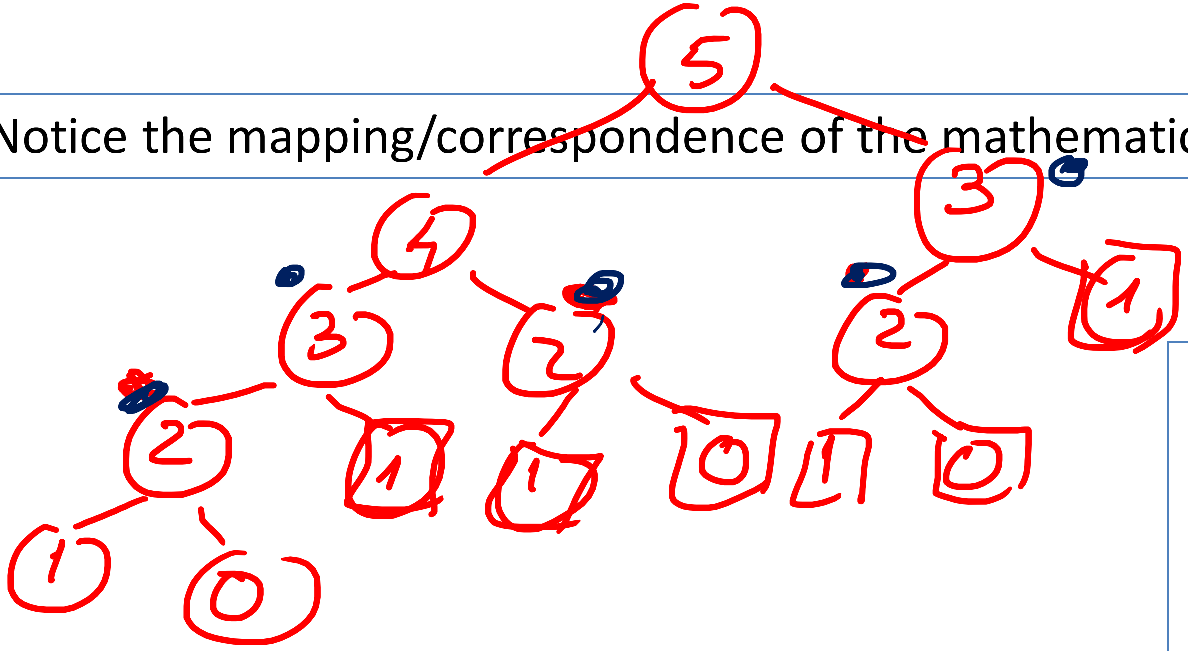
  that computes Fibonacci numbers

  E.g. `FibFct(7) -> 13` and `FibFct(1) -> 1`

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2:   Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Consider this function: what is its running time?

Notice the mapping/correspondence of the mathematical expression and code.



$O(2^i)$

```
int Fib(int i)
{
    if (i < 1) return 0;
    if (i == 1) return 1;
    return Fib(i-1) + Fib(i-2);
}
```

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Consider this function: what is its running time?
  - g(N) = g(N-1) + g(N-2) + constant

  $\Rightarrow$ g(N) ≥ Fibonacci(N) => g(N) = Ω(Fibonacci(N)) => g(N) = Ω(1.618^N)

  Also g(N) ≤ 2g(N-1)+constant => g(N) ≤ $c2^N$ => g(N) = O(2^N)

  => g(N) is exponential
  - We cannot compute Fibonacci(40) in a reasonable amount of time (with this implementation).
  - See how many times this function is executed.
  - Draw the tree

```
int Fib(int i)
{
   if (i < 1) return 0;
   if (i == 1) return 1;
   return Fib(i-1) + Fib(i-2);
}
```

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

Fib(100)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,

0 1 2 3 4 5 6 7 8 9 10 11

Notice the mapping/correspondence of the mathematical expression and code.

Bottom-up ,iterative, linear time:

$TC = \Theta(i)$

TOO $SC = \Theta(i)$

```
int Fib_iter (int i)   {
    int F[i+1];
    F[0] = 0;       F[1] = 1;
    int k;
    for (k = 2; k <= i; k++)
        F[k] = F[k-1] + F[k-2];
    return F[i];
}
```

Recursive ,exponential time:

```
int Fib(int i)   {
    if (i < 1) return 0;
    if (i == 1) return 1;
    return Fib(i-1) + Fib(i-2);
}
```

can use less space

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	0	1	1	2	3	5	8	13	21	34	53				

Applied scenario

- $F(N) = F(N-1)+F(N-2)$, $F(0) = 0$, $F(1) = 1$,
- Consider a webserver where clients can ask what the value of a certain Fibonacci number, F(N) is, and the server answers it.

  How would you do that? (the back end, not the front end)

(Assume a uniform distribution of F(N) requests over time most F(N) will be asked.)

- Constraints:
  - Each loop iteration or function call costs you 1cent.
  - Each loop iteration or function call costs the client 0.001seconds wait time
  - Memory is cheap
- How would you charge for the service? (flat fee/function calls/loop iterations?)
- Think of some scenarios of requests that you could get. Think of it with focus on:
  - "good sequence of requests"
  - "bad sequence of requests"
  - Is it clear what good and bad refer to here?

Fibonacci Numbers

- Fibonacci(0) = 0 , Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Alternative: remember values we have already computed.
- Draw the new recursion tree and discuss time complexity.

memoized :

```
int Fib_mem_wrap(int i) {
    int sol[i+1];
    if (i<=1) return i;
    sol[0] = 0;  sol[1] = 1;
    for(int k=2; k<=i; k++)  sol[k]=-1;
    Fib_mem(i,sol);
    return sol[i];
}
int Fib_mem (int i, int[] sol)  {
    if (sol[i]!=-1) return sol[i];
    int res = Fib_mem(i-1, sol) + Fib_mem(i-2, sol);
    sol[i] = res;
    return res;
}
```

exponential :

```
int Fib(int i)  {
    if (i < 1) return 0;
    if (i == 1) return 1;
    return Fib(i-1) + Fib(i-2);
}
```

Fibonacci and DP

- Computing the Fibonacci number is a DP problem.
- It is a counting problem (not an optimization one).
- We can make up an ‘applied’ problem for which the DP solution function is the Fibonacci function. Consider: A child can climb stairs one step at a time or two steps at a time (but he cannot do 3 or more steps at a time). How many different ways can they climb? E.g. to climb 4 stairs you have 5 ways: {1,1,1,1}, {2,1,1}, {1,2,1}, {1,1,2}, {2,2}

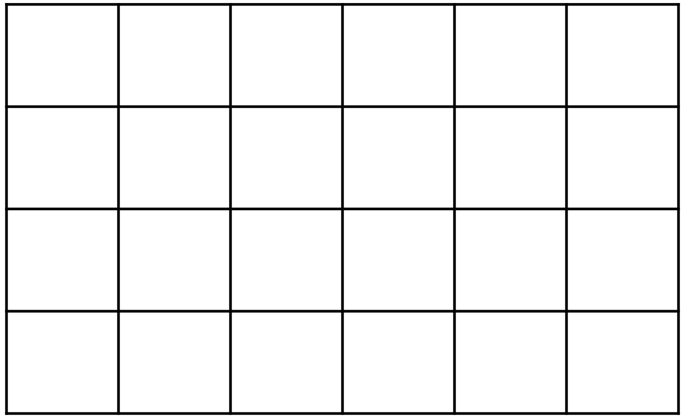
2D Matrix Traversal

P1. All possible ways to traverse a 2D matrix.

- Start from top left corner and reach bottom right corner.
- You can only move: 1 step to the right or one step down at a time. (No diagonal moves).
- Variation: Allow to move in the diagonal direction as well.
- Variation: Add obstacles (cannot travel through certain cells).

P2. Add fish of various gains. Take path that gives the most gain.

- Variation: Add obstacles.



Other DP Problems

- Stair climbing:
  - A child has to climb N stairs. She can jump over 1, 2 or 3 steps at a time. How many different way are there to climb the N stairs?
  - E.g. N=4 there are 6 ways:
    - {1,1,1,1},
    - {1,1,2},
    - {1,2,1},
    - {2,1,1},
    - {1,3},
    - {3,1}
- Make amount with smallest number of coins
- Matrix with gain
- House robber
- Many more on leetcode.

Variations of the Knapsack Problem

Unbounded:
Have unlimited number of each object.
Can pick any object, any number of times.
(Same as the stair climbing with gain.)

Fractional:
For each item can take the whole quantity, or a fraction of the quantity.

flour

soda

Bounded:
Have a limited number of each object.
Can pick object i, at most x_i times.

0-1 (special case of Bounded):
Have only one of each object.
Can pick either pick object i, or not pick it.
This is on the web.

All versions have:	
N	number of different types of objects
W	the maximum capacity (kg)
$v_1, v_2, \ldots, v_N$	Value for each object. (\$\$)
$w_1, w_1, \ldots, w_N,$	Weight of each object. (kg)

The bounded version will have the amounts: $c_1, c_2, \ldots, c_N$ of each item.

Application of the Knapsack problem

- https://en.wikipedia.org/wiki/Knapsack_problem

One early application of knapsack algorithms was in the construction and scoring of tests in which the test-takers have a choice as to which questions they answer. For small examples, it is a fairly simple process to provide the test-takers with such a choice. For example, if an exam contains 12 questions each worth 10 points, the test-taker need only answer 10 questions to achieve a maximum possible score of 100 points. However, on tests with a heterogeneous distribution of point values, it is more difficult to provide choices. Feuerman and Weiss proposed a system in which students are given a heterogeneous test with a total of 125 possible points. The students are asked to answer all of the questions to the best of their abilities. Of the possible subsets of problems whose total point values add up to 100, a knapsack algorithm would determine which subset gives each student the highest possible score

Worksheet: 0-1 Knapsack Example 1

Max capacity: W=8		
item	Weight (Kg)	Value ($)
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

Examples:
max capacity: W = 8
pick: A -> value_______, weight: _______ , fits? Y/N

pick: A,C -> value_______, weight: _______ , fits? Y/N

pick: A,B,D -> value_______, weight: _______ , fits? Y/N

pick: A,B,C,D -> value_______, weight: _______ , fits? Y/N

Best value was ______
Did we try all possible combinations?
Are we certain there was no better one?

What is a smaller problem than this? What affects pb size(s)?

What problem is trivial? (think 0)

Think of an optimal solution.

- can you see a last step/choice? (can you see choices?)
  - Or can you see a place where it breaks into subproblems?
  - Here you may redefine what a problem looks like
    - Something that allows an ordering of subproblems or writing one solution in terms of solutions to smaller pbs.

Table? Array?

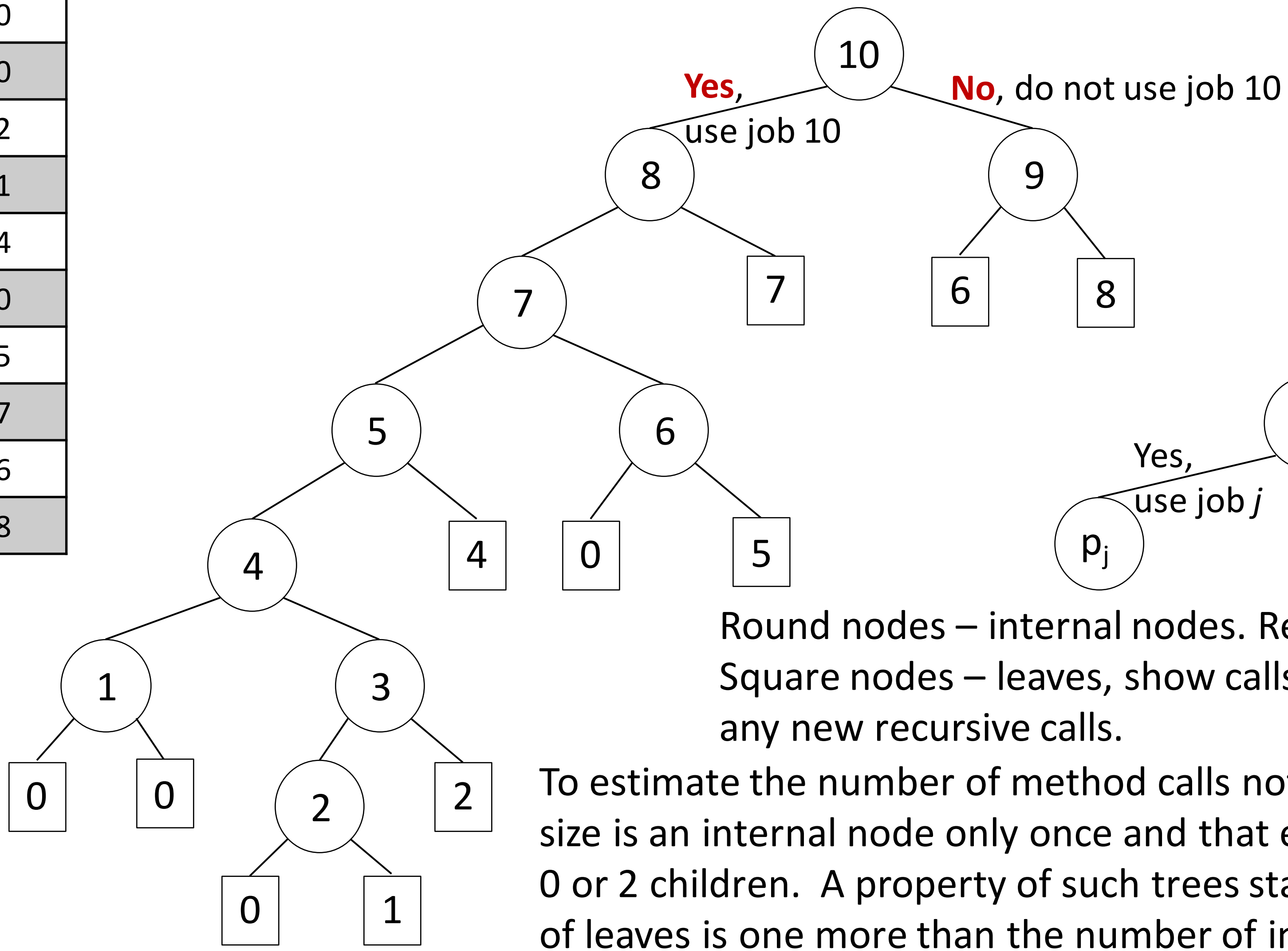
Worksheet: **0-1 Knapsack** Example 1

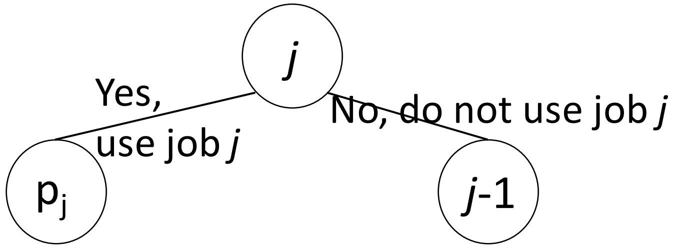
Write the formula for the solution function:

Max capacity: W=8		
item	weight	Value
A	4	*5*
B	3	*4*
C	2	*3*
D	1	*2*

Function call tree for the memoized version

Job j	p(j)
0	-1
1	0
2	0
3	2
4	1
5	4
6	0
7	5
8	7
9	6
10	8





Round nodes – internal nodes. Require recursive calls.
Square nodes – leaves, show calls that return without any new recursive calls.

To estimate the number of method calls note that every problem size is an internal node only once and that every node has exactly 0 or 2 children. A property of such trees states that the number of leaves is one more than the number of internal nodes => there are at most (1+2N) calls. Here: N = 10 jobs to schedule.