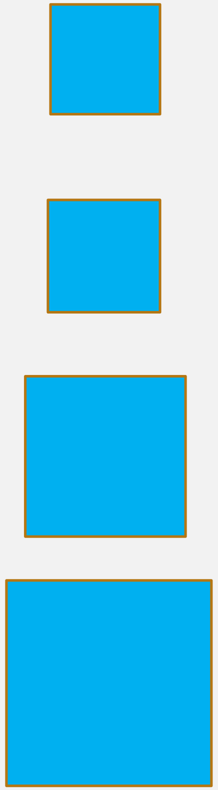
WEEK 11
DYNAMIC PROGRAMMING ALGORITHM

2024-03-28

0/1 KNAPSACK PROBLEM

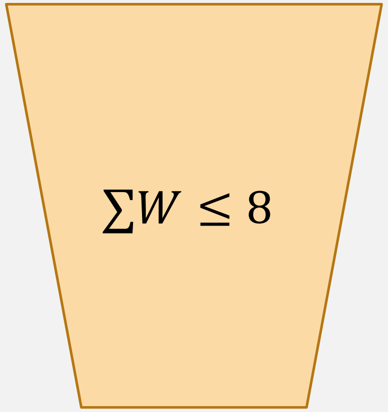


P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5



- We have a bag with capacity of 8 kg.
- There are 4 objects with different weights and profits.
- 0-1 means that you can either choose or not choose the object, not cut in a half.
- We want to carry objects so that the total weight does not exceed the bag capacity, and the total profit is maximized.

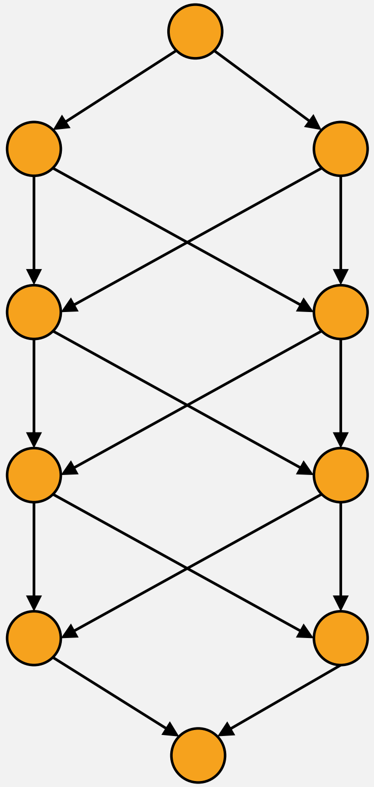
SAME OPTIMIZATION PROBLEM, WITH DIFFERENT APPROACH

- It is still the problem about decision to choose or not to choose the items, in order to optimize some objective.
- Greedy algorithm only cover just a few of these problem. Many of the rest can be solved by "dynamic programming algorithm".

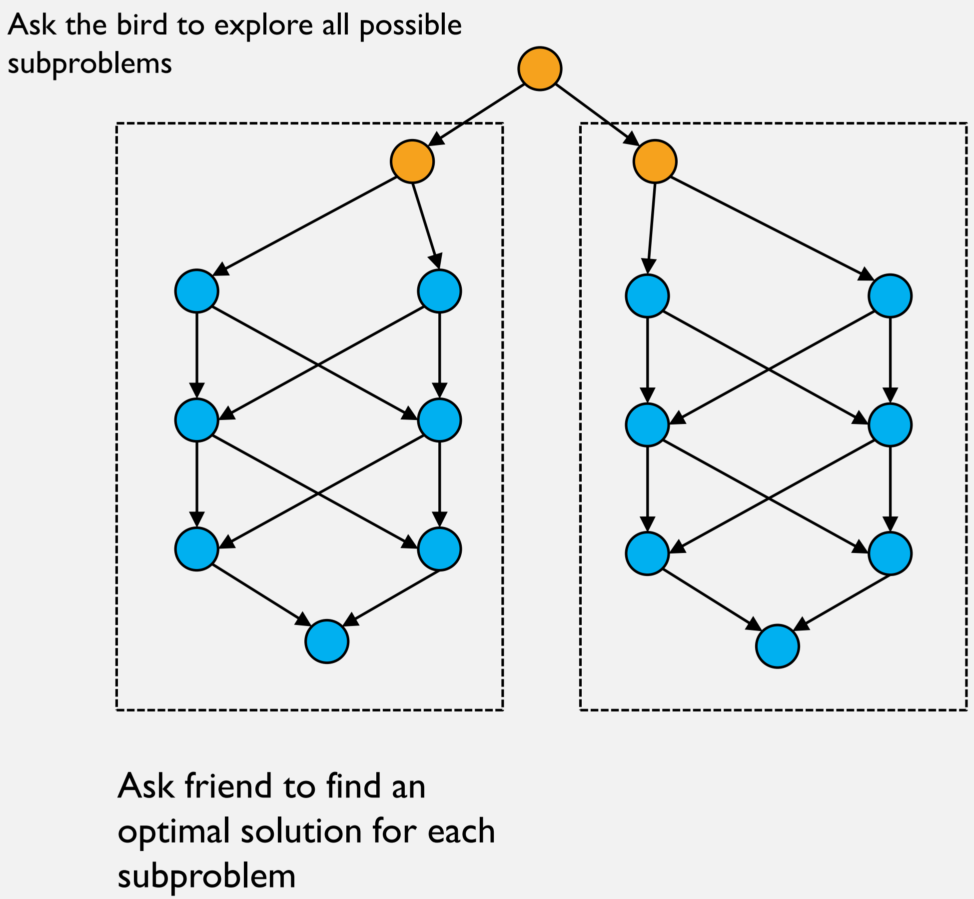
- How can we guarantee the optimality of the solution in any optimization problem?

BRUTE FORCE

- There are 4 steps of choosing / not choosing an object
- Each choice is independent from others
- Trying all possible ways takes $2^{O(n)}$
- Very inefficient

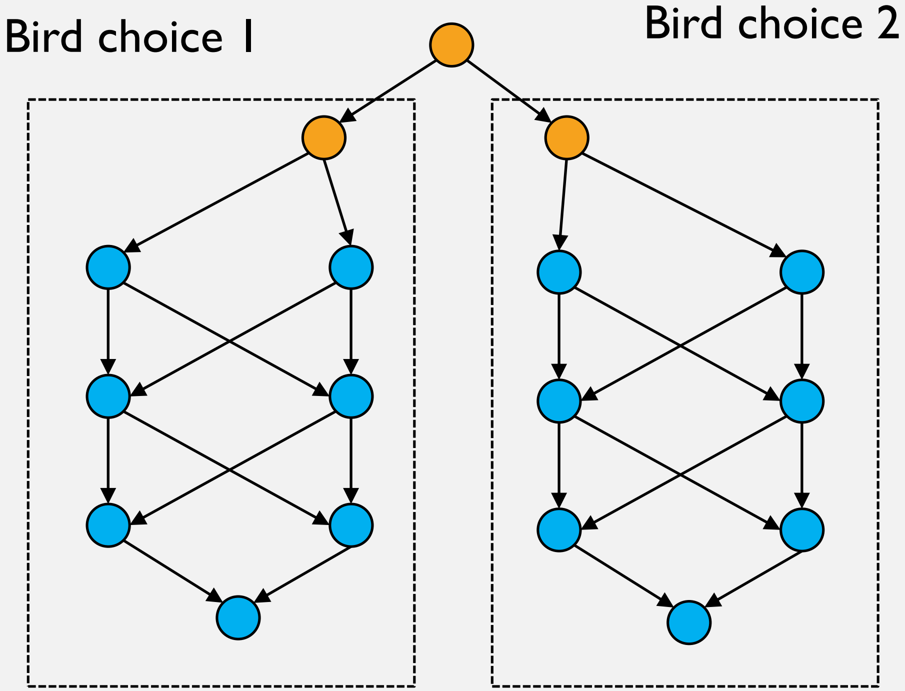


RECURSIVE BACKTRACKING



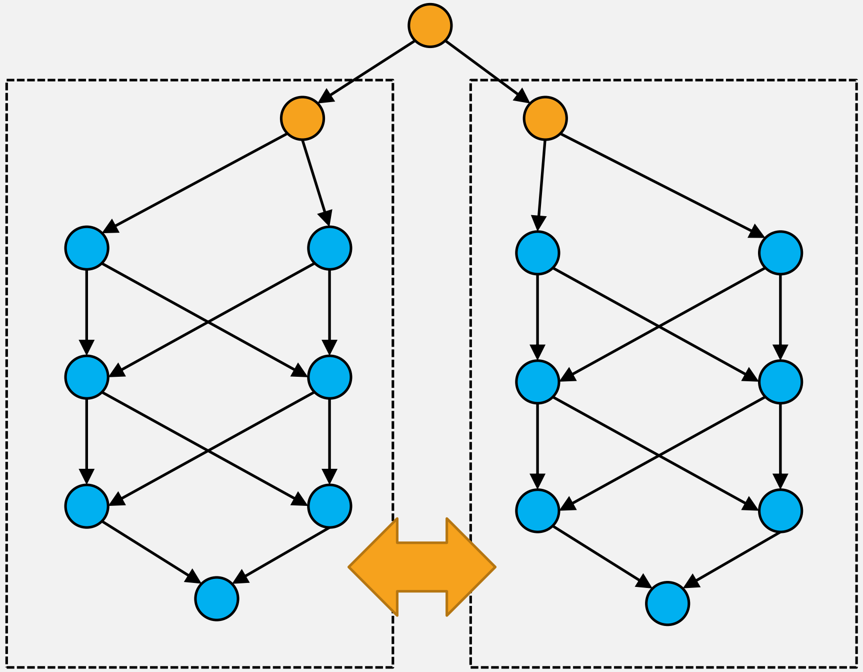
- Recursive algorithm:
  - Break down the big problem into some smaller parts that friends can help.
  - Let friends solve the same problem, but with the smaller input.
- Survey all the possible subproblem
  - Use a bird to guide to all possible subproblems
- In the textbook, it is called "bird and friend" algorithm

HOW MANY LOOPS DONE BY BIRDS AND FRIENDS?



- Unwinding the recursion, we can see the entire tree of all possible choices.
- Same size as the brute force
- In fact, it is just a rearrangement of the brute force

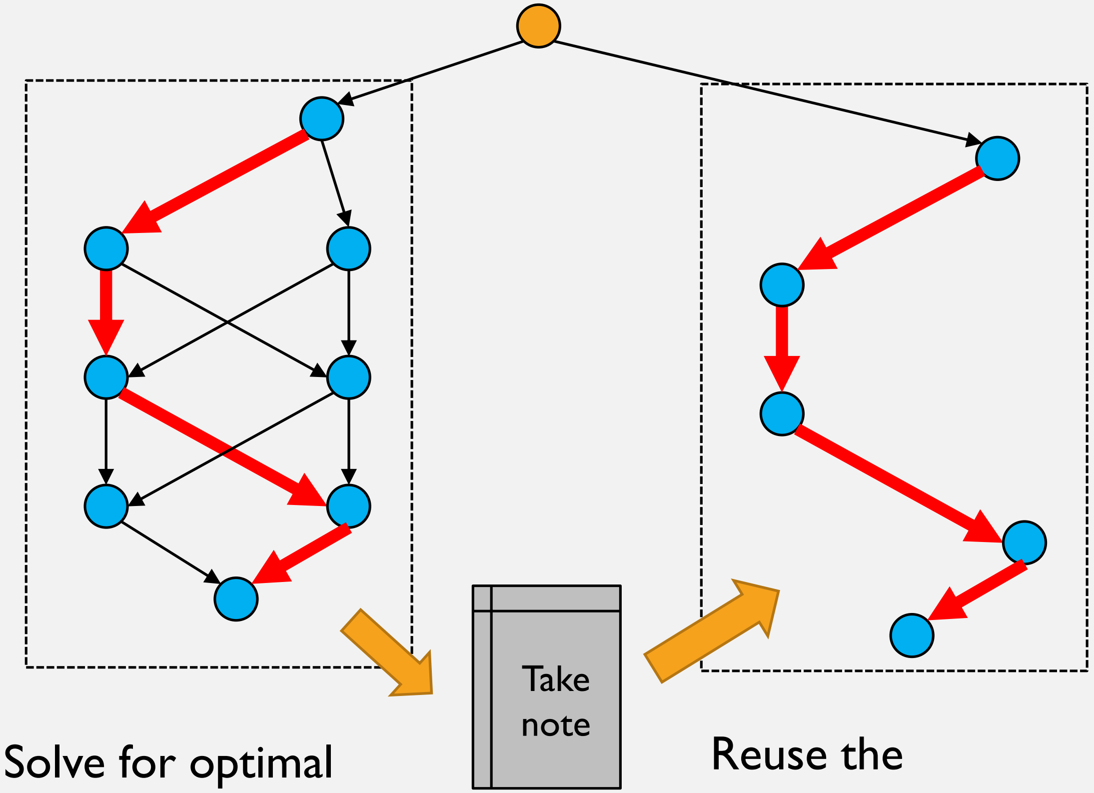
REDUCING REDUNDANCY



They are the exact same task

- Independent of birds' choice, works done by friends are the same, no matter what the bird choice is.
- We can reduce the redundancy of the friends work by **taking a note**.

DYNAMIC PROGRAMMING ALGORITHM

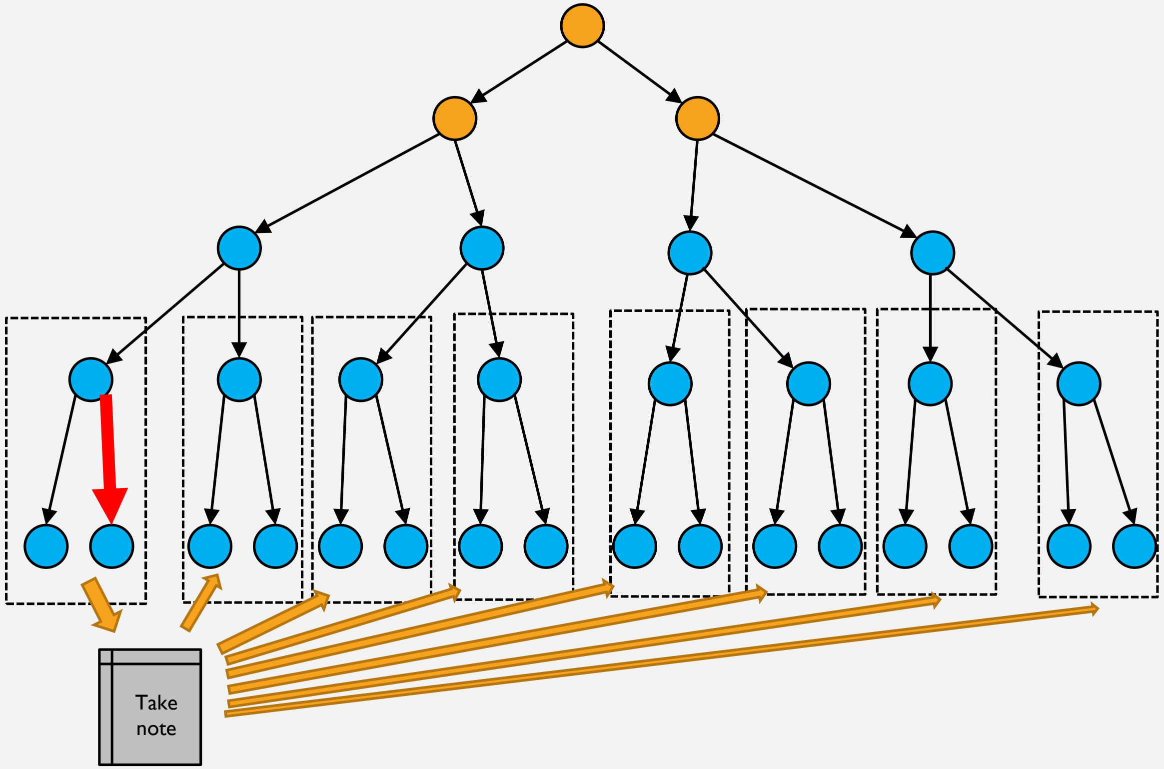


Solve for optimal solution just once

Reuse the recorded optimal solution

- Rearrangement of the work done in the brute force algorithm so that the each of the common sub-instance is optimized just once and is recorded for the future use.
- Next time the bird explores the same sub-instance again, the next friend can reuse the optimal solution solved by the first friend.

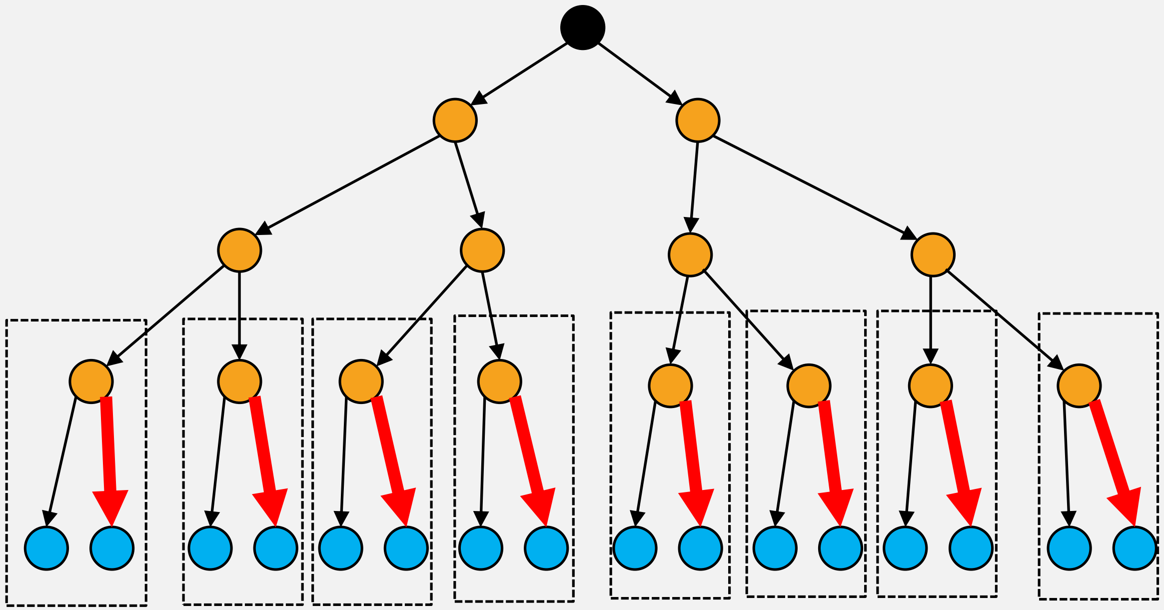
RECURSION FROM TOP TO BOTTOM



Solve it just once, then copy

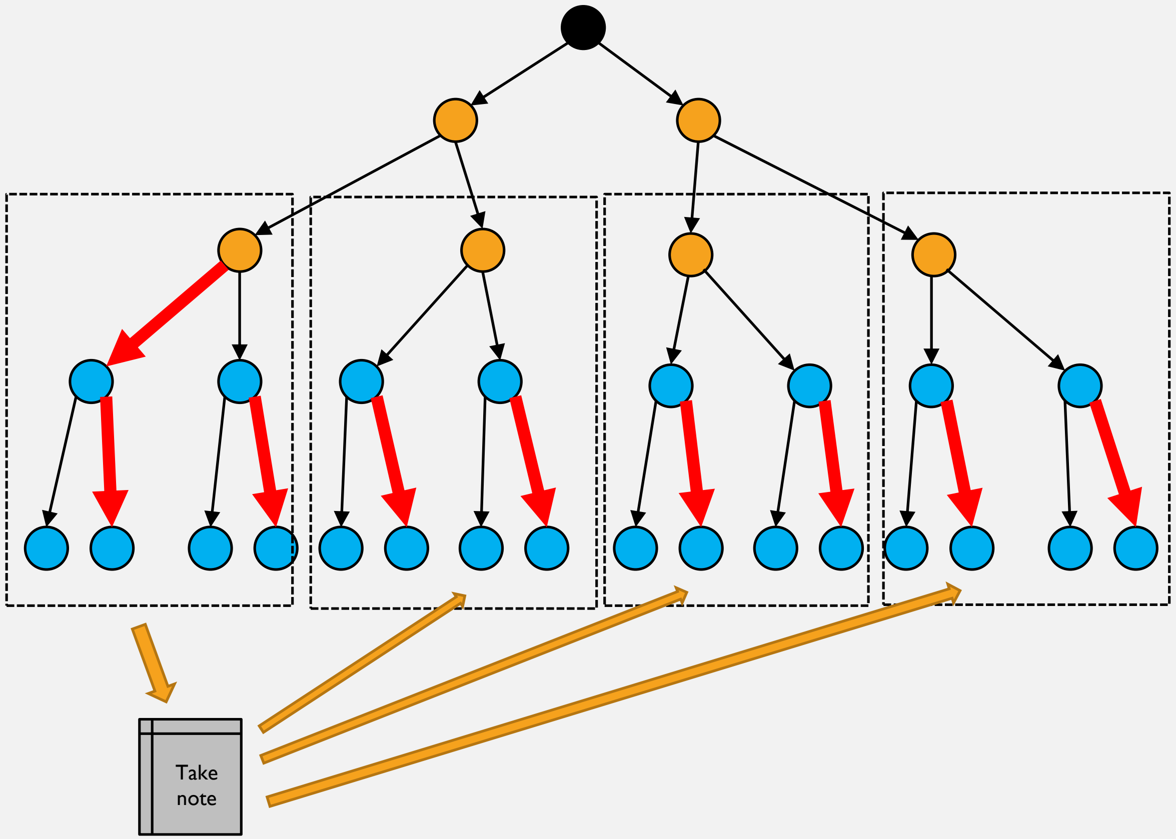
- The recursion call for solving the smaller problems from top to bottom.
- The recursion really get the optimal solution from the lowest level first (base case with sub-instance size = 1).
- Then the upper level takes these optimal solutions as a part of the longer solution.

GET THE SOLUTION FROM BOTTOM TO TOP



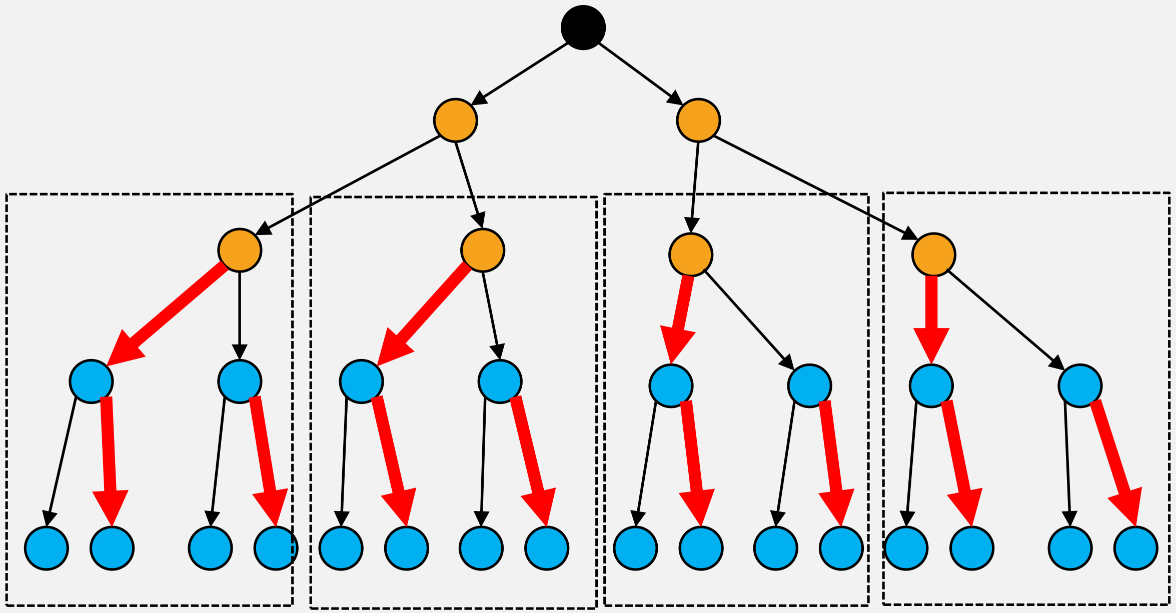
Optimal solution
for the last object

BEST OF THE LAST 2 OBJECTS



Optimal solution for
the last 2 objects

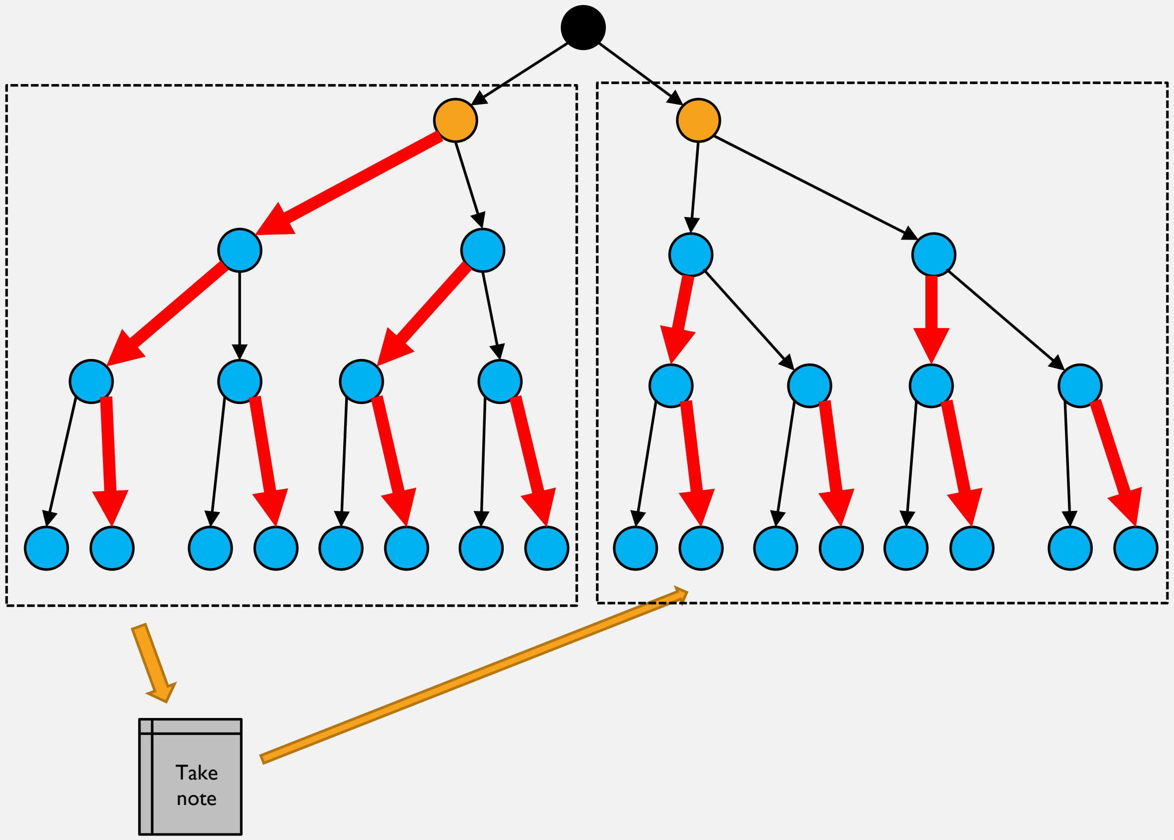
BEST OF THE LAST 2 OBJECTS



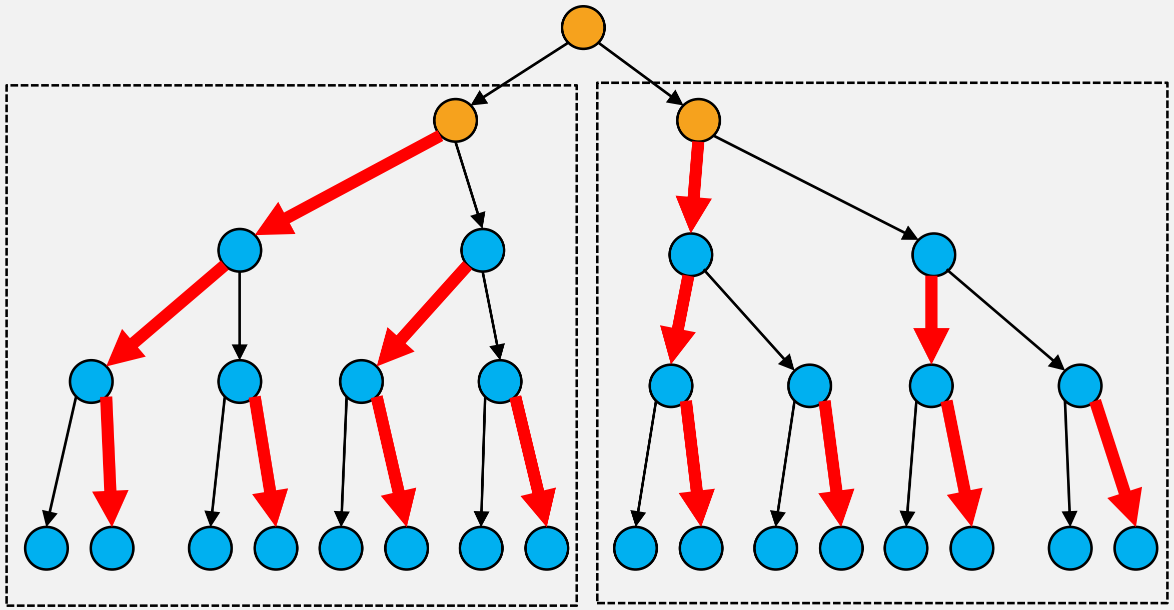
Optimal solution for the last 2 objects

BEST OF THE LAST 3 OBJECTS

Optimal solution for the last 3 objects



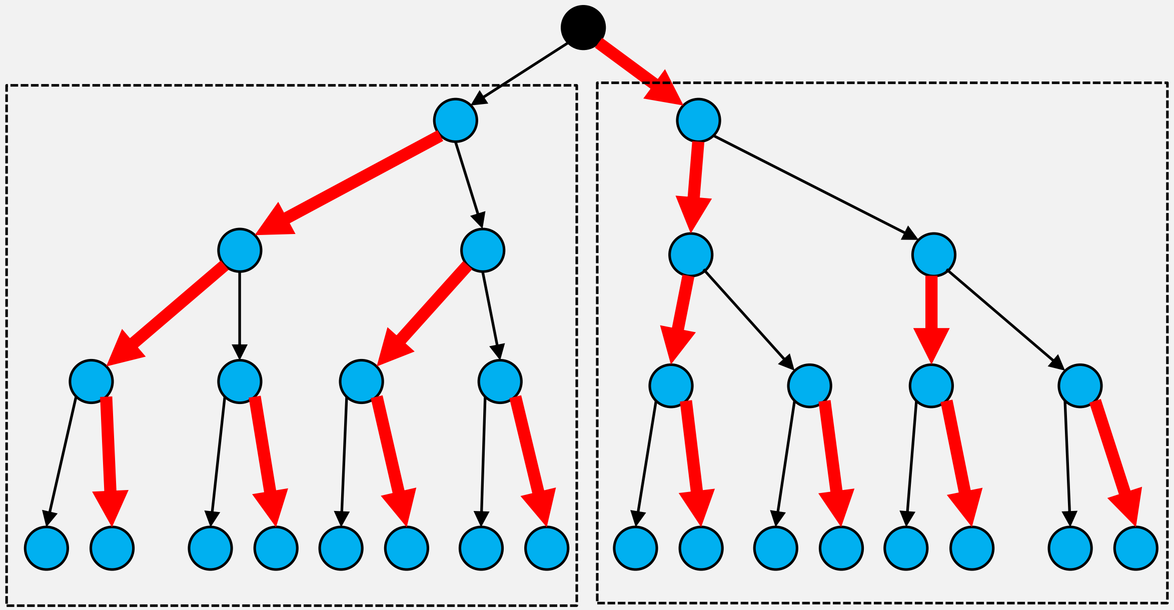
BEST OF THE LAST 3 OBJECTS



Optimal solution for
the last 3 objects

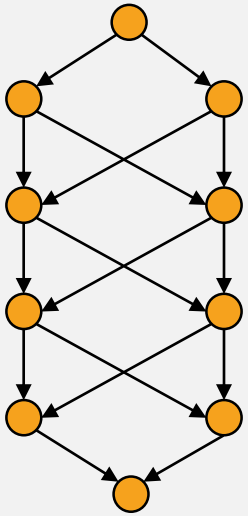
BEST OF ALL 4 OBJECTS

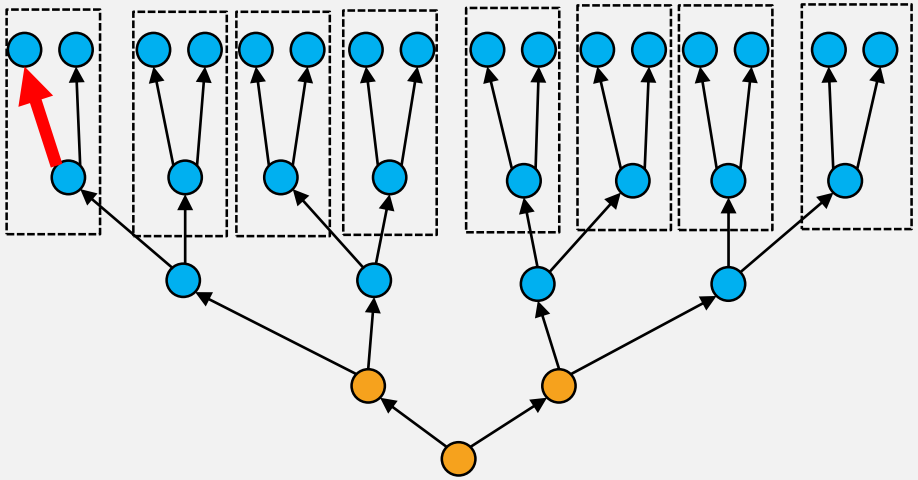
Optimal solution for all 4 objects



TURN UPSIDE DOWN

First item will be the
first one to take note





Explore choices for
the last item first

- Friends start taking note for the optimal solution on the last item first.
- It is ok, but it is better to start with the first item.
- Turning it upside down
  - Let the bird explores choices on the last item
  - Recursion on the smaller sub-instance, which is the first $n-1$ items.
  - The first item will be the first one to take note
  - So it makes more sense to start working with the first item

OVERALL DYNAMIC PROGRAMMING

- Take note among choices of the first object
- Proceed to the sub-instance with one more object. Compare between
  - The previous optimal solution with one less object
  - Some other smaller optimal solution combined to the current choice
  - Take the better choice, record the profit to the table.

FIBONACCI

0	1	2	3	4	5	6	7	8
0	1	1	2	3	5	8		

$f(n-1)$ and $f(n-2)$ are recorded in the table already. No need to recalculate

- Recursion vs dynamic programming
- Recursive Fibonacci takes a long time since $f(n-1)$ and $f(n-2)$ needs to be recalculate every time.
- If we just "take note" of what has been calculate, we can just go back and take it.

0/1 KNAPSACK PROBLEM WITH DYNAMIC PROGRAMMING

- **Bird**: explore every choices of choosing the last item.
- **Friend**: find optimal choice that maximize profit for the first n-1 items.

- **Dynamic programming**: First friend takes note for the optimal solution size n-1 so that other friend can use this information as well.
- Starts from the smallest n first.

WHAT TO TAKE NOTE

- Values in the table are the optimal profits of the smaller problems
- Smaller problem
  - Smaller bag size
  - Fewer number of objects
- Notetaking must be a 2D table storing optimal profit for all possible bag size and subset of the objects.

	1	2	3	4	5	6	7	8
0								
1								
2								
3								
4								

Optimal solution for bag size 4 with at most the first 2 objects

Optimal solution for bag size 6 with at most the first 3 objects

FILLING THE TABLE

P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

FIRST ROW

P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1									
2									
3									
4									

Optimal solution for 0 object with any size of the bag

NEXT ROW (FIRST OBJECT)

P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2									
3									
4									

Case I: bag size $<$ first object size $\rightarrow$ 0

Case II: bag size $\geq$ first object size $\rightarrow$ Profit of the first object

NEXT ROW (FIRST OBJECT)

		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
P = 1, W = 2	1	0	0	1	1	1	1	1	1	1
P = 2, W = 3	2									
P = 5, W = 4	3									
P = 6, W = 5	4									

Choice 1: not use the current object

NEXT ROW (FIRST OBJECT)

P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2									
3									
4									

Choice 2: use the current object
Must spare the empty slot for this object (w = 3)
Then add the profit of the current object (p = 2)

NEXT ROW (FIRST OBJECT)

P = 1, W = 2

P = 2, W = 3

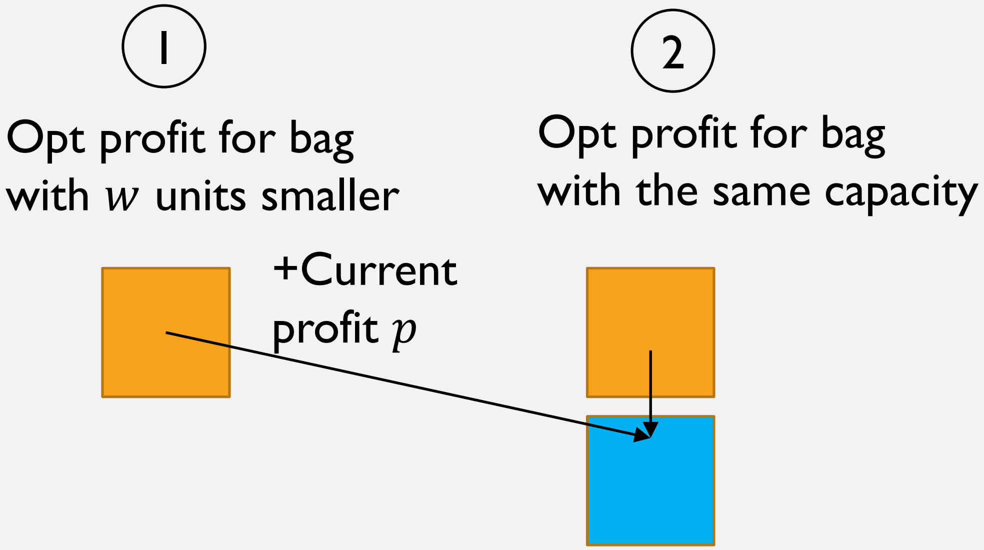
P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2					2				
3									
4									

+2

ANY OTHER OBJECT



- Example: weight = 3, profit = 2

- In order to put the second object, the bag must have 3 unit left.
- Refer to the optimal solution of the bag with 3 unit smaller.

- In each column j, there are 2 cases to compare
  - Optimal solution of column j in the previous row
  - Add profit of the second object to the optimal solution of the bag size $j - 3$

IN GENERAL

- In each column j, there are 2 cases to compare
  - Optimal solution of column j in the previous row
  - Add profit of the current object (p_i) to the optimal solution of the bag size $j - w_i$

$$opt_{i,j} = \max(opt_{i-1,j}, opt_{i-1,j-w_i} + p_i)$$

P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2									
3									
4									

P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3									
4									

		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
P = 1, W = 2	1	0	0	1	1	1	1	1	1	1
P = 2, W = 3	2	0	0	1	2	2	3	3	3	3
P = 5, W = 4	3	0	0	1	2	5	5	6	7	7
P = 6, W = 5	4									

		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
P = 1, W = 2	1	0	0	1	1	1	1	1	1	1
P = 2, W = 3	2	0	0	1	2	2	3	3	3	3
P = 5, W = 4	3	0	0	1	2	5	5	6	7	7
P = 6, W = 5	4	0	0	1	2	5	6	6	7	8

TRACE FOR THE CHOICES

Item		0	1	2	3	4	5	6	7	8	
	0	0	0	0	0	0	0	0	0	0	
P = 1, W = 2	1	0	0	1	1	1	1	1	1	1	✗
P = 2, W = 3	2	0	0	1	2	2	3	3	3	3	✓
P = 5, W = 4	3	0	0	1	2	5	5	6	7	7	✗
P = 6, W = 5	4	0	0	1	2	5	6	6	7	8	✓

W = 5

If the profit equals to the number on top, the object is not chosen.

Otherwise, step back to the left by w columns. The object is chosen.

LOOP INVARIANT

- General : what is stored in the table is the optimal value of all smaller subproblems.

- For 0-1 knapsack problem: Value in row i column j is the optimal profit for the first i objects in the bin size j.

(STRONG) INDUCTION PROOF

- Assume that all of the rows above (until $i-1$) and every preceding columns (until $j-1$) in the same row is filled with the optimal profit.
- We have to show that

$$\max(opt_{i-1,j}\ , opt_{i-1,j-w_i} + p_i)$$

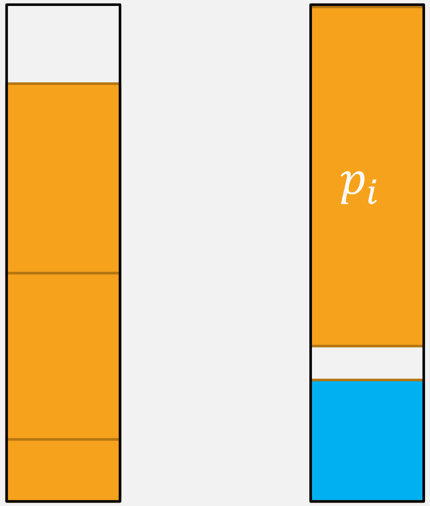
in the current cell is the optimal profit for i objects in the bag size j.

BASE CASE

- When using no object at all, it is clear that the profit = 0 in any bag size.
- So, the optimal profit of 0 object is 0.

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1									
2									
3									
4									

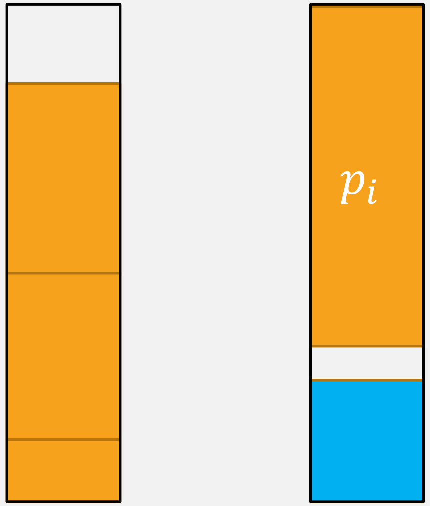
INDUCTIVE STEP



$opt_{i-1,j}$ $\quad$ $opt_{i-1,j-w_i} + p_i$

- There are only two choices to get the candidate of the optimal profit for bag size j:
  <u>include object i</u> or <u>not include object i</u>.
  - Case 1: Object i is included, then use $opt_{i-1,j}$
    - Since $opt_{i-1,j}$ is the optimal profit for the first $i-1$ objects,
    - We do not include object i in here.
    - So $opt_{i-1,j}$ is still the optimal profit for the first i objects, when not including object i.

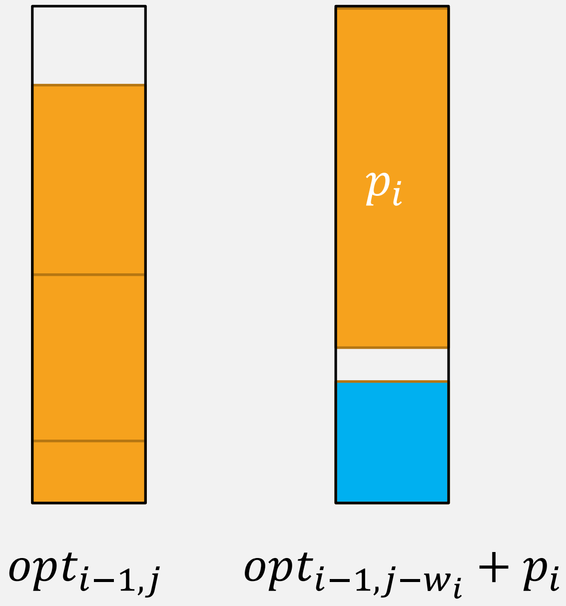
INDUCTIVE STEP



$opt_{i-1,j}$ $\quad$ $opt_{i-1,j-w_i} + p_i$

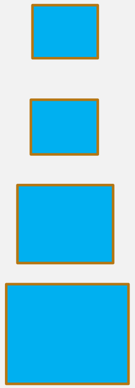
- Case 2: Object i is not included, then use $opt_{i-1,j-w_i} + p_i$
  - We need w_i empty space for including object i inside the bag size j. The remaining space is $j - w_i$.
  - In the remaining space, $opt_{i-1,j-w_i}$ is the optimal profit for the first $i - 1$ objects for the bag size $j - w_i$.
  - After including object i, we have that $opt_{i-1,j-w_i} + p_i$ is the optimal profit for the first i objects when including object i.

INDUCTIVE STEP



- The maximum value among the two cases is the highest possible value for profit with i objects within the bag size j.
- So we found the optimal profit for i objects within the bag size j.

TIME COMPLEXITY



P = 1, W = 2

P = 2, W = 3

P = 5, W = 4

P = 6, W = 5

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2					2				
3									
4									

+2

- The task is filling the table size

$$(w+1)(n+1)$$

Where w is the capacity of the bag, and n is the number of objects.

Time complexity for the knapsack problem is

$$O(wn)$$

For larger n, this big-O is much less than $2^{O(n)}$.

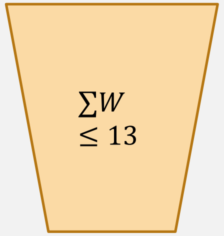
P = 3,   W = 4

P = 4,   W = 2

P = 9,   W = 7

P = 6,   W = 5

P = 4,   W = 3



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0														
1														
2														
3														
4														
5														