Dynamic Programming

Part I

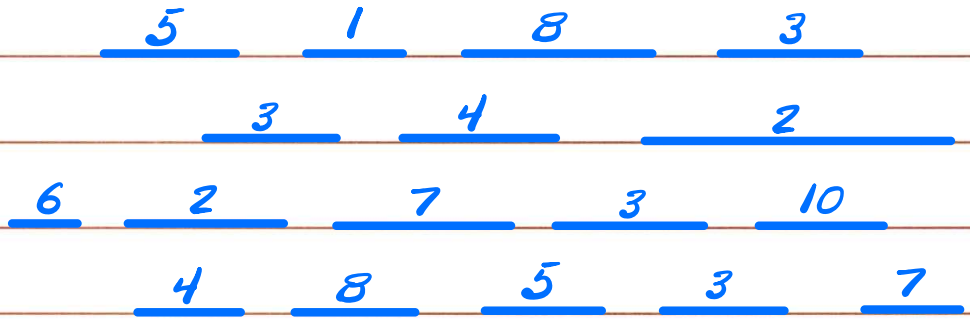
General Approach to Solving Optimization Problems Using Dynamic Programming

1. Characterize the structure of an opt. solution
2. Recursively define the value of an opt. solution
3. Compute the value of an opt. solution in a bottom up fashion
4. Construct an opt. solution from computed information

Weighted Interval Scheduling Problem

Input: A set of requests $\{1..n\}$, where the i^{th} request starts at $s(i)$, ends at $f(i)$, and has weight $w(i)$

Output: A subset of requests $S \subseteq \{1..n\}$ of mutually compatible intervals so as to maximize $\sum_{i \in S} w(i)$



Observation: Either job i is part of an optimal solution, or it isn't.

Case 1 - If it is,

value of the optimal solution =
w(i) + value of the optimal solution for the subproblem that consists only of compatible requests with job i.

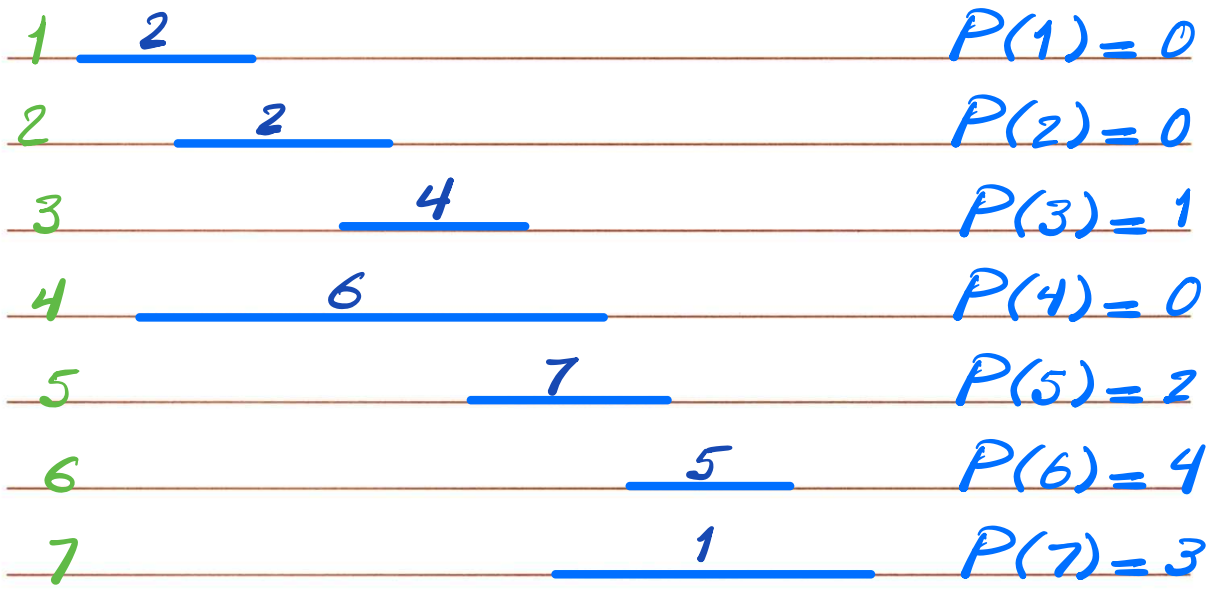
Case 2 - If it isn't,

value of the optimal solution =
value of the optimal solution without job i.

To find compatible subproblems quickly, sort requests in order of non-decreasing finish time

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

And then define $P(j)$ for an interval j to be the largest index $i < j$ such that intervals i and j are disjoint.



Def. Let O_j denote the optimal solution to the problem consisting of requests $\{1 \ldots j\}$, and let $OPT(j)$ denote the value of O_j

So, O_3 will be $\{1,3\}$, and $OPT(3) = 6$

We can now use this notation to describe the two cases:

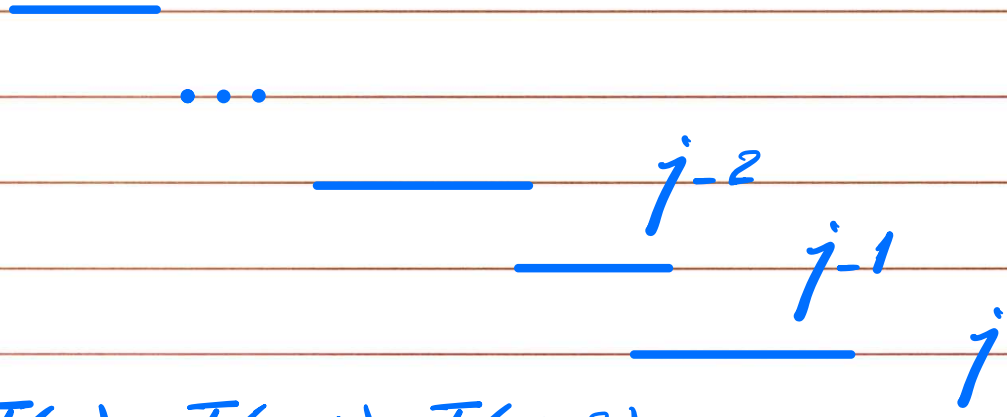
Case 1: $j \in O_j \Rightarrow OPT(j) = w(j) + OPT(P(j))$

Case 2: $j \notin O_j \Rightarrow OPT(j) = OPT(j-1)$

Solution

```
Compute_Opt(j)
    if j=0 then
        Return 0
    else
        Return Max(
            w(j)+Compute_Opt(P(j)),
            Compute_Opt(j-1)
    endif
```

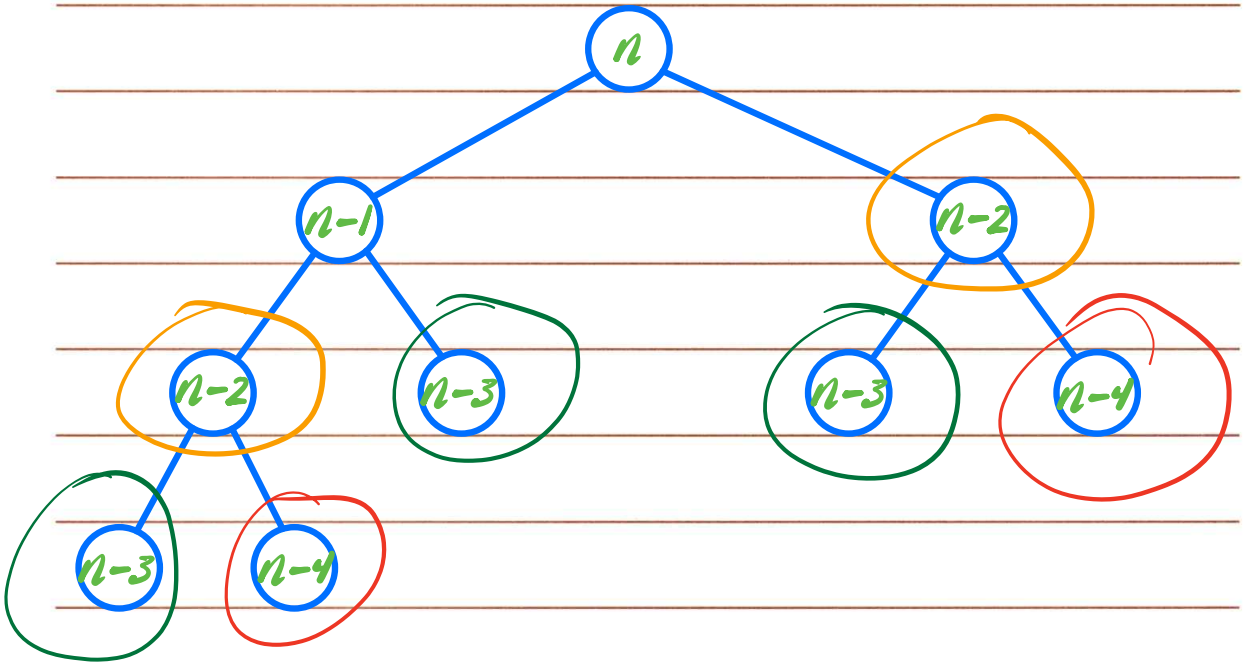
Worst case occurs when the sizes of subproblems go down very slowly, e.g.:



$T(n) = T(n-1) + T(n-2)$

↖ similar to the Fibonacci series, $T(n)$ grows exponentially!

Check recursion tree for redundancies



Observation: We are solving the same exact subproblems multiple times

Memoization

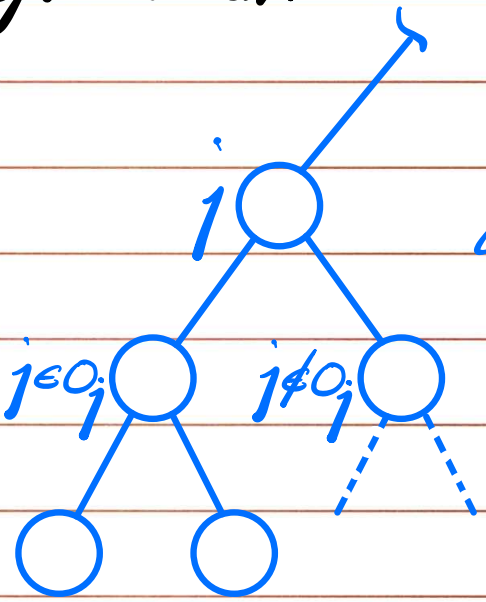
Store the value of Compute_Opt in a globally accessible place the first time we compute it. Then simply use this precomputed value in place of all future recursive calls.

```
M_Compute_Opt(j)
  if j=0 then
    Return 0
  else if M[j] is not empty then
    Return M[j]
  else
    Define M[j] = Max(
      w(j) + M_Compute_Opt(P(j)),
          M_Compute_Opt(j-1))
    Return M[j]
  endif
```

Complexity Analysis

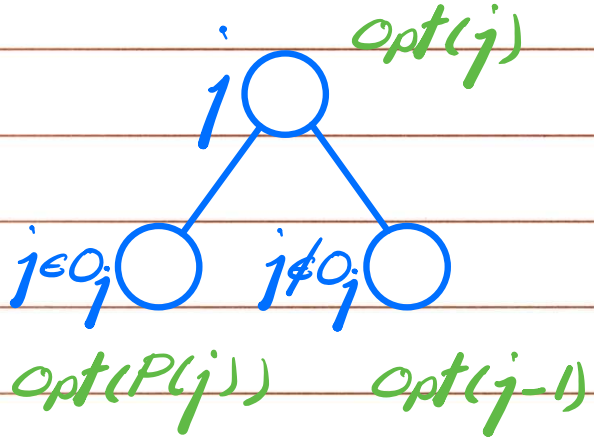
Initial Sorting:	$O(n \lg n)$
Build the P() array:	$O(n \lg n)$
M_Compute_Opt	$O(n)$
Overall complexity	$O(n \lg n)$

Cannot find the optimal solution in the bottom-up pass_only the value of the opt. solution.



At a higher up node we might realize this whole branch is not optimal.

Top-down pass: Compute an opt. solution



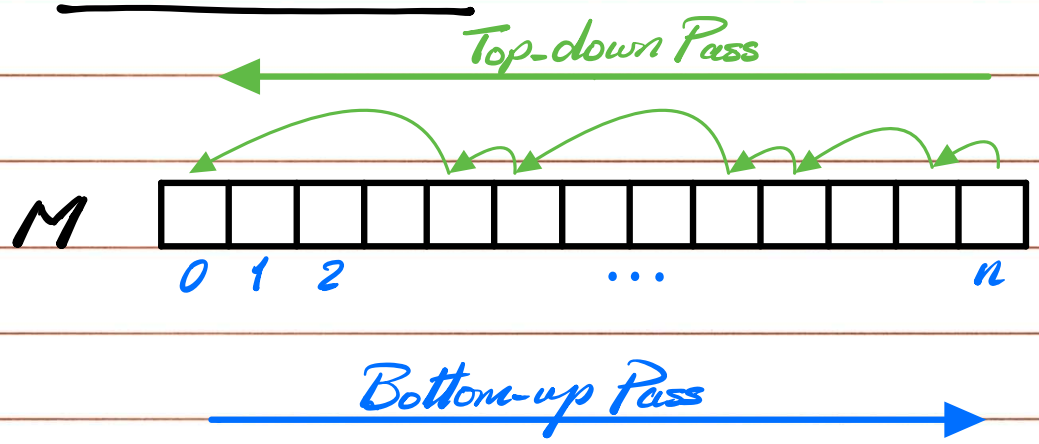
Having computed the value of the optimal solution for all unique subproblems in the bottom-up pass, it is easy to determine if j does or does not belong to an opt. solution:

j belongs to Oj iff

$$w(j) + OPT(P(j)) \geq OPT(j-1)$$

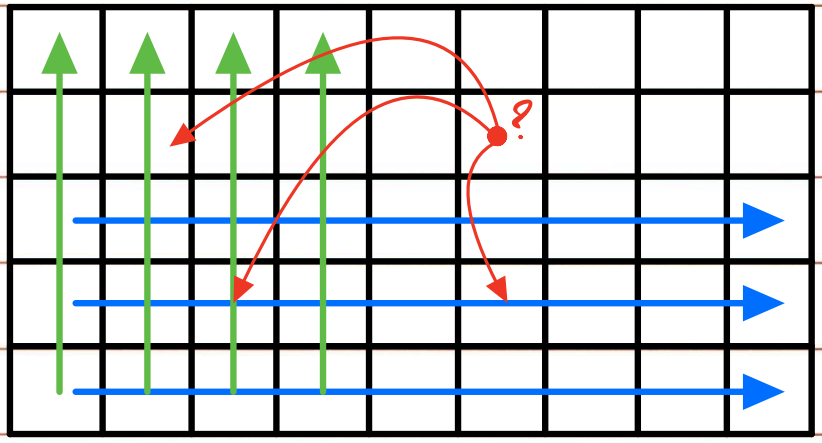
```
Find_Solution (j)
  if j > 0 then
    if w_j + M[P(j)] ≥ M[j-1] then
        Output j together with the
        results of Find_Solution(P(j))
    else
        Output the results of
            Find_Solution(j-1)
    endif
  endif
```

Iterative Solution

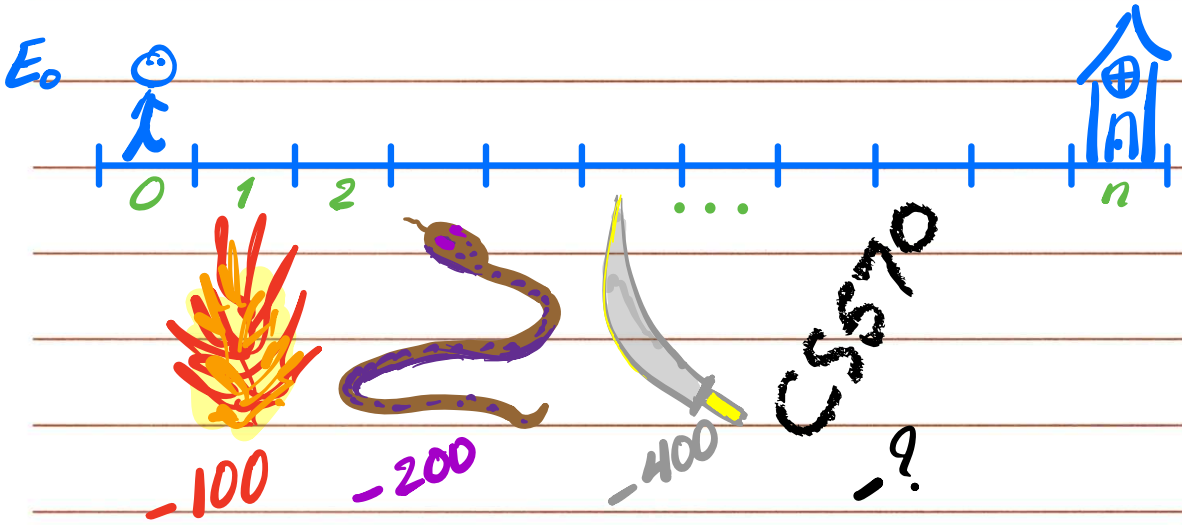



```
M[0] = 0
for i = 1 to n
    M[i] = Max(M[i-1], w(i) + M[P(i)])
endfor
```

Usually the values of the opt. solutions to unique subproblems are stored in 1, 2, 3, ... dimensional arrays. The order in which these values are computed depends on the recurrence formula.



Video Game Problem

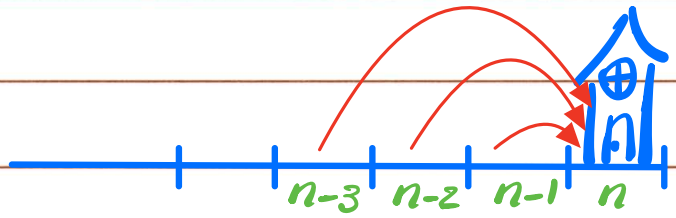


In general, the player loses e_i units of energy when landing in stage i.

Choices at each stage:

1- Walk to the next stage.
   This requires 50 units of energy
2- Jump over one stage
   This requires 150 units of energy
2- Jump over two stages
   This requires 350 units of energy

Objective: To reach home with the maximum level of remaining energy.



OPT(i) = Optimal level of energy when the player reaches stage i

Recurrence Formula:

$OPT(i) = Max(OPT(i-1) - 50 - e_i,$
$OPT(i-2) - 150 - e_i,$
$OPT(i-3) - 350 - e_i)$

Bottom up pass

$OPT(0) = E_0$

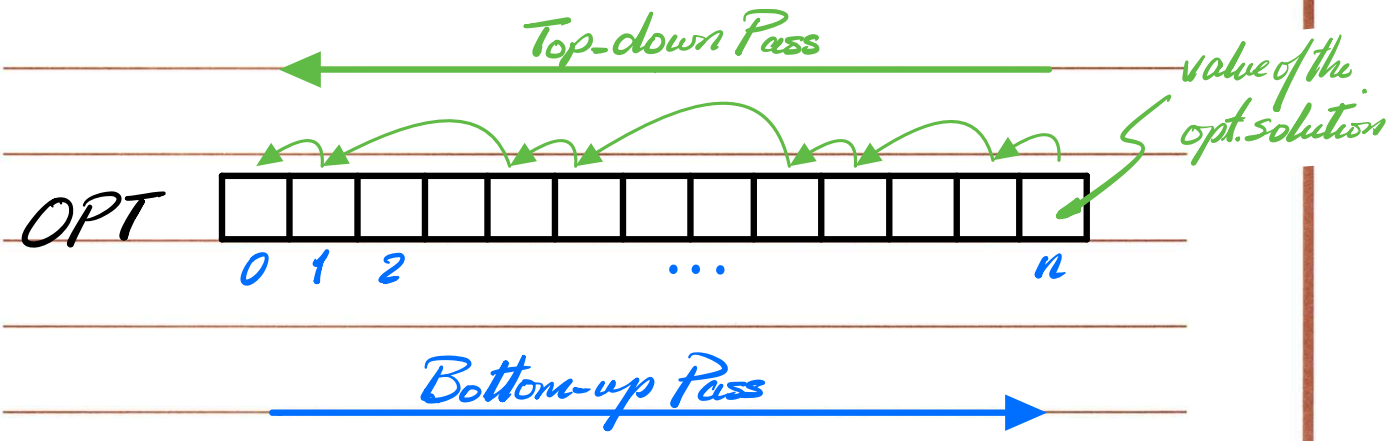
$OPT(1) = E_0 - 50 - e_1$

$OPT(2) = Max(E_0 - 150 - e_2, OPT(1) - 50 - e_2)$

for i = 3 to n

$OPT(i) = Max(OPT(i-1) - 50 - e_i,$
$OPT(i-2) - 150 - e_i,$
$OPT(i-3) - 350 - e_i)$

endfor



Top down pass

```
j=n
While(j>0)
    if OPT(j)=OPT(j-3)-350_ei then
        Print "from stage", j-3, "jump over 2 stages"
        j=j-3
    else if OPT(j)=OPT(j-2)-150_ei then
        Print "from stage", j-2, "jump over 1 stage"
        j=j-2
    else
        Print "from stage", j-1, "walk to next stage"
        j=j-1
endwhile
```

Coin Problem

Austrian coin denominations (Pre-Euro)

1

5

10

20

25

Question: How can we make change for n schillings using the minimum no. of coins?

$OPT(i)$ = Min no. of coins to make i schillings

Recurrence formula:

$$OPT(i) = \text{Min}\,(\,OPT(i-1)+1,\; OPT(i-5)+1,\; OPT(i-10)+1,\; OPT(i-20)+1,\; OPT(i-25)+1\,)$$

Bottom up pass

$OPT(0) = 0$
$OPT(1) = 1$
$\vdots$
$OPT(24) =$
} initializations

for $i = 25$ to n

$OPT(i) = \min(OPT(i-1) + 1,$
$OPT(i-5) + 1,$
$OPT(i-10) + 1,$
$OPT(i-20) + 1,$
$OPT(i-25) + 1)$

endfor

Time complexity : $O(n)$

0/1 Knapsack

&

Subset Sum

Subset Sum Problem

- Given a single resource available for W units of time, and
- A set of requests $\{1 \ldots n\}$ that can be scheduled at any time between 0 to W.
- And where request i takes w_i time to process,
- The objective is to schedule jobs such that we maximize the resource's utilization

First attempt to define the unique subproblems: (false start)

$OPT(i)$ = Value of the opt. solution for requests $1 \ldots i$.

Not the same* subproblem!

if $n \notin O_n$, then $OPT(n) = OPT(n-1)$

if $n \in O_n$, then $OPT(n) = w_n + OPT(n-1)$

* One subproblem has less remaining time.

Second attempt:

$OPT(i, w)$ = value of the opt. solution using a subset of the items $\{1 \ldots i\}$ with max. allowed time w.

if $n \notin O_n$, then $OPT(n, w) = OPT(n-1, w)$

if $n \in O_n$, then $OPT(n, w) = w_n + OPT(n-1, w - w_n)$

Recurrence Formula 1:

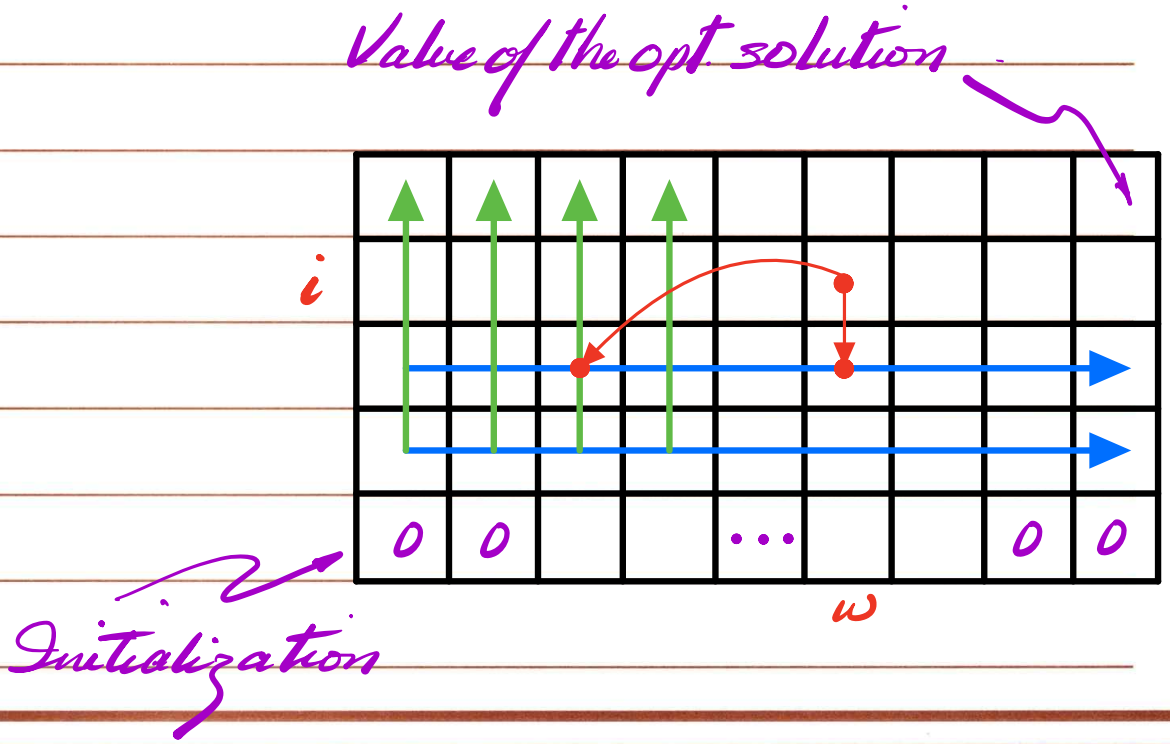
If $w < w_i$, then

$$OPT(i, w) = OPT(i-1, w)$$

else

$$OPT(i, w) = \text{Max}(OPT(i-1, w), w_i + OPT(i-1, w - w_i))$$

Bottom-up pass




```
Subset_Sum(n, w)
    array M[0, w] = 0 for each w = 0 to W

    for i = 1 to n
        for w = 0 to W
            Use recurrence formula 1
            to compute M[i, w]
        endfor
    endfor
    Return M[n, w]
```

Complexity Analysis

pseudo-polynomial complexity

Runtime Complexity: $O(nW)$

Is this considered an efficient solution? No!
W does not represent the size of the input, rather it is the numerical value of an input term.

What is the size of W? $\log_2 W$ bits.

Rewriting the function nW in terms of the size of the input:

$$nW = n \cdot 2^{\log_2 W}$$

Therefore, our complexity of $O(n \cdot 2^{\log_2 W})$ is exponential WRT the input size, and our solution is not considered efficient.

Polynomial Time

An algorithm runs in polynomial time if its running time is a polynomial in the length of the input

Pseudo-Polynomial Time

An algorithm runs in pseudo-polynomial time if its running time is a polynomial in the numeric value of the input.

How does the recurrence formula change for 0/1 Knapsack where each item i has weight $w(i)$ and value $v(i)$?

If $w < w_i$, then

$$OPT(i,w) = OPT(i-1,w)$$

else

$$OPT(i,w) = \text{Max}(OPT(i-1,w), v_i + OPT(i-1, w-w_i))$$

The only change