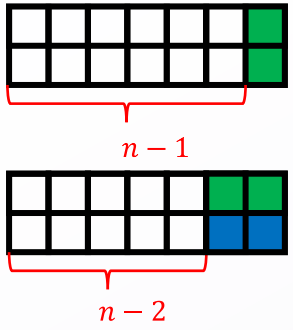
CSE 417 Autumn 2025

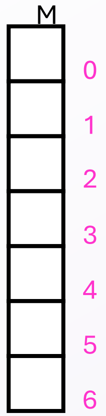
Lecture 15: Dynamic Programming – Adding Parameters

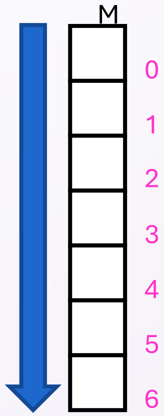
Nathan Brunelle

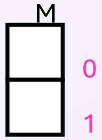
Four Steps to Dynamic Programming



Conclusion: a 1-dimensional memory of size n







1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?
3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a “Bottom-up” (iterative) algorithm
   Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space (Optional)
   Is it possible to reuse some memory locations?

Top-Down DP Idea

```
def myDPalgo(problem):
    if mem[problem] not blank:    // Check if we've solved this already
        return mem[problem]
    if baseCase(problem):    // Check if this is a base case
        solution = solve(problem)
        mem[problem] = solution    // Always save your solution before returning
        return solution
    for subproblem of problem:
        subsolutions.append(myDPalgo(subproblem)) // solve each subproblem
    solution = selectAndExtend(subsolutions) // Pick the subproblem to use
    mem[problem] = solution    // Always save your solution before returning
    return solution
```

Bottom-Up DP Idea

```
def myDPalgo(problem):
    for each baseCase:    // Identify which subproblems are base cases
        solution = solve(baseCase)
        mem[baseCase] = solution    // Save the solution for reuse
    for each subproblem in bottom-up order:
        // The order should be chosen so that every subsolution is
        // guaranteed to already be in memory when it's needed
        solution = selectAndExtend(subsolutions)
        mem[subproblem] = solution // Save the solution for reuse
    return mem[problem]
```

Log Cutting

Given a log of length n
A list (of length n) of prices P ($P[i]$ is the price of a cut of size i)
Find the best way to cut the log

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

1. Identify Recursive Structure

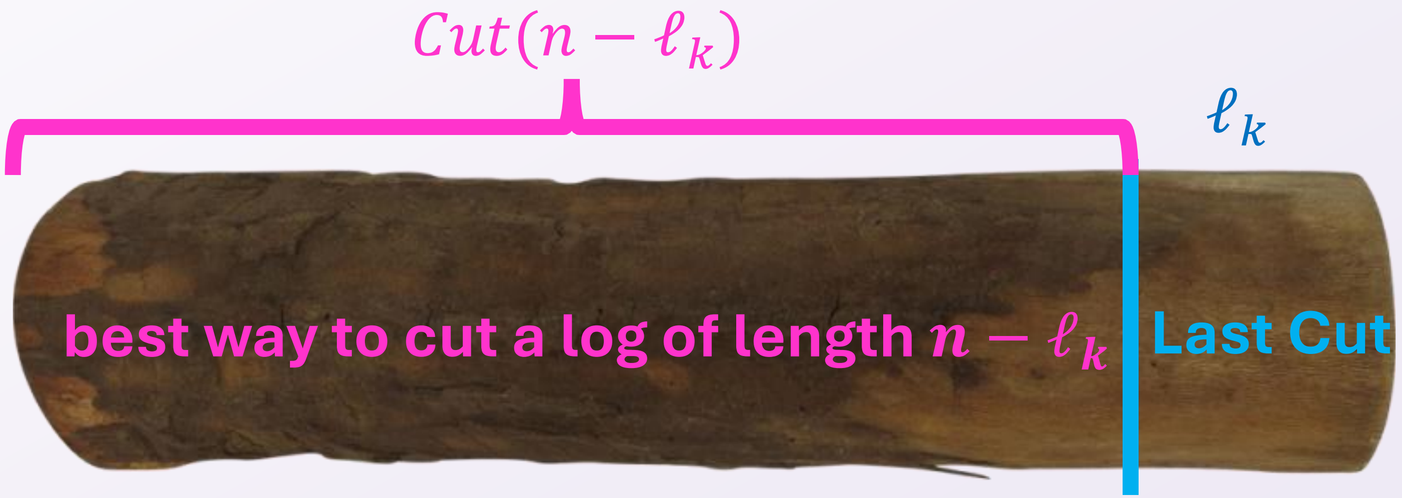
$P[i]$ = value of a cut of length i

$Cut(n)$ = value of best way to cut a log of length n

$$Cut(n) = \max \begin{cases} Cut(n-1) + P[1] \\ Cut(n-2) + P[2] \\ \dots \\ Cut(0) + P[n] \end{cases}$$

Base Case:

$Cut(0) = 0$

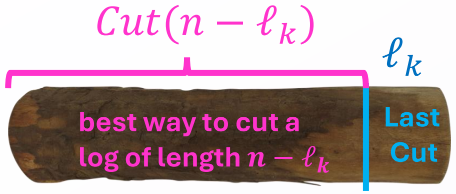


Log Cutting Top-Down

```
mem = [-1 for value in P]
def cut_log(n, P):
        if mem[n] < 0:
                return mem[n]
        if n == 0:
                solution = 0
                mem[n] = solution
                return solution
        subsolutions = []
        for i from 1 to n:
                subsolutions.append(cut_log(n-i,P)+P[i])
        solution = max(subsolutions)
        mem[n] = solution
        return solution
```

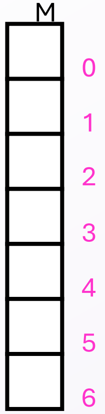
$$Cut(n) = \max \begin{cases} Cut(n-1) + P[1] \\ Cut(n-2) + P[2] \\ \ldots \\ Cut(0) + P[n] \end{cases}$$

DP’s Four Steps – Log Cutting – Step 3



1-dimensional memory of size n

$M[i]=$ best value achievable for an i-foot long log



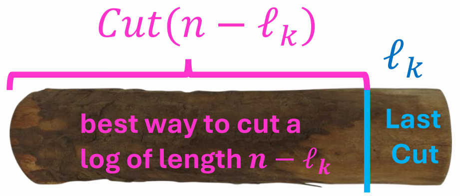
1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a “Bottom-up” (iterative) algorithm
   Without this step: a “Top-down” (recursive) algorithm

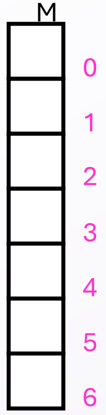
4. See if there’s a way to save space (Optional)
   Is it possible to reuse some memory locations?

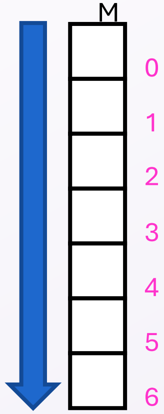
DP's Four Steps – Log Cutting – Step 4



1-dimensional memory of size n

$M[i]$ = best value achievable for an i-foot long log





1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a "Bottom-up" (iterative) algorithm
   Without this step: a "Top-down" (recursive) algorithm

4. See if there's a way to save space (Optional)
   Is it possible to reuse some memory locations?

Log Cutting Bottom-Up

```
def cutLog(n, P):
    mem = array of length n
    mem[0] = 0   // Solution for base case
    for (int i = 1; i <=n; i++): // For each subproblem I
        best = -1
        for(int length = 1; length <= i; length++): // For each choice of the last cut
            best = max(best, P[length] + mem[i-length]);
        mem[i] = best // Save the solution for reuse
    return mem[n]
```

DP Running Time

Bottom-Up DP:

- Done just as you would with any other iterative code
- Sum up the total work done across all iterations of the loops

Top-Down DP:

- Because of memorization, each subproblem is only solved once
- All subsequence uses for it will be a constant time look up
- To find running time: sum together the non-recursive work across all subproblems

Log Cutting Bottom-Up Running Time

```
def cutLog(n, P):
    mem = array of length n
    mem[0] = 0   // Solution for base case
    for (int i = 1; i <=n; i++): // For each subproblem I
        best = -1
        for(int length = 1; length <= i; length++): // For each choice of the last cut
            best = max(best, P[length] + mem[i-length]);
        mem[i] = best // Save the solution for reuse
    return mem[n]
```

Iterates n times

Iterates i times

O(1)

Overall: $O(n^2)$ time

Log Cutting Top-Down Running Time

```
mem = [-1 for value in P]
def cut_log(n, P):
        if mem[n] < 0:
                return mem[n]
        if n == 0:
                solution = 0
                mem[n] = solution
                return solution
        subsolutions = []
        for i from 1 to n:
                subsolutions.append(cut_log(n-i,P)+P[i])
        solution = max(subsolutions)
        mem[n] = solution
        return solution
```

n subproblems to solve

Each takes linear non-recursive work

Overall: $O(n^2)$ time

How to find the cuts?

This procedure told us the profit, but not the cuts themselves

Idea: remember the choice that you made, then backtrack

Log Cutting Bottom-Up Recording Choices

```
def cutLog(n, P):
        mem = array of length n
        choices = array of length n
        mem[0] = 0   // Solution for base case
        choices[0] = 0
        for (int i = 1; i <=n; i++): // For each subproblem I
                best = -1
                for(int length = 1; length <= i; length++): // For each choice of the last cut
                        if(best < P[length] + mem[i-length]):
                                best = P[length] + mem[i-length]
                                choices[i] = length
        mem[i] = best // Save the solution for reuse
        return mem[n], choices
```

Every time we find a better solution, remember which choice made it happen.

Log Cutting Top-Down recording Choices

```
mem = [-1 for value in P]
choices = [-1 for value in P]
def cutLog(n, P):
        if mem[n] < 0:
                return mem[n]
        if n == 0:
                mem[n] = 0
                choices[n] = 0
                return 0
        for(int length = 1; length <= i; length++):
                if(mem[n] < cutLog(n-i,P)+P[i]):
                        mem[n] = cutLog(n-i,P)+P[i]
                        choices[n] = length
        return mem[n]
```

Every time we find a better solution, remember which choice made it happen.

Remember the choice made

```
Initialize Memory C, Choices
Cut(n):
    C[0] = 0
    for i=1 to n:
        best = 0
        for j = 1 to i:
            if best < C[i-j] + P[j]:
                best = C[i-j] + P[j]
                Choices[i]=j
        C[i] = best
    return C[n]
```

Gives the size of the last cut

Reconstruct the Cuts

Backtrack through the choices

Choices:	0	1	1	2	4	3	4	1	2	4	3
Length:	0	1	2	3	4	5	6	7	8	9	10

Example to demo Choices[] only. Profit of 20 is not optimal!

Backtracking Pseudocode

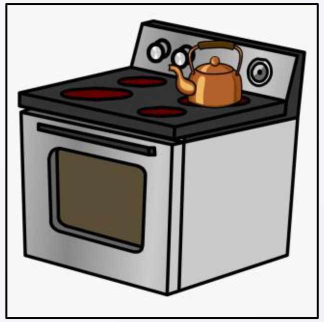
```
i = n // starting with the whole log
while i > 0: // until we've used the whole log
        print(choices[i]) // see what the last length was
        i = i – choices[i] // repeat for the subproblem
```

Our Example: Getting Optimal Solution

i	0	1	2	3	4	5	6	7	8	9	10
mem[i]	0	1	5	8	10	13	17	18	22	25	30
choice[i]	0	1	2	3	2	2	6	1	2	3	10

- If n were 5
  - Best score is 13
  - Cut at Choice[n]=2, then cut at Choice[n-Choice[n]]= Choice[5-2]= Choice[3]=3
- If n were 7
  - Best score is 18
  - Cut at 1, then cut at 6

Oven Allocation (aka subset sum)



Suppose we had an oven and a set of n items to bake

Each item $\mathrm{b}_i = (t_i, p_i)$ takes t_i minutes to bake, and can be sold for a profit of p_i dollars

We have M total minutes available to bake

What should we bake to maximize our profit?

Best solution: b_1, b_2, b_3

Uses 27 minutes

Earns $41

$$M = 30$$

$$b_0 = (23,20), b_1 = (10,15), b_2 = (12,18), b_3 = (5,8)$$

Four Steps – Oven Allocation Step 1

1. Formulate the answer with a recursive structure

   What are the options for the last choice?

   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.

   Figure out the possible values of all parameters in the recursive calls.

   How many subproblems (options for last choice) are there?

   What are the parameters needed to identify each?

   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)

   Want to guarantee that the necessary subproblem solutions are in memory when you need them.

   With this step: a “Bottom-up” (iterative) algorithm

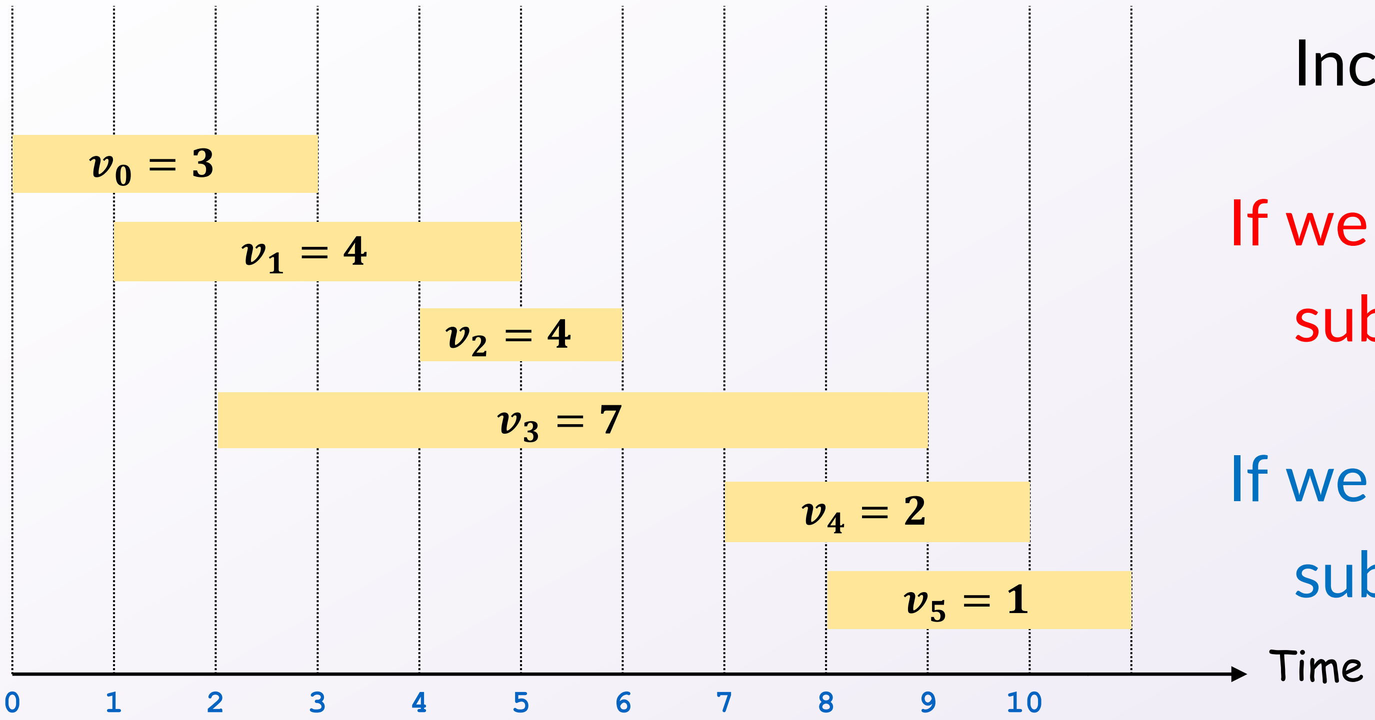
   Without this step: a “Top-down” (recursive) algorithm

4. See if there’s a way to save space (Optional)

   Is it possible to reuse some memory locations?

Comparison to Weighted Interval Scheduling

Oven allocation looks similar to weighted interval scheduling, except tasks do not have predefined start and end times



Choices:

Include/exclude the last event

If we exclude:

subproblem is second-to-last event

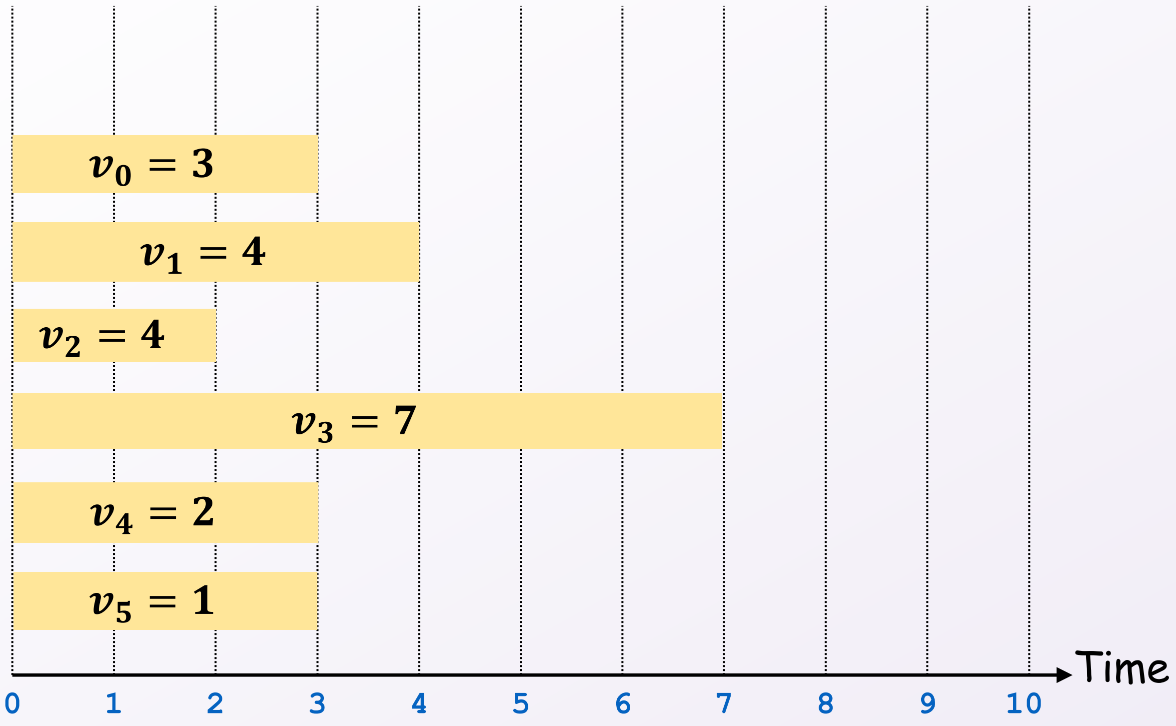
If we include:

subproblem is latest compatible event j

$$wis(i) = \max(wis(i-1), wis(j) + v_i)$$

Oven Allocation – First attempt

Let's try to adapt the weighted interval scheduling approach



Choices:

Include/exclude the last item

If we exclude:

subproblem is ??

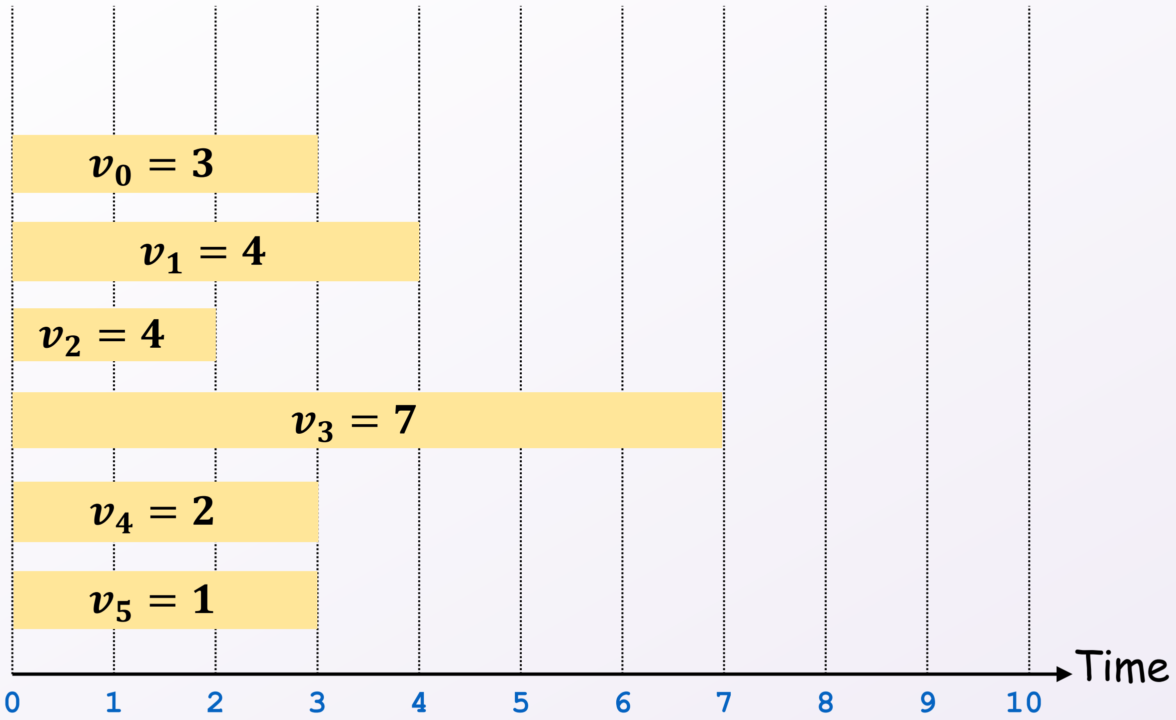
If we include:

subproblem is ??

Problem: selecting an item does not eliminate any others, it only reduces the amount of time

Oven Allocation – Recursive Structure (Almost)

Problems identified by 2 parameters! The “last” item, and the time remaining



Choices:

Include/exclude the last item

(For now, assume sufficient time for b_i)

If we exclude:

subproblem is defined by item $i-1$

AND by m

If we include:

subproblem is defined by item $i-1$

AND by $m - t_i$

$$oven(i, m) = \max(oven(i-1, m), oven(i-1, m-t_i) + p_i)$$

Oven Allocation – Recursive Structure (almost) Example

$$oven(5,10) = \max(oven(4,10), oven(4,7) + 1)$$

If we exclude:

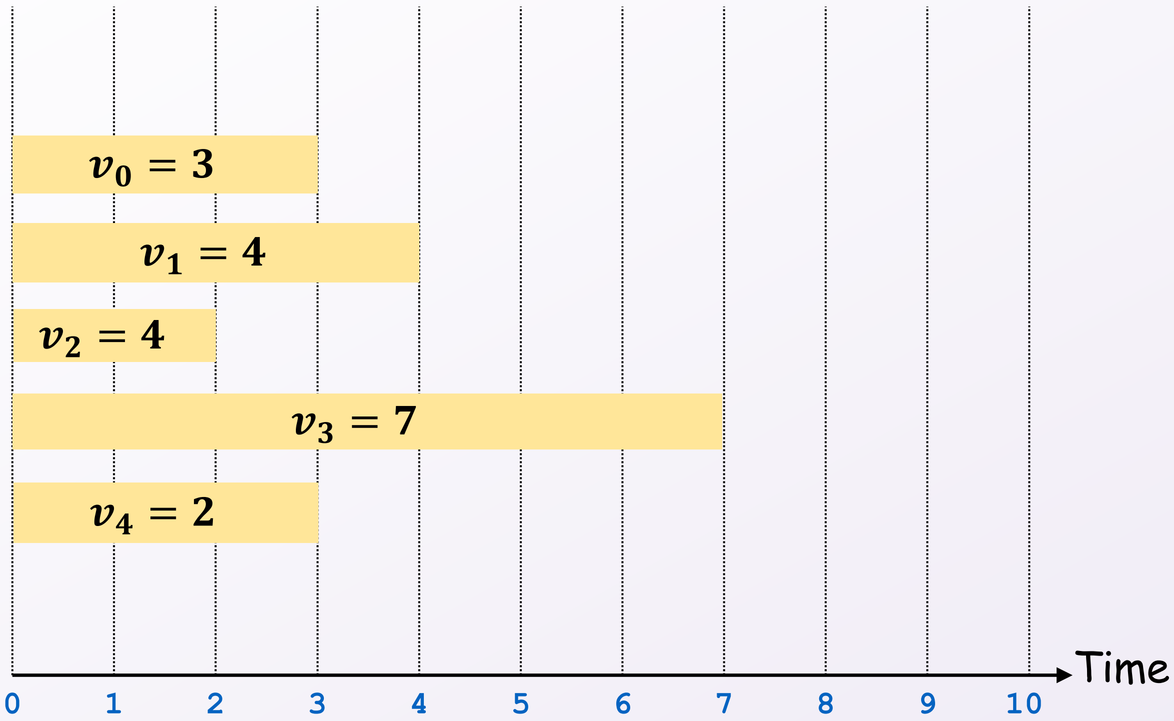
10 minutes left, \$0 earned

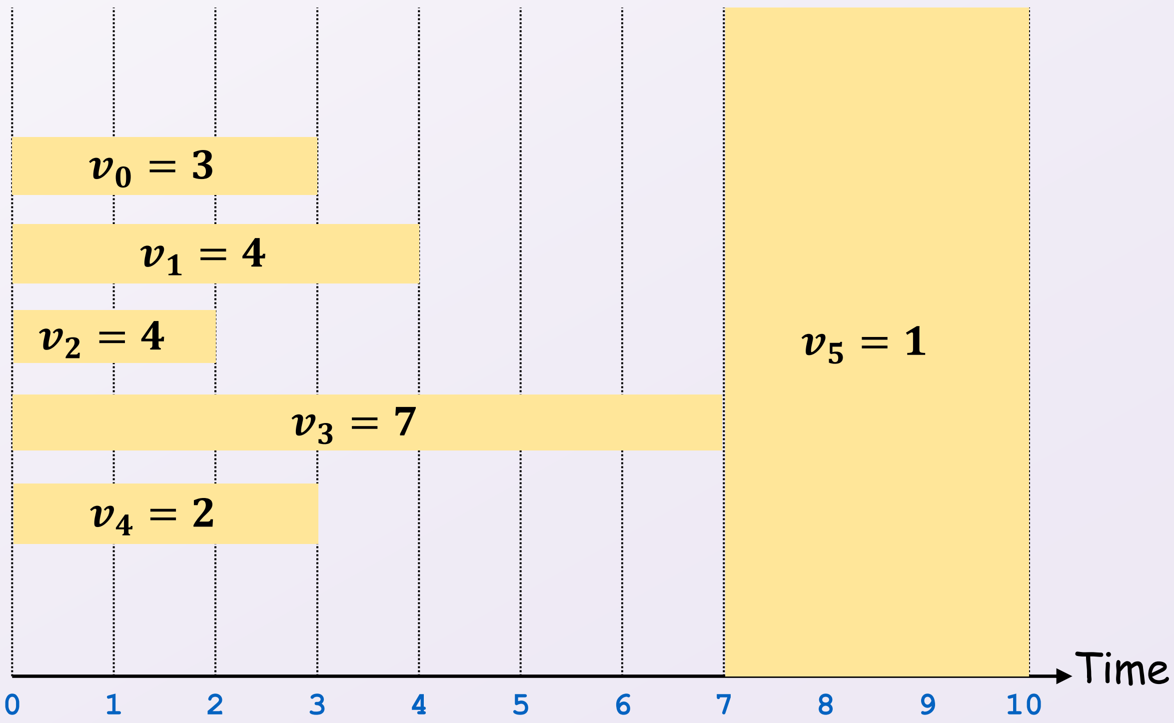
Both cases:

no longer considering item 5

If we include:

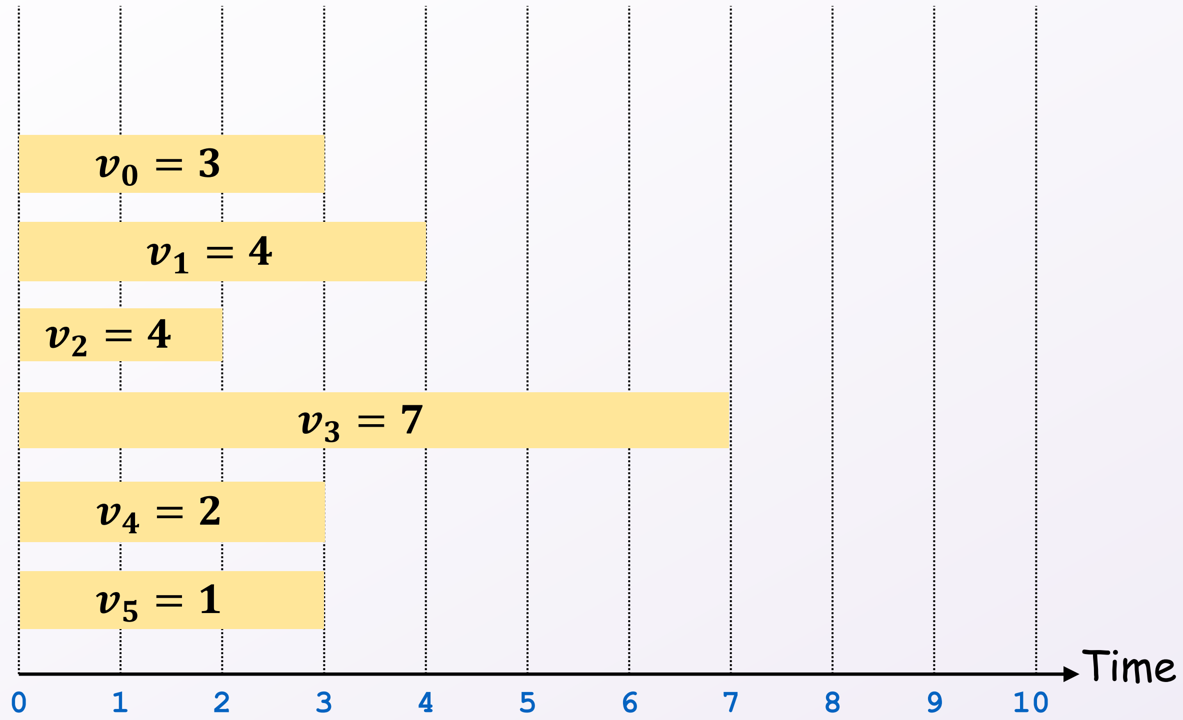
7 minutes left, \$1 earned





Oven Allocation – Full Recursive Structure

$$oven(i,m) = \begin{cases} \max(oven(i-1,m), oven(i-1,m-t_i)+p_i) & \text{if } t_i \le m \\ oven(i-1,m) & \text{otherwise} \end{cases}$$



Choices:

Include/exclude the last item

Including only allowable if $t_i \le m$

If we exclude:

subproblem is defined by item $i-1$

AND by m

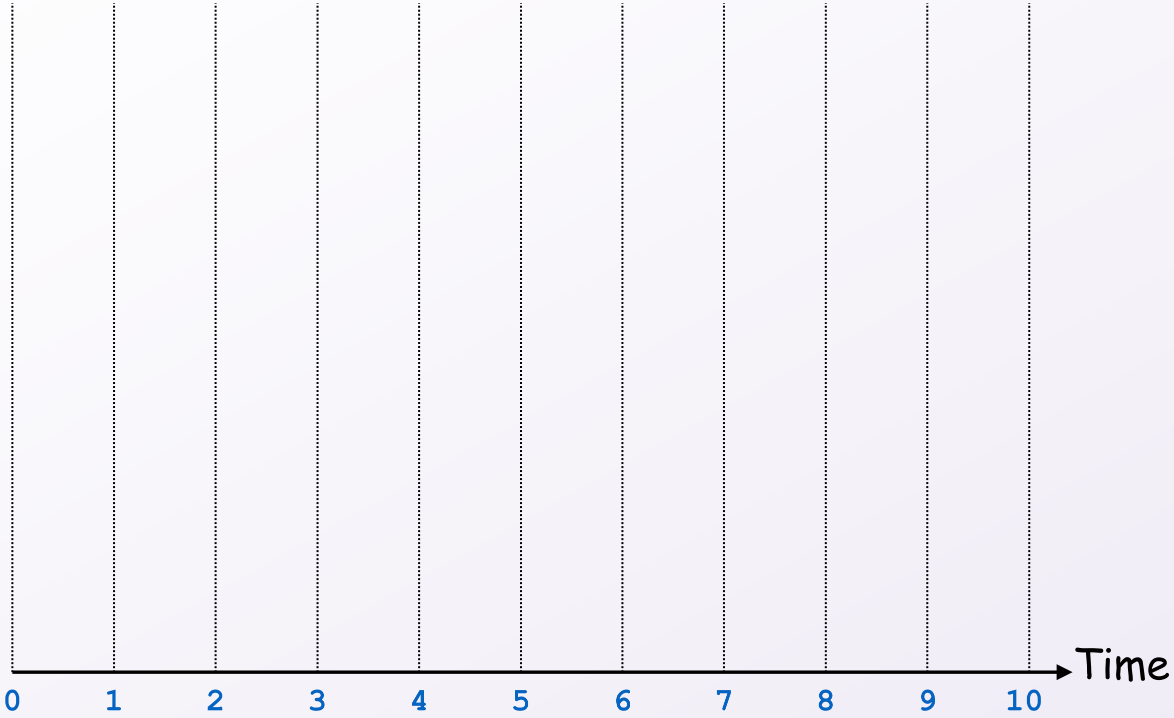
If $t_i \le m$ and we include:

subproblem is defined by item $i-1$

AND by $m - t_i$

Oven Allocation – Base Case

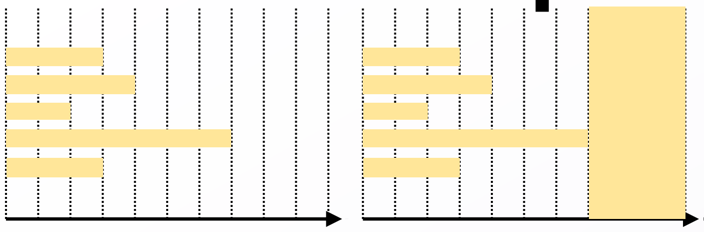
$$oven(i, m) = \begin{cases} \max(oven(i-1, m), oven(i-1, m-t_i) + p_i) & \text{if } t_i \leq m \\ oven(i-1, m) & \text{otherwise} \end{cases}$$



Base case: no items left

$oven(-1, m) = 0$

Four Steps – Oven Allocation Step 2



1. Formulate the answer with a recursive structure
   What are the options for the last choice?
   For each such option, what does the subproblem look like? How do we use it?

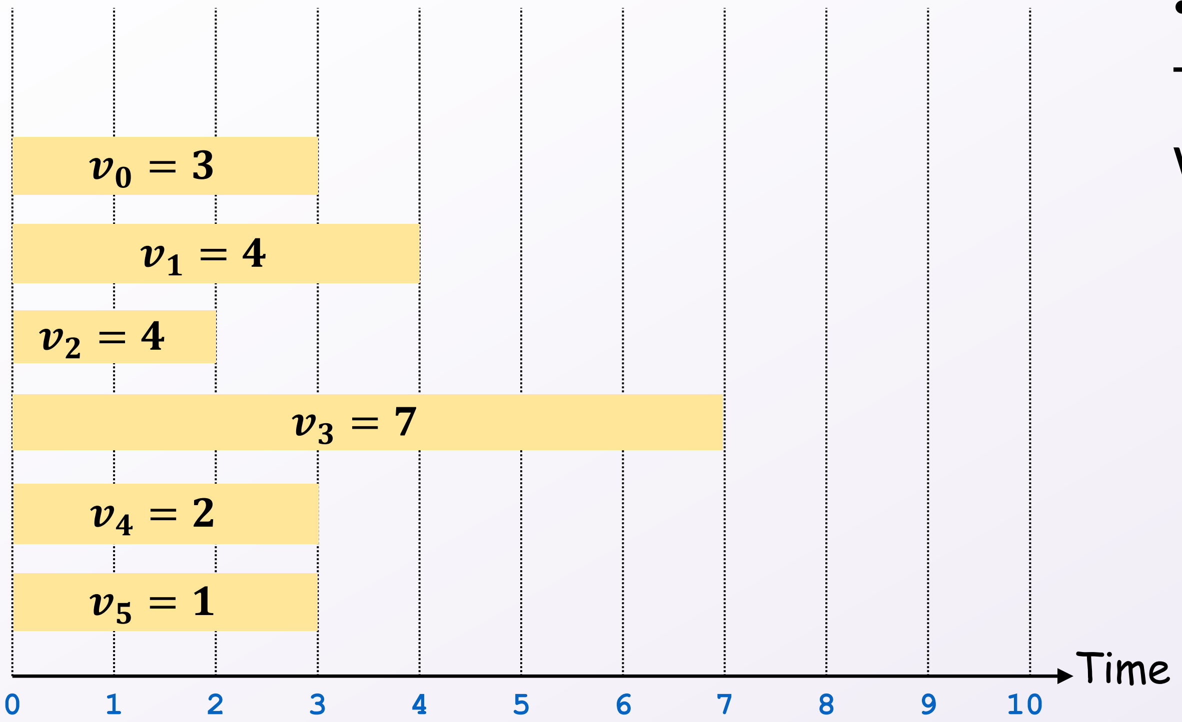
2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a “Bottom-up” (iterative) algorithm
   Without this step: a “Top-down” (recursive) algorithm

4. See if there’s a way to save space (Optional)
   Is it possible to reuse some memory locations?

Oven Allocation – Memory Structure

$$oven(i,m) = \begin{cases} \max(oven(i-1,m), oven(i-1,m-t_i)+p_i) & \text{if } t_i \le m \\ oven(i-1,m) & \text{otherwise} \end{cases}$$



Two parameters necessary to identify each subproblem:

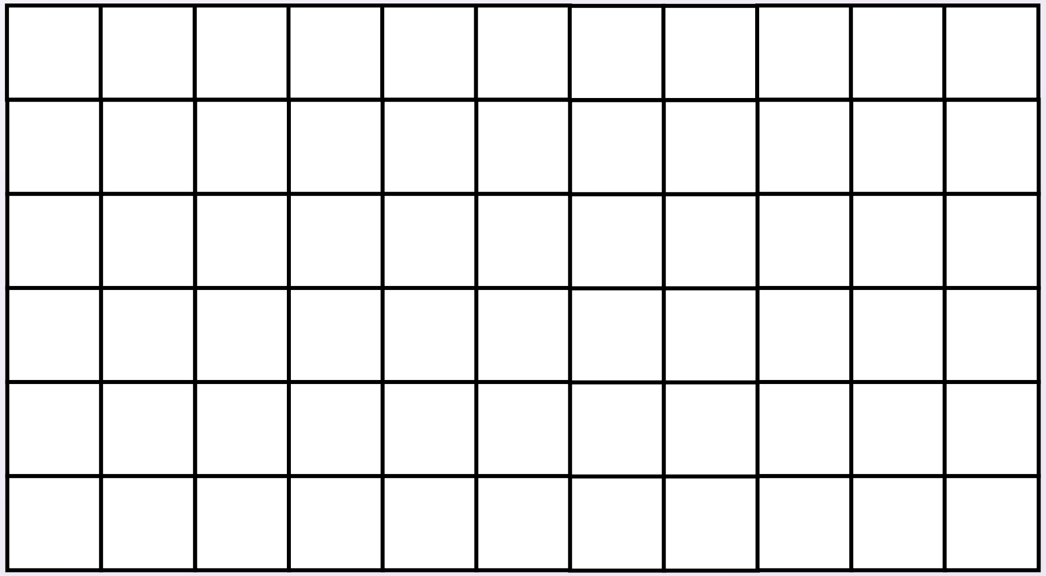
- The current item i
- The amount of time available m

There are n values of i and $M+1$ values of m (0 to M)

We will use a 2-dimensional array that is $n \times M$

$M+1$ columns

n rows



Oven Allocation Top-Down

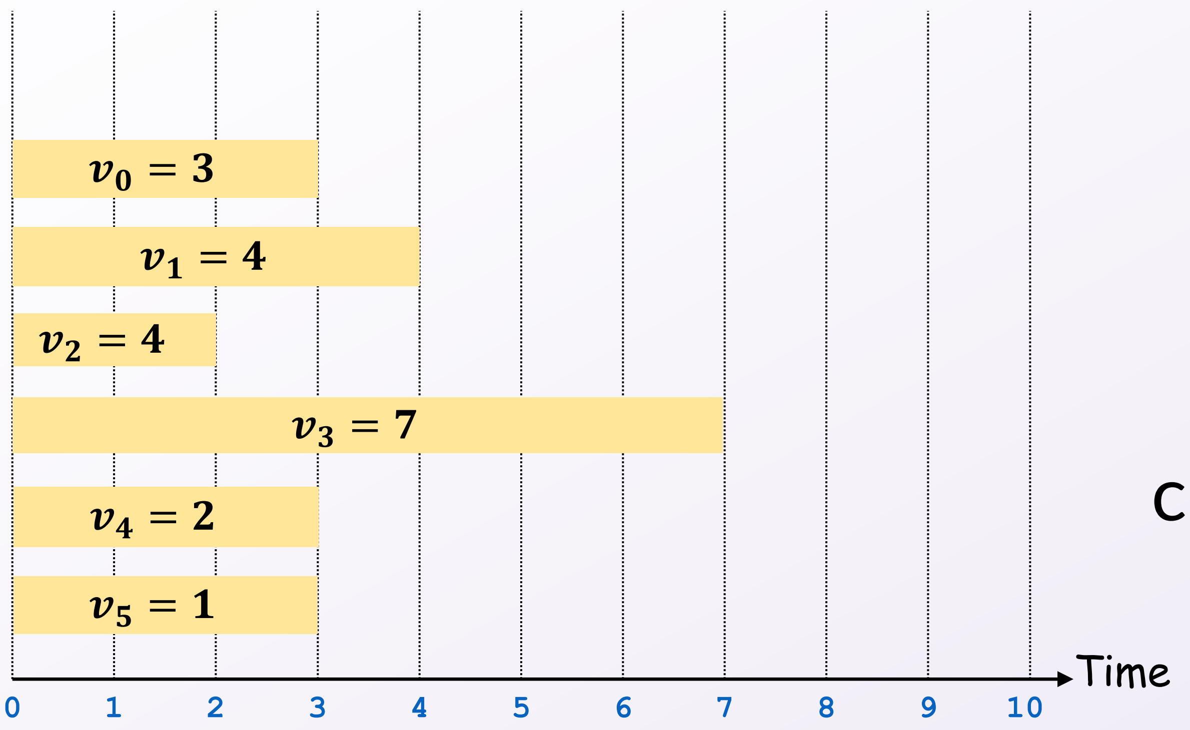
```
mem = an array of n rows and M columns full of -1s
def oven(i, m):
        if mem[i][m] > -1:
                return mem[i][m]
        if i == -1:
                return 0
        solution = oven(i-1,m)
        if(t_i ≤ m):
                solution = max(solution, oven(i-1,m-t_i)+p_i)
        mem[i][m] = solution
        return solution
```

Oven Allocation Top-Down with choices

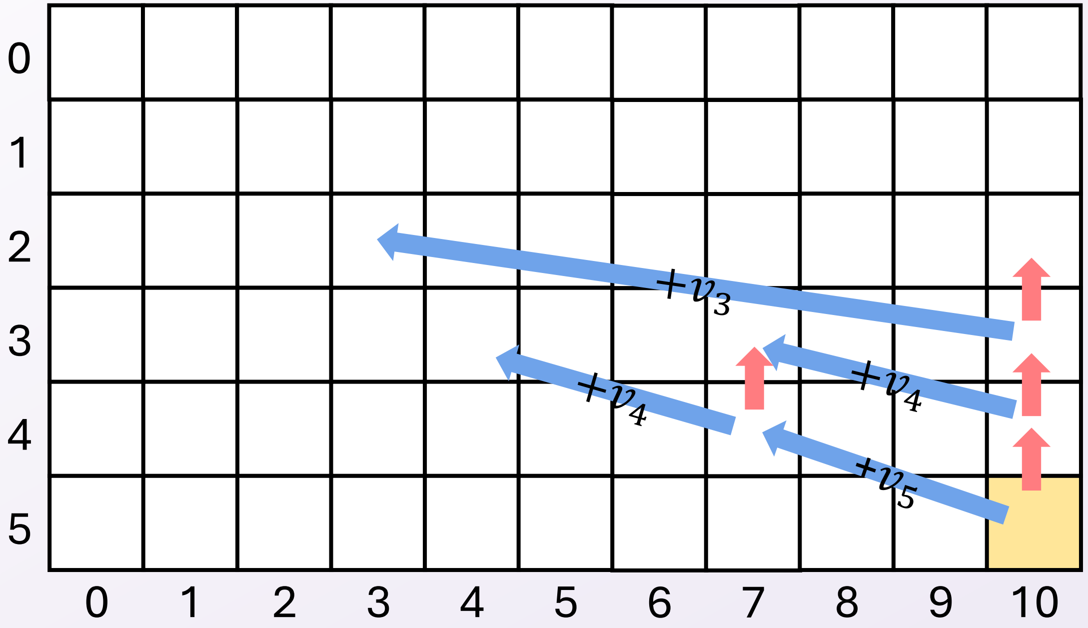
mem = an array of n rows and M columns full of -1s
choices = an array of n rows and M columns full of booleans
def **oven**(i, m):
    if mem[i][m] > -1:
        return mem[i][m]
    if i == -1:
        return 0
    solution = oven(i-1,m)
    choices[i][m] = False
    if($t_i \leq$ m and solution< oven(i-1,m-t_i)+p_i):
        solution = oven(i-1,m-t_i)+p_i
        choices[i][m] = True
    mem[i][m] = solution
    return solution

Oven Allocation Example

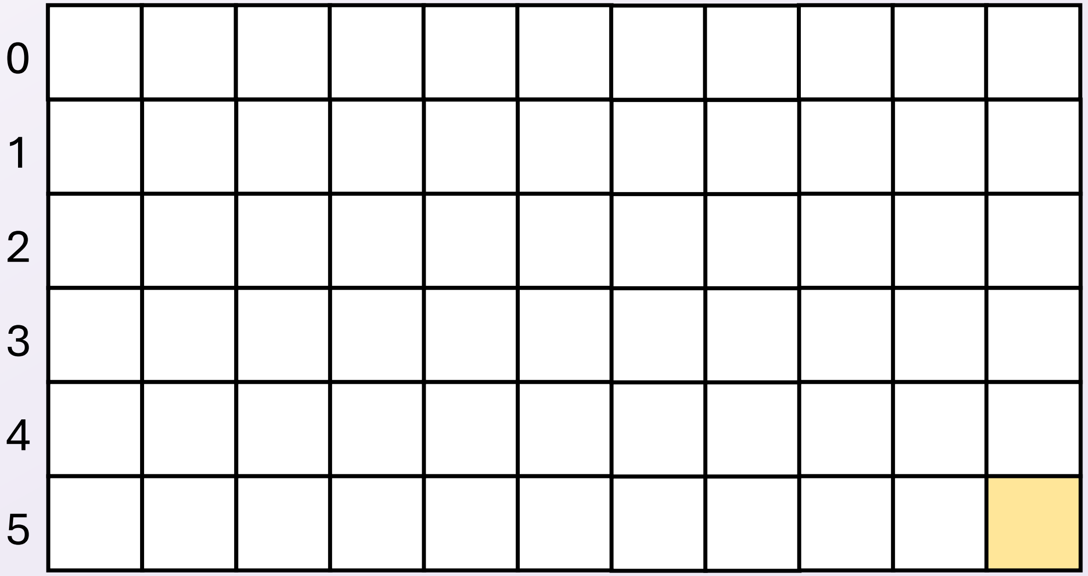
$$oven(i,m) = \begin{cases} \max(oven(i-1,m), oven(i-1,m-t_i)+p_i) & \text{if } t_i \le m \\ oven(i-1,m) & \text{otherwise} \end{cases}$$



mem

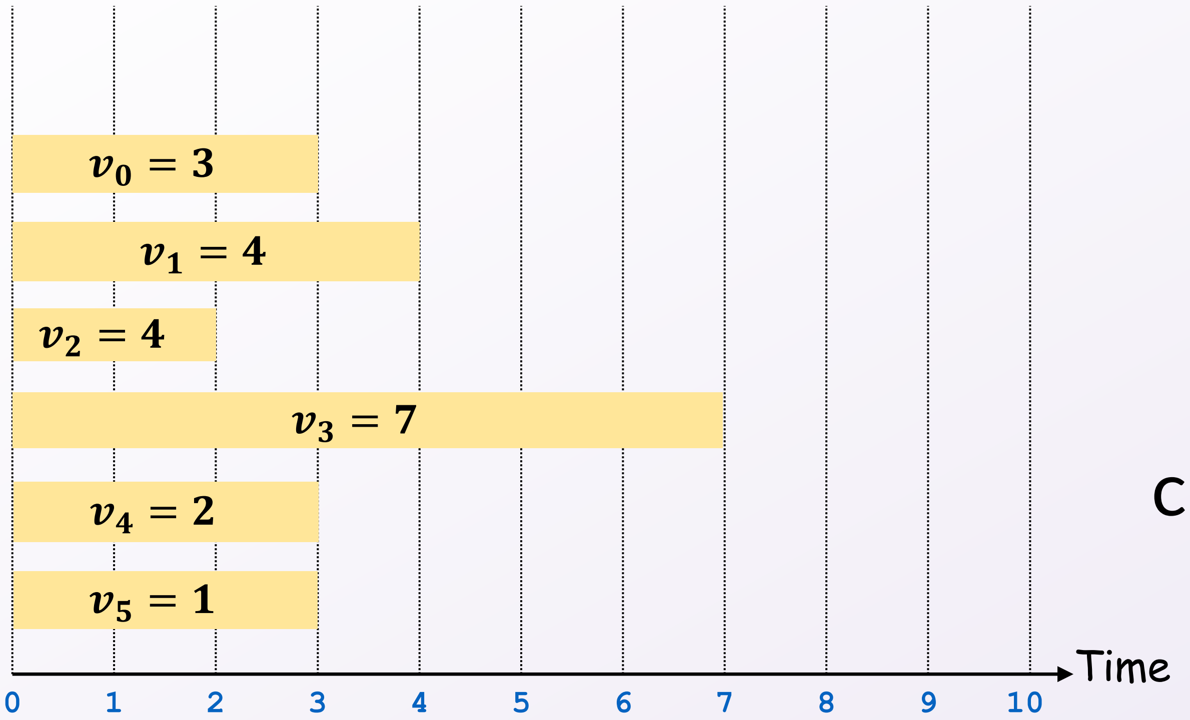


choices



Oven Allocation Example - complete

$$oven(i,m) = \begin{cases} \max(oven(i-1,m), oven(i-1,m-t_i)+p_i) & \text{if } t_i \le m \\ oven(i-1,m) & \text{otherwise} \end{cases}$$



mem

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

choices

	0	1	2	3	4	5	6	7	8	9	10
0	F	F	F	T	T	T	T	T	T		T
1	F	F	F	F	T	T		T	T		T
2	F			T	F			T			T
3					F			F			F
4								F			F
5											F

Using Choices

Def findChoices(choices, n, m):

    items = {}

    time = m

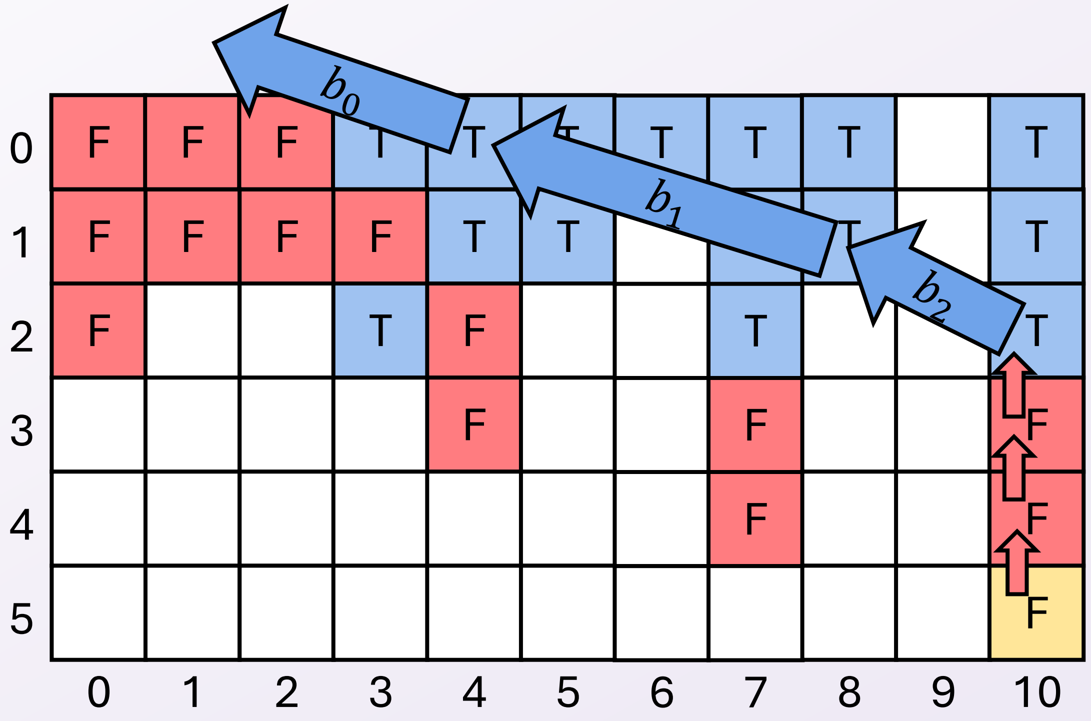
    for(item = n; item >= 0; item--) :

        if (choices[item][time]):

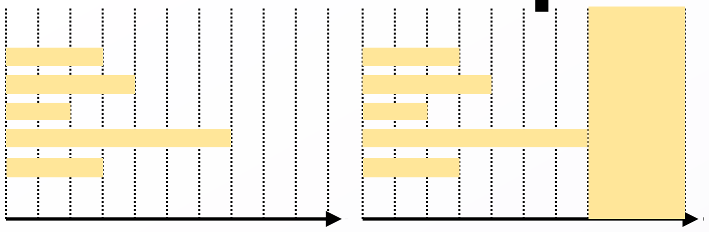
            items.add(item)

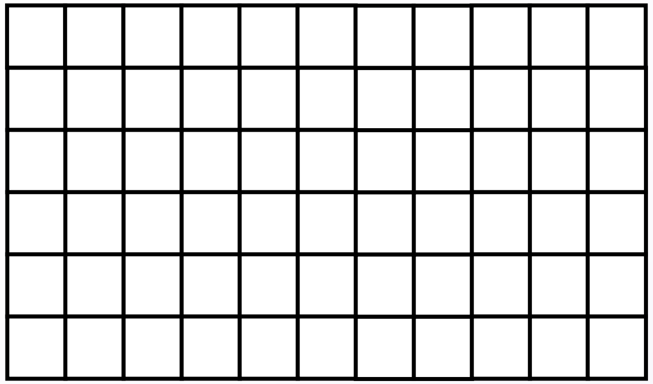
            time -= t_{item}

    return items



Four Steps – Oven Allocation Step 3





1. Formulate the answer with a recursive structure

   What are the options for the last choice?

   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.

   Figure out the possible values of all parameters in the recursive calls.

   How many subproblems (options for last choice) are there?

   What are the parameters needed to identify each?

   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)

   Want to guarantee that the necessary subproblem solutions are in memory when you need them.

   With this step: a “Bottom-up” (iterative) algorithm

   Without this step: a “Top-down” (recursive) algorithm

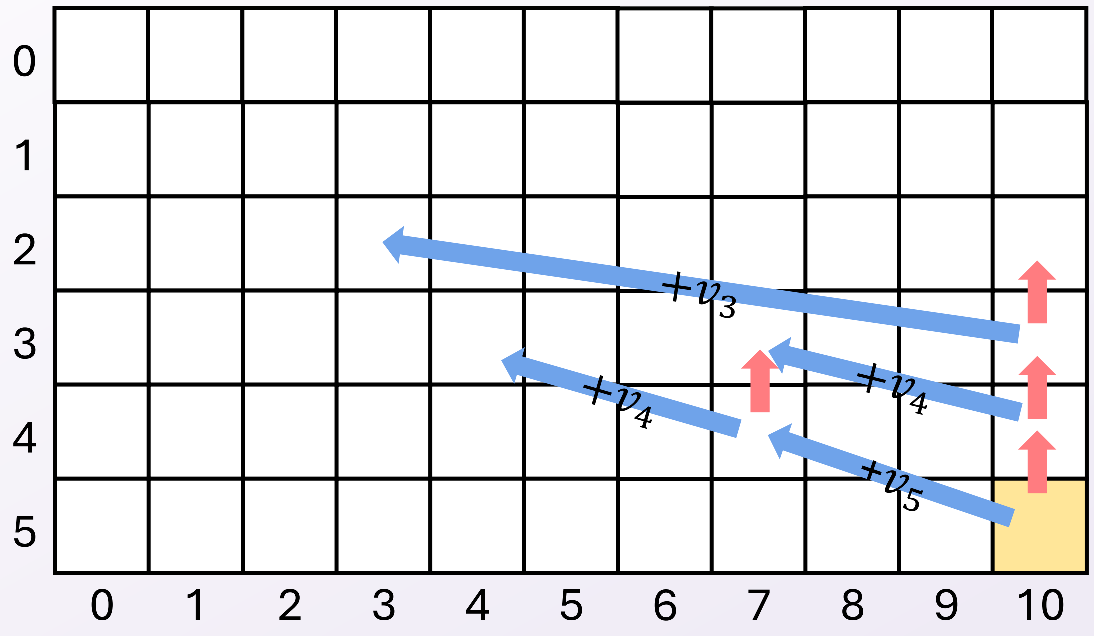
4. See if there’s a way to save space (Optional)

   Is it possible to reuse some memory locations?

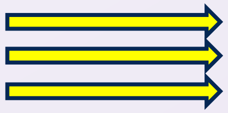
Selecting an order

$$oven(i,m) = \begin{cases} \max(oven(i-1,m), oven(i-1,m-t_i)+p_i) & \text{if } t_i \le m \\ oven(i-1,m) & \text{otherwise} \end{cases}$$

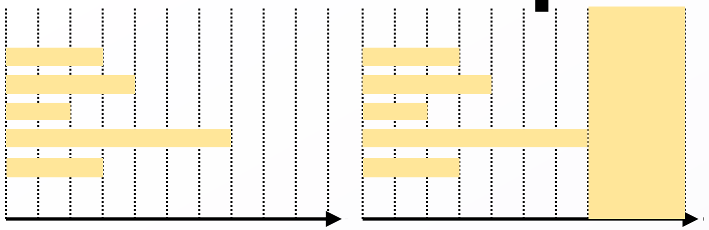
mem

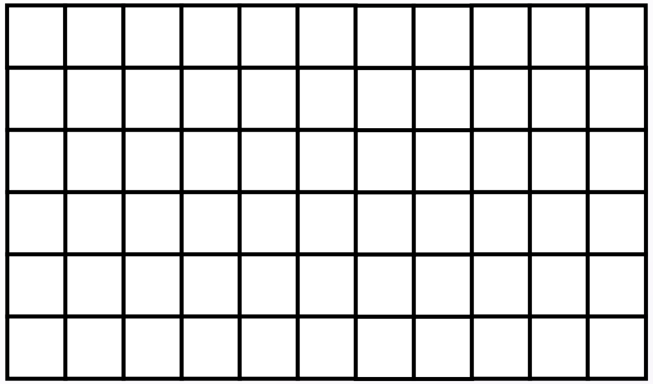


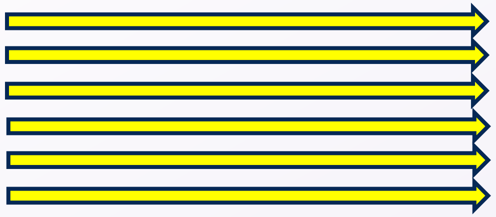
Each subproblem needs only cells in the row above it, and to its left.

Sufficient to fill in top to bottom, left to right 

Four Steps – Oven Allocation Step 4







1. Formulate the answer with a recursive structure

   What are the options for the last choice?

   For each such option, what does the subproblem look like? How do we use it?

2. Choose a memory structure.

   Figure out the possible values of all parameters in the recursive calls.

   How many subproblems (options for last choice) are there?

   What are the parameters needed to identify each?

   How many different values could there be per parameter?

3. Specify an order of evaluation. (Optional)

   Want to guarantee that the necessary subproblem solutions are in memory when you need them.

   With this step: a “Bottom-up” (iterative) algorithm

   Without this step: a “Top-down” (recursive) algorithm

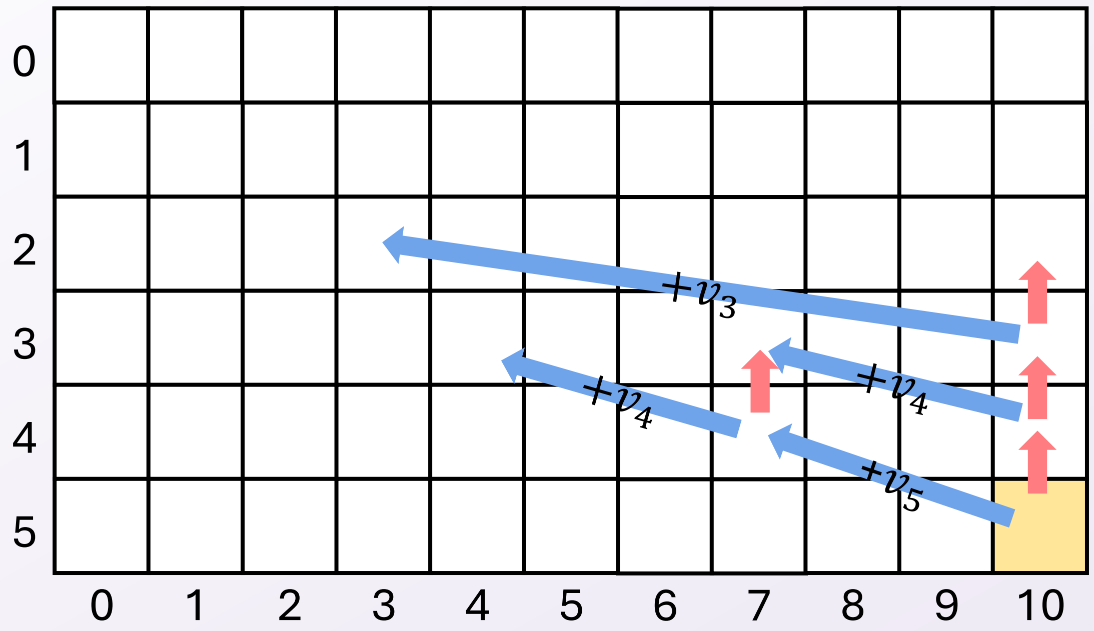
4. See if there’s a way to save space (Optional)

   Is it possible to reuse some memory locations?

Can we save space?

$$oven(i,m) = \begin{cases} \max(oven(i-1,m), oven(i-1,m-t_i)+p_i) & \text{if } t_i \le m \\ oven(i-1,m) & \text{otherwise} \end{cases}$$

mem



Each subproblem needs only cells in *the row above it*

Two rows are enough: the current one, and the one with subproblem solutions

Final reminders

HW3 resubmissions due Wednesday @ 11:59pm.

HW4 due Wednesday @ 11:59pm.

I have OH now-12:30pm:

- Meet at front of classroom, we'll walk over together
- CSE (Allen) 434 if you're coming later

Glenn has online OH 12–1pm