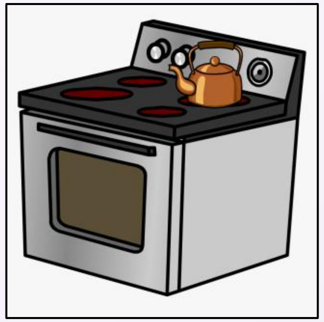
CSE 417 Autumn 2025

Lecture 16: Dynamic Programming – Sequences

Nathan Brunelle

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

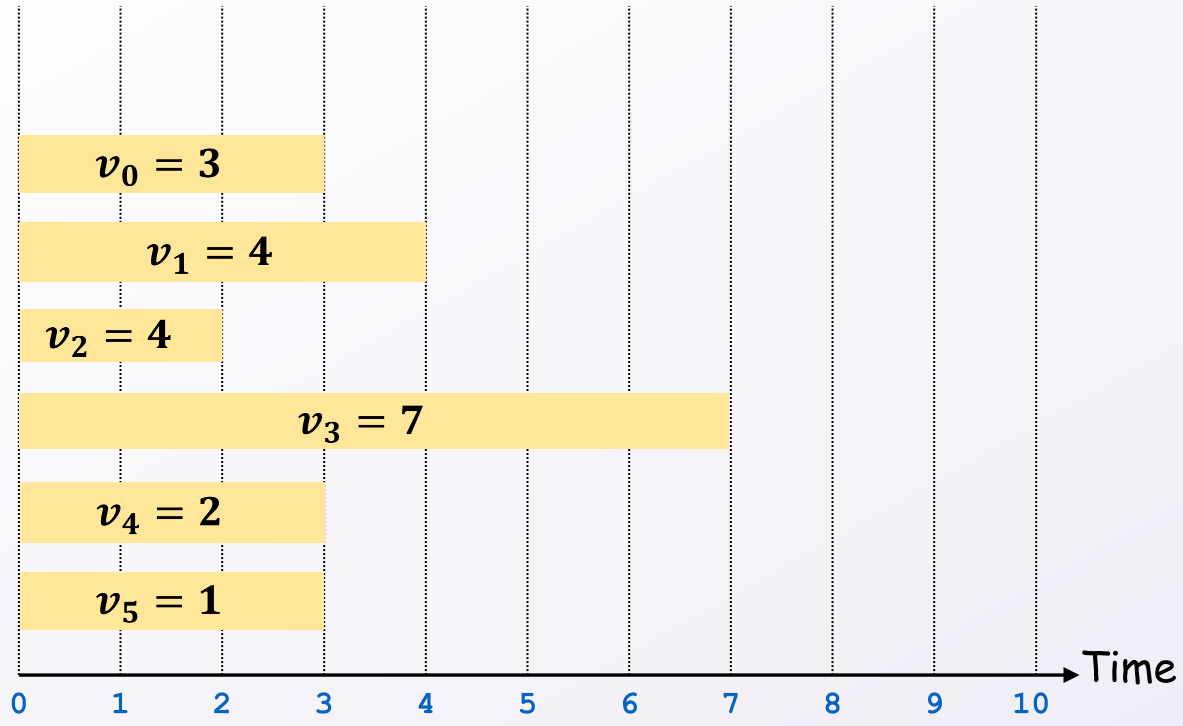
Earns $41

$$M = 30$$

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

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

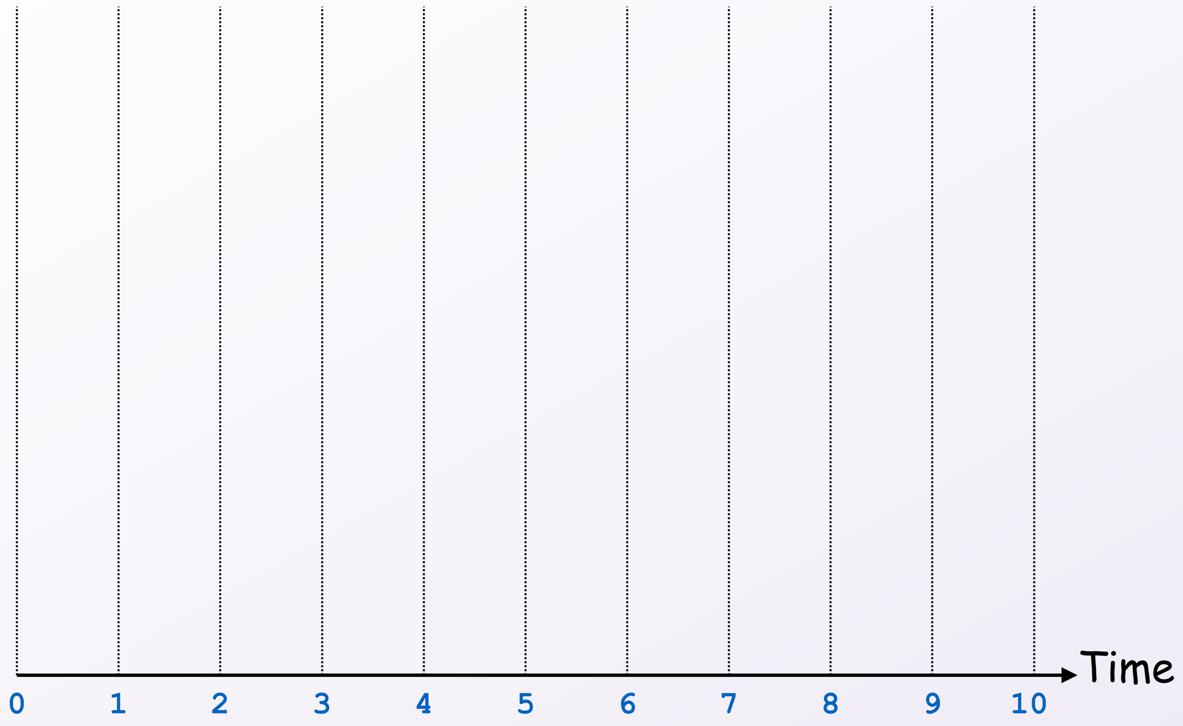
If $t_i \le m$ and we include:

subproblem is defined by item $i-1$

AND by $m - t_i$

Oven Allocation – Base Case

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

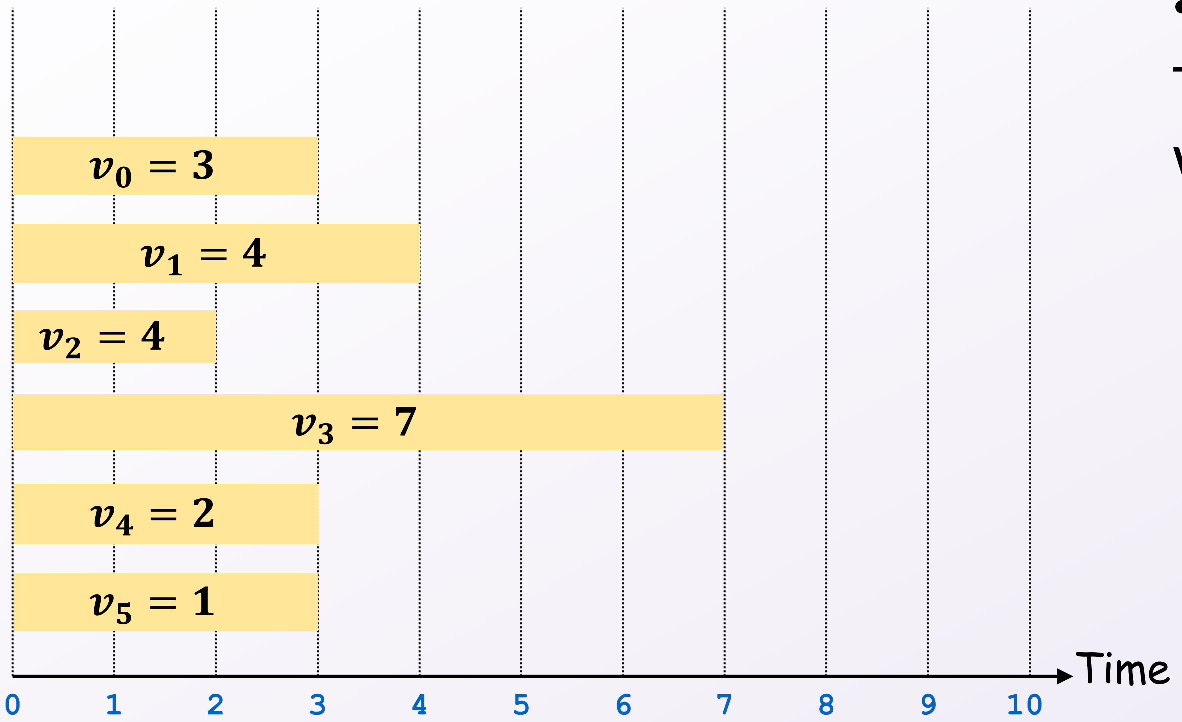


Base case: no items left

$oven(-1,m) = 0$

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:

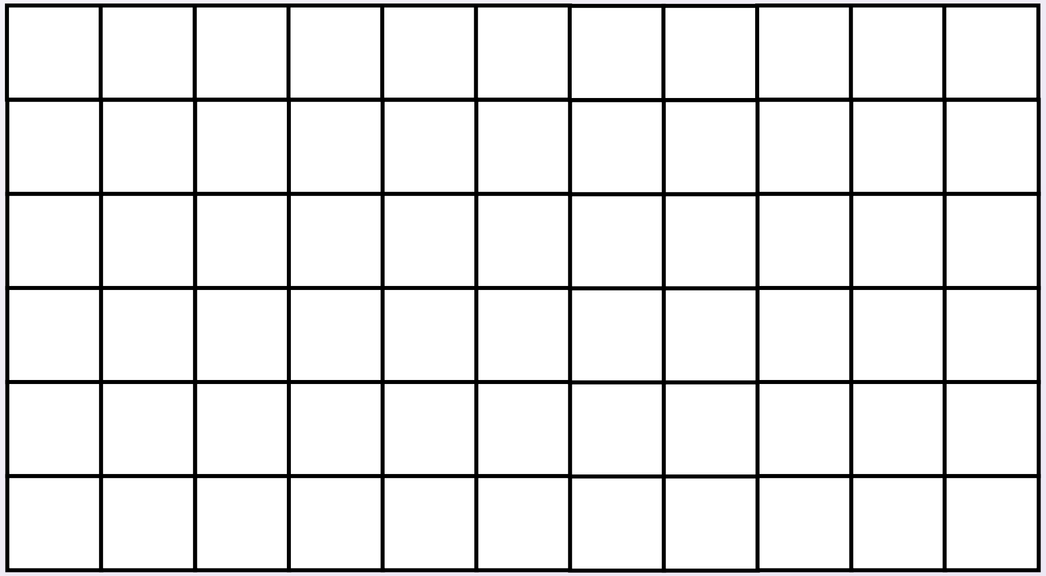
- The current item i
- The amount of time available m

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 $M + 1$ 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
```

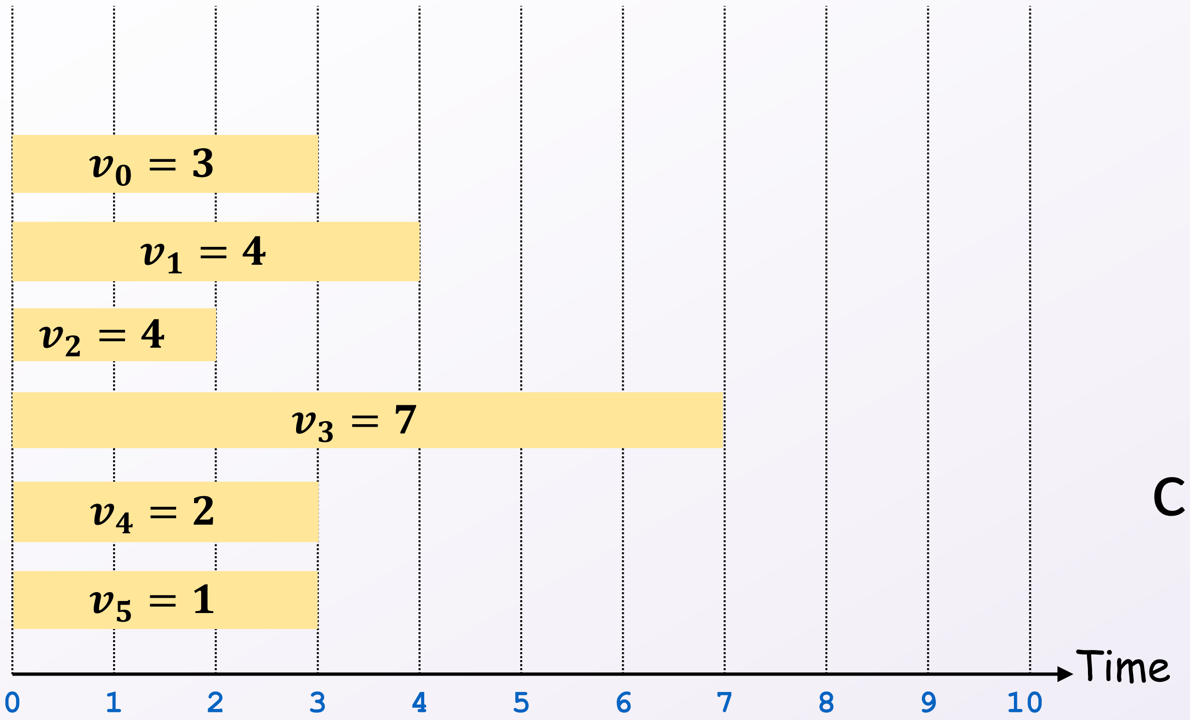
Oven Allocation Top-Down with choices

mem = an array of n rows and $M + 1$ columns full of -1s
choices = an array of n rows and $M + 1$ columns full of booleans
def **oven**(i, m):

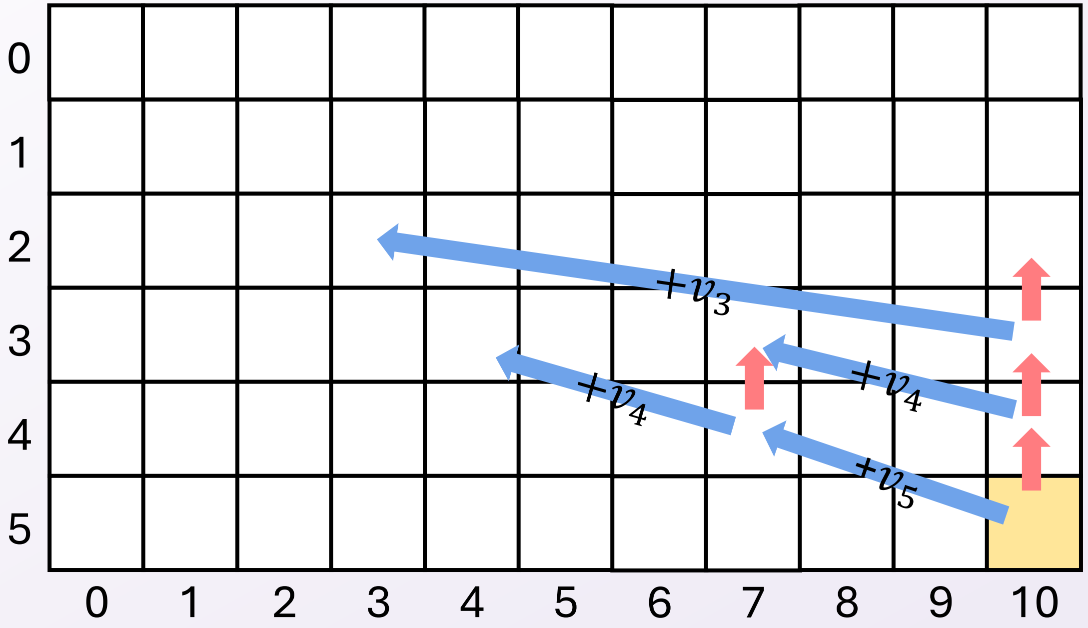
```
    if mem[i][m] > -1:
        return mem[i][m]
    if i == -1:
        return 0
    solution = oven(i-1,m)
    choices[i][m] = False
    if(t_i ≤ m and solution< oven(i-1,m-t_i)+p_i):
        solution = oven(i-1,m-t_i)+p_i
        choices[i][m] = True
    mem[i][m] = solution
    return solution
```

Oven Allocation Example

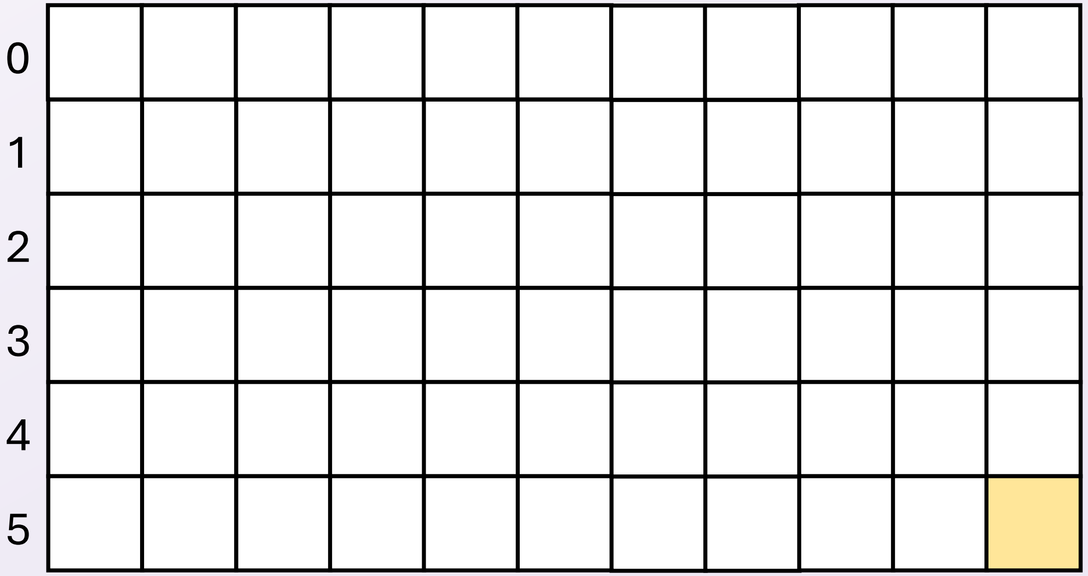
$$oven(i,m) = \begin{cases} \max(oven(i-1,m), oven(i-1,m-t_i)+p_i) & \text{if } t_i \le m \\ oven(i-1,m) & \text{otherwise} \end{cases}$$



mem

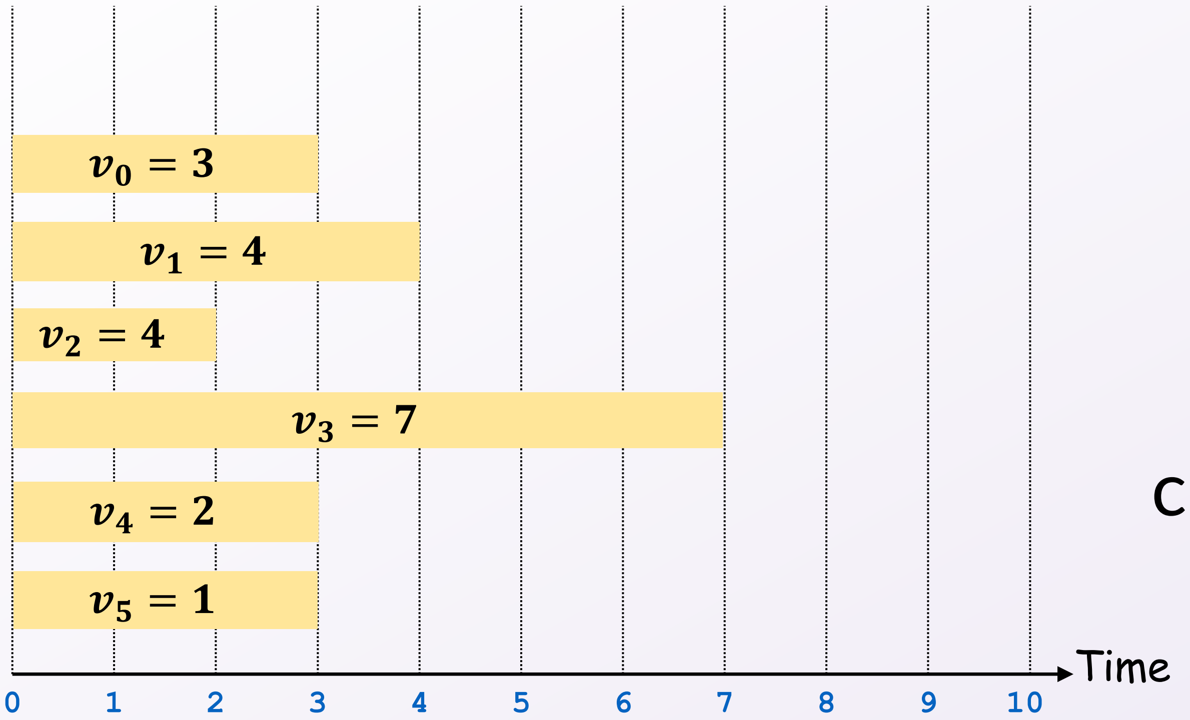


choices



Oven Allocation Example - complete

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



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

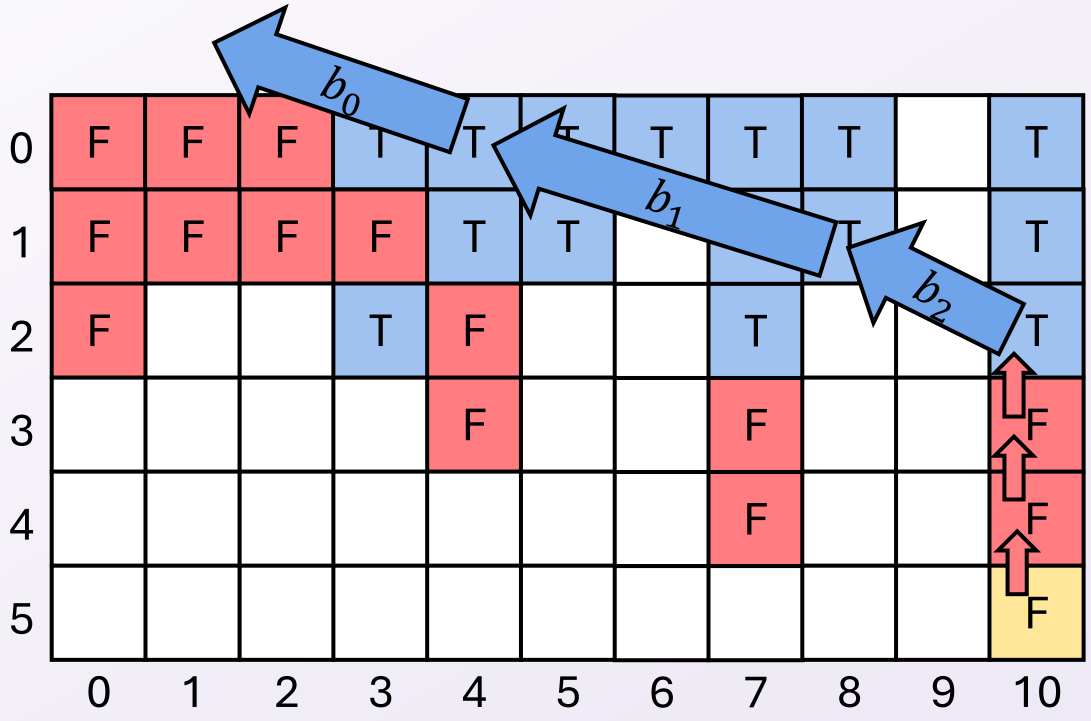
for(item = n; item >= 0; item--) :

if (choices[item][time]):

items.add(item)

time -= t_{item}

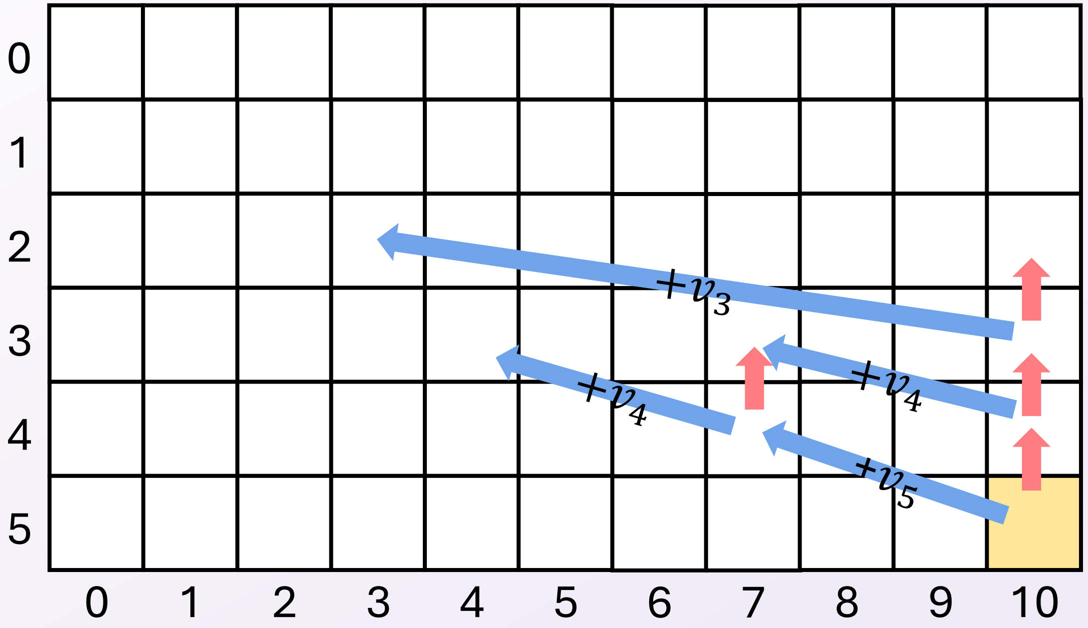
return items



Selecting an order

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

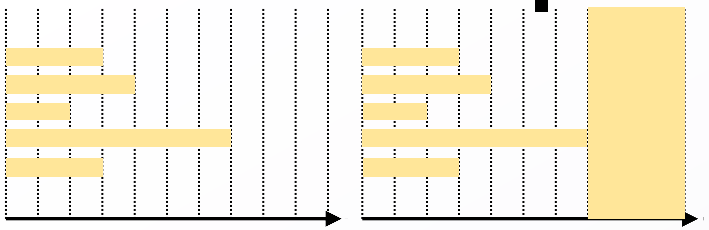
mem



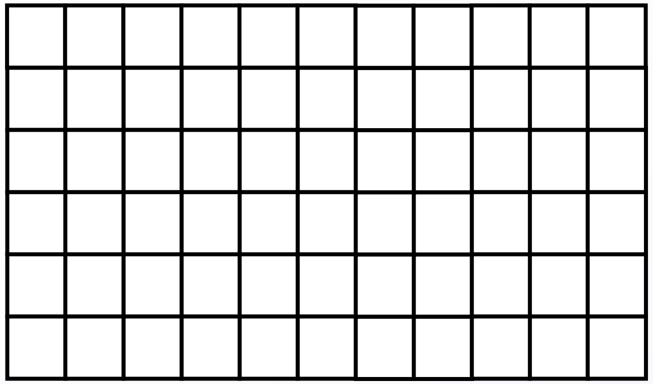
Each subproblem needs only cells in the row above it, and to its left.

Sufficient to fill in top to bottom, left to right

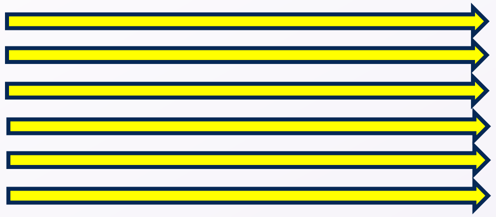
Four Steps – Oven Allocation Step 4



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?



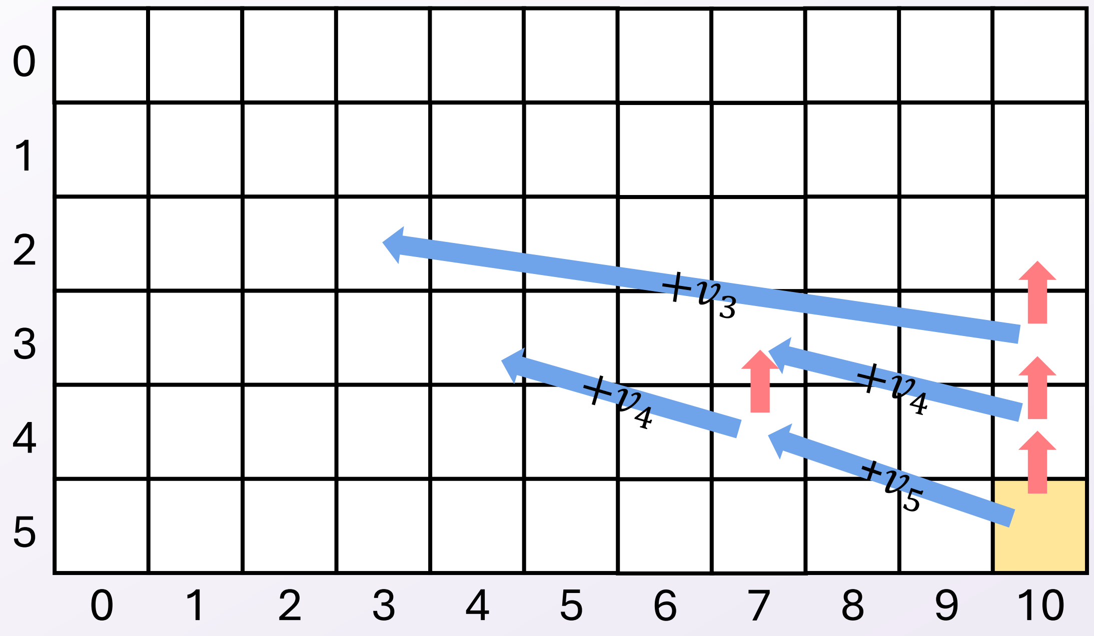
3. Specify an order of evaluation. (Optional)
   Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   With this step: a “Bottom-up” (iterative) algorithm
   Without this step: a “Top-down” (recursive) algorithm

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

String Similarity

How similar are two strings?

ocurrance

occurrence

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

Clearly a better matching

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

Maybe a better matching

- depends on cost of gaps vs mismatches

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

Edit Distance

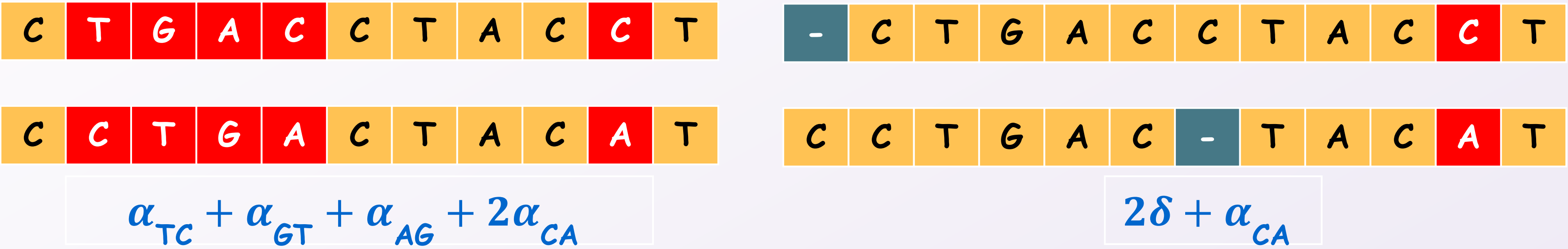
Applications:

Basis for Unix **`diff`**.

Speech recognition.

Computational biology.

autocorrect



Edit distance: [Levenshtein 1966, Needleman-Wunsch 1970]

Gap penalty δ; mismatch penalty α_{pq} if symbol p is replaced by symbol q.

Cost = gap penalties + mismatch penalties.

Sequence Alignment

Sequence Alignment:

Given: Two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$

Find: "Alignment" of X and Y of minimum edit cost.

Defn: An alignment M of X and Y is a set of ordered pairs x_i-y_j s.t. each symbol of X and Y occurs in at most one pair with no "crossing pairs".

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i:\, x_i \text{ unmatched}} \delta + \sum_{j:\, y_j \text{ unmatched}} \delta}_{\text{gap}}\ '.$$

Note: if $x_i = y_j$ then $\alpha_{x_i y_j} = 0$

Example:

CTACCG vs TACATG

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G
-	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

$M = \{x_2\text{-}y_1, x_3\text{-}y_2, x_4\text{-}y_3, x_5\text{-}y_4, x_6\text{-}y_6\}$

Edit Distance – Four Steps, 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?

Step 1: Identify Recursive Structure

x_1	x_2	x_3	x_4	x_5	x_6
C	T	A	C	C	G
T	A	C	A	T	G
y_1	y_2	y_3	y_4	y_5	y_6

Consider the last two indices x_i and y_j

Options for what to do with them:

Option 1:
Match them

x_1	x_2	x_3	x_4	x_5	x_6
C	T	A	C	C	G
T	A	C	A	T	G
y_1	y_2	y_3	y_4	y_5	y_6

We use up one index from x and y

Accrue a mismatch penalty

$$OPT(i-1, j-1) + \alpha_{x_i y_j}$$

Option 2:
Don't match x_i

	x_1	x_2	x_3	x_4	x_5	x_6
	C	T	A	C	C	G
T	A	C	A	T	G	-
y_1	y_2	y_3	y_4	y_5	y_6	

We use up one index from x only

Accrue a gap penalty

$$OPT(i-1, j) + \delta$$

Option 3:
Don't match y_i

x_1	x_2	x_3	x_4	x_5	x_6	
C	T	A	C	C	G	-
	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

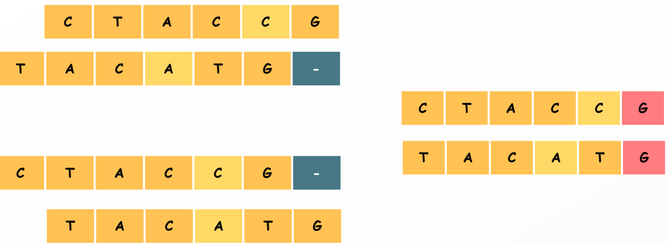
We use up one index from y only

Accrue a gap penalty

$$OPT(i, j-1) + \delta$$

$$OPT(i,j) = \begin{cases} j \cdot \delta & \text{if } i = 0 \\ i \cdot \delta & \text{if } j = 0 \\ \min \begin{cases} OPT(i-1, j-1) + \alpha_{x_i y_j} \\ OPT(i-1, j) + \delta \\ OPT(i, j-1) + \delta \end{cases} \end{cases}$$

Edit Distance – Four Steps, 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?

Step 2: Identify Memory Structure

x_1	x_2	x_3	x_4	x_5	x_6
C	T	A	C	C	G

T	A	C	A	T	G
y_1	y_2	y_3	y_4	y_5	y_6

$$OPT(i,j) = \begin{cases} j \cdot \delta & \text{if } i = 0 \\ i \cdot \delta & \text{if } j = 0 \\ \min \begin{cases} OPT(i-1, j-1) + \alpha_{x_i y_j} \\ OPT(i-1, j) + \delta \\ OPT(i, j-1) + \delta \end{cases} \end{cases}$$

How many parameters?

2

What does each represent?

The number of items in each sequence

How many different values?

Length of sequence x for i

Length of sequence y for j

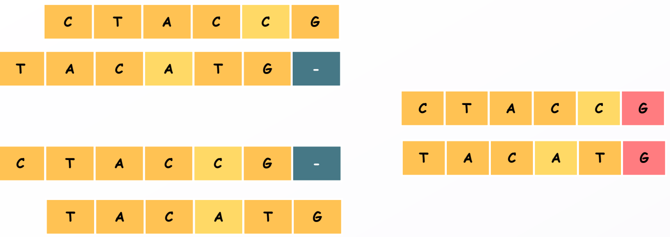
$n \cdot m$ overall

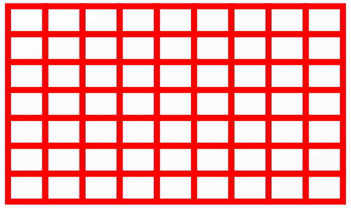
		y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
x_1									
x_2									
x_3									
x_4									
x_5									
x_6									

Top-Down Sequence Alignment

```
align(i, j):
    if OPT[i][j] not blank:    // Check if we've solved this already
        return OPT[i][j]
    if i · j == 0:    // Check if this is a base case
        solution = (i + j) · δ
        OPT[i][j] = solution    // Always save your solution before returning
        return solution
    match = align(i − 1, j − 1) // solve each subproblem
    gapx = align(i − 1,j) // solve each subproblem
    gapy = align(j, i − 1) // solve each subproblem
    solution = min(match + α_{x_i y_j}, gapx + δ, gapy+ δ) // Pick the subproblem to use
     OPT[i][j] = solution    // Always save your solution before returning
    return solution
```

Edit Distance – Four Steps, 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?

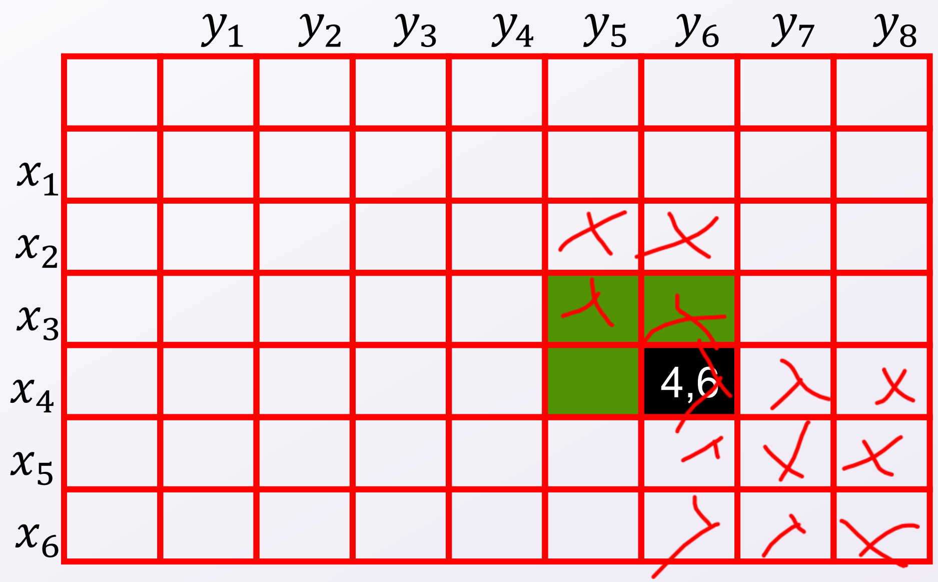
2. Choose a memory structure.
   Figure out the possible values of all parameters in the recursive calls.
   How many subproblems (options for last choice) are there?
   What are the parameters needed to identify each?
   How many different values could there be per parameter?

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

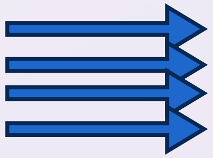
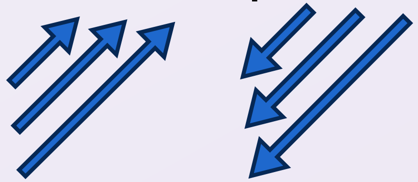
4. See if there's a way to save space (Optional)
   Is it possible to reuse some memory locations?

Step 3: Identify Order of Evaluation

$$OPT(i,j) = \begin{cases} j \cdot \delta & \text{if } i = 0 \\ i \cdot \delta & \text{if } j = 0 \\ \min \begin{cases} OPT(i-1,j-1) + \alpha_{x_i y_j} \\ OPT(i-1,j) + \delta \\ OPT(i,j-1) + \delta \end{cases} \end{cases}$$



Any of these orders will work:

- Top-to-bottom, then left-to-right 
- Left-to-right, then top-to-bottom 
- Diagonally 

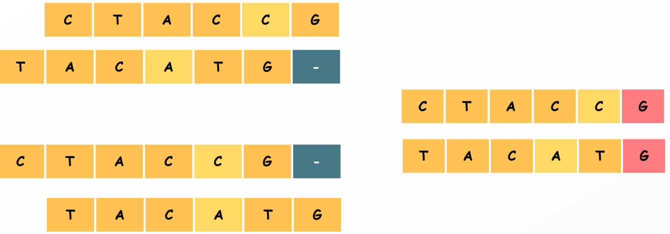
Each index depends on 3 others:

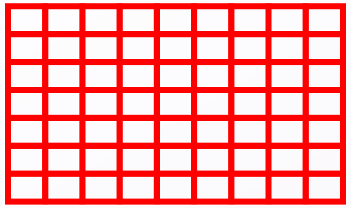
1. The one above it: $(i-1,j)$
2. The one to its left: $(i,j-1)$
3. The one to it's upper left: $(i-1,j-1)$

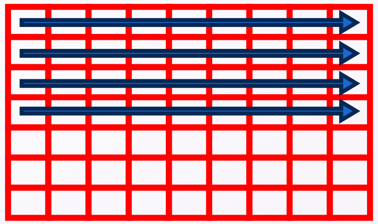
Bottom-Up Sequence Alignment

align(x, y):
    for $i = 0$ up to n:
        OPT[i][0] = $i \cdot \delta$ // Solve and save base cases
    for $j = 0$ up to m:
        OPT[0][j] = $j \cdot \delta$ // Solve and save base cases
    for $i = 1$ up to n:
        for $j = 1$ up to m :
            match = OPT[i – 1][j – 1] // solve each subproblem
            gapx = OPT[i][j – 1] // solve each subproblem
            gapy = OPT[i – 1][j] // solve each subproblem
            solution = min(match + $\alpha_{x_i y_j}$, gapx + δ, gapy+ δ) // pick solution
            OPT[i][j] = solution // save solution
    return OPT[n][m]

Edit Distance – Four Steps, 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?

Example run with *AGACATTG* and *GAGTTA*: $\delta = \alpha_{\text{mis}} = 1$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1								
A	2								
G	3								
T	4								
T	5								
A	6								

Example run with *AGACATTG* and *GAGTTA*: $\delta = \alpha_{\text{mis}} = 1$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2								
G	3								
T	4								
T	5								
A	6								

Example run with *AGACATTG* and *GAGTTA*: $\delta = \alpha_{\text{mis}} = 1$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1					
G	3								
T	4								
T	5								
A	6								

Example run with $AGACATTG$ and $GAGTTA$: $\delta = \alpha_{\text{mis}} = 1$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4								
T	5								
A	6								

Example run with *AGACATTG* and *GAGTTA*: $\delta = \alpha_{\text{mis}} = 1$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

Example run with *AGACATTG* and *GAGTTA*: $\delta = \alpha_{\text{mis}} = 1$

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

Example run with *AGACATTG* and *GAGTTA*: $\delta = \alpha_{\text{mis}} = 1$

		A	G	A	C	A	T	T	G
	0	**1**	2	3	4	5	6	7	8
G	1	1	**1**	2	3	4	5	6	7
A	2	1	2	**1**	2	3	4	5	6
G	3	2	1	2	**2**	**3**	4	5	5
T	4	3	2	2	3	3	**3**	4	5
T	5	4	3	3	3	4	3	**3**	4
A	6	5	4	3	4	3	4	4	**4**

Optimal Alignment

A G A C A T T G

_ G A G _ T T A

Final reminders

HW5 released today @ 11:30am.

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