CS 341: ALGORITHMS

Lecture 9: dynamic programming III

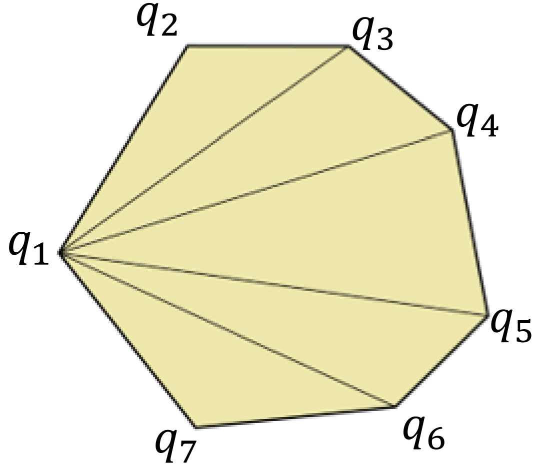
Readings: see website

Trevor Brown

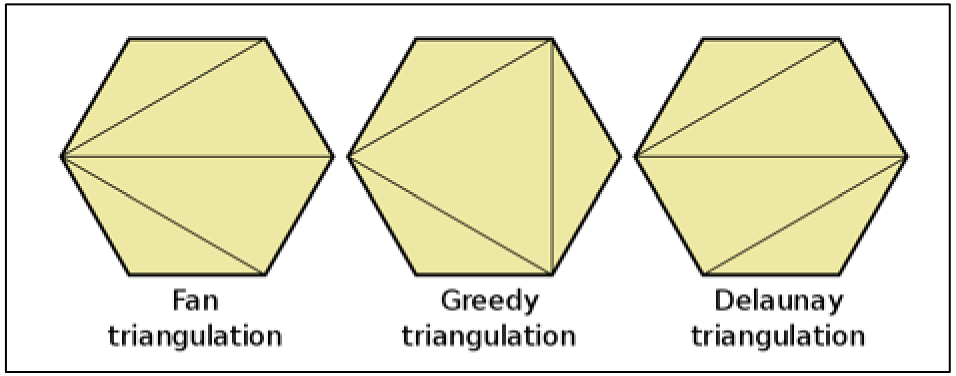
https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca

PROBLEM: MINIMUM LENGTH TRIANGULATION

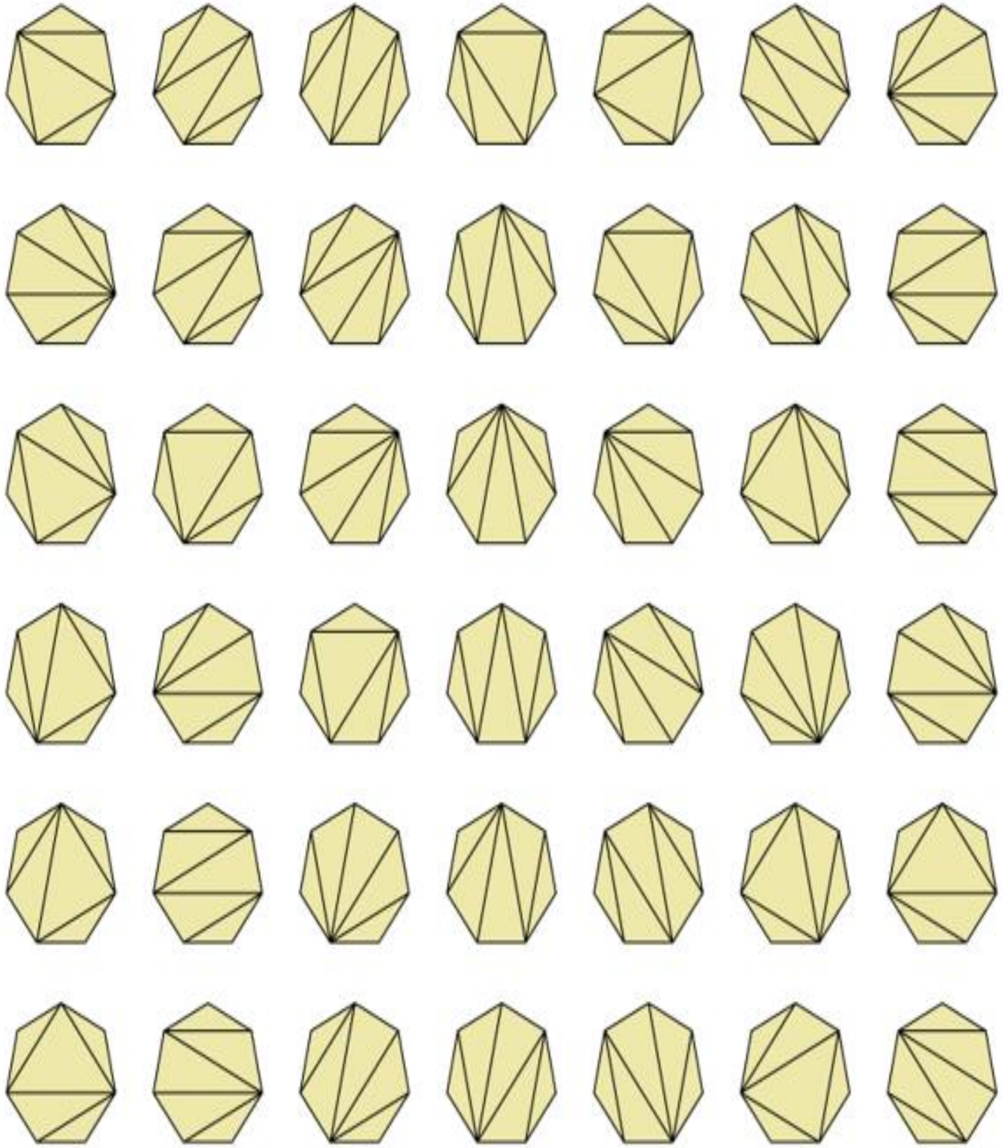


- **Input:** n points $q_1, \ldots, q_n$ in 2D space that form a **convex** n-gon P
  - Assume points are **sorted clockwise** around the center of P
- **Find:** a triangulation of P such that the sum of the perimeters of the $n-2$ triangles is minimized



- **Output:** the **sum** of the **perimeters** of the triangles in P

HOW HARD IS THIS PROBLEM?



How many triangulations are there?

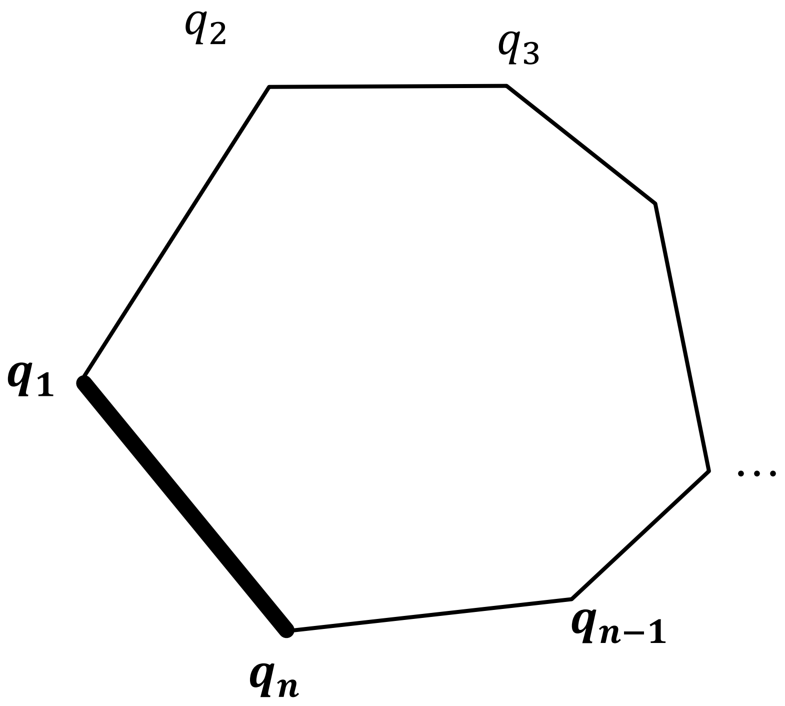
Number of triangulations of a convex n-gon = the $(\boldsymbol{n-2})$**nd Catalan number**

This is $C_{n-2} = \frac{1}{n-1}\binom{2n-4}{n-2}$

It can be shown that $C_{n-2} \in \Theta(\boldsymbol{4^n}/(n-2)^{3/2})$

PROBLEM DECOMPOSITION

The edge $q_n q_1$ is in a triangle with a third vertex q_k, where $k \in \{2, \ldots, n-1\}$.

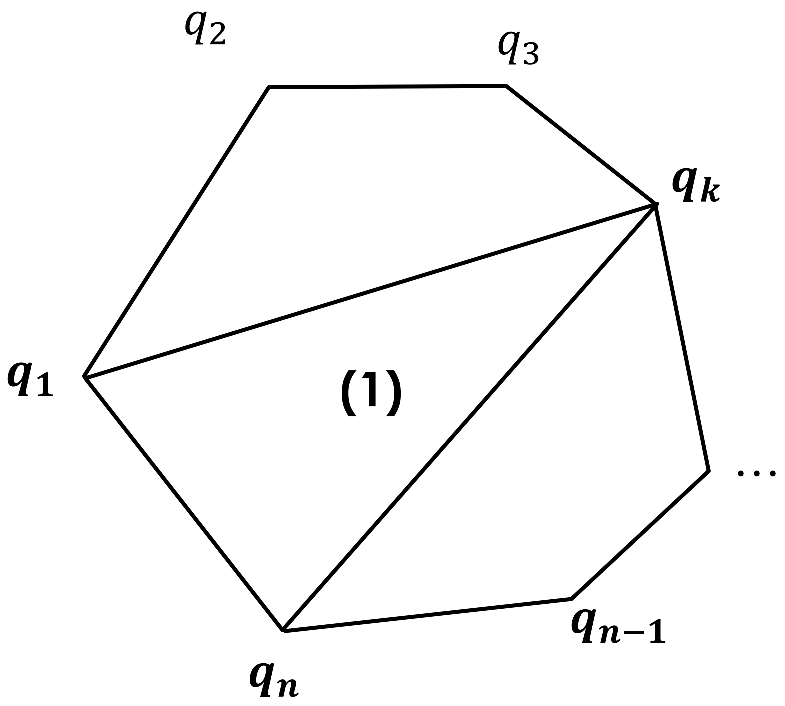


PROBLEM DECOMPOSITION

The edge $q_n q_1$ is in a triangle with a third vertex q_k, where $k \in \{2, \ldots, n-1\}$.

For a given k, we have:

the triangle $q_1 q_k q_n$, (1)



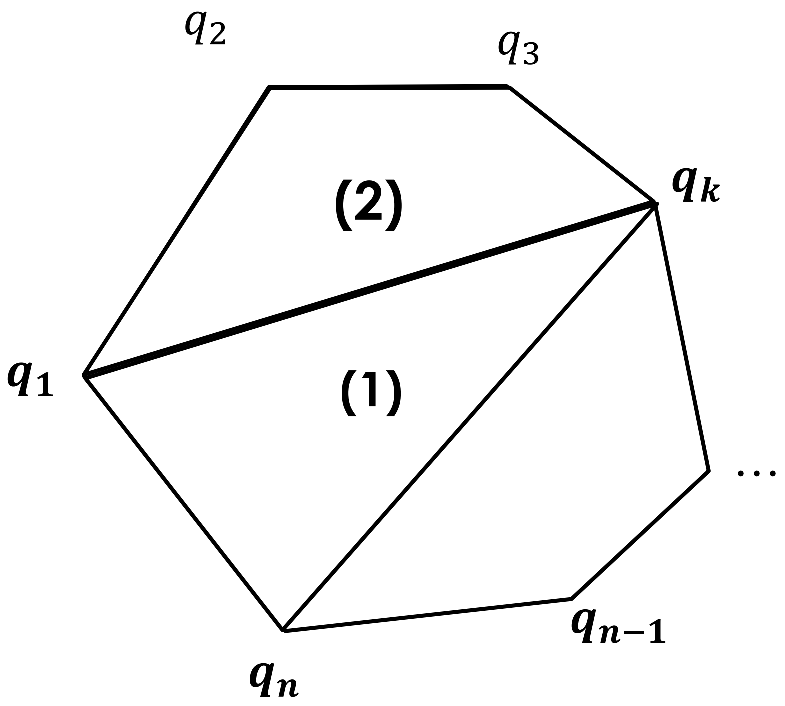
PROBLEM DECOMPOSITION

The edge $q_n q_1$ is in a triangle with a third vertex q_k, where $k \in \{2, \ldots, n-1\}$.

For a given k, we have:

the triangle $q_1 q_k q_n$, (1)

the polygon with vertices $q_1, \ldots, q_k$, (2)



PROBLEM DECOMPOSITION

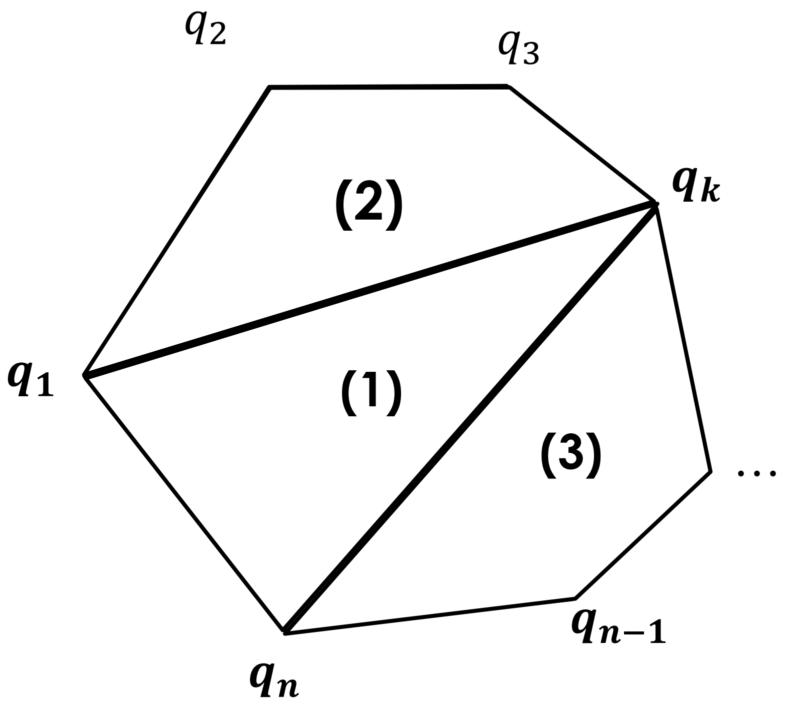
The edge $q_n q_1$ is in a triangle with a third vertex q_k, where $k \in \{2, \ldots, n-1\}$.

For a given k, we have:

the triangle $q_1 q_k q_n$, (1)

the polygon with vertices $q_1, \ldots, q_k$, (2)

the polygon with vertices $q_k, \ldots, q_n$. (3)



PROBLEM DECOMPOSITION

The edge $q_n q_1$ is in a triangle with a third vertex q_k, where $k \in \{2, \ldots, n-1\}$.

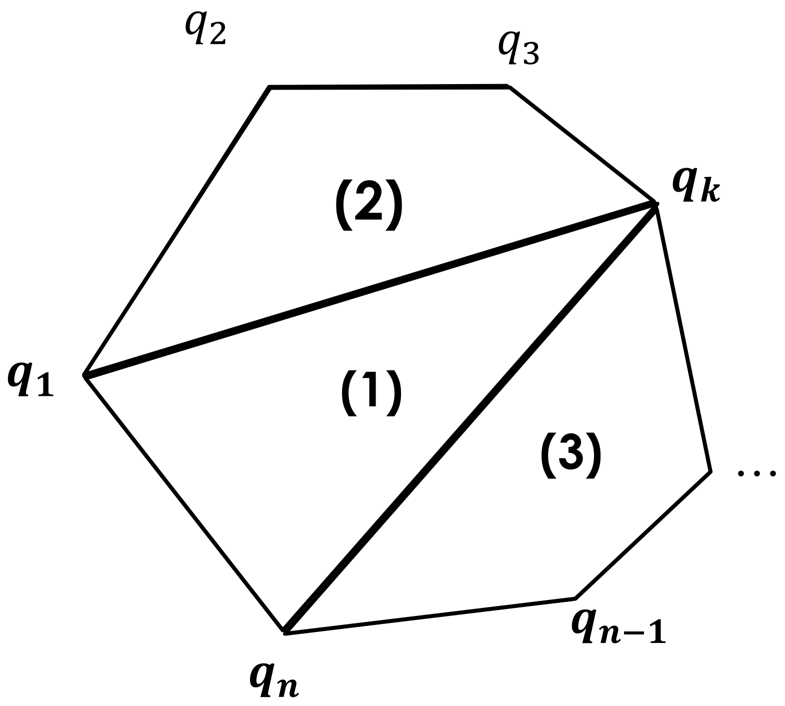
For a given k, we have:

the triangle $q_1 q_k q_n$, (1)

the polygon with vertices $q_1, \ldots, q_k$, (2)

the polygon with vertices $q_k, \ldots, q_n$. (3)

The optimal solution will consist of optimal solutions to the two subproblems in (2) and (3), along with the triangle in (1).



PROBLEM DECOMPOSITION

The edge $q_n q_1$ is in a triangle with a third vertex q_k, where $k \in \{2, \ldots, n-1\}$.

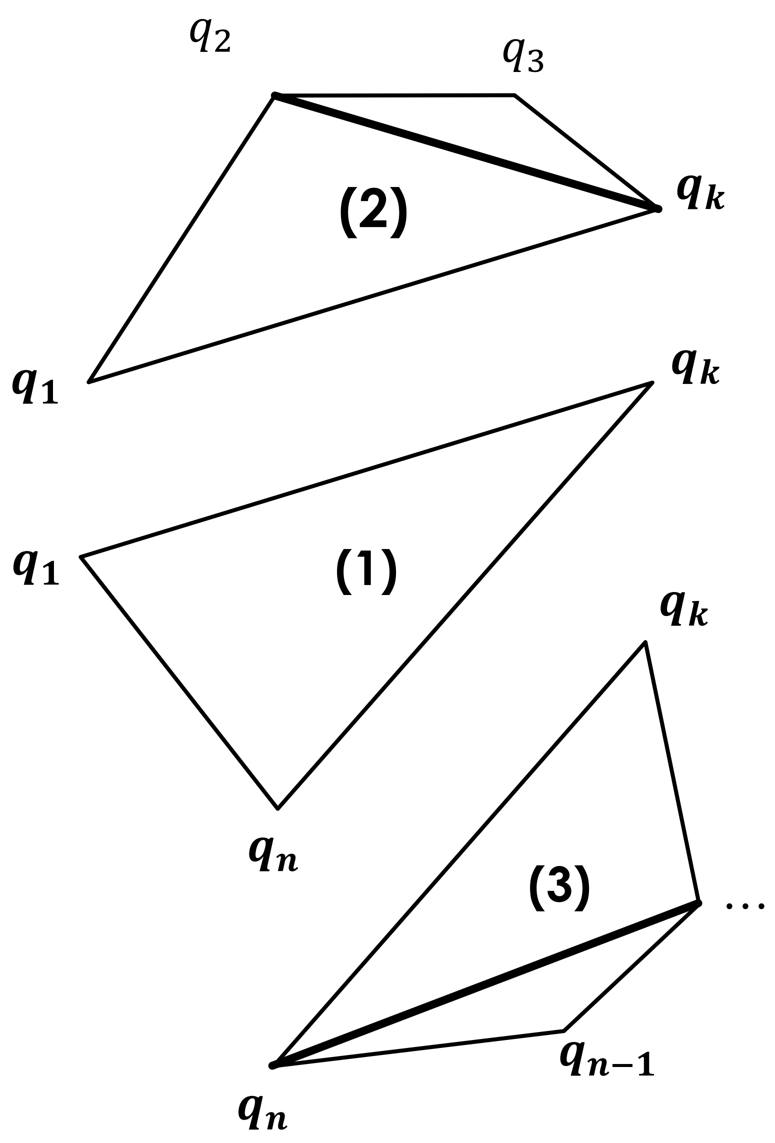
For a given k, we have:

the triangle $q_1 q_k q_n$, (1)

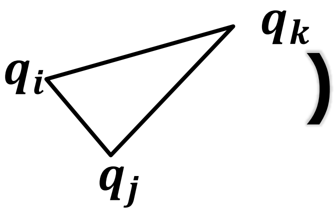
the polygon with vertices $q_1, \ldots, q_k$, (2)

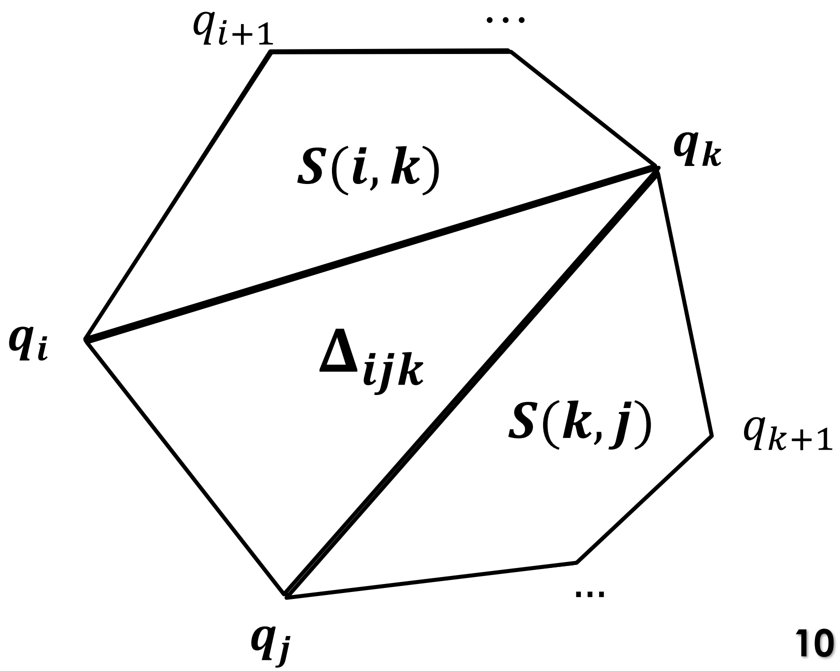
the polygon with vertices $q_k, \ldots, q_n$. (3)

The optimal solution will consist of optimal solutions to the two subproblems in (2) and (3), along with the triangle in (1).



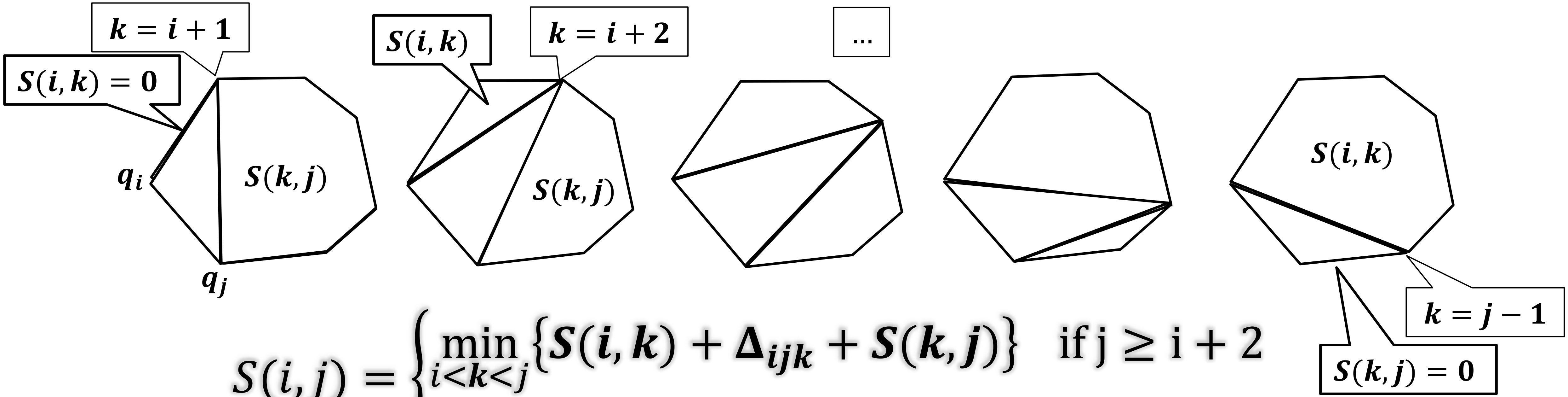
RECURRENCE RELATION

- Let $S(i,j)$ = optimal solution to the subproblem consisting of the polygon with vertices $q_i \ldots q_j$
- Let Δ_{ijk} denote **perimeter(**  **)**
- **If** a given **triangle** q_i, q_j, q_k is in the **optimal** solution, then $S(i,j) = S(i,k) + \Delta_{ijk} + S(k,j)$



RECURRENCE RELATION

- But we don't know the optimal k
  - Minimize over **all** $\boldsymbol{k}$ strictly between $\boldsymbol{i}$ and $\boldsymbol{j}$

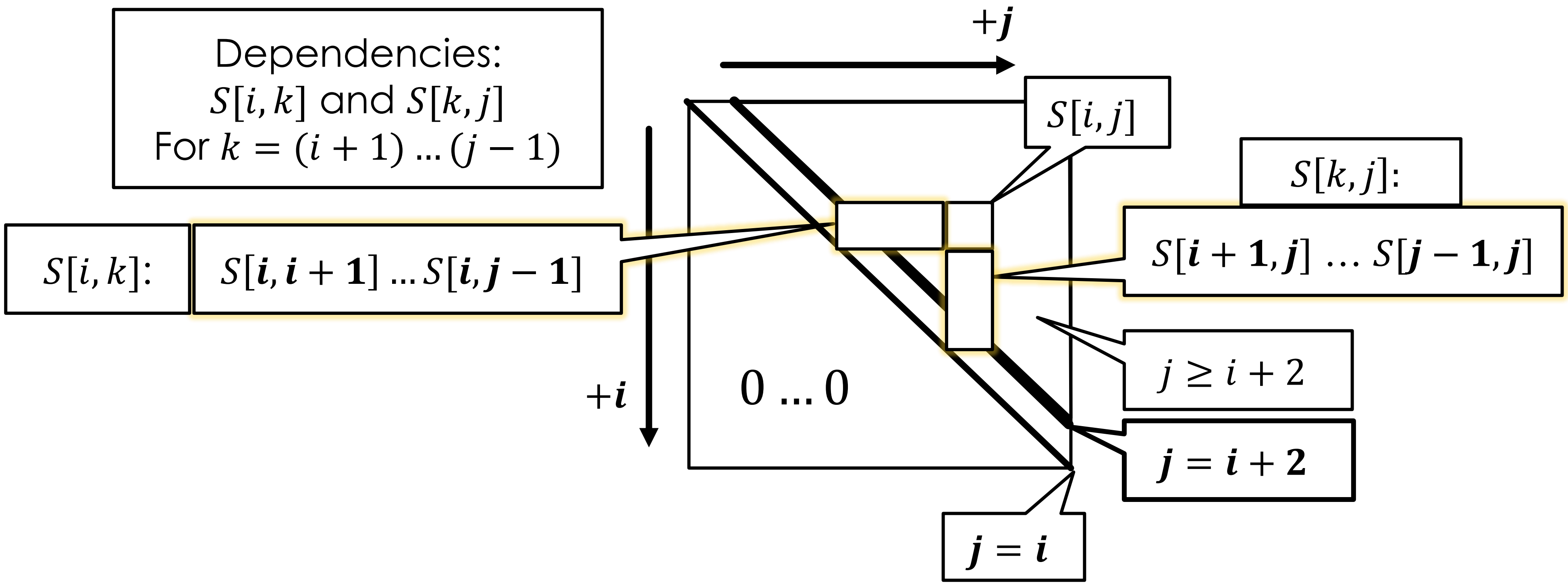


$$S(i,j) = \begin{cases} \min\limits_{i<k<j} \{\boldsymbol{S(i,k) + \Delta_{ijk} + S(k,j)}\} & \text{if } j \geq i + 2 \\ 0 & \text{otherwise} \end{cases}$$

FILLING IN THE TABLE

$$S(i,j) = \begin{cases} \min\limits_{i<k<j} \{ \boldsymbol{S(i,k)} + \boldsymbol{\Delta_{ijk}} + \boldsymbol{S(k,j)} \} & \text{if j} \geq \text{i} + 2 \\ 0 & \text{otherwise} \end{cases}$$

- Table $S[1..n, 1..n]$ of solutions to $S(i,j)$ for all $i,j \in \{1..n\}$



We depend on **larger** $\boldsymbol{i}$
And **same** $\boldsymbol{i}$ **but smaller** $\boldsymbol{j}$

What's a correct fill order?
for $\boldsymbol{i = n..1}$**, for** $\boldsymbol{j = 1..n}$

RUNTIME
WORD RAM MODEL

$$S(i,j) = \begin{cases} \min_{i<k<j} \{ \boldsymbol{S(i,k)} + \boldsymbol{\Delta_{ijk}} + \boldsymbol{S(k,j)} \} & \text{if j} \geq \text{i} + 2 \\ 0 & \text{otherwise} \end{cases}$$

- Number of subproblems: n^2
- Time to solve subproblem $S(i,j)$: $O(j-i) \subseteq O(n)$
- So total runtime is in $O(n^3)$
  - Some effort needed to show $\Omega(n^3)$, since so many subproblems are base cases, which take $\Theta(1)$ steps
- **Incidentally**, this is polynomial time (in the input size)
  - But basic runtime analysis does **not** require such an argument

PROBLEM: LONGEST COMMON SUBSEQUENCE (LCS)

Problem 5.3

Longest Common Subsequence

Instance: *Two sequences* $X = (x_1, \ldots, x_m)$ *and* $Y = (y_1, \ldots, y_n)$ *over some finite alphabet* Γ.

Find: *A maximum length sequence* Z *that is a subsequence of both* X *and* Y.

$Z = (z_1, \ldots, z_\ell)$ is a **subsequence** of X if there exist indices $1 \leq i_1 < \cdots < i_\ell \leq m$ such that $z_j = x_{i_j}$, $1 \leq j \leq \ell$.

Similarly, Z is a subsequence of Y if there exist (possibly different) indices $1 \leq h_1 < \cdots < h_\ell \leq n$ such that $z_j = y_{h_j}$, $1 \leq j \leq \ell$.

Let's **first** solve for the **length** of the LCS

EXAMPLES

- X=aaaaa Y=bbbbb Z=LCS(X,Y)=?
  - Z=ϵ (empty sequence)
- X=abcde Y=bcd Z=LCS(X,Y)=?
  - Z=bcd
- X=abcde Y=labef Z=LCS(X,Y)=?
  - Z=abe

POSSIBLE GREEDY SOLUTIONS?

- Alg: for each $x_i \in X$, try to choose a **matching** $y_j \in Y$ that is **to the right** of all previously chosen y_j values
  - X=**a**bcde Y=l**a**bef
  - X=**ab**cde Y=l**ab**ef
  - X=**ab**cde Y=l**ab**ef [no suitable y_j found]
  - X=**ab**cde Y=l**ab**ef [no suitable y_j found]
  - X=**ab**cd**e** Y=l**abe**f
  - Z=abe Optimal?

POSSIBLE GREEDY SOLUTIONS?

- Alg: for each $x_i \in X$, try to choose a **matching** $y_j \in Y$ that is **to the right** of all previously chosen y_j values
  - X=**a**zbracadabra Y=**a**bracadabraz
  - X=**az**bracadabra Y=**a**bracadabra**z**
  - X= **z**bracadabra Y=**a**bracadabra**z** [no y_j after **z**]
  - racadabra Y=**a**bracadabra**z** [no y_j after **z**]
  - Z=a Optimal?

Blindly taking z is bad. **How to decide** whether to take or leave z?

Try **both** possibilities! (Brute force / dynamic programming)

Similar greedy alg that goes right-to-left works for this input, but fails for other inputs.

DEFINING SUBPROBLEMS

- **Full problem:** $|\mathbf{LCS}(\boldsymbol{X},\boldsymbol{Y})|$ (i.e., **length** of LCS)
  - Reduce size by taking **prefixes** of X or Y
  - Let $\boldsymbol{X_i} = (\boldsymbol{x_1}, \dots, \boldsymbol{x_i})$ and $\boldsymbol{Y_i} = (\boldsymbol{y_1}, \dots, \boldsymbol{y_i})$

$\boldsymbol{X_m}$	x_1	x_2	x_3	x_4	…				x_{m-1}	x_m
$\boldsymbol{X_4}$	x_1	x_2	x_3	x_4						

Note $X = X_m$ and $Y = Y_n$

- **Subproblem:** $|\mathbf{LCS}(\boldsymbol{X_i},\boldsymbol{Y_j})|$
- **Shrinking the problem:** remove the **last letter** of X or Y

BUILDING SOLUTIONS FROM SUBPROBLEMS

EXAMPLE #1 TO BUILD INTUITION

X: a b r a c a z **z**

This cannot be the final **a** in Z

X_{m-1}

Since Z is a subsequence of X, $z_\ell =$ **a must appear in** X_{m-1}

Neither of these is part of Z

Y: a z b r a c a d a **b**

This cannot be the final **a** in Z

Y_{n-1}

$z_\ell =$ **a must be in** Y_{n-1}

Z: a b r a c **a**

Consider optimal solution $\boldsymbol{Z} = \mathbf{LCS}(\boldsymbol{X}, \boldsymbol{Y})$

Since $x_m, y_n \notin Z$ we know $\boldsymbol{Z} = \mathbf{LCS}(\boldsymbol{X_{m-1}}, \boldsymbol{Y_{n-1}})$

BUILDING SOLUTIONS FROM SUBPROBLEMS

EXAMPLE #2

Or maybe this is

X a b r a c a z **a**

This might be the final **a** in Z

Y a z b r a c a d a **b**

But this certainly is not :)

Y_{n-1}

Since Z is a subsequence of Y, $z_\ell =$ **a must appear in** Y_{n-1}

Z a b r a c **a**

Since $y_n \notin Z$ we know $Z = \mathbf{LCS}(X, Y_{n-1})$

Case $x_m \notin Z, y_n \in Z$ is symmetric

$Z = LCS(X_{m-1}, Y)$

BUILDING SOLUTIONS FROM SUBPROBLEMS

EXAMPLE #3

Or maybe this is…

X	a	b	r	a	c	a	z	**a**

This might be the final **a** in Z

Y	a	z	b	r	a	c	a	d	a	**a**

This might be the final **a** in Z

Z	a	b	r	a	c	a	**a**

Might as well match x_m and y_n with z_ℓ

Then we have $\boldsymbol{Z} = \mathbf{LCS}(\boldsymbol{X}_{\boldsymbol{m}-\mathbf{1}}, \boldsymbol{Y}_{\boldsymbol{n}-\mathbf{1}}) + \boldsymbol{z}_{\boldsymbol{\ell}}$

SUMMARIZING CASES

- z_ℓ matches **neither** x_m nor y_n $\quad Z = \mathrm{LCS}(X_{m-1}, Y_{n-1})$
- z_ℓ matches x_m but not y_n $\quad Z = \mathrm{LCS}(X_m, Y_{n-1})$
- z_ℓ matches y_n but not x_m $\quad Z = \mathrm{LCS}(X_{m-1}, Y_n)$
- z_ℓ matches **both** $\quad Z = \mathrm{LCS}(X_{m-1}, Y_{n-1}) + \boldsymbol{z_\ell}$
- **… but we don't know $\boldsymbol{z_\ell}$**
  - Try all cases and maximize
  - **Careful: last case is only valid if $\boldsymbol{x_m = y_n}$**
- Also note $\boldsymbol{x_m = y_n}$ **only holds in the last case**
  - Cases 2&3: trivial
  - Case 1: if $x_m = y_n \neq z_\ell$ then we can improve Z (contra)

DERIVING A RECURRENCE

Recall $Z = \text{LCS}(X_m, Y_n)$

- z_ℓ matches **neither** x_m nor y_n $(x_m \neq y_n)$ $Z = \text{LCS}(X_{m-1}, Y_{n-1})$
- z_ℓ matches x_m but not y_n $(x_m \neq y_n)$ $Z = \text{LCS}(X_m, Y_{n-1})$
- z_ℓ matches y_n but not x_m $(x_m \neq y_n)$ $Z = \text{LCS}(X_{m-1}, Y_n)$
- z_ℓ matches **both** $(x_m = y_n)$ $Z = \text{LCS}(X_{m-1}, Y_{n-1}) + z_\ell$
- **Let** $\boldsymbol{c(i,j) = |LCS(X_i, Y_j)|}$
- Brainstorming sensible base cases
  - $i = 0$ one string is empty, so $c(0,j) = 0$ (similarly for $j = 0$)
- General cases

$c(i,j) = c(i-1, j-1) + 1$	if $x_m = y_n$
$c(i,j) = \max\{c(i-1,j-1), c(i,j-1), c(i-1,j)\}$	if $x_m \neq y_n$

RECURRENCE

- Combining expressions

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1,j-1)+1 & \text{if } i,j \geq 1 \text{ and } x_i = y_j \\ \max\{c(i,j-1), c(i-1,j), c(i-1,j-1)\} & \text{if } i,j \geq 1 \text{ and } x_i \neq y_j \end{cases}$$

- Can simplify!
  - Observe $c(i-1,j-1) \leq c(i-1,j)$
    (former is a subproblem of the latter)

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1,j-1)+1 & \text{if } i,j \geq 1 \text{ and } x_i = y_j \\ \boldsymbol{\max\{c(i,j-1), c(i-1,j)\}} & \text{if } i,j \geq 1 \text{ and } x_i \neq y_j \end{cases}$$

Suppose $\boldsymbol{X}$ = **gdvegta**
and $\boldsymbol{Y}$ = **gvcekst**

$$c(i,j) = \begin{cases} 0 & if\ i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1 & if\ i,j \geq 1 \text{ and } x_i = y_j \\ \max\{c(i, j-1), c(i-1, j)\} & if\ i,j \geq 1 \text{ and } x_i \neq y_j \end{cases}$$

	X		g	d	v	e	g	t	a
Y		$i = 0$	1	2	3	4	5	6	7
	$j = 0$	Question 1							
g	1		Q3	1	Q6				
v	2		Q4	1	Q7				
c	3		Q5						
e	4	Q2				…			
k	5			…	…				
s	6		…						
t	7								

Suppose $\boldsymbol{X}$ = **gdvegta**
and $\boldsymbol{Y}$ = **gvcekst**

$$c(i,j) = \begin{cases} 0 & \textit{if } i = 0 \text{ or } j = 0 \\ c(i-1,j-1)+1 & \textit{if } i,j \geq 1 \text{ and } x_i = y_j \\ \max\{c(i,j-1), c(i-1,j)\} & \textit{if } i,j \geq 1 \text{ and } x_i \neq y_j \end{cases}$$

Y	X	$i=0$	g 1	d 2	v 3	e 4	g 5	t 6	a 7
	$j=0$	0	0	0	0	0	0	0	0
g	1	0	1	1	1	1	1	1	1
v	2	0	1	1	2	2	2	2	2
c	3	0	1	1	2	2	2	2	2
e	4	0	1	1	2	3	3	3	3
k	5	0	1	1	2	3	3	3	3
s	6	0	1	1	2	3	3	3	3
t	7	0	1	1	2	3	3	4	4

PSEUDOCODE

$$c(i,j) = \begin{cases} 0 & if\ i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1 & if\ i,j \geq 1 \text{ and } x_i = y_j \\ \max\{c(i, j-1), c(i-1, j)\} & if\ i,j \geq 1 \text{ and } x_i \neq y_j \end{cases}$$

Algorithm: $LCS1(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

```
for i ← 0 to m
        c[i, 0] ← 0
for j ← 0 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_j
                  then c[i, j] ← c[i − 1, j − 1] + 1
                  else c[i, j] ← max{c[i, j − 1], c[i − 1, j]}
return (c[m, n]);
```

Complexity: **Space? Time?** **(word RAM model)**
$\Theta(nm)$ for both

COMPUTING THE **LCS**

NOT JUST ITS LENGTH

To make it easy to find the actual LCS (not just its length),

Consider **which table entry** was used to calculate $c[i,j]$

$$= \begin{cases} 0 & if\ i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1 & if\ i,j \geq 1 \text{ and } x_i = y_j \\ \max\{c(i, j-1), c(i-1, j)\} & if\ i,j \geq 1 \text{ and } x_i \neq y_j \end{cases}$$

We store the **direction** to that entry in an array $\pi[i,j]$

Case 1: $\boldsymbol{c(i,j) = c(i, j-1)}$

We store "J" in $\pi[i,j]$ to indicate **decrementing** $\boldsymbol{j}$ (to get $i, j-1$)

Case 2: $\boldsymbol{c(i,j) = c(i-1, j)}$

We store "I" in $\pi[i,j]$ to indicate decrementing i (to get $i-1, j$)

Case 3: $\boldsymbol{c(i,j) = c(i-1, j-1) + 1}$

We store "IJ" in $\pi[i,j]$ to indicate decrementing **both** i and j

Recall in this case, $x_i = y_j$ so we **include** $\boldsymbol{x_i}$ **in the LCS**

In our example table we just **draw an arrow** to the entry…

SAVING THE DIRECTION TO THE **PREDECESSOR** SUBPROBLEM $\boldsymbol{\pi}$

```
LCS2(X[1..m], Y[1..n])
    c = new array[0..m][0..n]
    π = new array[0..m][0..n]

    for i = 0..m do c[i][0] = 0
    for j = 0..n do c[0][j] = 0

    for i = 1..m
        for j = 1..n
            if X[i] = Y[j]
                c[i][j] = c[i-1][j-1] + 1
                π[i][j] = "IJ"
            else if c[i][j-1] > c[i-1][j]
                c[i][j] = c[i][j-1]
                π[i][j] = "J"
            else // c[i][j-1] <= c[i-1][j]
                c[i][j] = c[i-1][j]
                π[i][j] = "I"

    return c, π
```

Case: $\boldsymbol{c(i,j) = c(i-1,j-1)+1}$

We store "IJ" in $\pi[i,j]$ to indicate decrementing **both** i and j

Recall in this case, $x_i = y_j$ so we **include** $\boldsymbol{x_i}$ **in the LCS**

Case: $\boldsymbol{c(i,j) = c(i,j-1)}$

We store "J" in $\pi[i,j]$ to indicate **decrementing** $\boldsymbol{j}$ (to get $i, j-1$)

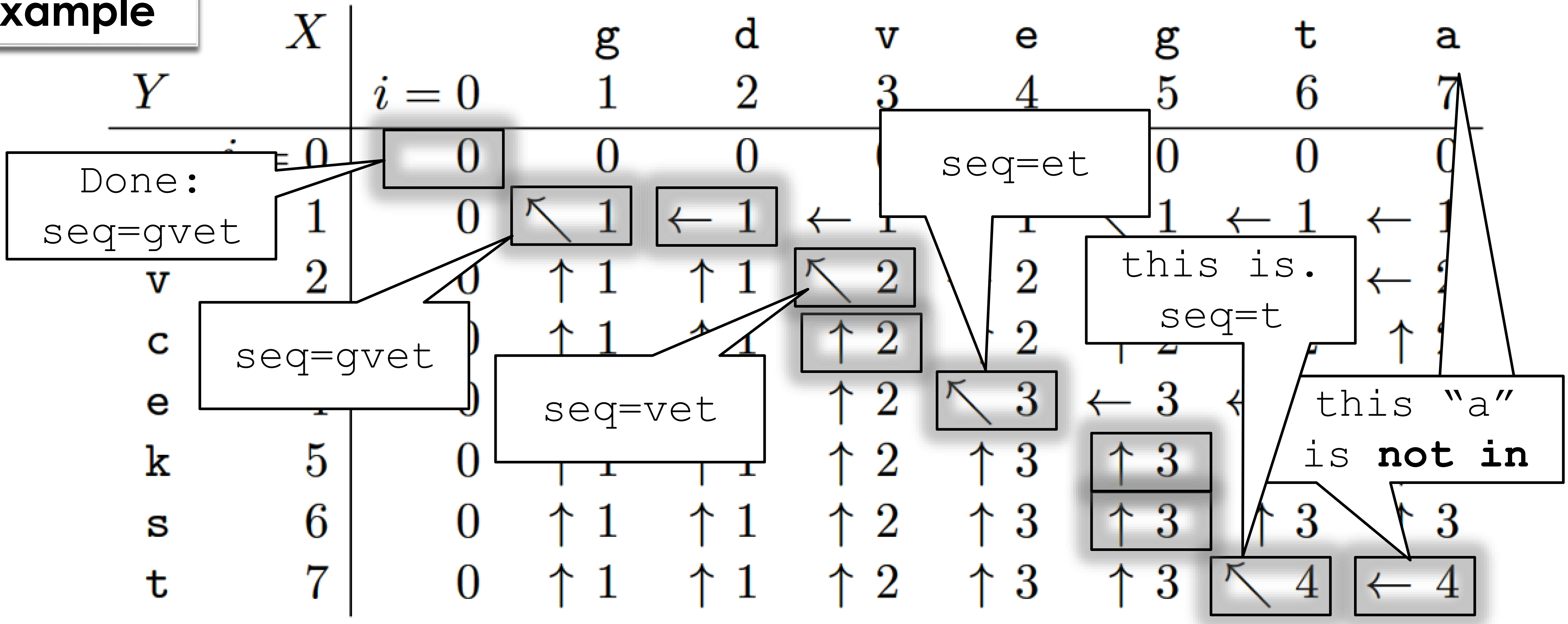
Case: $\boldsymbol{c(i,j) = c(i-1,j)}$

We store "I" in $\pi[i,j]$ to indicate decrementing i (to get $i-1, j$)

Suppose $X = \texttt{gdvegta}$ and $Y = \texttt{gvcekst}$.

How to obtain LCS=gvet from this table?

Example



FOLLOWING PREDECESSORS TO COMPUTE THE **LCS**

```
FindLCS(c[0..m][0..n], π[0..m][0..n], X[0..m])
    lcs = new string
    i = m
    j = n

    while i>0 and j>0
        if π[i][j] == "IJ"
            lcs.append(X[i])
            i--
            j--
        else if π[i][j] == "J"
            j--
        else // π[i][j] == "I"
            i--

    return reverse(lcs)
```

Complexities of this trace-back algo: Space? Time? (word RAM model)

space: O(n+m) words

time: O(n+m)

UNLIKELY TO GET THIS FAR

So this is likely just an exercise for you…

COIN CHANGING

Coin Changing

There **is** a denomination with **unit value**!

Problem 5.2

Coin Changing

Instance: *A list of* **coin denominations**, $1 = d_1, d_2, \ldots, d_n$, *and a positive integer* T, *which is called the* **target sum**.

Find: *An* n*-tuple of non-negative integers, say* $A = [a_1, \ldots, a_n]$, *such that* $T = \sum_{i=1}^{n} a_i d_i$ *and such that* $N = \sum_{i=1}^{n} a_i$ *is minimized.*

What subproblems should be considered?

What table of values should we fill in?

In 0-1 knapsack, we only considered **two subproblems** in our recurrence: **taking an item, or not.**

Here we can do **more than** *use a coin denomination or not*.

Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first i coin denominations $d_1, \ldots, d_i$ and target sum t.

Exploring: some sensible base case(s)?
General case: What are the different ways we could use coin denomination $\boldsymbol{d_i}$? What subproblems / solutions should we use?
Final recurrence relation

Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first i coin denominations $d_1, \ldots, d_i$ and target sum t.

Also $\boldsymbol{N[i, 0] = 0}$ for all i

Since $d_1 = 1$, we immediately have $N[1, t] = t$ for all t.

General case:
What are the different ways we could use coin denomination $\boldsymbol{d_i}$?
What subproblems / solutions should we use?

Final recurrence relation

Let $N[i,t]$ denote the optimal solution to the subproblem consisting of the first i coin denominations $d_1, \ldots, d_i$ and target sum t.

Also $\boldsymbol{N}[\boldsymbol{i}, \mathbf{0}] = \mathbf{0}$ for all i

Since $d_1 = 1$, we immediately have $N[1,t] = t$ for all t.

For $i \geq 2$, the number of coins of denomination d_i is an integer j where $0 \leq j \leq \lfloor t/d_i \rfloor$.

If we use j coins of denomination d_i, then the target sum is reduced to $t - jd_i$, which we must achieve using the first $i-1$ coin denominations.

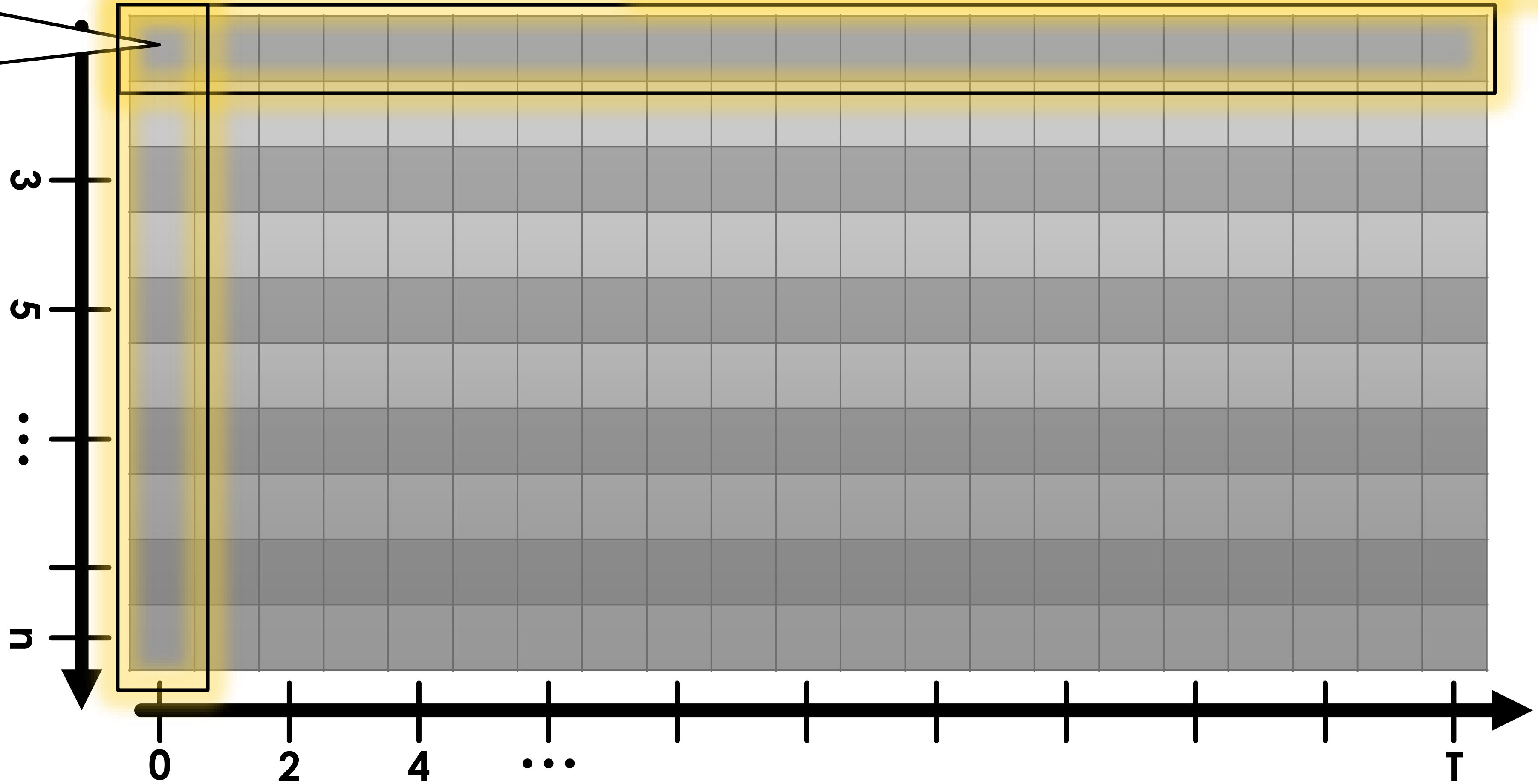
Thus we have the following recurrence relation:

$$N[i,t] = \begin{cases} \min\{j + N[i-1, t-jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i \geq 2 \\ t & \text{if } i = 1 \text{ OR } t = 0 \end{cases}$$

FILLING THE ARRAY $N[1 \ldots n, 0 \ldots T]$:

$$N[i,t] = \begin{cases} \min\{j + N[i-1, t - jd_i] : 0 \le j \le \lfloor t/d_i \rfloor\} & \text{if } i \ge 2 \\ t & \text{if } i = 1. \\ & \text{OR } t = 0 \end{cases}$$

No data dependencies on any other array cells.



***i*-axis (coin type)**

(recall: $N[i,t]$ uses coin types $1..i$)

***t*-axis (target sum remaining)**

FILLING THE ARRAY $N[1 \dots n, 0 \dots T]$:

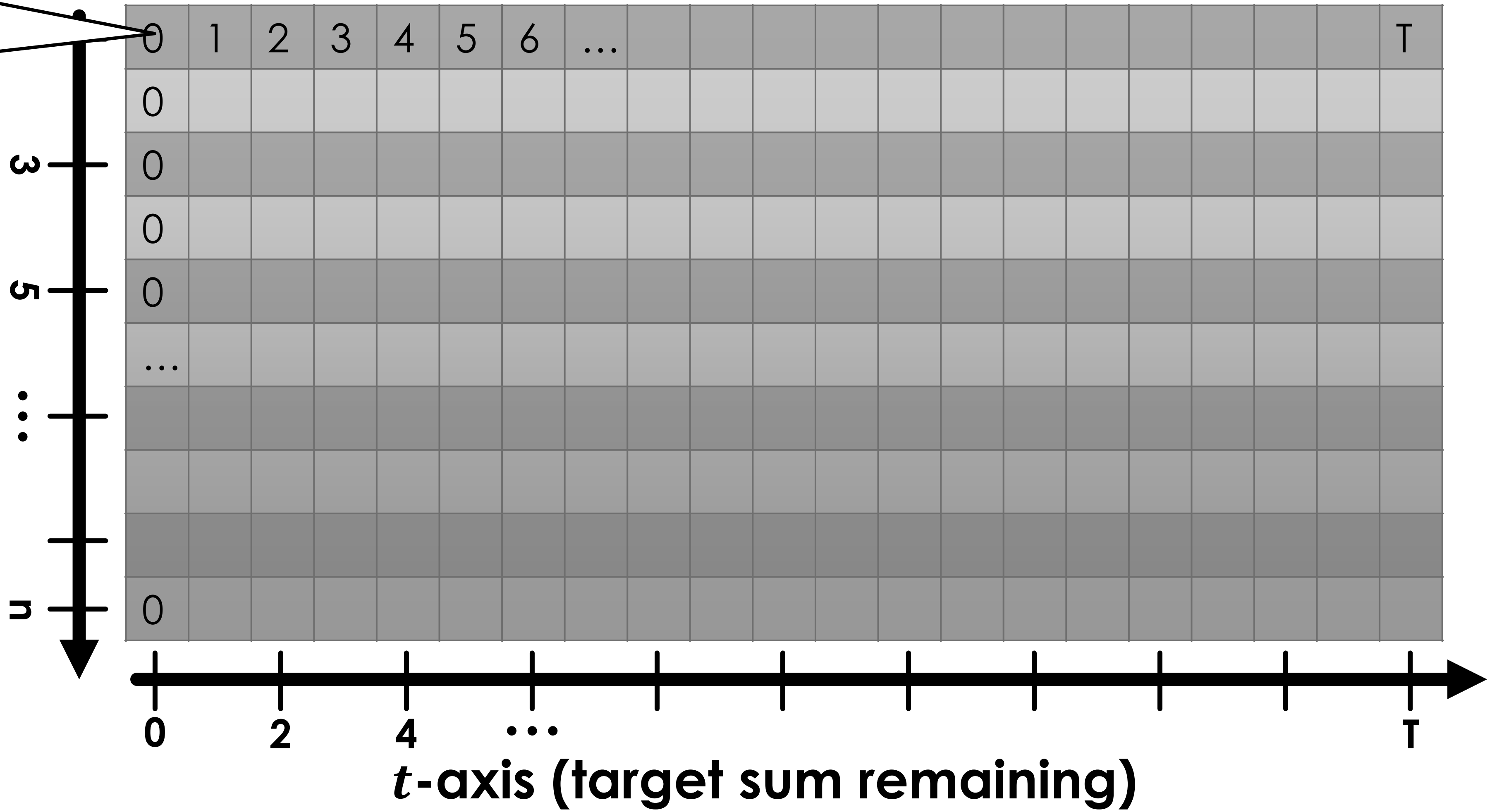
$$N[i,t] = \begin{cases} \min\{j + N[i-1, t - jd_i] : 0 \le j \le \lfloor t/d_i \rfloor\} & \text{if } i \ge 2 \\ t & \text{if } i = 1. \end{cases}$$

OR $t = 0$

No data dependencies on any other array cells.

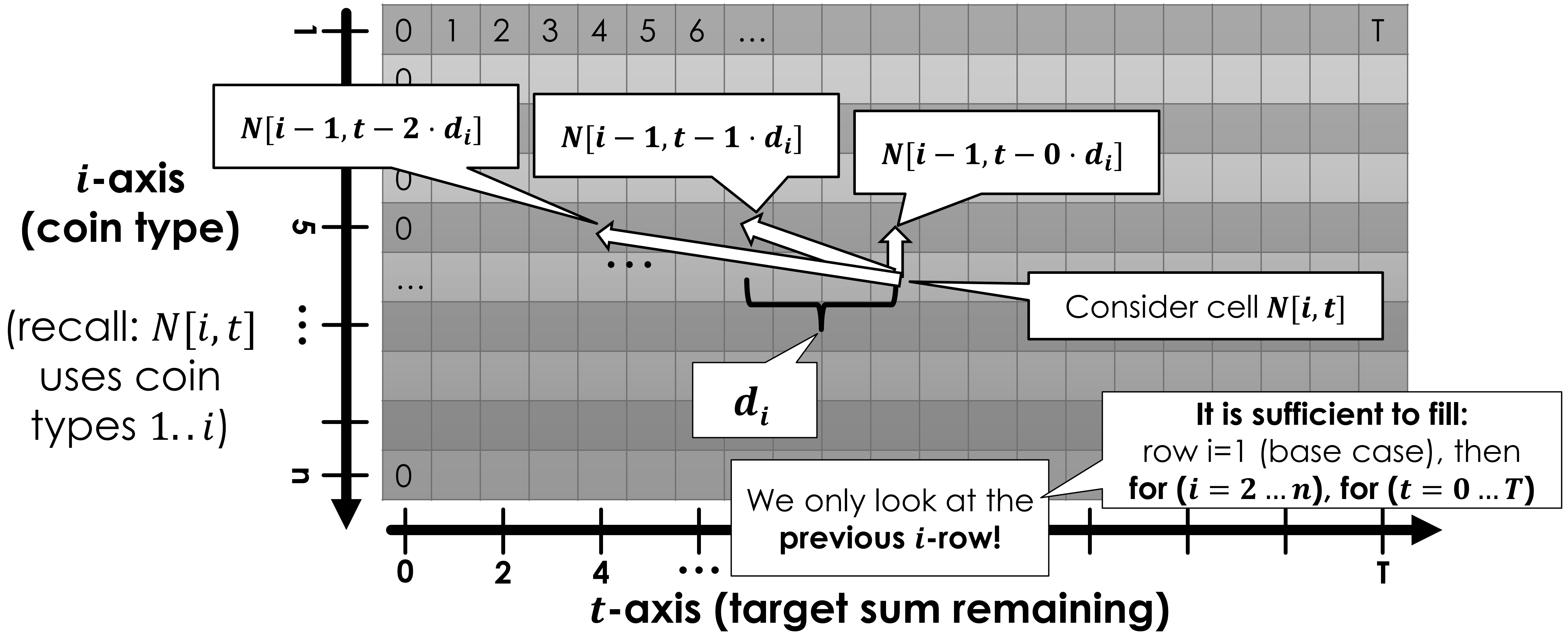
i-axis (coin type)

(recall: $N[i,t]$ uses coin types $1..i$)



FILLING THE ARRAY $N[1 \dots n, 0 \dots T]$:

$$N[i,t] = \begin{cases} \min\{j + N[i-1, t - jd_i] : 0 \le j \le \lfloor t/d_i \rfloor\} & \text{if } i \ge 2 \\ t & \text{if } i = 1. \text{ OR } t = 0 \end{cases}$$



$$N[i,t] = \begin{cases} \min\{j + N[i-1, t - jd_i] : 0 \le j \le \lfloor t/d_i \rfloor\} & \text{if } i \ge 2 \\ t & \text{if } i = 1. \end{cases}$$

```
CoinChangingDP(d[1..n], T)
    N = new table[1..n][0..T]
    J = new table[1..n][0..T]

    for t = 0..T     // base cases where i=1
        N[1][t] = t
        J[1][t] = t

    for i = 2..n     // general cases
        for t = 0..T
            // initially best solution is 0 of d[i]
            N[i][t] = N[i-1][t]
            J[i][t] = 0

            // try j>0 coins of type d[i]
            for j = 1..floor(t / d[i])
                if j + N[i-1][t-j*d[i]] < N[i][t]
                    N[i][t] = j + N[i-1][t-j*d[i]]
                    J[i][t] = j // best is currently j of d[i]

    return N[n][T] // can also return N, J
```

i.e., using coin $d_1 = 1$

$\boldsymbol{J[i,t]}$ = **# of coins** of type $\boldsymbol{d_i}$ used in $N[i,t]$

using *other* coin types

Compute min{…} over $\boldsymbol{j = 0 \ldots \lfloor t/d_i \rfloor}$

OUTPUTTING **OPTIMAL SET OF COINS**

```
CoinChangingDP_coins(d[1..n], J[1..n][0..T])
    counts = new array[1..n]
    t = T
    for i = n..1
        counts[i] = J[i][t]
        t = t - counts[i]*d[i]

    return counts
```

Recall $J[i,t]$ = # of coins of type d_i used in $N[i,t]$

We start at $J[n][T]$ = # of coins of type d_n used in the **optimal solution**

Exercise for later:
compute the correct output
without using $J[i,t]$
(i.e., using only N, d, T)

```
CoinChangingDP(d[1..n], T)
    N = new table[1..n][0..T]
    J = new table[1..n][0..T]

    for t = 0..T    // base cases where i=1
        N[1][t] = t
        J[1][t] = t

    for i = 2..n    // general cases
        for t = 0..T
            // initially best solution is 0 of d[i]
            N[i][t] = N[i-1][t]
            J[i][t] = 0

            // try j>0 coins of type d[i]
            for j = 1..floor(t / d[i])
                if j + N[i-1][t-j*d[i]] < N[i][t]
                    N[i][t] = j + N[i-1][t-j*d[i]]
                    J[i][t] = j // best is currently j of d[i]

    return N[n][T] // can also return N, J
```

Time complexity?

Unit cost computational model is reasonable here

Consider instance $I = (d, T)$

Runtime $\boldsymbol{R(I)} \in O\left(\sum_{i=2}^{n} \sum_{t=0}^{T} \left\lfloor \frac{t}{d_i} \right\rfloor\right)$

$$R(I) \in O\left(\sum_{i=2}^{n} \frac{1}{d_i} \sum_{t=0}^{T} t\right)$$

$$R(I) \in O\left(\sum_{i=2}^{n} \frac{1}{d_i} \left(\frac{T(T+1)}{2}\right)\right)$$

$\boldsymbol{R(I) \in O(DT^2)}$

where $D = \sum_{i=2}^{n} \frac{1}{d_i} < n.$

If T is small, this is much better than brute force

MEMOIZATION: AN ALTERNATIVE TO DP

Recall that the goal of dynamic programming is to eliminate solving subproblems more than once.

Memoization is another way to accomplish the same goal.

Memoization is a recursive algorithm based on same recurrence relation as would be used by a dynamic programming algorithm.

The idea is to remember which subproblems have been solved; if the same subproblem is encountered more than once during the recursion, the solution will be looked up in a table rather than being re-calculated.

This is easy to do if initialize a table of all possible subproblems having the value undefined in every entry.

Whenever a subproblem is solved, the table entry is updated.

EXAMPLE: USING MEMOIZATION TO COMPUTE FIBONACCI NUMBERS EFFICIENTLY

main

for $i \leftarrow 2$ **to** n

do $M[i] \leftarrow -1$

return $(\mathit{RecFib}(n))$

procedure $\mathit{RecFib}(n)$

if $n = 0$ **then** $f \leftarrow 0$

else if $n = 1$ **then** $f \leftarrow 1$

else if $M[n] \neq -1$ **then** $f \leftarrow M[n]$

else $\begin{cases} f_1 \leftarrow \mathit{RecFib}(n-1) \\ f_2 \leftarrow \mathit{RecFib}(n-2) \\ f \leftarrow f_1 + f_2 \\ M[n] \leftarrow f \end{cases}$

return (f);

If **M[n]** is already computed, **don't recurse!**

VISUALIZING MEMOIZATION

If **M[n]** is already computed, **don't recurse!**

procedure *RecFib*(n)
if $n = 0$ **then** $f \leftarrow 0$
else if $n = 1$ **then** $f \leftarrow 1$
else if $M[n] \neq -1$ **then** $f \leftarrow M[n]$

$$\textbf{else} \begin{cases} f_1 \leftarrow RecFib(n-1) \\ f_2 \leftarrow RecFib(n-2) \\ f \leftarrow f_1 + f_2 \\ M[n] \leftarrow f \end{cases}$$

return (f);