Dynamic Programming

CSE 101: Design and Analysis of Algorithms

Lecture 18

CSE 101: Design and analysis of algorithms

- Dynamic programming
  - Reading: Chapter 6
- Homework 7 will be assigned today
  - Due Dec 6, 11:59 PM

Dynamic Programming

- Dynamic programming is an algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until they are all solved

Dynamic programming steps

- Step 1: Define subproblems and corresponding array
- Step 2: What are the base cases?
- Step 3: Give recursion for subproblems
- Step 4: Find bottom-up order
- Step 5: What is the final output?
- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

- For analysis:
- Step 7: Correctness proof
- Step 8: Runtime analysis

Dynamic programming steps

- Step 0: Define the problem
- Step 1: Define subproblems and corresponding array
- Step 2: What are the base cases?
- Step 3: Give recursion for subproblems
- Step 4: Find bottom-up order
- Step 5: What is the final output?
- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

Dynamic programming example

STRING RECONSTRUCTION

String reconstruction

- Given a string of letters with no spaces or punctuation, how would you figure out how to separate the words? (Determine if there is a way to separate it into a string of words.)
  - Example: THESEARETHERULES

String reconstruction

- Given a string of letters with no spaces or punctuation, how would you figure out how to separate the words? (Determine if there is a way to separate it into a string of words.)
  - Example: THESEARETHERULES

Exhaustive search: 2^{n-1}

String reconstruction

- Given a string of letters with no spaces or punctuation, how would you figure out how to separate the words? (Determine if there is a way to separate it into a string of words.)
  - Example: THESEARETHERULES
  - Greedy approach: Find the first real word, remove it from the string, and repeat on the remaining string. If it does not work, then try a different way.

String reconstruction

- Given a string of letters with no spaces or punctuation, how would you figure out how to separate the words? (Determine if there is a way to separate it into a string of words.)
  - Example: THESE|ARE|THE|RULES
  - Step 0: Define the problem

String reconstruction

- Given a string of letters with no spaces or punctuation, how would you figure out how to separate the words? (Determine if there is a way to separate it into a string of words.)
  - Example: THESEARETHERULES
  - Step 0: Define the problem
    - Given a string of letters $x_1, \ldots, x_n$, output True if you can separate into a "sentence", output False if you cannot

String reconstruction

- Given a string of letters with no spaces or punctuation, how would you figure out how to separate the words? (Determine if there is a way to separate it into a string of words.)
  - Example: THESE|ARE|THE|RULES
  - Step 1: Define subproblems and corresponding array

String reconstruction

- Given a string of letters with no spaces or punctuation, how would you figure out how to separate the words? (Determine if there is a way to separate it into a string of words.)
  - Example: THESEARETHERULES
  - Step 1: Define subproblems and corresponding array
    - Given a string of letters $x_1, \ldots, x_k$, output True if you can separate into a "sentence", output False if you cannot

String reconstruction

- Step 1: Define subproblems and corresponding array
  - Given a string of letters $x_1, \ldots, x_k$, output True if you can separate into a "sentence", output False if you cannot

String reconstruction

- Step 1: Define subproblems and corresponding array
  - Given a string of letters $x_1, \ldots, x_k$, output True if you can separate into a "sentence", output False if you cannot
  - Let $S[k] = \begin{cases} \text{True if } x_1,\ldots,x_k \text{ is sentence separable} \\ \text{False} \qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$

String reconstruction

- Step 2: What are the base cases?
- Step 3: Give recursion for subproblems

String reconstruction

- Step 2: What are the base cases?
  $S[0] = \text{True}$
- Step 3: Give recursion for subproblems

String reconstruction

- Step 2: What are the base cases?

  $S[0] = \text{True}$

- Step 3: Give recursion for subproblems

  Case 1: $S[k-1]$ is True and x_k is a word

  Case 2: $S[k-2]$ is True and x_{k-1}, x_k is a word

  ⋮

  Case j:

  $S[k-j]$ is True and $x_{k-j+1}, \dots, x_k$ is a word

  $S[j-1]$ is True and $x_j, \dots, x_k$ is a word $\quad 1 \leq j \leq k$

String reconstruction

- Step 3: Give recursion for subproblems

  $S[k]$ is True if and only if there exists some $1 \leq j \leq k$ such that $S[j-1]$ is true and $x_j, \ldots, x_k$ is a word

- Step 4: Find bottom-up order

- Step 5: What is the final output?

String reconstruction

- Step 3: Give recursion for subproblems

  $S[k]$ is True if and only if there exists some $1 \leq j \leq k$ such that $S[j-1]$ is true and $x_j, \ldots, x_k$ is a word

- Step 4: Find bottom-up order

  $k = 0, \ldots, n$

- Step 5: What is the final output?

  $S[n]$

String reconstruction

- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

```
StringReconstruction(x[1…n])
        Initialize all S(.) to be false and all prev(.) to be nil
        S(0):=true    Base case
        for k from 1 to n
                j:=k-1
                while (not S(k)) and j>0
                        if S(j) is true and x[j+1,…k] is a valid word, then
                                S(k):=true
                                prev(k):=j      Found j
                        else
                                j:=j-1
        If S(n) then
                p:=n
                while p>0                       Reconstruct where the spaces are
                        print(p)
                        p:=prev(p)
```

String reconstruction

		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)																	
prev(k)																	

String reconstruction

		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T																
prev(k)																	

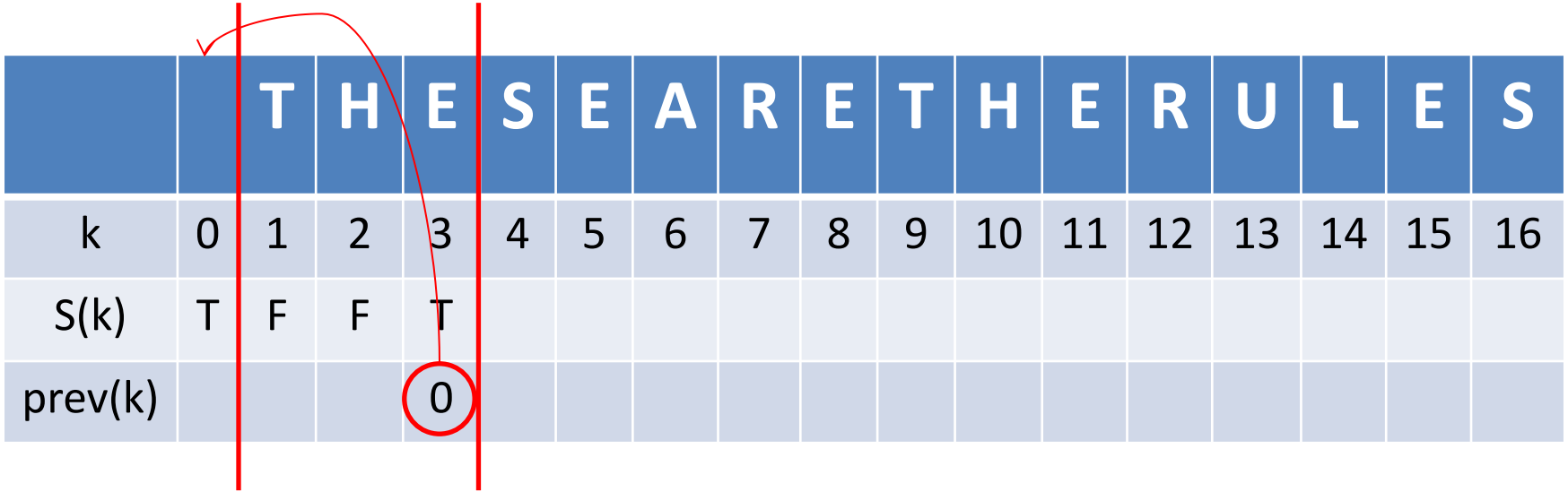
String reconstruction

		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F															
prev(k)																	

String reconstruction

		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F														
prev(k)																	

String reconstruction

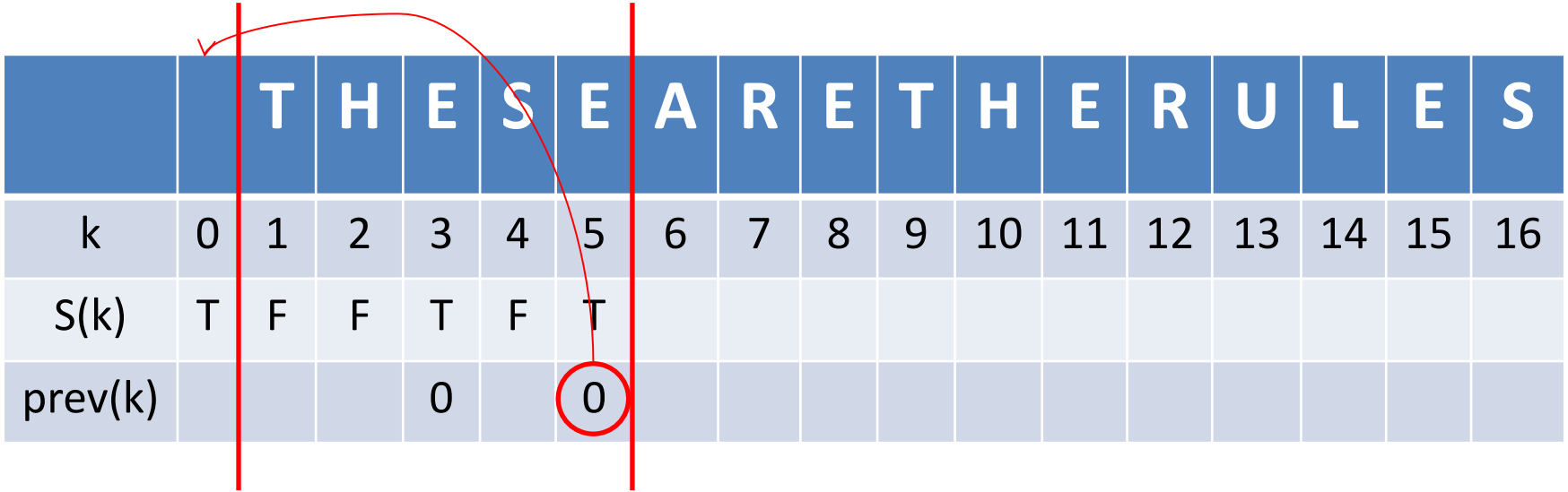


		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T													
prev(k)				0													

String reconstruction

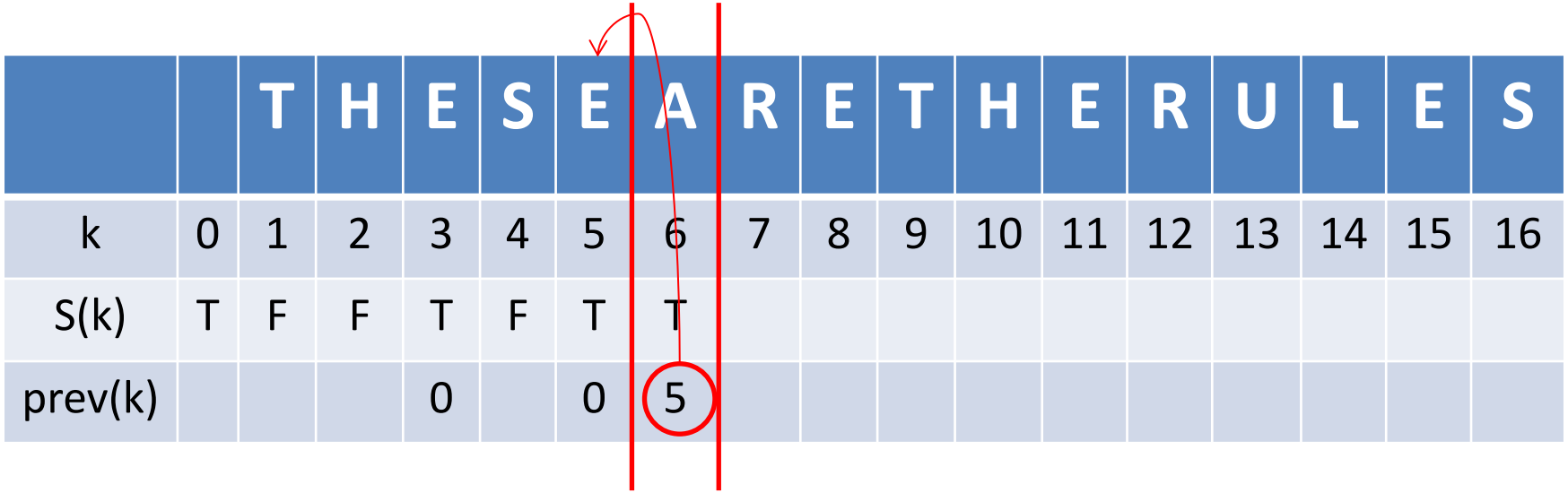
		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F												
prev(k)				0													

String reconstruction



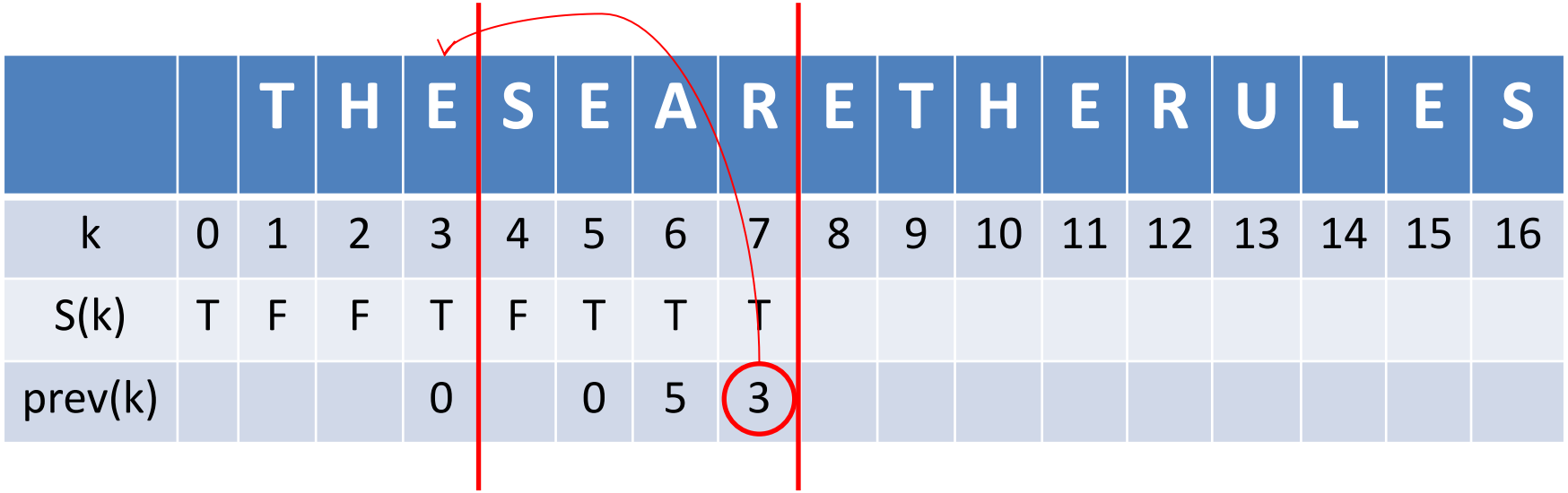
		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T											
prev(k)				0		0											

String reconstruction



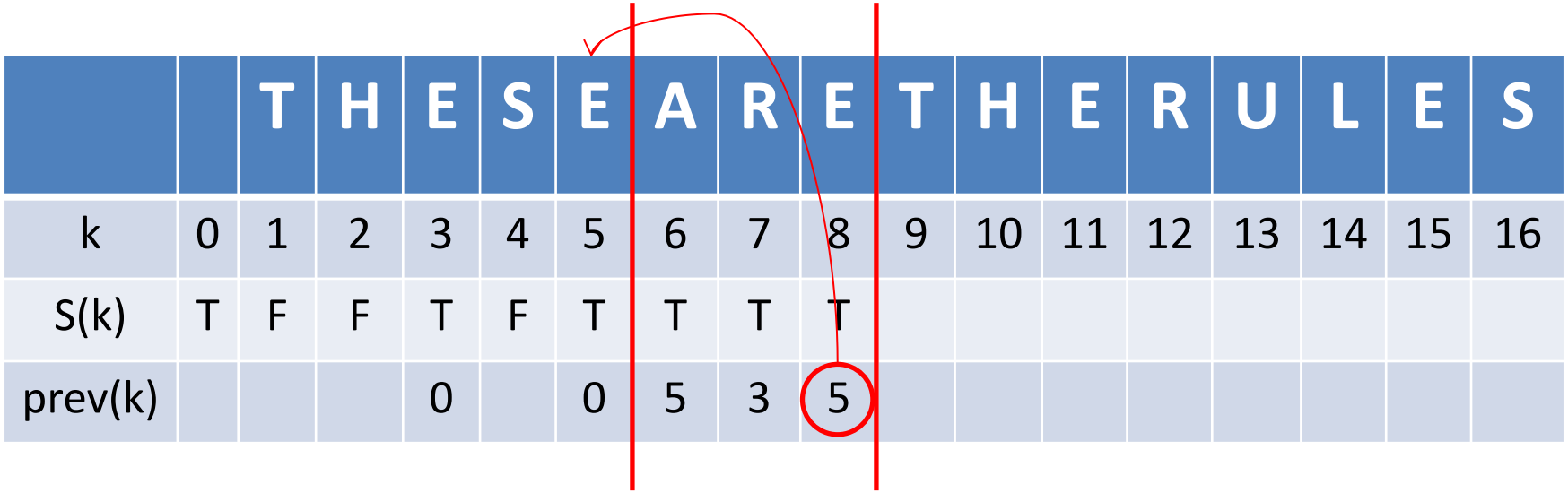
		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T										
prev(k)				0		0	5										

String reconstruction



		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T									
prev(k)				0		0	5	3									

String reconstruction



		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T								
prev(k)				0		0	5	3	5								

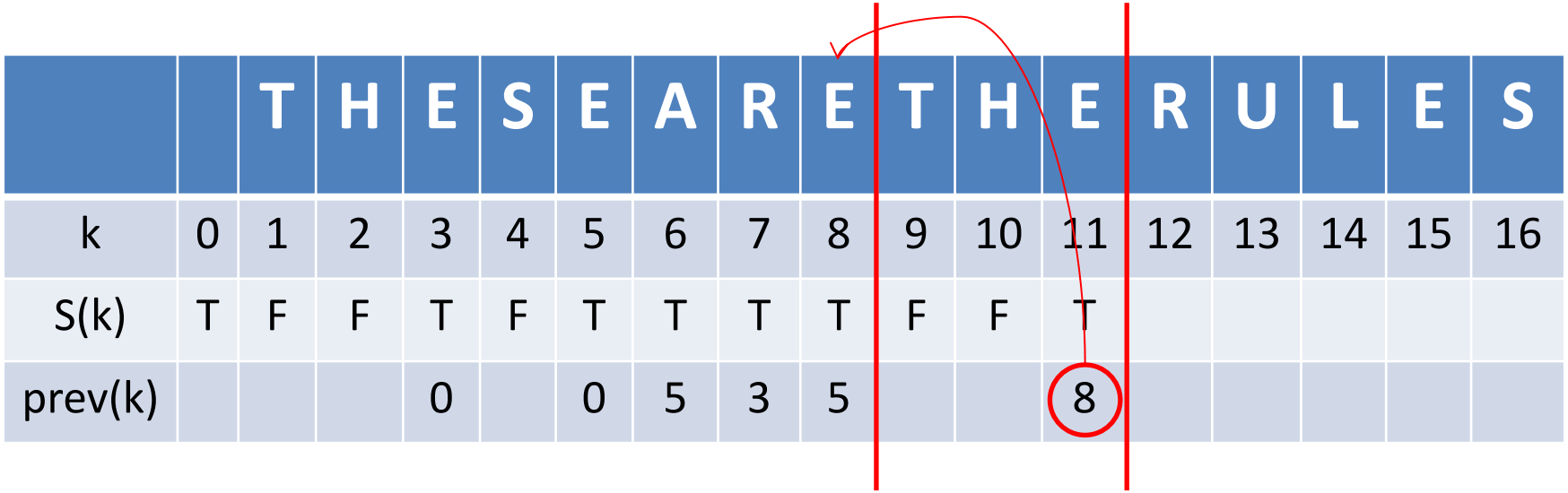
String reconstruction

		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F							
prev(k)				0		0	5	3	5								

String reconstruction

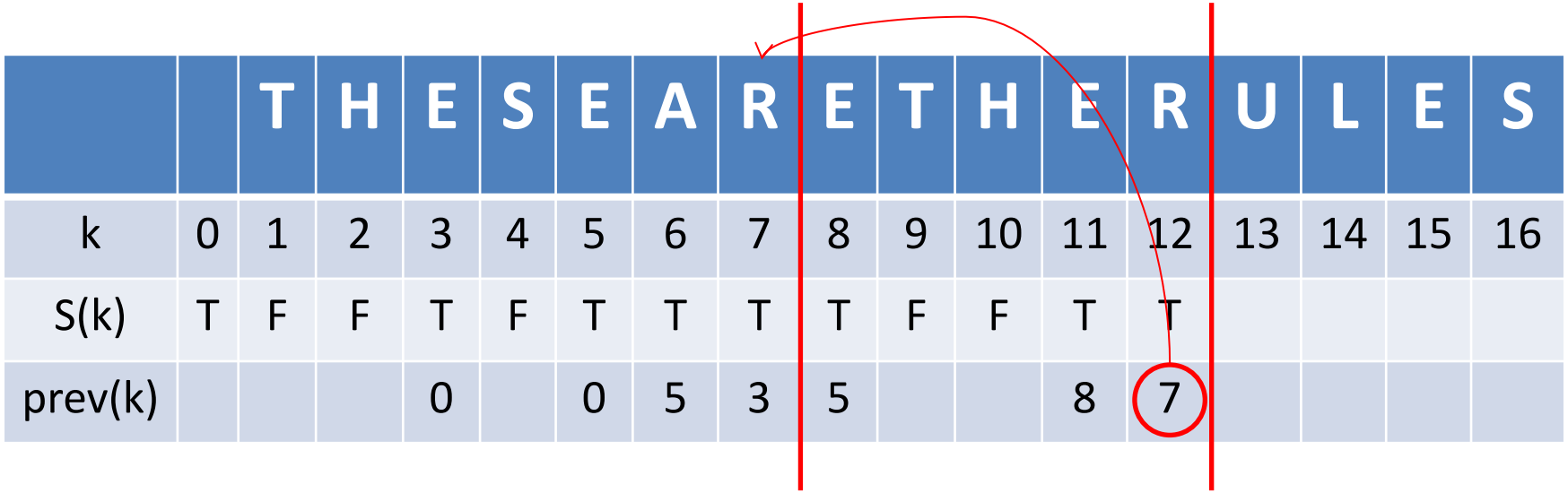
		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F	F						
prev(k)				0		0	5	3	5								

String reconstruction



		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F	F	T					
prev(k)				0		0	5	3	5			8					

String reconstruction



		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F	F	T	T				
prev(k)				0		0	5	3	5			8	7				

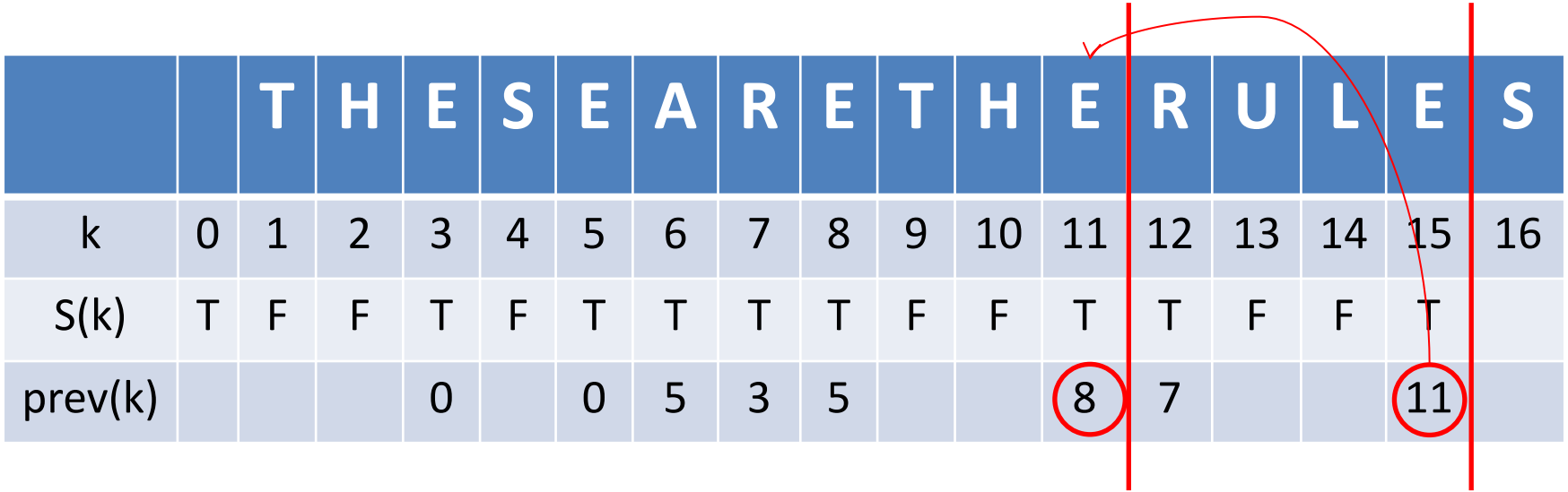
String reconstruction

		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F	F	T	T	F			
prev(k)				0		0	5	3	5			8	7				

String reconstruction

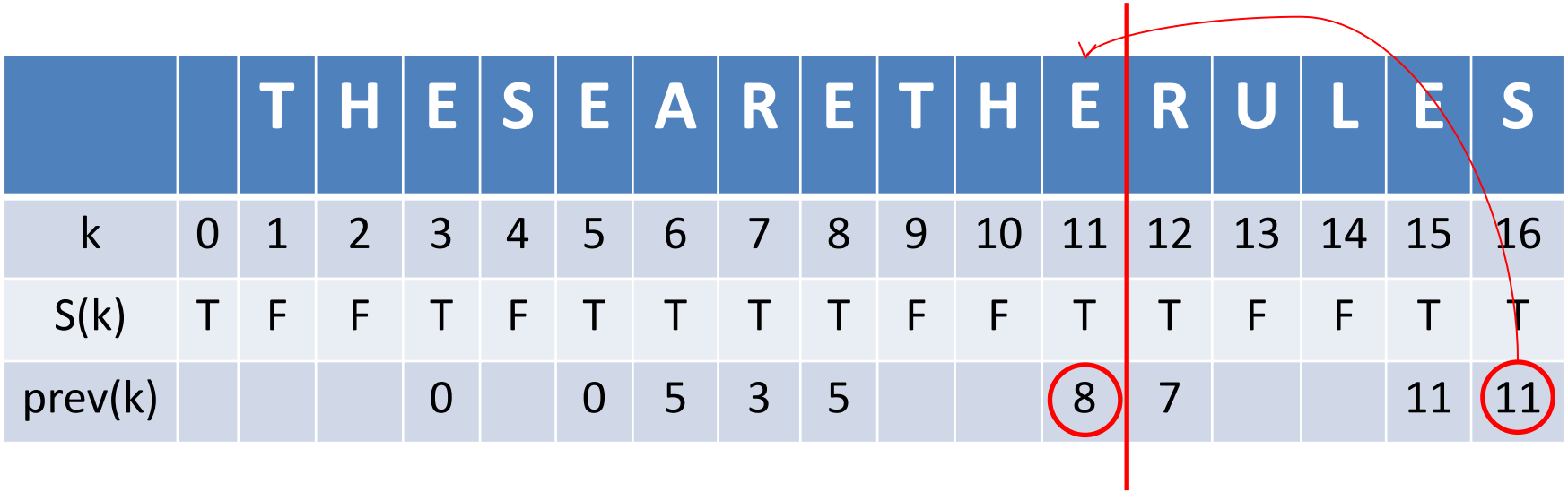
		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F	F	T	T	F	F		
prev(k)				0		0	5	3	5			8	7				

String reconstruction



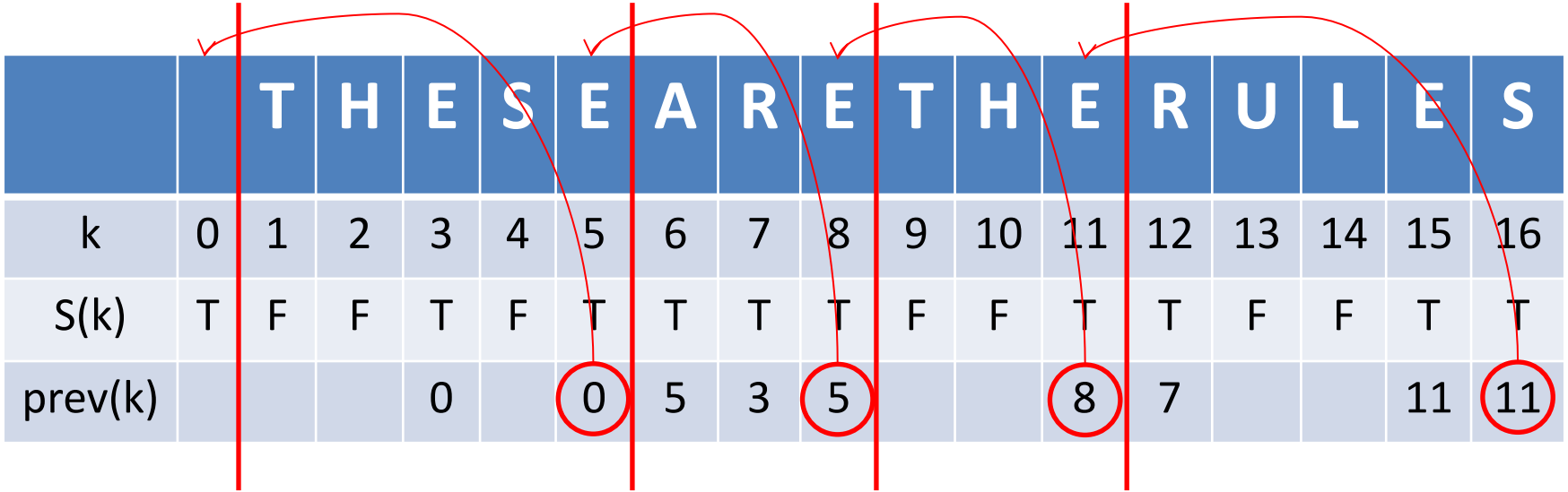
		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F	F	T	T	F	F	T	
prev(k)				0		0	5	3	5			8	7			11	

String reconstruction



		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F	F	T	T	F	F	T	T
prev(k)				0		0	5	3	5			8	7			11	11

String reconstruction

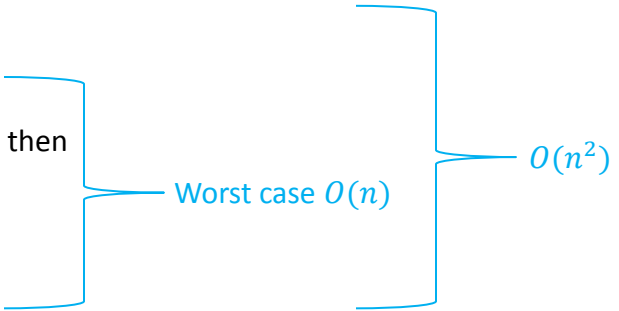


		T	H	E	S	E	A	R	E	T	H	E	R	U	L	E	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(k)	T	F	F	T	F	T	T	T	T	F	F	T	T	F	F	T	T
prev(k)				0		0	5	3	5			8	7			11	11

String reconstruction

- Step 8: Runtime analysis

```
StringReconstruction(x[1…n])
        Initialize all S(.) to be false and all prev(.) to be nil
        S(0):=true
        for k from 1 to n      n times
                j:=k-1
                while (not S(k)) and j>0
                        if S(j) is true and x[j+1,…k] is a valid word, then
                                S(k):=true
                                prev(k):=j
                        else
                                j:=j-1
        If S(n) then
                p:=n
                while p>0
                        print(p)
                        p:=prev(p)
```



Dynamic programming example

EDIT DISTANCE

Edit distance

- Given two words (strings), how can we define a notion of “closeness”
- For example, is PELICAN closer to PENGUIN or POLITICIAN?

Edit distance

- We can keep track of how many “changes” we need to change one word into another
- The changes can be
  - insertion,
  - deletion, or
  - substitution.
- For example, if we line up the words PELICAN and OSTRICH

P	E	L	I	C	A	N
O	S	T	R	I	C	H
s	s	s	s	s	s	s

7 substitutions

Edit distance

P	E	L	I	C	A	N
O	S	T	R	I	C	H
s	s	s	s	s	s	s

- Is 7 the cheapest cost?

Edit distance

P	E	L	I	C	A	N
O	S	T	R	I	C	H
s	s	s	s	s	s	s

- Is 7 the cheapest cost?

P	E	L	-	I	C	A	N
O	S	T	R	I	C	-	H
s	s	s	i			d	s

6 changes

Edit distance

- Another example, if we line up the words PELICAN and PENGUIN

P	E	L	I	C	A	N
P	E	N	G	U	I	N
		s	s	s	s	

4 changes

Edit distance

- Another example, if we line up the words PELICAN and POLITICIAN

P	E	L	I	C	A	N	-	-	-
P	O	L	I	T	I	C	I	A	N
	s			s	s	s	i	i	i

7 changes

Edit distance

- Brute force: try all possible combinations and find the minimum cost of all of them
- What is the lower bound of the number of possible combinations if the size of the words are n,m, $n \leq m$?

Edit distance

- Brute force: try all possible combinations and find the minimum cost of all of them
- What is the lower bound of the number of possible combinations if the size of the words are n,m, n≤m?
  - Each column could be one of three changes (at least for the first n columns), so there are at least 3^n different combinations

Edit distance

- Step 0: Define the problem

Edit distance

- Step 0: Define the problem
  - Given $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, output the minimum edit distance between them
- Step 1: Define subproblems and corresponding array

Edit distance

- Step 0: Define the problem
  - Given $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, output the minimum edit distance between them
- Step 1: Define subproblems and corresponding array
  - Let E(i,j) be the minimum cost of the two words x[1…i] and y[1…j]

Edit distance

- Step 2: What are the base cases?

Edit distance

- Step 2: What are the base cases?
  - When the first word is empty then the edit distance is the length of the second word and when the second word is empty the cost is the length of the first word

  $E(i,0)=i$

  $E(0,j)=j$

Edit distance

- Step 3: Give recursion for subproblems

Edit distance

- Step 3: Give recursion for subproblems
  - Split into cases depending on the last column of the alignment of x[1…i] and y[1…j]
  - Case 1: the last column looks like

x[i]
-

  - Case 2: the last column looks like

-
y[j]

  - Case 3: the last column looks like

x[i]
y[j]

Example:

x[1]		x[2]	x[3]	x[4]	x[5]		x[6]	x[7]
	y[1]	y[2]		y[3]	y[4]	y[5]		

Edit distance

- Step 3: Give recursion for subproblems
  - Case 1: the last column looks like

x[i]
-

    - This is a deletion with a cost of 1 so if the minimum cost of x[1…i] and y[1…j] has this in the last column then E(i,j)=1+E(i-1,j)

Example:

x[1]		x[2]	x[3]	x[4]	x[5]		x[6]	x[7]
	y[1]	y[2]		y[3]	y[4]	y[5]		

Edit distance

- Step 3: Give recursion for subproblems
  - Case 2: the last column looks like 

    - This is an insertion with a cost of 1 so if the minimum cost of x[1…i] and y[1…j] has this in the last column then E(i,j)=1+E(i,j-1)

Example:

x[1]		x[2]	x[3]	x[4]	x[5]		x[6]	x[7]
	y[1]	y[2]		y[3]	y[4]	y[5]		

Edit distance

- Step 3: Give recursion for subproblems
  - Case 3: the last column looks like

x[i]
y[j]

    - Case 3.1: x[i]=y[j] (no cost)

      E(i,j)=E(i-1,j-1)

    - Case 3.2: x[i]≠y[j] (substitution cost of 1)

      E(i,j)=1+E(i-1,j-1)

Example:

x[1]		x[2]	x[3]	x[4]	x[5]		x[6]	x[7]
	y[1]	y[2]		y[3]	y[4]	y[5]		

Edit distance

- Step 3: Give recursion for subproblems
  - So take the minimum of all three cases

Case 1 Case 2 Case 3

E(i,j)=min(1+E(i-1,j), 1+E(i,j-1), (1-$\delta_{x[i],y[j]}$)+E(i-1,j-1))

where delta function $\delta_{a,b} = \begin{cases} 0 \; if \; a \neq b \\ 1 \; if \; a = b \end{cases}$

Edit distance

- Step 4: Find bottom-up order

Edit distance

- Step 4: Find bottom-up order
  - To calculate E(i,j), we need to know E(i-1,j), E(i,j-1) and E(i-1,j-1)
  - Think of a 2D array and where are the indices in relation to (i,j)?

E(i-1,j-1)	E(i-1,j)
E(i,j-1)	E(i,j)

  - So, order them in such a way to visit all the necessary entries before you visit (i,j)
  - One way to do this is left to right through rows, top to bottom through columns

Edit distance

- Step 5: What is the final output?

Edit distance

- Step 5: What is the final output?

  E(n,m)

Edit distance

- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

EditDist(x[1…n],y[1…m])

Initialize for i from 1 to n, E(i,0)=I, and for j from 1 to m, E(0,j)=j Base cases

for i from 1 to n Each row

for j from 1 to m Each column

E(i,j)=min(1+E(i-1,j), 1+E(i,j-1), (1-$\delta_{x[i],y[j]}$)+E(i-1,j-1))

Return E(n,m)

	Ø	P	E	L	I	C	A	N
Ø	0	1	2	3	4	5	6	7
P	1							
O	2							
L	3							
I	4							
T	5							
I	6							
C	7							
I	8							
A	9							
N	10							

	Ø	P	E	L	I	C	A	N
Ø	0	1	2	3	4	5	6	7
P	1	0						
O	2							
L	3							
I	4							
T	5							
I	6							
C	7							
I	8							
A	9							
N	10							

Dynamic programming example

THE KNAPSACK PROBLEM

The knapsack problem

- Suppose you are a burglar who breaks into a store and you want to leave with the maximum value of items
- Example: your knapsack can only hold 13 lbs and the items in the store are

	Silver	4 lbs jewery	Diamond jacket	Golden suitcases	Platinum Nintendo	Solid ruby bowling ball
Value	4	9	12	15	19	21
Weight	2	4	5	7	8	9

The knapsack problem

- What is the maximum value you can have from a list of items a[1],…,a[n] where each item has a value v[i] and a weight w[i] given that you cannot have more weight than W
- How would you go about solving this problem using backtracking?

The knapsack problem

- What is the maximum value you can have from a list of items a[1],…,a[n] where each item has a value v[i] and a weight w[i] given that you cannot have more weight than W
- How would you go about solving this problem using backtracking?
  - Did you choose item n or not?
    - Case 1: a[n] is part of max value set (item n is worth taking)
    - Case 2: a[n] is not part of max value set (item n is not worth taking)

The knapsack problem

- How would you go about solving this problem using backtracking?
  - Did you choose item n or not?
    - Case 1: a[n] is part of max value set (item n is worth taking)
      - K[a[1]…a[n], W] = K[a[1]…a[n], W-w[n]] + v[n]
    - Case 2: a[n] is not part of max value set (item n is not worth taking)
      - K[a[1]…a[n], W] = K[a[1]…a[n-1], W]

The knapsack problem

- How would you go about solving this problem using backtracking?

```
BTKS(w[1…n],v[1…n],W)
 IN = BTKS(w[1…n],v[1…n],W-w[n])+v[n]
 OUT = BTKS(w[1…n-1],v[1…n-1],W)
 return max(IN,OUT)
```

The knapsack problem

- Convert from backtracking to dynamic programming
  - Replace the recursive calls with an array value. Let KS(j,w) be the maximum value you can fit in a w-capacity knapsack using only the items 1…j

The knapsack problem

```
BTKS(w[1…n],v[1…n],W)
 IN = BTKS(w[1…n],v[1…n],W-w[n])+v[n]
 OUT = BTKS(w[1…n-1],v[1…n-1],W)
 return max(IN,OUT)
```

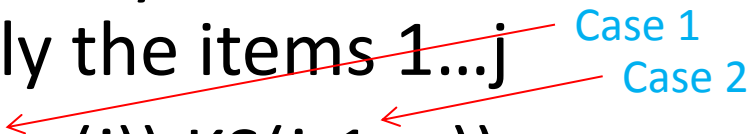
Backtracking

Dynamic programming

- Replace the recursive calls with an array value. Let KS(j,w) be the maximum value you can fit in a w-capacity knapsack using only the items 1…j

The knapsack problem

- Replace the recursive calls with an array value. Let KS(j,w) be the maximum value you can fit in a w-capacity knapsack using only the items 1…j
- Then, KS(j,w) = max(KS(j,w-w(j)),KS(j-1,w))
- Base cases: KS(j,0)=0 for all j, and KS(0,w)=0 for all w
- How to order the subproblems?
  - To determine KS(j,w), we need to know KS(j,w-w[j]) and KS(j-1,w)



The knapsack problem

```
Knapsack(w[1…n],v[1…n],W)
KS(j,0):=0 for all j
KS(0,w):=0 for all w
for w from 1 to W
    for j from 1 to n
        KS(j,w):= max(KS(j,w-w(j)),KS(j-1,w))
Return KS(n,W)
```

Base cases

Ordering

Next lecture

- Dynamic programming
  - Reading: Chapter 6