

Dynamic Programming

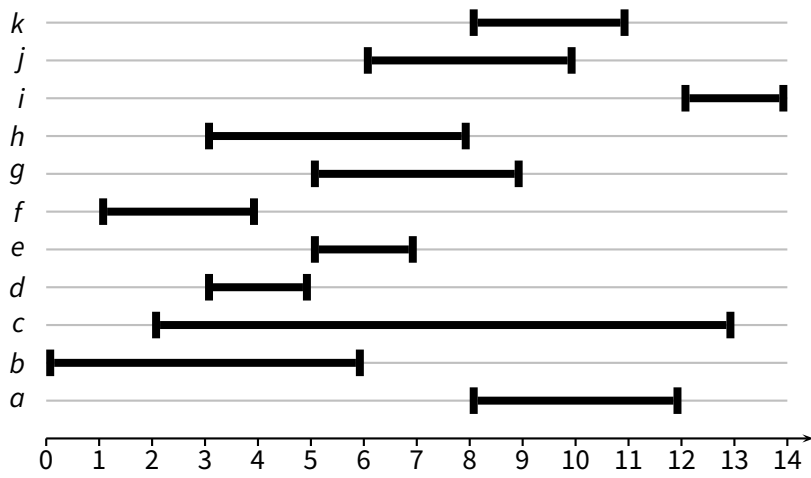
Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

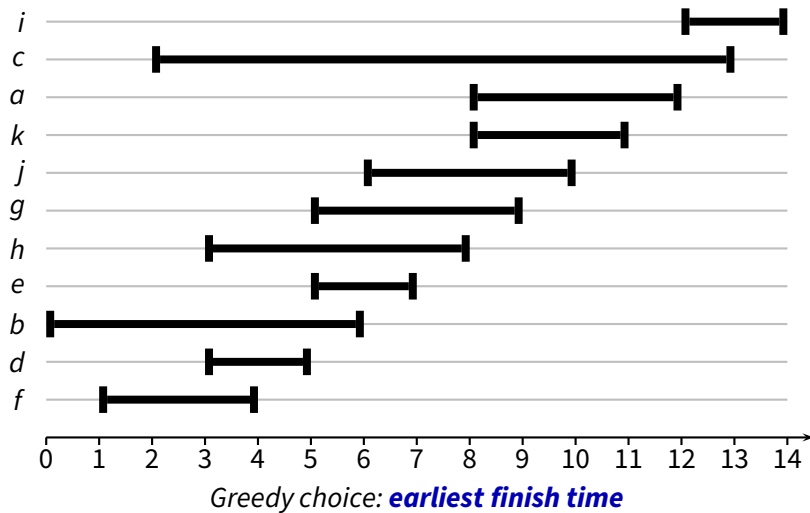
May 19, 2025

- Examples
- Dynamic programming strategy
- More examples

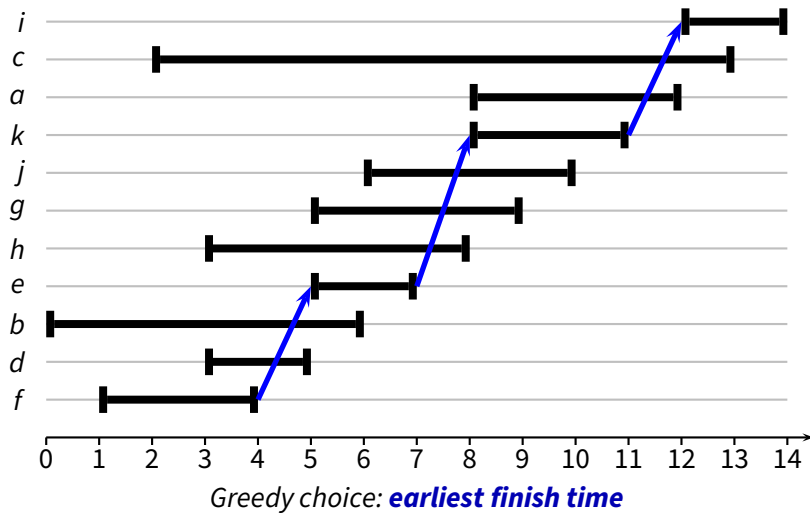
Activity-Selection Problem



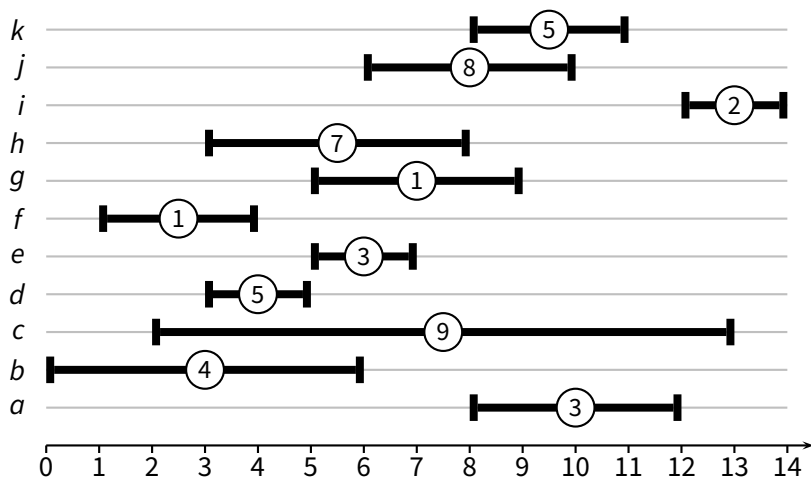
Activity-Selection Problem



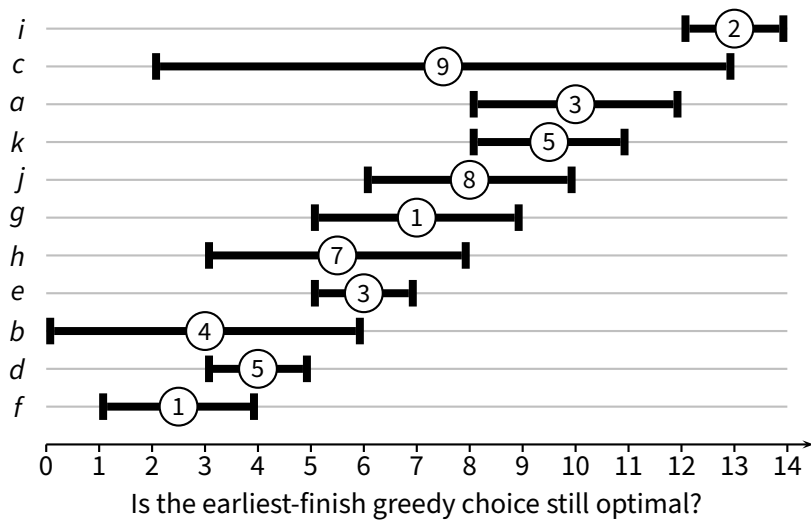
Activity-Selection Problem



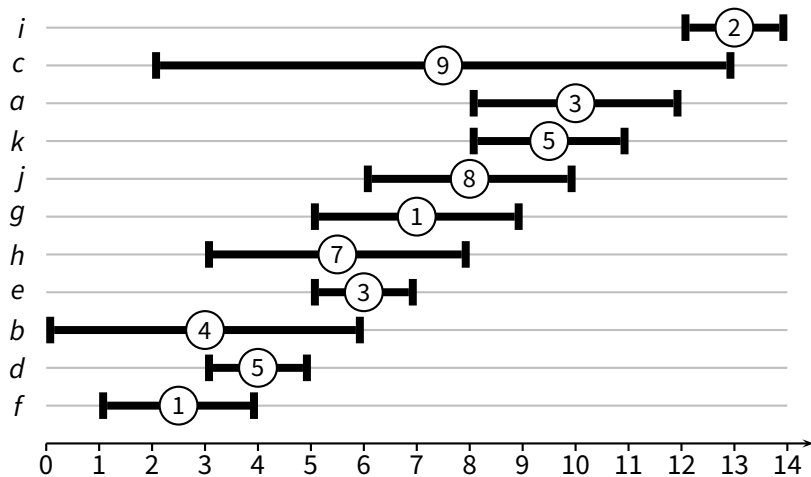
Weighted Activity-Selection Problem



Weighted Activity-Selection Problem

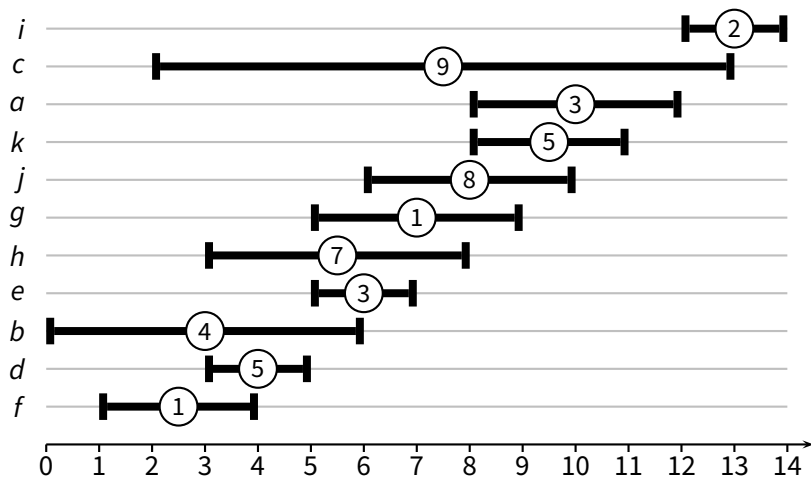


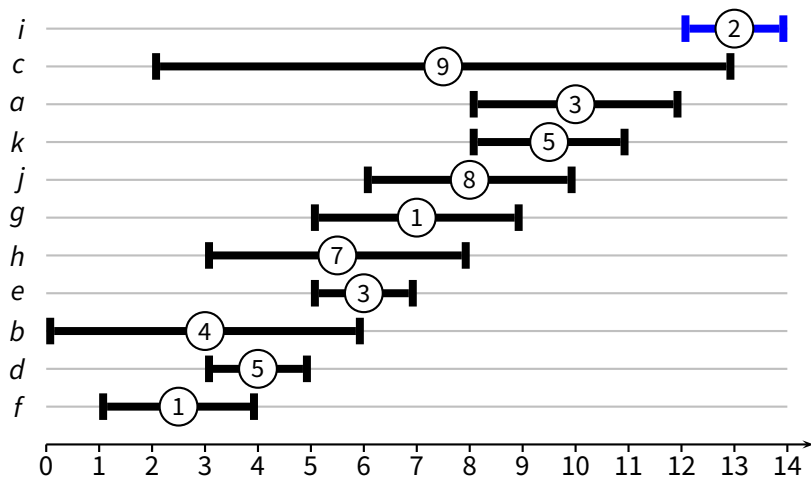
Weighted Activity-Selection Problem



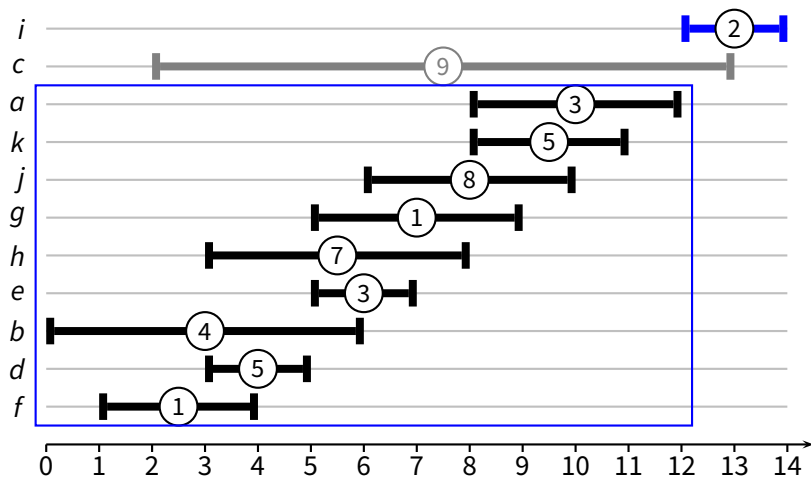
Is the earliest-finish greedy choice still optimal?

Is any greedy choice optimal?

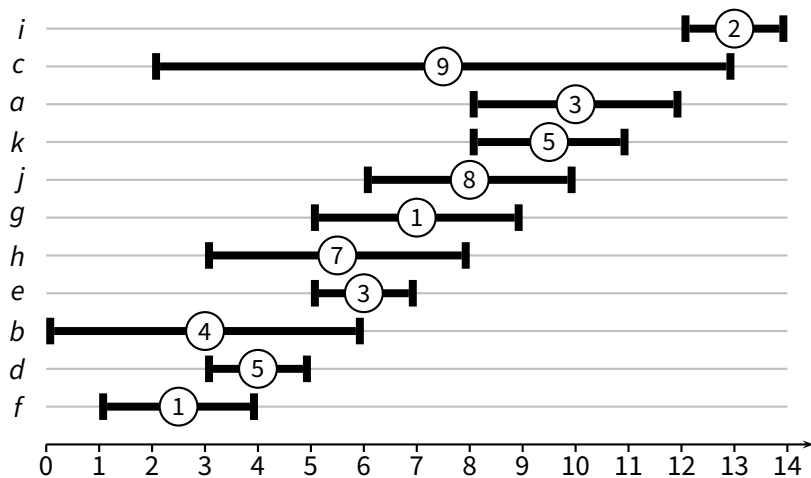


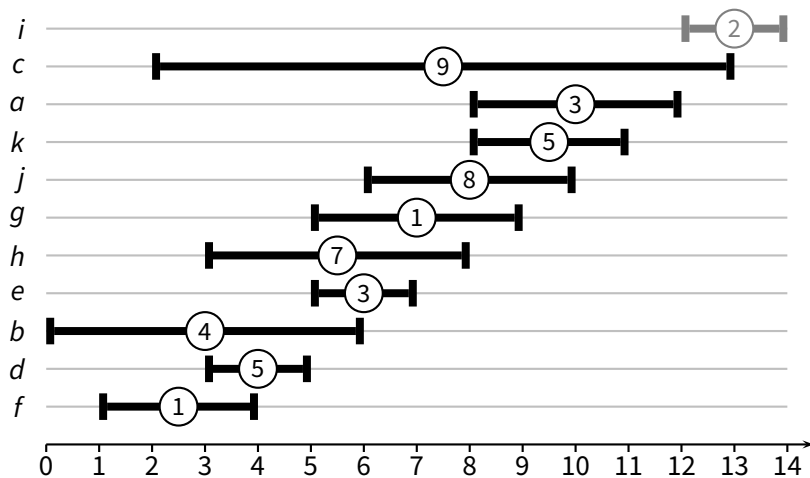


Case 1: activity *i* is in the optimal schedule

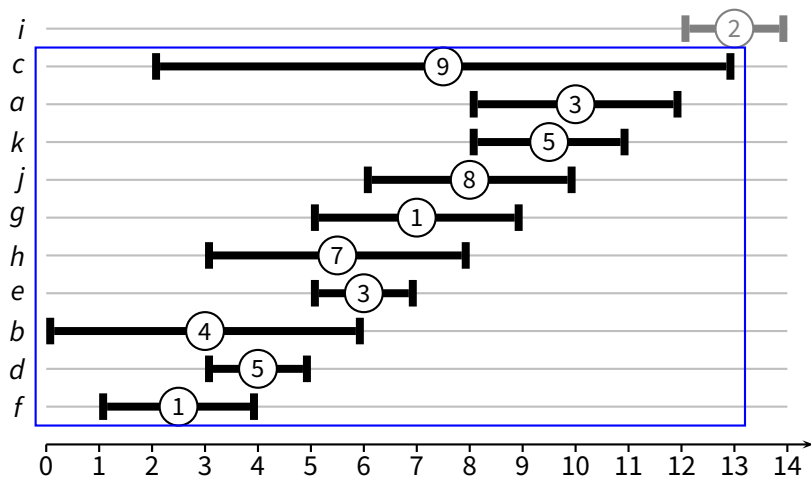


Case 1: activity i is in the optimal schedule





Case 2: activity *i* is not in the optimal schedule



Case 2: activity *i* is not in the optimal schedule

Bellman-Ford Algorithm

- Given a graph $G = (V, E)$ and a weight function w , we compute the shortest distance $D_u(v)$, from $u \in V$ to $v \in V$, using the *Bellman-Ford equation*

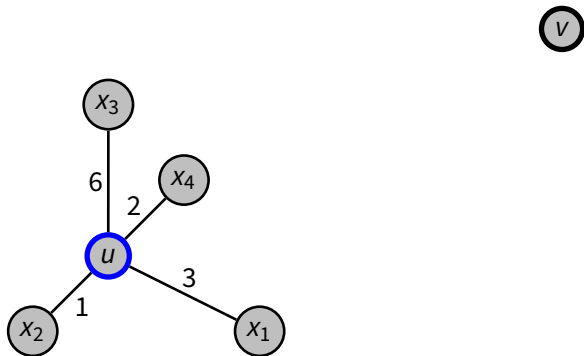
- Given a graph $G = (V, E)$ and a weight function w , we compute the shortest distance $D_u(v)$, from $u \in V$ to $v \in V$, using the *Bellman-Ford equation*

$$D_u(v) = \min_{x \in \text{Adj}(u)} [w(u, x) + D_x(v)]$$

Bellman-Ford Algorithm

- Given a graph $G = (V, E)$ and a weight function w , we compute the shortest distance $D_u(v)$, from $u \in V$ to $v \in V$, using the *Bellman-Ford equation*

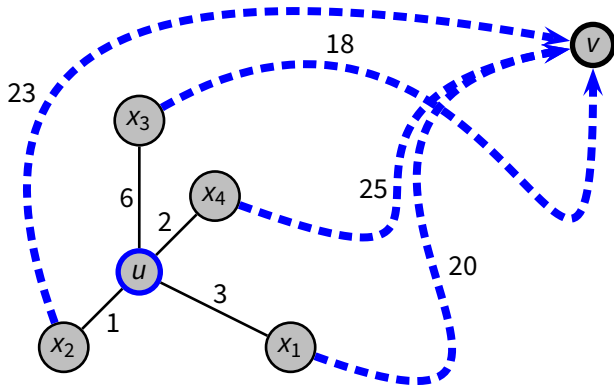
$$D_u(v) = \min_{x \in \text{Adj}(u)} [w(u, x) + D_x(v)]$$



Bellman-Ford Algorithm

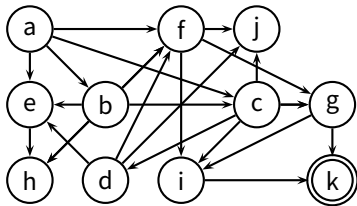
- Given a graph $G = (V, E)$ and a weight function w , we compute the shortest distance $D_u(v)$, from $u \in V$ to $v \in V$, using the *Bellman-Ford equation*

$$D_u(v) = \min_{x \in \text{Adj}(u)} [w(u, x) + D_x(v)]$$



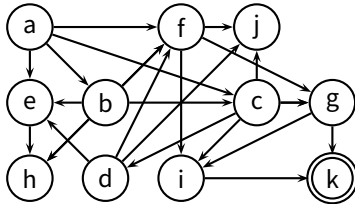
Shortest Paths on DAGs

- Given a *directed acyclic graph* $G = (V, E)$, this one with unit weights, find the shortest distances to a given node

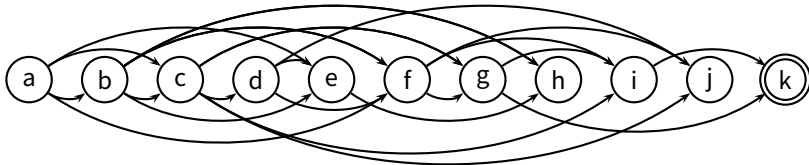


Shortest Paths on DAGs

- Given a *directed acyclic graph* $G = (V, E)$, this one with unit weights, find the shortest distances to a given node



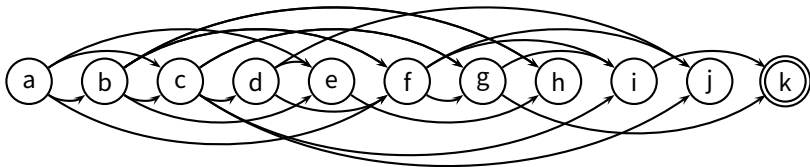
- Considering V in **topological order**...



Shortest Paths on DAGs (2)

- Considering V in *topological order*

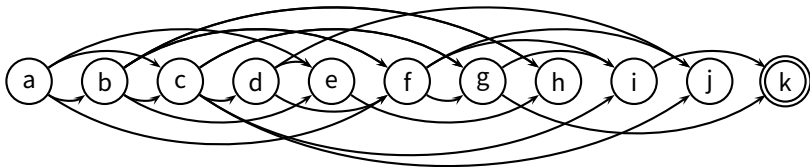
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$

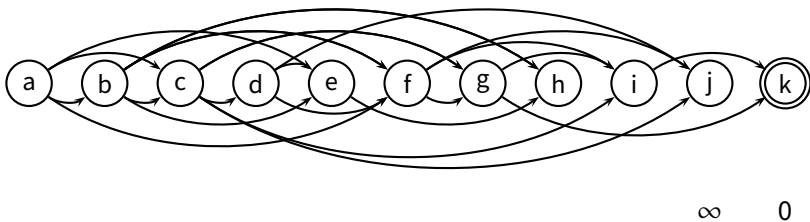


0

Shortest Paths on DAGs (2)

- Considering V in *topological order*

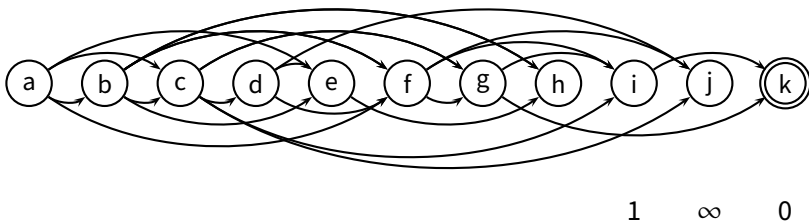
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

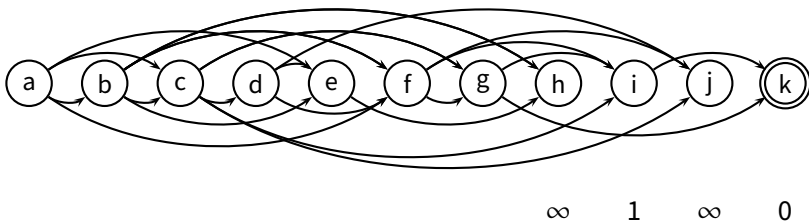
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

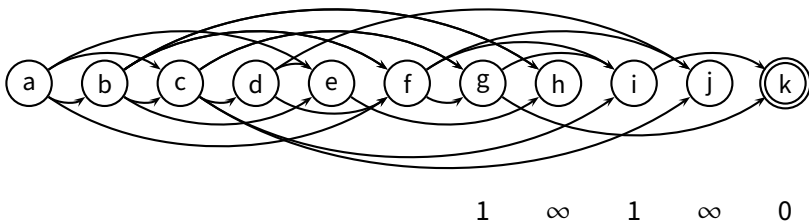
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

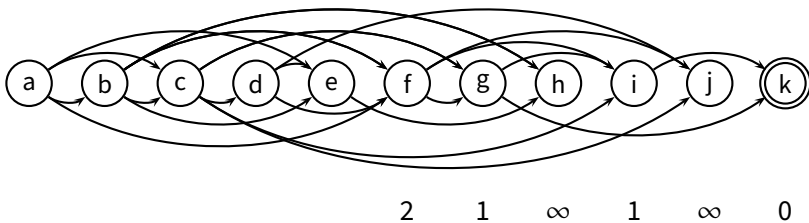
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

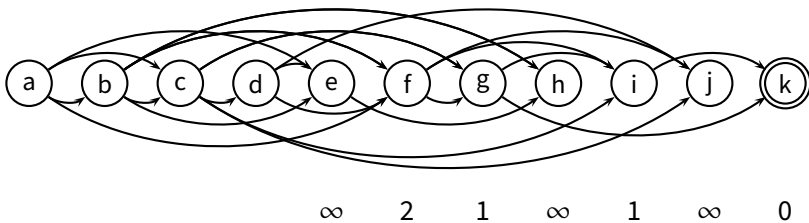
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

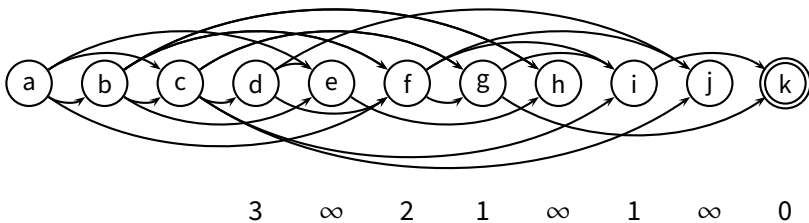
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

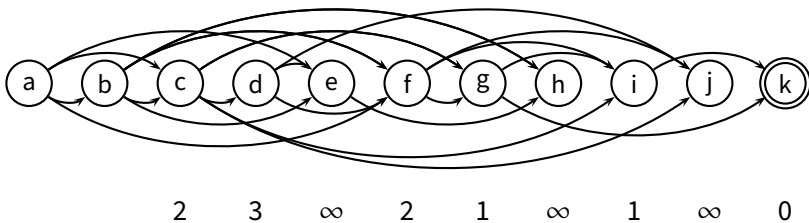
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

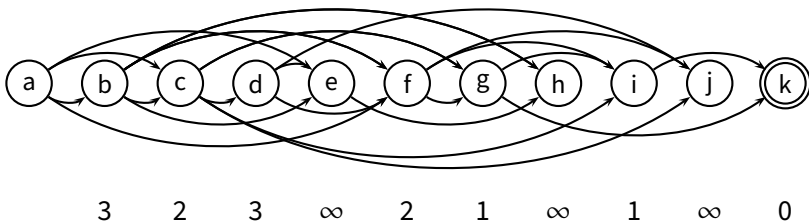
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

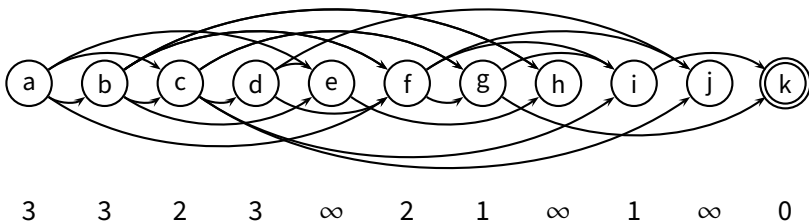
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

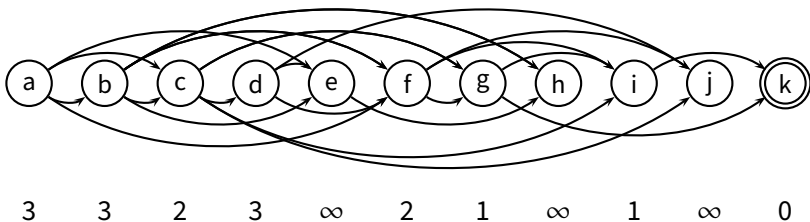
$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



Shortest Paths on DAGs (2)

- Considering V in *topological order*

$$D_x(k) = \min_{y \in \text{Adj}(x)} [w(x, y) + D_y(k)]$$



- Since G is a DAG, computing D_y with $y \in \text{Adj}(x)$ can be considered a *subproblem* of computing D_x
 - ▶ we build the solution bottom-up, storing the subproblem solutions

Longest Increasing Subsequence

Longest Increasing Subsequence

- Given a sequence of numbers a_1, a_2, \dots, a_n , an *increasing subsequence* is any subset $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ such that $1 \leq i_1 < i_2 < \dots < i_k \leq n$, and such that

$$a_{i_1} < a_{i_2} < \dots < a_{i_k}$$

- You must find the *longest increasing subsequence*

Longest Increasing Subsequence

- Given a sequence of numbers a_1, a_2, \dots, a_n , an *increasing subsequence* is any subset $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ such that $1 \leq i_1 < i_2 < \dots < i_k \leq n$, and such that

$$a_{i_1} < a_{i_2} < \dots < a_{i_k}$$

- You must find the *longest increasing subsequence*
- **Example:** find (one of) the longest increasing subsequence in

5 2 8 6 3 6 9 7

Longest Increasing Subsequence

- Given a sequence of numbers a_1, a_2, \dots, a_n , an *increasing subsequence* is any subset $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ such that $1 \leq i_1 < i_2 < \dots < i_k \leq n$, and such that

$$a_{i_1} < a_{i_2} < \dots < a_{i_k}$$

- You must find the *longest increasing subsequence*
- **Example:** find (one of) the longest increasing subsequence in

5 2 8 6 3 6 9 7

A maximal-length subsequence is

2 3 6 9

Longest Increasing Subsequence (2)

- *Intuition:* let $L(j)$ be the length of the longest subsequence ending at a_j

Longest Increasing Subsequence (2)

■ *Intuition:* let $L(j)$ be the length of the longest subsequence ending at a_j

▶ e.g., in

5 2 8 6 3 6 9 7

we have

$$L(4) = 2$$

Longest Increasing Subsequence (2)

■ *Intuition:* let $L(j)$ be the length of the longest subsequence ending at a_j

▶ e.g., in

5 2 8 6 3 6 9 7

we have

$$L(4) = 2$$

▶ this is our *subproblem structure*

Longest Increasing Subsequence (2)

- *Intuition*: let $L(j)$ be the length of the longest subsequence ending at a_j

- ▶ e.g., in

5 2 8 6 3 6 9 7

we have

$$L(4) = 2$$

- ▶ this is our *subproblem structure*

- Combining the subproblems

$$L(j) = 1 + \max\{L(i) \mid i < j \wedge a_i < a_j\}$$

- First, the name “dynamic programming”
 - ▶ does not mean writing a computer program
 - ▶ term used in the 1950s, when “programming” meant “planning”

- First, the name “dynamic programming”
 - ▶ does not mean writing a computer program
 - ▶ term used in the 1950s, when “programming” meant “planning”

- Problem domain
 - ▶ typically *optimization* problems
 - ▶ longest sequence, shortest path, etc.

- First, the name “dynamic programming”
 - ▶ does not mean writing a computer program
 - ▶ term used in the 1950s, when “programming” meant “planning”
- Problem domain
 - ▶ typically *optimization* problems
 - ▶ longest sequence, shortest path, etc.
- General strategy

- First, the name “dynamic programming”
 - ▶ does not mean writing a computer program
 - ▶ term used in the 1950s, when “programming” meant “planning”
- Problem domain
 - ▶ typically *optimization* problems
 - ▶ longest sequence, shortest path, etc.
- General strategy
 - ▶ decompose a problem in (smaller) ***subproblems***

- First, the name “dynamic programming”
 - ▶ does not mean writing a computer program
 - ▶ term used in the 1950s, when “programming” meant “planning”
- Problem domain
 - ▶ typically *optimization* problems
 - ▶ longest sequence, shortest path, etc.
- General strategy
 - ▶ decompose a problem in (smaller) ***subproblems***
 - ▶ must satisfy the ***optimal substructure*** property

- First, the name “dynamic programming”
 - ▶ does not mean writing a computer program
 - ▶ term used in the 1950s, when “programming” meant “planning”
- Problem domain
 - ▶ typically *optimization* problems
 - ▶ longest sequence, shortest path, etc.
- General strategy
 - ▶ decompose a problem in (smaller) ***subproblems***
 - ▶ must satisfy the ***optimal substructure*** property
 - ▶ subproblems may overlap (indeed they should overlap!)

- First, the name “dynamic programming”
 - ▶ does not mean writing a computer program
 - ▶ term used in the 1950s, when “programming” meant “planning”
- Problem domain
 - ▶ typically *optimization* problems
 - ▶ longest sequence, shortest path, etc.
- General strategy
 - ▶ decompose a problem in (smaller) ***subproblems***
 - ▶ must satisfy the ***optimal substructure*** property
 - ▶ subproblems may overlap (indeed they should overlap!)
 - ▶ solve the subproblems

- First, the name “dynamic programming”
 - ▶ does not mean writing a computer program
 - ▶ term used in the 1950s, when “programming” meant “planning”
- Problem domain
 - ▶ typically *optimization* problems
 - ▶ longest sequence, shortest path, etc.
- General strategy
 - ▶ decompose a problem in (smaller) ***subproblems***
 - ▶ must satisfy the ***optimal substructure*** property
 - ▶ subproblems may overlap (indeed they should overlap!)
 - ▶ solve the subproblems
 - ▶ derive the solution from (one of) the solutions to the subproblems

- ***Unweighted shortest path:*** given $G = (V, E)$, find the length of the shortest path from u to v

- **Unweighted shortest path:** given $G = (V, E)$, find the length of the shortest path from u to v
 - ▶ decompose $u \rightsquigarrow v$ into $u \rightsquigarrow w \rightsquigarrow v$

- **Unweighted shortest path:** given $G = (V, E)$, find the length of the shortest path from u to v
 - ▶ decompose $u \rightsquigarrow v$ into $u \rightsquigarrow w \rightsquigarrow v$
 - ▶ easy to prove that, if $u \rightsquigarrow w \rightsquigarrow v$ is minimal, then $w \rightsquigarrow v$ is also minimal
 - ▶ this is the **optimal substructure property**

- **Unweighted shortest path:** given $G = (V, E)$, find the length of the shortest path from u to v
 - ▶ decompose $u \rightsquigarrow v$ into $u \rightsquigarrow w \rightsquigarrow v$
 - ▶ easy to prove that, if $u \rightsquigarrow w \rightsquigarrow v$ is minimal, then $w \rightsquigarrow v$ is also minimal
 - ▶ this is the **optimal substructure property**

- **Unweighted longest simple path:** given $G = (V, E)$, find the length of the longest simple (i.e., no cycles) path from u to v
 - ▶ we can also decompose $u \rightsquigarrow v$ into $u \rightsquigarrow w \rightsquigarrow v$
 - ▶ however, we can not prove that, if $u \rightsquigarrow w \rightsquigarrow v$ is maximal, then $w \rightsquigarrow v$ is also maximal

- **Unweighted shortest path:** given $G = (V, E)$, find the length of the shortest path from u to v
 - ▶ decompose $u \rightsquigarrow v$ into $u \rightsquigarrow w \rightsquigarrow v$
 - ▶ easy to prove that, if $u \rightsquigarrow w \rightsquigarrow v$ is minimal, then $w \rightsquigarrow v$ is also minimal
 - ▶ this is the **optimal substructure property**

- **Unweighted longest simple path:** given $G = (V, E)$, find the length of the longest simple (i.e., no cycles) path from u to v
 - ▶ we can also decompose $u \rightsquigarrow v$ into $u \rightsquigarrow w \rightsquigarrow v$
 - ▶ however, we can not prove that, if $u \rightsquigarrow w \rightsquigarrow v$ is maximal, then $w \rightsquigarrow v$ is also maximal
 - ▶ **exercise:** find a counter-example

Dynamic Programming vs. Divide-and-Conquer

- Divide-and-conquer is also about decomposing a problem into subproblems

Dynamic Programming vs. Divide-and-Conquer

- Divide-and-conquer is also about decomposing a problem into subproblems
- Divide-and-conquer works by breaking the problem into ***significantly smaller subproblems***
 - ▶ in dynamic programming, it is typical to reduce $L(j)$ into $L(j - 1)$
 - ▶ this is one reason why recursion does not work so well for dynamic programming

Dynamic Programming vs. Divide-and-Conquer

- Divide-and-conquer is also about decomposing a problem into subproblems
- Divide-and-conquer works by breaking the problem into ***significantly smaller subproblems***
 - ▶ in dynamic programming, it is typical to reduce $L(j)$ into $L(j - 1)$
 - ▶ this is one reason why recursion does not work so well for dynamic programming
- Divide-and-conquer splits the problem into ***independent subproblems***
 - ▶ in dynamic programming, subproblems typically overlap
 - ▶ pretty much the same argument as above

Dynamic Programming vs. Greedy

- Greedy: requires the *greedy-choice property*
 - ▶ greedy: *greedy choice* plus *one subproblem*
 - ▶ greedy choice typically *before* proceeding to the subproblem
 - ▶ no need to store the result of each subproblem

Dynamic Programming vs. Greedy

- Greedy: requires the ***greedy-choice property***
 - ▶ greedy: ***greedy choice*** plus ***one subproblem***
 - ▶ greedy choice typically *before* proceeding to the subproblem
 - ▶ no need to store the result of each subproblem
- Dynamic programming: ***more general***
 - ▶ does not need the greedy-choice property
 - ▶ typically looks at several subproblems
 - ▶ “dynamically” choose one of them to obtain a global solution
 - ▶ typically works bottom-up
 - ▶ typically reuses solutions of the subproblems

- Prefix/suffix subproblems

- ▶ *Input:* x_1, x_2, \dots, x_n
- ▶ *Subproblem:* x_1, x_2, \dots, x_i , with $i < n$
- ▶ $O(n)$ subproblems

■ Prefix/suffix subproblems

- ▶ *Input:* x_1, x_2, \dots, x_n
- ▶ *Subproblem:* x_1, x_2, \dots, x_i , with $i < n$
- ▶ $O(n)$ subproblems

■ Subsequence subproblems

- ▶ *Input:* x_1, x_2, \dots, x_n
- ▶ *Subproblem:* x_i, x_{i+1}, \dots, x_j , with $1 \leq i < j \leq n$

■ Prefix/suffix subproblems

- ▶ *Input:* x_1, x_2, \dots, x_n
- ▶ *Subproblem:* x_1, x_2, \dots, x_i , with $i < n$
- ▶ $O(n)$ subproblems

■ Subsequence subproblems

- ▶ *Input:* x_1, x_2, \dots, x_n
- ▶ *Subproblem:* x_i, x_{i+1}, \dots, x_j , with $1 \leq i < j \leq n$
- ▶ $O(n^2)$ subproblems

- Given two strings x and y , find the *smallest set of edit operations* that transform x into y

- Given two strings x and y , find the *smallest set of edit operations* that transform x into y
 - ▶ edit operations: *delete*, *insert*, and *modify* a single character
 - ▶ very important applications
 - ▶ spell checker
 - ▶ DNA sequencing

- Given two strings x and y , find the *smallest set of edit operations* that transform x into y
 - ▶ edit operations: *delete*, *insert*, and *modify* a single character
 - ▶ very important applications
 - ▶ spell checker
 - ▶ DNA sequencing

- **Example:** transform “Lugano” into “Zurigo”

- Given two strings x and y , find the *smallest set of edit operations* that transform x into y
 - ▶ edit operations: *delete*, *insert*, and *modify* a single character
 - ▶ very important applications
 - ▶ spell checker
 - ▶ DNA sequencing

- **Example:** transform “Lugano” into “Zurigo”

L u g a n o

Z u r i g o

- Given two strings x and y , find the *smallest set of edit operations* that transform x into y
 - ▶ edit operations: *delete*, *insert*, and *modify* a single character
 - ▶ very important applications
 - ▶ spell checker
 - ▶ DNA sequencing

■ **Example:** transform “Lugano” into “Zurigo”

⇓		+	+		-	-	
Z		r	i				
L	u			g	a	n	o
Z	u	r	i	g			o

- Given two strings x and y , find the *smallest set of edit operations* that transform x into y
 - ▶ edit operations: *delete*, *insert*, and *modify* a single character
 - ▶ very important applications
 - ▶ spell checker
 - ▶ DNA sequencing

■ **Example:** transform “Lugano” into “Zurigo”

⇓	+	+	-	-									
Z	r	i											
L	u		g	a	n	o		L	u	g	a	n	o
Z	u	r	i	g		o		Z	u	r	i	g	o

- Given two strings x and y , find the *smallest set of edit operations* that transform x into y
 - ▶ edit operations: *delete*, *insert*, and *modify* a single character
 - ▶ very important applications
 - ▶ spell checker
 - ▶ DNA sequencing

■ **Example:** transform “Lugano” into “Zurigo”

	↓		+	+		-	-	
Z		r		i				
L	u				g	a	n	o
Z	u	r	i	g				o

	↓		↓	↓	↓
Z		r		i	g
L	u	g	a	n	o
Z	u	r	i	g	o

- Align the two strings x and y , possibly inserting “gaps” between letters
 - ▶ a gap in the source means *insertion*
 - ▶ a gap in the destination means *deletion*
 - ▶ two different character in the same position means *modification*

- Align the two strings x and y , possibly inserting “gaps” between letters
 - ▶ a gap in the source means *insertion*
 - ▶ a gap in the destination means *deletion*
 - ▶ two different character in the same position means *modification*
- Many alignments are possible; the alignment with the smallest number of insertions, deletions, and modifications defines the *edit distance*

- Align the two strings x and y , possibly inserting “gaps” between letters
 - ▶ a gap in the source means *insertion*
 - ▶ a gap in the destination means *deletion*
 - ▶ two different character in the same position means *modification*
- Many alignments are possible; the alignment with the smallest number of insertions, deletions, and modifications defines the *edit distance*
- So, how do we solve this problem?

- Align the two strings x and y , possibly inserting “gaps” between letters
 - ▶ a gap in the source means *insertion*
 - ▶ a gap in the destination means *deletion*
 - ▶ two different character in the same position means *modification*
- Many alignments are possible; the alignment with the smallest number of insertions, deletions, and modifications defines the *edit distance*
- So, how do we solve this problem?
- What are the subproblems?

- *Idea:* consider a prefix of x and a prefix of y

- *Idea:* consider a prefix of x and a prefix of y
- Let $E(i, j)$ be the smallest set of changes that turn the first i characters of x into the first j characters of y

- *Idea*: consider a prefix of x and a prefix of y
- Let $E(i, j)$ be the smallest set of changes that turn the first i characters of x into the first j characters of y
- Now, the last column of the alignment of $E(i, j)$ can have either
 - ▶ a gap for x (i.e., insertion)
 - ▶ a gap for y (i.e., deletion)
 - ▶ no gaps (i.e., modification iff $x[i] \neq y[j]$)

- *Idea*: consider a prefix of x and a prefix of y
- Let $E(i, j)$ be the smallest set of changes that turn the first i characters of x into the first j characters of y
- Now, the last column of the alignment of $E(i, j)$ can have either
 - ▶ a gap for x (i.e., insertion)
 - ▶ a gap for y (i.e., deletion)
 - ▶ no gaps (i.e., modification iff $x[i] \neq y[j]$)
- This suggests a way to combine the subproblems; let $diff(i, j) = 1$ iff $x[i] \neq y[j]$ or 0 otherwise

$$E(i, j) = \min\{1 + E(i - 1, j), \\ 1 + E(i, j - 1), \\ diff(i, j) + E(i - 1, j - 1)\}$$

■ Problem definition

- ▶ *Input*: a set of n objects with their weights w_1, w_2, \dots, w_n and their values v_1, v_2, \dots, v_n , and a maximum weight W
- ▶ *Output*: a subset K of the objects such that $\sum_{i \in K} w_i \leq W$ and such that $\sum_{i \in K} v_i$ is maximal

■ Problem definition

- ▶ *Input:* a set of n objects with their weights w_1, w_2, \dots, w_n and their values v_1, v_2, \dots, v_n , and a maximum weight W
- ▶ *Output:* a subset K of the objects such that $\sum_{i \in K} w_i \leq W$ and such that $\sum_{i \in K} v_i$ is maximal

■ Dynamic-programming solution

- ▶ let $K(w, j)$ be the maximum value achievable at maximum capacity w using the first j items (i.e., items $1 \dots j$)
- ▶ considering the j th element, we can either “use it or loose it,” so

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

- The breakdown of a problem into subproblem suggests the use of a recursive function. Is that a good idea?

- The breakdown of a problem into subproblem suggests the use of a recursive function. Is that a good idea?
 - ▶ No! As we already said, recursion doesn't quite work here

- The breakdown of a problem into subproblem suggests the use of a recursive function. Is that a good idea?
 - ▶ No! As we already said, recursion doesn't quite work here
 - ▶ Why?
- Remember the first algorithm of this course?

- The breakdown of a problem into subproblem suggests the use of a recursive function. Is that a good idea?
 - ▶ No! As we already said, recursion doesn't quite work here
 - ▶ Why?
- Remember the first algorithm of this course?

```
PINGALA( $n$ )
```

```
1 if  $n \leq 2$ 
```

```
2     return  $n$ 
```

```
3 return PINGALA( $n - 1$ ) + PINGALA( $n - 2$ )
```

- The breakdown of a problem into subproblem suggests the use of a recursive function. Is that a good idea?
 - ▶ No! As we already said, recursion doesn't quite work here
 - ▶ Why?
- Remember the first algorithm of this course?

```
PINGALA( $n$ )  
1  if  $n \leq 2$   
2      return  $n$   
3  return PINGALA( $n - 1$ ) + PINGALA( $n - 2$ )
```

- *Recursion solves the same problem over and over again*

- Problem: recursion solves the same problems repeatedly
- **Idea:** “cache” the results

- Problem: recursion solves the same problems repeatedly
- **Idea:** “cache” the results

```
PINGALA( $n$ )  
1  if  $n \leq 2$   
2      return  $n$   
3  if  $(n, x) \in H$  // a hash table  $H$  “caches” results  
4      return  $x$   
5  else  $x = \mathbf{PINGALA}(n - 1) + \mathbf{FIBONACCI}(n - 2)$   
6      INSERT( $H, n, x$ )  
7      return  $x$ 
```

- Idea also known as *memoization*

■ *Greedy*

1. start with the greedy choice
2. add the solution to the remaining subproblem

A nice tail-recursive algorithm

■ *Greedy*

1. start with the greedy choice
2. add the solution to the remaining subproblem

A nice tail-recursive algorithm

- ▶ the complexity of the greedy strategy *per-se* is $\Theta(n)$

■ *Greedy*

1. start with the greedy choice
2. add the solution to the remaining subproblem

A nice tail-recursive algorithm

- ▶ the complexity of the greedy strategy *per-se* is $\Theta(n)$

■ *Dynamic programming*

1. break down the problem in subproblems

■ **Greedy**

1. start with the greedy choice
2. add the solution to the remaining subproblem

A nice tail-recursive algorithm

- ▶ the complexity of the greedy strategy *per-se* is $\Theta(n)$

■ **Dynamic programming**

1. break down the problem in subproblems— $O(1)$, $O(n)$, $O(n^2)$, ... subproblems

■ *Greedy*

1. start with the greedy choice
2. add the solution to the remaining subproblem

A nice tail-recursive algorithm

- ▶ the complexity of the greedy strategy *per-se* is $\Theta(n)$

■ *Dynamic programming*

1. break down the problem in subproblems— $O(1)$, $O(n)$, $O(n^2)$, ... subproblems
2. you solve the main problem by *choosing* one of the subproblems

■ **Greedy**

1. start with the greedy choice
2. add the solution to the remaining subproblem

A nice tail-recursive algorithm

- ▶ the complexity of the greedy strategy *per-se* is $\Theta(n)$

■ **Dynamic programming**

1. break down the problem in subproblems— $O(1)$, $O(n)$, $O(n^2)$, ... subproblems
2. you solve the main problem by *choosing* one of the subproblems
3. in practice, solve the subproblems bottom-up

- **Puzzle 0:** is it possible to insert some '+' signs in the string "213478" so that the resulting expression would equal 214?

- **Puzzle 0:** is it possible to insert some '+' signs in the string "213478" so that the resulting expression would equal 214?
 - ▶ Yes, because $2 + 134 + 78 = 214$
- **Puzzle 1:** is it possible to insert some '+' signs in the strings of digits to obtain the corresponding target number?

<i>digits</i>	<i>target</i>
646805736141599100791159198	472004
6152732017763987430884029264512187586207273294807	560351
48796142803774467559157928	326306
195961521219109124054410617072018922584281838218	7779515