# Dynamic Programming

Antonio Carzaniga

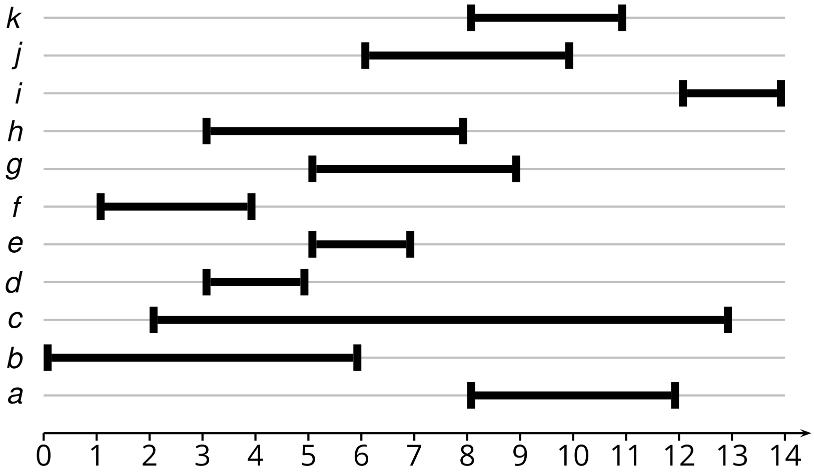
Faculty of Informatics
Università della Svizzera italiana

May 25, 2021

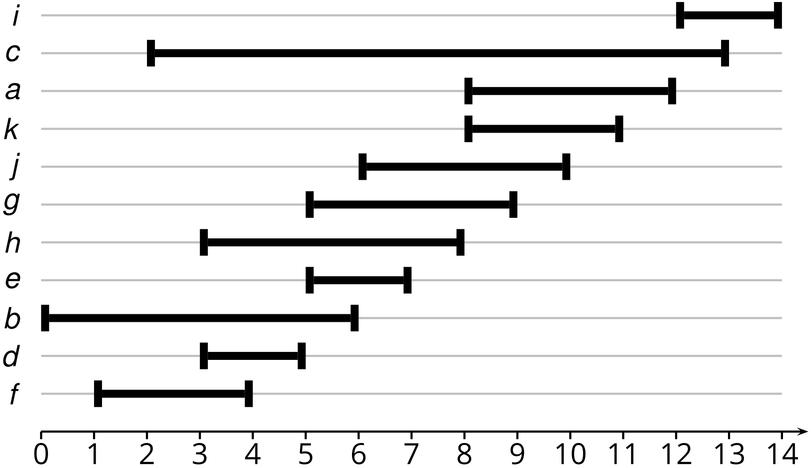
# Outline

- Examples
- Dynamic programming strategy
- More examples

# Activity-Selection Problem

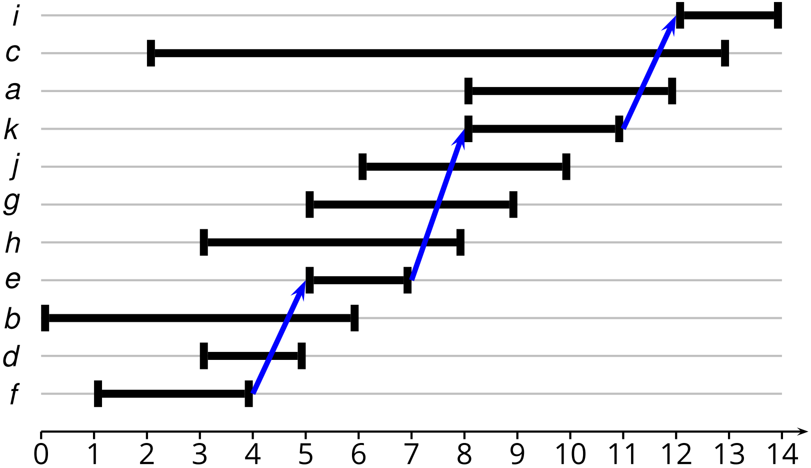


# Activity-Selection Problem



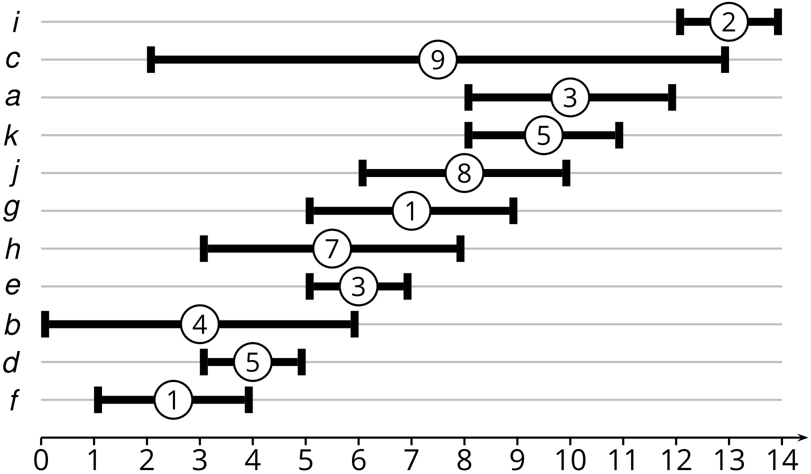

Greedy choice: ***earliest finish time***

# Activity-Selection Problem

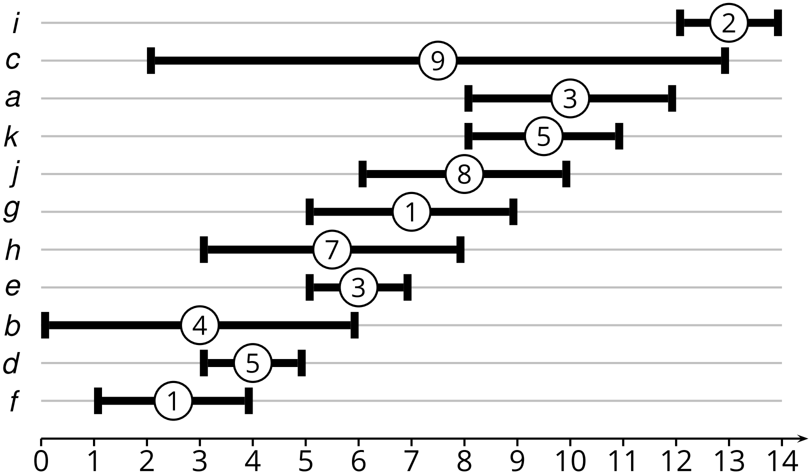



Greedy choice: ***earliest finish time***

# *Weighted* Activity-Selection Problem

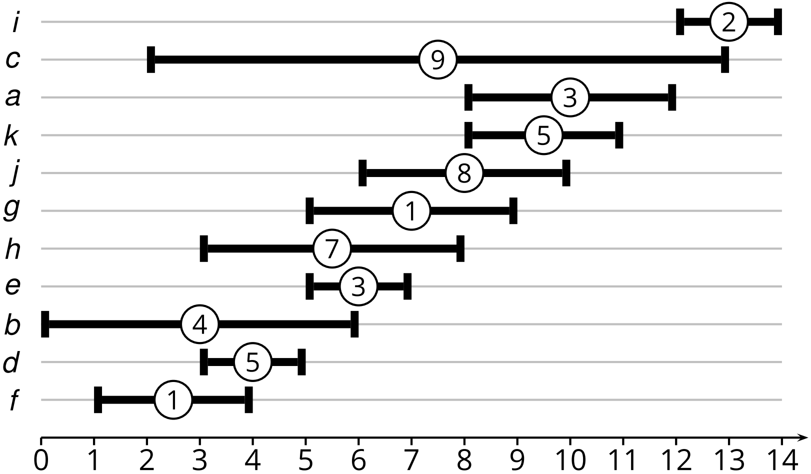


# *Weighted* Activity-Selection Problem




Is the earliest-finish greedy choice still optimal?

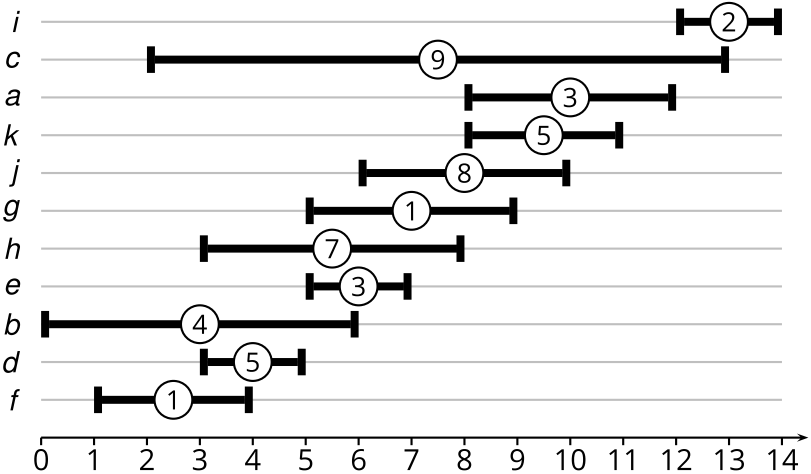
# *Weighted* Activity-Selection Problem

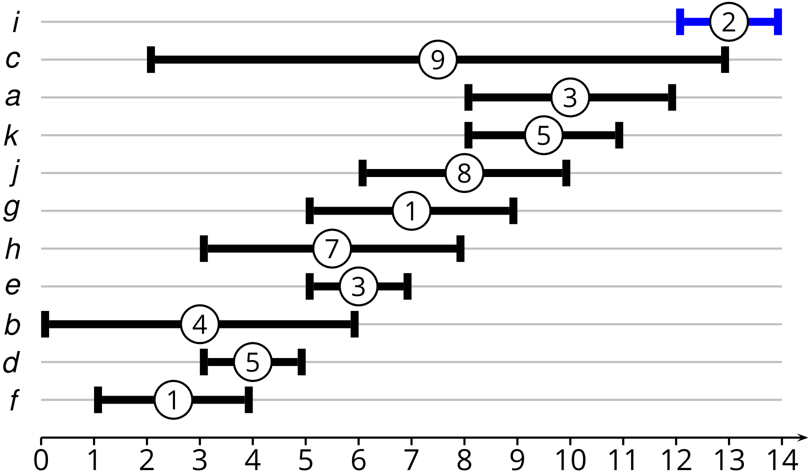



Is the earliest-finish greedy choice still optimal?

Is *any* greedy choice optimal?

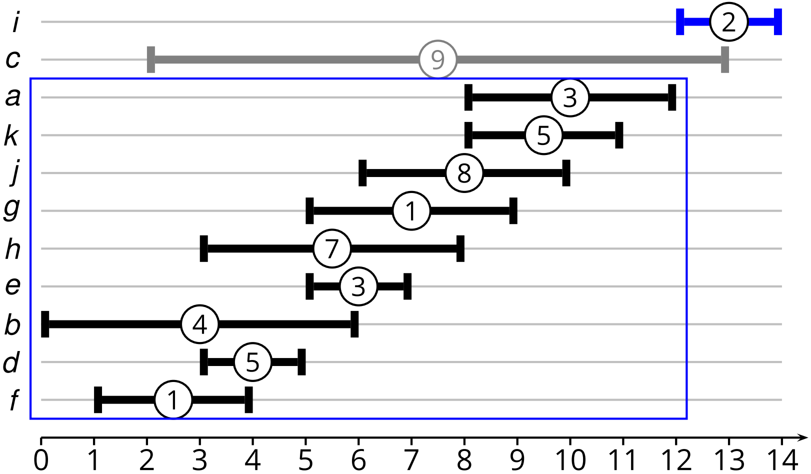
**Case 1**





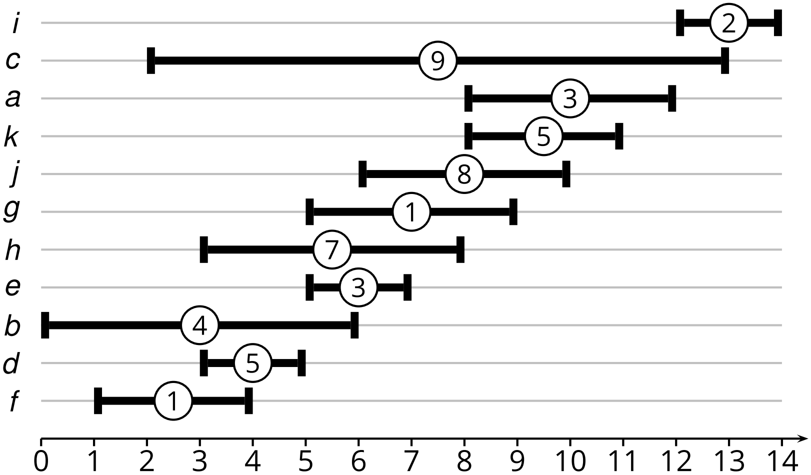

***Case 1:*** activity *i* is in the optimal schedule

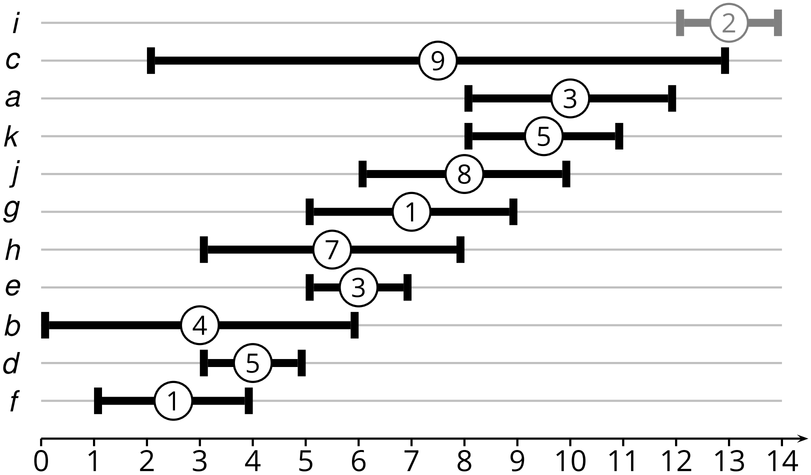
# Case 1




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

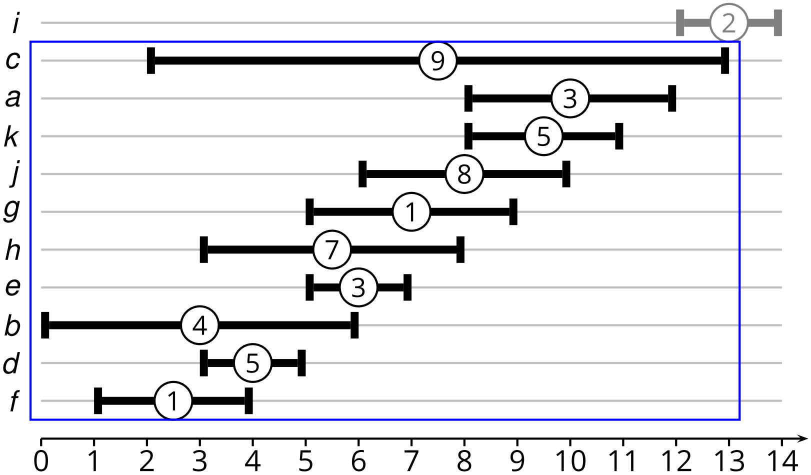
**Case 2**






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

# Case 2




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

# Bellman-Ford Algorithm

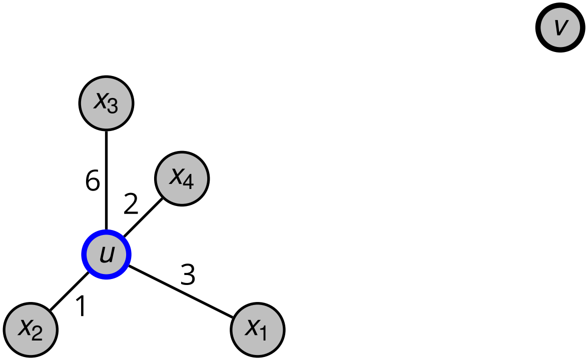
- Given a graph $G = (V, E)$ and a weight function $w$, we compute the shortest distance $D_u(v)$, from $u \in V$ to $v \in V$, using the *Bellman-Ford equation*

- 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*

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

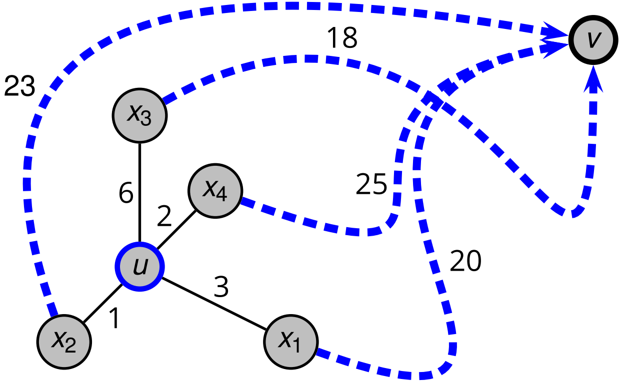
- Given a graph $G = (V, E)$ and a weight function $w$, we compute the shortest distance $D_u(v)$, from $u \in V$ to $v \in V$, using the *Bellman-Ford equation*

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

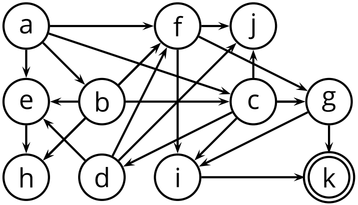


- Given a graph $G = (V, E)$ and a weight function $w$, we compute the shortest distance $D_u(v)$, from $u \in V$ to $v \in V$, using the *Bellman-Ford equation*

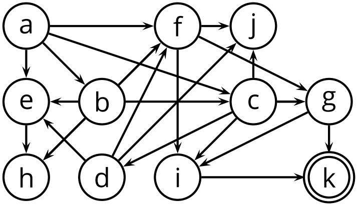
$$D_u(v) = \min_{x \in Adj(u)} [w(u, x) + D_x(v)]$$



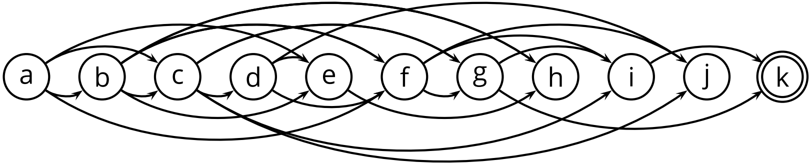
- Given a *directed acyclic graph* $G = (V, E)$, this one with unit weights, find the shortest distances to a given node



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

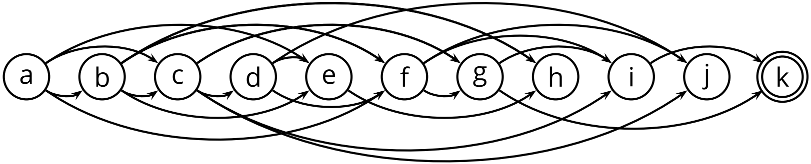



- Considering $V$ in ***topological order***…



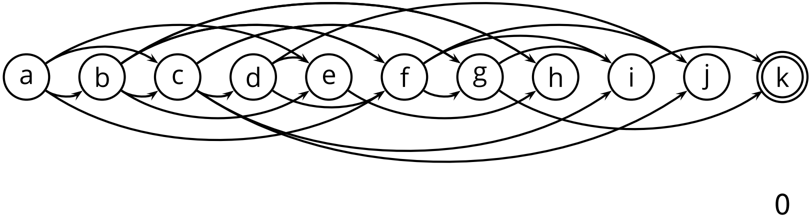
- Considering $V$ in *topological order*

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



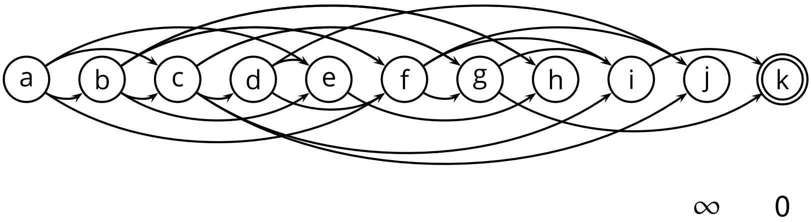
- Considering $V$ in *topological order*

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



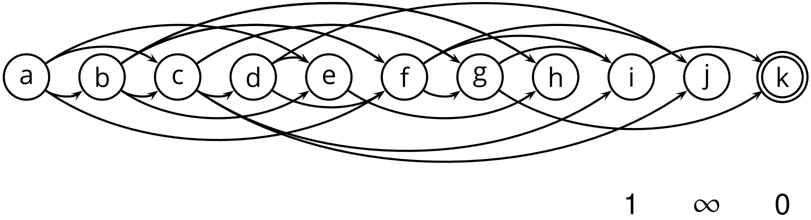
- Considering $V$ in *topological order*

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



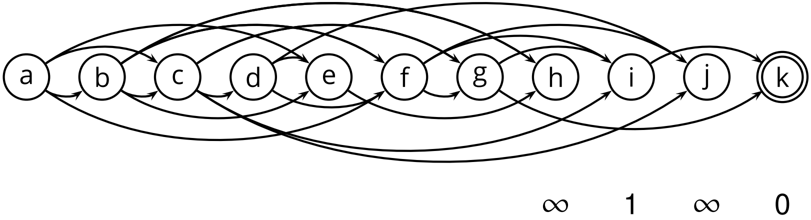
- Considering $V$ in *topological order*

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



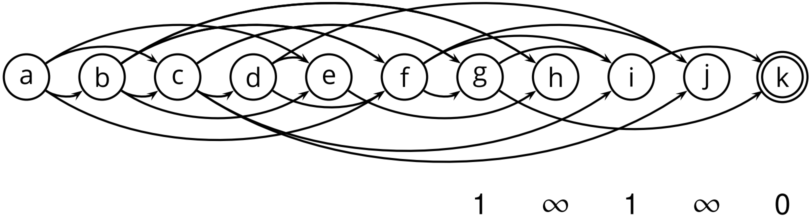
- Considering $V$ in *topological order*

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



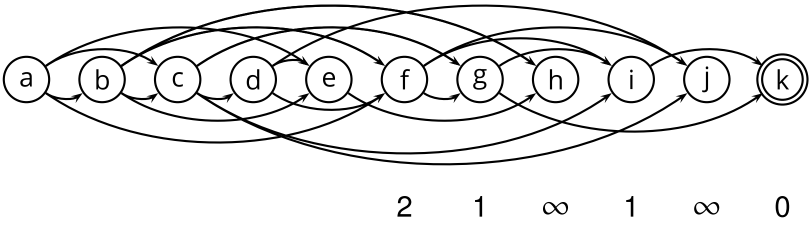
- Considering $V$ in *topological order*

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



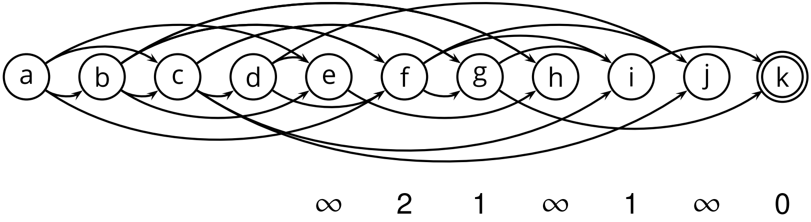
- Considering $V$ in *topological order*

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



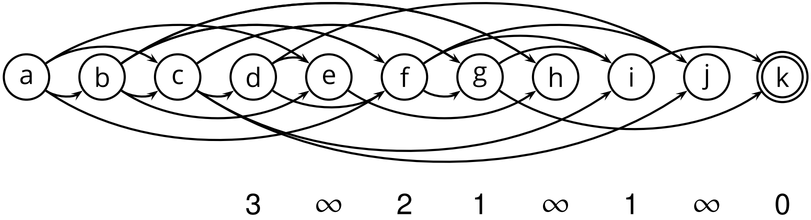
- Considering $V$ in *topological order*

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



- Considering $V$ in *topological order*

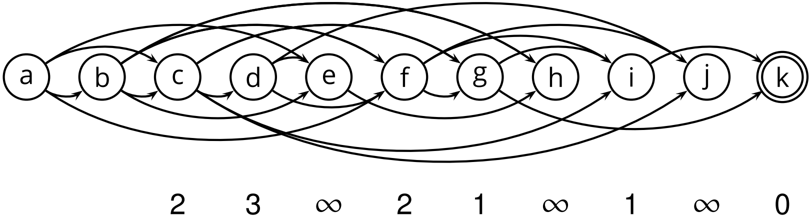
$$D_x(k) = \min_{y \in Adj(x)} [w(x,y) + D_y(k)]$$



# Shortest Paths on DAGs (2)

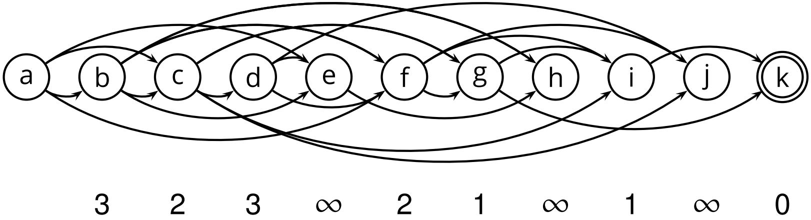
- Considering $V$ in *topological order*

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



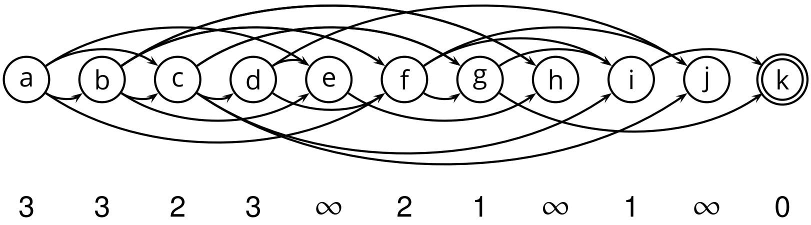
- Considering $V$ in *topological order*

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



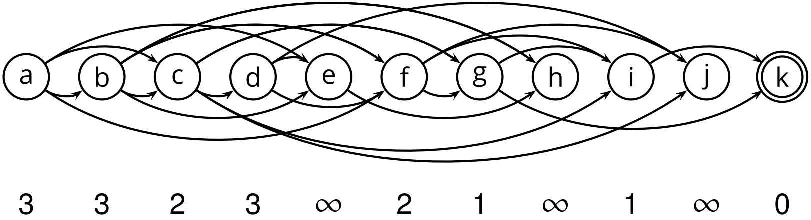
- Considering $V$ in *topological order*

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



- Considering $V$ in *topological order*

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




- Since $G$ is a DAG, computing $D_y$ with $y \in Adj(x)$ 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 $a_1, a_2, \ldots, a_n$, an *increasing subsequence* is any subset $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and such that

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

- You must find the *longest increasing subsequence*

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

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

- You must find the *longest increasing subsequence*

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

$$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

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

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

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

$$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

A maximal-length subsequence is

$$2 \quad 3 \quad 6 \quad 9$$

# Longest Increasing Subsequence (2)

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

- *Intuition:* let $L(j)$ be the length of the longest subsequence ending at $a_j$
  - e.g., in

    $$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

    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

    $$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

    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

    $$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

    we have

    $$L(4) = 2$$

  - this is our *subproblem structure*

- Combining the subproblems

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

# Dynamic Programming

- 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

# Dynamic Programming

- 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***

# Dynamic Programming

- 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

# Dynamic Programming

- 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

# Dynamic Programming

- 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 $u \leadsto v$ into $u \leadsto w \leadsto v$
  - easy to prove that, if $u \leadsto w \leadsto v$ is minimal, then $w \leadsto v$ 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 $u \leadsto v$ into $u \leadsto w \leadsto v$
  - however, we can not prove that, if $u \leadsto w \leadsto v$ is maximal, then $w \leadsto v$ 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

# Typical Subproblem Structures

- Prefix/suffix subproblems
  - *Input:* $x_1, x_2, \ldots, x_n$
  - *Subproblem:* $x_1, x_2, \ldots, x_i$, with $i < n$
  - $O(n)$ subproblems

# Typical Subproblem Structures

- Prefix/suffix subproblems
  - *Input:* $x_1, x_2, \ldots, x_n$
  - *Subproblem:* $x_1, x_2, \ldots, x_i$, with $i < n$
  - $O(n)$ subproblems

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

# Typical Subproblem Structures

- Prefix/suffix subproblems
  - *Input:* $x_1, x_2, \ldots, x_n$
  - *Subproblem:* $x_1, x_2, \ldots, x_i$, with $i < n$
  - $O(n)$ subproblems

- Subsequence subproblems
  - *Input:* $x_1, x_2, \ldots, x_n$
  - *Subproblem:* $x_i, x_{i+1}, \ldots, x_j$, 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 “Programming Algorithms” into “Jazayeri”

# Edit Distance

- 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 “Programming Algorithms” into “Jazayeri”

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| C | a | r | z | a | n | | | i | g | a |
| J | a | | z | a | y | e | r | i | | |

# Edit Distance

- 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 “Programming Algorithms” into “Jazayeri”

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| ⇓ | | – | | | ⇓ | + | + | | – | – |
| J | | | | | y | e | r | | | |
| C | a | r | z | a | n | | | i | g | a |
| | | | | | | | | | | |
| J | a | | z | a | y | e | r | i | | |

# Edit Distance (2)

- 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*

# Edit Distance (2)

- 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*

# Edit Distance (2)

- 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?

# Edit Distance (2)

- 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?

# Edit Distance (3)

- *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 $\mathit{diff}(i,j) = 1$ iff $x[i] \neq y[j]$ or 0 otherwise

$$
\begin{aligned}
E(i,j) = \min\{&1 + E(i-1,j), \\
&1 + E(i,j-1), \\
&\mathit{diff}(i,j) + E(i-1,j-1)\}
\end{aligned}
$$

- Problem definition
  - *Input:* a set of $n$ objects with their weights $w_1, w_2, \ldots w_n$ and their values $v_1, v_2, \ldots v_n$, 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 $w_1, w_2, \ldots w_n$ and their values $v_1, v_2, \ldots v_n$, 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 $1 \ldots j$)
- considering the $j$th element, we can either "use it or loose it," so

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

# Recursion?

- 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

# Recursion?

- 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 Fibonacci?

# Recursion?

- 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 Fibonacci?

```
Fibonacci(n)
if n == 0
        return 0
elseif n == 1
        return 1
else return Fibonacci(n − 1) + Fibonacci(n − 2)
```

# Recursion?

- 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 Fibonacci?

```
Fibonacci(n)
if n == 0
        return 0
elseif n == 1
        return 1
else return Fibonacci(n − 1) + Fibonacci(n − 2)
```

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

# Memoization

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

# Memoization

- Problem: recursion solves the same problems repeatedly

- **Idea:** “cache” the results

```
Fibonacci(n)
if n == 0
        return 0
elseif n == 1
        return 1
elseif (n, x) ∈ H // a hash table H “caches” results
        return x
else x = Fibonacci(n − 1) + Fibonacci(n − 2)
        Insert(H, n, x)
        return x
```

- Idea also known as *memoization*

# Complexity

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

# Exercise

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

# Exercise

- **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?

| *digits* | *target* |
|---|---:|
| 646805736141599100791159198 | 472004 |
| 6152732017763987430884029264512187586207273294807 | 560351 |
| 48796142803774467559157928 | 326306 |
| 195961521219109124054410617072018922584281838218 | 7779515 |