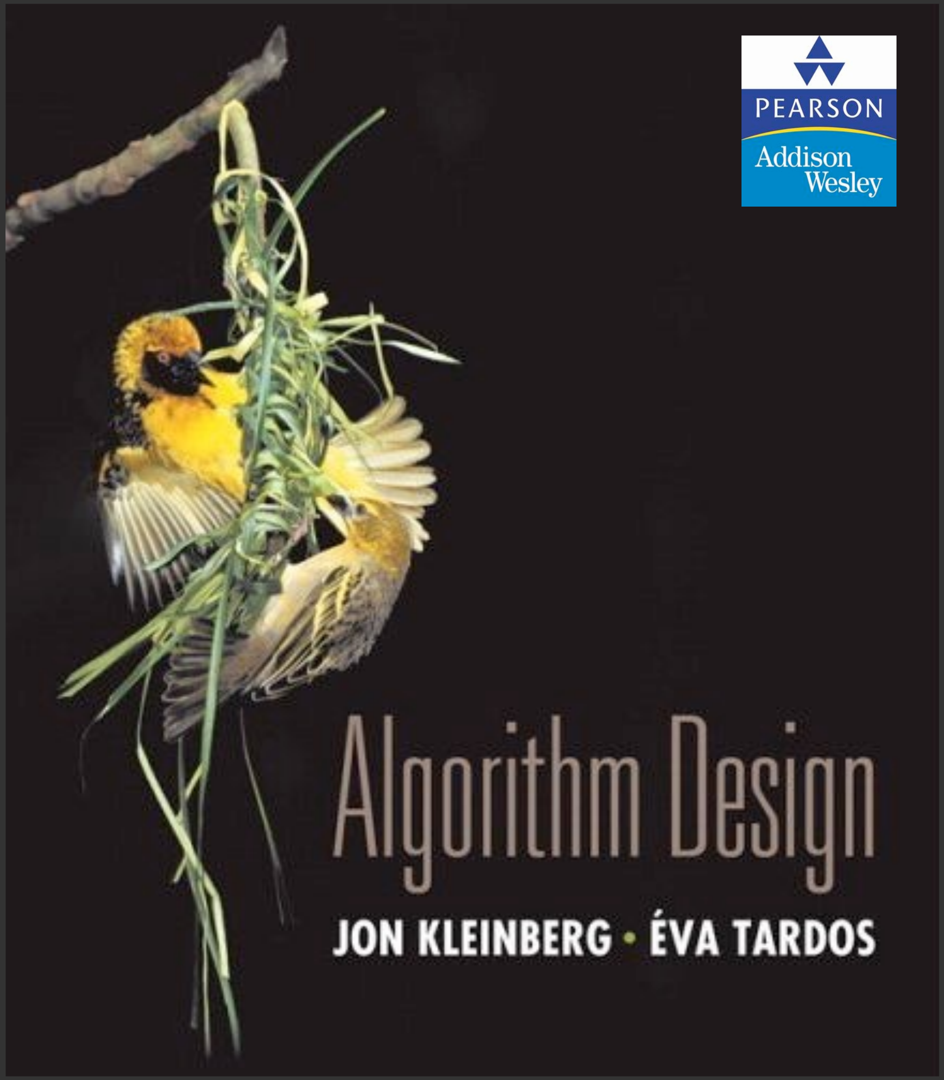


6. Dynamic Programming I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Last updated on 2/10/16 9:26 AM

Algorithmic paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

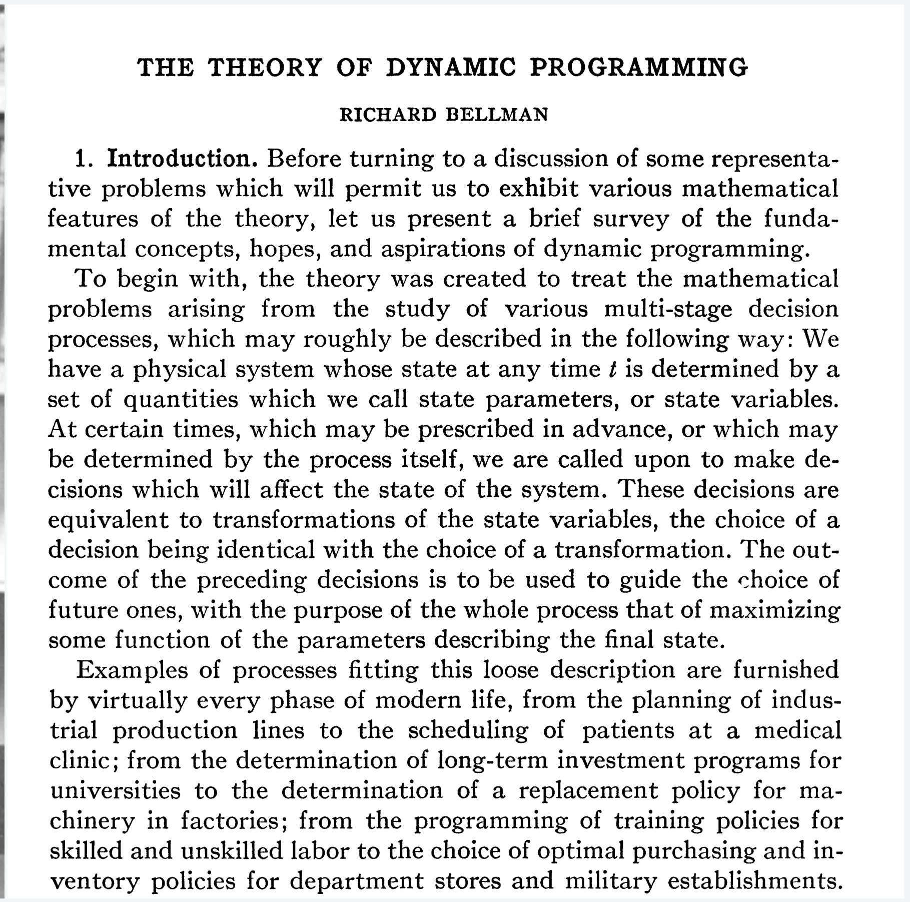
fancy name for
caching away intermediate results
in a table for later reuse

Dynamic programming history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.



THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time *t* is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

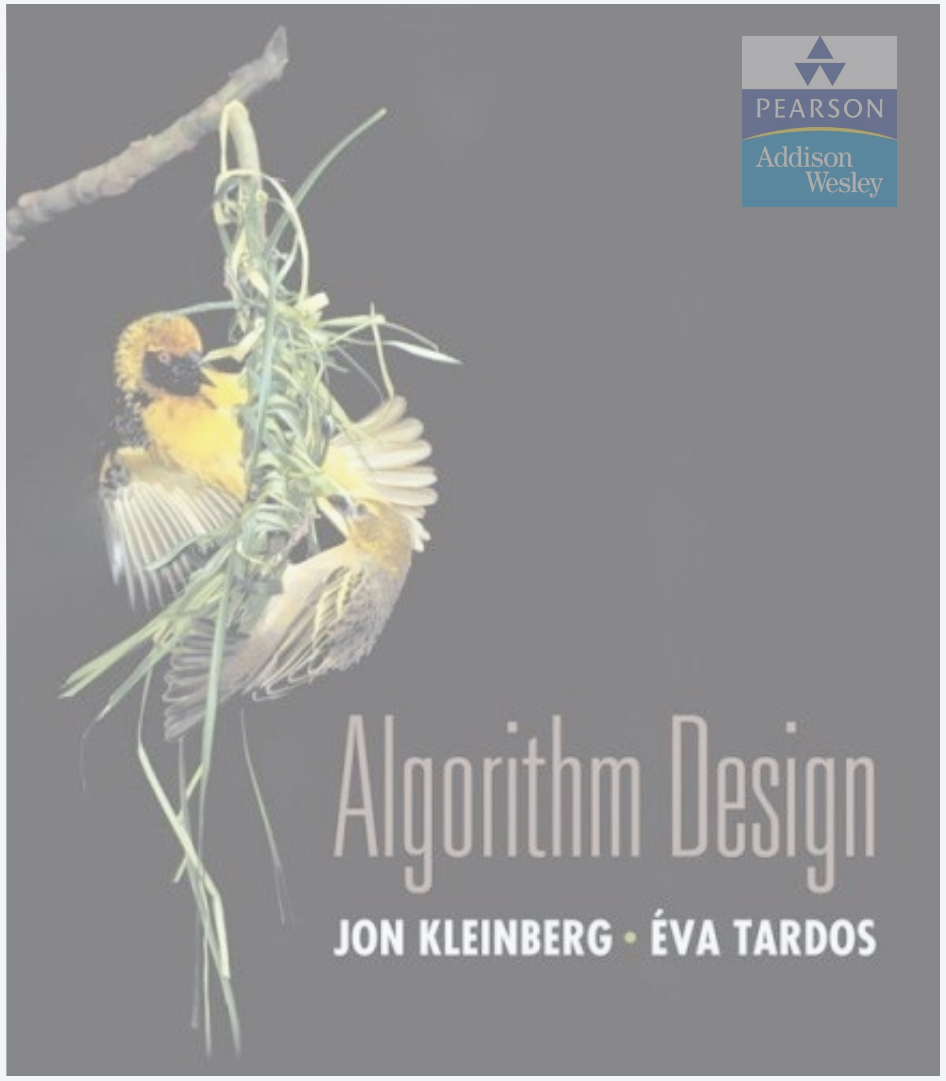
Dynamic programming applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ….
- …

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context-free grammars.
- …



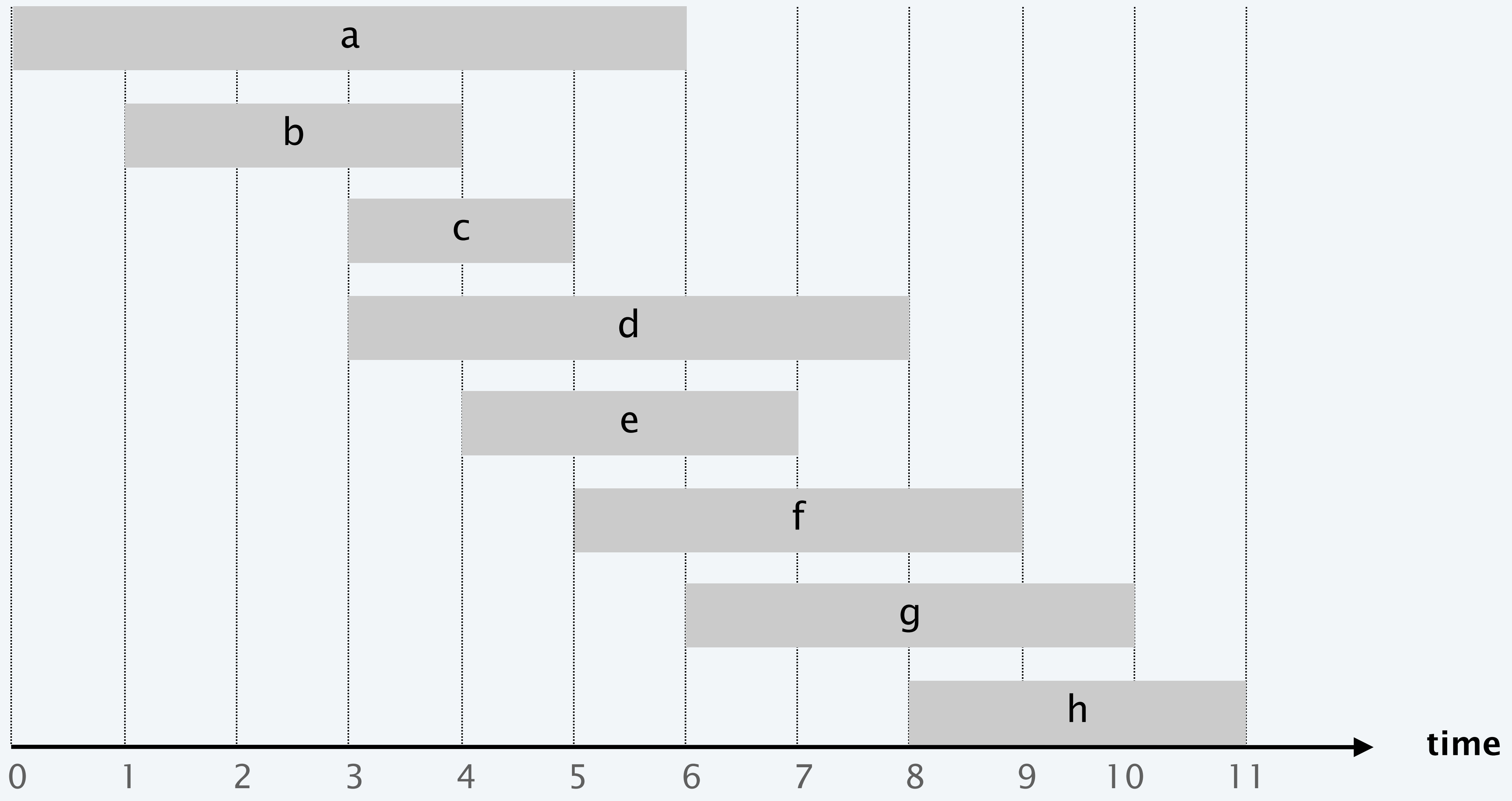
6. Dynamic Programming I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Weighted interval scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



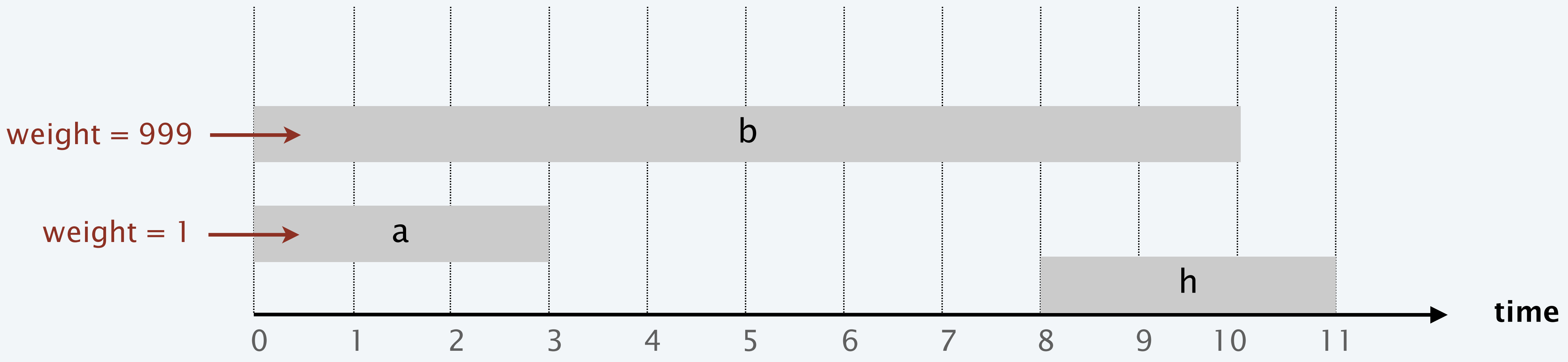
Earliest-finish-time first algorithm

Earliest finish-time first.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Recall. Greedy algorithm is correct if all weights are 1.

Observation. Greedy algorithm fails spectacularly for weighted version.

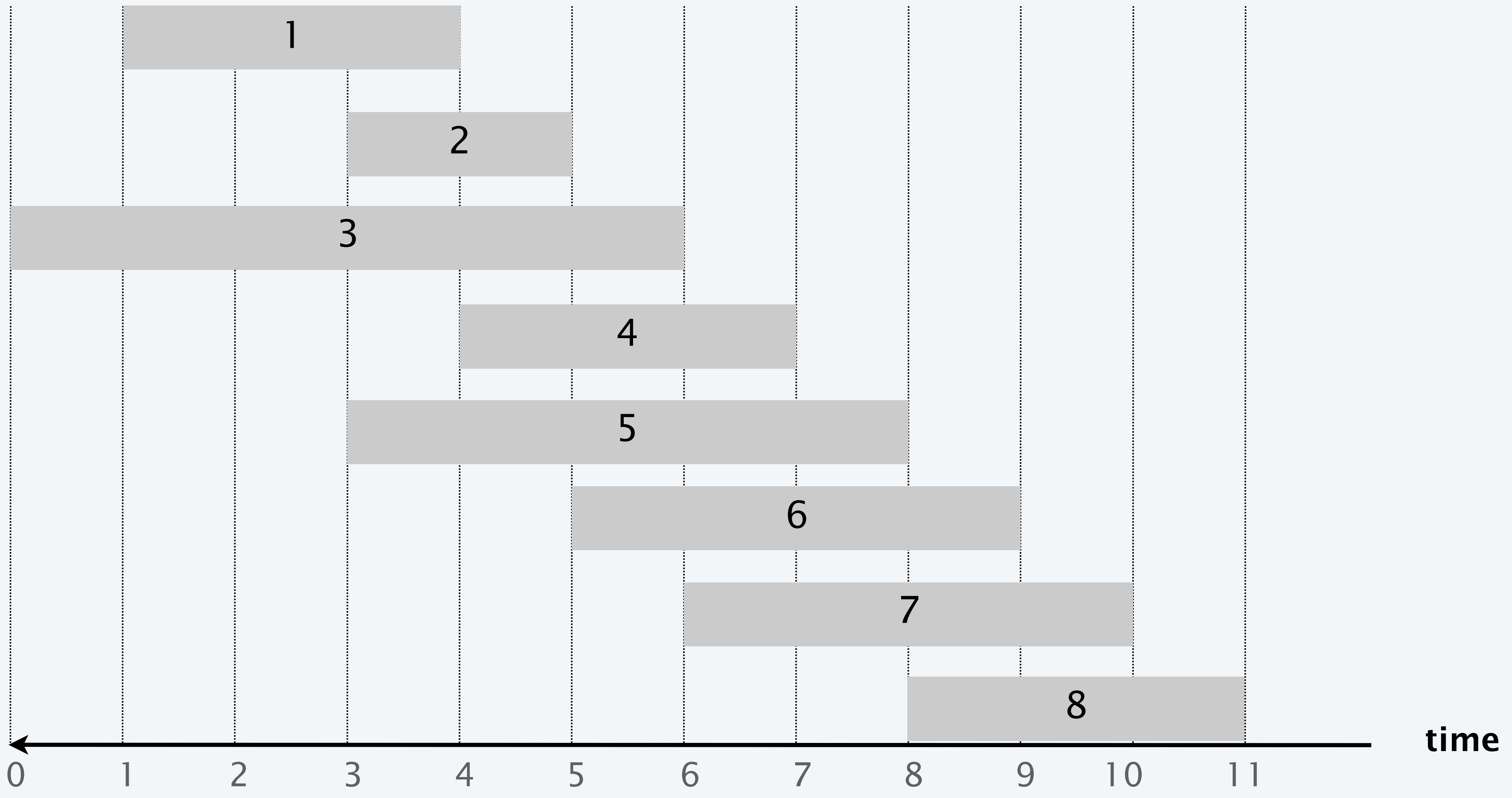


Weighted interval scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex. $p(8) = 5, p(7) = 3, p(2) = 0.$



Dynamic programming: binary choice

Notation. $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \ldots, j$.

Case 1. OPT selects job j.

- Collect profit v_j.
- Can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, \ldots, j - 1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

optimal substructure property
(proof via exchange argument)

Case 2. OPT does not select job j.

- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, j - 1$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Weighted interval scheduling: brute force

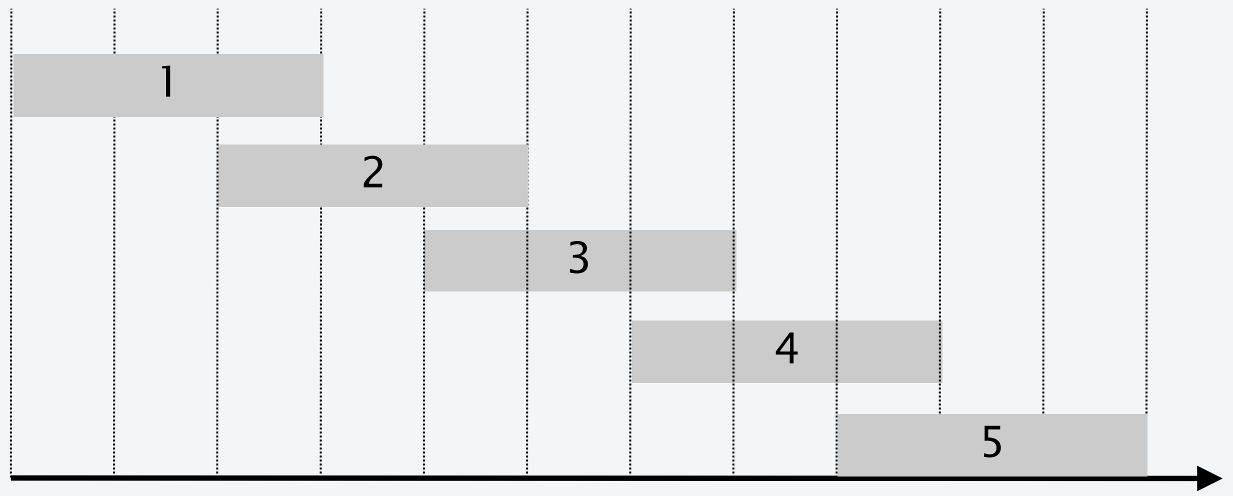
```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

Compute-Opt(j)
if j = 0
   return 0.
else
   return max(v[j] + Compute-Opt(p[j]), Compute-Opt(j–1)).
```

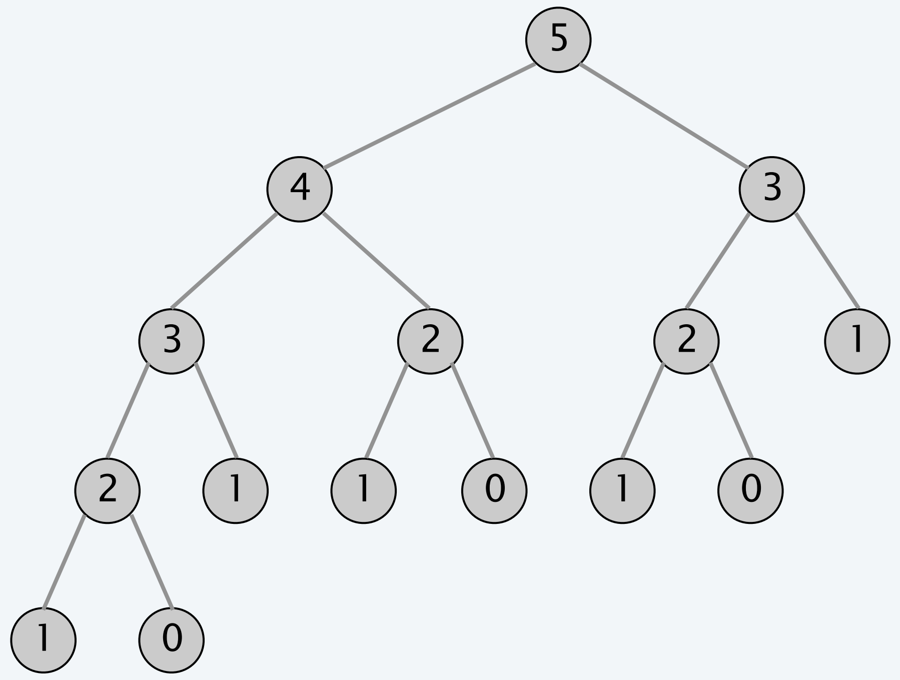
Weighted interval scheduling: brute force

Observation. Recursive algorithm fails spectacularly because of redundant subproblems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j–2



recursion tree

Weighted interval scheduling: memoization

Memoization. Cache results of each subproblem; lookup as needed.

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
_________________
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j – 1)).
return M[j].
```

Weighted interval scheduling: running time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- M-COMPUTE-OPT(j): each invocation takes $O(1)$ time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure $\Phi = \#$ nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of M-COMPUTE-OPT(n) is $O(n)$. ▪

Remark. $O(n)$ if jobs are presorted by start and finish times.

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass.

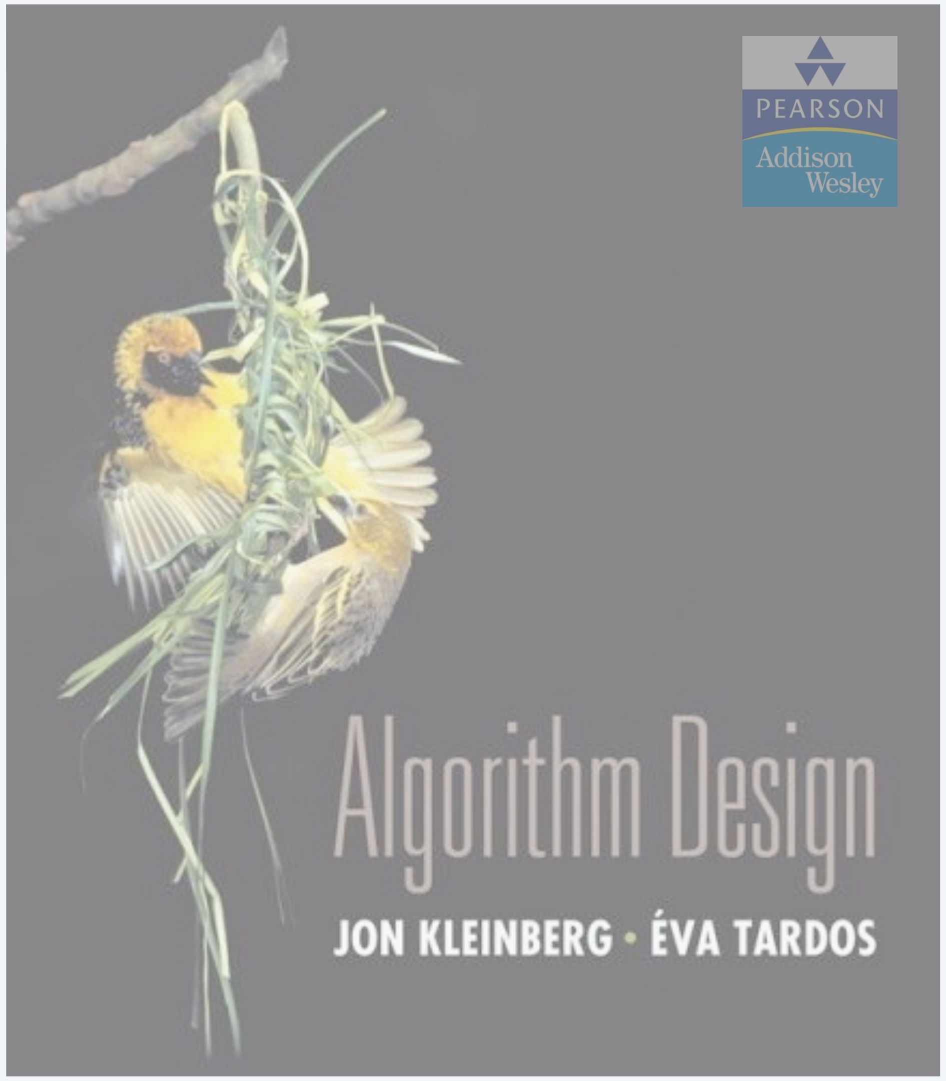
```
Find-Solution(j)
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j-1])
   return { j } ∪ Find-Solution(p[j]).
else
   return Find-Solution(j-1).
```

Analysis. # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted interval scheduling: bottom-up

Bottom-up dynamic programming. Unwind recursion.

```
BOTTOM-UP ($n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$)
________________________________________________
Sort jobs by finish time so that $f_1 \le f_2 \le \ldots \le f_n$.
Compute $p(1), p(2), \ldots, p(n)$.
$M[0] \leftarrow 0$.
FOR j = 1 TO $n$
    $M[j] \leftarrow \max \{ v_j + M[p(j)], M[j-1] \}$.
________________________________________________
```



6. Dynamic Programming I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

Knapsack problem

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.

Ex. $\{ 1, 2, 5 \}$ has value 35.
Ex. $\{ 3, 4 \}$ has value 40.
Ex. $\{ 3, 5 \}$ has value 46 (but exceeds weight limit).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

Greedy by value. Repeatedly add item with maximum v_i.
Greedy by weight. Repeatedly add item with minimum w_i.
Greedy by ratio. Repeatedly add item with maximum ratio v_i / w_i.

Observation. None of greedy algorithms is optimal.

Dynamic programming: false start

Def. $OPT(i)$ = max profit subset of items $1, \ldots, i$.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$.

optimal substructure property
(proof via exchange argument)

Case 2. OPT selects item i.

- Selecting item i does not immediately imply that we will have to reject other items.
- Without knowing what other items were selected before i, we don't even know if we have enough room for i.

Conclusion. Need more subproblems!

Dynamic programming: adding a new variable

Def. $OPT(i, w)$ = max profit subset of items $1, \ldots, i$ with weight limit w.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i - 1 \}$ using weight limit w.

optimal substructure property
(proof via exchange argument)

Case 2. OPT selects item i.

- New weight limit = $w - w_i$.
- OPT selects best of $\{ 1, 2, \ldots, i - 1 \}$ using this new weight limit.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ \ v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up

```
KNAPSACK ($n, W, w_1, \ldots, w_n, v_1, \ldots, v_n$)
________________________________________________
FOR $w = 0$ TO $W$
    $M[0, w] \leftarrow 0$.

FOR $i = 1$ TO $n$
    FOR $w = 0$ TO $W$
    IF $(w_i > w)$  $M[i, w] \leftarrow M[i-1, w]$.
    ELSE            $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$.

RETURN $M[n, W]$.
```

Knapsack problem: bottom-up demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i

	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

Knapsack problem: running time

Theorem. There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(n\, W)$ time and $\Theta(n\, W)$ space.

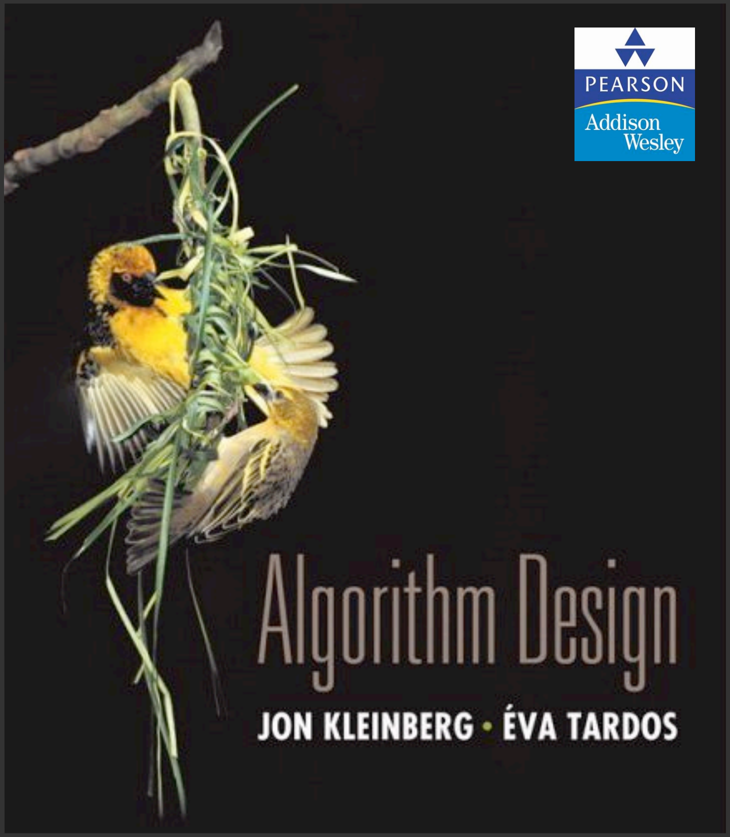
weights are integers between 1 and W

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\, W)$ table entries.
- After computing optimal values, can trace back to find solution: take item i in $OPT(i, w)$ iff $M[i, w] > M[i-1, w]$. ▪

Remarks.

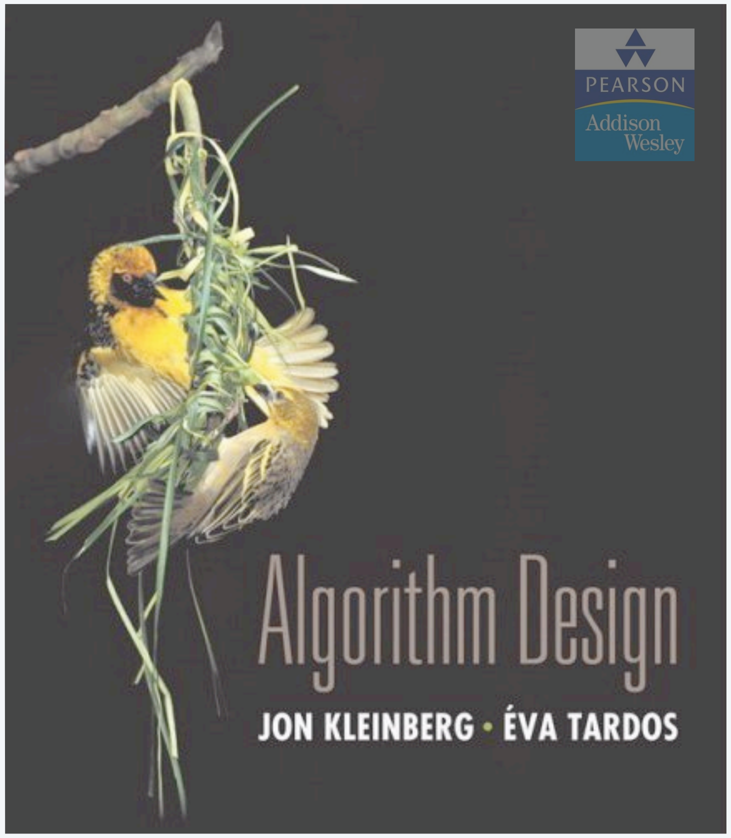
- Not polynomial in input size! ← "pseudo-polynomial"
- Decision version of knapsack problem is NP-COMPLETE. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]



6. Dynamic Programming II

- *sequence alignment*
- *Hirschberg's algorithm*
- *Bellman-Ford algorithm*
- *distance vector protocols*
- *negative cycles in a digraph*

Last updated on Oct 11, 2014, 9:26 AM



SECTION 6.6

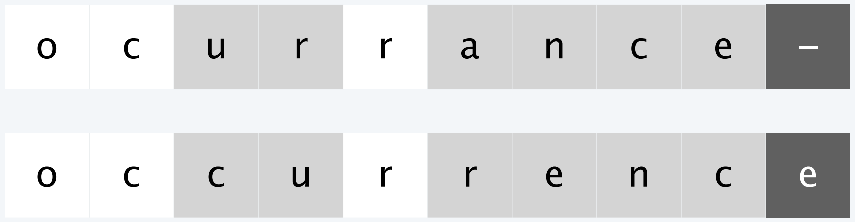
6. DYNAMIC PROGRAMMING II

- *sequence alignment*
- *Hirschberg's algorithm*
- *Bellman-Ford algorithm*
- *distance vector protocols*
- *negative cycles in a digraph*

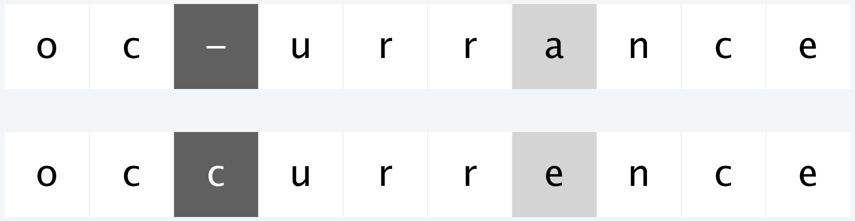
String similarity

Q. How similar are two strings?

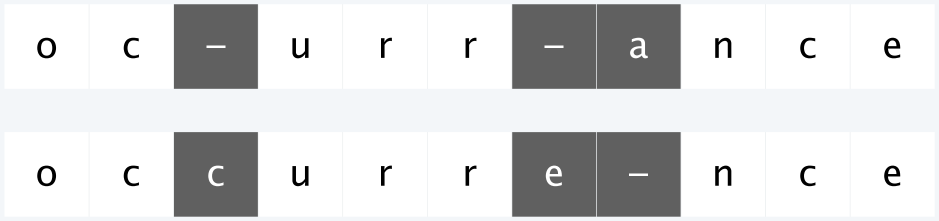
Ex. ocurrance and occurrence.



6 mismatches, 1 gap



1 mismatch, 1 gap

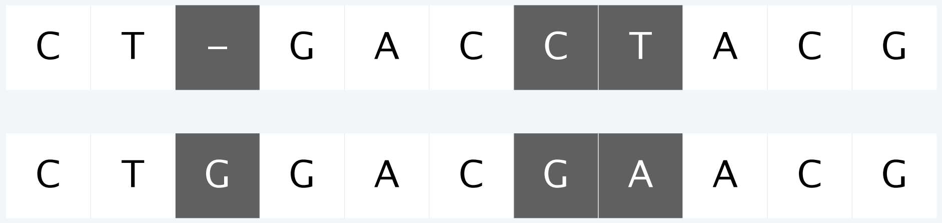


0 mismatches, 3 gaps

Edit distance

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

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.



cost = $\delta + \alpha_{CG} + \alpha_{TA}$

Applications. Unix diff, speech recognition, computational biology, ...

Sequence alignment

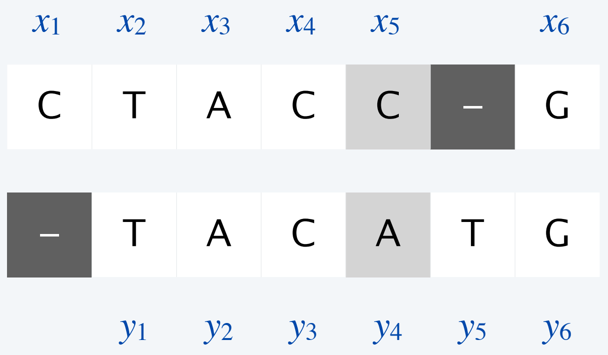
Goal. Given two strings $x_1\, x_2 \ldots x_m$ and $y_1\, y_2 \ldots y_n$ find min cost alignment.

Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

$x_i - y_j$ and $x_{i'} - y_{j'}$ cross if $i < i'$, but $j > j'$

Def. The cost of an alignment M is:

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



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

an alignment of CTACCG and TACATG:

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

Sequence alignment: problem structure

Def. $OPT(i, j)$ = min cost of aligning prefix strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

Case 1. OPT matches $x_i - y_j$.
Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_{j-1}$.

Case 2a. OPT leaves x_i unmatched.
Pay gap for x_i + min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_j$.

optimal substructure property
(proof via exchange argument)

Case 2b. OPT leaves y_j unmatched.
Pay gap for y_j + min cost of aligning $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_{j-1}$.

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

Sequence alignment: algorithm

```
SEQUENCE-ALIGNMENT ($m, n, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha$)

FOR $i = 0$ TO $m$
    $M[i, 0] \leftarrow i\delta$.
FOR $j = 0$ TO $n$
    $M[0, j] \leftarrow j\delta$.

FOR $i = 1$ TO $m$
    FOR $j = 1$ TO $n$
        $M[i, j] \leftarrow \min \{ \alpha[x_i, y_j] + M[i-1, j-1],$
                         $\delta + M[i-1, j],$
                         $\delta + M[i, j-1])$.

RETURN $M[m, n]$.
```

Sequence alignment: analysis

Theorem. The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length m and n in $\Theta(mn)$ time and $\Theta(mn)$ space.

Pf.

- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ▪

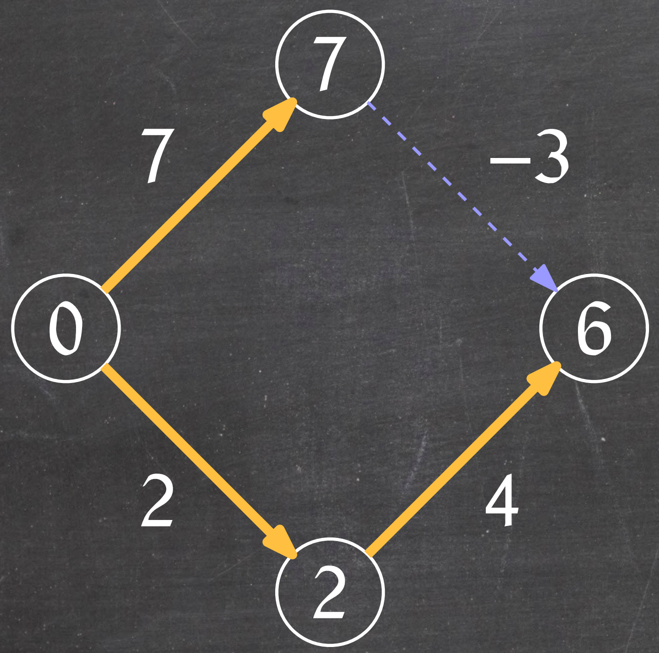
Q. Can we avoid using quadratic space?

A. Easy to compute optimal value in $O(mn)$ time and $O(m + n)$ space.

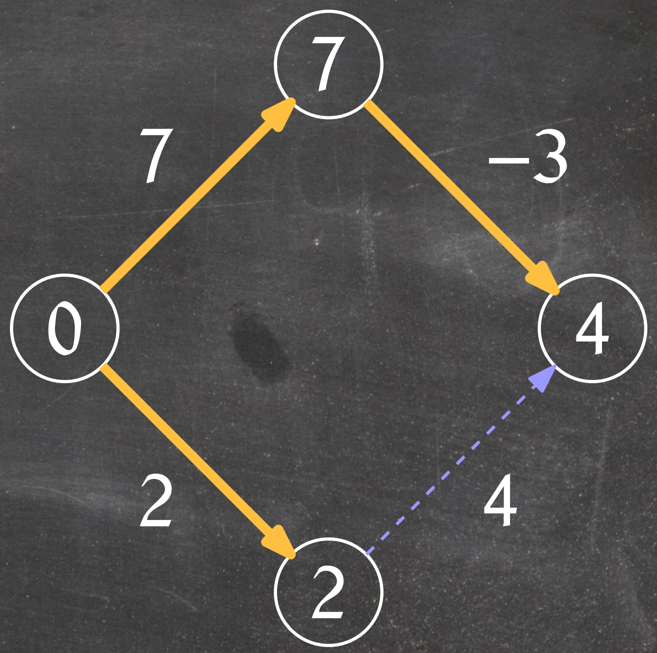
- Compute $\text{OPT}(i, \bullet)$ from $\text{OPT}(i - 1, \bullet)$.
- But, no longer easy to recover optimal alignment itself.

Single-Source Shortest Paths

Dijkstra's algorithm may fail in the presence of negative-weight edges:



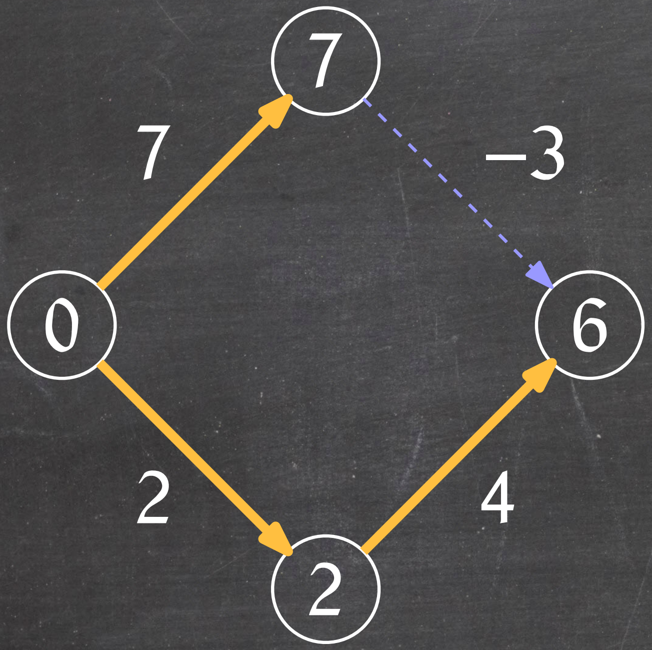
Dijkstra



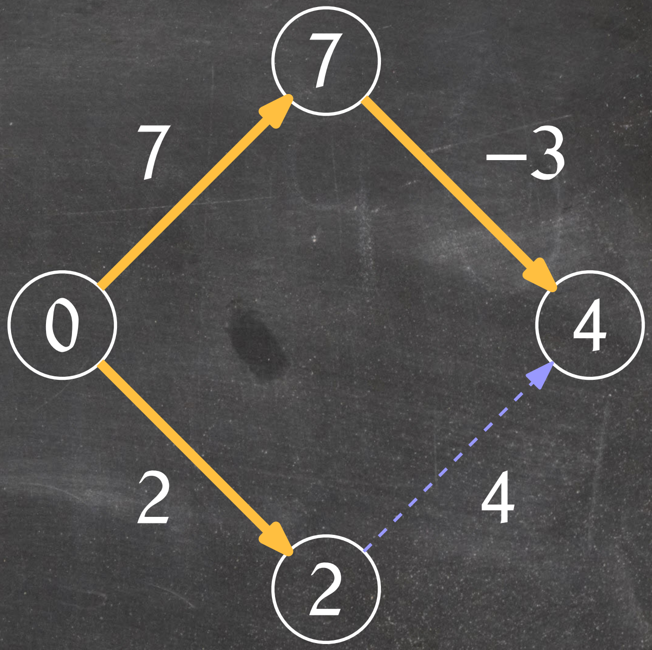
Correct

Single-Source Shortest Paths

Dijkstra's algorithm may fail in the presence of negative-weight edges:



Dijkstra

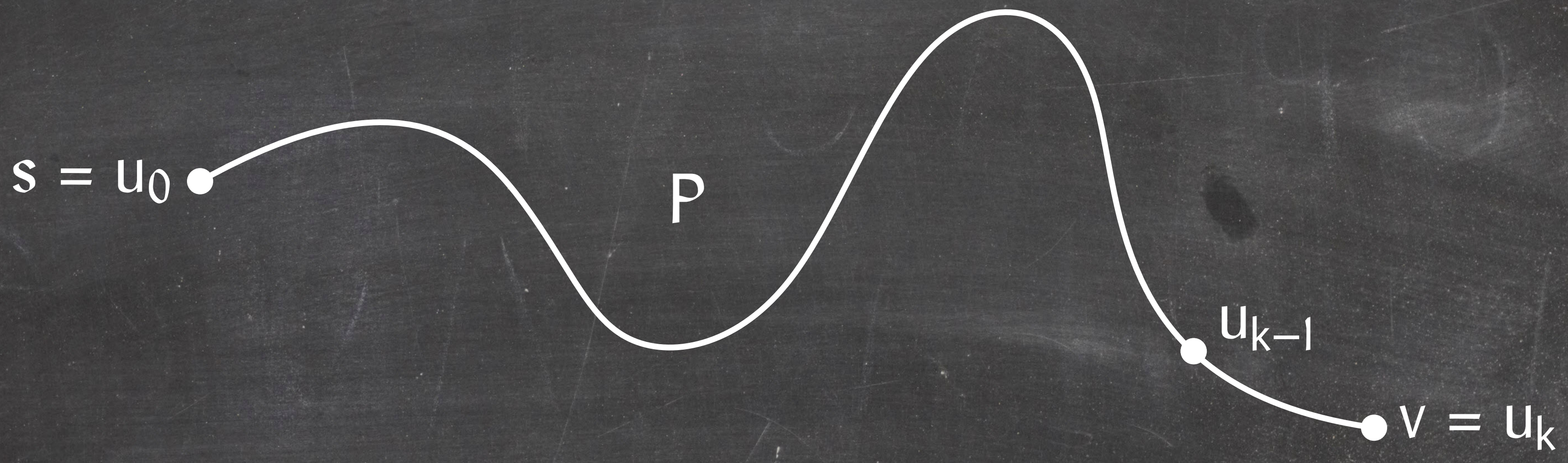


Correct

We need an algorithm that can deal with negative-length edges.

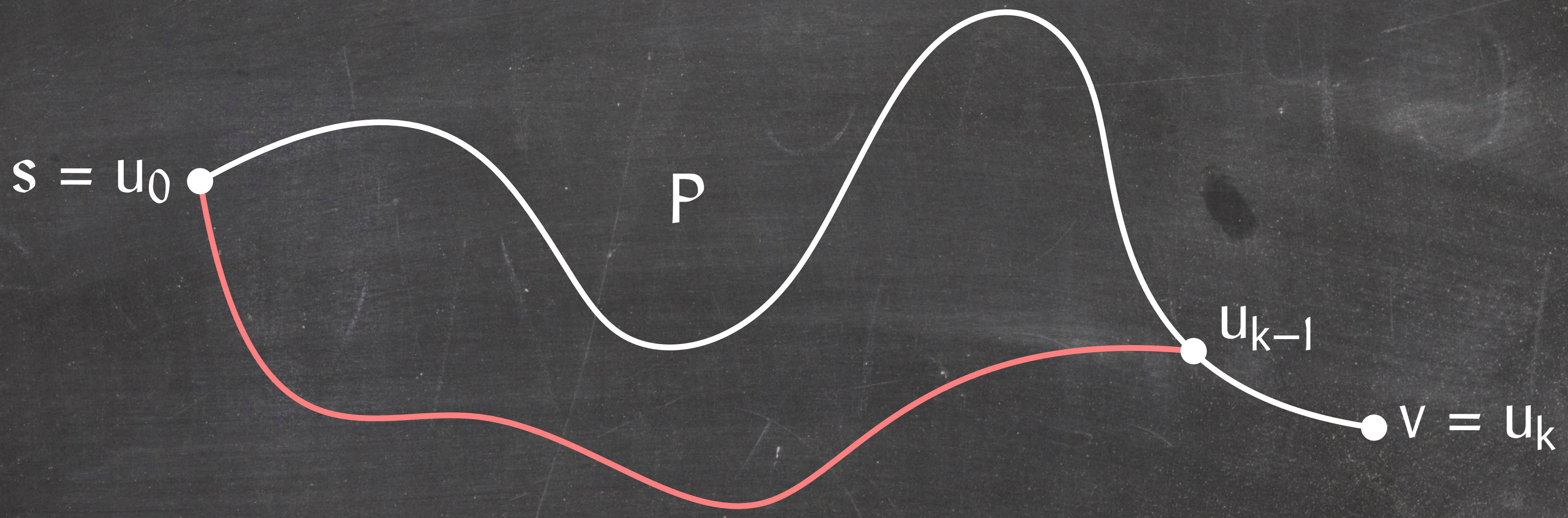
Single-Source Shortest Paths: Problem Analysis

Lemma: If $P = \langle u_0, v_1, \ldots, u_k \rangle$ is a shortest path from $u_0 = s$ to $u_k = v$, then $P' = (u_0, u_1, \ldots, u_{k-1})$ is a shortest path from u_0 to u_{k-1}.



Single-Source Shortest Paths: Problem Analysis

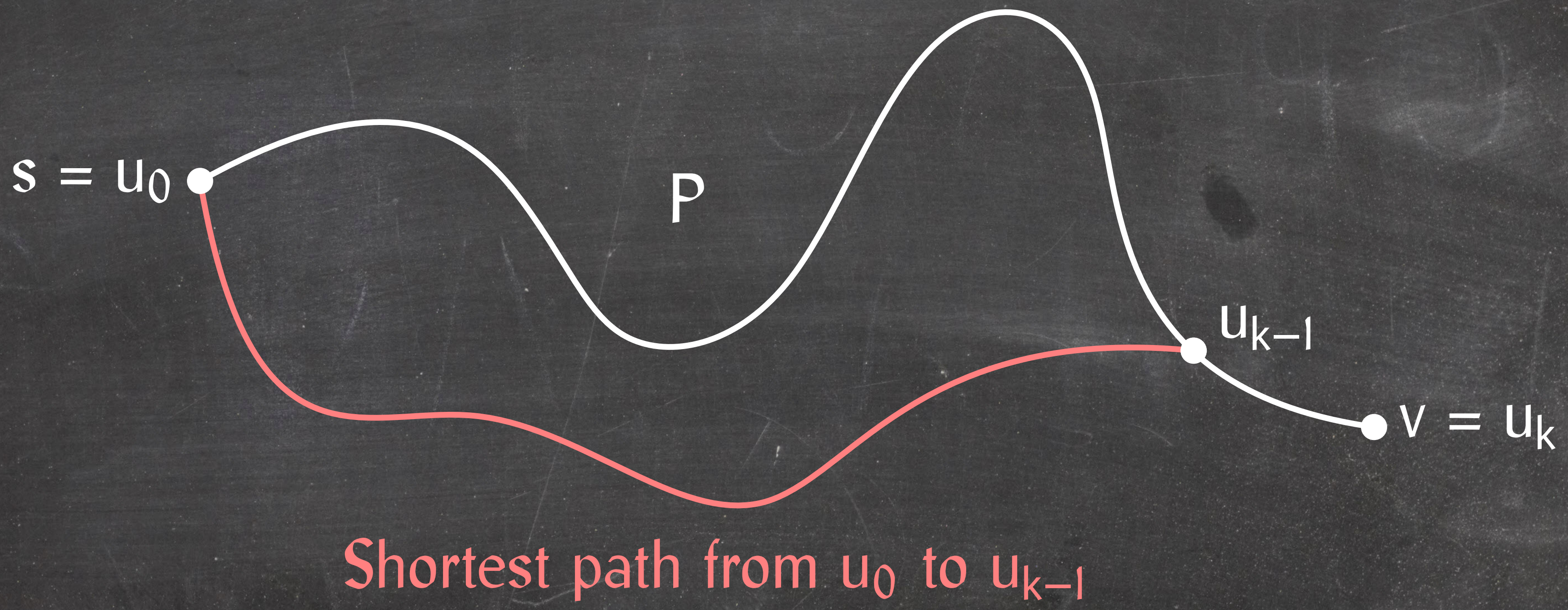
Lemma: If $P = \langle u_0, v_1, \ldots, u_k \rangle$ is a shortest path from $u_0 = s$ to $u_k = v$, then $P' = (u_0, u_1, \ldots, u_{k-1})$ is a shortest path from u_0 to u_{k-1}.



Shortest path from u_0 to u_{k-1}

Single-Source Shortest Paths: Problem Analysis

Lemma: If $P = \langle u_0, v_1, \ldots, u_k \rangle$ is a shortest path from $u_0 = s$ to $u_k = v$, then $P' = (u_0, u_1, \ldots, u_{k-1})$ is a shortest path from u_0 to u_{k-1}.



Observation: P' has one less edge than P.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

- $P_i(s, v)$ has at most $i - 1$ edges or

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

- $P_i(s, v)$ has at most $i - 1$ edges or

- $P_i(s, v)$ has i edges.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

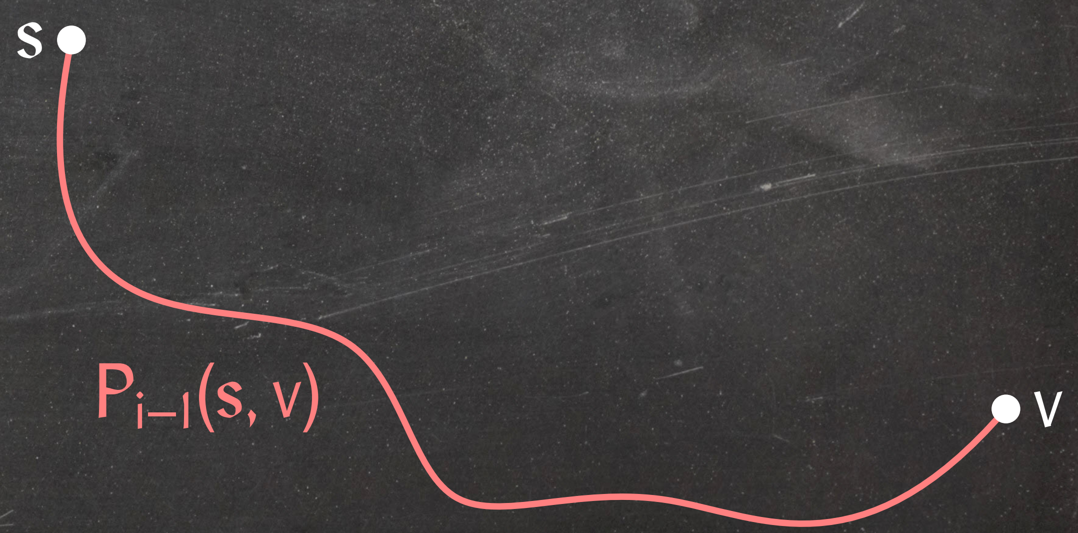
$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

- $P_i(s, v)$ has at most $i - 1$ edges or

$\Rightarrow$ $P_i(s, v) = P_{i-1}(s, v)$

- $P_i(s, v)$ has i edges.



Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

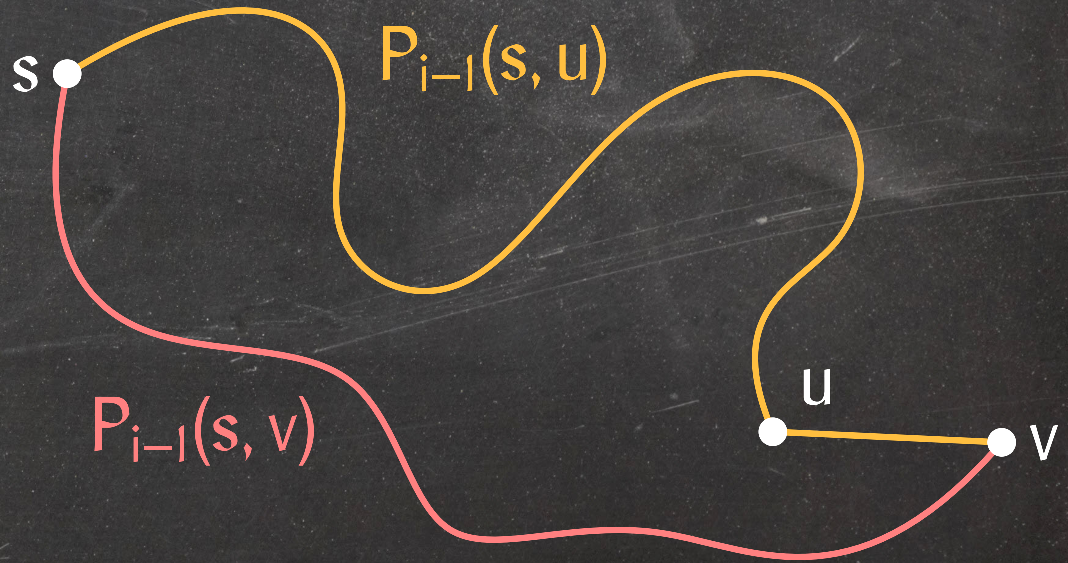
If $i > 0$, then

- $P_i(s, v)$ has at most $i - 1$ edges or

$\Rightarrow$ $P_i(s, v) = P_{i-1}(s, v)$

- $P_i(s, v)$ has i edges.

$\Rightarrow$ $P_i(s, v) = P_{i-1}(s, u) \circ \langle (u, v) \rangle$ for some in-neighbour u of v.



Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

$$d_i(s, v) = \min(d_{i-1}(s, v), \min\{d_{i-1}(s, u) + w(u, v) \mid (u, v) \in E\})$$

Single-Source Shortest Paths: The Bellman-Ford Algorithm

```
BellmanFord(G, s)

for every vertex $v \in G$
    do $d[v] = \infty$
       $P[v] = \emptyset$
$d[s] = 0$
$P[s] = [s]$
for $i = 1$ to $n - 1$
    do for every vertex $v \in G$
            do for every in-edge $e$ of $v$
                    do if $d[e.tail] + e.weight < d[v]$
                            then $d[v] = d[e.tail] + e.weight$
                                 $P[v] = [v] ++ P[e.tail]$
       return $(d, P)$
```

Single-Source Shortest Paths: The Bellman-Ford Algorithm

BellmanFord(G, s)

```
for every vertex $v \in G$
    do $d[v] = \infty$
       $P[v] = \emptyset$
$d[s] = 0$
$P[s] = [s]$
for $i = 1$ to $n - 1$
    do for every vertex $v \in G$
           do for every in-edge $e$ of $v$
                  do if $d[e.tail] + e.weight < d[v]$
                         then $d[v] = d[e.tail] + e.weight$
                              $P[v] = [v] ++ P[e.tail]$
       return $(d, P)$
```

Lemma: The single-source shortest paths problem can be solved in $O(nm)$ time on any weighted graph, provided there are no negative cycles.

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2 m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

Improved algorithms:

- Floyd-Warshall: $O(n^3)$

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2 m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

Improved algorithms:

- Floyd-Warshall: $O(n^3)$
- Johnson: $O(n^2 \lg n + nm)$ (really cool!)
  - Run Bellman-Ford from an arbitrary vertex s in $O(nm)$ time.
  - Change edge weights so they are all non-negative but shortest paths don't change!
  - Run Dijkstra n times.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

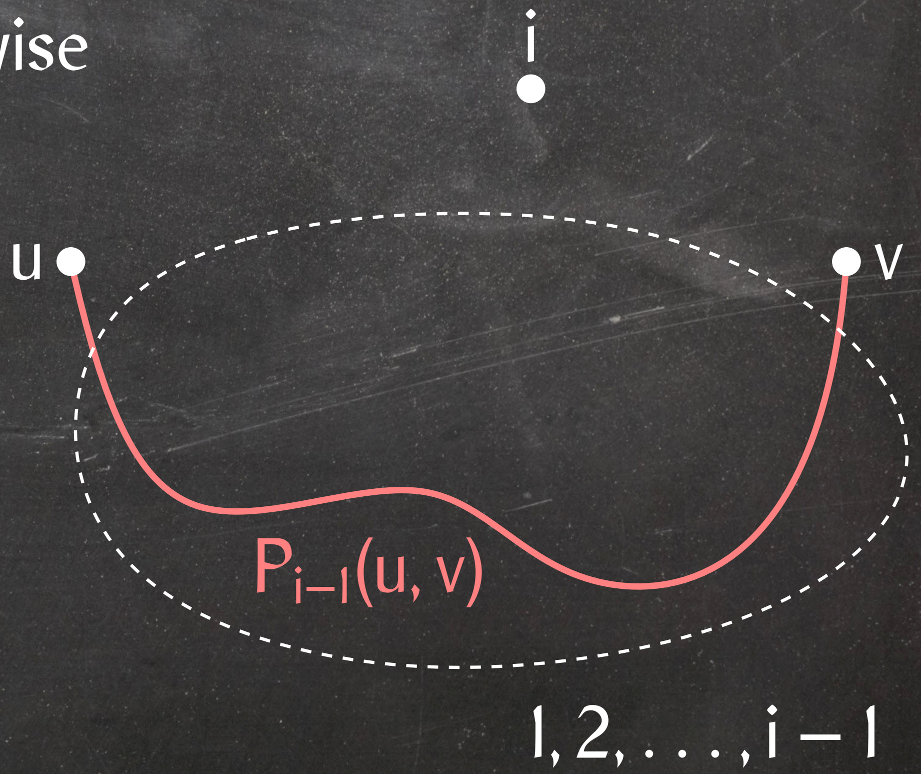
$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

If $i \notin P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, v)$.



All-Pairs Shortest Paths: The Recurrence

Number the vertices 1, 2, . . . , n.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

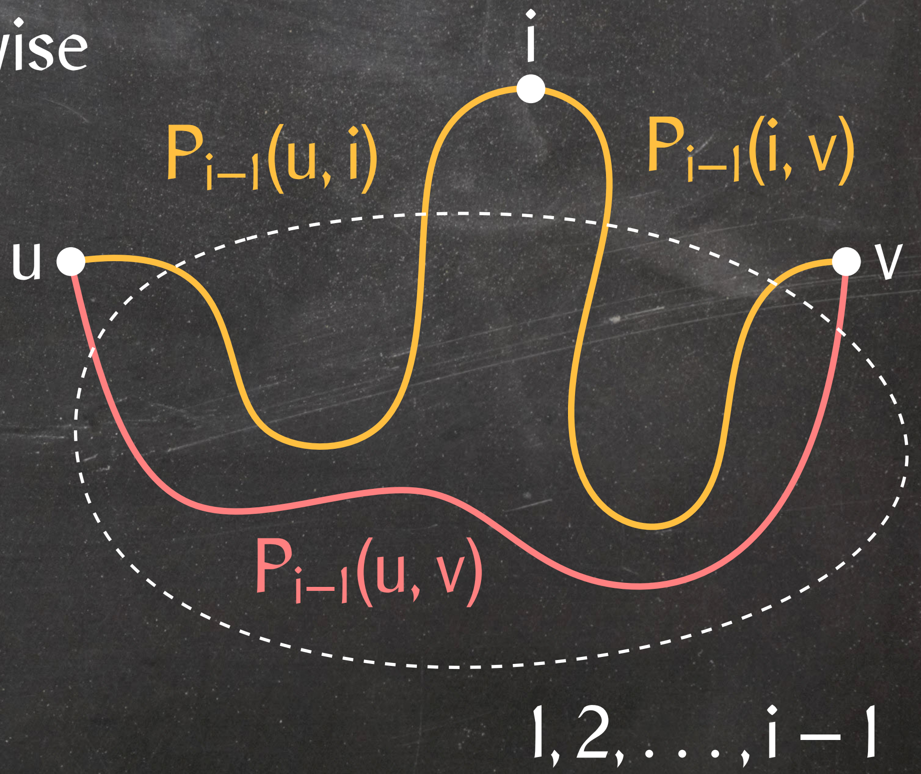
If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

If $i \notin P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, v)$.

If $i \in P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, i) \circ P_{i-1}(i, v)$.



All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u,v)$ be the length of the shortest path $P_i(u,v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u,v\}$.

$d(u,v) = d_n(u,v)$

If $i = 0$, $P_0(u,v)$ cannot visit any vertices other than u and v:

$$d_0(u,v) = \begin{cases} w(u,v) & (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u,v)$ includes vertex i or it doesn't.

If $i \notin P_i(u,v)$, then $P_i(u,v) = P_{i-1}(u,v)$.

If $i \in P_i(u,v)$, then $P_i(u,v) = P_{i-1}(u,i) \circ P_{i-1}(i,v)$.

$$d_i(u,v) = \min(d_{i-1}(u,v), d_{i-1}(u,i) + d_{i-1}(i,v))$$

All-Pairs Shortest Paths: The Floyd-Warshall Algorithm

FloydWarshall(G)

```
for every pair of vertices $u, v \in G$
    do $d[u, v] = \infty$
       $p[u, v] =$ Nothing
for every vertex $v \in G$
    do $d[v, v] = 0$
       $p[v, v] = v$
for every edge $e \in G$
    do $d[e.tail, e.head] = e.weight$
       $p[e.tail, e.head] = e.tail$
for $i = 1$ to $n$
    do for every pair of vertices $u, v \in G$ such that $i \notin \{u, v\}$
           do if $d[u, v] > d[u, i] + d[i, v]$
                 then $d[u, v] = d[u, i] + d[i, v]$
                      $p[u, v] = p[i, v]$
return $(d, p)$
```

All-Pairs Shortest Paths: The Floyd-Warshall Algorithm

FloydWarshall(G)

```
for every pair of vertices u, v ∈ G
    do d[u, v] = ∞
       p[u, v] = Nothing
for every vertex v ∈ G
    do d[v, v] = 0
       p[v, v] = v
for every edge e ∈ G
    do d[e.tail, e.head] = e.weight
       p[e.tail, e.head] = e.tail
for i = 1 to n
    do for every pair of vertices u, v ∈ G such that i ∉ {u, v}
           do if d[u, v] > d[u, i] + d[i, v]
                 then d[u, v] = d[u, i] + d[i, v]
                      p[u, v] = p[i, v]
return (d, p)
```

ReportPath(p, u, v)

```
if p[u, v] = Nothing
    then return Nothing
P = [v]
while v ≠ u
    do v = p[u, v]
       P.prepend(v)
return P
```