https://algs4.cs.princeton.edu

Dynamic Programming

- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*

https://algs4.cs.princeton.edu

Dynamic Programming

- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*

Dynamic programming

Algorithm design paradigm.

- Break up a problem into a series of overlapping subproblems.
- Build up solutions to larger and larger subproblems.

  (caching solutions to subproblems for later reuse)

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.

Richard Bellman, *46

Application areas.

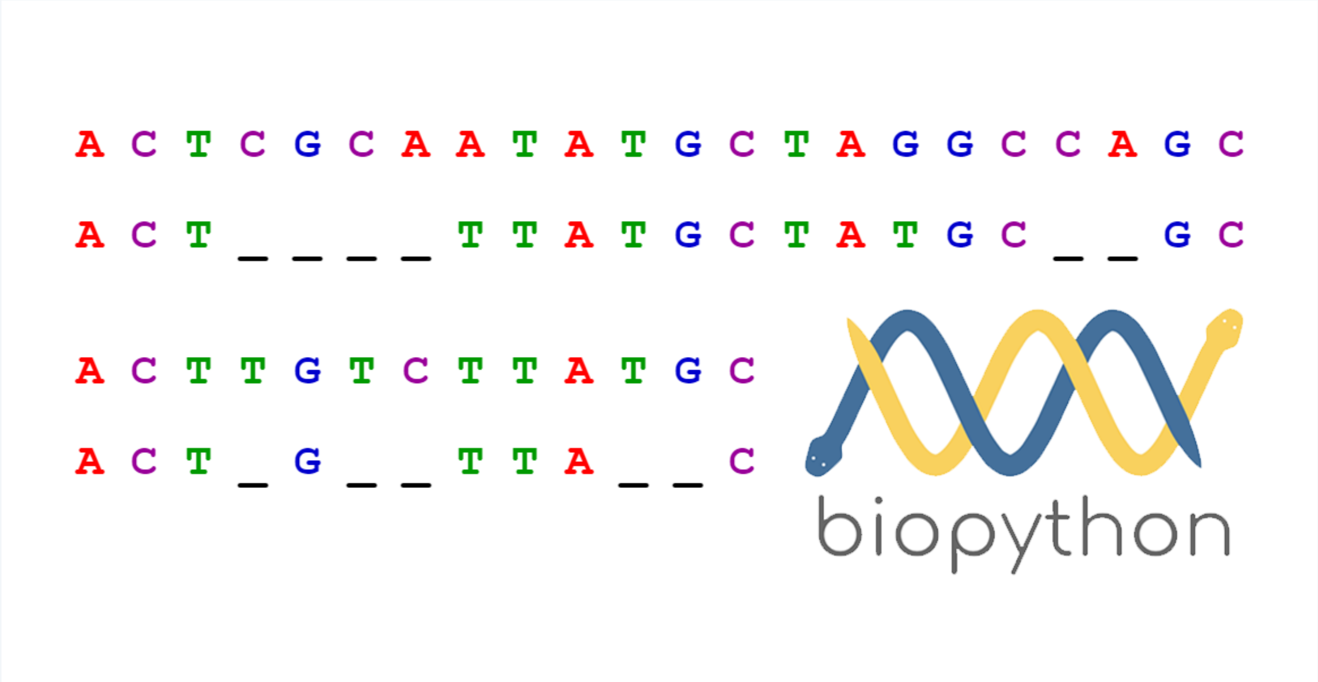
- Operations research: multistage decision processes, control theory, optimization, …
- Computer science: AI, compilers, systems, graphics, databases, robotics, theory, …
- Economics.
- Bioinformatics.
- Information theory.
- Tech job interviews.

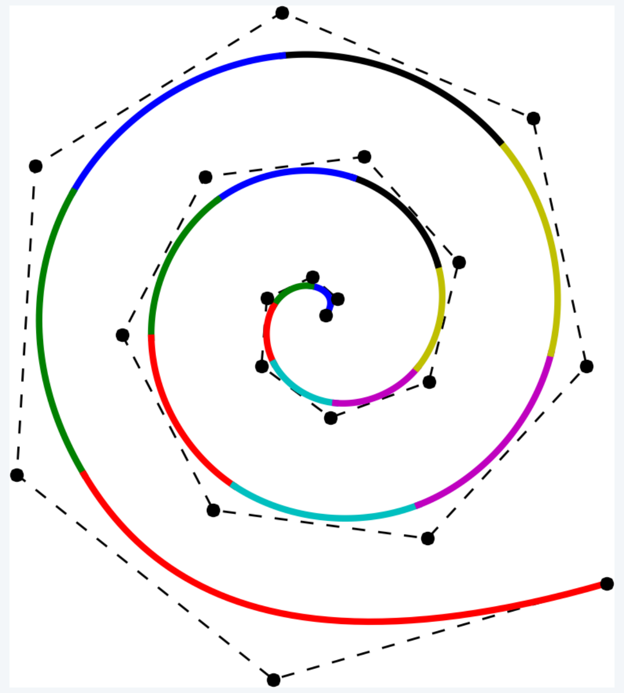
Bottom line. Powerful technique; broadly applicable.

Dynamic programming algorithms

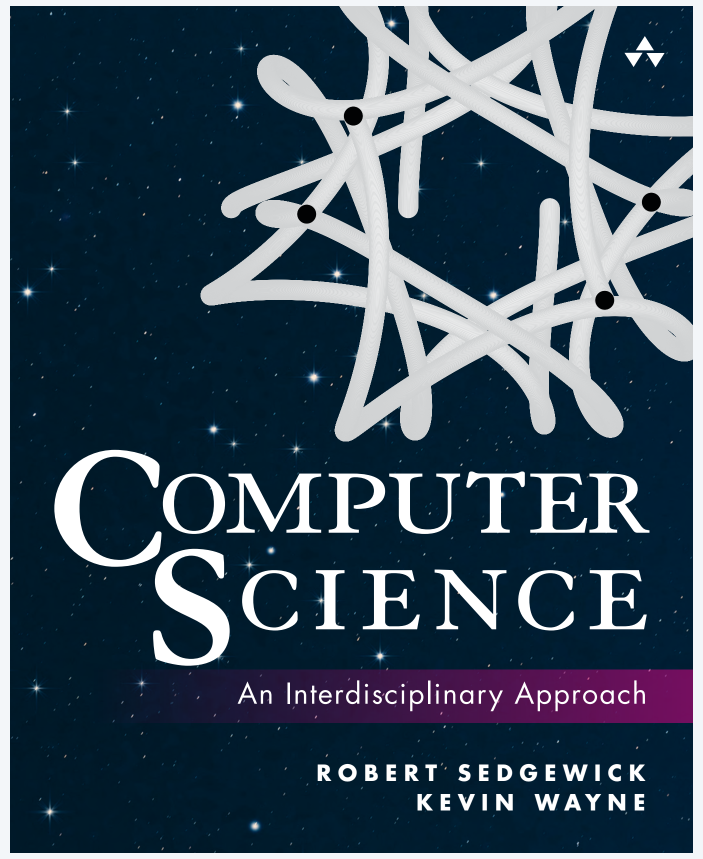
Some famous examples.

- System R algorithm for optimal join order in relational databases.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Bellman–Ford–Moore for shortest path. ⟵ *shortest paths lecture*
- De Boor for evaluating spline curves.
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Avidan–Shamir for seam carving. ⟵ *see Assignment 6*
- **NP**-complete graph problems on trees (vertex color, vertex cover, independent set, ...).
- ...





Dynamic programming books



pp. 284–289

https://algs4.cs.princeton.edu

Dynamic Programming

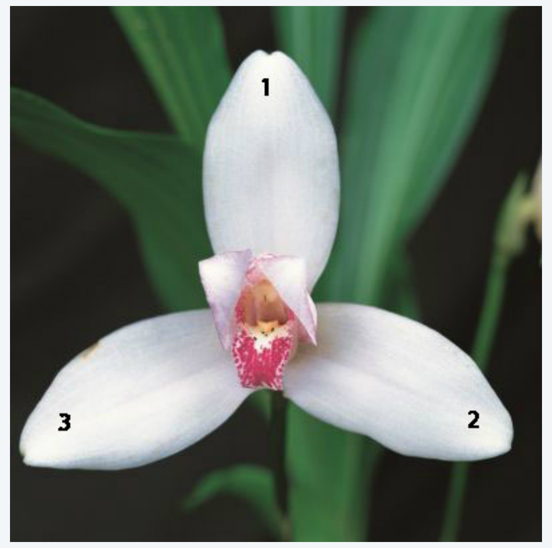
- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*

Fibonacci numbers

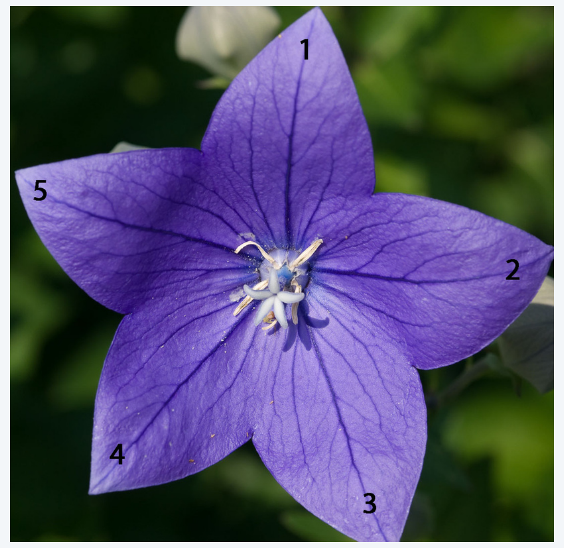
Fibonacci numbers. $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

$$F_i = \begin{cases} 0 & \textit{if } \ i = 0 \\ 1 & \textit{if } \ i = 1 \\ F_{i-1} + F_{i-2} & \textit{if } \ i > 1 \end{cases}$$

Leonardo Fibonacci



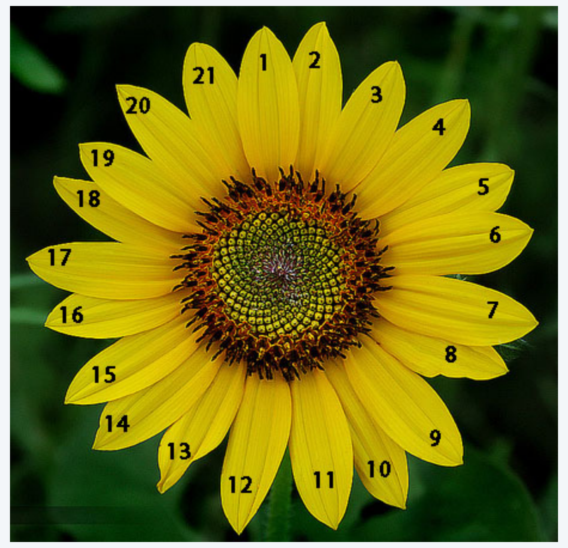
3



5

8

13



21

34

55

89

Fibonacci numbers: naïve recursive approach

Fibonacci numbers. $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ \ldots$

$$F_i = \begin{cases} 0 & \text{if } \ i = 0 \\ 1 & \text{if } \ i = 1 \\ F_{i-1} + F_{i-2} & \text{if } \ i > 1 \end{cases}$$

Goal. Given n, compute F_n .

Naïve recursive approach:

```
public static long fib(int i) {
   if (i == 0) return 0;
   if (i == 1) return 1;
   return fib(i-1) + fib(i-2);
}
```

Dynamic programming: poll 1

How long to compute `fib(80)` using the naïve recursive algorithm?

A. Less than 1 second.

B. About 1 minute.

C. More than 1 hour.

D. Overflows a 64-bit `long` integer.

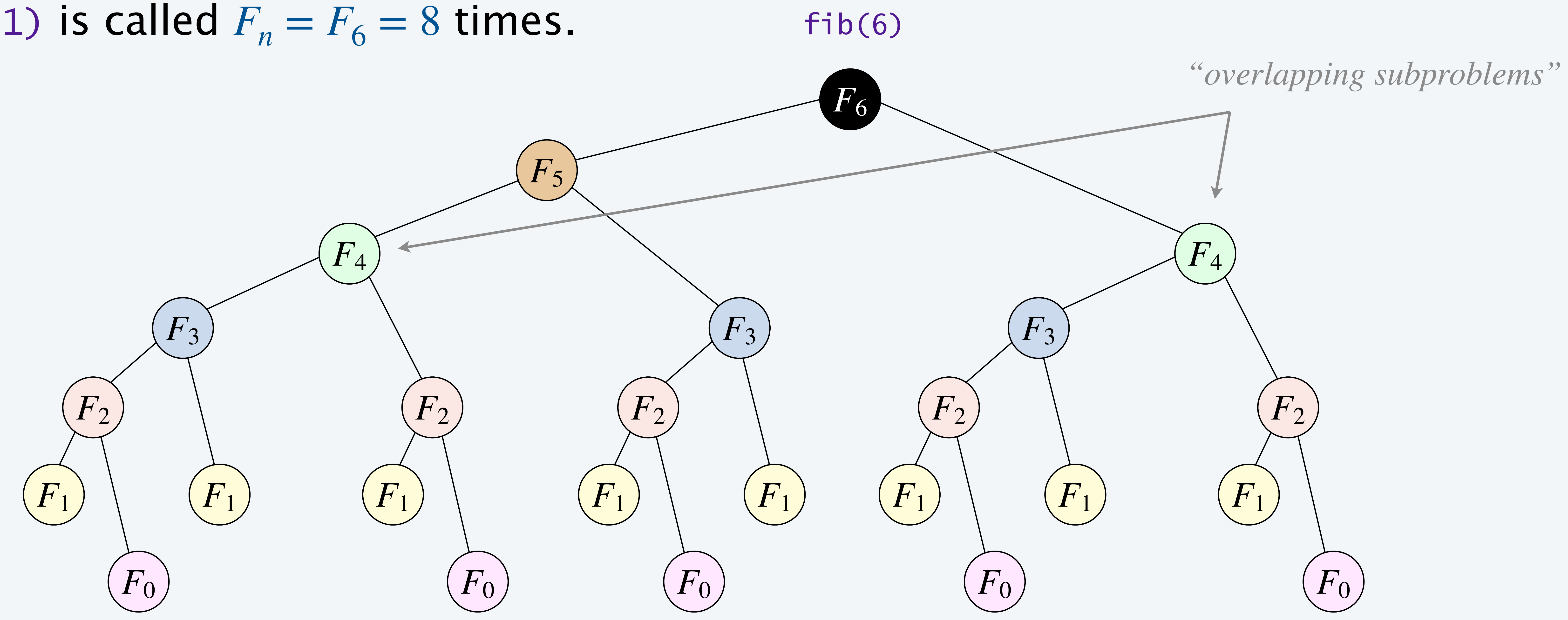
Fibonacci numbers: recursion tree and exponential growth

Exponential waste. Same overlapping subproblems are solved repeatedly.

Ex. To compute `fib(6)`:

- `fib(5)` is called 1 time.
- `fib(4)` is called 2 times.
- `fib(3)` is called 3 times.
- `fib(2)` is called 5 times.
- `fib(1)` is called $F_n = F_6 = 8$ times.

$$F_n \sim \phi^n, \quad \phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$



running time = # subproblems × cost per subproblem

Fibonacci numbers: top-down dynamic programming (memoization)

Memoization.

- Maintain an array (or symbol table) to remember all computed values.
- If value to compute is known, just return it;
  otherwise, compute it; remember it; and return it.

```
public static long fib(int i) {
   if (i == 0) return 0;
   if (i == 1) return 1;
   if (f[i] == 0) f[i] = fib(i-1) + fib(i-2);
   return f[i];
}
```

assume global `long` *array* `f[]`,
initialized to 0 *(unknown)*

Impact. Solves each subproblem F_i only once; $\Theta(n)$ time and space to compute F_n .

Fibonacci numbers: bottom-up dynamic programming (tabulation)

Tabulation.

- Build computation from the "bottom up."
- Solve small subproblems and save solutions.
- Use those solutions to solve larger subproblems.

```
public static long fib(int n) {
   long[] f = new long[n+1];
   f[0] = 0;
   f[1] = 1;
   for (int i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

smaller subproblems

Impact. Solves each subproblem F_i only once; $\Theta(n)$ time and space to compute F_n ; no recursion.

Fibonacci numbers: further improvements

Performance improvements.

- Reduce space by maintaining only two most recent Fibonacci numbers.

```
public static long fib(int n) {
   int f = 0, g = 1;
   for (int i = 0; i < n; i++) {
      g = f + g;
      f = g - f;
   }
   return f;
}
```

`f` and `g` are consecutive Fibonacci numbers

- Exploit additional properties of problem:

but our goal here is to introduce dynamic programming

$$F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} \right\rceil, \quad \phi = \frac{1+\sqrt{5}}{2}$$

$$\begin{pmatrix} F_{i+1} & F_i \\ F_i & F_{i-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^i$$

Dynamic programming recap

Dynamic programming.

- Divide a complex problem into a number of simpler overlapping subproblems.

  [define $n + 1$ subproblems, where subproblem i is computing Fibonacci number i]

- Define a recurrence relation to solve larger subproblems from smaller subproblems.

  [easy to solve subproblem i if we know solutions to subproblems $i - 1$ and $i - 2$]

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{if } i > 1 \end{cases}$$

- Store solutions to subproblems, solving each subproblem only once.

  [store solution to subproblem i in array entry `f[i]`]

- Use stored solutions to solve the original problem.

  [solution to subproblem n is original problem]

https://algs4.cs.princeton.edu

Dynamic Programming

- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*

House painting problem

Goal. Given a row of n black houses, paint some orange so that:

- Maximize total profit, where $profit(i)$ = profit from painting house i orange.
- Constraint: no two adjacent houses painted orange.

i	1	2	3	4	5	6
$profit(i)$	$\boxed{10}$	9	$\boxed{13}$	20	$\boxed{30}$	25

profit for painting houses 1, 3, and 5 orange
(10+ 13 + 30 = 53)

House painting problem

Goal. Given a row of n black houses, paint some orange so that:

- Maximize total profit, where $profit(i)$ = profit from painting house i orange.
- Constraint: no two adjacent houses painted orange.

i	1	2	3	4	5	6
$profit(i)$	$\boxed{10}$	9	13	$\boxed{20}$	30	$\boxed{25}$

profit for painting houses 1, 4, and 6 orange
(10+ 20 + 25 = 55)

House painting problem: dynamic programming formulation

Goal. Given a row of n black houses, paint some orange so that:

- Maximize total profit, where $profit(i)$ = profit from painting house i orange.
- Constraint: no two adjacent houses painted orange.

Subproblems. $OPT(i)$ = max profit to paint houses $1, \ldots, i$.

Optimal value. $OPT(n)$.

i	0	1	2	3	4	5	6
$profit(i)$		10	9	13	20	30	25
$OPT(i)$	0	10	10	23	$\boxed{30}$	$\boxed{53}$	$\boxed{55}$

$$
\begin{aligned}
OPT(6) &= \max \{ \underbrace{OPT(5)}_{\textit{keep house 6 black}},\ \underbrace{profit(6) + OPT(4)}_{\textit{paint house 6 orange}} \} \\
&= \max \{ 53,\ 25 + 30 \} \\
&= 55
\end{aligned}
$$

House painting problem: dynamic programming formulation

Goal. Given a row of n black houses, paint some orange so that:

- Maximize total profit, where $\mathit{profit}(i)$ = profit from painting house i orange.
- Constraint: no two adjacent houses painted orange.

Subproblems. $OPT(i)$ = max profit to paint houses $1, \ldots, i$.

Optimal value. $OPT(n)$.

Binary choice. To compute $OPT(i)$, either:

- Don't paint house i orange: $OPT(i-1)$.
- Paint house i orange: $\mathit{profit}(i) + OPT(i-2)$.

optimal substructure
(optimal solution can be constructed from optimal solutions to smaller subproblems)

take best

Dynamic programming recurrence.

$$OPT(i) = \begin{cases} 0 & \text{if } i = 0 \\ \mathit{profit}(1) & \text{if } i = 1 \\ \max\{\, OPT(i-1),\ \ \mathit{profit}(i) + OPT(i-2) \,\} & \text{if } i \geq 2 \end{cases}$$

House painting: naïve recursive implementation

Naïve recursive approach:

```
private int opt(int i) {
   if (i == 0) return 0;
   if (i == 1) return profit[1];
   return Math.max(opt(i-1), profit[i] + opt(i-2));
}
```

Dynamic programming recurrence.

$$OPT(i) \;=\; \begin{cases} 0 & \text{if } i = 0 \\ \mathit{profit}(1) & \text{if } i = 1 \\ \max\{\, OPT(i-1),\;\; \mathit{profit}(i) + OPT(i-2)\,\} & \text{if } i \ge 2 \end{cases}$$

Dynamic programming: poll 2

What is running time of the naïve recursive algorithm as a function of n ?

A. $\Theta(n)$

B. $\Theta(n^2)$

C. $\Theta(c^n)$ for some $c > 1$.

D. $\Theta(n!)$

```
private int opt(int i) {
   if (i == 0) return 0;
   if (i == 1) return profit[1];
   return Math.max(opt(i-1), profit[i] + opt(i-2));
}
```

“ Those who cannot remember the past are condemned to repeat it. ”

— Dynamic Programming

(Jorge Agustín Nicolás Ruiz de Santayana y Borrás)

Housing painting: bottom-up implementation

Bottom-up DP implementation.

```
int[] opt = new int[n+1];
opt[0] = 0;
opt[1] = profit[1];
for (int i = 2; i <= n; i++)
   opt[i] = Math.max(opt[i-1], profit[i] + opt[i-2]);
```

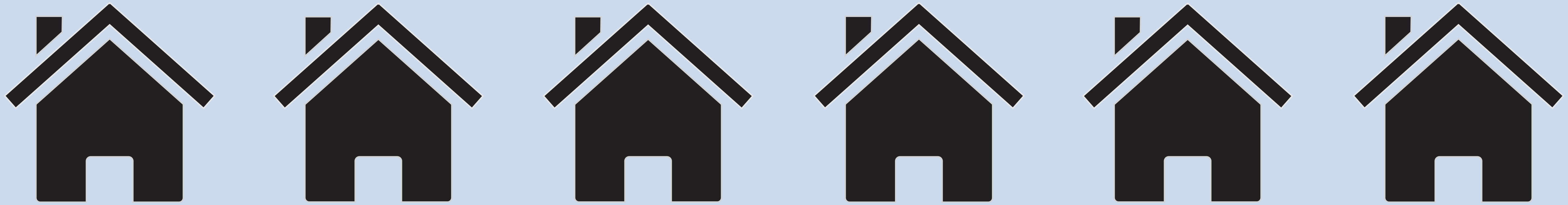
solutions to smaller subproblems already available

$$OPT(i) = \begin{cases} 0 & \text{if } i = 0 \\ \mathit{profit}(1) & \text{if } i = 1 \\ \max \{ OPT(i-1), \;\; \mathit{profit}(i) + OPT(i-2) \} & \text{if } i \geq 2 \end{cases}$$

Proposition. Computing $OPT(n)$ takes $\Theta(n)$ time and uses $\Theta(n)$ extra space.

Housing painting: trace

Bottom-up DP implementation trace.



i	0	1	2	3	4	5	6
$profit(i)$		10	9	13	20	30	25
$OPT(i)$	0	10	10	23	30	53	$\boxed{55}$

OPT(i) = max profit for painting houses 1, 2, …, i

Housing painting: traceback

Q. We computed the optimal value. How to reconstruct an optimal solution?

A. Trace back path that led to optimal value.

i	0	1	2	3	4	5	6
$profit(i)$		10	9	13	20	30	25
$OPT(i)$	0	10	10	23	30	53	$\boxed{55}$

OPT(i) = max profit for painting houses 1, 2, …, i

Coin changing problem

Problem. Given n coin denominations $\{ d_1, d_2, \ldots, d_n \}$ and a target value V, find the fewest coins needed to make change for V (or report impossible).

Ex. Coin denominations = $\{1, 10, 25, 100\}$, $V = 131$.

Greedy (8 coins). $131¢ = 100 + 25 + 1 + 1 + 1 + 1 + 1 + 1$.

Optimal (5 coins). $131¢ = 100 + 10 + 10 + 10 + 1$.

8 coins
(131¢)

5 coins
(131¢)

vending machine
(out of nickels)

Remark. Greedy algorithm is optimal for U.S. coin denominations $\{1, 5, 10, 25, 100\}$.

Dynamic programming: poll 3

Which subproblems for coin changing problem?

A. $OPT(i)$ = fewest coins needed to make change for target value V using only coin denominations $d_1, d_2, \ldots, d_i$.

B. $OPT(v)$ = fewest coins needed to make change for amount v, for $v = 0, 1, \ldots, V$.

C. Either A or B.

D. Neither A nor B.

Coin changing: dynamic programming formulation

Problem. Given n coin denominations $\{ d_1, d_2, \ldots, d_n \}$ and a target value V, find the fewest coins needed to make change for V (or report impossible).

Subproblems. $OPT(v)$ = fewest coins needed to make change for amount v.

Optimal value. $OPT(V)$.

Ex. Coin denominations $\{ 1, 5, 8 \}$ and $V = 10$.

v	0¢	1¢	2¢	3¢	4¢	5¢	6¢	7¢	8¢	9¢	10¢
# coins	0	1	2	3	4	1	2	3	1	2	2

$$OPT(10) = \min \{ 1 + OPT(10-1),\ 1 + OPT(10-5),\ 1 + OPT(10-8) \}$$

$$= \min \{ 1+2,\ 1+1,\ 1+2 \}$$

$$= 2$$

Coin changing: dynamic programming formulation

Problem. Given n coin denominations $\{ d_1, d_2, \ldots, d_n \}$ and a target value V, find the fewest coins needed to make change for V (or report impossible).

Subproblems. $OPT(v)$ = fewest coins needed to make change for amount v.

Optimal value. $OPT(V)$.

Multiway choice. To compute $OPT(v)$,

- Select a coin of denomination $d_i \le v$ for some i.
- Use fewest coins to make change for $v - d_i$.

take best (among all coin denominations)

optimal substructure

Dynamic programming recurrence.

$$OPT(v) = \begin{cases} 0 & \text{if } v = 0 \\ \min\limits_{i \,:\, d_i \le v} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0 \end{cases}$$

notation: min *is over all coin denominations of value* ≤ v
(min *is* ∞ *if no such coin denominations*)

Coin changing: bottom-up implementation

Bottom-up DP implementation.

```
int[] opt = new int[V+1];
opt[0] = 0;

for (int v = 1; v <= V; v++) {
   opt[v] = INFINITY;
   for (int i = 1; i <= n; i++) {
      if (d[i] <= v)
         opt[v] = Math.min(opt[v], 1 + opt[v - d[i]]);
   }
}
```

$$OPT(v) = \begin{cases} 0 & \text{if } v = 0 \\ \min\limits_{i \,:\, d_i \le v} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0 \end{cases}$$

Proposition. DP algorithm takes $\Theta(n\,V)$ time and uses $\Theta(V)$ extra space.

Note. Not polynomial in input size; underlying problem is **NP**-complete.

n, $\log V$

https://algs4.cs.princeton.edu

Dynamic Programming

- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*

Shortest paths in directed acyclic graphs: dynamic programming formulation

Problem. Given a DAG with positive edge weights, find shortest path from s to t.

Subproblems. $distTo(v)$ = length of shortest $s \leadsto v$ path.

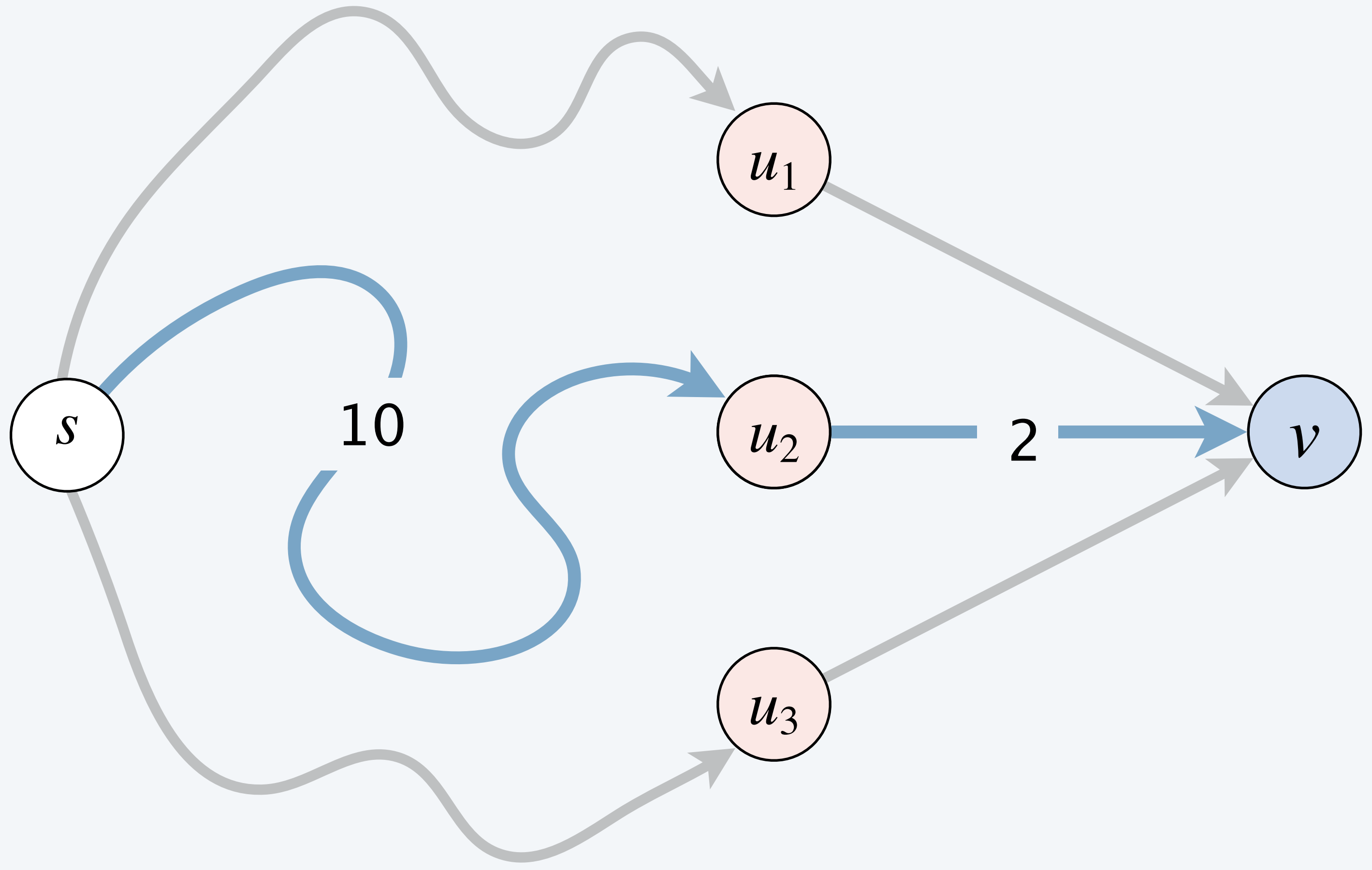
Goal. $distTo(t)$.

Multiway choice. To compute $distTo(v)$:

- Select an edge $e = u \to v$ entering v.
- Concatenate with shortest $s \leadsto u$ path.

take best among
$distTo(u)$ + $weight(e)$

optimal substructure



Dynamic programming recurrence.

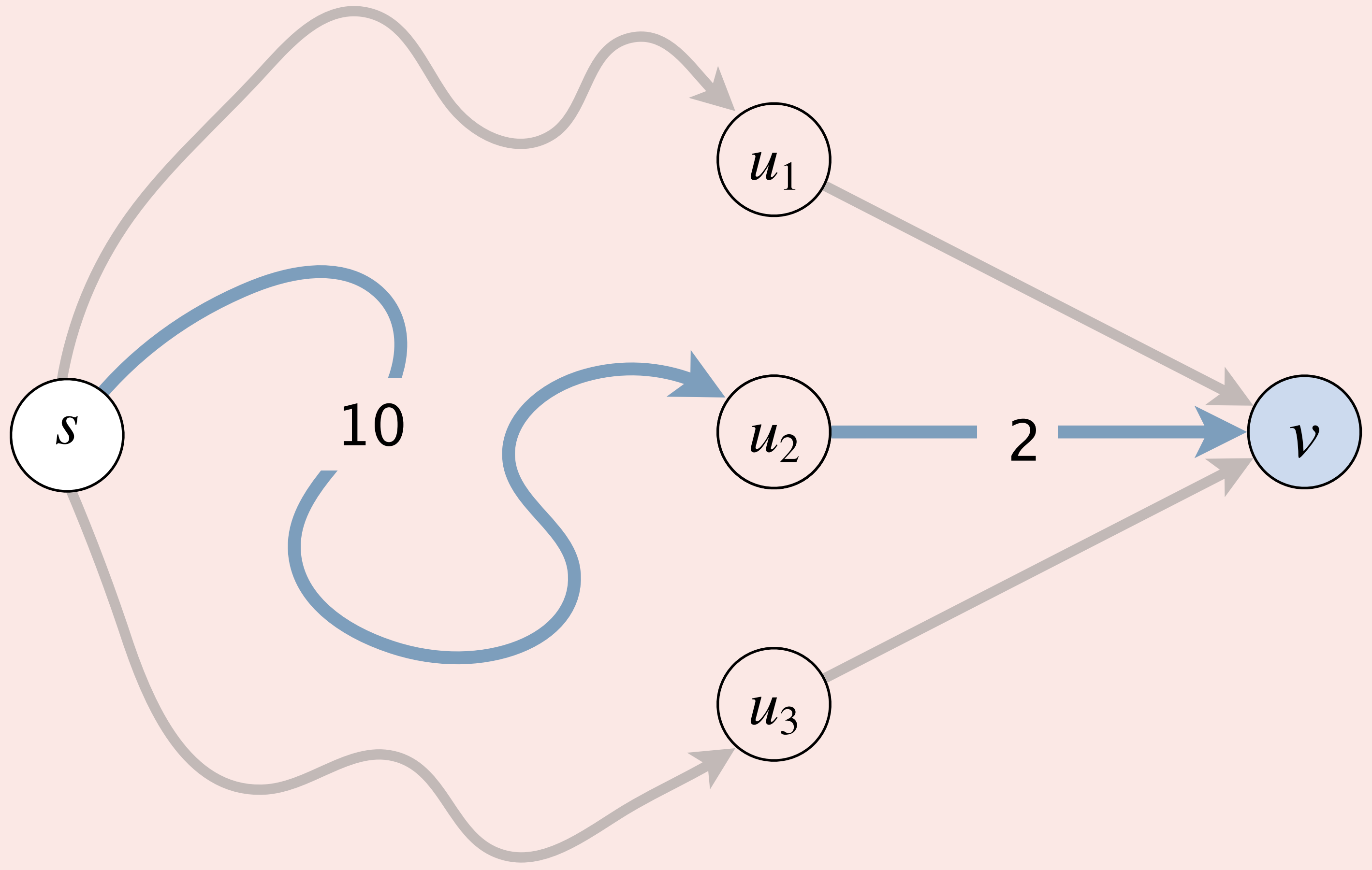
$$distTo(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{e = u \to v} \{ distTo(u) + weight(e) \} & \text{if } v \neq s \end{cases}$$

notation: min is over all edges e that enter v
(min is ∞ if no such edges)

Dynamic programming: poll 4

In which vertex order to apply the dynamic programming recurrence?

- **A.** Increasing order of distance from s.
- **B.** Topological order.
- **C.** Reverse topological order.
- **D.** All of the above.



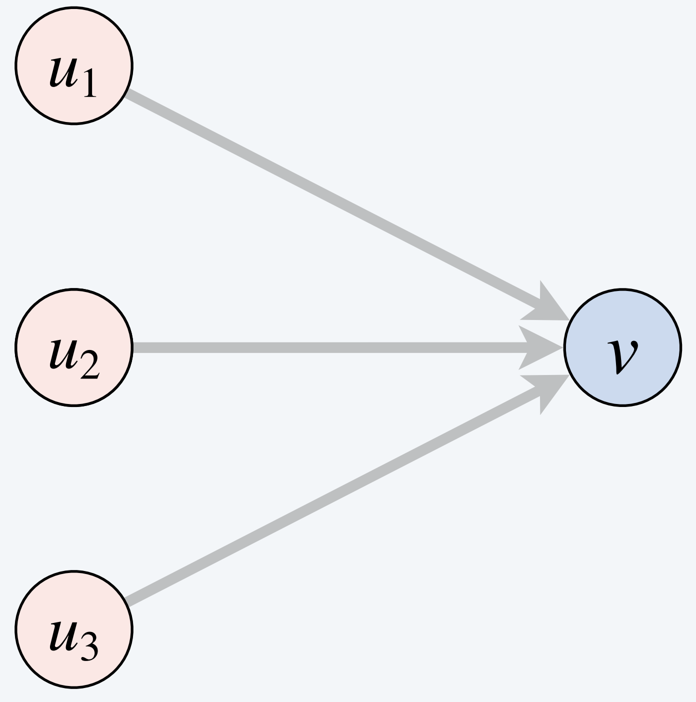
$$distTo(v) = \begin{cases} 0 & \text{if } v = s \\ \min_{e \,=\, u \to v} \{\, distTo(u) + weight(e) \,\} & \text{if } v \neq s \end{cases}$$

Shortest paths in directed acyclic graphs: bottom-up solution

Bottom-up DP implementation. Takes $\Theta(E + V)$ time with two tricks:

- Solve subproblems in topological order. ← *ensures that "small" subproblems are solved before "large" ones*
- Build reverse digraph G^R (to support iterating over edges incident to vertex v).

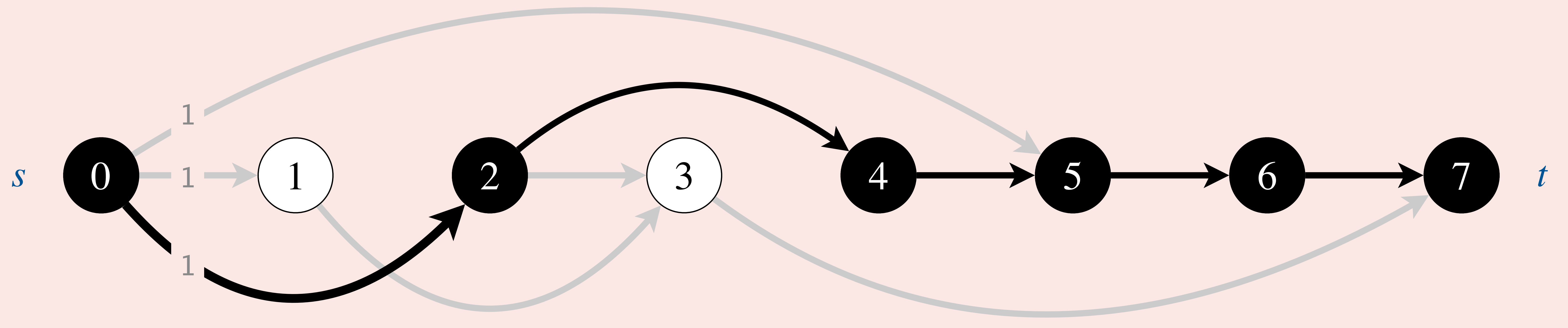
Equivalent (but simpler) computation. Relax vertices in topological order.




```
Topological topological = new Topological(G);
for (int v : topological.order())
   for (DirectedEdge e : G.adj(v))
      relax(e);
```

Backtracing. Can find the shortest paths themselves by maintaining `edgeTo[]` array.

Given a DAG, how to find longest path from s to t in $\Theta(E+V)$ time?



longest path from s to t in a DAG (all edge weights = 1)

A. Negate edge weights; use DP algorithm to find shortest path.

B. Replace *min* with *max* in DP recurrence.

C. Either A or B.

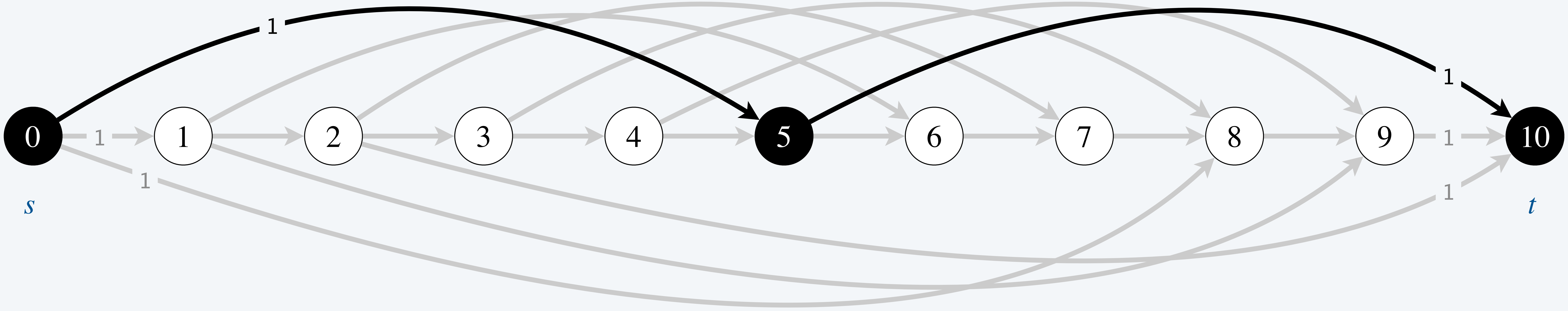
D. No poly-time algorithm is known (**NP**-complete).

Shortest paths in DAGs and dynamic programming

DP subproblem dependency digraph.

- Vertex v corresponds to subproblem v.
- Edge $v \to w$ means subproblem v must be solved before subproblem w.
- Digraph must be a DAG. Why?

Ex 1. Modeling the coin changing problem as a shortest path problem in a DAG.



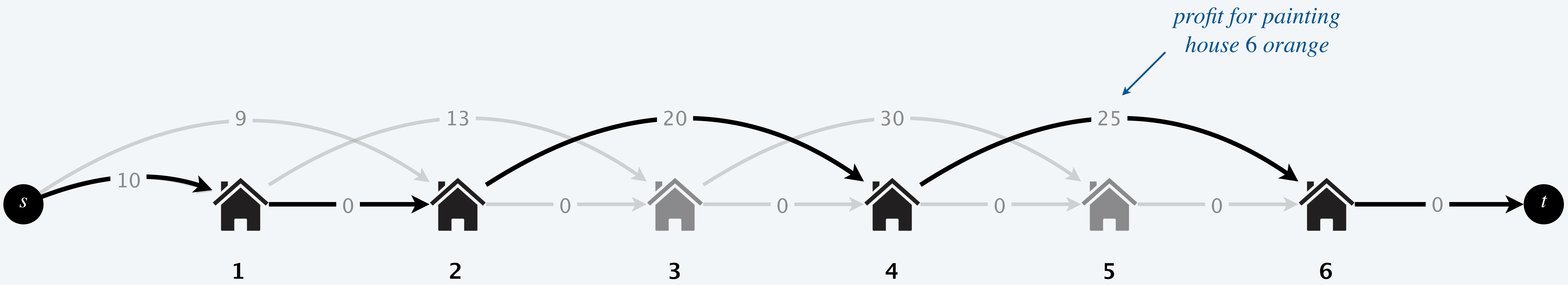
coin denominations = { 1, 5, 8 }, V = 10

Shortest paths in DAGs and dynamic programming

DP subproblem dependency digraph.

- Vertex v corresponds to subproblem v.
- Edge $v \rightarrow w$ means subproblem v must be solved before subproblem w.
- Digraph must be a DAG. Why?

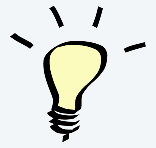
Ex 2. Modeling the house painting problem as a longest path problem in a DAG.

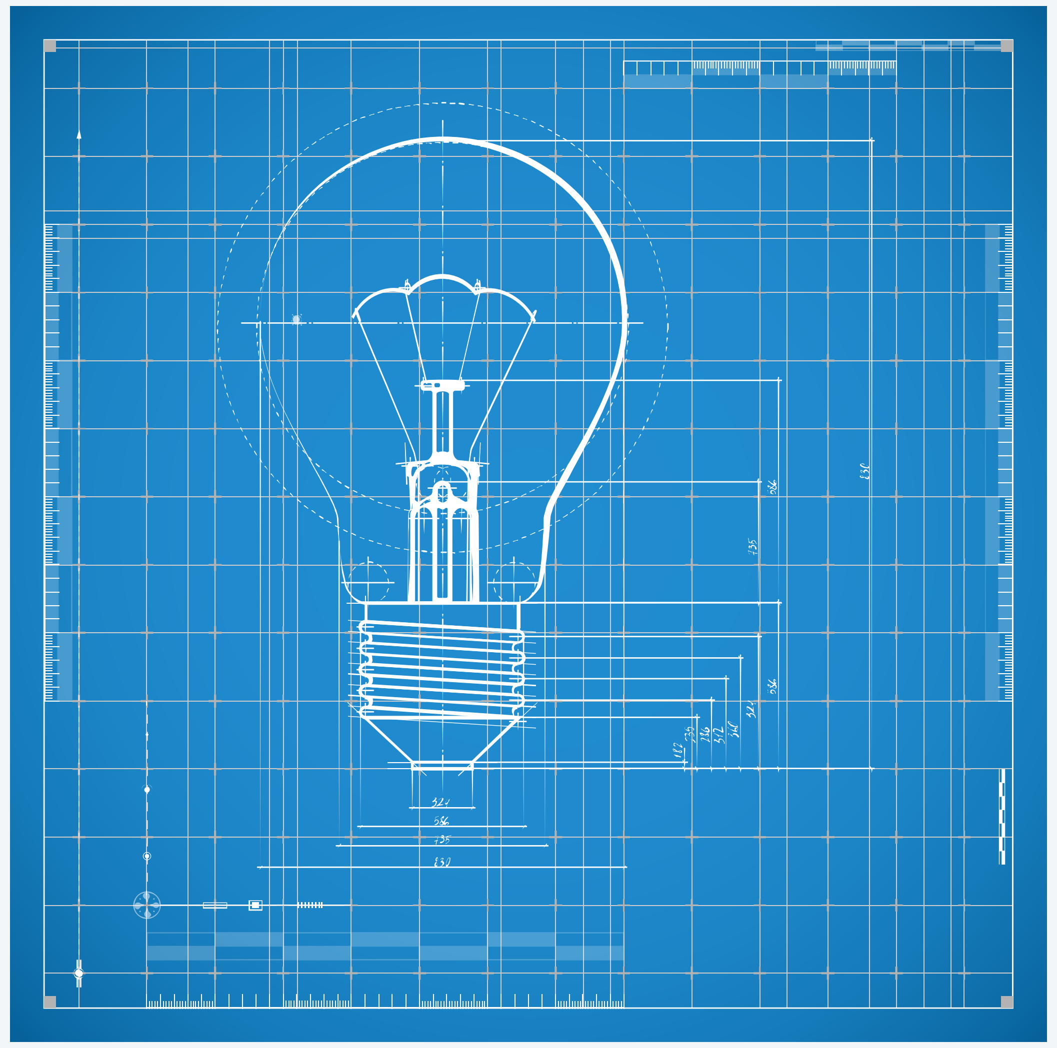


n = 6; profits = { 10, 9, 13, 20, 30, 25 }

Summary

How to design a dynamic programming algorithm.

- Find good subproblems. 
- Develop DP recurrence for optimal value.
  - optimal substructure
  - overlapping subproblems
- Determine dependency order in which to solve subproblems.
- Cache computed results to avoid unnecessary re-computation.
- Reconstruct the optimal solution via backtracing.



Credits

image	source	license
Richard Bellman	Wikipedia	
Biopython	biopython.org	
ImageMagick Liquid Rescale	ImageMagick	ImageMagick license
Cubic B-Spline	Tibor Stanko	
Leonardo Fibonacci	Wikimedia	public domain
Evoke 5 Vending Machine	U-Select-It	
U.S. Coins	Adobe Stock	education license
Seam Carving	Avidan and Shamir	
Broadway Tower	Wikimedia	CC BY 2.5
Blueprint of Light Bulb	Adobe Stock	education license
A is for Algorithms	Reddit	

Lecture Slides © Copyright 2025 Robert Sedgewick and Kevin Wayne

A final thought