DYNAMIC PROGRAMMING

- introduction
- Fibonacci numbers
- interview problems
- shortest paths in DAGs
- seam carving
DYNAMIC PROGRAMMING

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

https://algs4.cs.princeton.edu
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)

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.
- De Boor for evaluating spline curves.
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Avidan–Shamir for seam carving.
- \textbf{NP}-complete graph problems on trees (vertex color, vertex cover, independent set, ...).
- ...
Dynamic programming books
DYNAMIC PROGRAMMING

- introduction
- Fibonacci numbers
- interview problems
- shortest paths in DAGs
- seam carving
Fibonacci numbers

**Fibonacci numbers.** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

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

Leonardo Fibonacci
Fibonacci numbers: naïve recursive approach

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

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

Goal. Given \( n \), compute \( F_n \).

Naïve recursive approach:

```java
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: quiz 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 $\text{fib}(6)$:
- $\text{fib}(5)$ is called 1 time.
- $\text{fib}(4)$ is called 2 times.
- $\text{fib}(3)$ is called 3 times.
- $\text{fib}(2)$ is called 5 times.
- $\text{fib}(1)$ is called $F_n = F_6 = 8$ times.

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

![Recursion tree diagram]

**running time = # subproblems \times cost per subproblem**
Fibonacci numbers: top-down dynamic programming

**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.

```java
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 to compute \( F_n \).
Fibonacci numbers: bottom-up dynamic programming

Bottom-up dynamic programming.
- Build computation from the “bottom up.”
- Solve small subproblems and save solutions.
- Use those solutions to solve larger subproblems.

```java
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];
}
```

Impact. Solves each subproblem $F_i$ only once; $\Theta(n)$ time to compute $F_n$; no recursion.
Fibonacci numbers: further improvements

Performance improvements.

- Save space by saving only two most recent Fibonacci numbers.

```java
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;
}
```

- Exploit additional properties of problem:

\[
F_n = \begin{bmatrix} \phi^n \\ \sqrt{5} \end{bmatrix}, \quad \phi = \frac{1 + \sqrt{5}}{2}
\]

\[
\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}^n
\]
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 the $i^{th}$ Fibonacci number ]

- 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 each of these subproblems, solving each subproblem only once.
  [ use an array, storing subproblem $i$ in $f[i]$ ]

- Use stored solutions to solve the original problem.
  [ subproblem $n$ is original problem ]
Dynamic Programming

- introduction
- Fibonacci numbers
- interview problems
- shortest paths in DAGs
- seam carving
Goal. Install WiFi routers in a row of $n$ houses so that:

- Minimize total cost, where $\text{cost}(i) = \text{cost to install a router at house } i$.
- Requirement: no two consecutive houses without a router.

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost($i$)</td>
<td>1</td>
<td>4</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

cost to install router at house $i$
$(4 + 8 + 9 = 21)$
**Router Installation Problem:** Dynamic Programming Formulation

**Goal.** Install WiFi routers in a row of $n$ houses so that:

- Minimize total cost, where $cost(i) =$ cost to install a router at house $i$.
- Requirement: no two consecutive houses without a router.

**Subproblems.**

- $yes(i) = \min \text{ cost to install router at houses } 1, \ldots, i \text{ with router at } i.$
- $no(i) = \min \text{ cost to install router at houses } 1, \ldots, i \text{ with no router at } i.$
- Optimal cost = $\min \{ yes(n), no(n) \}.$

**Dynamic programming recurrence.**

- $yes(0) = no(0) = 0$
- $yes(i) = cost(i) + \min \{ yes(i - 1), no(i - 1) \}$
- $no(i) = yes(i - 1)$

“optimal substructure”
(optimal solution can be constructed from optimal solutions to smaller subproblems)
A mutually recursive implementation.

```java
private int yes(int i)
{
    if (i == 0) return 0;
    return cost[i] + Math.min(yes(i-1), no(i-1));  // yes(i) = cost(i) + min { yes(i-1), no(i-1) }
}

private int no(int i)
{
    if (i == 0) return 0;
    return yes(i-1);  // no(i) = yes(i-1)
}

public int minCost()
{
    return Math.min(yes(n), no(n));
}
```
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. \( \Theta(n!) \)
“Those who cannot remember the past are condemned to repeat it.”

— Dynamic Programming

(Jorge Agustín Nicolás Ruiz de Santayana y Borrás)
**Router Installation: Bottom-up Implementation**

Bottom-up DP implementation.

```java
int[] yes = new int[n+1];
int[] no  = new int[n+1];
for (int i = 1; i <= n; i++)
    { 
yes[i] = cost[i] + Math.min(yes[i-1], no[i-1]);
    no[i]  = yes[i-1];
    }
return Math.min(yes[n], no[n]);
```

**Proposition.** Takes $\Theta(n)$ time and uses $\Theta(n)$ extra space.

**Remark.** Could eliminate the `no[]` array by substituting identity `no[k] = yes[k-1]`. 
So far: we’ve computed the value of the optimal solution.

Still need: the solution itself (where to install routers).

\[
\begin{array}{cccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
  yes(i) & 0 & 1 & 4 & 13 & 12 & 21 & 23 \\
  no(i) & 0 & 0 & 1 & 4 & 13 & 12 & 21 \\
\end{array}
\]

yes(i) = cost to install routers at houses 1, 2, …, i with router at house i
no(i) = cost to install routers at houses 1, 2, …, i with router not at house i
Coin Changing

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 = $\{1, 10, 25, 100\}$, $V = 130$.

Greedy (8 coins). $131\text{¢} = 100 + 25 + 1 + 1 + 1 + 1 + 1 + 1$.

Optimal (5 coins). $131\text{¢} = 100 + 10 + 10 + 10 + 1$.

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

vending machine (out of nickels)
**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 $d_i \leq v$ for some $i$.
- Use fewest coins to make change for $v - d_i$.

**Dynamic programming recurrence.**

$$OPT(v) = \begin{cases} 0 & \text{if } v = 0 \\ \min \limits_{i : d_i \leq v} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0 \end{cases}$$
Dynamic programming: quiz 3

In which order to compute \( OPT(v) \) in bottom–up DP?

A. Increasing \( i \).

\[
\text{for (int } v = 1; v <= V; v++)
\text{ }
\text{opt[v]} = \ldots
\]

B. Decreasing \( i \).

\[
\text{for (int } v = V; v >= 1; v--)
\text{ }
\text{opt[v]} = \ldots
\]

C. Either A or B.

D. Neither A nor B.

\[
OPT(v) = \begin{cases} 
0 & \text{if } v = 0 \\
\min_{i: d_i \leq v} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0
\end{cases}
\]
**Coin Changing: Bottom-Up Implementation**

Bottom-up DP implementation.

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

for (int v = 1; v <= V; v++)
{
    // opt[v] = min_i { 1 + opt[v - d[i]] }
    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]]);
    }
}
```

\[
OPT(v) = \begin{cases} 
0 & \text{if } v = 0 \\
\min_{i : d_i \leq v} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0 
\end{cases}
\]

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

**Note.** Not polynomial in input size; underlying problem is \(\textbf{NP}\)-complete.
DYNAMIC PROGRAMMING

- introduction
- Fibonacci numbers
- interview problems
- shortest paths in DAGs
- seam carving
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.** $\text{distTo}(v) =$ length of shortest $s \rightarrow v$ path.

**Goal.** $\text{distTo}(t)$.

**Multiway choice.** To compute $\text{distTo}(v)$:

- Select an edge $e = u \rightarrow v$ entering $v$.
- Combine with shortest $s \rightarrow u$ path.

**Dynamic programming recurrence.**

$$
\text{distTo}(v) = \begin{cases} 
0 & \text{if } v = s \\
\min_{e = u \rightarrow v} \{ \text{distTo}(u) + \text{weight}(e) \} & \text{if } v \neq s
\end{cases}
$$
Shortest paths in directed acyclic graphs: bottom-up solution

**Bottom-up DP algorithm.** Takes $\Theta(E + V)$ time with two tricks:
- Solve subproblems in **topological order.** ensures that “small” subproblems are solved before “large” ones
- Form reverse digraph $G^R$ (to support iterating over edges incident to vertex $v$).

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

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

**Remark.** 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?

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 $v$ for each subproblem $v$.
- Edge $v \rightarrow w$, if 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.

V = 10; coin denominations = \{ 1, 5, 8 \}
Shortest paths in DAGs and dynamic programming

**DP subproblem dependency digraph.**
- Vertex $v$ for each subproblem $v$.
- Edge $v \rightarrow w$, if subproblem $v$ must be solved before subproblem $w$.
- Digraph must be a DAG. Why?

**Ex 2.** Modeling the router installation problem as a shortest path problem in a DAG.
4.4 Shortest Paths

- introduction
- Fibonacci numbers
- interview problems
- shortest paths in DAGs
- seam carving
Content-aware resizing

Seam carving. [Avidan–Shamir] Resize an image without distortion for display on cell phones and web browsers.

https://www.youtube.com/watch?v=vlFCV2spKtg
Content-aware resizing

**Seam carving.** [Avidan–Shamir]  Resize an image without distortion for display on cell phones and web browsers.

---

**In the wild.**  Photoshop, ImageMagick, GIMP, ...
Content-aware resizing

To find vertical seam in a picture:

- Grid graph: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = “energy function” of 8 neighboring pixels.
Content-aware resizing

To find vertical seam in a picture:

- Grid graph: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = “energy function” of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.
To remove vertical seam in a picture:

- Delete pixels on seam (one in each row).
Content-aware resizing: dynamic programming formulation

**Problem.** Find a min energy path from top to bottom.

**Subproblems.** \( \text{distTo}(col, row) = \text{energy of min energy path from any top pixel to pixel } (col, row). \)

**Goal.** \( \min \{ \text{distTo}(col, H-1) \}. \)
Summary

How to design a dynamic programming algorithm.

- Find good subproblems.
- Develop DP recurrence for optimal value.
  - optimal substructure
  - overlapping subproblems
- Determine order in which to solve subproblems.
- Cache computed results to avoid unnecessary re-computation.
- Reconstruct the solution: backtrace or save extra state.