Algorithm Design

- analysis of algorithms
- greed
- reduction
- dynamic programming
- divide-and-conquer
- randomization

https://algs4.cs.princeton.edu
Algorithm design

Algorithm design patterns.
- Analysis of algorithms.
- Greed.
- Reduction.
- Dynamic programming.
- Divide-and-conquer.
- Randomization.

Want more? See COS 340, COS 343, COS 423, COS 445, COS 451, COS 488, ....
INTERVIEW QUESTIONS
Algorithm Design

- analysis of algorithms
- greed
- reduction
- dynamic programming
- divide-and-conquer
- randomization
**Goal.** Find $T$ using fewest drops.
**Goal.** Find $T$ using fewest drops.

**Variant 0.** 1 egg.

**Solution.** Use sequential search: drop on floors 1, 2, 3, ... until egg breaks.

**Analysis.** 1 egg and $T$ drops.

---

Running time depends upon a parameter that you don't know a priori.
**EGG DROP**

**Goal.** Find $T$ using fewest drops.

**Variant 1.** $\infty$ eggs.

**Solution.** Binary search for $T$.

- Initialize $[lo, hi] = [0, n+1]$.
- Maintain invariant: egg breaks on floor $hi$ but not on $lo$.
- Repeat until length of interval is 1:
  - drop on floor $mid = \lfloor (lo + hi) / 2 \rfloor$.
  - if it breaks, update $hi = mid$.
  - if it doesn’t break, update $lo = mid$.

**Analysis.** $\sim \log_2 n$ eggs, $\sim \log_2 n$ drops.

Suppose $T$ is much smaller than $n$. Can you guarantee $\Theta(\log T)$ drops?
**EGG DROP**

**Goal.** Find $T$ using fewest drops.

**Variant 1’.** $\infty$ eggs and $\Theta(\log T)$ drops.

**Solution.** Use repeated doubling; then binary search.

- Drop on floors 1, 2, 4, 8, 16, ..., $x$ to find a floor
  
  $x$ such that the egg breaks on floor $x$ but not on $\frac{1}{2}x$.

- Binary search in interval $[\frac{1}{2}x, x]$.

**Analysis.** $\sim \log_2 T$ eggs, $\sim 2 \log_2 T$ drops.

- Repeated doubling: 1 egg and $1 + \log_2 x$ drops.

- Binary search: $\sim \log_2 x$ eggs and $\sim \log_2 x$ drops.

- Note that $T \leq x < 2T$. 
Algorithm design: quiz 1

Goal. Find $T$ using fewest drops.

Variant 2. 2 eggs.

In worst case, how many drops needed as a function of $n$?

A. $\Theta(1)$
B. $\Theta(\log n)$
C. $\Theta(\sqrt{n})$
D. $\Theta(n)$
**Egg Drop (Asymmetric Search)**

**Goal.** Find $T$ using fewest drops.

**Variant 2.** 2 eggs.

**Solution.** Use gridding; then sequential search.
- Drop at floors $\sqrt{n}$, $2\sqrt{n}$, $3\sqrt{n}$, ... until first egg breaks, say at floor $c\sqrt{n}$.
- Sequential search in interval $[c\sqrt{n} - \sqrt{n}, c\sqrt{n}]$.

**Analysis.** At most $2\sqrt{n}$ drops.
- First egg: $\leq \sqrt{n}$ drops.
- Second egg: $\leq \sqrt{n}$ drops.

**Signing bonus 1.** Use 2 eggs and at most $\sqrt{2n}$ drops.

**Signing bonus 2.** Use 3 eggs and at most $3n^{1/3}$ drops.
ALGORITHM DESIGN

- analysis of algorithms
- greed
- reduction
- dynamic programming
- divide-and-conquer
- randomization

https://algs4.cs.princeton.edu
Greedy algorithms

Make locally optimal choices at each step.

Familiar examples.
- Prim’s algorithm. [for MST]
- Kruskal’s algorithm. [for MST]
- Dijkstra’s algorithm. [for shortest paths]
- Huffman’s algorithm. [for data compression]

More classic examples.
- $A^*$ search algorithm.
- Gale–Shapley stable marriage.
- Greedy algorithm for matroids.
- ...

Caveat. Greedy algorithms rarely lead to provably optimal solutions.

[ but often used anyway in practice, especially for intractable problems ]
**Coin changing problem and cashier’s algorithm**

**Goal.** Given U. S. coin denominations \{ 1, 5, 10, 25, 100 \}, devise a method to pay amount to customer using fewest coins.

**Ex.** 34¢.  
![6 coins](image)

**Cashier’s (greedy) algorithm.** Repeatedly add the coin of the largest value that does not exceed the remaining amount to be paid.

**Ex.** $2.89.  
![10 coins](image)
Is the cashier’s algorithm optimal for U.S. coin denominations \{ 1, 5, 10, 25, 100 \}? 

A. Yes, greedy algorithms are always optimal.

B. Yes, for any set of coin denominations \( d_1 < d_2 < \ldots < d_n \) provided \( d_1 = 1 \).

C. Yes, because of special properties of U.S. coin denominations.

D. No.
Properties of any optimal solution (for U.S. coin denominations)

Property 1. Number of pennies $P \leq 4$.
Proof. Replace 5 pennies with 1 nickel.

Property 2. Number of nickels $N \leq 1$.

Property 3. Number of dimes $D \leq 2$.

Property 4. Number of quarters $Q \leq 3$.

Property 5. $N + D \leq 2$.
Proof.
- Properties 2 and 3 $\Rightarrow N \leq 1$ and $D \leq 2$.
- If $N = 1$ and $D = 2$, replace with 1 quarter.


- Property 1 contributes at most 4
- Property 5 contributes at most 20
- Property 4 contributes at most 75
Optimality of cashier’s algorithm (for U.S. coin denominations)

**Proposition.** Cashier’s algorithm yields unique optimal solution for denominations \{ 1, 5, 10, 25, 100 \}.

**Pf.** [ for dollar coins ]

- Suppose we are changing amount $x.yz$.
- Cashier’s algorithm takes $x$ dollar coins.
- Suppose (for the sake of contradiction) that an optimal solution takes fewer than $x$ dollar coins.
- Then, optimal solution satisfies $P + 5N + 10D + 25Q \geq 100$.
- This contradicts Property 6:

\[
P + 5N + 10D + 25Q \leq 99
\]

[ similar arguments justify greedy strategy for quarters, dimes, and nickels ]
Algorithm Design

- analysis of algorithms
- greed
- reduction
- dynamic programming
- divide-and-conquer
- randomization
Problem $X$ reduces to problem $Y$ if you can solve $X$ by using an algorithm for $Y$.

**Ex 1.** Finding the median reduces to sorting.
**Ex 2.** Bipartite matching reduces to maxflow.

Many many problems reduce to:
- Sorting.
- Maxflow.
- Suffix array.
- Shortest path.
- Minimum spanning tree.
- Linear/semidefinite programming. see ORF 307 or ORF 363
- ...

**Note.** Reductions also play central role in computational complexity (e.g., NP-completeness).
**Goal.** Given a digraph, where each edge has a positive weight and is orange or black, find shortest path from \( s \) to \( t \) that uses at most \( k \) orange edges.

\[
\begin{align*}
  k = 0: & \quad s \rightarrow 1 \rightarrow t \quad (17) \\
  k = 1: & \quad s \rightarrow 3 \rightarrow t \quad (13) \\
  k = 2: & \quad s \rightarrow 2 \rightarrow 3 \rightarrow t \quad (11) \\
  k = 3: & \quad s \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow t \quad (10) \\
  k = 4: & \quad s \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow t \quad (10)
\end{align*}
\]
**Shortest path with orange and black edges**

**Goal.** Given a digraph, where each edge has a positive weight and is orange or black, find shortest path from $s$ to $t$ that uses at most $k$ orange edges.

**Solution.**
- Create $k+1$ copies of the vertices in digraph $G$, labeled $G_0, G_1, \ldots, G_k$.
- For each black edge $v \rightarrow w$: add edge from vertex $v$ in graph $G_i$ to vertex $w$ in $G_i$.
- For each orange edge $v \rightarrow w$: add edge from vertex $v$ in graph $G_i$ to vertex $w$ in $G_{i+1}$.
- Compute shortest path from $s$ to any copy of $t$.

---

**Diagram:**
- $G_0, G_1, G_2$ with edges and vertices labeled accordingly.
- $k = 2$.
What is worst–case running time of algorithm as a function of k, the number of vertices V, and the number of edges E? Assume E ≥ V.

A. $\Theta(E \log V)$

B. $\Theta(k E)$

C. $\Theta(k E \log V)$

D. $\Theta(k^2 E \log V)$
ALGORITHM DESIGN

- analysis of algorithms
- greed
- reduction
- dynamic programming
- divide-and-conquer
- randomization

https://algs4.cs.princeton.edu
Dynamic programming

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

[ caching solutions to subproblems in a table for later reuse ]

Familiar examples.
- Bellman–Ford.
- Seam carving.
- Shortest paths in DAGs.

More classic examples.
- Unix diff.
- Viterbi algorithm for hidden Markov models.
- CKY algorithm for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for DNA sequence alignment.
- ...
**Goal.** Paint a row of $n$ houses red, green, or blue so that:

- Minimize total cost, where $\text{cost}(i, \text{color})$ is cost to paint $i$ given color.
- No two adjacent houses have the same color.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{cost}(i, \text{red})$</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td>$\text{cost}(i, \text{green})$</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>22</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>$\text{cost}(i, \text{blue})$</td>
<td>16</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Cost to paint house $i$ the given color

$3 + 6 + 4 + 8 + 5 + 8 = 34$
**House Coloring Problem: Dynamic Programming Formulation**

**Goal.** Paint a row of $n$ houses red, green, or blue so that:

- Minimize total cost, where $\text{cost}(i, \text{color})$ is cost to paint $i$ given color.
- No two adjacent houses have the same color.

**Subproblems.**

- $R(i) = \min \text{ cost to paint houses } 1, \ldots, i \text{ with } i \text{ red.}$
- $G(i) = \min \text{ cost to paint houses } 1, \ldots, i \text{ with } i \text{ green.}$
- $B(i) = \min \text{ cost to paint houses } 1, \ldots, i \text{ with } i \text{ blue.}$
- Optimal cost $= \min \{ R(n), G(n), B(n) \}$.

**Dynamic programming recurrence.**

- $R(0) = G(0) = B(0) = 0$
- $R(i) = \text{cost}(i, \text{red}) + \min \{ G(i - 1), B(i - 1) \}$
- $G(i) = \text{cost}(i, \text{green}) + \min \{ B(i - 1), R(i - 1) \}$
- $B(i) = \text{cost}(i, \text{blue}) + \min \{ R(i - 1), G(i - 1) \}$

"optimal substructure"

(optimal solution can be constructed from optimal solutions to smaller subproblems)
Bottom-up DP trace. Given $R(i)$, $G(i)$, and $B(i)$, easy to compute $R(i+1)$, $G(i+1)$, and $B(i+1)$.

\[
B(6) = \text{cost}(6, \text{blue}) + \min \{ R(5), G(5) \} \\
= 7 + \min \{ 29, 32 \} \\
= 36
\]
**House Coloring: Bottom-up Implementation**

Bottom-up DP implementation.

```java
int[] r = new int[n+1];
int[] g = new int[n+1];
int[] b = new int[n+1];

for (int i = 1; i <= n; i++) {
    r[i] = cost[i][RED] + Math.min(g[i-1], b[i-1]);
    g[i] = cost[i][GREEN] + Math.min(b[i-1], r[i-1]);
    b[i] = cost[i][BLUE] + Math.min(r[i-1], g[i-1]);
}

return min3(r[n], g[n], b[n]);
```

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

- analysis of algorithms
- greed
- reduction
- dynamic programming
- divide-and-conquer
- randomization
Divide and conquer

- Break up problem into two or more independent subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems to form solution to original problem.

Familiar examples.
- Mergesort.
- Quicksort.

More classic examples.
- Closest pair.
- Convolution and FFT.
- Matrix multiplication.
- Integer multiplication.
  ...

Prototypical usage. Turn brute-force $\Theta(n^2)$ algorithm into $\Theta(n \log n)$ one.
Personalized recommendations

Music site tries to match your song preferences with others.

- Your ranking of songs: 0, 1, ..., n–1.
- My ranking of songs: \( a_0, a_1, \ldots, a_{n-1} \).
- Music site consults database to find people with similar tastes.

**Kendall-tau distance.** Number of inversions between two rankings.

**Inversion.** Songs \( i \) and \( j \) are inverted if \( i < j \), but \( a_i > a_j \).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>you</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>me</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

3 inversions: 2–1, 3–1, 7–6
Problem. Given a permutation of length $n$, count the number of inversions.

$$\begin{array}{cccccccc}
0 & 2 & 3 & 1 & 4 & 5 & 7 & 6 \\
\end{array}$$

3 inversions: 2–1, 3–1, 7–6

Brute-force $\Theta(n^2)$ algorithm. For each $i < j$, check if $a_i > a_j$.

A bit better. Run insertion sort; return number of exchanges.

Goal. $\Theta(n \log n)$ time (or better).
### COUNTING INVERSIONS: DIVIDE-AND-CONQUER

<table>
<thead>
<tr>
<th>input</th>
<th>0</th>
<th>4</th>
<th>3</th>
<th>7</th>
<th>9</th>
<th>1</th>
<th>5</th>
<th>8</th>
<th>2</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>count inversions in left subarray</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>4-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count inversions in right subarray</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5-2</td>
<td>8-2</td>
<td>8-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count inversions with one element in each subarray</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>3-1</td>
<td>3-2</td>
<td>4-1</td>
<td>4-2</td>
<td>7-1</td>
<td>7-2</td>
<td>7-5</td>
<td>7-6</td>
<td>9-1</td>
<td>9-2</td>
</tr>
<tr>
<td>output</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 + 3 + 13 = 17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This step seems to require \( \Theta(n^2) \) time.
**COUNTING INVERSIONS: DIVIDE-AND-CONQUER**

<table>
<thead>
<tr>
<th>Input</th>
<th>0 4 3 7 9 1 5 8 2 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count inversions in left subarray and sort</td>
<td>0 3 4 7 9 1 5 8 2 6</td>
</tr>
<tr>
<td>Count inversions in right subarray and sort</td>
<td>0 3 4 7 9 1 2 5 6 8</td>
</tr>
<tr>
<td>Count inversions with one element in each sorted subarray</td>
<td>0 3 4 7 9 1 2 5 6 8</td>
</tr>
<tr>
<td>And merge into sorted whole</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

1 + 3 + 13 = 17
What is running time of algorithm as a function of $n$?

A. $\Theta(n)$
B. $\Theta(n \log n)$
C. $\Theta(n \log^2 n)$
D. $\Theta(n^2)$
ALGORITHM DESIGN

- analysis of algorithms
- greed
- reduction
- dynamic programming
- divide-and-conquer
- randomization
Randomized algorithms

Algorithm whose performance (or output) depends on the results of random coin flips.

Familiar examples.
- Quicksort.
- Quickselect.

More classic examples.
- Miller–Rabin primality testing.
- Rabin–Karp substring search.
- Polynomial identity testing.
- Volume of convex body.
- Universal hashing.
- Global min cut.
...
**Problem.** A disorganized carpenter has a mixed pile of \( n \) nuts and \( n \) bolts.

- The goal is to find the corresponding pairs of nuts and bolts.
- Each nut fits exactly one bolt; each bolt fits exactly one nut.
- By fitting a nut and a bolt together, the carpenter can determine which is bigger.

**Brute-force algorithm.** Compare each bolt to each nut: \( \Theta(n^2) \) compares.

**Challenge.** Design an algorithm that makes \( O(n \log n) \) compares.
**Nuts and Bolts**

**Shuffle.** Shuffle the nuts and bolts.

**Partition.**
- Pick leftmost bolt $i$ and compare against all nuts; divide nuts smaller than $i$ from those that are larger than $i$.
- Let $i'$ be the nut that matches bolt $i$. Compare $i'$ against all bolts; divide bolts smaller than $i'$ from those that are larger than $i'$.

**Divide-and-conquer.** Recursively solve two subproblems.
What is the expected running time of the randomized algorithm as a function of n?

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n \log^2 n)$

D. $\Theta(n^2)$
Chapter 27
Matching Nuts and Bolts in $O(n \log n)$ Time
(Extended Abstract)

János Komlós1,4  Yuan Ma2  Endre Szemerédi2,4

Abstract

Given a set of $n$ nuts of distinct widths and a set of $n$ bolts such that each nut corresponds to a unique bolt of the same width, how should we match every nut with its corresponding bolt by comparing nuts with bolts (no comparison is allowed between two nuts or between two bolts)? The problem can be naturally viewed as a variant of the classic sorting problem as follows. Given two lists of $n$ numbers each such that one list is a permutation of the other, how should we sort the lists by comparisons only between numbers in different lists? We give an $O(n \log n)$-time deterministic algorithm for the problem. This is optimal up to a constant factor and answers an open question posed by Alon, Blum, Fiat, Kannan, Naor, and Ostrovsky [3]. Moreover, when copies of nuts and bolts are allowed, our algorithm runs in optimal $O(\log n)$ time on $n$ processors in Valiant’s parallel comparison tree model. Our algorithm is based on the AKS sorting algorithm with substantial modifications.

Hiring bonus. Algorithm that takes $O(n \log n)$ time in the worst case.
ALGORITHM DESIGN

- analysis of algorithms
- greed
- reduction
- dynamic programming
- divide-and-conquer
- randomization
- credits
Co-instructors and graduate student AIs.

Precept facilitators. Aliya, Alkin, Allison, Ananya, Justin, Harvey, Ryan, and Tejas.

Undergrad graders and lab TAs. Apply to be one next semester!
A farewell video (from P04, Fall 2018)

COS 226 P04 Presents..
“Algorithms and data structures are love.
Algorithms and data structures are life.”

— anonymous COS 226 student