6.006 Introduction to Algorithms
Lecture Notes

Jason Ku & Erik Demaine, MIT

Yangming Li, self-study notes of MIT 6.006 (Spring 2020)

1. Course Objectives

- **Primary goal:** Teach you to solve computational problems.
- **Key skills:**
  - Designing correct algorithms and data structures.
  - Proving correctness (via induction, invariants).
  - Analyzing efficiency (asymptotic running time).
  - Communicating solutions clearly in writing.
- **Assessment:** Three quizzes covering
  1. Data structures & sorting
  2. Graph algorithms (shortest paths, etc.)
  3. Dynamic programming

2. Computational Problems & Algorithms

2.1 Formalizing a Problem

Inputs: a set I
Outputs: a set O
Problem specification: a relation $R \subseteq I \times O$; for each $i \in I$, the valid outputs $o \in O$ satisfy $(i, o) \in R$.
Predicate form: often given by a Boolean predicate $P(i, o)$.
Example: "Do any two students share the same birthday?"

2.2 Definition of an Algorithm

- A finite procedure (function) mapping each input $i \in I$ to exactly one output $a(i)$.
- *Correctness:* $\forall\, i \in I,\ P(i,\, a(i))$ holds.
- *Example:* Maintain a record of seen birthdays; for each new student, check record—if match, `return` pair; else append to record; at end `return` none.

2.3 Proving Correctness by Induction

Inductive hypothesis: After processing the first k inputs, if a match exists among them, the algorithm has already returned it.
Base case ($k = 0$): Vacuously true.
Inductive step: Show that if it holds for k, it holds for $k + 1$: either a match was found among the first k, or the $(k + 1)$-th element matches one in the record.

3. Efficiency & Asymptotic Analysis

3.1 Measuring Performance

- Abstract away machine speed; count *basic operations* on a Word-RAM.
- Express cost as a function of input size n.
- Use asymptotic notation: $\mathrm{O}(\cdot)$ for upper bounds, $\Omega(\cdot)$ for lower bounds, $\Theta(\cdot)$ for tight bounds.

3.2 Common Growth Classes

$$O(1), \quad O(\log n), \quad O(n), \quad O(n \log n), \quad O(n^c)\ (c > 1), \quad O(2^{\Theta(n)}).$$

4. Word-RAM Model

- Memory = array of w-bit words, byte-addressable, random access in $O(1)$.
- CPU can load/store one word and perform arithmetic/comparison/bitwise ops in $O(1)$.
- Word size w must grow as $\Omega(\log n)$ to address n memory cells.

5. Static Sequences & Arrays

5.1 Static Sequence Interface

`build(x)`: construct sequence from given items.
`length()`: return n.
`get_at(i) / set_at(i,v)`: access or modify element i.
`iter()`: traverse all items in order.

5.2 Implementation: Static Array

- Store items in a contiguous block of n words.
- Access via address base $+\, i$ in $O(1)$.
- build, iter $\in \Theta(n)$, `get/set_at`$\in O(1)$.
- Dynamic updates (insert/delete) require shifting $\Theta(n)$.

6. Dynamic Sequences

6.1 Interface Extension

`insert_at(i,v)`, `delete_at(i)`: insert or remove at position i.
`insert_first/last(v)`, `delete_first/last()`: common special cases.

6.2 Linked List Implementation

- Singly-linked nodes storing `item` and `next` pointer.
- Maintain `head` (and optionally `tail` and `length`).
- `insert_first/delete_first`$\in O(1)$.
- `get_at(i)`, `set_at(i)`, `insert_at(i)`, `delete_at(i)` require walking $O(n)$.

7. Dynamic Arrays ("Amortized Arrays")

- Store items in array of size size $\geq n$, maintain $\frac{1}{2}$size $\leq n \leq$ size.
- `get/set_at`$\in O(1)$ worst-case.
- `insert_last`: if $n <$ size, do $A[n] = v$, $n{+}{=}1$ in $O(1)$. If $n =$ size, allocate new array of size $2n$, copy $\Theta(n)$, then insert.
- *Amortized analysis:* over n inserts from empty, total resizing cost $\sum_{i=1}^{\log n} 2^i = O(n)$. `insert_last` is $O(1)$ amortized.
- `delete_last` similarly $O(1)$ amortized (with occasional shrink).

8. Comparative Summary

Operation	Static Array	Linked List	Dynamic Array
`get/set_at`	$O(1)$	$O(n)$	$O(1)$
`insert_first`	$\Theta(n)$	$O(1)$	$O(n)$
`insert_last`	$\Theta(n)$	$O(1)$	$O(1)$ amortized
`delete_first`	$\Theta(n)$	$O(1)$	$\Theta(n)$
`delete_last`	$\Theta(n)$	$O(n)$	$O(1)$ amortized
`insert/delete_at`	$\Theta(n)$	$O(n)$	$\Theta(n)$

Next up: exploring further data structures (balanced trees, hash tables), advanced amortized analyses, and practical implementations in Python's `list` type.

1 Hashing and Hash Tables

1.1 Why we can't beat $\Theta(\log n)$ in the comparison model

- In the *comparison model*, the only way to distinguish keys is by comparing them:

$$k_1 \overset{?}{<} k_2, \quad k_1 \overset{?}{=} k_2, \quad k_1 \overset{?}{>} k_2.$$

- Any comparison-based search algorithm can be viewed as a *decision tree* whose internal nodes are comparisons and whose leaves are *outputs* (either "found − return item" or "not found").

- To store n distinct keys plus the answer "not found," such a tree needs at least $n + 1$ leaves.

- A binary tree with L leaves has height at least $\lceil \log_2 L \rceil$. Hence any comparison-based `find` must make *at least* $\Omega(\log n)$ comparisons in the worst case.

1.2 Direct-Access Tables ("perfect hashing")

- If keys are integers in a small universe $[0, \ldots, U-1]$, one can allocate an array of size U and store key k at index k. Then

$$\texttt{find, insert, delete} = \Theta(1)$$

worst-case time.

- But if $U \gg n$, this uses $\Theta(U)$ space, which can be prohibitive.

1.3 Hash Tables with Chaining

- Allocate an array ("hash table") of size $m = \Theta(n)$.
- Pick a *hash function* $h : \{0, \ldots, U-1\} \to \{0, \ldots, m-1\}$.
- To `insert` key k: append it to a small auxiliary structure (a *chain*) at bucket $h(k)$.
- To `find` k: go to bucket $h(k)$ and scan its chain linearly looking for k.
- Time depends on the chain length at $h(k)$.

1.4 Universal Hashing

[Universal hash family] Let $p > U$ be prime. For any $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, \ldots, p-1\}$ define

$$h_{a,b}(k) \;=\; \big((a \cdot k + b) \bmod p\big) \bmod m.$$

The family

$$\mathcal{H} = \big\{h_{a,b} : 1 \leq a < p,\ 0 \leq b < p\big\}$$

is called *universal* if for any two distinct keys $x \neq y$,

$$\Pr_{h \leftarrow \mathcal{H}}\big[h(x) = h(y)\big] \;\leq\; \frac{1}{m}.$$

1.5 Expected Chain Length & Running Time

- Fix any key k_i stored in the table. Let

$$X_{ij} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j), \\ 0 & \text{otherwise,} \end{cases}$$

an indicator for "key k_j collides with k_i."

- Then the length of the chain at bucket $h(k_i)$ is

$$X_i \;=\; \sum_{j=1}^{n} X_{ij}.$$

- By linearity of expectation and universality,

$$[X_i] = \sum_{j=1}^{n} [X_{ij}] = 1 + \sum_{j \neq i} \Pr[h(k_i) = h(k_j)] \leq 1 + (n-1)\,\frac{1}{m} = O(1) \quad (\text{if } m = \Theta(n)).$$

- Thus under a random choice of $h \in \mathcal{H}$, each chain has *expected* length $O(1)$.
- Consequently, all three operations—

  `find`, `insert`, `delete`

  —take *expected* $O(1)$ time.

1.6 Summary

- In the comparison model, `find` on n items requires $\Omega(\log n)$ time.
- By using *hash tables* with *universal* hashing and *chaining*, we achieve

$$[\texttt{find, insert, delete}] = O(1), \quad \text{using } O(n) \text{ space.}$$

- This "randomized" worst-case guarantee holds over the choice of hash function and is independent of the key distribution.

Problem Session#2

1. Solving Recurrences via Master Theorem

Recall the standard recurrence

$$T(n) \;=\; a\,T\big(\tfrac{n}{b}\big) \;+\; f(n),$$

and let $\alpha = \log_b a$. There are three main cases:

Case 1: If $f(n) = O(n^{\alpha-\varepsilon})$ for some $\varepsilon > 0$, then

$$T(n) = \Theta\big(n^{\alpha}\big).$$

Case 2: If $f(n) = \Theta\big(n^{\alpha}\log^{k} n\big)$ for some $k \geq 0$, then

$$T(n) = \Theta\big(n^{\alpha}\log^{k+1} n\big).$$

Case 3: If $f(n) = \Omega(n^{\alpha+\varepsilon})$ for some $\varepsilon > 0$ and $a\,f(n/b) \leq c\,f(n)$ for some $c < 1$, then

$$T(n) = \Theta\big(f(n)\big).$$

(a) $T(n) = 2\,T(n/2) + O(\sqrt{n})$ Here $a = 2$, $b = 2 \implies \alpha = 1$, and $f(n) = O(n^{1/2}) = O(n^{1-\frac{1}{2}})$, so Case 1 applies:

$$T(n) = \Theta(n^{\alpha}) = \Theta(n).$$

(b) $T(n) = 8\,T(n/4) + O(n^{3/2})$ Here $a = 8$, $b = 4 \implies \alpha = \log_4 8 = 3/2$, and $f(n) = O(n^{3/2}) = \Theta(n^{\alpha})$, so Case 2 with $k = 0$:

$$T(n) = \Theta\big(n^{3/2}\log n\big).$$

2. Searching an Unbounded Sorted List ("Infinity Stones")

Problem: You have an infinite sequence of planets indexed $1, 2, \ldots$, each with a hidden index. You can query a planet to ask if its index is *less*, *equal*, or *greater* than the target k. Design an $O(\log k)$-time algorithm.

Solution: Exponential + Binary Search

1. *Exponential search:* Probe positions $1, 2, 4, 8, \ldots, 2^m$ until you find $2^m \geq k$. This takes $O(m) = O(\log k)$ queries.
2. *Binary search:* Now $k \in [2^{m-1}, 2^m]$, so binary-search that interval in $O(\log k)$ more queries.

Total time $O(\log k)$.

3. Layered-Image Document Data Structure

Maintain a document of n images (layers) with unique IDs, supporting:

$$\begin{array}{lll} \text{build}() & : O(1), & \\ \text{import}(x) & : O(n), & \text{add } x \text{ on top,} \\ \text{display}() & : O(n), & \text{list all IDs top-to-bottom,} \\ \text{move\_below}(x, y) & : O(\log n), & \text{move layer } x \text{ just underneath } y. \end{array}$$

Idea: Hybrid Set + Sequence

- Keep a *sorted array* of all IDs for $O(\log n)$-time `find` by binary search.
- Keep a *doubly-linked list* to store the current layer order, so that splicing (remove + insert) is $O(1)$ given pointers.
- In the array, store with each key x a pointer to its linked-list node.

Operations

1. `build`: allocate empty array + empty list, $O(1)$.
2. `import`(x): binary-search/insert x in array $= O(n)$; prepend to linked list $= O(1)$.
3. `display`: traverse linked list, $O(n)$.
4. `move_below`(x, y):
   (a) find x, y in array via binary search, $O(\log n)$;
   (b) in the linked list, splice out x and re-insert after y, $O(1)$.

4. West-to-East Brick-Blowing Wolf

Given an array $A[1..n]$ of positive integers (brick counts), define

$$D[i] \;=\; 1 \;+\; \big|\{\, j > i : A[j] < A[i] \}\big| \quad (1 \leq i \leq n).$$

Compute all $D[i]$ in $O(n)$ time.

Naïve: For each i, scan $j = i + 1 \ldots n$ to count all smaller, $O(n^2)$.

Two-Finger ("Sliding Window")

- Notice that $A[i]$ and the set of $\{j > i : A[j] < A[i]\}$ both *monotonically* change as i increases.
- Maintain two pointers $i = 1$ and $j = 1$.
- For each $i = 1 \ldots n$:
  1. Advance j while $j \leq n$ and $A[j] < A[i]$.
  2. Then $D[i] = (j - 1) - i + 1 = j - i$.

- Since neither i nor j ever decreases, each moves at most n steps $O(n)$ total.

"Monotone queue" tricks like this yield many other $O(n)$ solutions.

End of Problem Session #2.

3. Sorting: Beyond $\Theta(n \log n)$

3.1 Comparison–Model Lower Bound

In the *comparison model*, any sorting algorithm compares keys pairwise. We can view its flow as a binary decision tree whose leaves correspond to the $n!$ possible orderings (permutations) of the n inputs.

- A binary tree with L leaves has height at least $\lceil \log_2 L \rceil$.
- Here $L = n!$, so any comparison sort must make

$$\Omega(\log_2(n!)) = \Omega\big(n \log n\big)$$

  comparisons in the worst case (e.g. by Stirling's approximation or the simpler bound $n!/2 \geq (n/2)^{n/2}$).

  Thus $\Theta(n \log n)$ is optimal for any comparison-based sort.

3.2 Direct-Access (Counting) Sort

When keys are integers in a small universe $\{0, 1, \ldots, u-1\}$, and we allow *direct access* to an array of length u, we can sort in $O(n+u)$ time:

1. Allocate an array $\texttt{C}[0..\, u-1]$ of empty lists.
2. **for** $i \leftarrow 1$ to n: append item $A[i]$ onto list $\texttt{C}[A[i]]$.
3. Scan $k = 0$ to $u-1$, and for each element of $\texttt{C}[k]$ (in insertion order), output it.

Work and space: $O(n+u)$. If $u = O(n)$, this is $O(n)$ time.

3.3 Radix Sort

To handle larger integer ranges u, write each key K in base n as

$$K \;=\; d_0 + d_1\, n + d_2\, n^2 + \cdots + d_{m-1}\, n^{m-1}, \quad m = \lceil \log_n u \rceil,$$

with digits $0 \leq d_j < n$. Then sort by *tuples* of digits, from least to most significant, using a *stable* $O(n)$–time sort (e.g. counting sort) on each digit:

1. Decompose each K into its m base-n digits.
2. **for** $j = 0$ to $m-1$: stable-sort the array by digit d_j (counting sort on $0..\, n-1$).

Total time

$$O\Big(m\,(n+n)\Big) \;=\; O\Big(n\, \log_n u\Big).$$

In particular, if $u = n^c$ for constant c, then $\log_n u = c$ and radix sort runs in $O(n)$ time.

Next: practical stable-sorting implementations and applications of radix methods.

3. Problem Session 3.2: Hash-Backed Sequences

Goal. Using a black-box hash table supporting

$$\begin{aligned}&\mathsf{build}(\{(k_i,x_i)\}) : O(n) \text{ expected}\\&\mathsf{find}(k) \to (k,x) : O(1) \text{ expected}\\&\mathsf{insert/delete}(k,x) : O(1) \text{ expected amortized}\end{aligned}$$

implement a *sequence* interface on items x:

$$\begin{aligned}&\mathsf{build}(\langle x_0,\dots,x_{n-1}\rangle) \;:\; O(n) \text{ exp.}\\&\mathsf{get\_at}(i)\,,\ \mathsf{set\_at}(i,x) \;:\; O(1) \text{ exp.}\\&\mathsf{insert\_at}(i,x),\ \mathsf{delete\_at}(i) \;:\; O(n) \text{ exp.}\\&\mathsf{insert\_first/last}(x),\ \mathsf{delete\_first/last}() \;:\; O(1) \text{ exp. amortized}\end{aligned}$$

3.2.1 Index-Mapping for Random Access

Store each sequence entry x as a hash-table entry with

$$\mathsf{key} = i, \quad \mathsf{value} = x.$$

- get_at(i): find(i).value.
- set_at(i, x): find(i).value $\leftarrow x$.

3.2.2 Rebuild for Arbitrary Insert/Delete

To support insert_at or delete_at at index i:

1. `A` $\leftarrow$iter() // extract items in order
2. Perform the array-style insert/delete on `A` in $O(n)$.
3. build(`A`) in $O(n)$ to reconstruct the hash table.

3.2.3 Fast Deque Ends via *First* Offset

Maintain an integer *first* so that the valid keys are $\mathit{first}, \dots, \mathit{first} + n - 1$. Keep also $\mathit{length} = n$.

- insert_last(x): insert($\mathit{first} + n$, (x)), $n \mapsto n + 1$.
- delete_last(): delete($\mathit{first} + n - 1$), $n \mapsto n - 1$.
- insert_first(x): $\mathit{first} \leftarrow \mathit{first} - 1$; insert($\mathit{first}, x$), $n \mapsto n + 1$.
- delete_first(): delete(first), $\mathit{first} \leftarrow \mathit{first} + 1$, $n \mapsto n - 1$.

This preserves the invariant that the sequence occupies keys $\mathit{first}, \dots, \mathit{first} + n - 1$, and all operations run in $O(1)$ expected amortized.

—

4. "Critter-Sort" (Problem 3)

Sort n items by various key types; points for faster vs. slower correct solutions.

1. (a) Integers in $[-n..n]$: map to $[0..2n]$ by $k \mapsto k + n$, then *radix-sort* in $O(n)$.
2. (b) Strings of up to $10 \log n$ letters:

- *Radix-sort* in base n: pack each string into one integer in $[0, n^{10}]$ (treat as base-n digit string), then radix-sort in $O(n)$.

3. (c) Integers in $[0..n^2]$: radix-sort in $O(n)$.
4. (d) Rationals $w/f \leq 1,\ 0 < w \leq f \leq n^2$:
   - Comparison sort via compare$(w_i/f_i,\ w_j/f_j) \iff w_i\, f_j\ ?\ w_j\, f_i$ in $O(n \log n)$.
   - or *radix-sort*: map each w/f to $\lfloor w\, n^4/f \rfloor \in [0..n^6]$, then radix-sort in $O(n)$.

—

5. "Gank-Frehry" (Problem 4)

Given deck D of n cards, each labeled by a letter $a \ldots z$, define

$$P(D, i, k) = \mathsf{sort}\big(D[i..\, i + k - 1]\big) \quad \text{(cyclic indices).}$$

(a) Build in $O(n)$ time a DS answering in $O(1)$ whether $P(D, i, k) = P(D, j, k)$.
Solution: Maintain a length-26 *frequency table* of counts of $a, b, \ldots, z$ for each window of size k. Slide the window in $O(1)$ per step (increment one count, decrement one count), record the 26-tuple; equality reduces to comparing 26 integers in $O(1)$.

(b) Find the most frequent hand among all $P(D, i, k)$. Build all n frequency-tuples, radix-sort them (range $[0..n]$ in each of 26 digits) in $O(n)$, then scan to pick the mode in $O(n)$.

6. Binary Trees (Part 1)

We now introduce *rooted binary trees* as a flexible, dynamic DWV (dynamic, writable, versatile) data structure supporting both *sequence* and *set* interfaces in $O(h)$ time per operation, where h is the tree height.

6.1 Definitions

A *binary tree* is a collection of nodes, each with:

- $\mathsf{node.left}$, $\mathsf{node.right}$: pointers to children (or $\perp$ if absent),
- $\mathsf{node.parent}$: pointer to parent (or $\perp$ at the root),
- $\mathsf{node.item}$: the payload (key, value, or sequence element).

Key notions:

$$\text{Subtree rooted at } x : \{\, x\} \cup \{\text{all descendants of } x\}.$$
$$(x) = \#\{\text{edges from root to } x\},$$
$$(x) = \max\{(y) - (x) \;\big|\; y \in \text{subtree}(x)\},$$
$$\text{Tree height } h = (\text{root}).$$

6.2 In-Order (Traversal) Sequence

Define the "in-order" (or *traversal*) enumeration of nodes:

$$\mathsf{inorder}(x) = \Big\{ \mathsf{inorder}(x.\mathsf{left}),\ x,\ \mathsf{inorder}(x.\mathsf{right})$$

recursively. This yields a total order on every subtree.

6.3 Basic Tree-Walk Primitives

All run in $O(h)$ worst-case time:

[1] subtree_firstnode node.left$\neq \perp$ node $\leftarrow$ node.left node

[1] successornode node.right$\neq \perp$ subtree_first(node.right) node is not a left child of its parent node $\leftarrow$ node.parent node.parent

6.4 Insertion in In-Order Position

Insert 'new' immediately after 'node' in traversal order:

[1] insert_afternode, new node.right $= \perp$ attach new as node.right let $s \leftarrow$subtree_first(node.right) attach new as s.left

6.5 Deletion of an Arbitrary Node

Delete 'node' from tree, preserving in-order:

[1] deletenode node is a leaf detach from parent node.left$\neq \perp$ let $p \leftarrow$predecessor(node) swap $(p.\text{item}, \text{node.item})$ delete(p) let $s \leftarrow$successor(node) swap $(s.\text{item}, \text{node.item})$ delete(s)

Here predecessor is defined symmetrically to successor.

6.6 From Traversal to *Set* and *Sequence* Interfaces

- **Sequences:** store element x_i at the ith node in in-order; insert_at$(i, x) \to$ insert after the $i-1$st node, etc.
- **Sets (BSTs):** store key k in node so that the in-order is $\Uparrow k$; find(k) is just BST lookup in $O(h)$, as is find_prev, find_next via predecessor/successor.

Next lecture: *balance* the tree to ensure $h = O(\log n)$ and thus $O(\log n)$ per operation.

7. AVL Trees: Height-Balanced BSTs

We now enhance our basic binary tree to guarantee $h = O(\log n)$ by enforcing the *AVL (Adelson-Velskii–Landis) balance* condition: for every node x,

$$\left|(x.\mathsf{right}) - (x.\mathsf{left})\right| \leq 1.$$

This ensures any AVL tree with n nodes has height $h = O(\log n)$.

7.1 Subtree-Augmented Height

Augment each node with its subtree height:

$$x.\mathsf{height} \;=\; 1 \;+\; \max\{\, x.\mathsf{left.height},\, x.\mathsf{right.height}\}.$$

Since "height" is a *subtree property* (computed from children in $O(1)$ time), we maintain it in $O(1)$ per node. Whenever an insertion/deletion (which only adds/removes a leaf) alters the tree, we walk up the ancestor chain—up to h nodes—recomputing height in $O(h)$ total.

Define the *skew* of node x as

$$\text{skew}(x) \;=\; (x.\mathsf{right}) \;-\; (x.\mathsf{left}),$$

so height-balance means $\text{skew}(x) \in \{-1, 0, 1\}$ for every node.

7.2 Tree Rotations

Two local tree rewrites, *right-* and *left-rotations*, preserve in-order traversal:

A left-rotation is symmetric. Each rotation takes $O(1)$ pointer updates; afterward, update the two affected nodes' height.

7.3 Rebalancing After Insert/Delete

Upon inserting or deleting a leaf, walk up from the changed node to the root, updating height and checking skew. If $\text{skew}(x) = \pm 2$, let y be the "heavy" child of x:

$$y = \begin{cases} x.\mathsf{right}, & \text{if skew}(x) = +2, \\ x.\mathsf{left}, & \text{if skew}(x) = -2. \end{cases}$$

Compute skew(y), then apply one of four cases:

bskew(x)=+2 and skew(y) $\in \{0, +1\}$: *single right-rotate* at x.
skew(x) $= -2$ and skew(y) $\in \{0, -1\}$: *single left-rotate* at x.

skew(x) $= +2$ and skew(y) $= -1$: *double rotation*—first left-rotate at y, then right-rotate at x.

skew(x) $= -2$ and skew(y) $= +1$: *double rotation*—first right-rotate at y, then left-rotate at x.
After each rotation, update height on the rotated nodes, then continue upward. Each rotation and height-update costs $O(1)$, and we perform at most one rotation per unbalanced node, yielding $O(h)$ time for rebalancing.

7.4 AVL Tree Performance

- An AVL tree with n nodes satisfies $h = O(\log n)$.
- All basic BST operations—find, find_prev/next, insert, delete, get_at/insert_at/delete_at, etc.—now run in $O(h) = O(\log n)$ worst-case time.
- Build (from scratch) in $O(n)$ via repeated leaf-insert plus rebalancing, or $O(n \log n)$ via BST insert; iteration in $O(n)$.

Thus AVL trees give us a single, unified data structure supporting both the *set* and *sequence* interfaces in $O(\log n)$ time per update or query.

Problem Session 4: Binary Trees & Heaps

Data Structures Reviewed

- **Set Interface**

Array (unsorted)	$O(1)$ build, $O(n)$ find/insert/delete
Sorted array/set AVL	$O(n \log n)$ build, $O(\log n)$ query, $O(n)$ insert/delete
Hash table	$O(n)$ build, $O(1)$ expected ops, no order queries
Balanced BST (AVL)	$O(n \log n)$ build, $O(\log n)$ all ops including order queries

- **Sequence Interface**

  – Array / dynamic array: $O(1)$ random access, $O(n)$ insert/delete at ends or middle

- Sequence AVL tree: $O(\log n)$ all sequence ops (indexed access, insert/delete at index)
- **Priority Queue (Min/Max)**
  - Binary heap: $O(n)$ build, $O(\log n)$ insert/delete_max or delete_min

Measuring Empirical Performance

- Implemented all recitation data structures in Python
- Measured build, access, insert, delete times on same hardware
- Observations:
  - Array build/access: very fast (C-intrinsic), constant time
  - Dynamic array insert/delete at end: amortized $O(1)$
  - Linked-list sequence delete/insert at head: $O(1)$
  - AVL sequence insert/delete at arbitrary index: $O(\log n)$
  - Sorted array set: fast lookup, slow updates
  - AVL set: balanced performance across all ops

Sequence AVL Tree Deletion Example

1. Given a sequence AVL tree with stored *height* and *subtree size* at each node
2. Perform `delete_at(8)`:
   - Navigate by comparing target index to left-subtree size
   - Remove the found node, splice children as usual
   - On the path back to the root, update:

   $$\text{size} \leftarrow 1 + \text{size(left)} + \text{size(right)}, \quad \text{height} \leftarrow 1 + \max(\text{height(left)}, \text{height(right)})$$

   - If any node becomes unbalanced ($|\Delta| > 1$), apply rotations:
     - *Right rotation*: for left-heavy case
     - *Left–right* / *right–left* double rotations: for zig-zag cases
3. Each delete costs $O(\log n)$ for navigation + $O(1)$ per rotation, at most $O(\log n)$ rotations

Problem 4.1: "Nick Fury" Extremes

Goal: From an array of n opinions (positive/negative), find the $\log n$ most extreme values.

- *Model API:*

$$\textsc{build}(\text{array}) = O(n), \quad \textsc{delete\_max}() = O(\log n)$$

- **Reduction to priority queue:**
  1. Build heap on absolute values in $O(n)$
  2. Repeat $\log n$ times: DELETE_MAX to extract extremes
  3. Total: $O(n + (\log n) \cdot \log n) = O(n)$
- **Space-restricted variant:** only $O(\log n)$ extra space
  - Maintain a set AVL tree of size $\log n$ holding current top extremes
  - For each new opinion:
    - Insert into tree, remove smallest if size exceeds $\log n$
  - Each op $O(\log \log n)$ on size–$\log n$ tree; n such ops $\Rightarrow O(n \log \log n)$

Problem 4.3: Top-k Bidders

Operations:

- `new_bid(id, $)` / `update_bid(id, $)` in $O(\log n)$
- `get_revenue()`: sum of top k bids in $O(1)$

Data structure design:

- **DictAVL** keyed by bidder ID $\to$ node pointers
- **HighAVL** (size k) on bid amounts (max-heap semantics)
- **LowAVL** (size $n-k$) on bid amounts
- **TotalRevenue**: integer sum of bids in HighAVL

Maintaining invariants:

1. On `new_bid/update_bid`:
   - Via DictAVL, locate and remove old entry from one of HighAVL/LowAVL
   - Reinsert updated bid into appropriate tree:

$$\begin{cases} \text{if } \$ \geq \min(\text{HighAVL}) & \to \text{HighAVL}, \text{ update TotalRevenue} \\ \text{else} & \to \text{LowAVL} \end{cases}$$

   - If HighAVL size $> k$, move its minimum to LowAVL (adjust TotalRevenue)
   - If HighAVL size $< k$, move maximum from LowAVL to HighAVL
2. All operations use $O(1)$ cross-links + $O(\log n)$ AVL ops $\Rightarrow O(\log n)$
3. `get_revenue()`: return `TotalRevenue` in $O(1)$

Problem 4.4: Receiver Roster (Rank Queries)

Operations:

- `record_game(playerID, gameID, points)` / corrections in $O(\log n)$
- `find_kth_best(k)`: return player with k-th highest average in $O(\log n)$

Data structure design:

- **PlayerAVL** keyed by playerID $\to$ node pointers
  - Each node stores nested **GameAVL** (per-player game records)
  - Augment with *sumPoints* and *gameCount*
- **RankAVL** keyed by average performance ($\sum / c$) with subtree-size augment

Maintaining invariants:

- On `record_game`/update:
  1. Use PlayerAVL $\to$ locate player node
  2. Update its GameAVL (insert/delete), adjust *sumPoints*, *gameCount*
  3. Remove & reinsert player in RankAVL (key changes), update subtree sizes
- `find_kth_best(k)`: in RankAVL, perform SELECT via subtree sizes in $O(\log n)$

8. Binary Heaps & Heapsort

Today we introduce the *binary heap*, an array-based tree structure implementing the **priority queue** interface in $O(\log n)$ time per operation and yielding an in-place $n \log n$ sort.

8.1 Priority Queue Interface

Store a set of items with keys (priorities) and support:

$\mathsf{insert}(x)$:	add item x with key $x.\mathsf{key}$.
$\mathsf{find\_max}()$:	return item with largest key, or $\perp$.
$\mathsf{delete\_max}()$:	remove and return the max-key item, or $\perp$.
$\mathsf{build}(\{x_i\}_{i=1}^{n})$:	construct PQ on n items.

8.2 Heapsort by PQ

Any PQ with build in $T_{\text{build}}(n)$ and delete_max in $T_{\max}(n)$ induces:

$$T_{\text{sort}}(n) \;=\; T_{\text{build}}(n) \;+\; \sum_{i=1}^{n} T_{\max}(i) \quad \text{or} \quad n \cdot T_{\text{insert}}(n) \;+\; T_{\max}(n)\,.$$

In particular, a heap with $T_{\text{build}} = O(n)$ and $T_{\max} = O(\log n)$ yields $O(n \log n)$ in-place.

8.3 Complete Binary Trees in an Array

An n-node *complete binary tree* is filled level by level, left to right. Its nodes are stored in an array $Q[0..n-1]$ in *level order*. For index i:

$$\text{left child: } 2i+1, \quad \text{right child: } 2i+2, \quad \text{parent: } \lfloor (i-1)/2 \rfloor.$$

This array uses no pointers—an *implicit-tree* representation.

8.4 Max-Heap Property

Q is a *max-heap* iff for every i:

$$Q[i].\mathsf{key} \;\geq\; \begin{cases} Q[2i+1].\mathsf{key}, \\ Q[2i+2].\mathsf{key}, \end{cases}$$

whenever those children exist. By induction, each node's key is $\geq$ all keys in its subtree. Hence the maximum lies at $Q[0]$.

8.5 Insertion: heapify_up

[1] InsertQ, x append x at $Q[\text{size}-1]$ $\textsc{heapify\_up}(Q,\ i = \text{size}-1)$ heapify_upQ, i $i = 0$ at root $p \leftarrow \lfloor (i-1)/2 \rfloor$ $Q[p].\mathsf{key} < Q[i].\mathsf{key}$ swap $Q[p] \leftrightarrow Q[i]$ $\textsc{heapify\_up}(Q, p)$ Runs in $O() = O(\log n)$.

8.6 Delete-Max: heapify_down

[1] Delete_Max$Q$ size $= 0$ $\perp$ret $\leftarrow Q[0]$ swap $Q[0] \leftrightarrow Q[\text{size}-1]$ size $-= 1$ $\textsc{heapify\_down}(Q, 0)$ ret heapify_downQ, i let $\ell = 2i+1$, $r = 2i+2$children $\ell \geq$ size leaf $j \leftarrow \ell$ $r <$ size and $Q[r].\mathsf{key} > Q[\ell].\mathsf{key}$ $j \leftarrow r$ $Q[j].\mathsf{key} > Q[i].\mathsf{key}$ swap $Q[i] \leftrightarrow Q[j]$ $\textsc{heapify\_down}(Q, j)$ Also $O(\log n)$ worst-case.

8.7 Heapsort and In-Place Build

Heapsort on array $A[0..n-1]$:

1. *In-place build_heap*(A) in $O(n)$ by calling HEAPIFY_DOWN A, i for $i = \lfloor n/2 \rfloor - 1, \ldots, 0$.
2. For $k = n-1$ down to 1: swap $A[0] \leftrightarrow A[k]$, then HEAPIFY_DOWN$(A, 0)$ on prefix of size k.

Total time $O(n) + \sum_{k=1}^{n} O(\log k) = O(n \log n)$, uses only $O(1)$ extra space.

Summary: Binary heaps give an in-place priority queue with $O(\log n)$ insert/delete$_{\max}$, $O(n)$ build, and yield an in-place $O(n \log n)$ heapsort.

9. Graphs & Breadth-First Search

Today we begin PartII: graph theory. We introduce the fundamental problem of *single-source shortest paths* in an unweighted graph and give the classic $O(|V| + |E|)$ **Breadth-First Search** (BFS) algorithm.

9.1 Definitions & Notation

A (simple) graph $G = (V, E)$ has

$$V = \{v_1, \ldots, v_n\}, \quad E \subseteq \{\{u, v\} : u, v \in V,\ u \neq v\} \quad \text{(undirected)}.$$

$\mathrm{Adj}(u) = \{v : \{u, v\} \in E\}$ lists u's neighbors. A *path* from s to t is a sequence $(v_0 = s, v_1, \ldots, v_k = t)$ with $\{v_{i-1}, v_i\} \in E$; its *length* is k.

We seek, for a fixed source s, the distance $\mathrm{dist}(s, v) = \min\{\text{length of any path } s \to v\}$ and a corresponding *predecessor* tree $p[v]$ so that following p-pointers from v back to s yields a shortest path.

9.2 Level Sets

Define level-sets

$$L_0 = \{s\}, \quad L_i = \{\, v \in V : \mathrm{dist}(s, v) = i\}.$$

Then $L_0, L_1, \ldots$ partition those vertices reachable from s by increasing distance.

9.3 Breadth-First Search (BFS)

Compute dist and p by "growing" level-sets one layer at a time. [1] BFS$G = (V, E)$, s for each $v \in V$: set $\mathrm{dist}[v] \leftarrow \infty$, $p[v] \leftarrow \mathsf{nil}$ $\mathrm{dist}[s] \leftarrow 0$ initialize empty queue Q; *enqueue*Q, s Q not empty $u \leftarrow$ *dequeue*Q each $v \in \mathrm{Adj}(u)$ $\mathrm{dist}[v] = \infty$ v first discovered $\mathrm{dist}[v] \leftarrow \mathrm{dist}[u] + 1$ $p[v] \leftarrow u$ *enqueue*Q, v
After BFS:

$$\mathrm{dist}[v] = \mathrm{dist}(s, v), \quad p[v] = \text{predecessor of } v \text{ on some shortest } s \to v \text{ path}.$$

9.4 Correctness & Runtime

- BFS visits vertices in nondecreasing order of dist, so each is first discovered at the correct distance.
- Each edge $\{u, v\}$ is examined exactly twice (once from u, once from v), and each vertex enqueued/dequeued once.

Thus total time is

$$O\big(|V| + |E|\big),$$

and space is $O(|V|)$ for dist, p, and the queue.

Next: weighted graphs & Dijkstra's algorithm.

Review: Graphs

- A *graph* $G = (V, E)$ has vertex set V and edge set $E \subseteq V \times V$.
- *Directed* vs. *undirected* graphs (arrows vs. no arrows).
- A *simple graph*: no duplicate edges or self-loops.
- For $v \in V$, define outgoing neighbors

$$^{+}(v) \;=\; \{\, w : (v, w) \in E\}.$$

- A *path* is a sequence of vertices following edges. A *simple path* visits no vertex twice.
- Common problems: reachability, shortest path, connectivity, etc.

Notation: Linear-time in Graphs

"Saying an algorithm runs in linear time on a graph" means $O(|V| + |E|)$.

Breadth-First vs. Depth-First Search

BFS: explores in "waves" by distance from source, level by level.
DFS: explores by following one branch as far as possible, then backtracking.

Reachability & Parent-Tree

- *Reachability problem:* Given directed G and source s, determine all v reachable from s.
- Store an array $parent[v]$; when $parent[v] \neq \perp$, v has been reached.
- Recover path by backtracking parents from v to s.

Depth-First Search (DFS)

```
procedure DFS(G, s):
  parent[s] ← s
  call Visit(s)

procedure Visit(u):
```

```
for each v in Adj(u) do
  if parent[v] =  then
    parent[v] ← u
    Visit(v)
```

Correctness (by induction on distance k)

- *Claim:* For any v at distance k from s, DFS sets $parent[v]$ correctly.
- *Base* $k = 0$: Only s, and $parent[s]$ is initialized.
- *Inductive step:* Let v be at distance $k+1$, and let u be its predecessor at distance k. By IH, Visit(u) is called, sees $v \in (u)$, and sets $parent[v] := u$.

Runtime

$$\text{DFS}(s): \; O\big(|E_{\text{reachable}}|\big) \quad \Longrightarrow \quad O(|E|)$$

Full DFS over all components is $O(|V| + |E|)$.

Applications

(1) Connected Components (undirected)

- *Full-DFS:* for each $v \in V$, if unvisited, call DFS(v) to mark its entire component.
- Overall time: $O(|V| + |E|)$.

(2) Topological Order in DAGs

- A DAG is a directed acyclic graph.
- A *topological order* is a numbering $f : V \to \{1, \ldots, |V|\}$ so that $(u, v) \in E \implies f(u) < f(v)$.
- Compute *full DFS*, record vertices in the order they finish (post-order), then reverse that list.
- *Theorem:* In a DAG, reverse finishing order is a valid topological order.
  - *Proof sketch:* For each edge (u, v), either u calls v (so v finishes before u) or v cannot reach u (so finishes first); reversing restores u before v.

(3) Cycle Detection in Directed Graphs

- A graph has a directed cycle iff it is not a DAG.
- Run full DFS; check for any back-edge from u to an ancestor v in the recursion tree.
- Upon discovering such an edge, report the cycle by backtracking the parent pointers from u back to v.
- Runs in $O(|V| + |E|)$.

Review: Unweighted Single-Source Shortest Paths

- **BFS** finds, for unweighted $G = (V, E)$ and source s,

$$\delta(s, v) = \min\{\#\text{edges on any path } s \to v\}$$

  and parent pointers in $O(|V| + |E|)$.

- *Reachability:* list only reachable vertices in $O(|E|)$.
- *Connected components:* full-graph BFS/DFS loop $\Rightarrow O(|V| + |E|)$.
- *Topological sort* in a DAG via reverse DFS-finish order $\Rightarrow O(|V| + |E|)$.

Weighted Graphs

- A *weighted graph* is (V, E, w), $w : E \to \mathbb{Z}$ (edge weights may be negative, zero, or positive).
- *Path weight:* for path $\pi = e_1, \ldots, e_k$,

$$w(\pi) = \sum_{i=1}^{k} w(e_i).$$

- *Weighted shortest-path distance:*

$$\delta(s,t) = \begin{cases} \inf\limits_{\pi : s \to t} w(\pi) & \text{if a path exists and no negative cycle reachable,} \\ -\infty & \text{if a negative-weight cycle is reachable,} \\ +\infty & \text{if } t \text{ is unreachable.} \end{cases}$$

- Negative-weight cycle: any cycle whose total weight < 0 allows arbitrarily low $w(\pi)$.

Parent Pointers from Distances

Given all finite $\delta(s, v)$, we can build `parent`[] in $O(|V| + |E|)$:

```
for v in V:
  parent[v] ←
parent[s] ←
for each u in V:
  for each (u,v) in E:
    if parent[v]= and (s,u)+w(u,v)=(s,v):
      parent[v] ← u
```

Relaxation & Triangle Inequality

- Maintain estimates $d(v) \geq \delta(s, v)$.
- *Relax edge* (u, v): if $d(u) + w(u, v) < d(v)$ then set

$$d(v) \leftarrow d(u) + w(u, v).$$

- Invariant: after each relax, $d(v)$ equals weight of some $s \to v$ path or $+\infty$.
- Relaxation is *safe*: never underestimates true $\delta(s, v)$.

Single-Source Shortest Paths in a DAG

1. **Initialize:**

$$\forall v \in V : \quad d(v) \leftarrow +\infty, \quad d(s) \leftarrow 0.$$

2. Compute a topological order $v_1, \ldots, v_{|V|}$ of the DAG.

3. **Relaxation loop:**

```
for u in [v1, v2, ..., v|V|]:
  for each edge (u,v) in E:
    relax(u,v)
```

Correctness: by induction on the topo-order, every predecessor's $d(u) = \delta(s, u)$ when processed, so relaxing yields $d(v) = \delta(s, v)$. *Time:* $O(|V| + |E|)$ since each edge is relaxed once.

Recap: Weighted SSSP in DAGs

- Last lecture: *DAG-relaxation* solves single-source shortest-paths (SSSP) in any DAG (even with negative weights) in $O(|V| + |E|)$.
- We can reconstruct parent pointers in $O(|V| + |E|)$ once all finite distances are known.
- Today's goal: SSSP in *general* directed graphs (possibly with cycles and negative weights), returning
  - finite $\delta(s, v)$ for vertices with well-defined shortest distance,
  - $\delta(s, v) = +\infty$ if v unreachable,
  - $\delta(s, v) = -\infty$ if v is "pulled down" by a reachable negative-weight cycle,
  - plus an explicit negative cycle if one exists.

Negative-Weight Cycles

- A *negative-weight cycle* is a directed cycle whose total edge-weight < 0.
- If any cycle of negative total weight is reachable from s, then any vertex reachable from that cycle has $\delta(s, v) = -\infty$ (you can loop arbitrarily many times).
- To detect such vertices, we will identify a *witness* u:

$$\delta_k(s, u) < \delta_{k-1}(s, u)$$

  where $\delta_k(s, u)$ is the minimum weight of any $s-u$ path using at most k edges.
- Any negative-weight cycle contains at least one witness; and any vertex with $\delta(s, v) = -\infty$ is reachable from a witness.

k-Edge Shortest Paths

$$\delta_k(s, v) \;=\; \min\{\, w(\pi) \mid \pi : s \to v,\ \left|\pi\right| \le k\}.$$

- We only need to check $k \le |V| - 1$ since any simple path has $\le |V| - 1$ edges.
- If for some v, $\delta_{|V|}(s, v) < \delta_{|V|-1}(s, v)$, then v is a witness and $\delta(s, v) = -\infty$.

Graph-Duplication Trick

- Construct a new DAG $G' = (V', E')$ with $|V| + 1$ *levels*: for each $v \in V$ and $0 \le k \le |V|$ create $v^{(k)}$.
- Add zero-weight "stay" edges: $(v^{(k)}, v^{(k+1)})$ for all $0 \le k < |V|$.

- For each original edge $(u, v) \in E$, add $(u^{(k)}, v^{(k+1)})$ with weight $w(u, v)$, for all $0 \le k < |V|$.
- Then $|V'| = (|V| + 1)|V|$ and $|E'| = |V| \cdot |V| + |V| \cdot |E| = O(|V|\,(|V| + |E|))$.
- In G', any path from $s^{(0)}$ to $v^{(k)}$ uses $\le k$ original edges, so

$$\delta_{G'}\big(s^{(0)},\, v^{(k)}\big) = \delta_k(s, v).$$

Bellman–Ford via DAG Relaxation

1. Build G' as above.
2. Run *DAG-relaxation* on G' from source $s^{(0)}$:

```
// initialize all d[u^{(k)}]=+ except d[s^{(0)}]=0
topoOrder ← topological sort of G'
for u in topoOrder:
  for each edge (u,v) in G'.E:
    relax(u,v)
```

3. For each original $v \in V$, let

$$d_v \;=\; d_{G'}\big(s^{(0)},\, v^{(|V|-1)}\big).$$

   - If $d_v < +\infty$, then $d_v = \delta(s, v)$.
   - If $d_v = +\infty$, v is unreachable.
   - If $\exists\, u$ with $d_{G'}(s^{(0)}, u^{(|V|)}) < d_{G'}(s^{(0)}, u^{(|V|-1)})$, mark u a *witness*.
   - From each witness, do a DFS/BFS in the *original* graph to mark all reachable v with $d_v \leftarrow -\infty$.

Correctness & Running Time

- *Correctness:* By construction $\delta_{G'}(s^{(0)}, v^{(k)}) = \delta_k(s, v)$; if no relaxation occurs at level $|V|$, no witness $\rightarrow$ no negative cycle.
- *Time:* Building G' and running DAG-relaxation takes $O(|V|\,(|V| + |E|))$. Finding witnesses and marking reachable vertices adds $O(|V| + |E|)$. $\Rightarrow O\big(|V| \cdot |E|\big)$ overall.

Recap: SSSP on Weighted Graphs

- Three prior approaches:
  1. *BFS-reduction*: expand each edge of weight w into w unit-edges—linear only if $\sum w = O(|V| + |E|)$ and $w \ge 0$.
  2. *DAG-relaxation*: if G is a DAG (no cycles), run relaxation in topological order in $O(|V| + |E|)$.
  3. *Bellman–Ford*: works on any directed graph (even with negative-weight cycles), in $O(|V| \times (|V| + |E|)) = O(|V| \cdot |E|)$, marking distances $-\infty$ for vertices pulled down by a reachable negative cycle.
- Today: assume all edge-weights $w(e) \ge 0$, no negative cycles, and achieve $O((|V| + |E|) \log |V|)$ (or better in practice).

Key Observations

1. **Nonnegativity $\implies$ monotonicity.** Along any shortest path $s \to \cdots \to u \to v$,

$$\delta(s,u) \;\leq\; \delta(s,v),$$

since edge-weights ≥ 0. Thus "closer" vertices cannot become "farther" later.

2. **If vertex-distances were known in sorted order,** we could relax edges in that order (like DAG-relaxation) and compute all $\delta(s,v)$ in $O(|V|+|E|)$.

Dijkstra's Algorithm

[1] **Initialize:** $\forall v \in V : d[v] \leftarrow +\infty, \quad d[s] \leftarrow 0$ Build a *changeable priority queue* Q of all $v \in V$, keyed by $d[v]$ $Q \neq \varnothing$ $(u, d_u) \leftarrow Q.\text{delete\_min}()$ remove vertex of smallest current estimate each $(u,v) \in E$ $d[v] > d[u] + w(u,v)$ $d[v] \leftarrow d[u] + w(u,v)$ $Q.\text{decrease\_key}(v, d[v])$ $d[\cdot]$

Changeable Priority Queue

Supports three operations on items with unique `id` and numeric `key`:

$$\text{build(all items)}, \quad \text{delete\_min}(), \quad \text{decrease\_key}(\texttt{id}, \texttt{newKey})$$

Implementation strategies:

- **Array + linear scan**–$O(1)$ `decrease_key`, $O(|V|)$ `delete_min` $\implies O(|V|^2 + |E|\,|V|)$ overall (good if $|E| = \Theta(|V|^2)$).
- **Binary heap + direct-access array**–$O(\log|V|)$ both operations $\implies O((|V|+|E|)\log|V|)$.
- **Fibonacci heap**–$O(1)$ amortized `decrease_key`, $O(\log|V|)$ `delete_min` $\implies O(|V|\log|V| + |E|)$.

Correctness Sketch

- *Invariant:* once $d[v] = \delta(s,v)$, it never increases (relaxation only lowers estimates, and always to the length of some path).
- *Claim:* when a vertex v is extracted by `delete_min`, $d[v] = \delta(s,v)$.
  - Base (s): $d[s] := 0 = \delta(s,s)$.
  - Induction: let v be the kth extracted. Any shortest $s \to v$ path passes through some predecessor u either already extracted (so $d[u] = \delta(s,u)$ by IH, then edge-relaxation fixed $d[v]$ when u was extracted) or still in Q (so $d[v] \geq d[u] + w(u,v) \geq \delta(s,u) + w(u,v) = \delta(s,v)$). Since v had the minimal $d[\cdot]$ among Q, we must have $d[v] = \delta(s,v)$ at extraction.

Running Time

$$T \;=\; O\big(\underbrace{|V|\log|V|}_{\text{build} + |V| \times \text{delete\_min}} \;+\; \underbrace{|E| \times 1}_{\substack{\text{decrease\_key per relax} \\ \text{(amortized)}}}\big) \;=\; O(|V|\log|V| \;+\; |E|).$$

- In sparse graphs $(|E| = O(|V|))$: $O(|V|\log|V|)$.
- In dense graphs $(|E| \approx |V|^2)$: array-scan gives $O(|V|^2)$, which is still optimal up to constants.
- Fibonacci heaps achieve $O(|V|\log|V| + |E|)$ in all regimes.

Problem Statement

- **Input:** A directed graph $G = (V, E)$ with integer edge-weights $w : E \to \mathbb{Z}$, no negative-weight cycles.
- **Output:** For every ordered pair $(u, v) \in V \times V$, compute

$$\delta(u,v) \;=\; \min_{\pi: u \to v} \sum_{(x,y)\in\pi} w(x,y) \;\in\; \mathbb{Z} \cup \{+\infty\}.$$

- If G has a reachable negative-weight cycle, we may *abort*.
- Note: output size is $\Theta(|V|^2)$, so $\Omega(|V|^2)$ time is unavoidable.

Brute-Force via SSSP

$$\text{For each } s \in V: \text{ run SSSP}(G, s) \implies O\big(|V| \times T_{\text{SSSP}}\big).$$

- Bellman–Ford $\Rightarrow O\big(|V| \cdot (|V| + |E|)\big) = O(|V| \cdot |E|)$.
- Dijkstra (if $w \geq 0$) $\Rightarrow O\big(|V|^2 \log |V| + |V||E|\big)$.
- … suboptimal when $|V|$ large.

Johnson's Reweighting Technique

"Edge-Potential" Transformation

Choose an arbitrary *potential* function $h : V \to \mathbb{Z}$. Define new weights

$$w'(u,v) = w(u,v) \;+\; h(u) \;-\; h(v).$$

Claim: All shortest paths in G remain shortest in $G' = (V, E, w')$. [Sketch] Any directed path $\pi = v_0 \to v_1 \to \cdots \to v_k$ in G has

$$w'(\pi) = \sum_{i=1}^{k} \big[w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)\big] = w(\pi) \;+\; h(v_0) \;-\; h(v_k).$$

The term $h(v_0) - h(v_k)$ is constant for all paths from v_0 to v_k, so minima are preserved.

Finding a ***nonnegative*** Reweighting

1. Add a *super-source* s with 0-weight edges (s, v) for all $v \in V$, obtaining G_s.
2. Run Bellman–Ford from s to compute

$$h(v) = \delta_{G_s}(s, v) \quad (\text{finite} \Leftrightarrow \text{no neg. cycle reachable}).$$

   - If any $h(v) = -\infty$, **abort** (neg. cycle exists).
3. Reweight each $(u, v) \in E$ by

$$w'(u,v) = w(u,v) + h(u) - h(v) \;\geq 0,$$

   since Bellman–Ford distances obey the triangle inequality.

All–Pairs via Dijkstra

1. Build G' with nonnegative weights w' as above.
2. *For each* $s \in V$, run Dijkstra on (G', w') in $O\big((|V| + |E|) \log |V|\big)$ to get $d'(s, v) = \delta_{G'}(s, v)$.
3. Recover original distances:

$$\delta_G(s, v) \;=\; d'(s, v) \;+\; h(v) \;-\; h(s).$$

Running Time

$$T_{\text{Johnson}} = \underbrace{O(|V| + |E|)}_{\text{add super-source}} \;+\; \underbrace{O(|V| \cdot |E|)}_{\text{Bellman–Ford}} \;+\; \underbrace{O\big(|V|\,(|V| + |E|) \log |V|\big)}_{\text{V-times Dijkstra}} = O\big(|V|\,|E| \;+\; |V|^2 \log |V|\big).$$

- *Sparse graphs* ($|E| = O(|V|)$): $O(|V|^2 \log |V|)$.
- *Dense graphs* ($|E| = \Theta(|V|^2)$): $O(|V|^3)$ via array-scan Dijkstra.
- Avoids $O(|V| \cdot |E|)$ per source by limiting Bellman–Ford to once.

1. Recursive-Algorithm Design: SRTBOT

1. **S**ubproblems: Identify a (polynomial) set of subproblems.
2. **R**elation: Write each subproblem's solution in terms of smaller subproblems.
3. **T**opological order: Find an order (or prove acyclicity) so dependencies go "forward."
4. **B**ase cases: Specify trivial subproblems explicitly.
5. **O**riginal problem: Express the desired output as one of the subproblems.
6. **T**ime: Analyze total cost by summing nonrecursive work over all subproblems.

2. Add Memoization = ***Dynamic Programming***

$$\text{recursive\_solve}(\mathit{prob}) : \begin{cases} \text{if } \mathit{prob} \in \text{memo} : \;\; \mathit{return}\ \text{memo}[\mathit{prob}] \\ \text{if base-case} : \;\; \mathit{ans} \leftarrow \text{trivial-value} \\ \quad \text{else:} \;\; \mathit{ans} \leftarrow \text{Relation}(\{\text{recursive\_solve}(\mathit{sub})\}) \\ \text{memo}[\mathit{prob}] \leftarrow \mathit{ans} \\ \mathit{return}\ \mathit{ans} \end{cases}$$

$$T \;=\; \sum_{\substack{\text{each subproblem} \\ p}} \big[\text{cost of computing } p \text{ (excl. recursive calls)}\big].$$

3. Examples

3.1 Fibonacci Numbers

$$F_n = \begin{cases} 1 & n = 1, 2, \\ F_{n-1} + F_{n-2} & n > 2. \end{cases}$$

- **Subproblems:** $f(i) = F_i$, $i = 1, \ldots, n$.

- **Relation:** $f(i) = f(i-1) + f(i-2)$.
- **Order:** $i = 1, 2, \ldots, n$.
- **Base:** $f(1) = f(2) = 1$.
- **Original:** $f(n)$.
- **Time:** $\Theta(n)$ additions.

3.2 DAG Single-Source Shortest Paths

Let $G = (V, E)$ be a DAG, source s. Define

$$d(v) = \min\big\{\, d(u) + w(u, v) \;:\; (u, v) \in E\big\} \cup \{\, +\infty\}\big\}, \quad d(s) = 0.$$

- **Subproblems:** $d(v)$ for each $v \in V$.
- **Relation:** $d(v) = \min_{(u,v)\in E}\{d(u) + w(u, v)\}$.
- **Order:** Any topological sort of G.
- **Base:** $d(s) = 0$, all others initialized $+\infty$.
- **Original:** All $d(v)$.
- **Time:** $\sum_v O(\deg^-(v)) = O(|V| + |E|)$.

4. Case Study: "Linear" Bowling

- n pins in a row, pin i has score $v_i \in \mathbb{Z}$.
- Hitting one pin i yields $+v_i$; hitting two adjacent pins $(i, i+1)$ yields $+v_i\, v_{i+1}$.
- You may skip pins; maximize total score.

DP Formulation

- **Subproblems:** $B(i) =$ optimum score on suffix $i, \ldots, n-1$.
- **Relation:**
$$B(i) = \max\Big\{B(i+1),\ v_i + B(i+1),\ v_i\, v_{i+1} + B(i+2)\Big\}.$$
- **Order:** $i = n, n-1, \ldots, 0$.
- **Base:** $B(n) = 0$, and $B(n+1) = 0$.
- **Original:** $B(0)$.
- **Time:** $O(1)$ per i, $\Theta(n)$ total.

5. Key Takeaways

- *Dynamic programming* = recursion + memoization.
- *Design recipe (SRTBOT)* helps structure DP: choose subproblems, write recurrence, order, base, original, analyze.
- Many sequence problems use *prefixes*, *suffixes*, or *substrings* as subproblems.
- Local "brute force" + memoization turns exponential recurrences polynomial.

1. Review: SRTBOT & Memoization

- **S**ubproblems
  - Split your problem into a polynomial set of subproblems.
  - For *sequences*, try *prefixes*, *suffixes*, or *substrings*.
- **R**elation
  - Express each subproblem in terms of smaller subproblems.
  - *Brute-force* any unknown feature with a polynomial loop, then `max`/`min`/`sum`/etc.
- **T**opological order
  - Choose an order (usually simple for-loops) so all dependencies are computed first.
- **B**ase cases
- **O**riginal problem
  - Identify which subproblem(s) give the final answer.
- **T**ime

$$T = \sum_{\text{subproblem } p} \big[\text{nonrec. work to compute } p\big].$$

- + *Memoization*: cache each subproblem's answer to avoid recomputation.

2. Longest Common Subsequence (LCS)

Problem. Given two sequences $A[0..m-1]$, $B[0..n-1]$, find a longest sequence L that is a subsequence of both.

- **Subproblems:**

$$L(i,j) = \text{length of LCS of suffix } A[i..m-1] \text{ and } B[j..n-1].$$

- **Relation:**

$$L(i,j) = \begin{cases} 1 + L(i+1, j+1) & \text{if } A[i] = B[j], \\ \max\{\, L(i+1,j),\, L(i,j+1)\} & \text{otherwise.} \end{cases}$$

- **Order:** for $i = m \downarrow 0$, for $j = n \downarrow 0$.
- **Base:** $L(m,j) = L(i,n) = 0$ for all i, j.
- **Original:** $L(0,0)$.
- **Time:** $O(mn)$ (each of mn cells $O(1)$ work).
- **Recovering a solution:** store *parent pointers* to trace back one valid LCS.

3. Longest Increasing Subsequence (LIS)

Problem. Given one sequence $A[0..n-1]$, find a longest strictly increasing subsequence.

- **Trick:** we must know the *first element* of the LIS to enforce the increasing condition.

- **Subproblems:**

$$\text{LIS}(i) = \text{length of LIS of } A[i..n-1] \text{ } \textit{that} \text{ starts at } A[i].$$

- **Relation:**

$$\text{LIS}(i) = 1 + \max_{\substack{i<j<n \\ A[j]>A[i]}} \text{LIS}(j),$$

  defaulting to 1 if no j qualifies.
- **Original:** $\max_{0 \le i < n} \text{LIS}(i)$.
- **Order:** for $i = n-1 \downarrow 0$.
- **Base:** $\text{LIS}(n) = 0$ (empty suffix).
- **Time:** $O(n^2)$ (each i scans $O(n)$ choices).

4. Alternating-Move Coin-Taking Game

Problem. Coins $v_0, \ldots, v_{n-1}$ in a row. Two players alternate taking either the leftmost or rightmost coin. Score is sum of values you take. Zero-sum, both play optimally. Compute the first player's maximum guaranteed score.

- **Subproblems:** $X(i, j, P)$ = first player's max score on coins $i..j$ when it is player P's turn ($P \in \{\text{Me}, \text{You}\}$).
- **Relation:**

$$X(i,j,\text{Me}) \;=\; \max\bigl\{v_i + X(i+1,j,\text{You}),\; v_j + X(i,j-1,\text{You})\bigr\},$$

$$X(i,j,\text{You}) \;=\; \min\bigl\{X(i+1,j,\text{Me}),\; X(i,j-1,\text{Me})\bigr\}.$$

- **Order:** increasing substring length $j-i$.
- **Base:** $X(i,i,\text{Me}) = v_i, \quad X(i,i,\text{You}) = 0$.
- **Original:** $X(0, n-1, \text{Me})$.
- **Time:** $O(n^2)$ (each of $O(n^2)$ subproblems $O(1)$ work).

5. Subproblem-Expansion

- Sometimes the *obvious* subproblem (suffix/prefix) isn't enough to write a simple recurrence.
- *Expand* your subproblem definition by adding *constraints* (e.g. "starts at i," "which player's turn")—as long as the overall count remains polynomial.
- This yields simpler recurrences at the cost of a larger DP table.

1. Dynamic Programming *via* Subproblem Expansion

- We can "remember" additional *state* by *expanding* our subproblem index.
- E.g. in the two-player coin game we had both (i, j, Me) and (i, j, You).
- Today:
  - Bellman–Ford as DP

- Floyd–Warshall (all-pairs SSSP)
- Arithmetic parenthesization (max/min)
- Piano-/guitar- fingering

2. Bellman–Ford as DP

$$d_k(s,v) \;=\; \min\Big\{\, d_{k-1}(s,v),\ \min_{(u\to v)\in E}\big[d_{k-1}(s,u)+w(u,v)\big]\Big\},$$

- Subproblems: $d_k(s,v) =$ shortest $\le k$-edge path $s\to v$.
- Order: increasing $k=0,1,\dots,n-1$ (acyclic in k).
- Base: $d_0(s,s)=0$, $d_0(s,v)=+\infty$ for $v\neq s$.
- Original: $d_{n-1}(s,v)$ (or detect negative cycle if $d_n < d_{n-1}$).
- Time: $\displaystyle\sum_{k=1}^{n-1}\sum_{v\in V}\deg^-(v) \;=\; O(n\,|E|)$.

3. Floyd–Warshall: All-Pairs SSSP via DP

$$D^{(k)}_{u,v} = \min\Big\{D^{(k-1)}_{u,v}, D^{(k-1)}_{u,k} + D^{(k-1)}_{k,v}\Big\},$$

- Subproblems: $D^{(k)}_{u,v} =$ shortest $u\to v$ using only intermediate $\{1,\dots,k\}$.
- Order: $k=0,1,\dots,n$; for each k loop over all $u,v\in\{1..n\}$.
- Base ($k=0$): $D^{(0)}_{u,v} = \begin{cases}0, & u=v,\\ w(u,v), & (u\to v)\in E,\\ +\infty, & \text{else.}\end{cases}$
- Original: $D^{(n)}_{u,v}$ for all u,v.
- Time: $O(n^3)$ (three nested loops, $O(1)$ work each).

4. Parenthesization: Max/Min DP

$$X(i,j,\mathrm{opt}) = \mathrm{opt}\Big\{\,\big[X(i,k,\mathrm{opt}_L)\ \star_k\ X(k,j,\mathrm{opt}_R)\big]\ \Big|\ i<k<j,\ \mathrm{opt}_L,\mathrm{opt}_R\in\{\min,\max\}\Big\},$$

- **Subproblems:** $X(i,j,\mathrm{opt}) =$ value of best parenthesization of $a_i \star_{i+1} a_{i+1}\cdots\star_{j-1} a_{j-1}$ under $opt\in\{\min,\max\}$.
- Guess the *last* operator $\star_k$; recurse on two substrings.
- Order: increasing length $j-i$ (acyclic).
- Base: $X(i,i+1,\mathrm{opt}) = a_i$.
- Original: $X(0,n,\max)$.
- Time: $O(n^3)$ (two loops for i,j, inner loop over k, constant $\times 4$ for opt pairs).

5. Piano/Guitar Fingering (Sequence + State DP)

$$F(i,f) \;=\; \min_{f'\in[1..F]} \Bigl\{\, d\bigl(t_i,f;\,t_{i+1},f'\bigr) \;+\; F(i+1,f') \Bigr\},$$

- **Subproblems:** $F(i,f)$ = min. total difficulty to play suffix $t_i \dots t_{n-1}$ if note t_i uses finger f.
- Guess next-note finger f'; cost = transition $d(t_i,f;t_{i+1},f')$ + $F(i+1,f')$.
- Order: $i = n-1 \downarrow 0$, for each $f = 1..F$.
- Base: $F(n-1,f) = 0$ (last note, no further cost).
- Original: $\min_f F(0,f)$.
- Time: $O(n \cdot F^2)$.

General lesson: Expand subproblem indices to remember any finite *context/state*, then brute-force remaining choices.

1. Overview & SRTBOT Review

- Today: DP on integer inputs $\rightsquigarrow$ *pseudo-polynomial* time.
- Examples: **Rod-Cutting**, **Subset-Sum**.
- Then: "Diagonal" recap of all DP techniques seen.
- Reminder: SRTBOT = Subproblems, Relation, Topo-order, Base, Original, Time.
  - *Subproblem design* often hardest:
    - Sequences → prefixes/suffixes/substrings
    - Integers → all smaller e.g. Fibonacci: $F(n) \to F(0..n)$
    - Multiples → product of spaces
    - *Expansion*/state add'n → remember "past"
  - *Relation* by "guessing" answer to a question → recurse on smaller subproblems and *loop* over all guesses.
  - Ensure *acyclic*: give explicit topological order.
  - Add *base cases*; express *original* problem in terms of subproblems.
  - Run-time ≈#subproblems × non-recursive work + cost to combine original.

2. Rod-Cutting

Problem

Given integer length L and prices $v[1..L]$, choose a partition

$$L = i_1 + i_2 + \cdots + i_k, \quad i_j \in \{1,\dots,L\},$$

to maximize $\sum_{j=1}^{k} v[i_j]$.

SRTBOT

1. **Subproblems:**

$$R(\ell) \;=\; \max\Bigl\{\text{value of best cut of length } \ell\Bigr\}, \quad \ell = 0,1,\dots,L.$$

2. **Relation:** for each $1 \leq p \leq \ell$,

$$R(\ell) = \max_{1 \leq p \leq \ell} \big\{ v[p] \ + \ R(\ell - p) \big\}.$$

3. **Topo-order:** increasing $\ell = 0, 1, \ldots, L$ (calls only to smaller ℓ).
4. **Base:** $R(0) = 0$.
5. **Original:** $R(L)$.
6. **Time:** $\sum_{\ell=1}^{L} O(\ell) = O(L^2)$.

3. Subset-Sum (Decision DP)

Problem

Given integers $a_0, \ldots, a_{n-1} > 0$ and target T, decide if some subset sums to T.

SRTBOT

1. **Subproblems:**

$$S(i,t) \ = \ \begin{cases} \text{“Yes”} & \exists\, S \subseteq \{a_i, \ldots, a_{n-1}\} : \sum S = t, \\ \text{“No”} & \text{otherwise,} \end{cases}$$

for $0 \leq i \leq n$, $0 \leq t \leq T$.

2. **Relation:**

$$S(i,t) = \big[S(i+1,t)\big] \ \vee \ \big[t \geq a_i \ \wedge \ S(i+1,\, t - a_i)\big].$$

3. **Topo-order:** decreasing $i = n, n-1, \ldots, 0$ (suffix DP).
4. **Base:** for all $0 \leq t \leq T$,

$$S(n,t) = \begin{cases} \text{Yes}, & t = 0, \\ \text{No}, & t > 0. \end{cases}$$

5. **Original:** $S(0, T)$.
6. **Time:** $O(nT)$.

Pseudo-Polynomial Time

- Input size $=$ #words $= n + 1$ (the a_i plus T).
- Runtime $= O(nT)$ is **not** polynomial in $n + 1$ (unless $T = O(n^{O(1)})$).
- It is *pseudo-polynomial*: polynomial in n and in the *numeric value* T, a_i.
- If $T \leq n^{O(1)}$, then $O(nT) = n^{O(1)}$ so still efficient *in practice*.

4. DP-Technique Recap

Problem Type	Subproblems	Branching & Combine
Sequences	prefixes/suffixes/substrings	often $O(1)$ guesses (e.g. include/skip, left/right) + max / min
Integer-DP	all $0..N$	$O(N)$ guesses (rod-cutting, subset-sum)
Multi-seq DP	product of seq. spaces	e.g. LCS: 2-way guess at first letters
Graph-DP	per-vertex $\delta(s,v)$ or (u,v,k)	DAG-shortest: $\deg(v)$ guesses; BF: $\lvert E\rvert$ per level; FW: 2-way
State-Expansion	add state var. (min/max, player-turn, finger)	multiply #subprobs by small factor; guess next state

Key takeaway: By *choosing subproblems* (sequence-vs. integer-vs. graph), then *adding state* as needed, and *brute-forcing* a small set of guesses, one obtains a DP whose running time is (#subproblems) × (branching factor), often polynomial or pseudo-polynomial.

1. P vs. EXP vs. R

P: problems solvable in polynomial time $n^{O(1)}$ **EXP**: problems solvable in exponential time, e.g. $2^{n^{O(1)}}$ **R**: all decidable problems (finite-time solvable)

[scale=0.8, every node/.style=font=] [thick] (0,0) – (6,0) node[midway,above]P – (10,0) node[right]EXP – (14,0) node[right]R; [below] at (0,0) easy; [below] at (14,0) hard; [below] at (2,0) $\subset$; [below] at (8,0) $\subset$;

- P $\subsetneq$ EXP: e.g. N×N CHESS is EXP-complete.
- R strictly contains EXP: e.g. HALTING is not in R (undecidable).

2. Undecidability (Uncomputable)

- Every program $\leftrightarrow$ finite string of bits $\leftrightarrow$ an integer.
- Every decision problem $\leftrightarrow$ infinite bit-string $\leftrightarrow$ real in $[0,1]$.
- # programs = $\aleph_0$, # problems = continuum $\implies$ "most" problems have no solver.
- HALT: \Given program P, input x, does P(x) terminate?" is undecidable.

3. NP and "Lucky" Computation

NP: Decision problems solvable in *non-deterministic polynomial time*, i.e. by a "lucky" algorithm:

Make $O(\log n)$ bit-guesses, always correct, in $n^{O(1)}$ time.

Equivalently, *poly-time verifier + certificate y*:

- If answer is YES, $\exists y$ with $|y| = n^{O(1)}$ s.t. verifier $V(x, y)$ accepts in poly($|x|$).
- If answer is NO, $\forall y$, $V(x, y)$ rejects.

P$\subseteq$**NP**$\subseteq$**EXP**. Open: P vs. NP?

4. NP-Hardness & Completeness

Poly-time reduction $A \leq_P B$:

$$x \mapsto f(x) \quad \text{in polytime,} \quad \big[B(f(x)) = 1\big] \iff \big[A(x) = 1\big].$$

At least as hard: $A \leq_P B \implies$ ("B is at least as hard as A").

NP-hard: every $A \in$ NP reduces to B. **NP-complete**: $B \in$ NP and NP-hard.

$$\text{P} \ \subseteq \ \text{NP} \ \subseteq \ \text{NP-hard} \ \subseteq \ \text{EXP},$$

and NP-complete is the boundary between NP and NP-hard.

5. Examples

- 3-Partition is NP-complete.
- Jigsaw-Puzzle, Tetris (perfect-info), Subset-Sum, 3-SAT, … are NP-complete via poly-time reductions.
- n×n Chess is EXP-complete.
- "Most" natural puzzles and video games (Mario, Zelda, Sudoku, Minesweeper) are NP-complete or harder.

Key takeaway:

- **P** = efficiently solvable.
- **NP** = efficiently *verifiable* (lucky guesses).
- **NP-complete** = hardest puzzles in NP (all NP reduce here).
- P $\neq$ NP ? implies no poly-time alg for NP-complete.
- R = decidable; "most" problems lie outside R (uncomputable).

1. Course Goals (Lecture 1 Review)

Recall our three core goals for 6.006:

1. **Solve hard computational problems.** Design correct algorithms on unbounded inputs.
2. **Argue correctness.** Prove your procedure always returns the right answer (valid input).
3. **Argue efficiency.** Define a computation model (word-RAM, cost = operations) and show your algorithm scales well as $n \to \infty$.
4. *(Meta)* Communicate algorithms, proofs, and analyses clearly to others.

2. Complexity Recap (Lecture 19)

- **P** – decidable in poly(n) time.
- **EXP** – decidable in $2^{n^{O(1)}}$ time (P $\subsetneq$ EXP).
- **R** – all decidable problems; EXP $\subsetneq$ R.
- **Undecidable:** e.g. HALT (no algorithm).
- **NP** – "lucky" algorithms (non-deterministic polytime) or poly-time verifiers + certificates.
- **NP-hardness / completeness:** poly-time reductions; NP-complete sit at NP's hardest boundary.

3. Course Content Summary

Quiz 1: Fundamental Data Structures & Sorting

- *Sequence ADTs:* dynamic arrays (`push_pop_end`), sequence-AVL (insert/delete middle).
- *Set ADTs:* hash tables (expected $O(1)$), sorted arrays, set-AVL ($O(\log n)$ order operations).
- *Sorting via "find-and-extract":* Selection-sort variants ($\Theta(n^2)$), heap-sort ($n \log n$).
- *Indirect / counting / radix sort:* linear–time for integer keys in fixed range.

Quiz 2: Graph Algorithms

- *Single-source shortest paths:*
  - DAG-shortest (linear),
  - Bellman–Ford ($O(VE)$),
  - Dijkstra ($O(E + V \log V)$).
- *All–pairs shortest paths:*
  - Floyd–Warshall ($O(V^3)$),
  - Johnson's ($O(VE + V^2 \log V)$) for sparse graphs.
- *Minimum spanning tree:* Kruskal / Prim ($O(E \log V)$).
- *Union-find:* $\alpha(n)$ amortized.
- *Network flow:* augmenting-path algorithms ($O(E^2)$ etc.).

Quiz 3: Dynamic Programming (SRTBOT)

- *Subproblems* $\rightarrow$ DAG vertices, *Relation* $\rightarrow$ edges, *Topo order* + *base cases* + *combine.*
- Examples: Fibonacci, rod-cutting, subset-sum, LCS, coin-game, parenthesization, piano/guitar-fingering.

4. Beyond 6.006: Next Steps

6.046 (Design & Analysis of Algorithms)

- More advanced *data structures* (splay, skip lists, van Emde Boas).
- *Advanced paradigms*: greedy proof templates, randomized algorithms (Las Vegas vs. Monte Carlo), approximation algorithms.
- *Formal amortized analysis*: potential method.
- *Extended models*: parallel algorithms, cache-aware/oblivious.

Other Theory Frontiers

- **Randomization:** design analysis (hashing, primality, streaming).
- **Numerical algorithms:** real-number approximation, error bounds.
- **Approximation:** PTAS, FPTAS for NP-hard optimizations.
- **Advanced complexity:** P vs. NP open; space-complexity (PSPACE); parameterized complexity.
- **Quantum / parallel:** new computation models (entanglement, multi-core / distributed).

Congratulations on completing 6.006! You now have a toolbox of algorithms, proofs, and analyses to tackle real-world computational challenges.