1. Runtime Formulas & Basic Sums

Arithmetic Series: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$
Geometric Series: For $r \neq 1$, $\sum_{i=0}^{n} ar^i = a\frac{1-r^{n+1}}{1-r}$

- If $0 < r < 1$, $\sum_{i=0}^{\infty} r^i = \frac{1}{1-r} = O(1)$
- If $r > 1$, $\sum_{i=0}^{n} r^i = O(r^n)$

Sum of Squares: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$
Logarithms: $\log(n!) = \Theta(n \log n)$

- $\log_b(xy) = \log_b(x) + \log_b(y)$
- $\log_b(x^p) = p \log_b(x)$
- Change of Base: $\log_b(x) = \frac{\log_d(x)}{\log_d(b)}$
- $a^{\log_b(n)} = n^{\log_b(a)}$

2. Asymptotic Notation (Big O)

- **Big O (O): Upper Bound.** $f(n) \leq c \cdot g(n)$
- **Big Omega (Ω): Lower Bound.** $f(n) \geq c \cdot g(n)$
- **Big Theta (Θ): Tight Bound.** $c_1 g(n) \leq f(n) \leq c_2 g(n)$

Limit Comparison:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in (0, \infty) \implies f \in \Theta(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f \in O(g)$ but $f \notin \Theta(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f \in \Omega(g)$ but $f \notin \Theta(g)$

Common Runtimes (Fastest to Slowest): $O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^c) < O(c^n) < O(n!)$

3. Recursive Algorithms

Consists of a **base case** and a **recursive step**.
Master Theorem: For $T(n) = aT(n/b) + f(n)$:

1. If $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ for $c < 1$, then $T(n) = \Theta(f(n))$.

Other methods: Recursion Tree, Iteration (Unrolling), Substitution (Guess & Verify).

4. Arrays & Linked Lists

Dynamic Array Resizing:

- **Double Size when full:** Amortized cost $\Theta(1)$ per insertion.
- **Downsizing:** To avoid thrashing, halve size only when array is quarter full.

Comparison:

Operation	Array	Singly LL	Doubly LL
Access (i-th)	$O(1)$	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$	$O(n)$
Insert (end)	$O(1)$*	$O(n)$**	$O(1)$**
Insert (front)	$O(n)$	$O(1)$	$O(1)$
Delete (end)	$O(1)$	$O(n)$	$O(1)$**

*Amortized **With tail pointer

5. Stacks & Queues (ADTs)

Abstract Data Type (ADT): Defines operations, not implementation.

- **Stack (LIFO):** 'push()', 'pop()', 'peek()'. All $O(1)$*.
- **Queue (FIFO):** 'enqueue()', 'dequeue()', 'peek()'. All $O(1)$*. Can be implemented with a circular array or SLL with tail pointer.

*Amortized for array-based implementations due to resizing.

6. Trees (Binary & BST)

Tree Properties: Height of a complete binary tree is $h = \lfloor \log_2 N \rfloor$. Max nodes in a binary tree of height h is $2^{h+1} - 1$.

- **Full BT:** Every node has 0 or 2 children.
- **Complete BT:** All levels full except possibly the last, which is filled left-to-right. Used for heaps.

..
Let T be a noempty, full binary tree

1. If T has I internal nodes, the number of leaves is $I + 1$
2. If T has I internal nodes, the total number of nodes is $2I + 1$
3. If T has a total of N nodes, the number of internal nodes is $(N - 1)/2$
4. If T has a total of N nodes, the number of leaves is $(N + 1)/2$
5. If T has L leaves, the total number of nodes is $2L - 1$
6. if T has L leaves, the number of internal nodes is $L - 1$

..
Binary Search Tree (BST): For node 'n', left vals $< n <$ right vals.

- **Runtimes (Balanced):** Search, Insert, Delete are $O(\log n)$.
- **Runtimes (Worst-Case):** Skewed tree; all ops are $O(n)$.

Traversals ($O(n)$):

- **In-Order (LNR):** Gives sorted sequence for BST.
- **Pre-Order (NLR):** Useful for copying trees.
- **Post-Order (LRN):** Useful for deleting nodes.
- **Level-Order (BFS):** Uses a queue.

In-Order Traversal (Recursive)

```
function inorder(node)
  if node is null, return
  inorder(node.left)
  visit(node)
  inorder(node.right)
```

7. Binary Heaps (Priority Queue)

A **complete** binary tree with the **heap property**.

- **Min-Heap:** Parent $\leq$ Children. Root is minimum element.
- **Max-Heap:** Parent $\geq$ Children. Root is maximum element.

Array Repr. (0-indexed, node i):

- 'parent = floor((i-1)/2)', 'left = 2i+1', 'right = 2i+2'

Operations:

- 'insert': Add to end, swim up. $O(\log n)$.
- 'extractMin/Max': Replace root with last, sink down. $O(\log n)$.
- 'buildHeap' (Heapify): Bottom-up construction. $O(n)$.

Heap Swim/Sink (for Max-Heap)

```
function swim(i)
  p = parent(i)
  while i > 0 and heap[p] < heap[i]
    swap(heap[p], heap[i])
    i = p; p = parent(i)

function sink(i)
  while leftChild(i) < size
    bigger = leftChild(i)
    if rightChild(i) < size and
       heap[bigger] < heap[rightChild(i)]
      bigger = rightChild(i)
    if heap[i] >= heap[bigger], break
    swap(heap[i], heap[bigger])
    i = bigger
```

Build Max-Heap

```
function buildMaxHeap(A)
  heap_size = A.length
  // Start from last non-leaf node
  for i = floor(A.length/2)-1 down to 0
    sink(i) // Sink corrects the heap property
```

8. Comparison Sorting Algorithms

Lower Bound: Any comparison-based sort must make $\Omega(n \log n)$ comparisons in the worst case.

Algorithm	Avg Time	Worst Time	Space	Stable
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n)$	Yes
Quick Sort	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$	No
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(1)$	No

- **Merge Sort:** Divide & conquer. $T(n) = 2T(n/2) + \Theta(n)$.
- **Quick Sort:** Divide & conquer with pivot. Randomized pivot avoids worst case. In-place.
- **Heap Sort:** Uses a max-heap to sort in-place.

Quicksort Partition (Lomuto)

```
function partition(A, lo, hi)
  pivot = A[hi]
  i = lo
  for j = lo to hi-1
    if A[j] < pivot
      swap(A[i], A[j])
      i = i + 1
  swap(A[i], A[hi])
  return i
```

Merge Sort - Merge Step

```
function merge(A, p, q, r)
  // L and R are temp copies of subarrays
  let L be A[p..q], R be A[q+1..r]
  i=0, j=0, k=p
  while i < L.length and j < R.length
    if L[i] <= R[j] then A[k++] = L[i++]
    else A[k++] = R[j++]
  // Copy remaining elements, if any
  while i < L.length then A[k++] = L[i++]
  while j < R.length then A[k++] = R[j++]
```

9. Linear Time Sorting (Non-Comparison)

- **Counting Sort:**
  - Counts occurrences of elements in range k. Stable.
  - Runtime: $O(n + k)$, Space: $O(n + k)$.
- **Radix Sort:**
  - Sorts digit by digit using a stable sort (like Counting Sort).
  - Runtime: $O(d(n + k))$ where d is # of digits.
- **Bucket Sort:**
  - Distributes elements into buckets, sorts buckets. Assumes uniform input distribution.
  - Runtime: Avg $O(n + k)$, Worst $O(n^2)$.

Counting Sort

```
function countingSort(A, k)
  // C=counts array, B=output array
  let C be new array of size k+1, all 0s
  let B be new array of size A.length
  for j = 0 to A.length-1
    C[A[j]] = C[A[j]] + 1
  // C[i] now has number of elements == i
  for i = 1 to k
    C[i] = C[i] + C[i-1]
  // C[i] now has num elements <= i
  for j = A.length-1 down to 0
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
  return B
```

10. Hashing & Hash Tables

Maps keys to array indices via a hash function. Avg time for Search, Insert, Delete is $O(1)$. Load Factor $\alpha = n/m$.
Collision Resolution:

1. **Separate Chaining:** Each index stores a linked list. Search time is $O(1 + \alpha)$.
2. **Open Addressing:** Probe for the next available slot. Requires $\alpha < 1$. Deletion needs a **tombstone**.
   - **Linear Probing:** $(h(k) + i) \pmod m$

     Guaranteed to find a slot for $\alpha < 1$
     Expected search cost:
     –successful search: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$
     –unsuccessful search $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
     Causes primary clustering.
   - **Quadratic Probing:** $(h(k) + i^2) \pmod m$

     Guaranteed for find a slot for a prime table size and $\alpha < \frac{1}{2}$
     Expected search cost:
     –unsuccessful search: slides say no exact analysis is known
     Causes secondary clustering.
   - **Double Hashing:** $(h_1(k) + i \cdot h_2(k)) \pmod m$

     Expected search cost:
     –unsuccessful search: $\frac{1}{1-\alpha}$
     –successful search is faster
     Best option.

Resizing

1. **Chaining**
   (a) Goal: α constant
   (b) Double m when $\alpha \geq 8$
   (c) Halve m when $\alpha \leq 2$
2. Open Adressing
   (a) Goal: $\alpha < \frac{1}{2}$
   (b) Double m when $\alpha \geq \frac{1}{2}$
   (c) Halve m when $\alpha \leq \frac{1}{8}$

Limitations

1. Range Querires
2. Memory Consumption
3. Real Data (bias possible)

Search with Linear Probing

```
function search(key)
  i = hash(key)
  while table[i] is not empty
    if table[i].key == key
      return table[i].value
    i = (i + 1) mod m
  return null
```

11. Union-Find (Disjoint Set)

Tracks elements partitioned into disjoint subsets. **Implementations:**

- **Quick-Find:** 'find' is $O(1)$, 'union' is $O(n)$.
- **Quick-Union:** 'find' and 'union' can be $O(n)$ (skewed tree).

Optimizations for nearly constant time ($O(\alpha(n))$):

- **Union by Rank/Size:** Attach shorter/smaller tree to taller/larger tree's root. Keeps trees shallow.
- **Path Compression:** During 'find(x)', make all nodes on the path point directly to the root.

Find with Path Compression

```
function find(i)
  if parent[i] == i
    return i
  // Set parent directly to the root
  parent[i] = find(parent[i])
  return parent[i]
```

Union by Size

```
function union(i, j)
  rootI = find(i)
  rootJ = find(j)
  if rootI != rootJ
    // Attach smaller tree to larger
    if size[rootI] < size[rootJ]
      parent[rootI] = rootJ
      size[rootJ] += size[rootI]
    else
      parent[rootJ] = rootI
      size[rootI] += size[rootJ]
```