The cheatsheet will consist of the (1) defintions and theorems, (2) process, (3) canonical example with mark distribution.

1 Asymptotics

1.1 Big-O (Upper bound)

Definition: $f(n) = O(g(n))$ iff $\exists$ positive constants c and n_0 s.t. $0 \le f(n) \le cg(n)\ \forall\ n \ge n_0$

1.2 Big-Omega (Lower bound)

Definition: $f(n) = \Omega(g(n))$ iff $\exists$ positive constants c and n_0 such that $0 \le cg(n) \le f(n)\ \forall\ n \ge n_0$

1.3 Big-Theta (Tight bound)

Definition:
1. $f(n) = \Theta(g(n))$ iff $\exists$ positive constants $c_1,\ c_2,\ n_0$ s.t. $0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ \forall\ n \ge n_0$
2. $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

1.4 Small-o (Strictly slower)

Definition: $f(n) = o(g(n))$ iff $\forall c > 0, \exists n_0 > 0$ s.t. $0 \le f(n) < cg(n)$ for all $n \ge n_0$.

1.5 Small-Omega (Strictly Faster)

Definition: $f(n) = \omega(g(n))$ iff $\forall c > 0, \exists n_0 > 0$ s.t. $0 \le cg(n) < f(n)$ for all $n \ge n_0$.

1.6 Comparing function properties

1.6.1 Transitivity:

Definition:
- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$

1.6.2 Symmetry:

Definition:
Symmetry:
- $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.

Transpose symmetry:
- $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

1.6.3 Common functions

Definition:
1. $n^a = O(n^b)$, iff $a \le b$.
2. $\log_a(n) = O(\log_b(n))$, $\forall\ a, b$.
3. $c^n = O(d^n)$, iff $c \le d$.

4. If $f(n) = O(f_1(n))$ and $g(n) = O(g_1(n))$, then:
   (a) $f(n) \cdot g(n) = O(f_1(n) \cdot g_1(n))$.
   (b) $f(n) + g(n) = O(\max\{f_1(n), g_1(n)\})$.

1.6.4 Comparing functions cookbook:

Definition: Assume

- (1) $f(n) \ll g(n)$ means $f(n) = o(g(n))$
- (2) $f(n) \equiv g(n)$ means $f(n) = \Theta(g(n))$
- (3) f and h are eventually positive, i.e. $\lim_{n\to\infty} f(n) > 0$ and $\lim_{n\to\infty} h(n) > 0$.

1. $1 \ll \log^*(n) \ll \log^i n \ll (\lg n)^a \ll n^b \ll c^n \quad \forall i, a, b, c$
2. $f(n) \ll g(n) \Rightarrow h(n)f(n) \ll h(n)g(n)$
3. $f(n) \ll g(n) \Rightarrow f(n)^{h(n)} \ll g(n)^{h(n)}$
4. $f(n) \ll g(n)$ and $\lim_{n\to\infty} h(n) > 1 \Rightarrow h(n)^{f(n)} \ll h(n)^{g(n)}$

1.7 Limit method

Definition: Find the asymptotic relationship between two functions for which you might not have any intuition about.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n)) \tag{1}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n)) \tag{2}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \text{ i.e. is anything finite } \Leftrightarrow f(n) = O(g(n)) \tag{3}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \text{ i.e. non-zero } \Leftrightarrow f(n) = \Omega(g(n)) \tag{4}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \text{ s.t. } 0 < C < \infty \Leftrightarrow f(n) = \Theta(g(n)) \tag{5}$$

1.8 Polynomially-bounded

Definition:

- **Polylogarithmically bounded:** $f(n) = O((\lg n)^k) \quad \exists k > 0$
- **Polynomially bounded:** $f(n) = O(n^k) \quad \exists k > 0$
- **Exponentially bounded:** $f(n) = O(k^n) \quad \exists k > 0$

Theorem: $f(n) = O(n^k) \Leftrightarrow \lg(f(n)) = O(\lg n)$

Theorem: All logarithmically bounded functions are polynomially bounded, i.e. $(\lg n)^a = O(n^b) \quad \forall a, b > 0$

Theorem: All polynomially bounded functions are exponentially bounded, i.e. $f(n) = O(n^a) \Rightarrow f(n) = O(b^n) \quad \forall a > 0$ and $\forall b > 1$

1.9 Logarithm method

1.9.1 Limits of logs are logs of limits

Definition: $\lim_{x\to a}(\log_b f(x)) = \log_b\left(\lim_{x\to a} f(x)\right)$

Process: Suppose we want to compute $\lim_{n\to\infty}\frac{f(n)}{g(n)} = L$

1. **Take log of limit:**
$$\lg\left(\lim_{n\to\infty}\frac{f(n)}{g(n)}\right) = \lg L$$
2. **Change to limit of log and compute it:**
$$\lim_{n\to\infty}\left(\lg\frac{f(n)}{g(n)}\right) = \lg L$$
3. **Revert log by taking exponential with base 2:**
$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 2^{\lg L} = L$$

2 Logarithms

Definition:

$$a = b^c \iff \log_b a = c \tag{6}$$

- Always assume base 2.

2.1 Properties

Definition: $\forall$ real $a > 0,\ b > 0,\ c > 0,$ and $n,$ we have

1. $a = b^{\log_b a}$
2. $\log_c(ab) = \log_c a + \log_c b$
3. $\log_b a^n = n \log_b a$
4. $\log_b a = \dfrac{\log_c a}{\log_c b}$
5. $\log_b \left(\dfrac{1}{a}\right) = -\log_b a$
6. $\log_b a = \dfrac{1}{\log_a b}$
7. $a^{\log_b c} = c^{\log_b a}$
8. $\log_b \dfrac{a}{c} = \log_b a - \log_b c$

2.2 Logarithm iteration

Definition: $\log(n)$ iteratively applied i times to an initial value of n.

$$\log^{(i)}(n) = \begin{cases} n & \text{iff } i = 0, \\ \log\left(\log^{(i-1)}(n)\right) & \text{if } i > 0. \end{cases} \tag{7}$$

2.3 Iterated logarithm function

Definition: The minimum number of times i that the logarithm function must be applied to n for the result to be less than or equal to 1:

$$\lg^* n = \min\left\{i \geq 0 \ :\ \lg^{(i)} n \leq 1\right\} \tag{8}$$

Warning: $O(n \log^* n) \approx O(n)$

3 Summations

3.1 Fibonacci Numbers

Definition:

$$F_i = \begin{cases} 0 & \text{if } i = 0, \\ 1 & \text{if } i = 1, \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2. \end{cases} \tag{9}$$

3.2 Arithmetic series

Definition:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2) \tag{10}$$

3.3 General arithmetic series

Definition: For $a \geq 0$ and $b > 0$,

$$\sum_{k=1}^{n} (a + bk) = \Theta(n^2) \tag{11}$$

3.4 Sum of squares

Definition:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{12}$$

3.5 Sum of cubes

Definition:

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4} \tag{13}$$

3.6 Finite geometric series

Definition: For $x \neq 1$,

$$\sum_{k=0}^{n} x^k = 1 + x + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} \tag{14}$$

3.7 Infinite decreasing geometric series

Definition: For $|x| < 1$,

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \tag{15}$$

3.8 Harmonic series

Definition: For positive integers n, the nth harmonic number is

$$\sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1) \tag{16}$$

3.9 Telescoping series

Definition: For any sequence $a_0, a_1, \dots, a_n$,

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0 \quad \text{OR} \quad \sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n \tag{17}$$

3.10 Reindexing summations

Process:

1. **Start with the original sum**

$$S = \sum_{k=a}^{b} f(k)$$

2. **Introduce a new variable:** Let the new variable ℓ be defined in terms of the old variable k, $\ell = g(k)$
3. **Substitute the new variable into the sum:** $f(k) = h(\ell)$
4. **Adjust the limits of summation:** When $k = a$, $\ell = g(a)$, and when $k = b$, $\ell = g(b)$.
5. **Rewrite the summation in terms of the new variable:**

$$S = \sum_{\ell=g(a)}^{g(b)} h(\ell)$$

- **Intuition:** If the summation index appears in the body of the sum with a minus sign, it's worth thinking about reindexing.

4 Induction, Contradiction, & Combinatorial Arguments

4.1 Direct proof

Process:
1. Start with the givens
2. Mathematically manipulate the givens and/or reason about the givens to arrive at the conclusion.

4.2 Weak Induction

Process: Given predicate $P(n)$
1. **Basis Step:** Prove $P(n_0)$ for some value n_0.
2. **Hypothesis:** Assume $P(k)$ is true for $n = k$.
3. **Inductive step:** Use the hypothesis $P(k)$ to show its true for $n = k + 1$ s.t. $P(k + 1)$.

Therefore, $\forall n \geq n_0$, $P(n)$ is true.

4.3 Strong Induction

Process: Given predicate $P(n)$
1. **Basis:** Show $P(n_0), P(n_1), \ldots$ are true.
2. **Hypothesis:** Assume $P(k)$ is true, $\forall k \leq n$.
3. **Step:** Use the hypothesis $P(n_0) \wedge \cdots \wedge P(k) \wedge \cdots \wedge P(n)$ to show its true for $P(k + 1)$.

4.4 Contradiction

Process: Given predicate $P(n)$ either true or false.
1. Assume toward a contradiction $\neg P(n)$.
2. Make some argument by working with the expression $\neg P(n)$ to get to a contradiction.
3. Arrive at a contradiction
4. If this resulted in a contradiction then $P(n)$ is true.

4.5 Combinatorial argument

Process:
1. **Question:** Ask a question relating to the formula you would like to prove. Choose the easier side to start and make the question based on this side.
2. **LHS:** Argue why the LHS answers the question.
3. **RHS:** Argue why the RHS answers the question.

Intuition/Tips:
- $+$: Or event
- $\cdot$: And event
- Subtracting inside a combination usually means were **excluding** or **removing** objects.

5 Recurrences and the Master Theorem

5.1 Master Theorem

Theorem: Let $a \geq 1$, $b > 1$, and $f(n)$ be a function, so that the recurrence is

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{18}$$

Then the asymptotic behavior of $T(n)$ is

1. If $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ for $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b(a)}\right)$, then $T(n) = \Theta\left(n^{\log_b a}\log n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right)$ for $\epsilon > 0$ and $af\left(\frac{n}{b}\right) \leq cf(n)$ for $0 < c < 1$, then $T(n) = \Theta(f(n))$.

Process:

1. Identify $T(n)$
2. State a, b, and $f(n)$. Make sure the conditions are met.
3. Calculate $n^{\log_b a}$.
4. Compare $f(n)$ with $n^{\log_b a}$ to see which case the function applies too.
   (a) If ϵ case is used, then apply an abitrary value to see (usually natural numbers work well).
5. Write down the answer with the corresponding case.

5.2 Substitution

Process:

1. Guess the form of the solution for $T(n) =?$ (given on a test)
2. Use induction to show that the solution works.
   (a) Basis: Find the base case using values of n that correspond (i.e. make sense) with the guessed solution.
   (b) Inductive hypothesis:
   (c) Inductive step:
3. Find the constants.

5.3 Recursion tree

Process:

1. Draw a recursion tree using $T(n)$ and split it off into the other T's.
2. Sum the work within each level of the tree to obtain the per-level work.
3. Determine the height, which is $r^h n = 1$
   - 1: When the recursion problem size is 1.
   - r: Largest value child from the root.
   - n: Initial problem size
   - **Key:** The root's children bigger value determines the height of the tree as it will reach $O(1)$ (i.e. base-case) the slowest compared to the other branch, making it the longest path in the recursion tree.
4. Determine the total work: $h \cdot$ Work done at every level

6 Graphs

Definition: $G = (V, E)$, where $V = \{\text{vertices}\}$ and $E = \{\text{edges}\}$.

6.1 Directed and Undirected, Weighted Graphs

Terminology:

- **Directed graph (digraph):** Each edge has a direction from one vertex to another.
  - **Edges:** (v_1, v_2) and (v_2, v_1) are different.
  - **Self-loop:** Edges from a vertex to itself.
  - **In/Out Degree of V:** Out-degree is the # of edges leaving it, while in-degree is the # of edges entering it.
  - **Degree of V:** In-degree plus out-degree.
- **Undirected graph:** Each edge does not have a specific direction.
  - **Edges:** (v_1, v_2) and (v_2, v_1) are indifferent.
  - **Self-loop:** Forbidden.
  - **Degree of V:** Number of edges incident on it.
- **Weighted graph:** A graph where each edge is associated with a value (e.g. distance, profit, penalty).
- **Simple graph:** Graph with no self-loops, or multi-edges.
- **Induced graph:** Subset of G and the associated edges.
- **Spanning subgraph:** Let $G' = (V', E')$ be a subgraph of $G = (V, E)$, G' is a spanning subgraph if $V' = V$.

6.2 Paths and cycles

Terminology:

- **Path:** Going from one vertex to another.
- **Simple Path:** A path with no repetition of vertices.
- **Cycle:** A path that begins and ends at the same vertex, which can pass through the same vertex multiple times.
- **Simple Cycle:** A path that begins and ends at the same vertex, with no other repeated vertices.

6.3 AG, DAG

Terminology:

- **Acyclic Graph:** A graph with no cycles is an acyclic graph.
- **Directed Acyclic Graph:** A DAG is a directed acyclic graph.

6.4 Connected, disconnected graph

Terminology:

- **Connected:** Two vertices are connected if there is a path between them.
- **Connected graph:** $\exists$ path between $\forall$ 2 vertices.
- **Disconnected graph:** $\exists$ at least one pair of vertices such that no path exists between them.

6.5 Bipartite Gs

Definition: V can be partitioned into 2 sets V_1 and V_2 s.t. $V_1 \cap V_2 = \emptyset$ and $V_1 \cup V_2 = V$ and adjacencies only between elements of V_1 and V_2.

6.6 Clique (complete G)

Definition: $\exists$ edge between $\forall$ 2 vertices.

$$\#edges = \frac{V(V-1)}{2} = \binom{V}{2} \tag{19}$$

6.7 Degrees of all V

Definition:

$$\sum_{v \in V} degree(v) = 2|E| \tag{20}$$

6.8 Graph representation

Definition:
Adjacency matrix (AM): An $n \times n$ matrix where $M[i][j] = 1$ if there is an edge between v_i and v_j, and 0 otherwise (undirected) or $M[i][j] = 1$ if there is an edge from v_i to v_j, and 0 otherwise (directed).

- **Time to search for E:** $O(1)$ (i.e. since in matrix format)
- **Memory space:** $O(V^2)$ (i.e. matrix has x^2 entries)
- Good for dense G (i.e. $E >> V^2$)
- For a directed graph, when you take the transpose of G, you get the complement of the original graph.

Adjacency list (AL): For $n = |V|$ vertices, n linked lists. The ith linked list, $L[i]$ is a list of all the vertices that are adjacent to vertex i.

- **Time to search for E:** $O(V)$ (i.e. may be on the last vertex)
- **Memory space:** $O(V + E)$ (i.e. store all the vertices and edges once)
- Good for sparse G (i.e. $E << V^2$).

7 Trees

Definition: A tree is a connected, acyclic, undirected graph.

7.1 Properties

Definition: Let $G = (V, E)$ be an undirected graph. The following statements are equivalent:

1. G is a tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
4. G is connected, and $|E| = |V| - 1$.
5. G is acyclic, and $|E| = |V| - 1$.
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

7.2 Terminology

Terminology:

- **Parent:** A node y is the parent of node x if y is directly connected to x on the path from the root.
- **Child:** A node x is a child of node y if y is the parent of x.
- **Siblings:** Nodes are siblings if they share the same parent.
- **Leaf (or External Node):** A leaf is a node with no children.
- **Internal Node:** An internal node is a nonleaf node, which means it has at least one child.
- **Degree:** The degree of a node x is the number of children it has.
- **Level:** A level of a tree consists of all nodes at the same depth.

7.3 Height, depth

Definition:

- **Depth:** The depth of a node x is the number of edges from the root to x.
- **Height:** The height of a node is the number of edges in the longest path from that node to a leaf.
  - **Height of tree:** From root to any leaf.

7.4 K-ary trees

Definition: Each node $\leq k$ children. A binary tree has $k = 2$.

A **complete k-ary tree** is a k-ary tree in which all leaves have the same depth, and every internal node has exactly k children.

8 Permutations, Combinations

8.1 Rule of sum and product

Definition: If there are m-ways for event A to happen and n-ways for event B to happen then...

Rule of product: $\exists\ m \times n$ ways for A *and* B to happen.

Rule of sum: $\exists\ m + n$ ways for A *or* B to happen.

8.2 Factorials

Definition: Number of ways to arrange n distinct objects when order is important.

$$n! = n(n-1)(n-2)\cdots 2\cdot 1 \tag{21}$$

8.3 Permutations

Definition: Number of ways to pick r distinct objects out of n where *order matters* and *repetition isn't allowed.*

$$P(n,r) = n(n-1)(n-2)\cdots(n-r+1) = \frac{n!}{(n-r)!} \tag{22}$$

- n: total number of elements in the set.
- r: number of elements taken from the set.

8.4 Permutations with identical items

Definition: If there are m kinds of items and q_k, $k = 1, \ldots, m$ of each kind, then total number of permutations where *order matters* is

$$\binom{n}{q_1, \ldots, q_m} = \frac{n!}{q_1!\, q_2! \cdots q_m!} \tag{23}$$

- $\sum_{k=1}^{m} q_k = n$

8.5 Permutations with repetitions

Definition: Number of ways to arrange r-objects out of n objects with unlimited repetition is given by: n^r.

8.6 Combinations

Definition: Number of ways to choose r objects from n where *order doesn't matter.*

$$C(n,r) = \binom{n}{r} = \frac{P(n,r)}{r!} = \frac{n!}{r!(n-r)!} \tag{24}$$

9 Probability

9.1 Sample space

Definition: The set of *all possible outcomes* of a statistical experiment, denoted by S.

9.2 Event

Definition: A subset of a sample space S. An event is any outcome or combination of outcomes.

9.3 Probability axioms

Definition: For A, $B \subseteq S$

1. $0 \leq P(A) \leq 1$
2. $P(S) = 1$
3. $P(A \cup B) = P(A) + P(B)$ for two mutually exclusive events A and B.

9.4 Mutually exclusive events

Definition: If $A_1, A_2, A_3, \ldots$ is a sequence of mutually exclusive events, then

$$P(A_1 \cup A_2 \cup A_3 \cup \ldots) = P(A_1) + P(A_2) + P(A_3) + \ldots \tag{25}$$

- **Key**: Independent is **NOT** mutually exclusive.
- **Key Implication**: For independent events, we can multiply their probabilities to get their intersection (useful for **calculating favorable outcomes**).
  - $P(A \cap B \cap C \cap D) = P(A)P(B)P(C)P(D)$ as long as they are **INDEPENDENT**.
  - **Note**: Can also be applied to a sequence of dependent events as long as you adjust for changing conditions.

9.5 Additive rule

Definition:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \tag{26}$$

9.6 Uniform distribution

Definition: If $\forall s \in S$ has probability $P(s) = \frac{1}{|S|}$, then it is a uniform distribution.

9.7 Independence

Definition: $P(A \cap B) = P(A)P(B)$ if A, B independent.

9.8 Bayes theorem

Definition: For events with $P(A) > 0$ and $P(B) > 0$, the probability A happens given B happens is:

$$P(B|A) = \frac{P(B \cap A)}{P(A)} = \frac{P(A|B)P(B)}{P(A)} \tag{27}$$

9.9 Bayes' rule with total probability

Definition: Suppose $C_1, \ldots, C_k$ is a partition. Then

$$P(B|A) = \frac{P(B)P(A|B)}{\sum_{i=1}^{k} P\left(C_i\right) P\left(A|C_i\right)} \tag{28}$$

Often B is an element of $C_1, \ldots, C_k$, say $B = C_n$. Then

$$P\left(C_n|A\right) = \frac{P\left(C_n\right) P\left(A|C_n\right)}{\sum_{i=1}^{k} P\left(C_i\right) P\left(A|C_i\right)} \tag{29}$$

Process:

1. Write down all the probabilities.
2. Try solving the problem directly using definitions.

Intuition/Tips: If given P(A|B) and want P(B|A), then automatically use Bayes' Rule.

9.10 Discrete random variable

Definition: An RV is a function that associates a real number with each element of the sample space. Denote RVs with capital letters.

9.11 Probability mass function

Definition: The set of ordered pairs $(x, f(x))$ of the discrete RV X if, for each possible outcome x,

1. $f(x) \geq 0$ for each outcome $X = x$
2. $\sum_{x} f(x) = 1$ (i.e. total probability sums to 1)
3. $f(x) = P(X = x)$ (i.e. probability of each outcome)

9.12 Expectation

Definition: Let X be an RV with distribution $f(x)$, then

$$E[X] = \sum_{x \in X} x f(x) \tag{30}$$

where the sum is taken over all possible values of X.

9.13 Properties of expectation

Definition:

1. $E[X+Y] = E[X] + E[Y]$ (linearity)
2. $E[\alpha X] = \alpha E[X]$ (linearity)
3. $E[XY] = E[X]E[Y]$ if independent

10 Heaps & Heapsort

10.1 Overview

Summary:
- **Stable:** Relative order of ties is maintained.
  - e.g. $[2_a, 3, 2_b, 1] \rightarrow [1, 2_a, 2_b, 3]$
- **Inplace sorting:** Given an array A to sort the numbers, sorts within the array and uses a constant number of variables to do bookkeeping (i.e. only need the memory of the array)
  - e.g. merge sort is $O(n)$ space but it is still in place.

Intuition/Tips:
- **Height:** $2^h \leq n \leq 2^{h+1} - 1 \iff h = \lfloor \log n \rfloor$
  - h: Height
  - n: Number of elements.
- **Max:** The maximum (in the case of a max-heap) value in a heap will always occur at the root.
- **Time Complexity**: $O(n \log n)$.
- **Space complexity:** $O(1)$ (space complexity doesn't include the input).
- **Sorting algorithm properties:** Heap-sort is an in place algorithm, but is NOT stable.
- **Explanation:** Describe in terms of a tree, but only for visualization. Do not code a tree for heapsort.
- **Pseudo-code:** Uses the array representation.

10.2 Indexing

Definition: Given a node at index i in the array:
1. **Parent:** $\text{parent(i)} = \left\lfloor \frac{i}{2} \right\rfloor$
2. **Left child:** $\text{leftchild(i)} = 2i$
3. **Right child:** $\text{rightchild(i)} = 2i + 1$

10.3 Heap-tree: (2 properties)

Definition:
- **Heap Shape:** A complete binary tree where the last level is not filled, but leaves are pushed to the left.
- **Heap Order (maxheap):** $\text{A[Parent(i)]} > \text{A[i]}$
- **Heap Order (minheap):** $\text{A[Parent(i)]} < \text{A[i]}$

10.4 Height

Definition: A heap of n elements is based on a complete binary tree, its height is $\Theta(lgn)$.

10.5 Operations

Summary:

Operation	**Time Complexity**
Bubble down	$O(\log n)$
Build heap	$O(n)$
Extract max	$O(\log n)$
Heapsort	$O(n \log n)$
Insert	$O(\log n)$
Bubble up	$O(\log n)$

10.5.1 Bubble down

Algorithm:

```
    bubble_down (i):
        repeat
            compare A[i] with A[2i] and A[2i + 1] # Compare with its children.
            exit if A[i] is larger or A[i] is a leaf # Max heap property satisfied.
            swap A[i] <=> swap(A[2i], A[2i + 1]) # Swap with larger element between children.
```

10.5.2 Build heap

Algorithm:

```
    Build_heap (A):
        for i = floor(length / 2) down to 1 # Since floor(length/2) + 1 are all
                                            # leaf nodes, so no point in bubble_down.
                                            # floor(length/2) is the last internal node.
            bubble_down(A, i) # bubble down to appropriate spot.
```

10.5.3 Extract max

Algorithm:

```
Extract_Max (A):
    max = A[1] # Extract the root.
    A[1] = A[length] # Put last element at the top
    length = length - 1 # Decrease the size of the array
    bubble_down(A[1]) # Bubble down the last element to the proper location
```

10.5.4 Heapsort

Algorithm:

```
Heapsort (A):
    Build-Heap(A)       # O(n)
    for i = 1 to n - 1  # O(n) iterations
        Extract_Max(A)  # O(log n)
```

10.5.5 Insert

Algorithm:

```
    Insert:
        A[length + 1] = new_key # Add to end of array
        length = length + 1 # Increase the length
        bubble_up(A, length) # Bubble up to appropriate spot
```

10.5.6 Bubble up

Algorithm:

```
    bubble_up (A, i):
        repeat
            swap (A[i] <=> A[floor(i/2)]) # comparing yourself with parent
            if A[i] is larger, exit

```

11 Quicksort

Summary:

Algorithm:

```
        Quicksort (list in, int left, int right):
            pivot = Partition(in, left, right)   # Partition the list and return the pivot
            if (pivot > left):                  # If elements on the LS of pivot
                Quicksort(in, left, pivot)      # Apply QS on the left partition
            if (pivot < right):                 # If elements on the RS of pivot
                Quicksort(in, pivot + 1, right) # Apply QS on the right partition

```

```
        int Partition (in, left, right):
            ls = left                          # Left pointer (starting index)
            pivot = in(left)                   # Choose the leftmost element as the pivot
            for i = left + 1 to right:         # Iterate over the rest of the elements
                if (in(i) <= pivot):
                    ls = ls + 1                # Increment ls to track smaller elements
                    swap(in(i), in(ls))        # Swap current element with the element at ls
            swap(in(left), in(ls))             # Place pivot element in correct position
            return ls                          # Return the index of the pivot

```

11.1 Running times

Summary:

Case	Time Complexity
Worst case	$O(n^2)$
Best case	$O(n \log n)$
Average/expected case	$O(n \log n)$

11.2 Randomized QS

Algorithm:

```
        def QS(C):
            pivot = RAND_partition(C)

            if pivot > left:
                QS(C[left : pivot])

            if pivot < left:
                QS(C[pivot : right])

```

```
        Rand-Partition (list in, left, right)
            i = random(left, right)
            swap(in(left), in(i))
            return Partition(in, left, right)

```

12 Counting Sort

Summary:
- **Stable sorting:** Not in-place but stable sorting.
- **Stable:** Stable sorting in counting sort ensures that elements with equal values retain their relative order from the original input when sorted.
- **Time complexity:** Time $O(n+k)$ if $O(k)=O(n)$ implies $O(n)$

12.1 Lower bound on comparison-based sorting

Definition: No **comparison-based** sorting algorithm on **unrestricted** range (i.e. any numbers) can do better than $\Omega(n\log(n))$.

12.2 Decision tree

Definition: Any sorting algorithm can be written as a decision tree. (Goes both ways)

Theorem: Any decision tree for an n-element sorting algorithm has $h=\Omega(n\log n)$

Algorithm:

```
        C[i] = 0 for all i in [0...k]
        for j = 1 ... length(A) do  # O(n)
            C[A[j]] = C[A[j]] + 1
        for i = 1 ... k /* prefix sums */ # O(k)
            C[i] = C[i] + C[i-1]
        for j = length(A) ... 1 do # O(n)
            B[C[A[j]]] = A[j]
            C[A[j]] = C[A[j]] - 1

```

- A: array to be sorted.
- **Sorting range:** Assume numbers in range $[0...k]$.
- **Auxilliary arrays:** $C[0\ldots k]$ and $B[1\ldots n]$.

13 Radix Sort

Summary:

Algorithm:

```
        for i = Least significant bit (LSB) -> Most significant bit (MSB)
            counting_sort(digit) # or any stable sorting algorithm on digit i

```

13.1 Time complexity

Definition:

- **Variables:**
  - n: # numbers
  - r: range of numbers
  - d: # digits
- **One pass complexity:** $= O(n + r)$ (i.e. the time complexity of counting sort)
- **All passes complexity:** $dO(n + r) = O(dn + dr)$ (i.e. sorting all digits, so it's the time complexity of counting sort times the number of digits)
- **r=d complexity:** If $r = d = O(1)$, then $O(n)$ true for all passes.

14 BSTs

Summary:

Definition: A binary tree with the **BST property**:

- For each node in the tree, all values in the left subtree are smaller than the node's value, and all values in the right subtree are larger than the node's value.
- i.e. Keys let subtree $\leq$ Parent key $\leq$ Keys right subtree.

14.1 Operations

Summary:

Operation	**Time Complexity**
Search	$O(h)$
Insert	$O(h)$
Build	$O(n^2)$
Sort	$O(n)$
Delete	$O(h)$

14.1.1 Search

Algorithm: Probe the root and recursively go left or right according to the value to search.

```
search(x):
    start from root
    repeat
        recursively go left or right according to the value to search
    end

```

14.1.2 Insert

Algorithm: Search for it and insert it as a leaf.

```
insert(x):
    search(x)
    insert it as a leaf when hit a leaf

```

14.1.3 Build

Algorithm:

```
build(A[1...n])
    for i = 1 to n:
        insert(A[i]) into the tree

```

14.1.4 Sort

Algorithm:

```
inorder(root):
    visit L (left subtree)
    print node value
    visit R (right subtree)
```

```
```

14.1.5 Delete

Algorithm:

```
deleteBST(x):
    search(x) # (i.e. search for the key)
    if x is a leaf:
        # Just delete x
        delete(x)

    elif x has one child:
        # Delete x and replace it with its child (i.e. upgrade child)
        replace(x, x.child)

    elif x has two children:
        # Replace x with its in-order predecessor or successor
        replace(x, in_order_predecessor_or_successor(x))

```

14.2 Structurally balanced BSTs

Definition:

$$O(\log(n)) \text{ i.e. balanced} \leq O(h) \leq O(n) \text{ i.e. unbalanced} \tag{31}$$

15 RBTs

Summary:

Definition: Structurally balanced, i.e., $h = O(\log n)$. RBT is a BST with the following properties:

- Every node is either red or black.
- The root is always black.
- A red node has only black children.
- For all path from root to leaf, there is the same number of black nodes.
    - **Generalizable:** Any node to leaf has the same number of black nodes.

15.1 Black height

Definition: The black height $bh(x)$ is the number of black nodes on any path to a leaf, excluding x.

15.2 Balancing proof:

Theorem: A RBT with n internal nodes has

$$h \leq 2\log(n+1) = O(\log n) \tag{32}$$

Theorem: **Lemma:** A subtree in a Red-Black Tree rooted at x has **at least**

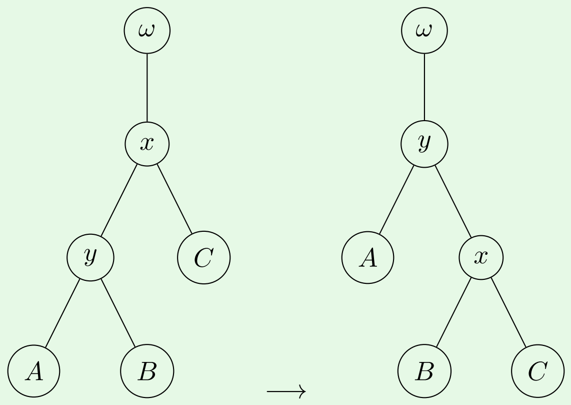
$$2^{bh(x)} - 1 \tag{33}$$

internal nodes.

15.3 Rotations

15.3.1 Right rotation

Definition: **Right Rotation:** A right rotation is performed on a node x when its left child y becomes the new root of the subtree. After rotation, x becomes the right child of y. This rotation maintains the BST property.



15.4 Insert operation

16 Hashing

16.1 Hash tables

Definition: A **hash table** uses a hash function to map keys from the universe of keys U to positions in an array of size m called buckets.

- The key is stored in the appropriate bucket.
- The number of keys stored is n.
- The **load factor** α is given by: $\alpha = \frac{n}{m}, \quad 0 < \alpha \leq 1$
  - Expect $n \leq m$ so the places to store in the array is bigger than the number of keys to store in n.

16.2 Hash function

Definition: A **hash function** $h(\text{key})$ is used to convert a key into a numeric value, which is then used to index the array.

- i.e. Maps a key in U to an index from 0 to $m - 1$.
- A good hash function minimizes collisions, where two or more keys are assigned to the same index.

16.3 Methods

16.3.1 Division Method

Definition: The **division method** computes the hash as:

$$h(\text{key}) = \text{key} \mod m$$

- m is typically a prime number.

16.3.2 Multiplication Method

Definition: The **multiplication method** computes the hash as:

$$h(\text{key}) = \lfloor m \cdot ((\text{key} \cdot A) \mod 1) \rfloor$$

- $A = \frac{\sqrt{5} - 1}{2}$, the golden ratio, is used to spread keys more evenly.
- $m = 2^p$ are good values.

16.4 Resolutions

16.4.1 Resolution by Chaining

Definition: Resolves collisions by maintaining a linked list at each bucket. If two or more keys hash to the same index, they are stored in the same linked list.

16.4.2 Resolution by open addressing

Definition: Resolves collisions by probing for the next available slot in the table.

- Instead of maintaining a list of keys at each bucket, each key is placed directly in the table, and a probing sequence is followed if collisions occur.

16.4.3 Probing Methods

Definition:

1. **Linear Probing:** Probe and insert next element:

$$L_0 = h(\text{key})$$

If the bucket is full (i.e., a collision), then we check the next slot in the table until an empty one is found (i.e., reprobe):

$$L_{j+1} = (L_j + 1) \mod m$$

- However, linear probing can lead to **clustering**, where groups of keys form large contiguous blocks in the array.

2. **Double Hashing:** Using two hash functions to avoid clustering. If a collision occurs at index i_0:

$$L_1 = h_1(\text{key})$$

The next probe location is computed by:

$$L_{j+1} = (L_j + h_2(\text{key})) \mod m$$

- This results in better distribution of keys across the table and reduces clustering.

17 Dynamic programming

Process:

1. Visualize example.
2. **Optimal substructure:** Characterize the structure of an optimal solution)
3. **Recursive formula:** Find a relationship among sub-problems (i.e. defines the values of an optimal solution recursively in terms of the optimal solution to sub-problems)
   (a) Base case(s)
   (b) Recursive formula
4. Compute the value of an optimal solution (bottom-up solving sub-problems in order or top-down solving problem recursively)
5. **Time complexity:** $O(n^{\text{\# subproblems per choice}})O(\text{\# choices})$

17.1 How to prove optimal substructure?

Process:

Example:

18 Greedy Algorithms

Process:

1.

Example: