1 Asymptotic Notation

1.1 Big-O Notation

$$f(n) = O(g(n)) \text{ if } \exists c > 0, n_0 > 0 \text{ s.t } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

- This means that $f(n)$ grows **at most** as fast as $g(n)$ for large n.
- Upper bound on the growth rate of a function.

1.2 Big-Omega Notation

$$f(n) = \Omega(g(n)) \text{ if } \exists c > 0, n_0 > 0 \text{ s.t } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0$$

- This means that $f(n)$ grows **at least** as fast as $g(n)$ for large n.
- Lower bound on the growth rate of a function.

1.3 Big-Theta Notation

$$f(n) = \Theta(g(n)) \text{ if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

- This means that $f(n)$ grows **as fast** as $g(n)$ for large n.
- Tight bound on the growth rate of a function.

1.4 Theorem

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

1.5 Polynomially Bounded Functions

$$P(n) = O(n^k) \text{ for some constant } k \in \mathbb{R}$$

Useful Theorem to prove that a function is polynomially bounded:

$$f(n) = O(n^k) \iff \lg(f(n)) = O(\lg(n))$$

1.6 Little-O Notation

$$f(n) = o(g(n)) \iff f(n) = O(g(n)) \wedge f(n) \neq \Theta(g(n))$$

1.7 Little-Omega Notation

$$f(n) = \omega(g(n)) \iff f(n) = \Omega(g(n)) \wedge f(n) \neq \Theta(g(n))$$

1.8 Limit Method

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = o(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c\ ;\ 0 < c < \infty \implies f(n) = \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c\ ;\ 0 \leq c < \infty \implies f(n) = O(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c\ ;\ 0 < c \leq \infty \implies f(n) = \Omega(g(n))$

1.9 Useful Facts

- $n^a = O(n^b) \iff a \leq b$
- $c^n = O(d^n) \iff c \leq d$
- $\log_a(n) = O(\log_b(n)) \quad \forall a, b > 1$

1.10 Function Order

From slowest to fastest growing functions:

1. $O(c)$
2. $O(\log^*(n))$
3. $O(\log^{(i)}(n))$
4. $O(\log(n))$
5. $O(n)$
6. $O(n \cdot \log(n))$
7. $O(n^{1+c})$
8. $O(c^n)$
9. $O(d^n)$, $d > c$
10. $O(n!)$
11. $O(n^n)$

2 Proof Methods

2.1 Contradiction

Prove P:

1. Assume towards a contradiction that $\neg P$.
2. Derive a contradiction. (e.g by showing that $\neg P \implies Q$ and $\neg Q$)
3. Conclude that P is true. $(\neg\neg P \to P)$

2.2 Induction

Prove $P(n)$ for all $n \geq k$:

IB Base case: Prove $P(k)$.

IH Inductive hypothesis: Assume $P(n)$ is true for **some** $n \geq k$ (**for all** $n \geq k$ in strong induction).

IS Inductive step: Assuming IH, prove $P(n+1)$.

3 Useful Math

3.1 Permutations

- Order matters.
- $P(n,k) = \frac{n!}{(n-k)!}$
- Let q_i be the number of objects of t classes. The number of ways to arrange n objects is $\frac{n!}{q_1! \cdot q_2! \cdot \ldots \cdot q_t!}$

3.2 Combinations

- Order does not matter.
- $C(n,k) = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$

4 Recurrence

Useful for divide-and-conquer algorithms.

$$T(n) = \sum_{i=0}^{k} a_i \cdot T(g_i(n)) + f(n)$$

- a_i is the number of subproblems.
- $g_i(n)$ is the size of the subproblems.
- $f(n)$ is the cost of dividing the problem and combining the solutions.

Note: It can happen that $\sum_{i=0}^{k} a_i \cdot g_i(n) \neq n$

4.1 Master Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

- $a \geq 1$ is the number of subproblems.
- $b > 1$ is the factor by which the problem size is reduced.
- $f(n) > 0$ is the cost of dividing the problem and combining the solutions.

4.1.1 Case 1

$$f(n) = O(n^c) \text{ for some } c < \log_b(a)$$
$$T(n) = \Theta(n^{\log_b(a)})$$

- The cost of the work done at the current level dominates the cost of the work done at the lower levels.

4.1.2 Case 2

$$f(n) = \Theta(n^c) \text{ for some } c = \log_b(a)$$
$$T(n) = \Theta(n^c \cdot \log(n))$$

- The cost of the work done at the current level is equal to the cost of the work done at the lower levels.

4.1.3 Case 3

$$f(n) = \Omega(n^c) \text{ for some } c > \log_b(a)$$
$$T(n) = \Theta(f(n))$$

- The cost of the work done at the lower levels dominates the cost of the work done at the current level.
- Always check if the regularity condition holds: $a \cdot f\left(\frac{n}{b}\right) \leq k \cdot f(n)$ for some constant $k < 1$ and sufficiently large n.

4.1.4 Recipe

1. Identify a, b, and $f(n)$.
2. Compute $\log_b(a)$.
3. Compare $f(n)$ with $n^{\log_b(a)}$.
4. Apply the corresponding case of the Master Theorem.

4.2 Substitution Method

1. Guess the form of the solution. (e.g $T(n) = O(g(n))$)

IH Assume that $T(k) \leq c \cdot g(k)$ for all $k < n$.

IS Prove that $T(n) \leq c \cdot g(n)$.

4.3 Recursion Tree Method

- Longest path: Upper bound.
- Shortest path: Lower bound.
- Total cost of the tree: $O(\text{cost of each level} \cdot h(T))$
- Total cost: $\sum_{i=0}^{h(T)} c_i$

5 Graphs

$G = (V, E)$

- V is a set of vertices.
- E is a set of edges.
- $E \subseteq V \times V$

5.1 Graph Representation

5.1.1 Adjacency List

- $j \in L[i] \iff (i, j) \in E$.
- $O(E)$ space. $O(V^2)$ worst case.
- $O(V)$ time to list all vertices adjacent to a vertex.
- Sparse graphs.

5.1.2 Adjacency Matrix

- $M[i][j] =$ weight of the edge between i and j.
- $O(V^2)$ space.
- $O(1)$ time to check if there is an edge between two vertices.
- Dense graphs.

6 Free Trees

A free tree is:

- Connected
- Undirected
- Acyclic

6.1 Equivalent Definitions

1. G is a free tree.
2. Any 2 vertices are connected by a unique simple path.
3. G is connected, but removing any edge disconnects it.
4. G is connected and has $|V| - 1$ edges.
5. G is acyclic and has $|V| - 1$ edges.
6. G is acyclic, but adding any edge creates a cycle.

6.2 Handshaking Lemma

$$\sum_{v \in V} \deg(v) = 2|E|$$

Note: any graph has an even number of vertices with odd degree.

6.3 More Graph Properties

Source: Discrete Mathematics IIC1253 - PUC Chile

6.3.1 1. Cycle

- $V = \{0, 1, 2, ..., n-1\}$
- $E = \{\{i, (i+1) \bmod n\} | 0 \leq i \leq n-1\}$
- Last element is connected to the first element.

6.3.2 2. Isomorphism

- $G_1 = (V_1, E_1)$
- $G_2 = (V_2, E_2)$
- $G_1 \cong G_2$ if there is a bijection $f : V_1 \to V_2$ such that $\{u, v\} \in E_1 \iff \{f(u), f(v)\} \in E_2$
- $\cong$ is an equivalence relation (reflexive, symmetric, transitive).

6.3.3 3. More Definitions

- **Degree**: Number of edges incident to a vertex. $\deg(v) = |\{u \in V | \{v, u\} \in E\}|$
- **Path**: Sequence of vertices $v_1, v_2, ..., v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$.
  - **Simple Path**: Path with no repeated vertices.
  - **Closed**: Path with $v_1 = v_k$. It ends where it starts.
  - **Cycle**: Closed path with no repeated edges.
- **Connected**: There is a path between any pair of vertices.

7 Trees

- $T = (V, E)$ is a tree if $\forall x, y \in V$, with $x \neq y$, there is a unique path between x and y.
- A tree is a connected acyclic graph.
- A leaf is a vertex with degree 1.

7.1 Theorem

- If T is a tree and v is a leaf, then $T - v$ is a tree.
- This theorem is useful for induction.

7.2 Binary Trees

- $\forall v \in V$, $\deg(v) \leq 3$
- if v is the root, $\deg(v) \leq 2$
- $\forall v \in V$, $\max(\text{children}(v)) \leq 2$

7.3 Theorem

$$\text{\# of vertices without children} = \text{\# of vertices with exactly two children} + 1$$

7.4 Complete Binary Trees

- $\#$ leaves $= 2^h$
- $\#$ vertices $= 2^{h+1} - 1$
- $h \leq \log_2(|V|)$

8 Heaps

An array A of elements such that:

- A heap is almost a complete binary tree.
- All except last row are full.
- Max-heap: $\forall i$, $A[\text{parent}(i)] \geq A[i]$
- Min-heap: $\forall i$, $A[\text{parent}(i)] \leq A[i]$

8.1 Indexing

Assume index array starts at 1.

- $\text{parent}(i) = \lfloor i/2 \rfloor$
- $\text{left}(i) = 2i$
- $\text{right}(i) = 2i + 1$

8.2 Key Takeaways

- $2^h \leq n \leq 2^{h+1} - 1 \iff h = \lfloor \log(n) \rfloor$
- Max-heap root is the largest element.
- Min-heap root is the smallest element.
- Max-heaps are useful for sorting in decreasing order. Sorting algorithm: Heap Sort.
- Min-heaps are useful for priority queues.

Note: Heaps is also referred to as garbage collection storage in PL such as Java and Python, note that this is not the same as the data structure.

9 Sorting Algorithms

9.1 Comparison-Based Sorting Algorithms

9.1.1 Heap Sort

- Time Complexity: $O(n \cdot \log(n))$
- Space Complexity: $O(1)$
- In-place sorting algorithm.
- Not stable: does not preserve the order of equal elements.

```
def Max-Heapify(A, i):
    """
    Enforces the max-heap property on the subtree rooted at index i.
    O(h) = O(log(n))
    """
    # Compare the root with the left and right children
    l = 2 * i
    r = 2 * i + 1

    # If A[i] is smaller, then swap with the largest child
    if A[i] < A[l] or A[i] < A[r]:
        largest = l if A[l] > A[r] else r
        A[i], A[largest] = A[largest], A[i]

        # Recursively downwards until the max-heap property is satisfied
        Max-Heapify(A, largest)

def Build-Max-Heap(A, n):
    """
    Builds a max-heap from an unsorted array A. O(h log(n))
    """
    for i in range(floor(n // 2), 1, -1):
        Max-Heapify(A, i)

def Heap-Sort(A, n):
    """
    Sorts an array A in-place using the heap sort algorithm.
    O(n log(n))
    """
    Build-Max-Heap(A, n)
    for i in range(n, 2, -1):
        A[1], A[i] = A[i], A[1]
        n -= 1
        Max-Heapify(A, 1)
```

9.1.2 Quick Sort

- Worst Case Time Complexity: $O(n^2)$. Occurs when the pivot is always the smallest or largest element.
- Best Case: Occurs when the pivot is always the median.
- Average Case Time Complexity: $O(n \cdot \log(n))$
- $T(n) = T(an) + T(bn) + O(n)$; $a + b = 1$
- In-place sorting algorithm.
- Not stable: does not preserve the order of equal elements.

```
def Partition(A, p, r):
    """
    Partitions the array A[p:r] around the pivot A[r]. O(n)
    """
    x = A[r]
    i = p - 1
    for j in range(p, r):
        if A[j] < x:
            i += 1
            A[i], A[j] = A[j], A[i]
    A[i + 1], A[r] = A[r], A[i + 1]
    return i + 1

def Quick-Sort(A, l, r):
    """
    Sorts the array A[l:r] using the quick sort algorithm. O(n log(n))
    """
    if l < r:
        p = Partition(A, l, r)
        Quick-Sort(A, l, p - 1)
        Quick-Sort(A, p - 1, r)
```

10 Sorting Algorithms

10.1 Randomized Quick Sort

Randomly select the pivot to avoid worst-case time complexity.

```
def Randomized-Partition(A, p, r):
    """
    Partitions the array A[p:r] around a random pivot. O(n)
    """
    i = random.randint(p, r)
    A[i], A[r] = A[r], A[i]
    return Partition(A, p, r)
```

10.1.1 Theorem

A comparison-based sorting algorithm requires $\Omega(n \cdot log(n))$ comparisons in the worst case.

10.2 Non-Comparison-Based Sorting Algorithms

10.2.1 Counting Sort

- Assumes elements are integers in the range 0 to k
- Not in-place. Requires additional space
- Stable: preserves the order of equal elements
- Time Complexity: $O(n + k)$, if $k = O(n)$ then $T(n) = \Omega(n)$
- Only works on countable sets
- Stability property is important for radix sort subroutines

```
def Counting-Sort(A, B, k):
    """
    Sorts the array A using the counting sort algorithm. O(n + k)
    """
    C = [0] * (k + 1)
    for j in range(1, len(A)):
        C[A[j]] += 1
        # C[i] now contains the number of elements equal to i
        # e.g C[3] = 2 means that 3 appears twice in A
    for i in range(1, k + 1):
        C[i] += C[i - 1]
        # C[i] now contains the number of elements less than or equal to i
    for j in range(len(A), 0, -1):
        B[C[A[j]]] = A[j]
        C[A[j]] -= 1

    return B
```

10.2.2 Radix Sort

- Sorts the elements by their digits
- Assumes all elements have $\leq d$ digits
- Each digit is in the range 0 to k
- Digit Time Complexity: $\Theta(d \cdot (n + k))$
- Overall Time Complexity: $\Theta(n)$
- Not in-place. Requires additional space
- Stable: preserves the order of equal elements

- Goes from the least significant digit to the most significant digit, running counting sort on each digit iteration

```
def Radix-Sort(A):
    """
    Sorts the array A using the radix sort algorithm. O(d(n + k))
    """
    max_element = max(A)

    # Apply counting sort to sort elements based on place value.
    place = 1
    while max_element // place > 0:
        Counting-Sort(A, place)
        place *= 10
```

10.2.3 Selection Sort

TODO

11 Binary Search Trees

- If y is in the left subtree of x, then $key(y) \leq key(x)$
- If y is in the right subtree of x, then $key(y) \geq key(x)$
- Given a n-node BST: $h = O(n)$
- If the tree is balanced: $h = O(log(n))$

11.1 Operations

- **Search, Min & Max**: $O(h)$
- **Insertion & Deletion**: $O(h)$
- **Predecessor & Successor**: $O(h)$
- **Building a BST**: $O(n \cdot h)$
  - Worst case: $O(n^2)$

11.2 Algorithms

1. Traversals: $O(n)$
   - **Inorder**:
     - Left, Root, Right
     - Sorting
   - **Preorder**:
     - Root, Left, Right

- Rotation
- **Postorder**:
  - Left, Right, Root
  - Deleting

2. Search: $O(h)$
   - **Min**: Always go left
   - **Max**: Always go right
   - **Search**: Compare with the root, go left or right. Divide and conquer

3. Successor & Predecessor: $O(h)$
   - **Successor**: Go right, then left until the leftmost node. If the node has a right child, then the successor is the minimum of the right subtree. Else, go up until the node is the left child of its parent
   - **Predecessor**: Go left, then right until the rightmost node. Symmetric to the successor

4. Deletion & Insertion: $O(h)$
   - **Deletion**:
     - Case 1: Leaf. Just delete
     - Case 2: One child. Replace with the child
     - Case 3: Two children. Replace with the successor
   - **Insertion**:
     - Search for the node to insert
     - Insert as a leaf

12 Red-Black Trees

Balanced BST with $O(log(n))$ for most operations.

Black-height(x) = Number of black nodes on the path from x (exclusive) to the leaves (inclusive).

12.1 Properties

1. Every node is either red or black
2. Root is black
3. Leaves are black (NIL nodes)
4. Red nodes have black children
5. Every path from a node to its leaves has the same number of black nodes (not including the node itself)

12.2 More Properties

- $h(T) \leq 2 \cdot bh(root)$
- $h(T) \leq 2 \cdot lg(n+1)$
- Subtree rooted at x has at least $2^{bh(x)-1}$ internal nodes
- Longest path (root to farthest leaf) is at most twice the shortest path
- Shortest: All black nodes
- Longest: Alternating red and black nodes

12.3 Operations

BST operations can break RBT properties. Fix with rotations.

13 Hash Tables

13.1 Definitions

- **Hash Table**: Data structure that implements an associative array abstract data type. Indices $\{0, ..., m-1\}$ are called slots
- **Hash Function**: Maps keys to indices in the hash table. $h(k) : K \subset U \to \{0, ..., m-1\}$
- **Collision**: Two keys map to the same index
- **Load Factor**: $\alpha = \frac{n}{m}$, where n is the number of elements and m is the number of slots
- **Simple Uniform Hashing**: Each key is equally likely to hash to any slot

13.2 Key Takeaways

- Hash Table ($O(m)$) uses less memory than array ($O(|U|)$)
  - Example: Storing 10 32-bit numbers in an array uses 2^{32} slots, while a hash table uses 10 slots
- Problem: Collisions
  - Example: if $|U| > m$, then there are more keys than slots

13.3 Chaining

- Each slot is a linked list
- Collisions are resolved by chaining
- Insert k: append k to the linked list at h(k)
- If $m > n$, then $\alpha = O(1)$
- With SUH, the expected length of the linked list is α

- Expected time complexity: $O(1+\alpha) \sim O(n)$
- If $\alpha = O(1)$, then $O(1)$ time complexity
- Note that if we want $O(lg(n))$ performance, we can use RBT instead of linked lists

13.4 Open Addressing

- Slower than Universal Hashing, but less memory
- Use hash function to find the next available slot

13.4.1 Techniques

1. **Linear Probing**: $h(k,i) = (h(k) + i) \mod m$
2. **Quadratic Probing**: $h(k,i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$
   - better performance than linear probing
   - still m probing sequences
   - "secondary clustering"
3. **Double Hashing**: $h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$
   - $h_2(k)$ must be relatively prime to m $\iff gcd(h_2(k), m) = 1, \forall k$
   - no clustering
   - $\Theta(m^2)$ different probing sequences

14 Dynamic Programming

- Epitome of divide-and-conquer
- Efficient recursion for solving **well-behaved** optimization problems (i.e, optimal solution given constraints)
- Concept of **Memoization**: Store the results of expensive function calls and return the cached result when the same inputs occur again

Used for problems with the following properties:

1. **Optimal Substructure**: Optimal solution can be constructed from optimal solutions of subproblems
2. **Overlapping Subproblems**: Solving the same subproblem multiple times. Memoization can be used to store the results of subproblems

- Dynamic Programming is good for tasks with small and repetitive search spaces
- Examples: Fibonacci, Longest Common Subsequence, Shortest Path, Knapsack, Matrix Chain Multiplication

Methodology in Tutorial 6.

15 Greedy Algorithms

- **P1. Greedy Choice Property**: A global optimal solution can be reached by making the first greedy choice
- **P2. Optimal Substructure**: Optimal solution to the problem contains optimal solutions to subproblems
- **P3. Smaller Subproblems**: After making a greedy choice, reduce the problem to a smaller instance
- Approach to solve $O(n^2)$ or NP complete problems
- Greedy is not always optimal, but provides a good approximation

Proof of correctness template in Tutorial 7.

Proof of optimality template in Tutorial 7 Notes.