COMP 2123 Cheat Sheet

Week 1 : Analysis

· Constant (1) < logarithmic (log n) < Polylogarithmic (log(log n)) < Square root ($\sqrt{n}$) < Linear (n) < quasilinear (n log n) < Quadratic (n^2) < $n^2 \log n$ < Cubic (n^3) < exponent (2^n) < Factorial ($2!$)

· Big O-notation → upper bound (worst case)
· Big Omega notation → lower bound (best case)
· Big Theta (θ) → Tight bound

Log Properties

· $\log_b a = \frac{1}{\log_a b}$ · $\log_b C = \frac{\log_a C}{\log_a b}$

· $a = b^{\log_b a}$ · $b^{\log_c a} = a^{\log_c b}$

Week 2 : ADT (Abstract Data Type)

· Array based List → element stored at A[i]
· Singly Linked List → Reference to the first node
· Doubly Linked List → Link to element, Prev, and next

Big O-notation

Case	Array	Singly linked list	Doubly linked list
Accessing element	O(1)	O(n)	O(n)
Insert/removing from beginning	O(n)	O(1)	O(1)
Insert/removing from middle	O(n)	O(n)	O(n)
Insert/removing from end	O(1)	O(n)	O(1)
Replacing an element somewhere	O(1)	O(1) or O(n)	O(1) or O(n)

· Stack = LIFO → Last In, First Out (Pop, Push)

Operations:

- Push = Insert an element at the beginning of the list → $\boxed{O(1) \text{ time}}$
- Pop = Remove and return the first element in the list → $\boxed{O(1)}$

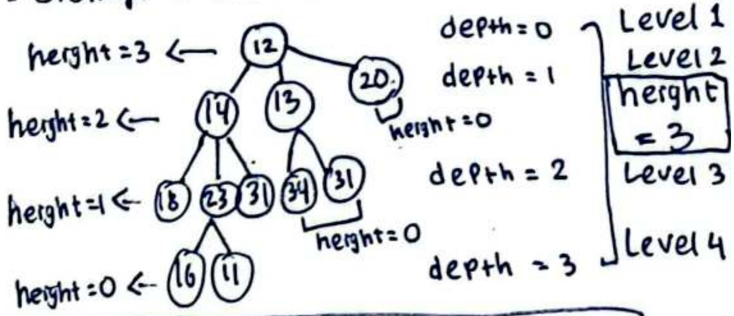
· Queue = FIFO → First in, First Out (dequeue 1 2 3 enqueue)

Operations:

- enqueue = Insert the element at the end of the list (keep a pointer at the end) → $\boxed{O(1) \text{ time}}$
- dequeue = remove and return the first element → $\boxed{O(1)}$

Week 3 : Trees

- Node has at most one parent
- root = node without parent
- Internal node = node with at least one child
- External / leaf node = node without children
- Ancestors = Parent, grandparent, great-grandparent
- Descendants = Child, grandchild, great-grandchild
- Siblings = Two node with same parent



$\boxed{\text{Root : height} = 3, \text{depth} = 0, \text{Level} = 1}$

Order trees:

· Pre-Order = Visit the node before its descendants
[12, 14, 18, 23, 16, 11, 31, 13, 34, 31, 20]

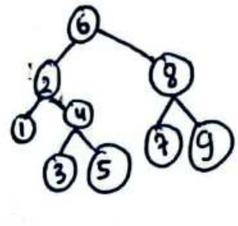
· Post-Order = Visit node after its descendants
[18, 16, 11, 23, 31, 14, 34, 31, 13, 20, 12]

B. Binary Trees

· Each node has at most 2 children
→ Proper BT = Every internal node has 2 child

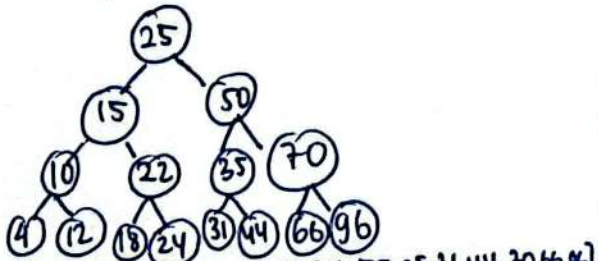
Order trees:

· In-Order = Visit L before R



Euler tour traversal = Visit each node three times

- On the left (PreOrder)
- From below (In Order)
- On the right (PostOrder)



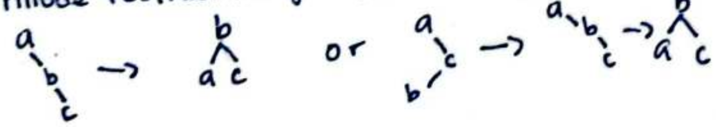
Pre-order = [25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 96]
Post-order = [4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 96, 70, 50, 25]
In Order = [4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 96]

Week 4 : Binary Search Trees

- Key(left) < key(node) < Key(R)
- Internal nodes -> store key value pairs
- External nodes -> does not store items
- Search() -> trace downward starting from root decide go left/right based on the comparison key at node with given key
  - ↳ runs in O(h) -> worst case = O(n)
    - ↳ O(log n) = balanced trees
- Insertion() = If present, replace value
  If not, expand node by replacing external node with new node
- Deletion() = If node has 1 child, promote the child and replace the node
  If node has 2 child,
  - Find internal node greater than the one that we want to remove
  - Find the internal node has the smallest value in the right tree according to in-order
  - Promote the node and replace the node using the new node.
- Duplicate keys -> key (L) ≤ key(node) < key (R)
  - ↳ use list to store duplicates
- Range Queries -> search all keys k such that $k_1 \le k \le k_2$

$key(V) < k_1$ -> search to right subtree
$key(V) > k_2$ -> search to left subtree
$k_1 \le key(V) \le k_2$ -> Add L to output, search R
Total running time = $\boxed{O(|output| + \text{tree height})}$

- Trinode restructuring -> Balancing trees



Takes O(1) time to update

- Rank Balanced trees - keep a rank for every AVL tree
- AVL trees -> rank balanced trees
- Where r(v) is its height of the subtree rooted at v
- The ranks of the two children of every internal node is differ by at most 1.

Height of an AVL tree = O(log n)
Space = O(n)
Operations = O(log n)

Week 5 : Priority Queue

- Store collection of key value items where we can only remove smallest (min heap) or highest (max heap)
- Sequence based PQ

Operations	Unsorted	Sorted
size, IsEmpty	O(1)	O(1)
Insert	O(1)	O(n)
remove min, min remove max, max	O(n)	O(1)

- Sorting = Selection Sort -> $O(n^2)$
  Insertion Sort -> $O(n^2)$
  Heap Sort -> O(n log n)
- Heap
  Binary tree storing (key, value) items in its nodes
- Max heap = every parent node is greater than or equal to its children
- Min heap = every parent node is smaller than or equal to its children
- Upheap = Operation where a node is moved up the heap until the heap property is restored
- Downheap = Operation where a node is moved down the heap until the heap property is restored
- Height of a heap = O(log n)
- Heap PQ
  - ↳ min = max = O(1)
  - ↳ Insert = O(log n)
  - remove min = remove max = O(log n)

From tutorial:

- We can calculate number of inversions by doing Selection Sort -> #Swaps = # Inversions for A[i] > A[j]
- Finding the kth value in sorted order given array A we can use min-heap or max-heap

1. min heap: use to track the smallest element and the root is the smallest. When we are using min heap, it means the root is kth biggest element which is small since we are replacing the root when A[i] > root
2. max heap = used the same as above but we replace the root when A[i] < root which means the root is kth smallest which is big.

- Given k sorted lists of m length, merge lists into one
  - We can create a min heap where smallest element is in the root. We insert the element into the heap that takes O(log k) and there are k items so, it will take O(k log k) time for insertion and deletion. It essential to we deletion since we have to delete the root and output it so that we can get the sorted order. While there are m length of lists -> $\boxed{O(mk \log k) \text{ time}}$

Week 6 : Hash Tables

- Use a hash Function h to map keys to Corresponding indices in array A
- Probability of Collision = 1/N
- Collision Handling :
  1. Separate Chaining
     - Create a list within hash value
     - Load Factor = n/N
     - Expected($1+\alpha$) , Worst case = O(n)
  2. Open addressing with Linear Probing
     - Colliding item placed in a different cell of the table
     - Linear Probing = Place Colliding item in the next available Cell
- Operations:
  1. Search
     - Start at cell h(k) -> Probe Consecutive locations until an item with key is Found or empty cell is Found or N cells have been Probed
     - Can't Put element in the deFunct
  2. Put()
     - If k is Found, replace value
     - If not, Store at Place after the First Defunct

get, Put, remove = $\boxed{O(n)}$ if randomly distributed

Number of probes is $1/(1-\alpha)$ where $\alpha = \frac{n}{N}$

If $\alpha < 1$, it's $\boxed{O(1)}$

3. Cuckoo hashing
   - Use 2 hash Function and 2 hash table
   - get, remove = O(1)
   - evict Previous item + insert new -> evicted goes to its other Possible Place
   - Handling eviction Cycle :
     1. Keep track of the sequence of evictions and check If the key evicted twice from the Same position
     2. Flag the entry, So that when we visit the Same entry, we know that we have visited before which means a cycle.

From Tutorial :

1. get, Put, delete -> O(1) by using hash table and doubly linked list with Pointer that Points to the element in the hash table
2. Finding a most Frequent value in O(n) time -> using hash table with linear Probe/chaining to handle the Collision and we will traverse the hash table and Same element will be addressed to the Same address which we can access in O(1) time and traversing = $\boxed{O(n)}$
3. Multimap to use get() in O(1+s) time -> using a hash table where we augment the multimap to the hash table. Since multimap Store Multiple value in the Same address, when we call get(), it will be O(1) time + O(s) where s is number of value -> $\boxed{O(1+s)}$

Week 7 : Graph

-> n = vertices, m = edges

- Consist of Pair (V, E)
- Edge -> directed = one way
  - -> undirected = both ways
- Graph representation
  1. Adjacency List
     - each vertex keeps a sequence of edge adjusted to it
  2. Adjacency matrix
     - 2D array = Reference to edge object For adjacent Vertices
     - Null For non adjacent Vertices

	Edge list	Adjacency List	Adjacency matrix
Space	O(n+m)	O(n+m)	$O(n^2)$
Incident edges	O(m)	O(deg(v))	O(n)
getEdge	O(m)	O(min(deg(u), deg(v)))	O(1)
Insert vertex	O(1)	O(1)	$O(n^2)$
Insert edge	O(1)	O(1)	O(1)
remove vertex	O(m)	O(deg(v))	$O(n^2)$
remove edge	O(1)	O(1)	O(1)

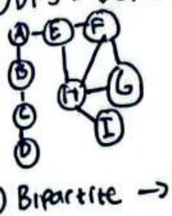
- Tree -> T is Connected -> no Cycle
- Forest -> graph with no cycle -> Connected Component are trees

From Tutorial :

① DFS = Depth , BFS = Layer/Level

BFS = {A}, {B, E}, {C, H, F}, {D, G, I}

DFS = A, B, C, D, H, E, F, G, I



② Bipartite -> BFS (Per layer) -> Same layer / intra layer = Not bipartite ; -> different = bipartite

BFS = Find Connected Component
DFS = Compute Connected Component } $\boxed{O(m+n) \text{ time}}$

③ Detect Cycles

1. Adjacency list : DFS -> mark every node, If we go to the Same = Cycle -> O(n)
2. Adjacency matrix : DFS -> mark every entries (rows and columns) We need to check and Find neighbour -> $O(n^2)$

④ get stuck in adjacency matrix -> check the in-degree and out-degree If in-degree is 1 but out is 0 -> Stuck

⑤ Identifying Cut Vertices and Cut edges

Cut edges	Cut Vertices
For an edge (u,v) to be cut edges where u = Parent, v = Child in the DFS tree $\boxed{\text{Down-and-UP}[v] > \text{level}(u)}$ -> Cut edges If: Down-and-UP[v] ≤ level(u) (u,v) is not Cut edges	The root of DFS tree Should has two child - If the node has two child then the node Should be the cut vertex - If there is internal node u which has two child, u is a cut vertex if and only if $\boxed{\text{Down-and-UP}[v] \geq \text{level}(u)}$

Week 8 : Min Spanning Tree x Shortest Path

- Weighted graph → edge weight consist of numerical value
- Shortest Path → Find min path of total weight between u and v.

1. Dijkstra's algorithm → used for finding the shortest path from starting node to all other nodes in weighted graph. It can handle directed and undirected graph with positive edge weight. The algorithm is initially 0 in the starting point and ∞ for all edges and updates accordingly.

   Performance:
   - $O(m)$ except PQ operations
   - heap as PQ : $O(m \log n)$
   - Fibonacci heaps = $O(m + n \log n)$
   - Binary heap = $O((m+n)\log n)$
   - Array = $O(m^2)$

2. Prim's algorithm → used for finding the minimum spanning tree of a connected graph. Grows the minimum spanning tree by adding the minimum edge weight that connects the MST to a new vertex.

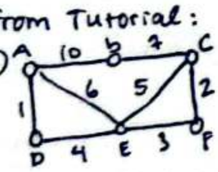
   Performance:
   - $O(m \log m)$ → binary heap
   - $O(m \log n)$ → heap
   - $O(m + n\log n)$ → Fibonacci heap

3. Kruskal's algorithm → Method for finding the minimum spanning tree of a connected graph by sorting all edges in increasing order of their weights. Choose the edge that doesn't contain a cycle

   Performance
   - Sorting = $O(m \log m)$
   - DFS to test if cycle occurs → $O(mn)$

- Simple-union Find
  - make-Sets = $\Theta(n)$
  - Find(U) = $\Theta(n)$
  - Kruskal's algo = $O(n^2)$ → edge weight sorted

From Tutorial:

① 

Kruskal = {AD, CF, EF, DE, CB}

Prims = {AD, DE, EF, CF, BC}

② Shortest Path Problem, edge has weight and vertex has cost. Find the path minimizes the total length + cost vertices
- Construct a new graph G' based on the graph G, adding edge (U', V') to G' with weight ℓ + cost(V). Run dijkstra's and the shortest path correspond to the original graph G. Using binary heap = $O((m+n)\log n)$ or Fibonacci heap : $O(m + n \log n)$
- binary heap • insertion = $O(\log n)$

  Dijkstra = $O(m+n)$, total = $O(m+n) * O(\log n)$
- Fibonacci heap

  the edges can trigger the decrease key operation when the edge weight is relaxed

  $O(m) * O(1) = O(m)$ and insertion on vertices = $O(n\log n)$

  Total = $\boxed{O(m + n\log n)}$

Week 9 : Greedy Algorithm

- making the locally optimal choice at each stage with the hope of finding global optimum

1. Fractional knapsack
   - Given a set S of n items with each item i having b_i (benefit) and W_i (weight), we choose items with max total benefit of weight

     ↳ best solution = $\frac{b_i}{w_i}$ (ratio)

   Complexity:
   - Sorting : $O(n \log n)$

     ↳ use PQ as heap < $O(\log n)$ removal, $O(n)$ process in the for loop

2. Task Scheduling
   - Find min number of classes to schedule all the lectures such that no 2 occur at the same time & place → Interval Partitioning = $O(n \log n)$

3. Text Compression
   - Given a string X, efficiently encode it to smaller Y

   a. Run-length Coding = encoding based on number of characters (12W, 8B, 1C)

   B. Huffman encoding → $O(n + d \log d)$ < n = size of X, d = distinct characters

   Let C be the set of characters in X. Compute Freq. f(C) for each character c in C. Encode high frequency char with short code words

   C. Encoding Tree = mapping each char to binary

a	b	c	d	e
00	010	011	10	11

→ 

From Tutorial:

① For the algorithm to always return optimal solution, it's not always the minimum difference between two things, instead it's the minimum of each element which will also be minimum in the difference as well

② When we are given $f = 2^i$ to construct a tree, it means that every internal node has an external node because the algo creates a new node and it picks the smallest frequency which corresponds to the leaf(ext node) which internal always have ext (2^i)

③ For every point in the set, to determine the length interval that contains all points, we have to sort it first and checking if the point has been covered or not. If not, we start from that point for the length to cover the point → $O(n \log n)$.

④ For each job and time to get optimal schedule, we need to sort the job weight and compute the ratio of job weight / time and hence we can return the optimal schedule maximizing the weight → $O(n\log n)$

Week 10: Divide and Conquer 1

- Divide and Conquer → divide, recurse, Conquer or divide, recurse, merge
- Binary Search → $T(n/2) + O(n) = \boxed{O(n) \text{ time}}$
  - If the array is empty → return "No"
  - Otherwise, compare x to middle element $A[\frac{n}{2}]$
    If $x > A[\frac{n}{2}]$ → Search in the $A[\frac{n}{2}+1]$ to $A[n]$
    If $x < A[\frac{n}{2}]$ → Search in the left $A[0]$ to $A[\frac{n}{2}-1]$
- Merge Sort → divide to two halves and recurse until one element only, Keep track of the smallest element with pointer for each halves and compare with the value → Repeat until both lists are merged
- Recurrence by Unrolling

$$T(n) \begin{cases} T(n) = 2T(n/2) + O(n) & \text{For } n>1 \\ T(n) = O(1) & \text{For } n=1 \end{cases}$$

$T(n) = 2T(n/2) + O(n)$
$T(n/2) = 2T(n/4) + O(n/2)$
$T(n/4) = 2T(n/8) + O(n/4)$

$\boxed{T(n) = 2T(n/2) + O(n) \quad (1)}$

Put $T(n/2)$ to the equation above

$T(n) = 2(2T(n/4) + O(n/2)) + O(n)$
$\quad = 4T((n/4) + O(n)) + O(n)$

$\boxed{T(n) = 4T(n/4) + 2O(n) \quad (2)}$

Put $T(n/4)$ to the equation above

$T(n) = 4(2T(n/8) + O(n/4)) + 2O(n)$
$\quad = 8T(n/8) + O(n) + 2O(n)$

$\boxed{T(n) = 8T(n/8) + 3O(n) \quad (3)}$

we have base case $T(n) = O(1)$ when $n=1$

$T(n) = 2^a T(n/2^a) + a\,O(n)$
$T(1) = T(n/2^a) \rightarrow 1 = \frac{n}{2^a} \rightarrow a = \log_2 n$
$T(n) = 2^{\log_2 n} + O(1) + \log_2 n \; O(n)$

$\boxed{T(n) = O(n \log n)}$

From tutorial:

1. Majority element: Divide the array until each array has only two elements, Compare → If the same then majority → merge the array → $O(n \log n)$
2. Inversion while merge sort → $\boxed{O(n \log n)}$ Divide the array into two halves and while doing the sort, count the number of inversions by updating the number of element in the left each time we pick from right → $A[i] > A[j]$
3. Smallest non-negative integer → Check the index with the middle, if the same → wrong in right halves
   If not same → wrong in the left halves → $\boxed{O(\log n)}$
4. Majority in $O(n)$ time → Pairing the element, and removes the element that don't form a majority
   This method traverse the array and checking $O(1)$ takes $\boxed{O(n)}$ time

Week 11: Divide and Conquer 2

Recurrence	Solution
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 2T(n/2) + O(\log n)$	$T(n) = O(n)$
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(n)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$

- Maxima Set → Point is max if all other points in the set either have a smaller x or y coordinates
  1. Sort the point by increasing X coordinate and store in array → $O(n \log n)$
  2. Divide → sorted array into two halves $O(n)$
  3. Recur → Recursively find the MS of each half $T(n) = 2T(n/2)$
  4. Conquer → Compute the MS of union left and right $O(n)$

  Total $= 2T(n/2) + O(n) = \boxed{O(n \log n) \text{ time}}$
- Integer Multiplication → Compute the product of two n-digit numbers by making 3 recursive call on $\frac{n}{2}$ digit numbers and then combine
  Total $= 3T(n/2) + O(n) \rightarrow O(n^{\log_2 3})$
- Geometric Series Fact:
  Let r be positive real and k a positive integer
  $1 + r + r^2 + \dots + r^k = (r^{k+1} - 1)/(r-1)$
  If $r > 1$ then
  $1 + r + r^2 + \dots + r^k < (r^{k+1})/(r-1)$
  If $r < 1$ then
  $1 + r + r^2 + \dots + r^k < (1/1-r)$
- Master's Theorem

Case 1
If $F(n)$ is $O(n^c)$ where $c < \log_b a$
$T(n) = O(n^{\log_b a})$

Example
$8T(n/2) + O(n^2)$
$a=8, b=2, c=2 \rightarrow T(n) = \boxed{O(n^3)}$
$\log_b a = 3 > 2$

Case 2
If $f(n)$ is $O(n^c \log^k n)$ for $k \geq 0$
$T(n) = O(n^c \log^{k+1} n)$

Example
$2T(n/2) + O(n \log n) \rightarrow k=1$
$a=2, b=2, c=1 \rightarrow T(n) = \boxed{O(n \log^2 n)}$
$\log_2(2) = 1 = c$

Case 3
If $f(n)$ is $O(n^c)$ where $c > \log_b a$
If $a f(n/b) \leq k f(n)$ for $k < 1$
$T(n) = O(f(n))$

Example Case 3

$2T(n/2) + O(n^2)$

$a=2,\ b=2,\ C=2$

$\log_2 2 = 1 < C = 1 < 2$

$T(n) = \boxed{O(n^2)}$

From tutorial:

$T(n) = 8T(n/2) + O(n^2)$

$$T(n) = 8^i T\left(\frac{n}{2^i}\right) + \underbrace{4O(n^2) + 2O(n^2) + O(n^2)}_{\frac{2^i - 1}{2-1}}$$

base case $T(n) = O(1)$ when $n=1$

$1 = \frac{n}{2^i} \rightarrow i = \log_2 n$

$$T(n) = 8^{\log_2 n} O(1) + \frac{2^{\log_2 n} - 1}{1} \quad (n-1)$$

$$= n^3 \cdot O(1) + O(n^2)(n-1)$$

$$= \boxed{O(n^3)}$$

② $7T(n/2) + O(n^2)$

$T(n) = 7^i T(n/2^i) + 49O(n^2/16) + 7O(n^2/4) + O(n^2)$

$T(n) = 7^i T(n/2^i) + \dots + \frac{7^i}{4^i} O(n^2)$

$i = \log_2 n$

$$T(n) = 7^{\log_2 n} \cdot O(1) + \frac{7^{\log_2 n}}{4^{\log_2 n}} \cdot O(n^2)$$

$$= O(n^{\log_2 7}) \cdot O(1) + O(n^{\log_2 7 - 2 + 2})$$

$$= \boxed{O(n^{\log_2 7})}$$

③ $T(n) = 3T(2n/3) + O(1)$ for $n>1$

$T(1) = O(1)$ when $n=1$

$T(2n/3) = 3T(4n/9) + O(1)$

$T(4n/9) = 3T(8n/27) + O(1)$

$T(n) = 3T\left(\frac{2^k n}{3^k n}\right) + O(1)$

$k = \log_{3/2} n$

$T(n) = O(n^{\log_{3/2} 3}) = \boxed{O(n^{2.71})}$

④ Finding local optimal index

Each recursion step cuts the problem size by a half

If there's only one elements, we return the element

otherwise, find the middle index

If middle < mid - 1 and mid < mid + 1 -> return middle

If mid > mid - 1 -> recursively search the left half

else -> recursively search the right half

runs in $O(\log n)$ since it cuts the problem into a half.

⑤ Two sorted lists of size m and n. Find the kth smallest element

We can conducting binary search across two lists which used to eliminate the half of the remaining elements in every step. Since the list are sorted, we can compare the middle element of the remaining portions of the list to decide which half to eliminate. Based on the comparison, eliminate the half from either the first or second list and hence we find the kth smallest. -> runs in $\boxed{O((m+n)\log n)}$