# CS2040S Midterm Cheatsheet — YashyPola AY22/23 S2

## Asymptotic Analysis

The order of growth of an algorithm can be represented by $T(n)$, which can be said to be bounded loosely by $O(n)$ and $\Omega(n)$, and tightly by $\Theta(n)$.

### Definitions

1. $T(n) = O(F(n)) \iff \exists c, n_0 > 0$ such that $\forall n > n_0$, $T(n) \leq cF(n)$.
2. Similarly, $T(n) = \Omega(F(n)) \iff \exists c, n_0 > 0$ such that $\forall n > n_0$, $T(n) \geq cF(n)$.
3. Similarly, $T(n) = \Theta(F(n)) \iff \exists c, n_0 > 0$ such that $\forall n > n_0$, $T(n) = cF(n)$

### Common Recurrence Relations (Runtime)

1. $T(n) = 2T(n/2) + O(n) = O(nlogn)$
2. $T(n) = 2T(n/2) + O(1) = O(logn)$
3. $T(n) = 2T(n/4) + O(1) = O(\sqrt{n})$
4. $T(n) = T(n/2) + O(n) = O(n)$
5. $T(n) = T(n/2) + O(1) = O(logn)$
6. $T(n) = T(n-c) + O(n) = O(n^2)$
   where c is some small number relative to n
7. $T(n) = 2T(n-c) + O(1) = O(2^n)$
   where c is some small number relative to n
8. $T(n) = 2T(n/2) + O(nlogn) = O(n(logn)^2)$
9. $T(n) = aT(\frac{n}{b}) + f(n) \quad a \geq 0, b > 1$

$$= \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$

### Notables

1. $\sqrt{n}logn = O(n)$
2. $O(2^{2n}) \neq O(2^n)$ [Can always find an n for which $2^{2n} > 2^n$]
3. $O(log(n!)) = O(nlogn)$ by Sterling's Approximation
4. $T(n-1) + T(n-2) + .... + T(1) = 2T(n-1)$

### Properties of Asymptotic Notation

1. Addition: $f(n) + s(n) = O(max(f(n), s(n)))$
2. Multiplication: $O(f(n)) * O(s(n)) = O(f(n) * s(n))$
3. Composition: $f \circ s = O(f \circ s)$ only if both are increasing
4. $max(f(n), s(n)) \leq f(n) + s(n)$
5. Reflexivity - any function is its own asymptotic bound
6. Transitivity - if $f(n) = O(g(n)) \wedge g(n) = O(h(n))$, $f(n) = O(h(n))$
7. Symmetry - if $f(n) = \Theta(g(n))$, $g(n) = \Theta(f(n))$
8. Transpose Symmetry - if $f(n) = O(g(n))$, $g(n) = \Omega(f(n))$
9. If $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$

**Note:** 7 is only for Theta, 8 is only for O and Omega. The rest can be generalized.

### Properties of Logarithms / Exponents

1. $a = b^{log_b a}$
2. $log_c ab = log_c a + log_c b$
3. $log_b a^n = nlog_b a$
4. $log_b a = \frac{log_c a}{log_c b}$
5. $e^x \geq 1 + x$

### Summations

Sum of Infinite Geometric Series : $\frac{k}{1-r}$
Sum of Arithmetic Series: $\frac{n}{2}[2a + (n-1)d]$

### Hierarchy

1
$loglogn$
$logn$
$(logn)^c$ where $c > 1$
$n^c$ where $(0 < c < 1)$
$n$
$nlogn$
$nlogn = nlog(n!)$
$n^2$
$n^c$
$c^n$ where $(c > 1)$
$n!$

## Searching

### Binary

Runtime = O(log(n/m)) for arrays where m = no. of copies of key.

**Invariants (Class Implementation)**:

1. Array remains sorted
2. Left Pointer ≤ Right Pointer
3. Key can be found between Left and Right Pointer

**Postcondition (Class Implementation)**: Left Pointer will point to key if key is in array

### QuickSelect (find k-smallest)

QuickSelect randomly identifies a pivot, partitions the array around it, and then either recurses on the left, right or returns the pivot based on its index. Paranoid QuickSelect is expected to be $T(n) = O(n) + T(9/10n)$ which is $O(n)$, while deterministic QuickSelect is $O(n^2)$ for bad pivots

## Sorting

All sorting algorithms in class are O(1) space except mergeSort which is O(n) additional space

| sort | best | average | worst | stable? |
|---|---|---|---|---|
| bubble | $O(n)$ | $O(n^2)$ | $O(n^2)$ | ✓ |
| selection | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | × |
| insertion | $O(n)$ | $O(n^2)$ | $O(n^2)$ | ✓ |
| merge | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | ✓ |
| heap | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | × |
| quick | $O(n)$ | $O(n \log n)$ | $O(nlogn)$ | × |

sorting invariants

| sort | **invariant** (after $k$ iterations) |
|---|---|
| bubble | largest $k$ elements are sorted |
| selection | smallest $k$ elements are sorted |
| insertion | first $k$ slots are sorted, rest untouched |
| merge | subarray(s) is sorted |
| quick | pivot is in the right position |
| heap | smallest $n-k$ elements form a max/min heap |

**Notes:**

1. $O(n)$ Best case for BS is only for fully-sorted array.
   For partially-sorted array, insertion is still $O(n)$ but BS degrades to $O(n^2)$
2. Deterministic vs Randomized is important for searching and sorting. Deterministic algorithms' runtime should be treated as worst case while randomized runtime depends on what is expected probabilistically
3. $O(nlogn)$ for all QS cases is only for 3-way, randomized. Bad versions of QS listed below
4. Additional invariant for BubbleSort: Each element moves at most k positions left after k iterations
5. Additional invariant for SelectionSort: Initial values at the start swapped into positions where smallest k values came from

### QuickSort Addendum

**Partitioning Variants**:

1. 2way : 2 unsorted regions. $O(n^2)$ for array of all duplicates
2. 3way: 2 unsorted regions and 1 equal region. $O(n)$ for duplicates array
3. First / Middle / Last as p: $O(n^2)$ worst case
4. Median (of all elements) as p: $O(nlogn)$
5. Median (of first, middle, last) as p: $O(n^2)$ worst case
6. Random as p: Paranoid QS expects $T(n) = T(9n/10) + T(n/10) + 2O(n) = O(nlogn)$
7. Stable Partioning: $O(n)$ additional space
8. K pivots: $O(klogk) + O(nlogk) = O(nlogk)$ partition runtime as long as $n \geq k$
9. K distinct elems where k < n: $O(nlogk)$
10. Lomuto Partition: Worst case runtime of QuickSort will degrade to $O(n^3)$

**Note on QuickSelect + QuickSort** : Finding k-smallest elements of array is $O(n + klogk)$ if use QuickSelect to find kth-smallest ($O(n)$), then QuickSort to sort Arr[1...k] ($O(klogk)$)

## Trees

### BSTs

A tree where everything smaller / bigger than a node is arranged to its left / right respectively. The height of a tree is the longest path from the lowest leaf to the root excluding the root itself

**Modification Operations**

1. Search, insert, delete = $O(h)$
2. Construction = $O(nh)$

**Query Operations**

1. searchMin = $O(h)$, recurse left
2. searchMax = $O(h)$, recurse right
3. successor = $O(h)$ - searchMin(right) or traverse upwards to first parent
4. traversals = $O(n)$

**Note** : The efficiency of operations on a tree depend on its shape. And its shape depends on how its nodes were inserted.

## AVL Trees

A dynamically balanced BST whose nodes are augmented with their height. A node $u$ of an AVL-Tree is height-balanced $\iff |u.left.height - u.right.height| \leq 1$. Also, $Height(u) = Max(Height(u.left), Height(u.right)) + 1$

**Characteristics**

1. Max no. of nodes in AVL-Tree = $2^{h+1} - 1$
2. Min no. of nodes in AVL-Tree = $2^{h/2}$
3. Max height of AVL-Tree = $h \leq 2log(n)$
4. "Base Cases": Height(leaf) = 0, Height(null) = -1

**Note:** The maximum height is found through solving $N(h) = N(h-1) + N(h-2) + 1$ which represents the worst-case example of an AVL-Tree

**Operations**

1. Search, insert, delete = $O(logn)$
2. Order statistics ops takes $O(n)$
3. Construction runtime = $O(nlogn)$
4. Space consumption = $O(n)$
5. Split and merge = $O(logn)$
6. Search, insert, delete for a string of length M = $O(mlogn)$

**Rebalancing Motivation**

1. Rebalancing is required when $|Height(left) - Height(right)| = 2$
2. $O(1)$ rotations after insertion (max 2), and $O(logn)$ rotations after deletion
3. $\Theta(n)$ nodes' height change after insertion

**Rebalancing Permutations**

1. Left-Left: Left subtree heavier, and left-heavy. Right-rotate node
2. Left-Right: Left subtree heavier, and right-heavy. Left-rot subtree, then right-rot node
3. Right-Right: Right subtree heavier, and right-heavy. Left-rotate node
4. Right-Left: Right subtree heavier, and left-heavy. Right-rot subtree, then left-rot node

# Augmented Trees

1. Choose underlying data structure
2. Determine additional info needed
3. Maintain info as structure is modified
4. Develop new operations using new info

**Note:** If the tree is augmented from and maintained as an AVL, it can surely do what an AVL can do in the same time

## DOS Trees

An AVL Tree whose nodes are augmented with weight

**Characteristics**

1. Rank(node) = number of elements ordered before it, including itself
2. Weight(node) = W(node.left) + W(node.right) + 1 and W(leaf) = 1
3. Weight-balanced $\iff node.left/right.weight < 2/3 * node.weight$
4. Update weights on rotation = $O(1)$

**Operations**

1. Select, rank = $O(logn)$
2. Select counts up while rank counts down

## Interval Trees

An AVL Tree whose nodes are augmented with the maximum endpoint of that subtree

**Search(key)**

1. If value is in root interval, return
2. If value > max(root.left), recurse right
3. Else, recurse left (go left only when can't go right)

**Characteristics**

1. Invariant: The search interval for a left-traversal at a node includes the maximum value in the subtree rooted at the node

**Operations**

1. Search (single interval) = $O(logn)$
2. Search (all intervals) = $O(klogn)$ for k overlapping intervals

**Note:** Search ops work because of the idea that the initial left-or-right recursion is made based on max property, i.e. if it fails to find the interval in the left subtree, it surely cannot find it in the right, and vice versa

## Orthogonal Range Finding

A 1D-range Tree is exactly an AVL-Tree. However, From 2 onwards the operation cost differs.

**Query(low,high)**

1. v = findSplit(); find key between low and high
2. if low $\leq$ v.key, all-leaf-traversal(v.right) and leftTraversal(v.left, low, high);
   else leftTraversal (v.right, low, high)
3. if high $\geq$ v.key, all-leaf-traversal(v.left) and rightTraversal(v.right, low, high);
   else rightTraversal (v.left, low, high)

**Operations- dth dimension range tree**

1. Query cost: $O(log^d n + k)$ where k is the number of points found
2. Left and right traversals: $O(k)$
3. Construction and space cost: $O(nlog^{d-1}n)$
4. Insert, delete operations not supported on dimension $\geq 2$ because rotations too expensive ($O(n)$)

**Notables - 2D Range Tree**

1. Query - $O(logn)$ for split node, $O(logn)$ recursing steps, $O(logn)$ y-tree searches of cost $O(logn)$, $O(k)$ enumerating output
2. Space - $O(nlogn)$ : Each pt appears in at most 1 y-tree per level, and $O(logn)$ levels so each node apprs in at most $O(logn)$ y-trees. Rest of x-tree takes $O(n)$ space.

**Note**: 2D-Range Trees implement a search on a y-tree instead of all-leaf. The y-tree is a 1D-Range tree sorted by its y-coordinate instead of x.

## KD-Trees

AVL Tree constructed through randomized QuickSort, with each level alternating with the axes of the space that the tree is partitioning

**Operations**

1. Construction takes $O(nlogn)$
2. findMin and findMax takes $O(\sqrt{n})$ only if tree is perfectly-balanced

Note: $O(nlogn)$ construction is achieved by doing by $O(nlogn)$ randomized quicksort on the array of points and filling up the tree with the sorted points along the recursion.

## Prefix Trees

Not a binary tree.

**Operations**

1. Search, insert, delete takes $O(M)$ where M is length of string in question
2. Space complexity is O(size of text * overheads) where size of text is sum of length of all strings, and overheads are the end-flags for each string, and other nodes that store information for each char node

Note: Overheads are important to expanding the operations that can be done on tries.

## Heaps

Not a BST.

**Operations and Characteristics - Max Heap**

1. Height(heap) $\leq$ floor($logn$)
2. Insert, extractMax, increaseKey, decreaseKey, delete: $O(logn)$
3. Included operations: bubbleDown, bubbleUp: $O(logn)$
4. Complete binary tree with the nodes in lowest level flushed to the left

**heapSort motivation**

**Idea**

1. Build a complete binary tree from the original array
2. Heapify recursively. Invariant: every subtree in heap is a valid heap
3. Perform n extractMax operations and reorder the unsorted array by swapping the max elem to the last

**Note:** Always populate og array by doing level-order traversal of heap

**Specs**

1. Runtime - Worst / Average / Best : $O(nlogn)$
2. Invariant - Properties of Max Heap are always preserved
3. Unstable
4. Space - $O(n)$

**Notables**

1. Invert min-heap to max - $O(n)$
2. Find k smallest elems using min heap: $O(nlogn + nlogk)$
3. Find k smallest using max-heap: $O(k + nlogk)$

# Probability Theory

1. Linearity of Expectation - $E[A + B] = E[A] = E[B]$
2. For random variables, expectation = probability
3. If an event occurs with probability p, the number of iterations needed for this event to occur is $\frac{1}{p}$
4. Probability of an item remaining in its original position among n items after n! permutations is $\frac{1}{n}$
5. Expected Value = Sum of products of each event with its probability

# General Tips for Runtime Analysis

1. Recursion (top-down): Height of call stack * cost of single call (where height = no. of calls to get to base case)
2. Iteration (bottom-up): No. of iterations * cost of n-dependent operations per iteration
3. Identify dominant term (i.e most expensive part of algorithm)
4. Use recurrence trees to solve recurrence relations easier