CS2040S AY22/23 SEM 2 MIDTERMS CHEATSHEET

Orders of Growth:

O	T(n) = O(f(n)) if $\exists\, c, n_0 > 0$ s.t $\forall\, n > n_0, T(n) \le cf(n)$ T(n) = O(f(n)) if T grows no faster than f
Ω	T(n) = O(f(n)) if $\exists\, c, n_0 > 0$ s.t $\forall\, n > n_0, T(n) \geqslant cf(n)$ T(n) = Ω(f(n)) if T grows no slower than f
Θ	T(n) = Θ(f(n)) ↔ T(n) = O(f(n)) & T(n) = Ω(f(n)) T(n) = Θ(f(n)) if T grows at same rate as f

Order of Size:

$k < \log\log(n) < \log(n) < \sqrt{n} < \log^2(n) < n < n\log(n) < n^2 < n^3 < n^3\log(n) < n^4 < 2^n < 2^{2n} < n!$

$\log_a(n) < n^a < a^n < n! < n^n$

In general:

Loops	Cost = (# iterations) * (Max cost of 1 iteration)
Sequential	Cost = (cost of 1st) + (cost of 2nd)+ … + (cost of nth)
If/Else	Cost = Max(Cost of 1st, Cost of 2nd) <= (Cost of 1st) + (Cost of 2nd)
Recursion	**Master Theorem:** $a > 0, b > 1, d \ge 0$ $T(n) = aT\left(\frac{n}{b}\right) + O(n^d);$ $T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$

Common Recurrence Relations:

O(n)	
T(n) = T(n-1) + O(1) → T(n) = O(n)	T(n) = T(n/2) + O(n) → T(n) = O(n)
T(n) = 2T(n/2) + O(1) → T(n) = O(n)	T(n) = O(n/2) + T(n/2) → T(n) = O(n)
O(log n)	**O(n $\log^2$ n)**
T(n) = T(n/2) + O(1) → T(n) = O(log n)	T(n) = 2T(n/2) + O(n log n) → T(n) = O(n $\log^2$ n)
O(n log n)	
T(n) = T(n-1) + O(log n) → T(n) = O(n log n)	T(n) = 2T(n/2) + O(n) → T(n) = O(n log n)
O(n^2)	**O(n^{k+1})**
T(n) = T(n-1) + O(n) → T(n) = O(n^2)	T(n) = T(n-1) +O(n^k) → T(n) = O(n^{k+1})
O(2^n)	
T(n) = 2T(n-1) + O(1) → T(n) = O(2^n)	T(n) = T(n-1) + T(n-2) +O(1) → T(n) = O(2^n)

Important:

- $\sqrt{n}\log n = O(n)$ $\qquad T(n) = 2T\left(\frac{n}{4}\right) + O(1) = O(\sqrt{n})$
- $O(2^{2n}) = O((2^n)^2) \neq O(2^n)$ **Exp power is significant
- $\log(n!) = \Theta(n \log n)$ (Sterling's Approximation)
  $$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right) \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$
- GP: $n + \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^n} = 2n = O(n)$
  $$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$
  $$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \text{ when } |x| < 1$$
- AP: $T(n-1) + T(n-2) + \cdots + T(1) = 2T(n-1)$
  $$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n = \frac{1}{2}n(n+1) = \Theta(n^2)$$
- Harmonic Series
  $$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n}\frac{1}{k} = \ln(n) + O(1)$$
- $f(n) = \frac{1}{n} \to f(n) = O(1)$

** Note for strings: Concatenation takes n time;

public String s(int n) {
String s = “”;
for (int i = 0; i < n; i++) {
s += “?”; }
return s; }
	public String s(int n) {
	StringBuilder sb = “”;
	for (int i = 0; i < n; i++) {
	sb.append(“?”); }
	return sb.toString(); }

Strings are immutable; Takes O(# char) time to create new string → O(n^2)

* Use Stringbuilder function for O(n)

Space Complexity:

- Max space used during computation
- $\Theta(f(n))\ time \to \Theta(f(n))\ space$

Logarithmic Rules: (Example: $2^{4\log n} = (2^{\log n})^4 = n^4$)

$a^{mn} = (a^m)^n$	$a^{\log_a b} = b$	$\log_b(a) = \frac{\log_x(a)}{\log_x(b)}$
$log_b\left(\frac{1}{a}\right) = -log_b(a)$	$log_b(a)^n = nlog_b(a)$	$log_b(a) = \frac{1}{log_a b}$

PeakFinding: (Key idea: Binary Search)

log n

```
FindPeak(A, n)
  if A[n/2+1] > A[n/2] // right
    FindPeak (A[n/2+1..n], n/2)
  else if A[n/2–1] > A[n/2] // left
    FindPeak (A[1..n/2-1], n/2)
  else A[n/2] is a peak; return n/2
```

✅ Only 1 peak.
❌ multiple/ no peaks

n

```
FindPeak(A, n)          //STEEP PEAKS
  if A[n/2-1] == A[n/2] == A[n/2+1] // L&R
    FindPeak (A[n/2+1..n], n/2)
    FindPeak (A[1..n/2-1], n/2)
  else if A[n/2+1] > A[n/2] // R
    FindPeak (A[n/2+1..n], n/2)
  else if A[n/2–1] > A[n/2] // L
    FindPeak (A[1..n/2-1], n/2)
  else A[n/2] is a peak; return n/2
```

✅ Homogeneous arrays.
❌ multiple/ no peaks

Sorting

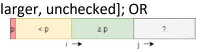
Bubble Sort	
Algo	Swaps adjacent elements if they are not in order.
Invariant	After i^{th} iteration, last i elems are sorted
Stability	Stable. Since swapping is only done between adjacent elements.
Time	**O(n^2);** Worst Case Input: Reversed Array
Extra Space	O(1); in-place
Insertion Sort	
Algo	Maintain a sorted prefix (begin w 1^{st} element). For every iteration, insert the new element into the correct position in the sorted prefix.
Invariant	After i^{th} iteration, 1^{st} i elems are relatively sorted
Stability	Stable. Since swapping is only done between adjacent elements.
Time	Best: O(n); Average: O(n^2); Worst: O(n^2) Worst Case Input: Reversed Array Best Case: Already/Almost sorted
Extra Space	O(1); in-place
Selection Sort	
Algo	Repeatedly find the smallest element, and swap it to the front.
Invariant	After i^{th} iteration, 1^{st} i elements are sorted
Stability	No. Since swapping is not adjacent
Time	**O(n^2);** Worst Case Input: All inputs
Extra Space	O(1); in-place
Merge Sort	
Algo	Divide-and-conquer. Split the array in ½. MergeSort each half. Merge the two sorted halves together.
Invariant	For every call to merge, both its arguments are always sorted.
Stability	Stable
Time	**O(n log n)**
Extra Space	**O(n)**
Quick Sort	
Algo	Divide & Conquer. Random Pivot → High prob O(nlogn)
Invariant	After performing the partition, all elements smaller than the pivot occur before the pivot, and all elements larger than the pivot occur after the pivot
Stability	No. Swaps not adjacent
Time	Best: O(nlogn); Average: O(nlogn); Worst: O(n^2) Worst Case Input: Reversed Array
Extra Space	O(1); in-place

- For insertionSort, end part of array will not be touched unless swapped.
- SelectionSort uses the lest no. of swaps, but has a lot of comparisons

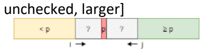
**For QuickSort, More pivot =/= better time complexity
But IRL, hardware are optimized for 2-3 pivots
Best pivot → Random pivot; Median

2Way Partitioning: (Use this if no duplicated)

- Invariant = [smaller, larger, unchecked]; OR



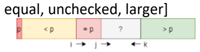
- Invariant = [smaller, unchecked, larger]



- Array of same elements → O(n^2)

3Way Partitioning: (Use this if there are duplicates)

- Invariant = [smaller, equal, unchecked, larger]



- Array of same elements → O(n)

Making QuickSort Stable:

- Use an auxiliary array. Extra space → Stable

Note:

If length < 1024 → InsertionSort is faster than MergeSort

Quick Select: (Θ(n), O(n^2))

Find the k-th smallest/largest element using QuickSelect

- Select a pivot & partition the array
- If pivot == k^{th} smallest element, return
- Else recurse into < pivot partition, if pivot's position is > k
- Or recurse into > pivot partition, if pivot's position is < k
  - O(n) time to find k^{th} smallest element
  - Invariant similar to QuickSort

Counting Sort

1. Find the maximum value in the input array and create a new array of that size, initialized to all zeros.
2. Iterate through the input array, incrementing the count of the value at the corresponding index in the new array.
3. Iterate through the new array, constructing a sorted output array by adding each value a number of times equal to its count.

- Works by counting the no. of occurrences of each element in the input array and using that information to construct a sorted output array.
- Time Complexity → O(n + k)
  - n → # elements in input array
  - k → range of values in input array
- Counting Sort is v efficient when k <<<< n
- Not in-place, requires auxiliary array → Space: O(k + n)

Radix Sort

1. Find the max element in the input array & determine the # of digits in its base-k representation (k is the radix of the sort)
2. For each digit position i, starting from the least significant digit, sort the input array by that digit position using a stable sorting algorithm such as counting sort.
3. After the final iteration → sorted

- Works by iterating through each digit of the elements to be sorted, from the least significant digit to the most significant digit, and sorting the elements based on the value of the current digit.
- Time Complexity → O(d * (n + k))
  - d → # digits in the max element
  - k → radix of the sort
  - n → # elements in array
- Not in-place, requires auxiliary array → Space: O(k + n)

Binary Search

- Time: O(log n); Space: O(1)
- Pre-Condition: Array A is sorted; Post-Condition: A[begin] = key
- Loop Invariant: A[begin] ≤ key ≤ A[end]

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```

- Use when input is sorted/ can manipulate input to be sorted
- By checking a value, eliminate all that are smaller/larger
- To find local minimums or maximums

Trees Height of Tree = max(left.height, right.height) + 1

Binary Search Trees (BST):

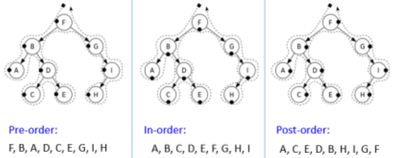
A BST is a binary tree what satisfies the BST invariant: left subtree has smaller elements & right subtree has larger elements

Function	Average	Worst
Insert, Delete, Search	O(h) = O(log n)	O(n)
Successor/Pred	O(h) = O(log n)	O(n)
findMax/ findMin	O(h) = O(log n)	O(n)
In-order Traversal	O(n)	

Balanced Tree: Insert/Delete/Search → O(log n)

Normal BST: Might degrade into linear tree, Worst O(n).

Traversal for both is still O(n). As you need to visit every node



*The In-Order of a BST can give us the whole tree structure.

Balanced Binary Search Tree (BBSTs)

- Balanced if h = log n OR O(h)

Function	Average	Worst
Insertion, Delete, Search	Θ(log n)	O(log n)

AVL Tree

- Allows for O(log n) insert, delete & search
- Balanced Factor (BF): H(node.R) – H(node.L)
- Height Balanced: |left.height – right.height| ⩽ 1
  - Right rotate → root of the subtree moves R
  - Left rotate → root of the subtree moves L
- **Left rotation requires Right child (vice versa)
- A height-balanced tree with height h has at least n > $2^{h/2}$ nodes; (at most h < 2log(n))
- IF BF ∉ { -1, 0, +1 } → rebalance tree
  - **LeftLeft → Rotate Right**
  - **LeftRight → Rotate Left then Right**
  - **RightRight → Rotate Left**
  - **RightLeft → Rotate Right then Left**
- Space Complexity: O(LN),

Storing String in AVL tree:

- AVL tree are selfbalancing → O(log n)
- Comparison between strings → O(L)
- ∴ Insert/Delete/Search → O(L log n)

Tree Rotation: (Tree rotations can create every possible tree shape)

After Insertion	Max 2 rotations to balance
After Deletion	O(log n) rotations to balance

Tries:

- 1 path down the Trie can represent multiple words (depending on flag)

Search (String length L)	O(L)
Insert (String length L)	O(L)
Space	O(size of text) * OverHead

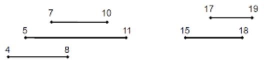
Tries Trade Offs:

Time	Tries tend to be faster: O(L) vs Tree: O(hL) • Does not depend on size of text • Does not depend on no. of string
Space	Tries tend to use more space • ACSII chat set: 256 (OverHead) • A lot of children; but wasted space

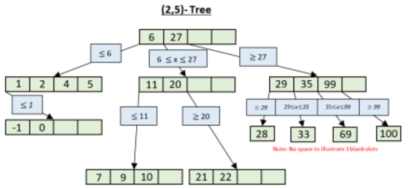
(a, b) -Trees

Node Type	# Keys (Slots) Min	# Keys (Slots) Max	# Children Min	# Children Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

- All leaf nodes are at the same level/depth.
- Non-leaf nodes must have 1 more child than keys
- Keys in each node are stored in increasing order.
- NOTE: BST = (1, 2)- Tree



- Order k tree → has k-1 keys
- (a,b)-trees → root go upwards, (not leaf go down)
  - Hence leave nodes alw same level



When #Child > #Keys, choose median key & use it to split keylist into 2 ½ s

- Left half goes into a new node
- Move median to parent (i.e shift median to a key in parent node)
- Link Node to parent

QNS	Why must split "offer" one key to parent?
ANS	After splitting, the parent will have one more child than before, therefore it must also have one more key. Taking that key from the node to be split is convenient.

- If parent is full, and child to split, we would need to split upwards until grandparents, to resolve the over-capacities.
- The root has min 1 key instead of a-1, to make space to permit split operation at the root

Time Complexities:

Search, Insert, Delete	O(log n) = O(h)
Merge (2 nodes)	O(b)
Share (2 nodes)	O(b)
Split (1 node)	O(b)

- Max height → $O(\log_a n) + 1$; Min Height → $O(\log_b n)$

Probability Theory:

- If an event occurs with probability p, the expected number of iterations needed for this event to occur is 1/p .
- For random variables: expectation is always equal to the probability
- linearity of expectation: E[A + B] = E[A] + E[B]

Uniform Random Permutation:

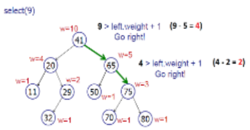
- if we have n elements, there are n! possible orderings, and each ordering has a probability of 1/n! of being chosen as the random permutation.
- E.g Fisher-Yates shuffle Algo: Start with an ordered sequence of elements, and iteratively swap each elem with another randomly chosen element from the remaining sequence. → O(n)

1. Start with an ordered sequence of n elements.
2. For i from n-1 down to 1, do the following:
   - Pick a random int j from 0 to i inclusive.
   - Swap the i-th elem with the j-th elem.

$\frac{\#Outcome}{\#Permutations} \in \mathbb{N}$ P(item remaining in initial pos) = 1/n

Dynamic Order Statistics on Balanced Tree:

Store size of sub-tree in every node



Define weight:

- w(leaf) = 1
- w(v) = w(v.left) + w(v.right) + 1
- "rank in subtree" = left.weight + 1

Let n in select(n) be the rank of node

Let L be left.weight +1 (i.e weight of left child node)

- If n > L → Go right, then select(n-L) on child node
- If n < L →Go left, then select(n) on child node
- Rank(v) → Computes the rank of a node, v.
- Select(k) → Finds the node with rank k.

Key Invariant:

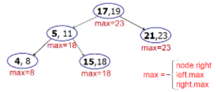
- After every iteration, the rank is equal to the its rank in the subtree rooted at node.
- At the end of the program, the rank would be equal to the rank of the subtree rooted at the root (correctness)

- In every node, the rank is either the same (if no nodes have come before it), or the rank is increased by the number of nodes that came before it)

Note: Update Weights & Rotate to maintain D.S when insert/delete.

Interval Queries/Searching

Interval Trees:



- Each node is an interval.
- Tree is sorted by the left endpoint.
- Augment: Maximum endpoints in subtree
- Stores the max endpoint (right) in the subtree at the node too
- Rotate tree & update intervals to maintain D.S

Searching: (If there are k leaves → total nodes in tree ≤ 2k)

- interval-search(x): find interval containing x. → O(log n) time
- **If search goes right**: then no interval in left subtree. → Either search finds key in right subtree or it is not in the tree.
- **If search goes left**: if there is no interval in left subtree, then there is no interval in right subtree either. →Either search finds key in left subtree or it is not in the tree.
- **Conclusion**: search finds an overlapping interval, if it exists.
- If search goes left & no overlap → key < every interval in right subtree.

Listing all intervals that overlap a point: **(All-Overlaps Algorithm)**

- **Repeat** until no more intervals:
  - Search for interval
  - Add to list
  - Delete interval
- **Repeat** for all intervals on list:
  - Add interval back to tree.
- Running Time: O(k log n) (lecture eg); irl, best time: O(k + log n)

Orthogonal Range Searching

1D Range Queries

Strategy:

1. Use a binary search tree. (Need to maintain BST invariant)
2. Store all points in the leaves of the tree. (Internal nodes store only copies; i.e internal nodes are guide posts.)
3. Each internal node v stores the MAX of any leaf in the left sub-tree.
   - Leaf nodes → Data nodes; Internal Nodes → Guide Posts

- Find "split" node → v = FindSplit(low, high); O(log n)
- Do left traversal → LeftTraversal(v, low, high); O(2k) = O(k)
  At every step, we either:
  1. Output all right sub-tree and recurse left. OR
  2. Recurse right.
- Do right traversal → RightTraversal(v, low, high); O(2k) = O(k)
  At every step, we either:
  1. Output all left sub-tree and recurse right. OR
  2. Recurse left.

Invariant:

The search interval for a left-traversal at node v includes the maximum item in the subtree rooted at v

Note:

- Binary Search is good for finding 1 item in the tree.
- However, now that we want to find a range of items in a tree, we have to augment the tree, such that we can find range with an efficient algorithm

2D Range Tree: (Use Augmented Trees)

- Build an x-tree using only x-coordinates.
- Create a 1d-range-tree on the x-coords.
- For every node in the x-tree, build a y-tree out of nodes in subtree using only y-coordinates. (I.e Range-trees inside range trees)

d-dimensional Range Queries:

Idea:

- Store d–1 dimensional range-tree in each node of a 1D range-tree.
- Construct the d–1-dimensional range-tree recursively.

Query time Q_d (not including point reporting) given by the recurrence:

$Q_d(n) = O(\log n) + O(\log n) \cdot Q_{d-1}(n)$

Query cost:	$O(\log^d n + k)$
buildTree cost:	$O(n \log^{d-1} n)$
Space:	$O(n \log^{d-1} n)$

Priority Queue: Can be either min/max

Maintain a set of prioritized objects:

– insert: add a new object with a specified priority

– extractMax: remove and return the object with max valued priority

Sorted Array	Unsorted Array
insert: O(n) • Find insertion location in array. • Move everything over. **extractMax: O(1)** • Return largest element in array	**insert: O(1)** • Add object to end of list **extractMax: O(n)** • Search for largest element in array. • Remove and move everything over.
AVL Tree (indexed by priority)	
insert: O(log n) • Insert object in tree **extractMax: O(log n)** • Find maximum item. • Delete it from tree.	Max Height of heap = floor(log n) = O(log n)

Heap (Binary Heap/ Max Heap)

- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.

2 properties of a Heap:

1. **Heap Ordering**
   - Priority[parent] ≥ priority[child]
2. **Complete Binary Tree**
   - Every level is full, except possibly the last.
   - All nodes are as far left as possible.

Insert	Steps: (e.g insert(25)) 1. Add a new leaf with priority 25 2. BubbleUp
increaseKey	Just bubble up
decreaseKey	1. Update the priority 2. BubbleDown ALWAYS bubble down the LEFT side
delete	1. Swap(n, last()); 2. Remove(last()); 3. BubbleDown
extractMax	1. Node v = root 2. Delete(root)

Heap vs. AVL Tree

- Same asymptotic cost for operations
- Faster real cost (no constant factors!)
- Simpler: no rotations
- Slightly better concurrency

How to store a Tree (Heap) in an Array?

- Map each node in the complete binary tree into a slot in an array
- Level-Order (BFS)

What about inserting?

- Insert at the next available index.
- Note: fill the leaf nodes from left :D
- Bubble up priority, s.t the heap invariant is preserved

Check for Child node	Check for Parent node
left(x) = 2x + 1 right(x) = 2x + 2	parent(x) = floor((x-1) / 2)

* NOTE, we cannot store AVL-trees as an array:

- Many "holes" in array → waste space
- Need to rotate to maintain invariant, but it is not O(1) → costly
- Unless you have a complete balanced tree, storing trees in an array is NOT efficient

Advantage of heap:

- Heaps can be stored in an array instead of nodes
- Everything will be stored in the same memory area → cache locality will be faster

---

HeapSort	
Algo	Find the max element and place it at the end. Repeat the same process for the remaining elements.
Invariant	Step 1 Heapify → Maintain Heap Invariant Step 2 ExtractMax → Maintain Heap Invariant
Stability	Not Stable. Since swapping is not between adjacent elements.
Time	Always complete in O(n log n)
Extra Space	O(1); in-place, only uses O(n) space

How to perform HeapSort?

1. Build a heap (from unsorted list)
2. HeapSort (By finding the max node over and over again)

Cost of Building a Heap → O(n)

- Initial: Start with a Complete Tree (recursion)
- Base Case: Each leaf is a Heap
- Recurse: Left + Right are Heaps

```
// int[] A = array of unsorted integers
  for (int i=(n-1); i>=0; i--) {
  bubbleDown(i, A); // O(log n) = O(height)
}
```

Note: cost(bubbleDown) = height

- More than n/2 nodes are leaves (h = 0)
- Most nodes have small height

Heapify → O(n) ; HeapSort → O(n log n)

More Interval Trees:

- Process intervals into a tree, sorted by their lower bound (this.low)
- Each node also stores this.max = max(left.max, right.max, this.high). Recursively bubbled up from bottom to top.

FindInterval: Always go to the left subtree if possible

FindAllIntervals: Find & remove all intervals, then add all intervals back

FindInterval:
```
FindInterval(x) // or interval-search in lecture
  c = root;
  while (c != null and x is not in c.interval) do
    if (c.left == null) then c = c.right;
    else if (x > c.left.max) then c = c.right;
    else c = c.left;
  return c.interval;
```

Stacks & Queue:

Stacks	Queue
Push, Pop, Peek → O(1) • Diff permutations	Enqueue, Dequeue, Peek → O(1) • Process items in a sequence

Pancake Sort:

- Sorts an array by repeatedly flipping adjacent elements
- Number of flips is O(n);
- Time Complexity: $O(n^2)$, (linear searching will be required per 2 flips)
- Invariant: After each iteration, the subarray from index 0 to the current iteration index is sorted in non-descending order, and the remaining unsorted elements are in the right half of the array.

Tree Rotation Diagram