CS2040S Midterm Cheatsheet — YashyPola AY22/23 S2

Orders of Growth

The order of growth of an algorithm is the measure of its growth in time and space consumption as its input size approaches an asymptotically large number. 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 $\forall n > n_0$, $T(n) \leq cF(n)$. Informally, T(n) is bounded by F(n) if and only if, beyond some input size n_0, T(n) is always less than F(n) by some constant c.
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. $T(n) = 2T(n/2) + O(nlogn) = O(n(logn)^2)$

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

Geometric Series : $\sum_{k=1}^{n} k = 1 + 2 + 3 + ... + n = \frac{1}{2}n(n+1)$
Arithmetic Series: $\sum_{k=1}^{n} x^k = 1 + x + x^2 + ... + x^n = \frac{x^{n+1}-1}{x-1}$

Hierarchy

$1 < logn < \sqrt{n} < n < nlogn < n^2 < n^3 < k^n$
In general, $log_a n < n^a < a^n < n! < n^n$

Searching

Key things to consider:
1. Is the data structure sorted in anyway?
2. Is it possible to reduce / divide the structure and if so how?
3. How does the ordering of the structure influence search time?

Linear

The naive approach to searching. Usually used on unsorted arrays. Runtime = O(n).

Binary

Divide-and-conquer approach to searching. Runtime = O(logn) for Arrays

Preconditions:
1. Array is sorted
2. Possible to find key by starting at middle, then recursing on left or right

Invariants:
1. Array remains sorted
2. Left Pointer $\leq$ Right Pointer
3. Key can be found between Left and Right Pointer

Postcondition (class implementation): Left Pointer will point to key if key is inside

QuickSelect

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. Runtime is $O(n)$ because it does not need to sort all its partitions. Worst case is $O(n^2)$ if the chosen pivot is the biggest / smallest.

Sorting

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

Insertion

```
int end = a.length;
for (int i = 1; i < end; i++)
int j = i;
 while (j > 0 && a[j] < a[j-1]))
  swap(a, j, j-1);
  j--;
```

Good Inputs
The number of inversions made by insertion sort = number of adjacent unsorted elements
1. Partially sorted arrays: each entry is close to final position, a small array appended to a large sorted array, an array with only a few entries that are not in place
2. Small arrays of only 5-15 elements

Characteristics
1. Runtime - Worst / Average: $O(n^2)$, Best: $O(n)$
2. Invariant: First i elements sorted after i iterations, rest n - i elements untouched
3. Stable

Optimization: shellSort - Split array into h independent subsequences and use insertion sort to sort each of them. Allows for elements to swap over larger distances instead of only adjacently

Selection

```
int end = a.length;
for (int i = 0; i < end; i++)
int min = i;
 for (int j = i+1; j < end; j++)
  if a[j] < a[min], min = j;
 swap(a, i, min)
```

Characteristics
1. Runtime - Worst / Average / Best: $O(n^2)$
2. Invariant: Smallest i elements sorted after i iterations
3. Unstable
4. Runtime is insensitive to input
5. Data movement is minimal

Bubble

```
boolean flag;
for int i = 0; i < end; i++
 flag = false;
 for int j = 0; j < end - i; j++
  if a[j] > a[j+1], swap(a, j, j+1), flag = true;
 if !flag, break
```

Characteristics
1. Runtime - Worst / Average: $O(n^2)$, Best: $O(n)$
2. Invariant - Largest i elements are sorted after i iterations and then untouched
3. Stable

Note: $O(n)$ best case is only for optimized bubbleSort. Vanilla bubble sort is always $O(n^2)$ and therefore worse than Insertion.

$O(n)$ **Optimization**: Use a boolean flag to check for swaps to terminate sort if no swaps were done. This ensures O(n) runtime for already sorted array

Merge

Characteristics
1. Runtime - Worst / Average / Best: $O(nlogn)$
2. Invariant - Subarrays are sorted when merging
3. Stable

Quick

Characteristics

1. Runtime - Worst: $O(n^2)$, Average / Best: $O(nlogn)$
In general, runtime of Qs is O(T(partition) * T(sort))
2. Invariant - Subarrays l / r of p are < / > than p, and p is sorted
3. Unstable [Because partition is unstable]

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: $O(n^2)$ for (reverse) sorted array
4. Median as p: $O(nlogn)$
5. Random as p: Paranoid QS gives us $T(n) = T(9/10n) + T(1/10n) + 2O(n) \approx O(n^{0.152})$
6. Stable Partioning: $O(n)$ additional space
7. K pivots: $O(klogk) + O(nlogk) = O(nlogk)$ partition runtime as long as $n \geq k$
8. Lomuto Partition: Worst case runtime of QuickSort will degrade to $O(n^3)$

Trees

Always be clear what the tree is sorted by.

BSTs

Characteristics

1. A BST is either empty, or a node pointing to 2 BSTs.
2. For a balanced tree, height $< 2log(n)$ and nodes $> 2^{h/2}$
3. For a full binary tree, $n = 2^k - 1$ where $k \in \mathbb{Z}^+$
4. The height of a tree is number of nodes in the longest path from leaf to root

Modify 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

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

Characteristics

1. Height-balanced
$\iff |node.left.height - node.right.height| \leq 1$
2. height, h(node) = max(h(left), h(right)) + 1.
h(leaf) = 0, h(null) = -1
3. Space complexity is $O(M * n)$ for n strings of length M
4. Search, insert, delete = $O(logn)$, for strings its $O(Mlogn)$ where M = string.length(), and construction = $O(nlogn)$
5. Split and merge take $O(logn)$ as well.
7. Rotation is required when $|h(left) - h(right)| \geq 2$
8. Max 2 rotations after insertion, and max $O(logn)$ rotations after deletion. Only need to rebalance the lowest unbalanced node after insertion.

Rebalancing

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

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. Rank, select = $O(logn)$

Balancing

1. Make sure tree is height-balanced to maintain AVL property
2. Update weights on rotation: O(1)

Interval Trees

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

Search(key)

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

Characteristics

1. Search (all intervals) = $O(klogn)$ for k overlapping intervals
2. Invariant: The search interval for a left-traversal at a node includes the maximum value in the subtree rooted at the node

Orthogonal Range Finding

An AVL Tree

Query(low,high)

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

Characteristics - dth dimension range tree

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

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

Augmented AVL Tree

Characteristics

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.

Characteristics

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.

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, and 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
4. Always populate og array by doing level-order traversal of heap

Characteristics

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