CS 477 Midterm Two Guide

February 20, 2026

Contents

0.0.1 Intro

This guide references this presentation which includes all the material on the upcoming test.

1 Slides Overview

1.1 Why Randomize Quicksort?

Ref. Slides: 14-17

Quicksort is pretty efficient when sorting an unsorted array, but it has less than ideal timing when handling a sorted array. To avoid worse timing for edge cases, the pivot can be chosen randomly. Normally, the pivot chosen in any given method is one of the extremes of the array. Picking the pivot for a sorted array will partition the array in an unbalanced way, resulting in slower times. Since Quicksort already is known to run at its best in $n \log(n)$ time when sorting an unsorted array, swapping the pivot with a random element before partitioning doesn't affect the timing in the case of an unsorted array and avoids bad partitions for sorted arrays.

1.2 Partitioning Using a Loop Invariant

Ref. Slides: 18-33

Partitioning the array by using a loop invariant; the pivot is the last element and the elements less than (or equal to) and greater than the loop invariant are placed into their correct positions.

When the partition is complete, the pivot can be placed between the two arrays and then Quicksort can get called yet again on the two new unsorted ranges. Since the ranges are guaranteed to be greater and less than the pivot that has been placed in between those ranges, the pivot will already be in the correct spot.

1.3 Expected Outcomes of Random Events

Ref. Slides: 35-39

When given a set of probabilities associated with outcomes (all examples of which have surrounded money associated with random games), just multiply the probability of an event with its associated 'value' and sum the results. For example, a game can be played with a 20 sided die. If the player rolls a 1, then they lose 10 points. If the player rolls a 20, then they gain 15 points. If the player rolls a 2-9, they lose 2 points. If they roll a 10-19, they gain 3 points. What is the expected point value from rolling the die?

$$\frac{1}{20} \times -10 + \frac{1}{20} \times 15 + \frac{8}{20} \times -2 + \frac{10}{20} \times 3 = 0.95 \text{ points}$$

1.4 Selection

Ref. Slide: 49

The General Selection problem is the one that aims to find the i^{th} smallest element in a set of n distinct numbers where there are $i-1$ other elements smaller than element i in the set. In simpler terms, what's the fastest way of finding the smallest value in an array? (The array can't have dupes). Instead of iterating through the array, an $n \log(n)$ sorting algorithm can be used to sort the array after which the i^{th} element can be retrieved from the result at constant time. $n \log(n)$ algorithms include Merge Sort and Quicksort.

Ref. Slides: 58-73

Of course, getting the i^{th} smallest element of an array is trivial if the array is sorted and then queried. This, however, isn't as efficient as finding the i^{th} element can get. Instead of spending computational power sorting the entire array, a Quicksort-esque method can be used while not calling the method unnecessarily. For example, after partitioning the array, the algorithm knows whether it wants to search for an element in the first or second half of the array. If it hasn't located the element is looking for by chance (i.e. it partitioned around the i^{th} element and can terminate, it simply has to recur on the half that it desires and not waste time recurring the other half of the array that won't be touched. On average, this will take $O(n)$, which is faster than $O(n \log(n))$.

1.5 Being Efficient While Finding Min and Max

Ref. Slides: 52-55

The min and max elements of an array can be found independently in $2n - 2$ time, but the process can be optimized to be solved in $\frac{3n}{2}$ by finding them both at the same time.

1. Keep track of min and max locally.
2. Set both the min and max to the first element when handling an odd array and set min and max to be their respective values among the first two elements in the case of an even array. The algorithm now has the same rule to work with in the case of both an odd and even array: the fact that there will always be an even number of elements left after step 2.
3. Check the next two elements in the array and only compare the max and min of the next two elements to the current max and min. Repeat this process for the rest of the pairs remaining in the array.

This makes it so that there are only 3 comparisons made for every 2 elements.

1.6 Sorting

1.6.1 Counting Sort

Ref. Slides: 83-89

Counting Sort is not an in-place algorithm. The final sorted array is a newly allocated array that stores the result of the sort as it is performed. The algorithm first creates an array and populates it with the number of times each value in the range of values there are. For example, if the input array is $A = [2, 5, 3, 0, 2, 3, 0, 3]$, then the counting array will be $C = [2, 0, 2, 3, 0, 1]$ (i.e. there are 2 zeroes, 0 ones, 2 twos, etc...). After the initial counts are tallied up, the array then begins at index 1 and adds $i-1$ to itself for all elements in C. This will result in an array that looks like $C = [2, 2, 3, 7, 7, 8]$. C now holds the indexes at which any given value can be safely placed. To get the final sorted array, the algorithm iterates through A and populates $C[A[i]]$ with $A[i]$. The value at $C[A[i]]$ then gets decreased - $C[A[i]] = C[A[i]] - 1$. Elements that aren't present in the original array A won't affect the final result and when the algorithm is done iterating through A and placing each element in its corresponding spot, the final array B is completely populated with A's sorted elements.

1.6.2 Radix Sort

Ref. Slides: 90-93

Radix Sort is a stable algorithm that sorts an array in passes. It is not an in-place algorithm because it must be copied into a new resulting array. For example, an array of numbers would get sorted by their least significant first and their most significant digit last.

1.6.3 Bucket Sort

Ref. Slides: 94-100

Bucket Sort is not an in-place algorithm. It allocates additional memory in buckets to store a given range of data. After sorting the data into the buckets, it sorts each bucket and then appends all of the buckets together.

1.7 Heaps

Ref. Slides: 109-137

Heaps are complete binary trees with two properties:

1. All levels except for the last must be full. Each level must be populated from left to right.
2. All child nodes must be less than or equal to their parent node.

Heaps can be stored as arrays as they never have gaps (as per their first rule).

1.7.1 Quick Info

1. Node Height: The number of edges on the longest path from a node to a leaf
2. Node depth: The length of a path from the node to the root of a tree
3. Tree height: The height of the root node, also $\lfloor \lg(n) \rfloor$

1.7.2 Heap Types

1. Max Heaps have their largest element at the root and all non-root nodes have the property `A[PARENT(i)]` $\geq$ `A[i]`
2. Min Heaps have their smallest element at the root and all non-root nodes have the property `A[PARENT(i)]` $\leq$ `A[i]`

1.7.3 A quick sidenote on Priority Queues

Priority Queues are just Max Heaps that support the following operations:

1. `INSERT(S, x)`: Inserts element x into set S
2. `EXTRACT-MAX(S)`: Removes and returns element of S with largest key
3. `MAXIMUM(S)`: Returns element of S with largest key
4. `INCREASE-KEY(S, x, k)`: Increases value of element x's key to k (assume k $\geq$ current key value at x)

1.7.4 `MAX-HEAPIFY`

`MAX-HEAPIFY` runs under a couple of assumptions:

1. There is only one violating node in the tree
2. The left and right subtrees of the violating node are both Max Heaps

Max Heapify exchanges the violating element with the larger child among its children (to maintain the Max Heap property), and then continues checking validity and doing so as long as the node isn't in a valid location.

1.7.5 `BUILD-MAX-HEAP`

To build a Max Heap from an unordered array, simply perform `MAX-HEAPIFY` on all non-leaf nodes.

1.7.6 `HEAP-MAXIMUM`

Returns the root.

1.7.7 `HEAP-EXTRACT-MAX`

Extracts and returns the root from the tree. Execution:

1. Exchange the root with the last element
2. Store the value of the last element (which was root)
3. Decrease the size of the heap by one
4. Call `MAX-HEAPIFY` on the new root
5. Return the cached value of the previous root

1.7.8 `HEAP-INCREASE-KEY`

Increases the value of a passed key to the newly passed value. The new value passed in the function must be greater than the old value.

1. Check if the new value is indeed greater than or equal to the old value
2. Set the node's value to the new value
3. Push the node up the tree as long as it's greater than its parent

1.7.9 `MAX-HEAP-INSERT`

1. Adds $-\infty$ to the end of the heap
2. Calls `HEAP-INCREASE-KEY` passing $-\infty$ and the value to insert

1.7.10 `MAX-HEAP` Funcs & Timing Summary

1. `MAX-HEAPIFY` $O(\lg(n))$
2. `BUILD-MAX-HEAP` $O(n)$
3. `HEAP-SORT` $O(n\lg(n))$ (not listed above)
4. `MAX-HEAP-INSERT` $O(\lg(n))$
5. `HEAP-EXTRACT-MAX` $O(\lg(n))$
6. `HEAP-INCREASE-KEY` $O(\lg(n))$
7. `HEAP-MAXIMUM` $O(1)$

1.8 Binary Search Trees

Ref. Slides: 138-140

Binary Search Trees are distinct because their left children always store a lesser value and their right children always store a greater value. Binary Search Trees are generally fine, but inserting consecutive elements into a Binary Search Tree may lead to a chain that's inefficient to search.

1.9 Red Black Trees

Ref. Slides: 141-181

Red Black trees are stricter Binary Search Trees. Their rules include:

1. They have red nodes and black nodes
2. The root of a RBT and the NIL elements of RBT are always black
3. There may never be two sequential red nodes
4. New nodes inserted into the tree are always red. If the tree then breaks the RBT rules then adjustments must be made to the tree

1.10 Order Statistic Trees

Ref. Slides: 184-197

Order Statistic trees are Red Black Trees with sizes stored in their nodes. Keys are sorted in normal BST fashion while the size of a node is equal to `1 + sizeof(left_child) + sizeof(right_child)`.

1. `OS-SELECT(x, i)` returns the pointer to the node containing the i^{th} smallest key in the subtree rooted at x. The initial call is always `OS-SELECT(root[T], i)`
2. `OS-RANK(T, x)` returns the total summed rank of the passed node
3. `OS-INSERT` inserts like a regular BST and then restores RBT properties

1.11 Interval Trees

Ref. Slides: 201-214

Interval trees are yet another variation of the Red Black Tree, but these store ranges as well as the maximum extreme contained in their children. Operations are performed relative to the lower bound of the range and the tree's red black properties are restored during insertions.

2 Study Guide

2.1 Problem 1

1. Describe the reason for randomizing the `Partition` procedure used in the Randomized-Quicksort algorithm.

   `Partition` is randomized to avoid worst-case scenarios. `Quicksort` is at its worst when the array is close to or completely sorted when the algorithm is called to sort it. If the partition is randomized, it makes it so that `Quicksort` is more efficient as it will always start with its partition as a random element for more even separations instead of an incredibly unbalanced one if the element is near or already sorted.

2. Supposed we use `RANDOMIZED-SELECT`...

   The maximum element is found immediately during the first call if the first pivot is `9` as `Partition` would sort everything to the left of `9` as it is already the greatest element, meaning it would be found before `Quicksort` recurs.

3. What is the difference between the `MAX-HEAP` property and the binary search tree property?

   The main difference between the two is their inherent rule:

   (a) In a `MAX-HEAP`, each parent is guaranteed to be greater than both of their children

   (b) In a `Binary Search Tree`, a parent's left child always stores a greater value than its value and its right child stores a lesser value than its value