**CS 284 F2020 Endterm Review Sheet**

### **Topics**

* Stacks
* Queues
* Recursion
* Trees
  + Binary Trees: add, remove, find
* Heaps & Priority Queues
  + min heaps, maxheaps, add, remove
  + Array-based implementation (only implementation we saw)
* Hash Tables
  + Collision resolution: open addressing (linear probing) and channing
* Sorting
  + Insertion sort, selection sort, bubble sort
  + quicksort

### **Table of Contents** (topic | pg number)

Stacks | 2

Queues | 2

Recursion | 3

Trees | 3

Heaps & Priority Queues | 4

Hash Tables | 6

Sorting | 7

### **Stacks**

Functionality

* List where you can only access the top/head element
  + Works liked a linked list
  + Think stacks of plates
* Stores data via Last-In, First-Out (LIFO)

Use

* Checking palindromes
* Balanced parenthesis
* Can either be implemented with a list (linked list or normal array) component or as a subclass of vector

Methods

* Boolean empty ()
  + Tests if stack is empty
* E peek()
  + Inspects the top element
* E pop()
  + Removes top element from stack, returning it
* E push (E obj)
  + Puts a new element on stack

### **Queues**

Functionality

* A list where you can only access the front & end object
  + Like a line at a checkout
  + Can again, like a stack, be used through the modification of a linked list
* First-In, First-Out (FIFO)
  + Add to tail, remove from head

Use

* Operating Systems
  + Keep track of tasks waiting for resources and ensure tasks are carried out in order they are generated
* Study performance of physical systems and estimate performance
  + Useful in modeling and simulation

Methods

* Single Linked list implementation
  + Boolean offer(E item)
    - Inserts item at rear of queue
  + E poll()
    - Remove and return the item at front of queue
  + E peek()
    - Return item at from of queue without removing it from queue

### **Recursion**

Functionality

* A program or function that self-refers to itself as a smaller unit
* Functions exactly like a loop but instead of modifying the contents of a function within a loop, the function does the modifying by calling itself with different parameters and returning components that accumulate to the final/correct output
* Tail recursion
  + Only one recursive call and it is the last instruction performed within the method

Use

* Whenever you use a loop, some problems can be done quicker and simpler if done recursively

Methods

* Note that stop condition for recursion must be built in to the body of the method, as opposed to a loop that has the stop condition built into the run condition for the loop

### **Trees**

Functionality

* A nonlinear and hierarchical data structure used to store data
* Tree nodes can have multiple successors (children) but only one predecessor (parent/ancestor)
* Binary Tree
  + Tree where each node has at most two successors
  + Starts from root, node with no parents, and continues downward and stops at leafs, nodes with no children
    - Leafs are external nodes, other nodes are external
  + Each child is located on a branch
  + If child node’s share the same parent they are sibling
  + Height of tree is determined by longest sequence of nodes from root to a leaf
    - Levels start at level 1, level of root
  + Binary Search Tree
    - A binary tree where either the Tree is empty or the left children of a node contain all the values less than the node’s and the right children of that same node contain all the values greater than that node
  + Full Binary Tree
    - All nodes either have 2 or 0 children
  + Perfect Binary Tree
    - Full binary tree of height n
    - All leafs end at same depth, i.e. every path from node to leaf ends at same level
  + Complete binary tree
    - Is perfect through level n-1
    - Level n is filled from left to right
* Tree Traversal
  + For examples below follow this explanation of visiting nodes (done recursively)
    - Hit node then go down left subtree, return to node, then go down right subtree and returning at the node, visiting a node 3 times in that specific order
    - Hit node before going down the left subtree = first hit
    - Hit node returning from left subtree to go down right subtree = second hit
    - Hit node returning from right subtree = third hit
  + Inorder (order-visited: left-subtree, node, right-subtree)
    - Counts node once it has hit the same node twice, counts node on second hit
  + Preorder (node, left-subtree, right-subtree)
    - Counts node once it has hit node once, counts node on first hit
  + Postorder (left-subtree, right-subtree, node)
    - Counts node once it has hit node three times, counts node on third hit

Methods

* Check eclipse for others not listed
* insert()
  + For BST
    - Takes Node to be added and compares it to current node and places it accordingly, addedNode>currentNode, go down right subtree, addedNode<currentNode, go down left subtree
      * Update pointers once correct place is found, can update as you go down BST
* removal()
  + For BST
    - Must remove reference from parent to node being removed, must keep structure of BST with removal
    - If node being deleted has children, compare node’s children’s values and swap node with greater child, repeat until node becomes a leaf, then remove reference of node to delete it
      * If one child is null, make parent of node reference non-null child, and delete node

### **Heaps & Priority Queues**

**Heap**

Functionality

* A complete binary tree with the following properties
  + The value of the root is either the smallest item in the tree (minheap) or the largest item in the tree (maxheap)
  + Every subtree is also a heap
  + Is a complete binary tree in structure
  + Insertion and removal is O(log n)
  + Top to bottom follows BST ordering, but left to right ordering on levels don’t always follow BST rules

Use

* Can be implemented efficiently using an array instead of a linked data structure
  + [root, left child of root, right child of root, left child of left, right child of left, left child of right, right child of left, ….]
    - I = 0 is root, level 1
    - I = 1-2, level 2
    - I = 3-6, level 3, etc.
  + Node at p: left child at 2p+1, right child at 2p+2, parent at (p-1)/2
* Not useful by itself

Methods

* add()/insert()
  + If heap is empty, add item as root, otherwise add to as either the left/right child at the highest level depending on how that level is filled, fill the first open spot
  + After adding the element as a leaf, compare its value to its parent and shift accordingly (whether it is a minheap/maxheap) and bubble up until correct placement is reached
    - Code:
      * Insert e in the next position at the bottom of the Heap
      * while (e not at the root and is smaller than its root){ Swap e with its parent, moving e up the heap }
* remove()/poll()
  + Removing root
    - Remove the item in the root by replacing it with the last item in the heap (LIH)
    - while (item LIH has children and it is larger than than either of its children) { Swap item LIH with its smaller child, moving LIH down the heap }
  + Heaps only remove from node, maxheaps remove largest item and minheaps remove smallest item

**Priority Queues**

Functionality

* Utilizes a heap
* Abstract Data Type where only the highest-priority item is accessible (like a queue but the highest priority item is not necessarily the first item entered)

Use

* Custom priority queue using ArrayList as opposed to java native version that uses Object[] array

Methods

* Int compare (E left, E right)
  + If null it uses compareTo
  + private int compare(E left, E right) {
    - if (comparator != null) { // A Comparator is defined.
      * return comparator.compare(left, right);
    - } else { // Use left’s compareTo method.
      * return ((Comparable) left).compareTo(right); } }
* Boolean offer (E item)
  + @Override
  + public boolean offer(E item) {
    - // Add the item to the heap. theData.add(item);
    - // child is newly inserted item.
    - int child = theData.size() - 1;
    - int parent = (child - 1) / 2; // Find child’s parent.
    - // Reheap
    - while (parent >= 0 && compare(theData.get(parent), theData.get(child)) > 0) {
      * swap(parent, child);
      * child = parent;
      * parent = (child - 1) / 2; }
    - return true; }
* E poll()
  + @Override
  + public E poll() {
    - if (isEmpty()) { return null; }
    - // Save the top of the heap.
    - E result = theData.get(0);
    - // If only one item then remove it.
    - if (theData.size() == 1) { theData.remove(0); return result; }
    - /\* Remove the last item from the ArrayList and place it into the first position. \*/
    - theData.set(0, theData.remove(theData.size() - 1));
    - // The parent starts at the top.
    - int parent = 0;
    - while (true) {
      * int leftChild = 2 \* parent + 1;
      * if (leftChild >= theData.size()) { break; /\* Out of heap.\*/ }
      * int rightChild = leftChild + 1;
      * // Assume leftChild is smaller.
      * int minChild = leftChild;
      * // See whether rightChild is smaller.
      * if (rightChild < theData.size() && compare(theData.get(leftChild), theData.get(rightChild)) > 0)
        + { minChild = rightChild; }
      * // assert: minChild is the index of the smaller child.
      * // Move smaller child up heap if necessary.
      * if (compare(theData.get(parent), theData.get(minChild)) > 0) {
        + swap(parent, minChild);
        + parent = minChild;
      * } else {
        + // Heap property is restored.
        + break; } }
    - return result; }

### **Hash Tables**

Functionality

* Data structure that is based on a collection of Key-Value pairs
  + Like a dictionary in python
  + Access data based on key
    - Key can be any data type, as well as the values
* Allow for fast and immediate access to data
* Data is stored in an array from index 0-n-1
  + Each array slot is called a bucked
  + <Key,Pair>
  + Set-up/insertion of all elements = O(n)
  + Retrieval/search/find(ideally) = O(1)
* Can have collisions
  + Collisions = 2+ keys mapped to same index
  + Fixes
    - Separate chaining (closed addressing)
      * Making a linked list of elements at a specific bucket where collision happened
        + First inserted element of bucket is head and sequential collisions add onto linked list
    - Linear probing (open addressing)
      * Place <Key,Pair> in next open bucket in array
        + Can result in clustering
  + This is bad as it makes time to find <Key,Pair> O(n)

Use

* Storing data in a structure that allows for immediate access of said data
* Ex:
  + Phonebook, key=name, value=phone number
  + Account, key=username, value=password

Methods

* Adding
  + Key is put through a hash function and is mapped to a specific index in the array (a bucket)
  + This allows for constant lookup time, as it is just referencing an index in an array
* Common hashing methods
  + Calculator applied to a key to transform it into an address
  + When key is a number
    - Take key, divide by number of available addresses, n, and take remainder
    - Address = key Mod n
  + When key is a string
    - Divide sum of ASCII codes for each symbol in key by number of available addresses, n, and take the remainder

### **Sorting**

Functionality

* Arranging data in order

Uses

* Sorting arrays of primitive types
  + Built off of quicksort algorithm, O(nlogn)
* Sorting arrays of objects and lists
  + Built off of merge sort algorithm, O(nlogn)

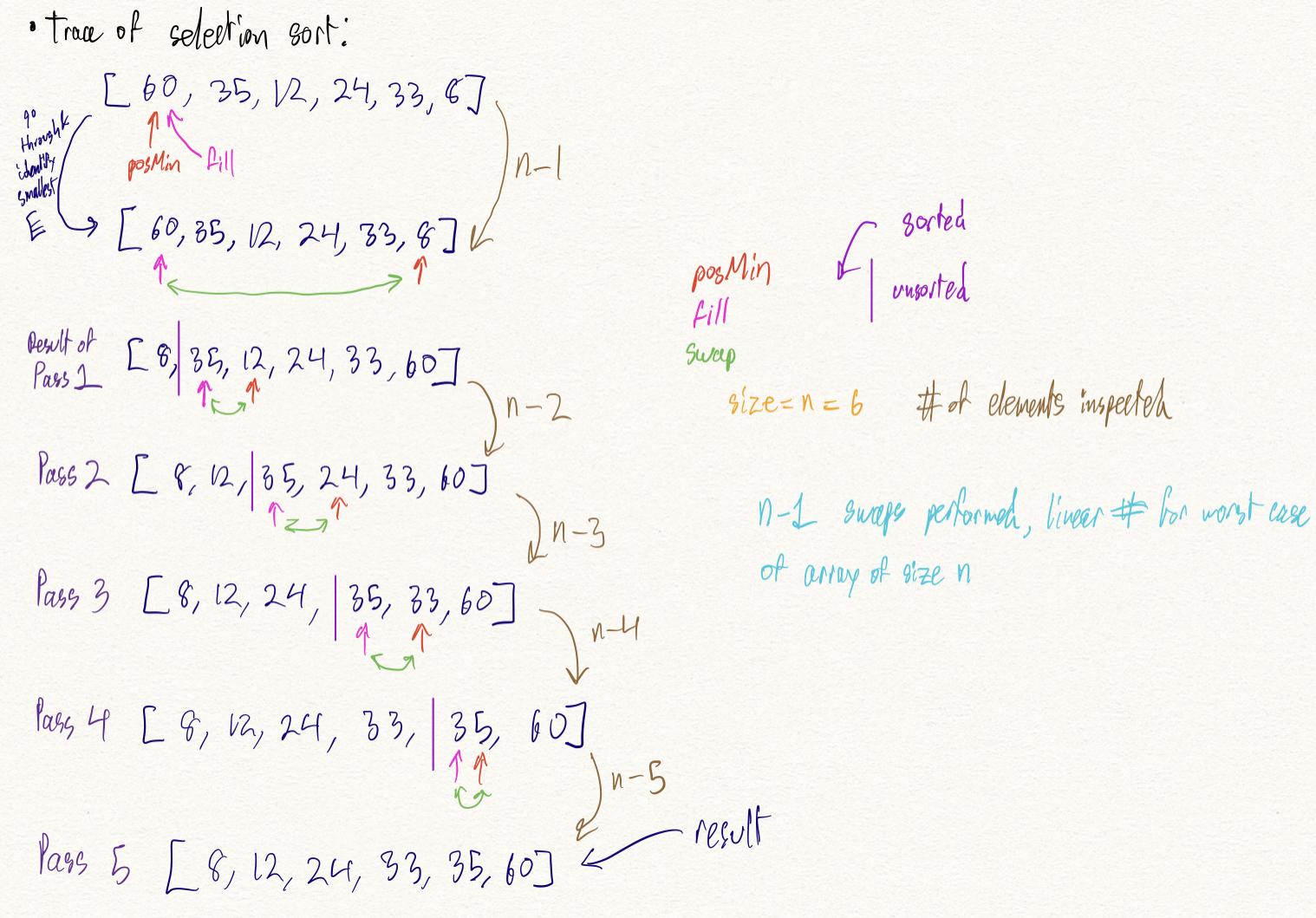
Algorithms

* Quadratic Sorts
  + Similarities
    - Require storage for array being sorted, array is sorted in place, variables have storage requirements, none are good for large arrays
  + Selection Sort
    - Time complexity = O(n^2)
    - # comparisons
      * Best = O(n^2)
      * Worst = O(n^2)
    - # exchanges
      * Best = O(n)
      * Worst = O(n)
    - Passes through array multiple times, selecting next smallest element in array each time and placing it where it belongs
    - Code
      * N = number of elements in array
      * for fill = 0 to n - 2 {
        + posMin = fill
        + for next = fill + 1 to n - 1 {

if (a[next]<a[posMin])

posMin = next

* + - * + }
        + swap(a,posMin,fill);}



* + Bubble Sort
    - Time complexity = O(n^2)
    - # comparisons
      * Best = O(n)
      * Worst = O(n^2)
    - # exchanges
      * Best = O(1)
      * Worst = O(n^2)
    - Compares adjacent elements and bubbles up smaller values to front of array and sinks larger values, placing them at the end
      * Every iteration, starting at front of list, largest value that’s unsorted gets pushed back
    - Code
      * do
        + exchanges=false;
        + for each pair of adjacent array elements

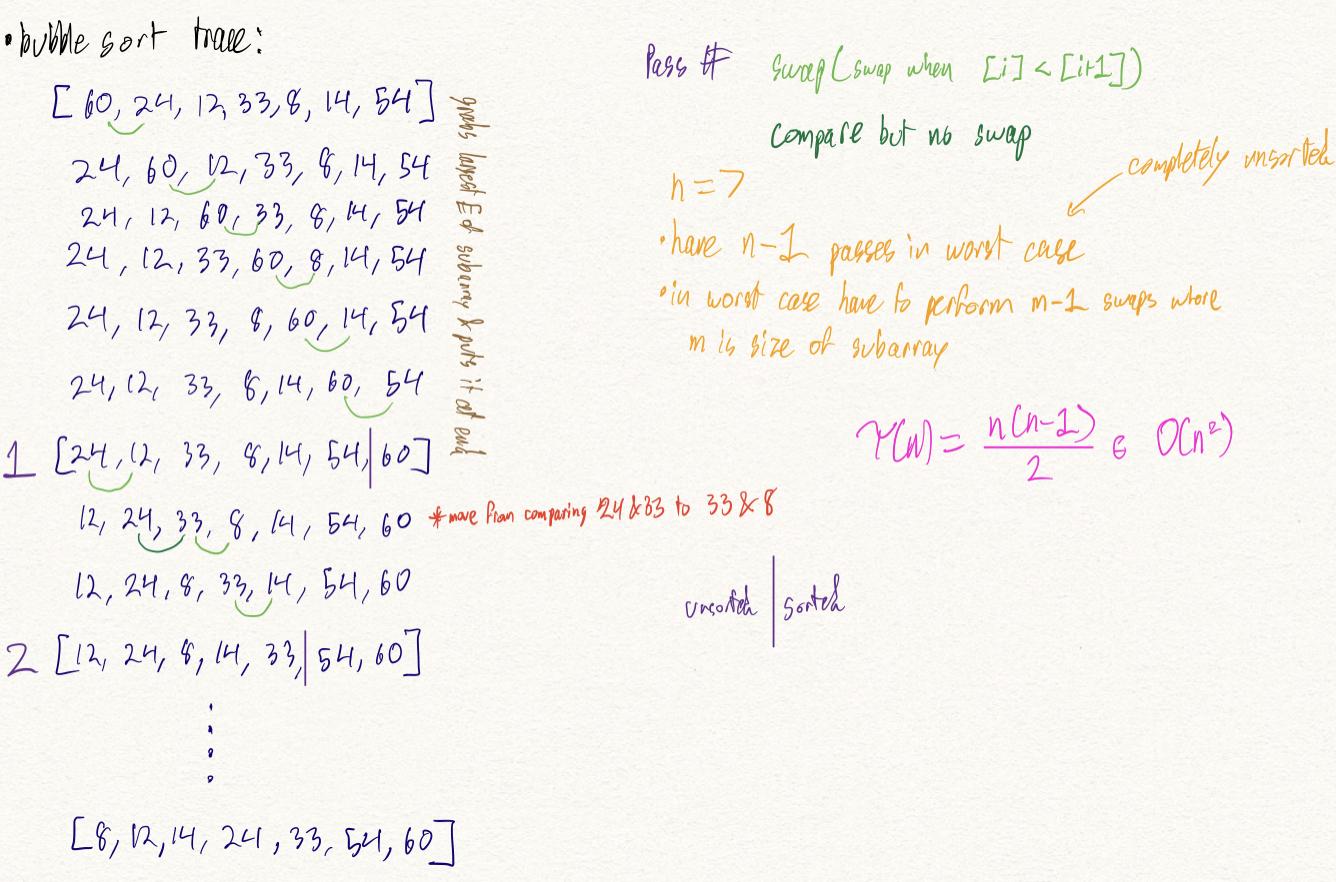
if the values in a pair are out of order {

Exchange the values

exchanges=true;

}

* + - * while exchanges==true

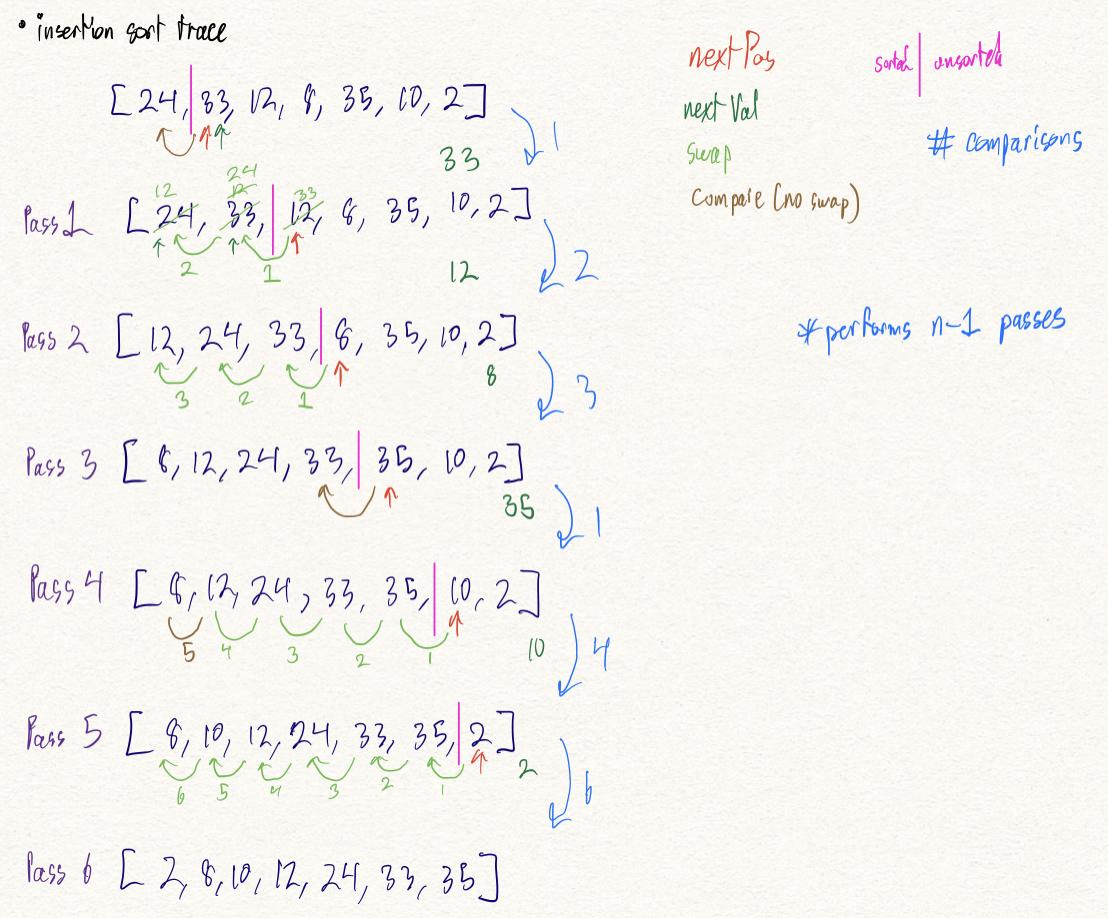


* + Insertion Sort
    - Time complexity = O(n)
    - # comparisons
      * Best = O(n)
      * Worst = O(n^2)
    - # exchanges
      * Best = O(n)
      * Worst = O(n^2)
    - Like sorting cards that are dealt to you, as you get the cards (move down array) you place them in their proper places
      * Get new card, move to new index in array, sort card, place value at that instance in correct position in array
    - Code
      * for nextPos = 1 to n-1 {
        + nextPos is the position of the element to insert;
        + nextVal = a[nextPos];
        + while (nextPos>0 and a[nextPos-1] > nextVal) {

Shift the element at nextPos-1 to position nextPos;

nextPos--; }

* + - * + Insert nextVal at nextPos; }



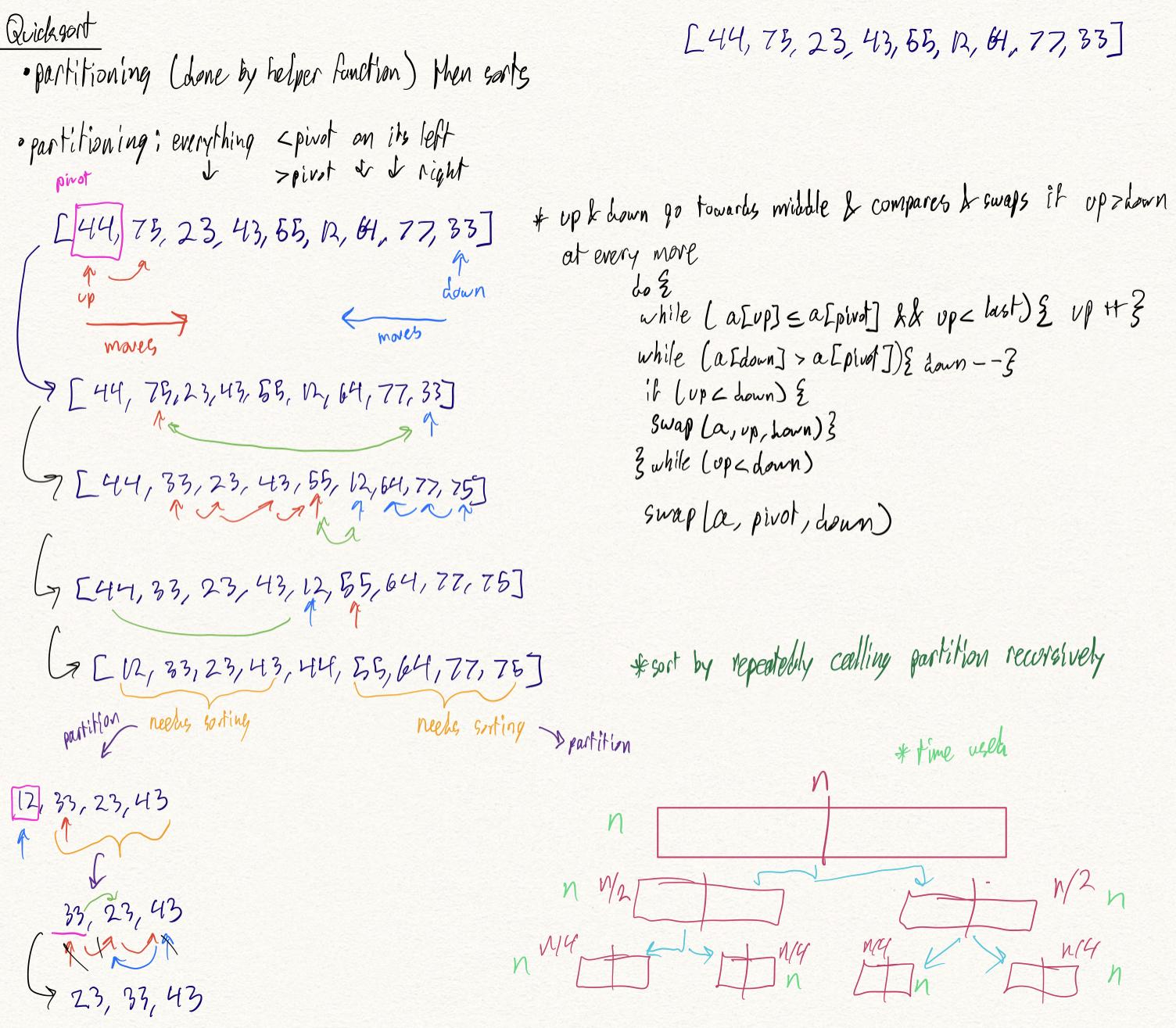
* Quicksort
  + Time complexity = O(n log n)
    - Performs poorly if each time array is partitioned, a subarray is empty
      * Time complexity = O(n^2) for this case
  + Done in 2 phases: partitioning then sorting
    - Sorting is actually done by recursively repeating the partitioning process as partitioning correctly places the pivot in its final spot in the sorted list
    - Partitioning
      * Pivot: specific value which array is being rearranged around
      * Up: pointer at front half of array
      * Down: pointer at back half of array
      * Partitioning works by comparing up first to the pivot, and moving the pointer accordingly
        + If up <= pivot move it to the next node
      * After this up and down compare each other and swap elements as needed, while moving towards the middle of list
      * Partitioning stops when up > down
    - Sorting
      * A repeat of the partitioning but breaking the array into two subarrays of the elements to the left of the partition (these elements are less than the value of the partition) and the elements to the right (larger than partition)
  + Code
    - partition(table, first, last)
      * pivot = table[first];
      * up = first;
      * down = last;
      * do {
        + Increment up until it selects the first element greater than pivot or up == last
        + Decrement down until it selects the first element less than or equal to pivot
        + if (up<down)

{ swap(table[up],table[down]) }

* + - * + } while (up < down)

swap(table[first],table[down]);

* + - * return down;
    - quicksort(table,first,last) { //pivIndex is index of pivot after partitioning
      * if (first < last) {
        + Partition the elements in the subarray first...last so that the pivot value is in its correct place (subscript pivIndex)
        + quicksort(table, first, pivIndex - 1);
        + quicksort(table, pivIndex + 1, last); } }



**Sorting Comparisons**

Number of Comparisons

| Sorting algorithm | Best | Average | Worst |
| --- | --- | --- | --- |
| selection | O(n^2) | O(n^2) | O(n^2) |
| bubble | O(n) | O(n^2) | O(n^2) |
| insertion | O(n) | O(n^2) | O(n^2) |
| shell | O(n^7/6) | O(n^5/4) | O(n^2) |
| merge | O(n log n) | O(n log n) | O(n log n) |
| heapsort | O(n log n) | O(n log n) | O(n log n) |
| quicksort | O(n log n) | O(n log n) | O(n^2) |