Ch 13

Searching and Sorting

Sequential Search

A sequential search starts at the first element and compares the key to each element in turn until the key is found, or until the end of the list is reached.

If the list is sorted in ascending order, stop searching as soon as the key is greater ~~that~~ than the current list element, or until the end of the list is reached.

Note:

- The best case has the key in the first slot

- The worst case, the key is in the last slot or not in the list. All n elements must be examined.

- On average there will be n/2 comparisons.

Binary Search

If the elements are in a sorted array, a divide-and-conquer approach is more efficient.

The following recursive pseudo-code algorithm shows how binary search works.

Assume that a[low] ... a[high] is sorted in ascending order and that a method binSearch returns the index of key. If key is not in the array, it returns -1.

Suppose 5 is the key to be found. (How many Comparisons or Passes would you need to find that value?)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a
1	4	5	7	9	12	15	20	2

low	high	mid(a[mid])
0	8	4
0	3	1
~~First Pass mid = (8+0)/2 = 4~~		
2	3	2

First Pass mid = (8+0)/2 = 4 Check a[4]

Second Pass mid = (3+0)/2 = 1 check a[1]

Third Pass mid = (3+2)/2 = 2 Check a[2]

Yes! key is found

Exp

a[0] a[1] . . . . . a[8]

1 4 5 7 9 12 15 20 21

Note:

- In the best case, the key is found on the first try.
- Worst case, the key is not in the list or is at either end of a sublist.

An easy way to find the number of comparisons in the worst case is to round n up to the nearest power of 2 and take the exponent. For example, in the array above, n = 9. Suppose 21 were the key. Round 9 up to 16 which equals 2^4. Thus, you would need four comparisons to find it.

Be familiar with the Sequential and binary search algorithims. You should know That a binary search is more efficient than a sequential search and that a binary search can only be used for a list that is sorted

n! Searching & Sorting

Selection Sort

This is a Search-and-Swap algorithm. (Assume that we want to sort in ascending order).

8 1 4 6

[1] 8 4 6 after 1st pass

[1] [4] 8 6 after Second Pass

[1] [4] [6] [8] after 3rd Pass.

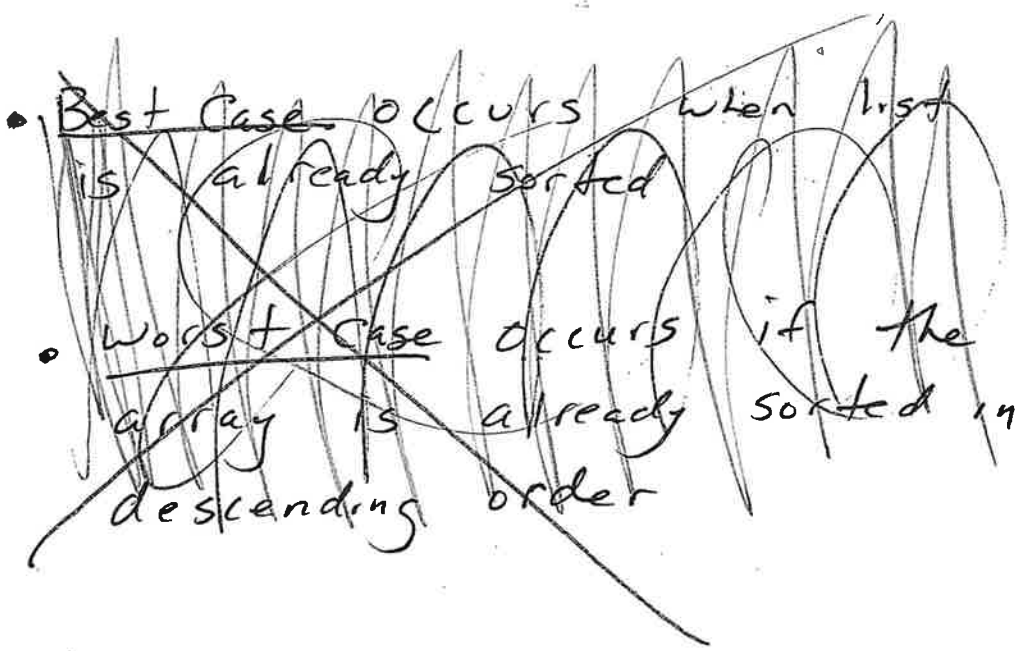
Here's how it works:

- Find the smallest element in the array and exchange it with a[0].
- Now find the smallest element in the subarray a[1] ... a[n-1] and swap it with a[1].
- Continue this process until just the last two elements remain to be sorted a[n-2] and a[n-1]. The smaller of these two elements is placed in a[n-2]; the large in a[n-1]

Note:

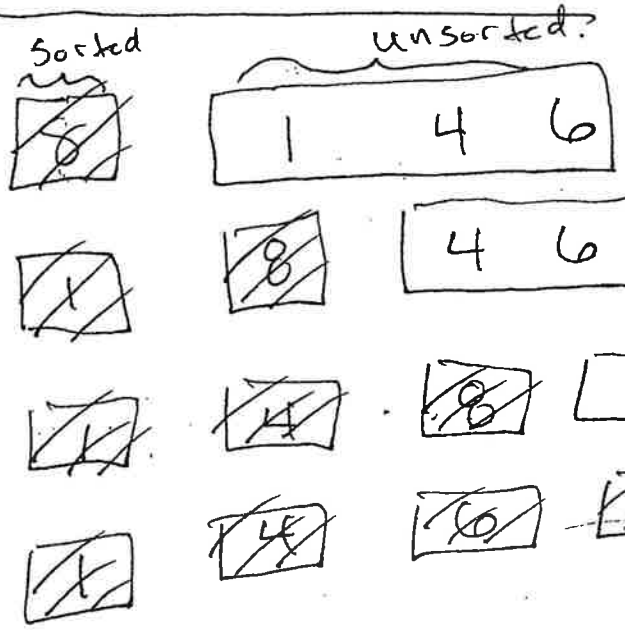
- For an array of n elements, the array is sorted after n-1 passes.

- After the k^{th} pass, the first k elements are in their final sorted position.



Best Case, worst case, Average Case, are same for selection sort

Insertion Sort



Think of the first element a[0] as being sorted w/ respect to itself

Now the array can be thought of as consisting of two parts, a sorted list followed by an unsorted list.

The idea is to move elements from the unsorted list to the sorted list one at a time; as each item is moved, it is inserted into its correct position in the sorted list. Elements in the sorted list will need to be shifted to create a slot for the new item.

Note:

- For an array of n elements, it is sorted after n-1 passes.

- After the k^{th} pass a[0], a[1]... a[k] are sorted w/ respect to each other

- Worst case is when the array is initally sorted in reverse order. (Same as average case.)

- The best case occurs if the array is already sorted in increasing order.

Both insertion and selection sorts are inefficient for large n.

nlogn

Recursive Sorts

Selection and insertion sorts are inefficient for large n. More efficient divide-and-conquer algorithms can be used for large sets of data.

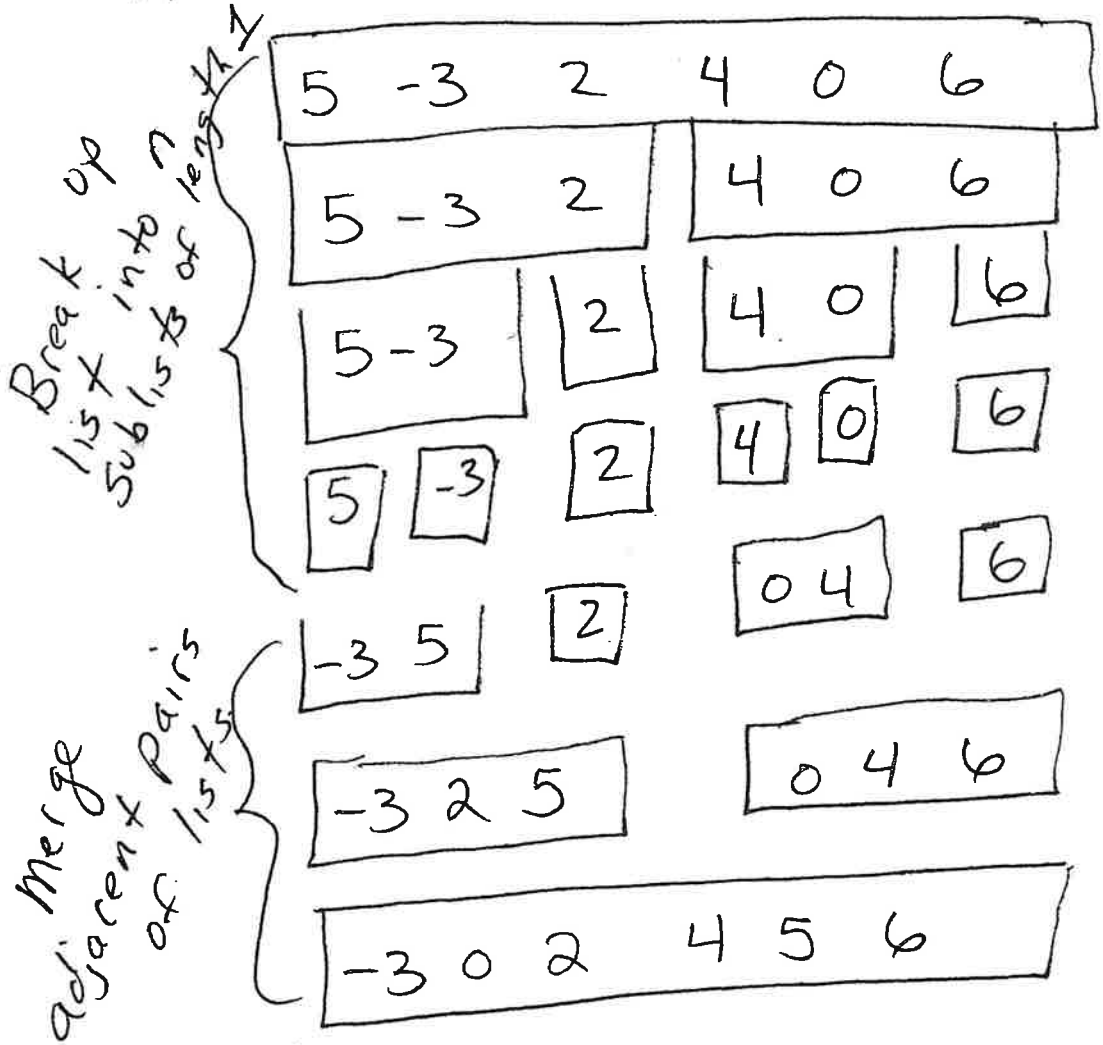
Mergesort

Here is a recursive description of how mergesort works:

If there is more than one element
- Break the array into two halves.
- mergesort the left half
- mergesort the right half
- merge the two subarrays into a sorted array.

Mergesort uses a merge method to merge two sorted pieces of an array into a single sorted array. For example,

Here's what happens in Mergesort



- Start with an unsorted list of n elements.
- The recursive calls break the list into n sublists, each of length 1. Note that these n arrays, each containing just one element are sorted.
- Recursively merge adjacent pairs of list until there is just one list of length n.

- The main disadvantage of mergesort is that it uses temporary arrays

- Best case, worst case, and average case have similar run times.

Quicksort

For large n, quicksort, is on average, the fastest known sorting algorithm.

Here is a recursive description of how quicksort works:

If there's at least two elements in the array
- Partition the array
- Quicksort the left subarray
- Quicksort the Right subarray.

Notes:

1) You need three pointers: The left pointer (L), you need a right pointer (R), and you need a pivot pointer (P).

(2) The pivot can start anywhere; Sometimes the starting point of the pivot is randomly selected. We will start the pivot on the left hand side.

3) Once the left pointer, the right pointer, and the pivot line up that value is in its correct place. At this point, all the values to the left of the pivot are smaller and all the values to the right of the pivot are larger.

4) Also, remember that the pointer that is attached to the pivot does not move. The other pointer always moves towards the pivot.