Homework 7

1. Why use Insertion Sort?

Insertion sort has its uses even though quick and merge sort are faster. These two faster sorts are great at things involving incredibly large data sets. But with smaller data sets it can be a bit of overkill since they take significantly more code to set up. Also Insertion Sort can be good for situations where data is given to you one at a time. Say each value of user input is given one at a time into an array. Since you don’t know all the datapoints upfront it can be helpful to juts sort them as they come in.

1. If you had an array of numbers, averaged them all and used that for the pivot, would that be faster or slower?

Let’s create a sample array

Array = {42, 8, 68, 11, 7, 13, 12, 1}

Size = 8

Total of all numbers = 162

Average = 162/8 = 20.25

Closet to average is number 13 so we will make that the pivot

New array = {8, 11, 7, 12, 1, 13, 42, 68}

Left average = (8+11+7+12+1) / 5 = 7.8

New Left Array = {1, 7 , 8, 11, 12}

Right Average = (42+68) / 2 = 55. Both of these are equidistant away and pick either as the pivot will keep it in this order since it is already sorted

Putting all sub arrays are sorted you get new array = {1, 7, 8, 11, 12, 13, 42, 68}.

This method seems to keep it spend around O(n log2(n)). Similar to other situations with quicksort there is a possibility that this ends up in the worst case o(n^2) but it would need somethings like a large array of mostly single digit numbers and a few incredibly large numbers to skew the average and make the few big numbers consistently the least useful pivot and giving it the worst speed.

1. Should we use the middle instead of random section to be the pivot in quick sort? Why or why not?

I think the pivot is easiest when it’s random. There is a theoretical case to be made that it is *possible* that the random pivot selection is the least helpful pivot almost every time; which could give you a worst case of O(n^2). But just because it’s theoretical and can happen doesn’t mean that in everyday practice that turns out. Maybe if you’re working on something like the Mars Rover you would want something that can’t theoretically crash the computer but for the majority of programmers who don’t work on expensive closed systems using a random int as the index is in practice faster.

1. Given a skyscraper of n height where you throw a suitcase out the window each time needing a new case to test the minimal height a breach is made in the case. Design an algorithm that will find minimal height of breech and using minimal items for test

Let’s say our skyscraper is 50 stories tall and that it starts breaching at 30 stories (but our tester doesn’t know that yet). Since the floors are, in a sense, sorted already; all we need to do is search the floors. The quickest way for this I believe would be binary sort.

* Go to midway point of current sample of floors (i.e. to start halfway between floor 1 and floor 50 is floor 25).
* Throw it out the window
* Did the case breech?
  + If yes go lower (if breech at level 25 go to midway point between 25 and 1 which would be 13)
  + If no go higher (if no breech at 25 go to ((50-25)/2)+25 = 38.
* Repeat until new midway points of sets are exhausted. (using the formula ((y-x)/2 + x; y and x are equal you will get a divide by zero error and you will know all pivots are exhausted)

To run through this example of minimal break at 30 and a tower of 50 let’s run through

* (50-0)/2 = 25
* Test at 25; no breach
* ((50-26)/2)+26) = 38
* Test at 38; breach
* ((38-26)/2)+26 = 32
* Test at 32; breach
* ((32-26)/2)+26 = 29
* Test at 29; no breech
* ((32-30)/2)+30 = 31
* Test at 31; breech
* ((31-30)/2)+30 = 31
* 31 already tested; only datapoint in set not exhausted is 30
* Test at 30; breech
* ((30-30)/2)+30 = division by zero error.
* No more midpoints; 30 is minimal drop point for breech.

In a sample of 50 stories to test we made it through with only busting 6 suitcases for our test. A linear search would have needed 30 suitcases.

1. A pile of mixed nuts and bolts need to be sorted. Array of nuts and Array of bolts are same size n. Nut cannot be compared to other nuts. Bolts cant be compared to other bolts. Each nut matches one bolt exactly. Can only tell if a nut is bigger or smaller compared to bolt it is tried to be matched with

Let’s make 2 arrays of equal length of measurements in millimeters. Two objects of the same mm length fit together.

* Go through one array linearly. Put all the other ones in the array smaller than target to left and bigger to right.
* Use the index of the second array to move that object of the first array know it will have the same amount of spaces
* Add the index to a new array to be checked each pass to see if item is in it
  + If item is at an index in the index position array; we know it is already sorted and can be skipped
  + If not it should be sorted.
* Repeat until size of new array = size of other two arrays

1 5 7 12 13 4 20 9 3 3

Nuts = {1, 7, 5, 12, 4, 20, 13, 3, 9, 3}

Bolts = {12, 13, 4, 20, 3, 1, 7, 5, 3, 9}

Let’s go through the nuts array to sort the bolts array

1st pass: first nut is 1mm; all others are naturally bigger and already to the right in first array so just need to move it to index 0 in bolt array

Nuts = {1, 7, 5, 12, 4, 20, 13, 3, 9, 3}

Bolts = {1, 12, 13, 4, 20, 3, 7, 5, 3, 9}

Correct Position = {0}

2nd pass: next is 7mm. we try every unmatched bolt and find 4 are too small and 4 are too big. This puts bolt at index 5 which means this nut should also be at index 5.

Nuts = {1, 5, 12, 4, 20, 7, 13, 3, 9, 3}

Bolts = {1, 4, 3, 5, 3, 7, 12, 13, 20, 9}

Correct Position = {0, 5}

3rd pass: 5mm. 3 unmatched smaller, 4 unmatched bigger. Puts 5mm at index 4. Add index to array and move nut item to 4th index. Size of position array != to 10 so repeat.

Nuts = {1, 5, 12, 4, 20, 7, 13, 3, 9, 3}

Bolts = {1, 4, 3, 3, 5, 7, 12, 13, 20, 9}

Nuts = {1,12, 4, 20, 5, 7, 13, 3, 9, 3}

Bolts = {1, 4, 3, 3, 5, 7, 12, 13, 20, 9}

Correct Position = {0, 5, 4}

4th pass: 12mm. 2 unmatched bigger, 4 unmatched smaller. Places 12mm at 7th index so we add to array and move nut item to 7th array.

Nuts = {1,12, 4, 20, 5, 7, 13, 3, 9, 3}

Bolts = {1, 4, 3, 3, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4}

Nuts = {1, 4, 20, 5, 7, 13, 3, 12, 9, 3}

Bolts = {1, 4, 3, 3, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7}

5th pass: 4mm. 2 unmatched smaller. 2 unmatched

Nuts = {1, 4, 20, 5, 7, 13, 3, 12, 9, 3}

Bolts = {1, 4, 3, 3, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7}

Nuts = {1, 20, 4, 5, 7, 13, 3, 12, 9, 3}

Bolts = {1, 3, 3, 4, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7, 3}

6th pass: 20mm. 4 smaller. 0 bigger. Place at index 9

Nuts = {1, 20, 4, 5, 7, 13, 3, 12, 9, 3}

Bolts = {1, 3, 3, 4, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7, 3}

Nuts = {1, 4, 5, 7, 13, 3, 12, 9, 3, 20}

Bolts = {1, 3, 3, 4, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7, 3, 9}

7th pass: 13mm. 3 smaller. 0 bigger.

Nuts = {1, 4, 5, 7, 3, 12, 9, 3, 13, 20}

Bolts = {1, 3, 3, 4, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7, 3, 9, 8}

8th pass: 3mm. 1 equal. 1 bigger.

Nuts = {1, 3, 4, 5, 7, 12, 9, 3, 13, 20}

Bolts = {1, 3, 3, 4, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7, 3, 9, 8, 2}

9th pass: 3mm. 1 bigger.

Nuts = {1, 3, 3, 4, 5, 7, 12, 9, 13, 20}

Bolts = {1, 3, 3, 4, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7, 3, 9, 8, 2, 1}

final pass: 9mm. all objects sorted

Nuts = {1, 3, 3, 4, 5, 7, 9, 12, 13, 20}

Bolts = {1, 3, 3, 4, 5, 7, 9, 12, 13, 20}

Correct Position = {0, 5, 4, 7, 3, 9, 8, 2, 1, 6}