COMP26120 - January 2014 - Answers

Todd Davies

January 6, 2015

Please don't assume these answers are right. This is me attempting a past paper for revision purposes; I could have got it all wrong ;)

I chose to answer questions 1 & 2.

Question 1

Part a

```java
public static boolean hasSum(int[] input, int sum) {
  for(int i = 0; i < input.length; i++) {
    for(int j = 0; j <= i; j++) {
      if((input[i] + input[j]) == sum) {
        return true;
      }
    }
  }
  return false;
}
```

Listing 1: Add all of the pairs of integers together in the list. Do this by looping through the list, and looping from 0→i on each iteration.

Since as we're iterating through the list, we are describing an arithmetic series (we do one additions, then two, then three etc), we can use the formula $\frac{n(1+n)}{2}$ to find how many operations we're doing. This, in the Big-Oh notation, equates to $O(n^2)$ since constant terms are eliminated. In most cases, the result will be found significantly faster though.

Part b

```java
import java.util.ArrayList;

public class part2a {
  public static ArrayList<Integer> intersect(int[] input, int[] input2) {
    BitSet bs = new BitSet(input.length);
    for(int i : input) bs.set(i);
    ArrayList<Integer> intersection = new ArrayList<Integer>();
    for(int i : input2) if(bs.get(i)) intersection.add(i);
    return intersection;
  }
```

Listing 2: Add the first list to a BitSet (a very compact array of bits), then iterate through the second list and add items that are in the bit set to the output array.

The worst case runtime of this algorithm is $O(n + m)$ since it iterates through both arrays once, and the loops aren't nested.

Part c

```
public static char[] shift(char[] input, int shiftNum) {
  char[] firstHalf = Arrays.copyOfRange(input, 0, shiftNum);
  char[] secondHalf = Arrays.copyOfRange(input, shiftNum, input.length);
  for(int i = 0; i < (input.length - shiftNum); i++) {
    input[i] = secondHalf[i];
  }
  for(int i = (input.length - shiftNum); i < input.length; i++) {
    input[i] = firstHalf[i - (input.length - shiftNum)];
  }
  return input;
}
```

Listing 3: Create a new array containing the first half, and a new array containing the second half, then replace them insert them into the original array.

The worst case runtime of this function is $O(n)$, since it only iterates through the array once. The space complexity is $O(2n)$, since I couldn't figure out how to do an in-place circular shift.

Question 2

Part a

i)

Merge Sort

ii)

Quick Sort

iii)

Radix Sort

Part b

i)

It never ends.

ii)

The average time that the smallest element will take to reach it's correct position is $n \cdot n$, since we need one die roll to be 0, and the other to be the current index of the smallest element, and there is a $\frac{1}{n}$ chance of both occurring, so there is a $\frac{1}{n^2}$ chance of them occurring on the same iteration.

In order for all of the n items in the list to settle into their current position in the list, we need the above to happen n times, so we end up with an average running time of $n \cdot n \cdot n = O(n^3)$.

iii)

If every roll of the dice happened to be a valid swap, and the first roll was swapping the smallest element with the element at position 0, the second roll was swapping the next smallest element with the one at position 1, and so on, then after $O(n)$ iterations, the list would be sorted.

Part c

i)

- If $x > y$ and $y > z$ then $x > z$ (transitivity).
- If $x > y$ and $y > x$ if and only if $x = y$ (anti-symmetry).

ii)

The notation for lower bound is big omega (Ω).

Comparison based sorts must at have a runtime of at least $n \log n$, therefore their asymptotic running time is $\Omega(n \log n)$.

iii)

The reason that comparison based sorts must be $\Omega(n \log n)$ is that they sorting algorithms must satisfy the transitivity condition.

Comparison sorts basically just traverse a decision tree of comparisons of elements in the list. The tree can have at most $n!$ leaves (since that's how many permutations of a list of length n there are). An optimally balanced decision tree with $n!$ leaves would be $\log_2(n!)$ levels deep. This means we have to do at most $\log_2(n!)$ comparisons, and by Stirling's approximation:

$$\Omega(\log_2(n!)) \approx \Omega(\log_2(n!))$$