After Lecture 01 & 02 – Answer any questions on HW1
Practice Problems (all taken from previous exams)

1. Which of the following is not true of improved bubble sort (keep track of last swap position on the inner loop and use that to reduce outer loop iterations) on the case on input elements sorted?

    a) It is stable

    b) **Consumes less memory**. Optimized Bubble sort is one of the simplest sorting techniques and perhaps the only advantage it has over other techniques is that it can detect whether the input is already sorted. It is faster than other in case of sorted array and consumes less time to describe whether the input array is sorted or not. It consumes same memory than other sorting techniques. Hence it is not an advantage.

    c) Detects whether the input is already sorted

    d) Consumes less time

2.

Statement 1: In insertion sort, after m passes through the array, the first m elements are in sorted order.

Statement 2: And these elements are the m smallest elements in the array.

   a) Both of the statements are true.

   b) **Statement 1 is true but statement 2 is false**. There may be a smaller value $> m$ indexes from the start of the array, which is why statement 2 is false, however, the first m elements would be sorted amongst themselves.

   c) Statement 1 is false but statement 2 is true

   d) Both of the statements are false

3. Consider the following program that attempts to locate an element x in a sorted array $a[]$ using binary search. The program is erroneous. Under what conditions does the program fail?

Algorithm 1.1 Erroneous Binary Search

```
1:  function BS
2:      int $i \leftarrow 1$, $j \leftarrow 100$, $k$, $x$          ▷ assume x is assigned a value to search for
3:      int[] $a \leftarrow$ new int[100];                  ▷ assume values loaded in sorted order
4:      repeat
5:          $k \leftarrow \frac{i+j}{2}$
6:          if $a[k] < x$ then
7:              $i \leftarrow k$;
8:          else
9:              $j \leftarrow k$;
10:         end if
11:     until $((a[k] == x) \mid\mid (i \geq j))$
12:     if $a[k] == x$ then
13:         System.out.println("x is in the array")
14:     else
15:         System.out.println("x is not in the array")
16:     end if
17: end function
```

a) x is the last element of the array $a[]$

b) x is greater than all elements of the array $a[]$

c) **Both of the Above** The above program doesn't work for the cases where element to be searched is the last element of $a[]$ or greater than the last element (or maximum element) in $a[]$. For such cases, program goes in an infinite loop because i is assigned value as k in all iterations, and i never becomes equal to or greater than j. So the while condition never becomes false.

d) x is less than the last element of the array $a[]$

4. What's the worst case of insertion sort if the correct position for inserting element is calculated using binary search?

a) $O(\log n)$

b) $O(n)$

c) $O(n \log n)$

d) $\mathbf{O(n^2)}$ The use of binary search reduces the time of finding the correct position from $O(n)$ to $O(\log n)$. But the worst case of insertion sort remains $O(n^2)$ because of the series of swaps required for each insertion.

5. The following routine takes as input a list of n numbers, and returns the first value of i for which $L[i] < L[i-1]$, or n if no such number exists.

```
int firstDecrease(int* L, int n){
        for(int i=2; i <= n && L[i] >= L[i-1]; i++){}
        return i;
}
```

5a) What is the big-O runtime for the routine, measured as a function of its return value i? $O(n) \to O(i)$. $T(i) = ai + b = O(i)$ (a and b are constants).

5b) If the numbers are chosen independently at random, then the probability that firstDecrease(L) returns i is $\frac{i-1}{i!}$, except for the special case of $i = n+1$ for which the probability is $\frac{1}{n!}$ Use this fact to write an expression for the expected value returned by the algorithm. (Your answer can be expressed as a sum, it does not have to be solved in closed form. Do not use O-notation.) Use expectation

$$\begin{aligned}
\sum_{i=1}^{n+1} \frac{i-1}{i!} &= \frac{1-1}{1!} + \frac{2-1}{2!} + \frac{3-1}{3!} + \cdots + \frac{n-1}{n!} + \frac{(n+1)-1}{(n+1)!} \\
&= \frac{0}{1!} + \frac{1}{2!} + \frac{2}{3!} + \cdots + \frac{n-1}{n!} + \frac{n}{(n+1)!} \\
&= 0 + \frac{1}{2} + \frac{2}{6} + \cdots + \frac{n-1}{n!} + \frac{n}{(n+1)!}
\end{aligned}$$

$$\begin{aligned}
E_{\text{value of } i \text{ returned}} &= \sum_{i=2}^{n+1} [\text{Probability}(\text{return } i) \times i] \\
&= \sum_{i=2}^{n} [\text{Probability}(\text{return } i) \times i] + (\text{Probability}(\text{return } n+1) \times (n+1)) \\
&= \sum_{i=2}^{n} \left[\frac{i-1}{i!} \times i\right] + \frac{1}{n!}(n+1) \\
&= \sum_{i=2}^{n} \left[\frac{i-1}{(i-1)!}\right] + \frac{n+1}{n!} \\
&= \sum_{i=2}^{n} \left[\frac{1}{(i-2)!}\right] + \frac{n+1}{n!}
\end{aligned}$$

5c) What is the big-O average case running time of the routine? Hint: Simplify the previous summation until you see a common taylor series. TODO: Check these corrections

$$\begin{aligned}
\sum_{i=2}^{n} \left[\frac{1}{(i-2)!}\right] + \frac{n+1}{n!} &= \frac{1}{(2-2)!} + \frac{1}{(3-2)!} + \frac{1}{(4-2)!} + \frac{1}{(5-2)!} + \cdots + \frac{1}{(n-2)!} + \frac{n+1}{n!} \\
&= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{(n-2)!} + \frac{n+1}{n!} \\
&= \frac{1}{1} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{(n-2)!} + \frac{n+1}{n!} \\
&= 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \cdots + \frac{1}{(n-2)!} + \frac{n+1}{n!} \\
&= O(e)
\end{aligned}$$

6. Some sorting algorithms are NOT stable. However if every key in $A[i]$ is changed to $A[i] * n + i - 1$ (assume $1 \le i \le n$) then all the new elements are distinct (and therefore stability is

no longer a concern). After sorting, what transformation will restore the keys back to their original values? What is the effect on the runtime of any of the sorting algorithm if you add this transformation before executing the sort and un-transformation after the sort?

$$A[i] \to A[i] * n + i - 1$$

To transform back to the original keys replace each $A[i]$ with int$[A[i]/n]$, you cannot use $A[i] = (A[i] - i + 1)/n$ because the index i of each value has changed when we sorted. Add $O(2n)$ runtime, which does not affect the runtime of any sort because the growth of those is greater than $O(n)$.

7. a) Use pseudocode to specify a brute-force algorithm that determines when a sequence of n positive integers is given as input, whether there are two distinct terms of the sequence that have as sum a third term. The algorithm should loop through all triples of the sequence, checking whether the sum of the first two terms equals the third.

Algorithm 1.2 Brute Force Sum

```
function BruteForceSum($A$:int[], $n$:int)
    for $i \leftarrow 0 \dots n-1$ do
        for $j \leftarrow$ ~~$i$~~$0 \dots n-1$ do
            for $k \leftarrow$ ~~$j$~~$0 \dots n-1$ do
                if ~~$A[i] \neq A[j]$~~ $i \neq j$ && $j \neq k$ && $i \neq k$ && $A[i] + A[j] == A[k]$ then
                    return true
                end if
            end for
        end for
    end for
    return false
end function
```

b) Give a big-O estimate for the complexity of the brute-force algorithm from part (a). $O(n) \times O(n) \times O(n) = O(n^3)$

c) Devise a more efficient algorithm for solving the problem that first sorts the input sequence and then checks for each pair of terms whether their sum is in the sequence.

Algorithm 1.3 Smart Sum

```
function SmartSum($A$:int[], $n$:int)
    ImprovedQuicksort($A$)                                  ▷ Runtime: $O(n \lg n)$
    for $i \leftarrow 0 \ldots n-1$ do
        for $j \leftarrow i \ldots n-1$ do
            for $k \leftarrow j \ldots n-1$ do
                if $i \neq j$ && $j \neq k$ && $i \neq k$ && $A[i] + A[j] == A[k]$ then
                    return true
                end if
            end for
        end for
    end for
    return false
end function
```

d) Give a big-O estimate for the complexity of this algorithm. Is it more efficient than the brute-force algorithm?