COMS 311: Homework 2
Due: Sept 21, 11:59pm
Total Points: 40

Late submission policy. If you submit by Sept 22, 11:59PM (and after the specified deadline), there will be 10% penalty. That is, if your score is x points, then your final score for this homework after the penalty will be $0.9 \times x$. Similarly, if you submit by Sept 23, 11:59PM (after Sept 22, 11:59PM), there will be 20% penalty.

Submission after Sept 23, 11:59PM will not be graded without (prior) explicit permission from the instructors.

Submission format. Your submission file should be in pdf format. Name your submission file: `<Your-net-id>-311-hw2.pdf`. For instance, if your netid is `asterix`, then your submission file will be named `asterix-311-hw2.pdf`.

You are required to type-set your solution. You can use word, latex and any other type-setting tool to type your solution; you need to make sure you export/save it in pdf before submission.
For questions where you are asked to write an algorithm, you are required to write pseudocode. Do not write code in any specific language such as Java, Python, C/C++, etc. You are also required to present the runtime of your algorithm. Part of the grade will depend on the efficiency of your algorithm.

Learning outcomes.

1. Determine whether or not a function is Big-O of another function
2. Analyze asymptotic worst-case time complexity of algorithms
3. Design of algorithms (heap and hashing)

Some Useful (in)equalities.

- $\sum_{i=1}^{n} i = n(n+1)/2$
- $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$
- $2^{\log_2 n} = n, \quad a^{\log_b n} = n^{\log_b a}, \quad n^{n/2} \le n! \le n^n, \log x^a = a \log x.$
- $\log(a \times b) = \log a + \log b, \quad \log(a/b) = \log a - \log b.$
- $a + ar + ar^2 + \cdots + ar^{n-1} = \dfrac{a(r^n - 1)}{(r-1)}$
- $1 + 1/2 + 1/2^2 + \ldots + 1/2^n = 2(1 - 1/2^{n+1})$
- $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1.$

1. Prove or disprove the following statements. (10pts)

   (a) $\log n \in O(n^{1/3})$

   $\log n \leq cn^{1/3}$

   $3 \log n^{1/3} \leq cn^{1/3}$

   $\frac{3}{c} \leq \frac{n^{1/3}}{\log n^{1/3}}$

   Therefore, **there exists** a c such that $c = 1$ **there exists** a n_0 such that $n_0 \geq 2$ where $\forall n \geq n_0 : \log n \leq cn^{1/3}$.

   (b) $f(n) \in O(g(n))$ implies that $2^{f(n)} \in O(2^{g(n)})$

   $2n \in n$ when $c = 2$ and $n_0 = 1$

   $2^{2n} \leq c2^n$

   $2^n \leq c$.

   This is a **contradiction** since n can't be bound by a constant.

   Therefore, $f(n) \in O(g(n))$ DOES NOT imply that $2^{f(n)} \in O(2^{g(n)})$

2. Derive the runtime for the following (in terms of big-O of some function of n). (10pts)

   (a)
```
for (i=1; i<=n; i++) {
  for (j=1; j<=n*i; j++) {
    <Do-something-atomic>
  }
}
```

   $\sum_{i=1}^{n} \sum_{j=1}^{n*i} 1$

   $\sum_{i=1}^{n} n * i$

   $n(1 + 2 + 3 + ... + n)$

   $n\left(\frac{n(n+1)}{2}\right)$

   Answer: $O(n^3)$

   (b)
```
// Given two sorted arrays A and B each containing n integers
i=0; j=0; cnt=0;
while ((i<n) and (j<n)) {
   if (A[i] < B[j]) {
      i++
   }
   if (B[j] < A[i]) {
      j++;
   }
   else {
      i++;
```

```
        j++;
        cnt++;
      }
    }
    return cnt;
```

All of the internal if, else if, and else statements have a runtime complexity of $O(1)$. The while loop runs until either i or j reaches n. Therefore, the worst-case runtime occurs after n iterations of the while loop with $O(1)$ operations within.

Answer: $O(n)$

3. Consider an array A of integers. Write an algorithm that outputs the length of the longest subarray where the sum of the elements in the subarray is equal to 0. For instance, for the array $A = \{13, 11, -2, 1, 7, -15, -2, 3, -3, 10\}$ the output of your algorithm should be 8, for the array $A = \{13, 1, 0, -2\}$ the output should be 1, and the for the array $A = \{13, 1, -2\}$ the output should be 0. Derive the runtime of your algorithm. Part of the grade will depend on the efficiency of your algorithm. Use of hash tables and expected runtime is acceptable. (20pts)

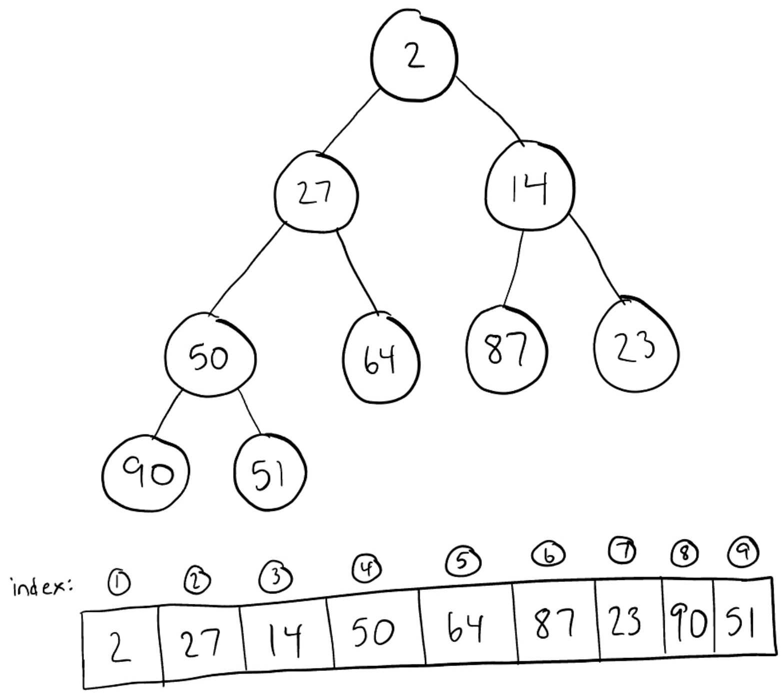
```
largestSubarray(array A){
    Table T<sum, index>
    prefixSums = []
    subarrayLength = 0
    for(i = 1 to n){
        if(i == 1){
            prefixSums[i] = A[i]
        }
        else{
            prefixSums[i] = prefixSums[i - 1] + A[i]
        }
        // We are assuming the hashing starts at 1 (established in for loop)
        if(prefixSums[i] == 0){
            subarrayLength = i
        }
        else if((prefixSums[i] exists in T already at an index < i) AND
        (i - T[prefixSums[i]] + 1 > subarrayLength)){
            subarrayLength = i - T[prefixSums[i]] + 1
        }
        else{
            Add the sum and i to T (sum is the key and i is the value)
        }
    }
    return subarrayLength
}
```

Runtime Analysis:
The hashtable search has an EXPECTED RUNTIME of $O(1)$.
All other functions have a runtime of $O(1)$.
We loop through this n times.
Answer: Expected $O(n)$

4. **Extra Credit** (10pts)

   (a) Draw a binary min-heap tree and the corresponding array where the elements are inserted to the heap in following order:

   50 64 87 90 23 14 2 51 27



   (b) Given an array of integers, write an algorithm to check whether the ordering of integers satisfy the max-heap property.

```
** ASSUMING THE ARRAY STARTS AT INDEX 1 LIKE THE IN-CLASS EXAMPLES **

satisfiesMaxHeap(array A){
    index = size

    while(index > 1){
        if(A[index] > A[index/2]){
            return false
        }
        index--
    }

    return true
}
```

Runtime: $O(n)$