COMS 311: Homework 2Due:
Sept 23, 11:59pm Total
Points: 100

Late submission policy. If you submit by Sept 24, 11:59PM, there will be 20% penalty. That is, if your score is x points, then your final score for this homework after the penalty will be $0.8 \times x$. Submission after Sept 24, 11:59PM will not be graded without explicit permission fromthe instructors.

Submission format. Your submission 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.

If you are planning to write your solution and scan+upload, please make sure the submission is readable (sometimes scanned versions are very difficult to read - poor scan quality to blame).

If you plan to snap pictures of your work and upload, please make sure the generated pdf is readable - in most cases, such submissions are difficult to read and more importantly, you may endup submitting parts of your work.

If you would like to type your work, then one of the options you can explore is latex (Overleaf).

Rules for Algorithm Design Problems.

1. Unless otherwise mentioned, for all algorithm design problems, part of the grade depends onruntime. Better runtime will lead to higher grade.
2. You will write pseudo-code for your algorithm. It should not be some code in a specific programming language (e.g., Java, C/C++, Python).
3. You can use any operation already covered in class-lectures in your solution without explicitly writing the algorithm. However, you need to use them appropriately (i.e., indicate the inputs to and outputs from these algorithms). For instance, you can use heapifyUp with input as the index of a heap element and you do not need to write the code for heapifyUp.

Learning outcomes.

1. Runtime analysis
2. Basic algorithm design (Binary Search, Heaps and Hash)

---

1. Prove or disprove the following statements. Provide a proof for your answers. (30 Points)

(a)$\forall a \geq 1$, $2^{an} \in O(2^n)$.

If $\forall a \geq 1$, $2^{an} \in O(2^n)$, then there exist C and n_0 such that $\forall a \geq 1$, $n_0 \geq 1$: $2^{an} \leq 2^n$ …i
Left hand side: $(2^n)^a$ …ii
Right hand side: 2^n …iii
∵ ii, iii, and $a \geq 1$
$2^{an} \geq 2^n$…iv
∵ iv contradicts i
$\forall a \geq 1, 2^{an} \notin O(2^n)$

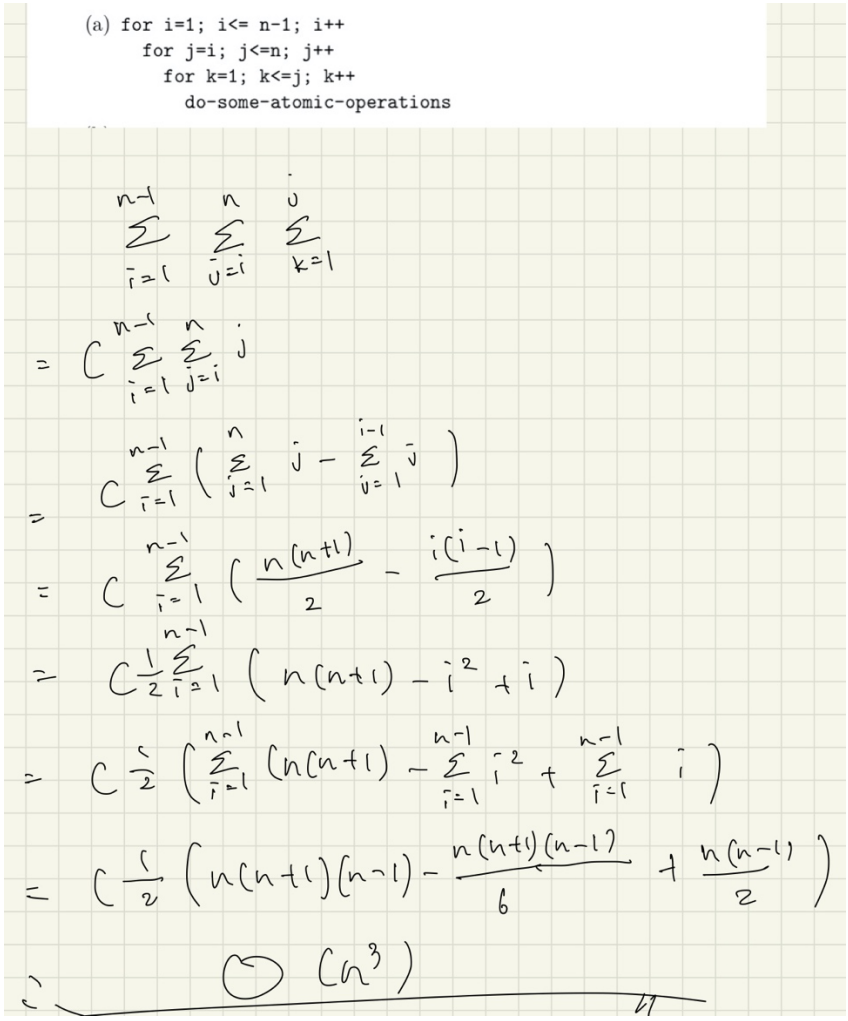
(b)$2^{f(n)} \in O(2^{g(n)})$ implies $f(n) \in O(g(n))$.
If $2^{f(n)} \in O(2^{g(n)})$, then there exist C and n_0 such that $\forall n_0 \geq 1$: $2^{f(n)} \leq 2^{g(n)}$
∵ We only consider nonnegative function
$f(n) \leq g(n)$
∴ there exist C and n_0 such that $\forall n_0 \geq 1$: $f(n) \leq g(n)$
∴ $f(n) \in O(g(n))$
∴$2^{f(n)} \in O(2^{g(n)})$ implies $f(n) \in O(g(n))$.

(c)$f(n) \in O(g(n))$ implies $2^{f(n)} \in O(2^{g(n)})$.

When f(n) = 2n and g(n) = n, then $f(n) \in O(g(n))$.
However,
∵(a)
$2^{an} \notin O(2^n)$
∴$2^{f(n)} \notin O(2^{g(n)})$.

2. Derive the runtime of the following as a function of *n* and determine its Big-O upper bound. You must show the derivation of the end result. Simply stating the final answer without anyderivation steps will result in 0 points. (30 Points)

(a)
```
for i=1; i<= n-1; i++
    for j=i; j<=n; j++
      for k=1; k<=j; k++
        do-some-atomic-operations
```



(b)

```
1: i=0
2:j=n-1
3:while(i<j){
4:      while(a[i]<k)i++;
5:      while(a[j]>k)j--;
6:      if(i<j){
7:              t =a[i];
8:              a[i]=a[j];
9:              a[j]=t;
        }
}
```

Assume a is an array of n integers and k is some integer not present in a.

In line 1 and 2, i is set at the beginning of the array and b is set to the last of the array. Line 6-9 operation is constant time. Line 3's while loop runs as long as i<j. In line 4 and 5, it compares a[i] and a[j] with k and based on it, it increases i and decrease j. Therefore, they only iterate the element in the array at most once.

∴ O(n)

3. Consider an array *A* of 0's and 1's such that the 0's appear in the array before all the 1's. The objective is to find the largest index *i* such that *A*[*i*] = 0. Write an algorithm that takes as input *A* and outputs *i*. Derive the runtime of your algorithm. (20 Points)

```
Int findIndex(A){
        return helper(0, A.length-1,A);
}

helper(start,end, A){
if(start>end) return -1;
mid = (end+start)/2;
if(A[mid] == 0){
        if(A[mid+1] == 1){
        return mid;
        }
        else{
        return helper(mid,end,A);
        }
}
else{
        if(A[mid-1] == 0){
        return mid-1;
        }
        else{
        return helper(start,mid,A);
        }
}
}
```

This algorithm binary0searches and looks for the index of pair of 0 and 1.
∴ O(log(n))

4. Consider the binary min-heap structure implemented as an array. We have reviewed the operations such as heapifyUp, heapifyDown, extractMin and insert. Consider the following operations and write efficient algorithms for them

(a) delete(i): the function deletes the element at the *i*-th index of the array implementing the heap and rearranges the rest of the elements in the array appropriately to maintain the heap property after the delete operation.

```
delete(i){
    t = heap[0];
    heap[0] = heap[i];
    heap[i]= t;
    heap.extractMin();
}
```

First three lines are constant run time and extractMin takes O(log(n)).

∴ O(log(n))

(b) delete(v): if there exists an element whose value is equal to *v* in the array implementing the heap, then the function deletes the element and rearranges the rest of the elements in the array appropriately to maintain the heap property after the delete operation.

```
Delete(v){
    Index = -I;
    for(int i=0;i<heap.length;i++){
        if(heap[i]==v){
                index =i ;
        }
    }
    If(index!=-1){
        t = heap[0];
        heap[0] = heap[index];
        heap[index]= t;
        heap.extractMin();
    }

}
```

The for loop takes O(n) and extractMin takes O(log(n)).

∴ O(n)

[*Think of appropriately using hashing.*] (20 Points)

5. **Extra Credit** Consider an array of integers that is *k-almost sorted*. By *k*-almost sorted, we refer to the situation where the index of any element can be at most *k* indices away from its correct index as per ascending order. For instance, the array

0	1	2	3	4	5	6
2	1	0	6	5	3	4

is 3-almost sorted because every element is at most 3 positions away from its correct index. Write an algorithm that takes as input *A*, the valuation *k* and returns a sorted array. Derive the runtime of your algorithm. (20 Points)

```
int[] sortKAlmostSorted(A, k){
    PriorityQueue queue;
    for(int i = 0; i<k+1;i++) {
        queue.add(A[i]);
    }
    Index = 0;
    for(int i=k+1;i<A.length;i++){
        A[index++]=queue.peek();
        queue.poll();
        queue.add(A[i]);
    }
    Iterator it = queue.iterator();
    while(it.hasNext()){
        A[index++]=queue.peek();
        queue.poll();
    }
    return A;                                                    }
}
```

To create k+1 size heap with priority queue, it takes O(k)

Take min from the heap one by one and place it into "A"and add new element to the heap.

Removing element and adding element takes O(log(k))

$\therefore O(k)+O((n-k)*\log(k)) = O((n-k)*\log(k))$