Homework 2
Com S 311
Due: Sep 11, 11:59PM
Total Points: 100

Late submission policy. If you submit by Sept 12, 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 12, 11:59PM will not be graded without explicit permission from the 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 end up submitting parts of your work.

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

Rules for Algorithm-design Problems.

1. For all algorithm design problems, part of the grade depends on the run-time. Better the run-time, higher the grade.
2. You may use Heap operations such as BuildHeap, HeapifyUp, heapifyDown, add, delete, extractMin/Max in your solutions without describing the procedure.
3. Similarly you can use hash table operations such as build Hash table, add, delete, search without describing them.

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 algorithms (including using Heap and/or Hash-Table)

1. Prove or disprove the following statements. (20 Points)

   (a) $2^{2n} \in O(2^n)$

   (b) $f_1(n) \in O(g_1(n)) \ \wedge \ f_2(n) \in O(g_2(n)) \ \Rightarrow \ f_1(n) \times f_2(n) \in O(g_1(n) \times g_2(n))$

   Ans. Not provided. Please see TA/instructor in office hours

2. Derive runtime for (20 Points)

   (a)
```
for i in the range [1, n-1]
    for j in the range [i, n]
      for k in the range [1, j]
         do-something-atomic
```
   Ans. Not provided. Please see TA/instructor in office hours

   (b)
```
i=0
j=n
while (i<=j)
  while (a[i] < k)  i++;
  while (a[j] > k)  j--;
  if (i<=j)
     t = a[i];
     a[i] = a[j];
     a[j] = t;
```


   Ans. Assumption is that k is not in the array. Consider the outer while loop. During each iteration, at least one of the following must happen: i) i is incremented, ii)j is decremented, iii) $a[i]$ is swapped in $a[j]$. Let $x = a[i]$ and $y = a[j]$. Suppose that in an iteration i) and ii) do not happen. This is because $x = a[j] > k$, and $y = a[i] < k$. Since the swap operation is performed, we have that $a[i] = y$ and $a[j] = x$. Since $y < k$, we have $a[i] < k$, thus in the next iteration, i must be incremented, and similarly since $x > k$ and $a[j] = x$, in the next iteration, j must be decremented. Thus every time a swap operation is performed, i is incremented or j is decremented (or both could happen). Since i can be incremented at most n times and j can be decremented at most n times, we can conclude that the swap operation is performed at most $2n$ times. Thus the total number of operations are bounded by number of times i is incremented, plus number of times j is decremented plus number of times swap operation is performed. Time taken to increment i, decrement j and swap is a constant. Thus the total time can be at most cn (Time to increment i) + cn (time to decrement j) + $2cn$ (time for swap). Thus the runtime is $O(n)$.

3. Design algorithms for the following problems (60 Points)

   (a) Given a sorted array A of integers and an integer T, the goal is to determine if there exists two numbers in the array that sum to T. Write an algorithm that takes A as input and outputs true or false. *You are not allowed to use hash-tables in this algorithm.* Derive the runtime of your algorithm.

   Ans. Consider the following algorithm.

```
Input: Array a, Integer T.
left = 1;
right = length of the array;
while (left <=right){

  x = a[left] + a[right];

  if (x==T)
        return true;
  if (x <  T)
     left++;
  if (x > T)
     right--;

}

return false;
```

Runtime: Consider an iteration of the while loop. One of the following three will happen: left is incremented by one, right is decremented by 1 or the loop quits. The worst-case time when the loop does not quit because x is never equal to T. So we consider this worst-case and put a bound on number of times the while loop can be done. Left starts at 1 and right starts at n. Suppose that left is incremented a times and right is decremented b times and the loop condition does not hold and thus the loop is terminated. Now the value of left is $a+1$ and the value of right is $n-b$. Note that if $a+b > n-1$, then it must be the case that $a+1 > n-b$ and the loop is terminated. Thus when $a+b$ reaches n, the loop is terminated. Since during each iteration either $left$ is incremented or $right$ is decremented, it must be the case that the loop is performed exactly $a+b$ times. Thus the loop is performed at most n time. Each iteration of the loop takes constant time. Thus the time taken is $O(n)$.

(b) Given an array A consisting of integers, the *goal* is to construct a two-dimensional array B such that $B[i,j] = \sum_{k=i}^{k=j} A[k]$. Write an algorithm that takes A as input and outputs B. Derive the runtime of your algorithm.

Ans. Consider the following algorithm.

for i from 1 to $n-1$ **do**
  $D[i, i+1] \leftarrow a[i] + a[i+1]$
  for j from $i+2$ to n **do**
    $D[i,j] \leftarrow D[i, j-1] + a[j]$
  end for
end for

Correctness: $D[i, i+1] = a[i] + a[i+1]$, as it should. Assuming that $D[i, j-1] = \sum_{k=i}^{j-1} a[k]$, the algorithm will set $D[i,j]$ to $\sum_{k=i}^{j-1} a[k] + a[j] = \sum_{k=i}^{j} a[k]$.

Runtime: The innermost for loop runs $n-i-1$ times, and each time it performs two constant-time operations. Call the time taken by them c. The outermost for loop

performs another constant-time operation taking d time, before entering the inner loop. The total time is thus given by the sum

$$\sum_{i=1}^{n-1}\left(d+\sum_{j=i+1}^{n} c\right)$$

Again, we break up the inner sum into $\sum_{j=1}^{n} c - \sum_{j=1}^{i} c$, which simplifies to $nc - ic$. The outer sum is thus

$$\begin{aligned}
&\sum_{i=1}^{n-1}(d+nc-ic)\\
=&(n-1)d+(n-1)nc-\frac{(n-1)n}{2}c\\
=&(n-1)d+\frac{(n-1)n}{2}c\\
&\in O(n^2)
\end{aligned}$$

(c) Given an array A containing 0s and 1s, such that all the 0s appear in the array before all the 1s. The *goal* is to find the smallest index i such that $A[i] = 1$. Write an algorithm that takes as input A and outputs i. Derive the runtime of your algorithm.

Ans. We modify the binary-search algorithm.

i. $left = 0, right = n-1$

ii. while $(left \leq right)$

- $mid = \frac{left+right}{2}$.
- If $A[mid] = 1$ and $a[mid-1] = 0$, return mid.
- if $A[mid] = 1$ and $a[mid-1] = 1$, $right = mid$.
- if $A[mid] = 0$, $left = mid$.

iii. Return false.

Runtime. The worst-case time is when the loop does not terminate because, it found mid. At the start of an iteration let x be the value of $left$ and y be the value of $right$. Thus $right - left = y - x$. After the iteration, there are two possibilities: $right = \frac{x+y}{2}$ or $left = \frac{x+y}{2}$. In both cases $right - left$ is $(y-x)/2$. Thus after each iteration, the difference between right and left is halved. Initial difference between them is n. Thus after $O(\log n)$ iterations, $left$ will be higher than $right$ and the loop terminates. Each iteration of the loop takes $O(1)$ time, thus the time is $O(\log n)$.

(d) A binary Heap is implemented using array. There are several standard operations that are defined over such a heap structure:

`getMin`: parameters include array. Runtime: $O(1)$.

`add`: parameters include array, value of the element being added, the number of elements in the heap. Runtime: $O(\log n)$

`delete`: parameters include array, index of the element being deleted, the number of elements in the heap. Runtime $O(\log n)$

Notice that the `delete` operations allows you to delete element at a specific index. The *goal* is add a new operation `deleteValue(x)`, where the parameter `x` denotes the value of the element(s) that needs to be deleted from the heap, and ensure that the operation can be done in *expected* $O(\log n)$ time, where n is the number of elements in the heap.

Augment of the heap structure (i.e., add additional information/property to the heap structure) which will allow you to design the algorithm for `deleteValue(x)` with specified runtime, *without altering the runtime for the rest of the standard operations.*

If your augmentation of the heap structure requires changes in the way the existing standard operations are implemented, then you need to mention those changes.

Ans. We will augment the basic heap data structure H with a hash table T, consisting of $\langle key, value\rangle$ pairs, while maintaining the following property: If $H[i] = v$ if and only if $\langle v, i\rangle$ belongs to T. We also maintain a counter to indicate the number of elements stored in the heap. Note that we can do operations such as $add(key)$, $delete(key)$ and $search(key)$ operations on the hash table in expected $O(1)$ time. Modify the $search(key)$ method so that it returns i such that $\langle key, i\rangle \in T$. If there is no such i, then this method returns -1. Note that this modified method can also be done in expected $O(1)$ time.

We now modify the $HeapifyUp$ and $HeapifyDown$ operations of the heap so that the above mentioned property is maintained. We describe $HeapifyUp(i, y)$: The goal of this method is to replace $H[i]$ with y (if $y < H[i]$) and maintain the heap property. Set $H[i] = y$. If i equals 1, then $H[1]$ is y and the heap property is maintained (since $y < H[1]$). Let p be the index of the parent of i in the heap. Compute the minimum among $H[p], H[2p]$ and $H[2p + 1]$. Note that i is either $2p$ or $2p + 1$. Note that the minimum must be one of $H[p]$ or $H[i]$. If the minimum is $H[p]$, then the procedure stops. If the minimum is $H[i]$, then we we replace $H[i]$ with $H[p]$, delete $\langle H[p], p\rangle$ from the hash table and add $\langle H[p], i\rangle$ to the hash table. And call $HeapifyUp(p, y)$. Similarly we can define $HeapifyDown(i, y)$ operation. Note that both the the additional overhead with respect to standard heap operations is removing and adding element from the hash table. With each heap operation, we do one add operation and one delete operation on hash table. Since the hash table operations take expected $O(1)$ time, this will increase the total time by a factor of $O(1)$. Thus the time taken for these operations is expected $O(\log n)$ time. These operations can be used to add and element to the heap or extract minimum element from the heap.

We are now ready to describe the methods:

$Add(v)$. Place $\langle v, counter + 1\rangle$ in T and add v to the binary heap (by using the above described methods $HeapifyUp$ and $HeapifyDown$. Increment counter by 1. The time taken is $O(\log n)$.

$DeleteValue(v)$: Call $search(v)$, if this returns i, then remove $\langle v, i\rangle$ from the hash table and delete $H[i]$ from the heap by using above described methods $HeapifyUp$ and $HeapifyDown$. The time taken is $O(\log n)$.

$min()$. Returns $H[1]$. Time taken is $O(1)$.

$extractMin()$. Remove $\langle H[1], 1\rangle$ from the hash table. Decrement counter by 1. Perform extract Min operation on H (by using described methods $HeapifyUp$ and $HeapifyDown$. Time taken is $O(\log n)$.