CS 32 Week 7 Solutions

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.

Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

If you have any questions or concerns please go to any of the LA office hours.

Concepts

Algorithm Analysis, Sorting

1. What is the time complexity of the following code?

```
int randomSum(int n) {
  int sum = 0;
  for(int i = 0; i < n; i++) { 1^2 + 2^2 + … + (n-1)^2 is
O(n^3)
    for(int j = 0; j < i; j++) { O(i^2)
      if(rand() % 2 == 1) {
        sum += 1;
      }
      for(int k = 0; k < j*i; k += j) { O(i)
        if(rand() % 2 == 2) {
          sum += 1;
        }
      }
    }
  }
  return sum;
}
```

Time complexity: O(n^3).
Note that it doesn't matter that "if(rand() %2 == 2)" is always false, because rand is still run each time.

2. FInd the time complexity of the following code:

```
int operationFoo(int n, int m, int w) {
  int res = 0;

  for (int i = 0; i < n; ++i) { // the outer loop runs n
iterations
    for (int j = m; j > 0; j /= 2) { // every time this loop is
entered, this loop runs log2(m) iterations
      for (int jj = 0; jj < 50; jj++) { // every time this loop
is entered, it runs 50 iterations, which is constant, or O(1)
        for (int k = w; k > 0; k -= 3) { // every time this
loop  is entered, it runs ⅓ * w iterations, or O(w)
          res += i*j + k; // this line runs in total n*log(m)*w
        }
      }
    }
  }
  return res;
}
```

Time complexity: O(n * log(m) * w).

3. Binary search is an efficient algorithm finding if an element (x) exists in the input array (arr). Find the time complexity of the following function

   Hint: try to trace through the code to find the mechanism of this algorithm, then consider what will be the worst case

```
// At start, left = 0 and right = length of array - 1
int binarySearch(int arr[], int left, int right, int x)
{
    while (left <= right) { // Iteration stops when left >
right
        int middle = left + (right - left) / 2;

        // If the exact match is not found
        // Either left or right will be assigned value of
middle
        // So next time in iteration, the new left and right
pair
        // interval is half of the original interval
        if (arr[middle] == x)
            return middle;
```

```
        else if (arr[middle] < x)
            left = middle + 1;
        else
            right = middle - 1;
    }
    return -1;
}
int main()
{
    int arr[] = {2, 3, 4, 10, 40, 60, 80};
    int x = 60;
    int index = binarySearch(arr, 0, 6, x);
    if (index == -1) {
        cout << x << " doesn't exist in array." << endl;
    } else {
        cout << x << " is at " << index << " position." <<
endl;
    }
}
```

Time complexity: O(log(n)).
Binary search **halves** the search interval after every iteration; it either traverses through the right half of the original interval or the left half.

4. Find the time complexity of the following function

```
int obfuscate(int a, int b) {
     vector<int> v;
     set<int> s;
     for (int i = 0; i < a; i++) {
          v.push_back(i);
          s.insert(i); // each insert is O(log(size of set))
     }
     v.clear();

     int total = 0;
     if (!s.empty()) {
          for (int x = a; x < b; x++) {
               for (int y = b; y > 0; y--) {
                    total += (x + y);
               }
          }
     }
```

```
        return v.size() + s.size() + total;
    }
```

Time complexity: O(a log(a) + b(b-a)).

5. Consider this function that returns whether or not an integer is a prime number:

```
bool isPrime(int n) {
  if (n < 2 || n % 2 == 0) return false;
  if (n == 2) return true;
  for (int i = 3; (i * i) <= n; i += 2) {
    if (n % i == 0) return false;
  }
  return true;
}
```

What is its time complexity?

Time complexity: O($\sqrt{n}$).
The function's loop only runs while i <= $\sqrt{n}$, and the square root is not the same as a constant multiple of n, so we include it in our Big-O analysis.

6. Fill out the following table:

Time complexity	Doubly linked list (given head)	Array/vector
Inserting an element to the beginning	O(1)	O(n)
Inserting an element to some position i	O(n)	O(n)
Getting the value of an element at position i	O(n)	O(1)
Changing the value of an element at position i	O(n)	O(1)
Deleting an element given a reference to it	O(1)	O(n)

7. Write a function for which, given a vector of words and a character, returns the number of times that character is present in the entire vector. Then, find the time complexity of your algorithm.

```
int countNumOccurrences(const vector<string>& words, char c);
```

Note: When calculating the time complexity, you can consider the size of the vector as N and the average length of one word is K.

```
int countNumOccurrences(const vector<string>& words, char c) {
      int count = 0;
      for (vector<string>::const_iterator it = words.begin(); it !=
words.end(); it++) {
            const string& word = *it;
            for(int i = 0; i < word.size(); ++i) {
                  if (word[i] == c) {
                        ++count;
                  }
            }
      }
      return count;
}
```

Time complexity: O(N * K).

Note: it doesn't matter if your code isn't exactly the same, but at a high level, your algorithm probably should be the same: for every word, look through each character in the word, compare with char c and add it to the count. The time complexity of this algorithm is O(N*K), or in other words, the total number of characters in the vector.

8. Given an array of *n* integers, where each integer is guaranteed to be between 1 and 100 (inclusive) and duplicates are allowed, write a function to sort the array in O(*n*) time.
(Hint: the key to getting a sort faster than O(*n log n*) is to avoid directly comparing elements of the array!)

```
void sort(int a[], int n);
```

```
void sort(int a[], int n) {
  int counts[100] = {};  // Count occurrences of each integer.
  for (int i = 0; i < n; i++)
```

```
    counts[a[i] - 1]++;

  // Add that many of each integer to the array in order.
  int j = 0;
  for (int i = 0; i < 100; i++)
    for (; counts[i] > 0; counts[i]--)
      a[j++] = i + 1;
}
```

9. Here are the elements of an array after each of the first few passes of a sorting algorithm discussed in class. Which sorting algorithm is it?

3 7 4 9 5 2 6 1
3 7 4 9 5 2 6 1
3 **7** 4 9 5 2 6 1
3 **4** 7 9 5 2 6 1
3 4 7 **9** 5 2 6 1
3 4 **5** 7 9 2 6 1
2 3 4 5 7 9 6 1
2 3 4 5 **6** 7 9 1
1 2 3 4 5 6 7 9

   a. bubble sort
   b. insertion sort
   c. quicksort with the pivot always being chosen as the first element
   d. quicksort with the pivot always being chosen as the last element

Notice that the section to the left of the underlined digit is in sorted order. One of the most prominent insertion sort characteristics is having a section that is already in sorted order and continue to put the current element in the right place.

10. Given the following vectors of integers and sorting algorithms, write down what the vector will look like after 3 iterations or steps and whether it has been perfectly sorted.

   a. `{45, 3, 21, 6, 8, 10, 12, 15}` insertion sort (first step is at a[1])
   b. `{5, 1, 2, 4, 8}` bubble sort (consider the array after 3 “passes” and after 3 “swaps”)

c. {-4, 19, 8, 2, -44, 3, 1, 0} quicksort (where pivot is always the last element)

a. {3, 6, 21, 45, 8, 10, 12, 15} (assume first step starts by sorting a[1] since a[0] is trivial) not perfectly sorted

b. {1, 2, 4, 5, 8} perfectly sorted, within 3 "steps" it does not know it's complete, but within 2 "passes" it will

c. 1st iteration -> {-4, -44, **0**, 2, 19, 3, 1, 8}
-4 and -44 might be in a different order
2, 19, 3, 1, 8 might be in a different order
2nd iteration, if starting from the order shown after the 1st iteration, and left part is sorted before right part -> {**-44**, -4, 0, 2, 19, 3, 1, 8}
l3rd iteration, if starting from the order shown after the 2nd iteration, and left part is sorted before right part -> {-44, -4, 0, 2, 1, 3, **8**, 19}
2, 1, 3 might be in a different order
Not perfectly sorted
Next recurses into [2,1,3] and [19]