# CS 32 Week 7 Worksheet

**Concepts**: Algorithm Analysis, Sorting

### Algorithmic Analysis Problems

1. (5 min) What's the time complexity of the following function? 🔍

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

Solution:

```
Time complexity: O(n^3).

For the innermost loop, even though the condition is k < j*i, each
increment is k += j, so the loop runs exactly i times, making it O(i). The
middle loop runs i times, each time doing something O(i), so it's O(i^2).
The outer loop does something O(i^2) times for each i up to n, so is O(1^2
+ 2^2 + 3^2 + ... + (n-1)^2), which gives the overall complexity of O(n^3).

Note: For the innermost for loop, it doesn’t matter that `if(rand() %2 ==
2)` is always false, because rand is still run each time, meaning the
innermost loop will still take O(N) time since rand() runs in O(1) time.
```

2. (5 min) Nice! 😄 Now, what's the time complexity of this function?

```
int operationFoo(int n, int m, int w) {
    int res = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = m; j > 0; j /= 2) {
            for (int jj = 0; jj < 50; jj++) {
                for (int k = w; k > 0; k -= 3) {
                    res += i*j + k;
                }
            }
        }
    }
    return res;
}
```

Solution:

Let's take a look at the following annotated code:

```
  int res = 0;
  for (int i = 0; i < n; ++i) { // (1)
      for (int j = m; j > 0; j /= 2) { // (2)
          for (int jj = 0; jj < 50; jj++) { // (3)
              for (int k = w; k > 0; k -= 3) { // (4)
                  res += i*j + k; // (5)
              }
          }
      }
  }
  return res;
```

(1) *The outer loop runs n iterations*
(2) *Every time this loop is entered, this loop runs log2(m) iterations*
(3) *This loop always runs 50 iterations, which is constant, or O(1)*
(4) *Every time this loop is entered, it runs w/3 iterations, or O(w)*
(5) *This line runs in total n*log(m)*w*

Therefore, the overall time complexity is: O(n * log(m) * w)

3. [Binary search](#) 🤓 is an efficient algorithm finding if an element (x) exists in a sorted array. What's the time complexity of the following function? *Hint: try tracing through to find the mechanism of this algorithm, then consider what will be the worst case!*

```
int binarySearch(int arr[], int left, int right, int x)
{
    while (left <= right) {
        int middle = left + (right - left) / 2;

        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;
    }
}
```

Solution:

```
Let's take a look at the following annotated binarySearch() function:

// 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;
```

Binary search halves the search interval after every iteration; it either traverses through the right half of the original interval or the left half.

Therefore, the overall time complexity is **O(log(n))**, as the worst case scenario involves us having to keep halving the array until there only remains 1 element. This involves log2(n) halves, since 2^{log2(n)} = n.

4. Just a few more to go! 😁 What's the time complexity of the following code?

```
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);
	}
	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;
}
```

Solution:

```
We see in the line `s.insert(i);`, each insert is O(log(size of set)).

Thus, the for loop will execute in O(log 1 + log 2 + ... + log a-1) time.
Mathematically, we see that log 1 + log 2 + ... log a-1 = log 1*2*...*(a-1)
= log (a-1)! (where the ! denotes factorial). By Stirling's approximation,
log (a-1)! is approximately equal to (a-1) log (a-1), which is O(a log a).

The next for loop will take O(b(b-a)) time since the outer loop has b - a
iterations (from x = a to x = b) and the inner loop has b iterations.

Therefore, the overall time complexity is the sum, O(a log(a) + b(b-a)).
```

5. So far, so good! 😅 How about this function, what's its time complexity?

```
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;
}
```

Solution:

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. Great job! Now, let's switch things up... 😏 For each operation (row), fill out the time complexities of performing that action using the given data structure (col).

| **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. Neat! Let's try some code now. 😎 Write a function which, given a vector of words and a character, returns the number of times that character is present in the vector.

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

Then, find the time complexity of your algorithm! *Note: When calculating the time complexity, let the size of the vector be N and the average length of one word be K.*

Solution:

```
int countNumOccurrences(const vector<string>& words, char c) {
    int count = 0;
    // Going through all the words in the string
    for (vector<string>::const_iterator it = words.begin();
         it != words.end(); it++) {

        const string& word = *it;
        // Going through all the letters in the word
        for(int i = 0; i < word.size(); ++i) {
            if (word[i] == c) {
                ++count;
            }
        }
    }
    return count;
}
```

Time complexity: **O(N * K).**

Note: It's more than okay 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 it with char c and add it to the count*. The time complexity of this algorithm is the number of words (N) times the average number of characters per work (K). This gives us O(N*K), or in other words, the total number of characters in the vector.

**Sorting Problems**

8. Last code writing question! 🙏 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);
```

Solution:

```
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;
      j++;
    }
}
```

This solution leverages the fact that **each integer is guaranteed to be between 1 and 100 (inclusive)**! So, we just create a frequency-counting array `counts` of size 100, and store the count of each number in there. `counts[i]` tells us how many times we've seen the number i+1. Once we've populated this, we go back through `counts` and add the corresponding number of each number into the original array `a`.

9. Cooldown with some MCQ! 🥰 Here are the elements of an array after each of the first few passes of a sorting algorithm. Which of the four sorting algorithms is it?

<u>3</u> 7 4 9 5 2 6 1

**3** <u>7</u> 4 9 5 2 6 1

3 **7** <u>4</u> 9 5 2 6 1

3 **4** 7 <u>9</u> 5 2 6 1

3 4 7 **9** <u>5</u> 2 6 1

3 4 **5** 7 9 <u>2</u> 6 1

**2** 3 4 5 7 9 <u>6</u> 1

2 3 4 5 **6** 7 9 <u>1</u>

**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

Solution:

```
(B) Insertion Sort. 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
continuing to put the current element in the right place.
```

10. And voila, last question! 🥳 For each of the following cases: given the vectors of integers and sorting algorithm, write down what the vector will look like after 3 iterations or steps and state whether the vector has been perfectly sorted.

A. **Applying Insertion Sort** on: `{45, 3, 21, 6, 8, 10, 12, 15}`

   *(Note: 1st step starts at comparing `a[1]`)*

   Solution (A):

   **Not perfectly sorted** after 3 steps: **{3, 6, 21, 45, 8, 10, 12, 15}**
   (Assume first step starts by sorting a[1] since a[0] is trivial)

B. **Applying Bubble Sort** on: `{5, 1, 2, 4, 8}`

   *(Note: Consider the array after 3 "passes" and after 3 "swaps." Do the results differ? Does the algorithm know when it's "done" in either case?)*

   Solution (B):

   **Perfectly sorted** after 3 steps: **{1, 2, 4, 5, 8}**. Within 3 "steps" it does not know it's complete, but within 2 "passes" it will

C. **Applying Quicksort** on: `{-4, 19, 8, 2, -44, 3, 1, 0}`

   *(Note: in this case, the pivot is always the last element)*

   Solution (C):

   **Not perfectly sorted.** Let's examine each step of the sort:

   - 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 -> {-44, -4, 0, 2, 19, 3, 1, 8}
     - Assuming we're starting from the order shown after the 1st iteration, and left part is sorted before right part
   - 3rd iteration -> {-44, -4, 0, 2, 1, 3, 8, 19}
     - Assuming we're starting from the order shown after the 2nd iteration, and left part is sorted before right part
     - 2, 1, 3 might be in a different order

   Next, the sort would recurse into [2,1,3] and [19]...