Section Handout #5

Problem 1: Big-O
For each of the following functions, determine its computational complexity expressed in Big-O notation. Briefly justify your answer.

a)

```
int Mystery1 (int n)
{
  int sum = 0;
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < i; j++)
    {
      sum += i * j;
    }
  }
  return (sum);
}
```

b)

```
int Mystery2 (int n)
{
  int sum = 0;
  for (int i = 0; i < 10; i++)
  {
    for (int j = 0; j < i; j++)
    {
      sum += j * n;
    }
  }
  return (sum);
}
```

c)

```
int Mystery3 ( int n )
{
  if( n <= 1 ) return 1;
  return (Mystery3( n / 2 ) + 1);
}
```

d) Determine the complexity of function **`Mystery4`**:

```
int Mystery4(int n)
{
  return Pinky(0, n);
}

int Mystery5(int a, int b)
{
  if (a == b)
```

```
  {
    return 0;
  }
  else
  {
    return Pinky(a+1, b) + Pinky(a, b-1);
  }
}
```

Problem 2: Searching and Sorting

You have a data set of size n that's currently unsorted. You're trying to decide whether it will be worthwhile to sort the data before performing repeated searches, to take advantage of binary search. If you use selection sort to sort the data, how many binary searches would you need to perform to "buy back" the cost that went into sorting your data if:

a.) n is equal to 16 (2^4)
b.) n is equal to 1024 (2^{10})

What if you use merge sort instead of selection sort?

Problem 3: Those Big-O Constant Factors

The formal definition of big-O states that a function t(N) is big-O of f(N) if there exists a constant N_0 and a positive constant C, such that for every value of $N \geq N_0$, $t(N) \leq C * f(N)$. Big-O analysis helps us to reason about the runtime of an algorithm as the input size grows, but sometimes, due to these constant factors, a function that has a worse big-O runtime can outperform one with a better big-O runtime. This question hopes to show some of these issues.

Say you are trying to choose whether to use selection sort or merge sort to sort some amount of data. As we saw in class, selection sort has a runtime of $O(N^2)$ while merge sort has a runtime of O(N log N). However, because merge sort requires us to create temporary data structures to copy values in to and out of during the sorting, merge sort has higher constant factors than selection sort and for certain input sizes these can outweigh the advantage of being N log N rather than N^2. Assume selection sort has a constant factor of 10 (that is, the time to run selection sort is less than or equal to $10 * N^2$) and assume merge sort has a constant factor of 100.

a.) Which algorithm would you choose if you needed to sort 50 items?
b.) Which algorithm would you choose if you needed to sort 100 items?

Based on the above, if you were writing a sorting function and could only use merge sort or selection sort (or a hybrid approach), what would you do to make it as fast as possible?

Problem 4: Search Algorithms

For Assignment 2, you implemented a breadth-first search algorithm for maze solving. To refresh your memory, breadth-first search examines possibilities in order of their

distance from the start point. In this case, that means it examines paths in order of their length. An alternative search method is depth-first search, where each path is explored fully before another is attempted. When a solution is found, it is immediately returned. An example implementation of this search method for the maze problem can be found in chapter 6.

Compare and contrast the two search methods as applied to the problem of solving a (not necessarily perfect) maze. Is one faster than the other? Does one use more memory? What are their Big O values based on the maze dimensions, l and w (remember that Big O bounds the worst case)? Do the two algorithms necessarily return the same paths? If the two algorithms sometimes return different paths, sketch an example maze where they return different paths.

Problem 5: Algorithmic Problem Solving

(Adapted from a problem from *Programming Pearls*, by Jon Bentley)

Suppose that you have a Vector of integers and you would like to find the maximum sum of any its subvectors (a subvector is some contiguous set of numbers in the Vector). That is, you want to figure out what is the maximum sum you can get using a sequence of consecutive numbers from the Vector. Note that since the numbers can be positive or negative, this is not simply the entire Vector. For example, if the Vector contained [2, -5, 12, 9, -3, 10], the largest sum is found in the subvector [12, 9, -3, 10] for a total sum of 28. There are a number of different ways to solve this problem of different computational complexities.

a) Using pseudocode, describe an O(N^3) algorithm that works by summing every subvector to identify and return the maximum sum.

b) Summing each subvector does a lot of redundant work. For example, if your Vector contains [2, -5, 12, 9, -3, 10], the subvectors [-5, 12] and [-5, 12, 9, -3] both require summing -5 and 12. How can you change the algorithm from part a to reuse computation rather than repeating it? Describe the new version of the algorithm using pseudocode. The resulting algorithm should be O(N^2)

c) Now consider solving the problem using a divide-and-conquer approach that divides the vector into two halves to be handled recursively. How are result(s) from each half combined to solve the larger problem? What is the base case for this algorithm? You should describe the algorithm using pseudocode. The algorithm should run in O(NlogN) time.

d) While the previous algorithm is pretty good, there is an extremely clever way to solve the problem which only uses O(N) time. This means it only examines each element in the vector some constant number of times, in this case one time. Describe this algorithm using pseudocode. As a hint, consider the following case: suppose you have determined what is the greatest sum of any subvector in an existing vector. Then you add a single element to the end of the vector. If this changes what the subvector with the greates sum is, what must be true about the new greatest subvector? What extra piece of information

(besides the current greatest sum and the value of the new element) would you need to immediately determine whether the new element changes the greatest subvector? Now consider processing the Vector from left to right. How is this like the situation just described?

e) Finish the following table of run times of the previously described algorithms based on the big-O time of the algorithms:

Algorithm	O(N^3)	O(N^2)	O(NlogN)	O(N)
Size of N				
100	3 seconds	.1 seconds	.03 seconds	.003 seconds
200				
1000				
10000				