Data Structures & Algorithms 2
Lab 2
Algorithm Complexity

Exercise 1

For each of the six program fragments in Figure 1 :

1. Give an analysis of the running time (Big-O)
2. Implement the code and give the running time for several values of N
3. Compare your analysis with the actual running times

```
(1)  sum = 0;
     for( i = 0; i < n; ++i )
         ++sum;
(2)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < n; ++j )
             ++sum;
(3)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < n * n; ++j )
             ++sum;
(4)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < i; ++j )
             ++sum;
(5)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < i * i; ++j )
             for( k = 0; k < j; ++k )
                 ++sum;
(6)  sum = 0;
     for( i = 1; i < n; ++i )
         for( j = 1; j < i * i; ++j )
             if( j % i == 0 )
                 for( k = 0; k < j; ++k )
                     ++sum;
```

Figure 1

Exercise 2

1. Write a program to determine if a positive integer N is prime.
2. In terms of N, what is the worst-case running time of your program?
(you should do this in O($\sqrt{N}$)

Exercise 3

Given an array A of integers with $A_1 < A_2 < A_3 < \ldots < A_N$, write an algorithm to determine if there is in the array an integer i such that $A_i = i$. What is the running time of your algorithm?

Exercise 4

Suppose you need to generate a random permutation of the first N integers. For example, {4, 3, 1,5, 2} and {3, 1, 4, 2, 5} are legal permutations, but {5, 4, 1, 2, 1} is not because one number (1) is duplicated and another (3) is missing. This routine is often used in simulation of algorithms.

We assume the existence of a random number generator r with a method randInt(i, j) that generates integers between i and j with equal probability. Here are three different algorithms:

1. Fill the array A from A[0] to A[N − 1] as follows: To fill A[i], generate random numbers until you get one that is not already in A[0], A[1], . . . , A[i − 1].

2. Same as the first algorithm, but keep an extra array called *used*. When a random number ran is first put in the array A, set used[ran] = true. This means that when filling A[i] with a random number, you can see in one step whether the random number has been used, instead of the possibly i steps in the first algorithm.

3. Fill the array such that A[i] = i + 1. Then for(i = 1;i < n; + + i)
swap(A[i], A[randInt(0, i)]);

Now, answer the following questions:

- Prove that the three algorithms generate only legal permutations and that all permutations are equally likely.
- Give as accurate (Big-O) analysis as you can of the expected running time of each algorithm.
- Write separate programs to execute each algorithm 10 times to get a good average. Run the first program for N = 250, 500, 1,000, 2,000; the second program for N = 25.000, 50.000, 100.000, 200.000, 400.000, 800.000; and the third program for N = 100.000, 200.000, 400.000, 800.000, 1.600.000, 3.200.000, 6.400.000.
- Compare your analysis with the actual running times.
- What is the worst-case running time of each algorithm?

Exercise 5

A majority element in an array A of size N is an element that appears more than N/2 times (thus, there is at most one). For example, the array [3, 3, 4, 2, 4, 4, 2, 4, 4] has a majority element 4, whereas the array [3, 3, 4, 2, 4, 4, 2, 4] does not.

If there is no majority element, your programshould indicate this. Here is a sketch of an algorithm to solve the problem:

”First, a candidate majority element is found (this is the harder part). This candidate is the only element that could possibly be the majority element. The second step determines if this candidate is actually the majority. This is just a sequential search through the array. To find a candidate in the array A, form a second array B. Then compare A1 and A2. If they are equal, add one of these to B; otherwise do nothing. Then compare A3 and A4. Again, if they are equal, add one of these to B; otherwise do nothing. Continue in this fashion until the entire array is read. Then recursively find a candidate for B; this is the candidate for A, why?”.

1. How does the recursion terminate?
2. How is the case where N is odd handled?
3. What is the running time of the algorithm?
4. How can we avoid using an extra array B?
5. Write a program to compute the majority element.