CMPSCI 187 (Fall 2018) Lab 04: Big-O

The lab is due by 5:00 pm today. Please make sure that you complete your lab assignment in time and submit it to Gradescope by the deadline time.

- Go to ***File -> Make a Copy*** to make an editable copy of this Google Doc for your account
- Follow the instructions below to complete the lab
- When you are done, go to ***File -> Download As -> PDF Document***
- Log in to Gradescope and submit your PDF. Remember to submit to **Lab 04**, **NOT Project**!

In this lab, you will first analyze each algorithm and derive the Big-O cost on paper, then check the experimental results to verify them. These are good exercises to train your understanding of an algorithm's time complexity analytically, then verify your predictions through experiments.

For each question, first write down the Big-O cost by carefully analyzing the algorithm. Remember, during exams, you won't have access to a computer, so this part of the exercise is extremely important. The better you can learn to analyze the algorithm cost, the more successful you will be at the exam.

After writing down the Big-O cost, run the starter code to check whether your prediction is correct. For each experiment, try a few different values of N as suggested in the results table, and write down the cost for each N. In the starter code, the cost counts the number of instructions in the innermost loop, so the results are consistent no matter how fast or slow your computer is. (In practice, we can also directly measure the program's running time, but as a lot of factors can influence a program's actual running time, and these factors are not always predictable, the results won't be as consistent as counting the instructions). Finally write down 1 to 2 sentences to explain the experimental results. For example, you can say something like

- When N doubles, the cost also doubles, therefore the Big-O cost is O(N)
- When N doubles, the cost becomes four times as large (alternatively, say the cost quadruples), therefore the Big-O cost is $O(N^2)$.

Method 1 (Print Squares):

```
for(long i=0; i<N; i+=2) {
      System.out.println(i+"^2="+(i*i));
}
```

[1 point]:

The Big-O cost of method 1 is: O(N)

Experimental Results [1 point]:

N=	128	256	512	1024
cost=	64	128	256	512

Based on the experimental results, write down 1-2 sentences to justify the Big-O cost: [1 points]

As N doubles so does cost, the algorithm is linear and is dependent on N/2 to be specific. The notation used in this case would be O(N)

Method 2 (Print Savings):

```
for(double i=1.0; i<=N; i=i+(i/10.0)) {
      System.out.printf("Year %d, $%.2f\n", count, i);
}
```

[2 points]:

The Big-O cost of method 2 is: $O(\log_{1.1} N)$

Experimental Results [1 point]:

N=	256	512	1024	2048
cost=	59	66	73	80

Based on the experimental results, write down 1-2 sentences to justify the Big-O cost: [2 points]

By analyzing the algorithm, we can see that i increases by 10% every iteration, and if we check $\log_{1.1}256$ it comes out as ~59, and the other values check out too. Therefore the algorithm is logarithmic and dependent on $\log_{1.1} N$, and the notation used would be $O(\log_{1.1} N)$

Method 3 (Print Odd-Even Pairs):

```
for(long i = 1; i < N; i += 2) {
      for(long k = i+1; k < N; k += 2) {
            System.out.println("("+i+","+k+")");
      }
}
```

[2 point]:

The Big-O cost of method 3 is:$O(N^2)$

Experimental Results [1 point]:

N=	256	512	1024	2048
cost=	8128	32640	130816	523776

Based on the experimental results, write down 1-2 sentences to justify the Big-O cost: [2 point]

As N doubles the cost increases by 4 times, hence the algorithm is exponential and is dependent on N^2. The notation used in this case would be $O(N^2)$

Method 4 (Print Powers):

```
for(long i = 1; i < N; i <<= 1) {
      for(long k = 0; k < i; k ++) {
            System.out.println("("+i+","+k+")");
      }
}
```

[2 point]:

The Big-O cost of method 4 is: O(N)

Experimental Results [1 point]:

N=	256	512	1024	2048
cost=	255	511	1023	2047

Based on the experimental results, write down 1-2 sentences to justify the Big-O cost: [2 point]

As N doubles so does cost, the algorithm is linear and is dependent on N directly. The notation used in this case would be O(N)

Method 5 (Print Primes):

This method prints out all prime numbers from 2 to N. The outer loop runs from 2 to N, and the inner loop (which is wrapped into a function called isPrime) checks whether a given number x is prime or not. The way to check whether a number x is prime or not is to loop from 2 to `sqrt(N)` and check if x is divisible by any integer in that range.

This is a very straightforward but not very efficient way to print all prime numbers from 2 to N (if you search Wikipedia, you will find a lot more efficient ways, but those are more complicated).

The outer loop:

```
for (long i=2; i<=N; i++) {
      if(isPrime(i)) System.out.println(i);
}
```

The inner loop (wrapped into a function):

```
private boolean isPrime(long x) {
      boolean is_prime = true;
      for(long k=2;k*k<x;k++) {
            if(x%k==0) is_prime = false;
      }
      return is_prime;
}
```

[3 point]:

The Big-O cost of method 5 is: $O(N^{1.5})$

As this question is more challenging than all previous, we will give you a few choices to choose from. The correct answer is one of the following:

$O(N)$, $O(N^{1.5})$, $O(N^2)$, $O(N^{2.5})$, $O(\log N)$, $O(N \log N)$, $O(2^N)$

Experimental Results [1 point]:

N=	100000	200000	400000	800000
cost=	20931855	59328481	168054861	475827906

Based on the experimental results, write down 1-2 sentences to justify the Big-O cost: [3 point]

As N doubles cost increases by a factor of 2.83 and when N increases 4 times cost increases by ~8.02. this rate of change is most comparable to $O(N^{1.5})$. Hence the algorithm is exponential and the notation used would be $O(N^{1.5})$