Lab 2: Numerical Integration

Ariel Kohanim

November 18, 2018

1 Introduction

For this assignment I integrated the function:

$$\int_1^{100} 1/x dx$$

via a Java program. When integrated we solve for the natural $log100$; which approximates roughly to 4.605

There are many methods to approximate an integral, such as by using Simpson's method or Trapezium. The following program integrates the formula above using the rectangular midpoint. This was done in 3 different methods:
1. for-loop
2. sequential stream
3. parallel stream

2 Time Complexity

The time complexity of the three methods approximating the same integral $\int_1^{100} 1/x dx$ are all $\Theta(n)$ Therefore all functions grow linearly with respect to 'n'

2.1 For-loop

```
private static double rectangular(double a, double b, int n, FPFunction f, int mode)
  {
    double range = checkParamsGetRange(a, b, n);
    double modeOffset = (double)mode / 2.0;
    double nFloat = (double)n;
    double sum = 0.0;
    for (int i = 0; i < n; i++)
    {
      double x = a + range * ((double)i + modeOffset) / nFloat;
```

```
      sum += f.eval(x);
    }
    return sum * range / nFloat;
  }
```

This function has a time complexity of an upper bound of $O(n)$ and lower bound $\Omega(n)$ and therefore has $\Theta(n)$ since all variables stay in constant size, except for the input of 'n'.

2.2 Sequential Stream

```
public static double rectangularStream(double a, double b, int n, FPFunction f, int mode)
{
  double range = checkParamsGetRange(a, b, n);
  double modeOffset = (double)mode / 2.0;
  double nFloat = (double)n;

  double sum = IntStream.range (0,n)
          .mapToDouble(i -> a + range * ((double)i + modeOffset) / nFloat)
          .map(x -> f.eval(x))
          .reduce(0, (y, z) -> y+z);
      return sum * range / nFloat;
}
```

This function has a time complexity of an upper bound of $O(n)$ and lower bound $\Omega(n)$ and therefore has $\Theta(n)$ Even though it is a stream; all variables stay in constant size, except for the input of 'n'.

2.3 Parallel Stream

```
 public static double rectangularParallelStream(double a, double b, int n, FPFunction f, i
{
  double range = checkParamsGetRange(a, b, n);
  double modeOffset = (double)mode / 2.0;
  double nFloat = (double)n;

  double sum = IntStream.range (0,n)
          .parallel()
          .mapToDouble(i -> a + range * ((double)i + modeOffset) / nFloat)
          .map(x -> f.eval(x))
          .reduce(0, (y, z) -> y+z);
      return sum * range / nFloat;
}
```

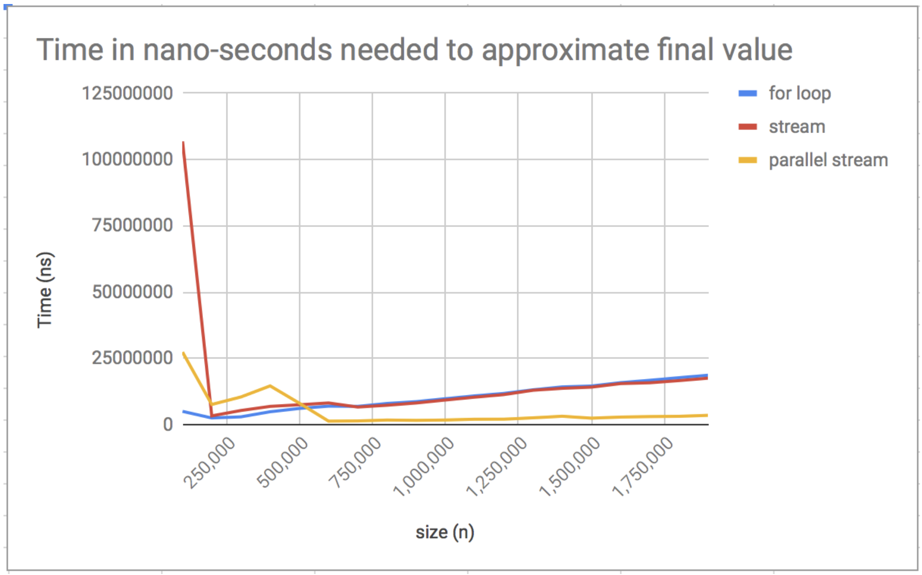

Figure 1: Graph of Integration methods

This function, is nearly identical to the previous one. Therefore it should nto surprise us that it has a time complexity of an upper bound of $O(n)$ and lower bound $\Omega(n)$ and therefore has $\Theta(n)$ Even though it is a stream; all variables stay in constant size, except for the input of 'n'. Reference figure 1 for a visual of this phenomena.

3 Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs. That means how much memory, in the worst case, is needed at any point in the algorithm.

The for-loop is the most efficient with respect to space complexity for this algorithm.

Streams on the other hand utilize arrays, hence have a higher space complexity relative to the for loop.

size	for loop	stream	parallel stream
100,000	5032467	106880401	27382883
200,000	2540272	3338483	7633342
300,000	2971924	5366247	10468583
400,000	4891809	6913359	14640298
500,000	6135641	7529309	8110309
600,000	6997552	8206686	1341344
700,000	6922006	6659927	1415062
800,000	8011948	7335684	1774933
900,000	8705471	8192154	1647817
1,000,000	9812604	9310139	1783528
1,100,000	10880410	10384040	2055613
1,200,000	11801474	11351067	2082881
1,300,000	13168486	12990408	2593493
1,400,000	14269505	13716231	3190506
1,500,000	14607716	14171289	2462337
1,600,000	15866778	15509504	2888512
1,700,000	16759217	15815109	3064957
1,800,000	17736618	16635903	3156951
1,900,000	18663649	17556368	3545499

Figure 2: Raw Data