LAB 1

By Jason Ivey

Testing R for measuring run-time growth in Fibonacci number generation.

Contents

List of Figures

List of Tables

1 Introduction

The Fibonacci number sequence is a simple number sequence defined by the following function:

$$Fib(n) = \begin{cases} 0 & \text{if } x0 \\ 1 & \text{if } x1 \\ Fib(n-1) + Fib(n-2) & \text{if } x > 1 \end{cases}$$

With the first 10 values being: $[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]$

There are numerous ways of generating these numbers however the two algorithms used in this lab are a iterative and recursive method.

Let the iterative version be: $Fib_0\ (n)$
Let the recursive version be: $Fib_1\ (n)$

The purpose of this lab is to measure the run-time growth of each and characterize the general behavior of each version of Fibonacci generation. The underlying goal is to discover the geometric difference in exponential versus linear run-time. To meet these goals C++ is used with R to process the runtime data into vectors.

2 Methods

Directly translating the piece wise definition of the Fib function in the introduction the $Fib_0(n)$ can be described in C++ as.

$Fib_0\ (n) =$

```
int fib_0( int n ) {
  long prev2 = 0, prev = 1, current = 0;
  for( int x = 0; x < n; x++ ){
    current = prev + prev2;
    prev2 = prev;
    prev = current;
  }
  return prev2;
}
```

The recursive version $fib_1(n)$ was provided by Ph.D. Song of NMSU Computer Science department for this lab.

$Fib_1\ (n) =$

```
int fib_1(int n)    {
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fib1(n-1) + fib1(n-2);
}
```

In order to time and process both functions in R the following R comment was added to the C++ program:

```
 1  // Below is the R portion to visualize the performance
 2  /*** R
 3  k <- 10
 4  ns <- 34+(1:k)
 5  runtime <- vector(length=k)
 6  runtime2 <- vector(length=k)
 7  for(i in 1:k) {
 8      n <- ns[i]
 9      runtime[i] <- (system.time(fib1(n))["user.self"])
10  }
11  plot(ns, runtime, type="b", xlab="n",ylab="runtime (second)")
12  k <- 30
13  ns <- 10000000*(1:k)
14  for(i in 1:k) {
15    n <- ns[i]
16    runtime2[i] <- (system.time(fib2(n))["user.self"])
17  }
18  plot(ns, runtime2, type="b", xlab="n",ylab="runtime (second)")
19  grid(col="blue")
20  */
```

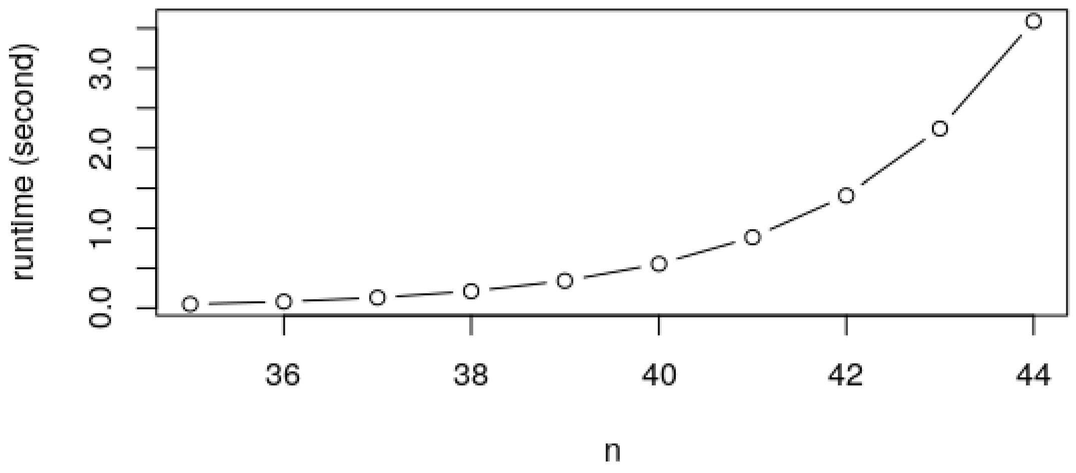
To explain the above R C++ embedded comment it must be broken into two parts. Part one is lines 1-11 and part two is lines 12-18. The first part is recording the run-times of $Fib_1\ (n)$ for an input vector of size 10 where the base value is 34 and steps of 1 are made at each position in the positive direction. After the $Fib_1\ (n)$times are recorded to a vector the are plotted. The second part of the comment records the run-times of $Fib_0\ (n)$ for an input vector of size 30 where the base value is 10000000 and steps of 10000000 are made at each position in the positive direction. The output vector with the run-times for each value is then plotted on a separate graph.

A note on why the value provided to each function are not the same and why $Fib_0\ (n)$ must use much larger steps. When measuring algorithms on a live operating system a process controller/manager controls execution and dynamically allocated ram based on algorithms. Unfortunately in this case this algorithm does not preload $Fib_0\ (n)$ to memory. The speed of program execution on the CPU versus the speed of loading the short program from memory is very disproportionate therefore a large amount of noise is introduced in the first few seconds of execution. This noise is extreme compared to the linearly increasing run-time. There are many ways to offset this, however the method that demands the least amount of effort is to make the programs run-time so great compared to the disk loading to RAM that the noise disappears into the background. Therefore one extreme is trumped by another, the lab run-time suffers greatly from this, however the goal is to compare iterative and recursive methods and this does not detract from that.

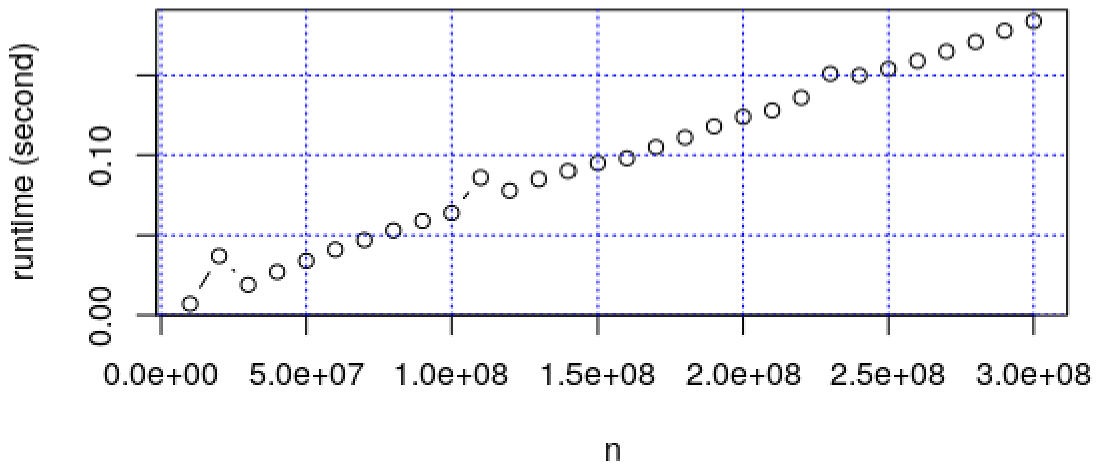
3 Results

Although the numbers generated by the recursive and iterative functions were orders of magnitudes different with the recursive inputs being smaller. The recursive function grew much quicker and had a longer overall run-time.

$Fib_1\ (n)$ Results:



$Fib_0\ (n)$ Results:



4 Discussion

Despite the use of numbers several orders larger than in $Fib_0\ (n)$, $Fib_1\ (n)$ still had a run-time nearly triple that of the iterative algorithm. This is due to the Big-0 behavior of each method and not necessarily due to the superiority of iterative versus recursive methods. There are cases such that recursive algorithms are faster or even must be used because iterative methods cannot be used. In a perfect situation noise would not be an issue such as when your program is executed on a supercomputer where the operating system is optimized for the code being measured or ran to produce some sort of output.

5 Conclusion

From these results in the form of run-times its clear that the worst case Big-0 has a huge influence of how run-time grows as input increase. However, the actual second measurements cannot be accurately predicted due to noise and language related problems, therefore it is better to characterize behavior in a general manner.