Algorithms in Systems Engineering ISE 172

Lecture 3

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 2
- References
  - D.E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (Third Edition), 1997.
  - CLRS Chapter 2

Designing Algorithms

- We have already motivated the development of algorithms that are both correct and efficient.
- How do we know if an algorithm is correct and what do we mean by efficient?

Proving Correctness

- Correctness of an algorithm must be proven mathematically.
- For the algorithms we'll study, this will usually be easy, but it can be difficult in some cases.
- Most algorithms follow one of the following paradigms.
  - Iterative: The algorthm executes a loop until a termination condition is satisfied.
  - Recursive: Divide the problem into one or smaller instances of the same problem.
- In both cases, we must prove both that the algorithm terminates and that the result is correct.
  - Correctness of iterative algorithms is typically proven by showing that there is an *invariant* that holds true after each iteration.
  - Recursive algorithms are almost always proven by an induction argument.

Example: Insertion Sort

A simple algorithm for sorting a list of numbers is insertion sort

```
def insertion_sort(l):
    for i in range(1, len(l)):
        save = l[i]
        j = i
        while j > 0 and l[j - 1] > save:
            l[j] = l[j - 1]
            j -= 1
        l[j] = save
```

Why is this algorithm correct? What is the invariant?

Example: Calculating Fibonacci Numbers

Our initial algorithm for calculating the n^{th} Fibonacci number is an example of a recursive algorithm.

```
def fibonacci1(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci1(n-1) + fibonacci1(n-2)
```

- The correctness of this algorithm does not really need to be proven formally, but to illustrate, we could prove it using induction.
- A formal inductive proof requires
  - a base case; and
  - an inductive hypothesis
- What are they in this case?
- How do we know the algorithm terminates?

Analyzing Efficiency

- The goal of analyzing an algorithm is twofold.
  - First, we want to determine how quickly it will execute in practice.
  - Second, we want to compare different algorithms for the same problem in order to choose the "best" one.
- This can be done either *empirically* or *theoretically*.
- Empirical analysis involves implementing the algorithm and testing it on various inputs.
- In general, the speed of execution of an algorithm depends on
  - the instance,
  - the algorithm, and
  - the hardware.
- Separating out the effects of these factors can be difficult.
- Theoretical analysis allows us to do this, but let's first look at a basic empirical analysis.

Example: Calculating Fibonacci Numbers

- Let's try to measure how long it takes to calculate the n^{th} Fibonacci number using the two basic algorithms discussed last time.
- Here is a small routine that returns the execution time of a function for calculating a fibonacci number.

```
def timing(f, n):
    print 'Calculating fibonacci number', n
    start = time()
    f(n)
    return time()-start

>>> print timing(fibonacci1, 10)
0.00299978256226
>>> print timing(fibonacci3, 10)
0.0
```

Example: Calculating Fibonacci Numbers (cont'd)

- Notice that we are passing a function as an argument to another function.
- Since functions are just objects, we can put them on lists and pass them as argument, which is very useful.
- What happened to the result of the `fibonacci3()` call?

Example: Calculating Fibonacci Numbers (cont'd)

- The problem with the previous example was that the execution time of the function was so small, it could not be measured accurately.
- To overcome this problem, we can call the function repeatedly in a loop and then take an average.

```
def timing(f, n, iterations = 1):
    print 'Calculating fibonacci number', n

    start = time()
    for i in range(iterations):
        f(n)
    return (time()-start)/iterations

>>> print timing(fibonacci1, 10, 1000)
0.0021319996948
>>> print timing(fibonacci3, 10, 1000)
3.61999988556e-05
```

Example: Calculating Fibonacci Numbers (cont'd)

- Note that the third argument to the function has a default value and is optional in calling the function.
- With this new function, we get a sensible measurement.
- Here, `fibonacci1()` is not obviously inefficient—we do not see this until we try larger numbers.

Running Time as a Function of “Input Size”

- Typically, running times grow as the “amount” of data (number of inputs or magnitude of the inputs) grows.
- We are interested in knowing the general trend.
- Let’s do this in the fibonacci case.

```
algos = [fibonacci1, fibonacci3]
symbols = ['bs', 'rs']
symboldict = dict(zip(algos, symbols))
actual = {}
for a in algos:
    actual[a] = []
    for i in range(1, 30):
        actual[a].append(timing(a, i))
# create plots
for a in algos:
    plt.plot(range(1, 30), actual[a], symboldict[a])
plt.show()
```

Plotting the Data

- To plot the data, we use `matplotlib`, a full-featured package that provides graphing capabilities similar to Matlab.
- Plotting the results of the code on the previous slide, we get the following.

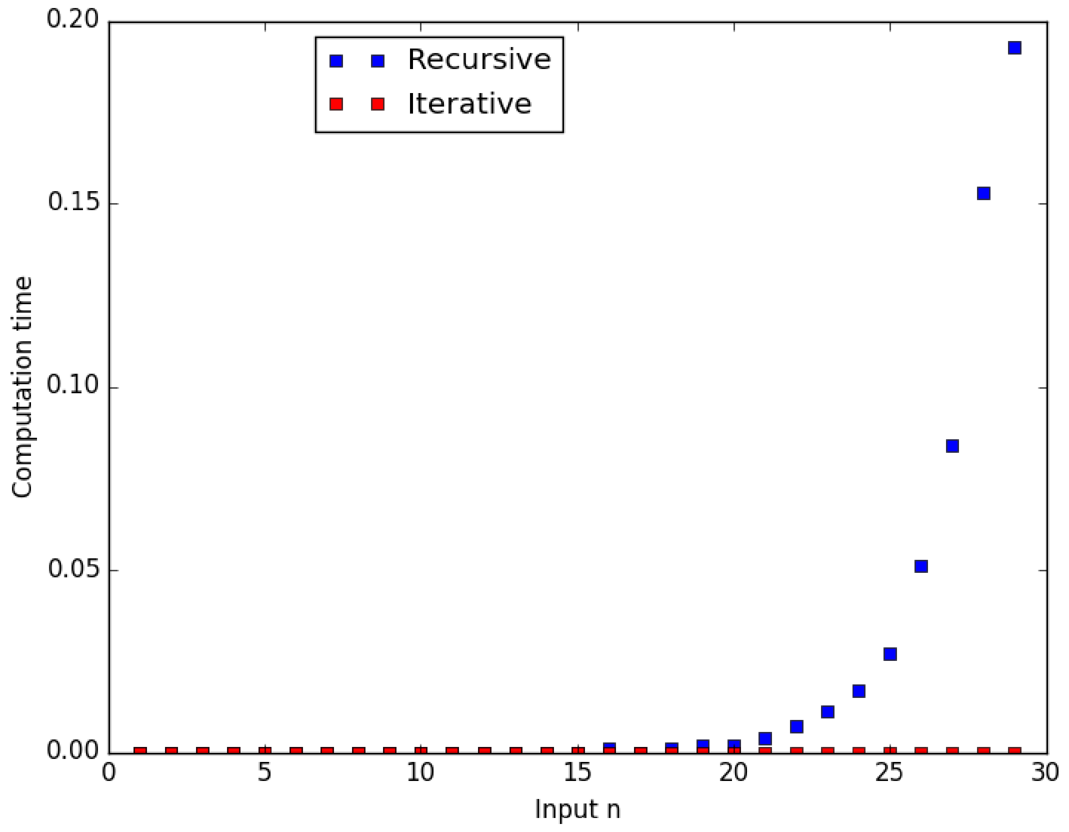

Figure 1: Running times of recursive versus iterative algorithms

Theoretical Analysis

- Can we derive the graph on the previous slide “theoretically”?
- In a basic theoretical analysis, we try to determine how many “steps” would be necessary to complete the algorithm.
- We assume that each “step” takes a constant amount of time, where the constant depends on the hardware.
- We might also be interested in other resources required for the algorithm, such as memory.
- What is the “theoretical” running time for each of the fibonacci algorithms?
  - Aside from the recursive calls, there are only roughly 2 “steps” in each function call.
  - The number of function calls **is** the n^{th} Fibonacci number!

Theortical Analysis

- Let's try to compare our theoretical prediction to the empirical data from earlier.
- What are the "units" of measurement?
- To put the numbers on the same scale, we need to either determine the hardware constant or count the number of "representative operations"

```
theoretical = []
actual = []
n = range(1, 30)
for i in n:
    actual.append(timing(fibonacci1, i))
    theoretical.append(fibonacci3(i))
# figure out the constant factor to put times on the same sc
scale = actual[-1]/theoretical[-1]
theoretical = [theoretical[i]*scale for i in range(len(n))]
plt.plot(n, actual, 'bs')
plt.plot(n, theoretical, 'ys')
plt.show()
```

Plotting the Data

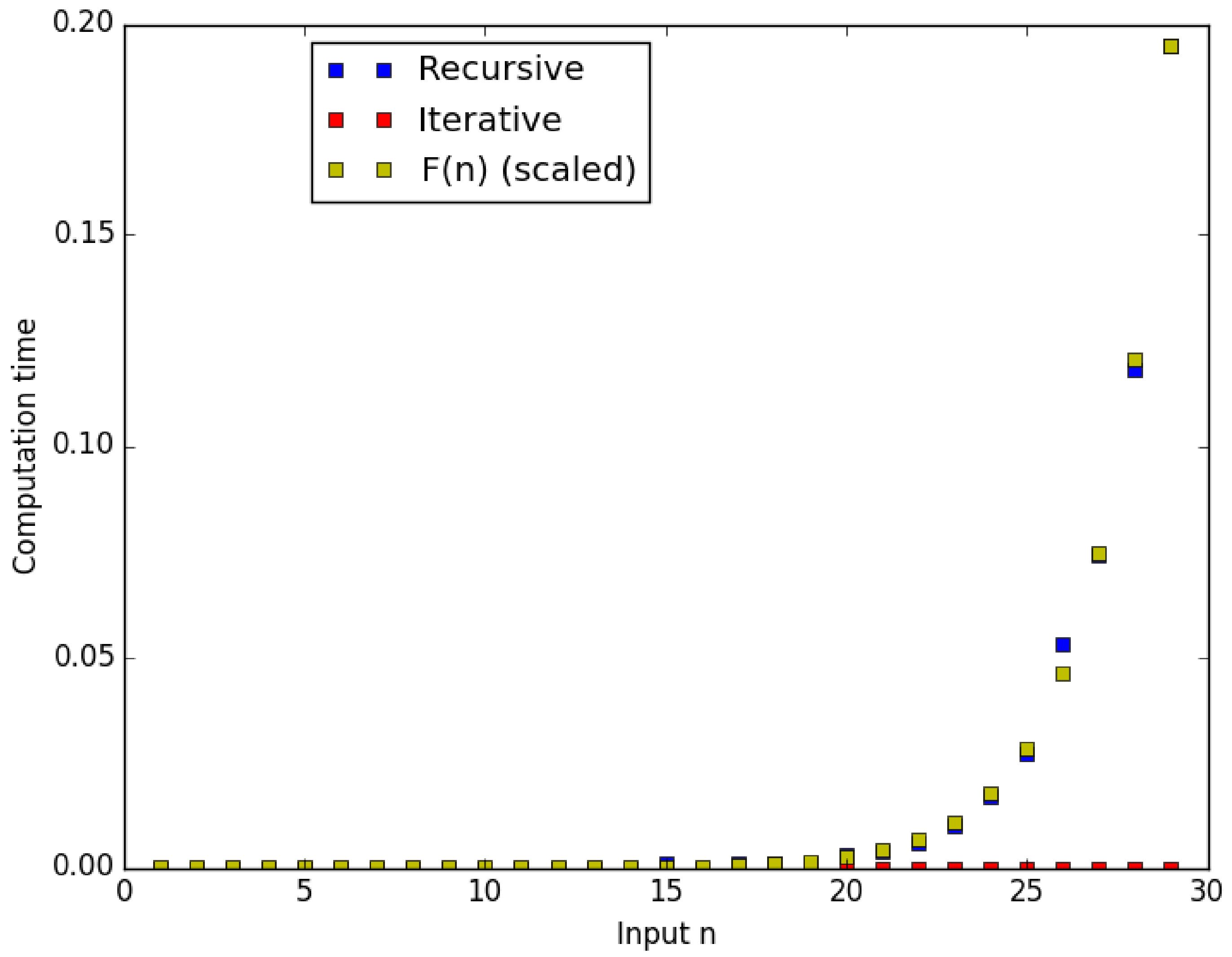

Figure 2: Comparing theoretical and empirical running times

More Complex Algorithms

- For most algorithms we will encounter, the analysis is not quite so straightforward.
  - We may not be able to derive the theoretical running time so easily.
  - The algorithm may behave very differently on different inputs.
- What do we want to know?
  - Best-case
  - Worst-case
  - Average-case
- It may depend on how much we know about the instances that will be encountered in practice or how risk-averse we are.

Example: Sorting

- Let's again consider the insertion sort algorithm.
  - How should we test it?
  - How about random instances?
  - Can we guess anything about the algorithm theoretically?

Insertion Sort: Simple Empirical Analysis

Generating random inputs of different sizes, we get the following empirical running time function.

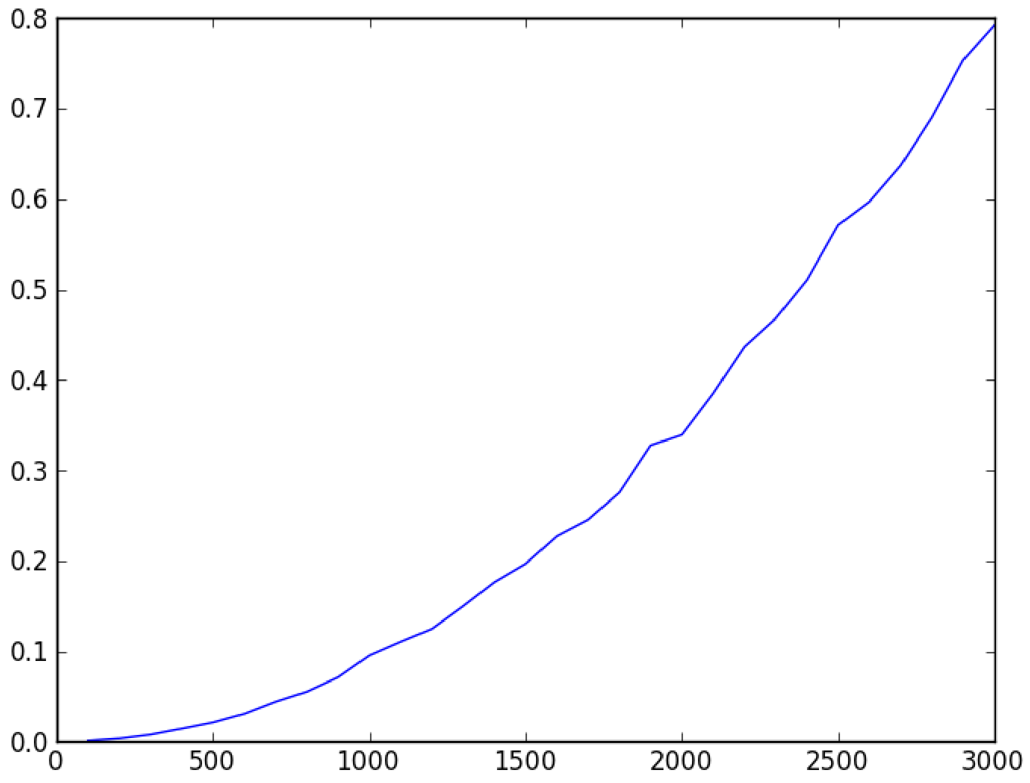

Figure 3: Running time of insertion sort on randomly generated lists

What is your guess as to what function this is?

Insertion Sort: Theoretical Analysis

- What is the maximum number of steps the insertion sort algorithm can take?
- On what kinds of inputs is the worst-case behavior observed?
- What is the "best" case?
- On what kinds of inputs is this best case observed?
- Do you think that empirical analysis will tell us everything we need to know about this algorithm?

Operation Counts

- One way of avoiding the dependence on hardware is to count "representative operations".
- What are the basic operations in a sorting algorithm?
  - Compare
  - Swap
- Most sorting algorithms consist of repetitions of these two basic operations.
- The number of these operations performed is a proxy for the empirical running time that is independent of hardware.

Plotting Operation Counts

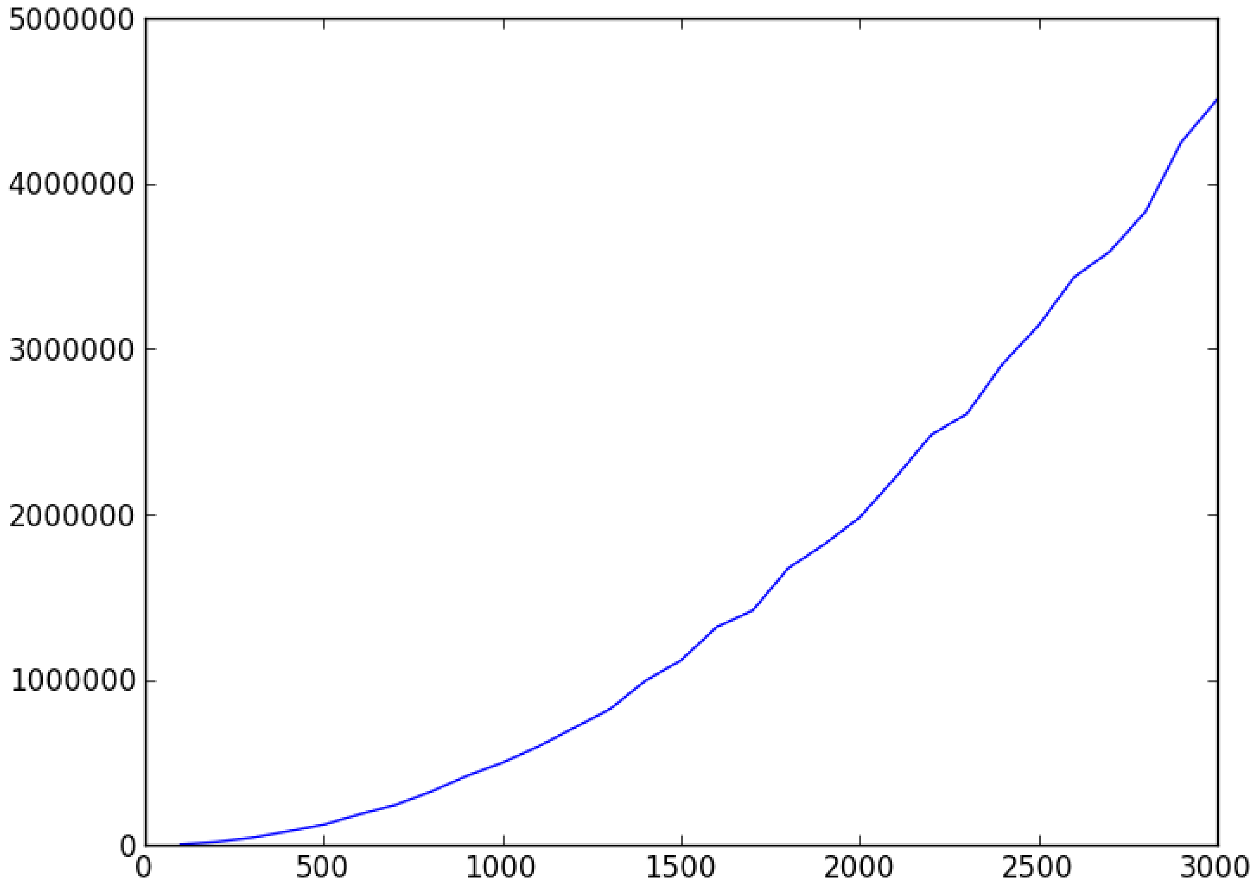

Figure 4: Operation counts for insertion sort on randomly generated lists

Obtaining Operation Counts

- One way to obtain operation counts is using a profiler.
- A profiler counts function calls and all reports the amount of time spent in each function in your program.

```
>>> cProfile.run('insertion_sort_count(aList)', 'cprof.out')
>>> p = pstats.Stats('cprof.out')
>>> p.sort_stats('cumulative').print_stats(10)

   ncalls  tottime  percall  cumtime  percall function
        1    1.011    1.011    3.815    3.815 insertion_sort
   251040    0.507    0.000    0.507    0.000 shift_right
   252027    0.393    0.000    0.393    0.000 compare
      999    0.002    0.000    0.002    0.000 assign
```

Bottleneck Operations

- If an algorithm is not running as efficiently as we think it should, we may want to know where efforts to improve the algorithm would best be spent.
- Bottleneck analysis breaks an algorithm into parts (mpdules) and analyzes each part using the same analysis as we use for the whole.
- By determine the running times of individual modules, we can determine which part is the most crucial in improving the overall running time.
- To do this, we can make a graph showing the percentage of the running time taken by each module as a function of input size.
- This should make it obvious which module is the bottleneck.
- As we analyze more complex algorithms, we will do some of these kinds of analyses.

Final Example: Repeated Squaring

- Recall the repeated squaring algorithm from Lecture 1 for computing x^n.
- Here is a complete implementation for a basic version of the repeated squaring algorithm .

```
def pow_improved(x, n):
    if n is 0:
        return 1
    if n is 1:
        return x
    y = x
    m = int(floor(log(n, 2)))
    for i in range(m):
        y *= y
    for i in range(2**m, n):
        y *= x
    return y
```

- What do you expect to happen when we actually run this algorithm?

Comparing Running TImes for Exponentiation

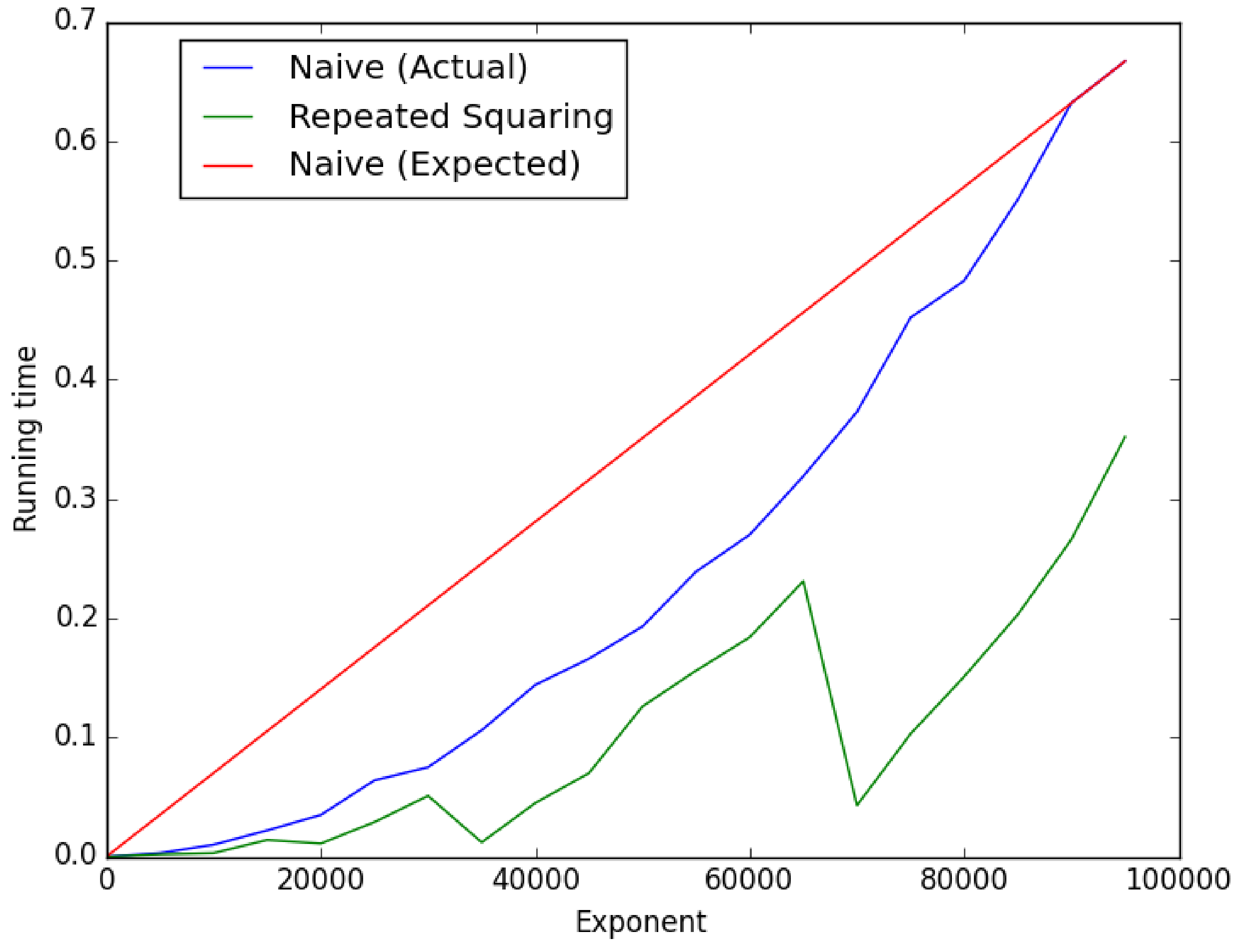

Figure 5: Comparing the expected and actual running times of the naive algorithm for exponentiation against repeated squaring

What's going on here???