# CSCI 315: Data Structures Performance Analysis

Paul E. West, PhD

Department of Computer Science
Charleston Southern University

August 29, 2019

# Analysis of Algorithms

- Dilemma: you have two (or more) methods to solve problem, how to choose the BEST?
- One approach: implement each algorithm in C, test how long each takes to run.
- Problems:
  - Different implementations may cause an algorithm to run faster/slower
  - Some algorithms run faster on some computers
  - Algorithms may perform differently depending on data (e.g., sorting often depends on what is being sorted)

# Better Approach Step 1

- characterize performance in terms of key operation(s)
- Sorting:
  - count number of times two values compared
  - count number of times two values swapped
- Search:
  - count number of times value being searched for is compared to values in array
- Recursive function:
  - count number of recursive calls

# Better Approach Step 2

- Want to comment on the “general” performance of the algorithm
- Emperical: Measure for several examples, but what does this tell us in general?
- Analytical:
  - Instead, assess performance in an abstract manner
  - Idea: analyze performance as size of problem grows
  - Examples:
    - Sorting: how many comparisons for array of size N?
    - Searching: #comparisons for array of size N
  - May be difficult to discover a reasonable formula

# Analsysis With Varying Results

- Example: for some sorting algorithms, a sorting routine may require as few as N-1 comparisons and as many as $\frac{N^2}{2}$
- Types of analyses:
  - Best-case: what is the fastest an algorithm can run for a problem of size N?
  - Average-case: on average how fast does an algorithm run for a problem of size N?
  - Worst-case: what is the longest an algorithm can run for a problem of size N?
- Computer scientists *usually* use worst-case analysis

# Notice: We Are **Estimating**

- What is often done is to approximate or estimate the performance of an algorithm
- Estimation is an important skill to learn and to use
- Example Question: How many hotdogs tall is the Empire State Building?

# Simple Example

- Simplier Question: How tall is the Empire State Building?
- Answer: The ESB is 1250 feet tall.
- Assuming that a hotdog is 6 inches from end to end, you would need, 1250 * 2 = 2500 hotdogs.

# Analysis

- An objective way to evaluate the cost of an algorithm or code section.
- The cost is computed in terms of space or time, usually
- The goal is to have a meaningful way to compare algorithms based on a common measure.
- Complexity analysis has two phases,
  - Algorithm analysis
  - Complexity analysis

# Algorithm Analysis

- Algorithm analysis requires a set of rules to determine how operations are to be counted.
- There is no generally accepted set of rules for algorithm analysis.
- In some cases, an exact count of operations is desired; in other cases, a general approximation is sufficient.
- The rules presented that follow are typical of those intended to produce an exact count of operations.

# Rules

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
   1. assignment operation
   2. single I/O operations
   3. single Boolean operations, numeric comparisons
   4. single arithmetic operations
   5. function return
   6. array index operations, pointer dereferences

# More Rules

3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loop execution time is the sum, over the number of times the loop is executed, of the body time + time for the loop check and update operations, + time for the loop setup.
5. Always assume that the loop executes the maximum number of iterations possible Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

# Example 1

```
count = count + 1;   // Cost: c1
sum = sum + count;   // Cost: c2
```

Total Cost = c1 + c2.
Since we assume '+' cost 1 and assignment cost 1, the total cost is 4.

# Example 2

```
if (n < 0) { // Cost: c1
    absval = -n; // Cost: c2
} else {
    absval = n; // Cost: c3
}
```

- Total Cost <= c1 + max(c2,c3)
- c1 is the cost of boolean evaluation. Since there is 1 evaluation ($<$), Cost(c1) = 1.
- c2 is the cost of negating a number (1) + the cost of assignment (1). Cost(c2) = 2.
- c3 is the cost of assignment(1). Cost(c3) = 1
- Cost of the worse-case is 3.
- Cost of the best-cast is 2.
- Average case is 2.5.

# Example 3

```
i = 1; // Cost: c1
sum = 0; // Cost: c2
while (i <= n) { // Cost: c3
    i = i + 1; // Cost: c4
    sum = sum + i; // Cost: c5
}
```

# Example 3

```
i = 1; // Cost: c1
sum = 0; // Cost: c2
while (i <= n) { // Cost: c3
    i = i + 1; // Cost: c4
    sum = sum + i; // Cost: c5
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1.

# Example 3

```
i = 1; // Cost: c1
sum = 0; // Cost: c2
while (i <= n) { // Cost: c3
    i = i + 1; // Cost: c4
    sum = sum + i; // Cost: c5
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1.
- Cost(c4) = 1 + 1 = 2 (remember assignment and + both cost 1!).

# Example 3

```
i = 1; // Cost: c1
sum = 0; // Cost: c2
while (i <= n) { // Cost: c3
    i = i + 1; // Cost: c4
    sum = sum + i; // Cost: c5
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1.
- Cost(c4) = 1 + 1 = 2 (remember assignment and + both cost 1!).
- Cost(c5) = 2.

# Example 3

```
i = 1; // Cost: c1
sum = 0; // Cost: c2
while (i <= n) { // Cost: c3
    i = i + 1; // Cost: c4
    sum = sum + i; // Cost: c5
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1.
- Cost(c4) = 1 + 1 = 2 (remember assignment and + both cost 1!).
- Cost(c5) = 2.
- How many time does the loop execute?

# Example 3

```
i = 1; // Cost: c1
sum = 0; // Cost: c2
while (i <= n) { // Cost: c3
    i = i + 1; // Cost: c4
    sum = sum + i; // Cost: c5
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1.
- Cost(c4) = 1 + 1 = 2 (remember assignment and + both cost 1!).
- Cost(c5) = 2.
- How many time does the loop execute?
- Loop: n times, so total cost is:

# Example 3

```
i = 1; // Cost: c1
sum = 0; // Cost: c2
while (i <= n) { // Cost: c3
    i = i + 1; // Cost: c4
    sum = sum + i; // Cost: c5
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1.
- Cost(c4) = 1 + 1 = 2 (remember assignment and + both cost 1!).
- Cost(c5) = 2.
- How many time does the loop execute?
- Loop: n times, so total cost is:
- Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5 = c1 + c2 + c3 + n(c3 + c4 + c5)

# Nested Example

```
i = 1; // Cost c1
sum = 0;  // Cost c2
while (i <= n) {  // Cost c3
    j = 1; // Cost c4
    while (j <= n) { // Cost c5
        sum = sum + i; // Cost c6
        j = j + 1;     // Cost c7
    }
    i = i + 1; // Cost c8
}
```

# Nested Example

```
i = 1; // Cost c1
sum = 0;  // Cost c2
while (i <= n) {  // Cost c3
    j = 1; // Cost c4
    while (j <= n) { // Cost c5
        sum = sum + i; // Cost c6
        j = j + 1;    // Cost c7
    }
    i = i + 1; // Cost c8
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1, Cost(c4) = 1, Cost(c5) = 1, Cost(c6) = 2, Cost(c7) = 2, Cost(c8) = 2

# Nested Example

```
i = 1; // Cost c1
sum = 0;  // Cost c2
while (i <= n) {  // Cost c3
    j = 1; // Cost c4
    while (j <= n) { // Cost c5
        sum = sum + i; // Cost c6
        j = j + 1;    // Cost c7
    }
    i = i + 1; // Cost c8
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1, Cost(c4) = 1, Cost(c5) = 1, Cost(c6) = 2, Cost(c7) = 2, Cost(c8) = 2
- First (outer) while loop execution: n

# Nested Example

```
i = 1; // Cost c1
sum = 0;  // Cost c2
while (i <= n) {  // Cost c3
    j = 1; // Cost c4
    while (j <= n) { // Cost c5
        sum = sum + i; // Cost c6
        j = j + 1;    // Cost c7
    }
    i = i + 1; // Cost c8
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1, Cost(c4) = 1, Cost(c5) = 1, Cost(c6) = 2, Cost(c7) = 2, Cost(c8) = 2
- First (outer) while loop execution: n
- Second (inner) while loop execution: n, total cost is:

# Nested Example

```
i = 1; // Cost c1
sum = 0;  // Cost c2
while (i <= n) {  // Cost c3
    j = 1; // Cost c4
    while (j <= n) { // Cost c5
        sum = sum + i; // Cost c6
        j = j + 1;    // Cost c7
    }
    i = i + 1; // Cost c8
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1, Cost(c4) = 1, Cost(c5) = 1, Cost(c6) = 2, Cost(c7) = 2, Cost(c8) = 2
- First (outer) while loop execution: n
- Second (inner) while loop execution: n, total cost is:
- $c1 + c2 + (n + 1) * c3 + n * c4 + n * (n + 1) * c5 + n * n * c6 + n * n * c7 + n * c8 =$
  $c1 + c2 + c3 + n * (c3 + c4 + c8) + n * n * c5 + n * c5 + n * n * c6 + n * n * c7 =$

  $c1 + c2 + c3 + n * (c3 + c4 + c5 + c8) + n * n(c5 + c6 + c7)$

# Nested Example

```
i = 1; // Cost c1
sum = 0;  // Cost c2
while (i <= n) {   // Cost c3
    j = 1; // Cost c4
    while (j <= n) { // Cost c5
        sum = sum + i; // Cost c6
        j = j + 1;     // Cost c7
    }
    i = i + 1; // Cost c8
}
```

- Cost(c1) = 1, Cost(c2) = 1, Cost(c3) = 1, Cost(c4) = 1, Cost(c5) = 1, Cost(c6) = 2, Cost(c7) = 2, Cost(c8) = 2
- First (outer) while loop execution: n
- Second (inner) while loop execution: n, total cost is:
- $c1 + c2 + (n+1) * c3 + n * c4 + n * (n+1) * c5 + n * n * c6 + n * n * c7 + n * c8 =$
  $c1 + c2 + c3 + n * (c3 + c4 + c8) + n * n * c5 + n * c5 + n * n * c6 + n * n * c7 =$

  $c1 + c2 + c3 + n * (c3 + c4 + c5 + c8) + n * n(c5 + c6 + c7)$
- **Important Note:** n*n ($n^2$) is the highest (largest) term!

# Comparing Algorithms

- We measure an algorithm's time requirement as a function of the problem size.
- Problem size depends on the application: e.g. number of elements in a list for a sorting algorithm, the number disks for towers of hanoi.
- So, for instance, we say that (if the problem size is n)
  - Algorithm A requires $5 * n^2$ time units to solve a problem of size n.
  - Algorithm B requires $7 * n$ time units to solve a problem of size n.
- An algorithm's proportional time requirement is known as growth rate.
- We can compare the efficiency of two algorithms by comparing their growth rates.

# Example

- Which is better?
  - $50N^2 + 31N^3 + 24N + 15$
  - $3N^2 + N + 21 + 4 * 3^N$

# Example

- Which is better?
  - $50N^2 + 31N^3 + 24N + 15$
  - $3N^2 + N + 21 + 4 * 3^N$
- Well, it depends on N:

| N | $50N^2 + 31N^3 + 24N + 15$ | $3N^2 + N + 21 + 4 * 3^N$ |
|---|---|---|
| 1 | 120 | 37 |
| 2 | 511 | 71 |
| 3 | 1374 | 159 |
| 4 | 2895 | 397 |
| 5 | 5260 | 1073 |
| 6 | 8655 | 3051 |
| 7 | 13266 | 8923 |
| 8 | 19279 | 26465 |
| 9 | 26880 | 79005 |
| 10 | 36255 | 236527 |

# What happened?

| N | $3N^2 + N + 21 + 4 * 3^N$ | $4 * 3^N$ | % of Total |
|---|---|---|---|
| 1 | 37 | 12 | 32.4 |
| 2 | 71 | 36 | 50.7 |
| 3 | 159 | 108 | 67.9 |
| 4 | 397 | 324 | 81.6 |
| 5 | 1073 | 972 | 90.6 |
| 6 | 3051 | 2916 | 95.6 |
| 7 | 8923 | 8748 | 98.0 |
| 8 | 26465 | 26244 | 99.2 |
| 9 | 79005 | 78732 | 99.7 |
| 10 | 236527 | 236196 | 99.9 |

- One term dominated the others.
- This implies we *really* only care about the dominating (highest/largest) term.

# As N Grows, Some Terms Dominate

| Function | N=10 | N=100 | N=1000 | N=10000 | N = 100000 |
|---|---|---|---|---|---|
| $log_2 N$ | 3 | 6 | 9 | 13 | 16 |
| $N$ | 10 | 100 | 1000 | 10000 | 100000 |
| $N * log_2 N$ | 30 | 664 | 9965 | $10^5$ | $10^6$ |
| $N^2$ | $10^2$ | $10^4$ | $10^6$ | $10^8$ | $10^{10}$ |
| $N^3$ | $10^3$ | $10^6$ | $10^9$ | $10^{12}$ | $10^{15}$ |
| $2^N$ | $10^3$ | $10^{30}$ | $10^{301}$ | $10^{3010}$ | $10^{30103}$ |

- Ordering:
  $1 < log_2 N < N < N * log_2 N < N^2 < N^3 < 2^N < 3^N$

# Big O

- If Algorithm A requires time proportional to $f(n)$, Algorithm A is said to be order $f(n)$, and it is denoted as $O(f(n))$.
- The function $f(n)$ is called the algorithm's growth-rate function.
- Since the capital O is used in the notation, this notation is called the Big O notation.
- If Algorithm A requires time proportional to $n^2$, it is $O(n2)$.
- If Algorithm A requires time proportional to $n$, it is $O(n)$.

# Example 1

- If an algorithm requires $n^2 - 3 * n + 10$ seconds to solve a problem size n. If constants $k$ and $n_0$ exist such that $k * n^2 + n_0 > n^2 - 3 * n + 10$ for all $n$ and $n_0$.
- Then the algorithm is order $n^2$ (In fact, $k$ is 3 and $n_0$ is 2)
- Thus, the algorithm requires no more than $k * n^2$ time units.
- So it is $O(n^2)$

# Examples

- This is actually not that difficult. It is a game of "spot the highest term!"
- $50N^2 + 31N^3 + 24N + 15 = O(N^3)$
- $3N^2 + N + 21 + 4 * 3^N = O(3^N)$
- It can get somewhat tricky:
- $N(3 + N(9 + N)) + N^2 = O(N^3)$
- $N(10 + log_2 N) + N = O(N * log_2 N)$

# Growth Reate Function Explained

- $O(1)$ Time requirement is constant, and it is independent of the problem's size.
- $O(log_2 n)$ Time requirement for a logarithmic algorithm increases increases slowly as the problem size increases.
- $O(n)$ Time requirement for a linear algorithm increases directly with the size of the problem.
- $O(n * log_2 n)$ Time requirement for a $n * log_2 n$ algorithm increases more rapidly than a linear algorithm.
- $O(n^2)$ Time requirement for a quadratic algorithm increases rapidly with the size of the problem.
- $O(n^3)$ Time requirement for a cubic algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.
- $O(2^n)$ As the size of the problem increases, the time requirement is too rapid to be practical.

# Example 1

```
i = 1; // Cost c1
sum = 0; // Cost c2
while (i <= n) { // Cost c3
    i = i + 1; // Cost c4
    sum = sum + i; // Cost c5
}
```

# Example 1

```
i = 1; // Cost c1
sum = 0; // Cost c2
while (i <= n) { // Cost c3
    i = i + 1; // Cost c4
    sum = sum + i; // Cost c5
}
```

- T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*c5
  = (c3+c4+c5)*n + (c1+c2+c3)
  = a*n + b
- So, the growth-rate function for this algorithm is $O(n)$

# Example 2

```
i=1; // Cost c1
sum = 0;   // Cost c2
while (i <= n) {   // Cost c3
    j=1; // Cost c4
    while (j <= n) {  // Cost c5
        sum = sum + i; // Cost c6
        j = j + 1;   // Cost c7
    }
    i = i +1 // Cost c8
}
```

# Example 2

```
i=1; // Cost c1
sum = 0;   // Cost c2
while (i <= n) {   // Cost c3
     j=1; // Cost c4
     while (j <= n) { // Cost c5
         sum = sum + i; // Cost c6
         j = j + 1;   // Cost c7
     }
     i = i +1 // Cost c8
}
```

- T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8
  = (c5+c6+c7)*n*n + (c3+c4+c5+c8)*n + (c1+c2+c3)
  = a*n*n + b*n + c
- So, the growth-rate function for this algorithm is $O(n^2)$

# Example 3

```
for (i=1; i<=n; i++) { // Cost(c1)
    for (j=1; j<=i; j++) { // Cost(c2)
        for (k=1; k<=j; k++) { // Cost(c3)
            x=x+1; // Cost(c4)
        }
    }
}
```

# Example 3

```
for (i=1; i<=n; i++) { // Cost(c1)
    for (j=1; j<=i; j++) { // Cost(c2)
        for (k=1; k<=j; k++) { // Cost(c3)
            x=x+1; // Cost(c4)
        }
    }
}
```

- T(n) = c1*(n+1) + c2*((n+1)*(n+2)) / 2) + c3* ( estimated: (n * (n + 1) * (2n + 1)) / 6) + c4*( estimated: (n * (n + 1) * (2n + 1)) / 6)
  = a*n3 + b*n2 + c*n + d
- So, the growth-rate function for this algorithm is $O(n^3)$
- **Notice:** You do NOT need to know the exact number of iterations to find Big-O.

# Example 4

Unfortunately, recursive can be hard...

```
void hanoi(int n, char source, char dest, char spare) { // Cost of function call
  if (n > 0) { // Cost(c1)
      hanoi(n-1, source, spare, dest);   // Cost(c2)
    cout << "Move top disk from pole " << source
             << " to pole " << dest << endl; // Cost(c3)
      hanoi(n-1, spare, dest, source);   // Cost(c4)
} }
```

- By now, I hope you see that constance costs are virtually supurfulous when working with Big O.
- To find the growth-rate function for a recursive algorithm, we have to solve its recurrence relation.
- You will learn how to do this in Discrete Structures.

# Example 4 continued

- What is the cost of hanoi(n,'A','B','C')?
- when n=0 T(0) = c1
- when n>0 T(n) = c1 + c2 + T(n-1) + c3 + c4 + T(n-1)
  = 2*T(n-1) + (c1+c2+c3+c4)
  = 2*T(n-1) + c -> recurrence equation for the growth-rate function of hanoi-towers algorithm
- Now, we have to solve this recurrence equation to find the growth-rate function of hanoi-towers algorithm
- This turns out to be $O(2^n)$ because for every N we make 2(n-1) calls.

# Crash Course

- gnuplot makes graphs
- type “gnuplot” at your terminal
- type “plot sin(x) with line”
- type “plot sin(x) with point”
- Type “set terminal postscript color”
- Type “set output “nameofplot.ps” ”
- Type “replot” or “rep”

# Continued

- Type “set title “plotname” ”
- Type “set ylabel “ylabel” ”
- Type “set xlabel “xlabel” ”
- to covert to something readable (like pdf). On the command line do:
  $ ps2pdf nameofplot.ps
- Then you should have nameofplot.pdf
  $ evince nameofplot.pdf

# Huge Time Saver!

- The commands to gnuplot can be saved to a file and then automatically used:
- simple.plot:

```
set terminal postscript color
set output "simple.ps"
set ylabel "time (seconds)"
set xlabel "size"

...
```

- $ gnuplot simple.plot
- $ ps2pdf simple.ps
- $ evince simple.pdf
- Will display the graph.
- This is especially useful if you put it in a makefile!

```
plot:
	gnuplot simple.plot
	ps2pdf simple.ps
	evince simple.pdf
```

# Perf

- Tool to find what part of the code is running slow.
- Steps:
  1. compile with debug flags! (-g)
  2. run:
     $ sudo perf record <your-programs-name>
  3. view the report:
     $ sudo perf report
- Now you should see what functions are taking up all the time!

```
Samples: 991  of event 'cycles:ppp', Event count (approx.): 2061166086
Overhead  Command  Shared Object       Symbol
  95.29%  a.out    a.out               [.] sumOfOneTo
   4.39%  a.out    ld-2.28.so          [.] 0x000000000000a25a
   0.20%  a.out    a.out               [.] sumOfOneToSquared
   0.08%  a.out    [kernel.kallsyms]   [k] __do_page_fault
   0.03%  a.out    [kernel.kallsyms]   [k] kmem_cache_alloc
   0.00%  perf_4.  [kernel.kallsyms]   [k] native_sched_clock
   0.00%  perf_4.  [kernel.kallsyms]   [k] native_apic_mem_write
   0.00%  perf_4.  [kernel.kallsyms]   [k] native_write_msr
```

# Emperical Measurement

- While Analytical measurement of performance is important sometimes an emperical approach is most useful.
- The naive way to take timings:
  1. Record time as start
  2. Run section of code you wish to time
  3. Record time as end
  4. You answer is (end - start).
- While there are numerous issues with this approach, it will give sufficient approximate timings for this class.

```
#include <iostream> // To print
#include <time.h> // Required for taking timings

int main(int argc, char *argv[]) { // Standard main heading.
    /* clock_t is the data type for storing timing information.
     * We must make two variables, one for the start and the other to capture
     * the difference.
     */
    clock_t start, diff;

    // timeAmount is used to print out the time in seconds.
    double timeAmount;

    // We want to run our algorithm over varying sizes.
    for (int i = 1000; i < 1000000; i += 1000) {
        // Capture the start clock
        start = clock();

        // This is were your algorithm should be called.
        functionCallToYouAlgorithm(i);

        // Capture the clock and subtract the start to get the total time elapsed.
        diff = clock() - start;

        // Convert clock_t into seconds as a floating point number.
        timeAmount = diff * 1.0 / CLOCKS_PER_SEC;

        // Print out first the size (i) and then the elapsed time.
        std::cout << i << " " << timeAmount << "\n";

        // We flush to ensure the timings is printed out.
        std::cout << std::flush;
```

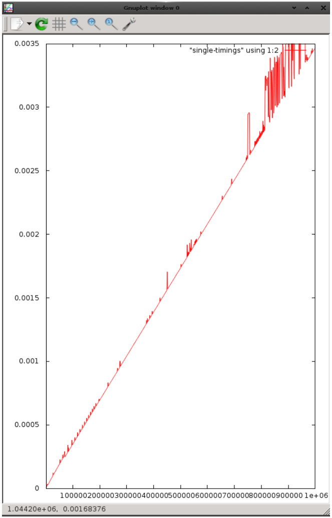
# Example Output

```
1000  4e-06
2000  8e-06
3000  1.2e-05
4000  2.5e-05
5000  2.9e-05
6000  2.4e-05
7000  3.5e-05
8000  2.9e-05
9000  3.2e-05
10000  3.5e-05
11000  3.9e-05
12000  4.2e-05
13000  4.5e-05
14000  5.2e-05
15000  5.6e-05
16000  6e-05
17000  6.5e-05
18000  6.8e-05
19000  7e-05
20000  7.6e-05
```

- First is the size (1000) and second is the number of seconds (pretty small.)

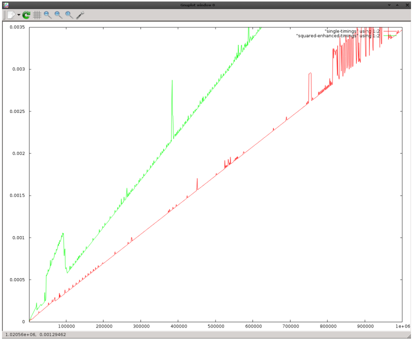
# Plotting

- First lets us assume we have the full listing in a file named 'single-timings'
- With gnuplot we can simply graph the timings:
  plot [:][:] "single-timings" using 1:2 with line



# Multiple Data Collections

- In addition to 'single-timings' let us assume we have another file (in the same format) named 'squared-enhanced-timings'
- With gnuplot we can graph both timings: plot [:][:] "single-timings" using 1:2 with line, "squared-enhanced-timings" using 1:2 with line
- You can append more and more data files in this manner.

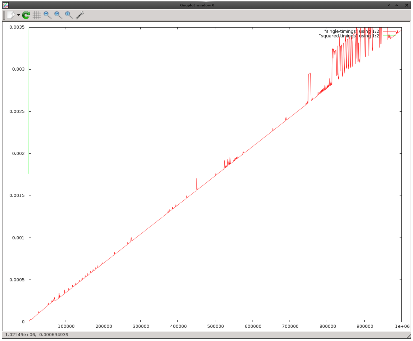


# Summary

- Hopefully you can see from today that we can:
  - Analyze algorithm analytically to predict performance.
  - Profile code to find what piece of code is the bottleneck.
  - Get & plot timings to see actual performance.
- Performance Analysis could take the whole class time.
- We stick with the basics for this class.
- I do want to alert you to a couple of things that will help!

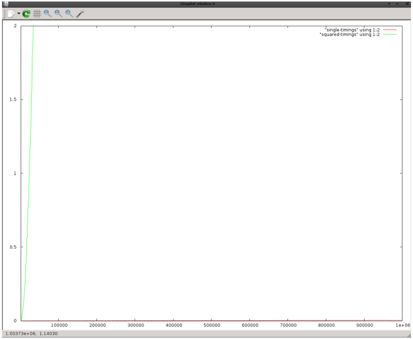
# Gotcha 1

- You plot the data of two algorithms, but you can see only one!



# Gotcha 1

- You plot the data of two algorithms, but you can see only one!
- Check your data and axis, usually it is because it is too small to see.

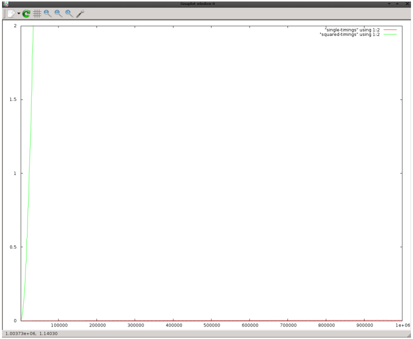


# Gotcha 1

- You plot the data of two algorithms, but you can see only one!
- Check your data and axis, usually it is because it is too small to see.
- plot [:][:] "single-timings" using 1:2 with line, "squared-timings" using 1:2 with line ->

  plot [:][:2] "single-timings" using 1:2 with line,

  "squared-timings" using 1:2 with line



# Performance Tips

- Make sure you are working on optimizing the correct function and looking for the correct code improvements.
- From our example:

```
Samples: 991  of event 'cycles:ppp', Event count (approx.): 2061166086
Overhead  Command  Shared Object          Symbol
  95.29%  a.out    a.out                  [.] sumOfOneTo
   4.39%  a.out    ld-2.28.so             [.] 0x000000000000a25a
   0.20%  a.out    a.out                  [.] sumOfOneToSquared
   0.08%  a.out    [kernel.kallsyms]      [k] __do_page_fault
   0.03%  a.out    [kernel.kallsyms]      [k] kmem_cache_alloc
   0.00%  perf_4.  [kernel.kallsyms]      [k] native_sched_clock
   0.00%  perf_4.  [kernel.kallsyms]      [k] native_apic_mem_write
   0.00%  perf_4.  [kernel.kallsyms]      [k] native_write_msr
```

- Most of the time is spent in sumOfOneTo, so improving sumOfOneTo's performance may help.
- Improving performance of sumOfOneToSquared would not help that much.
  - Maybe we can memoize past results to use in the future – or use a better data structure.

# Summary

- Again, there is a lot and we are scratching the surface.
- Important outcomes:
    - Be able to analytically deduce the performance of code.
    - Be able to profile code to find the hot spots.
    - Be able to emperically run programs to evaluate performance.
    - Understand there are anomalies that will not be addressed in this class.