COMPUTING SCIENCE PEER TUTORING

Starting Week 3!
Are You A CMPT 120, 125, 127, 129 Student?
Need Help?
Drop In To Our CS Peer Tutoring Sessions Today!
We Are Located In CSIL ASB 9820

- http://www.sfu.ca/computing/current-students/undergraduate-students/student-resources/cs_peer_tutoring.html

Algorithm Performance (The Big-O)

CMPT 125
Mo Chen
SFU Computing Science
20/1/2020

Lecture 6

Today:

- Worst-case Behaviour
- Counting Operations
- Performance Considerations
- Time measurements
- Order Notation (the Big-O)

Pessimistic Performance Measure

- Often consider the *worst-case* behaviour as a benchmark.
  - make guarantees about code performance under all circumstances

- Can predict performance by counting the number of “elementary” steps required by algorithm in the worst case
  - derive total steps (T) as a function of input size (N)

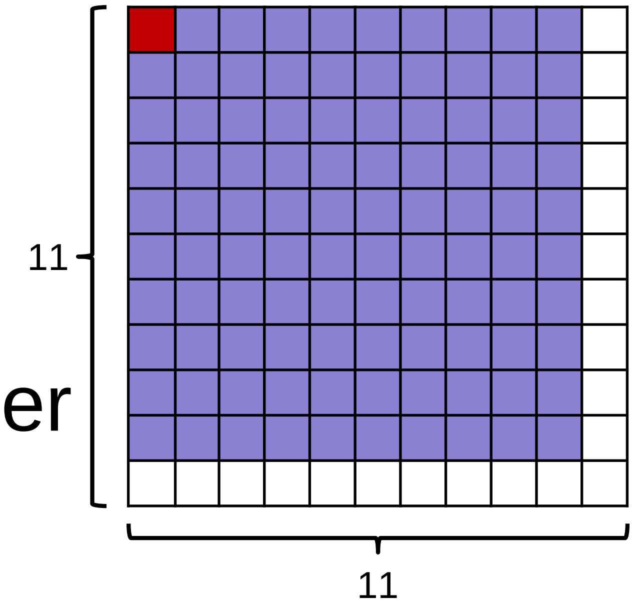
2D Maximum Density Problem

Problem: Given a 2-dimensional array (NxN) of integers, find the 10x10 swatch that yields the largest sum.

Applications:

- Resource management and optimization
- Finding brightest areas of photos

Algorithm / Code?



- Try all possible positions for upper left corner
  - $(N-9) \times (N-9)$ of them
    - (*correction from last class)
  - use a nested loop
- Total each swatch using a 10×10 nested loop
- A *brute-force* approach!
  - Generate a possible solution [naively]
  - Test it [naively]

In C

Precise accounting:

$348N^2 - 6956N + 34762$ operations

$\times (N-9)$

$\times (N-9)$

$\times \mathbf{10}$

```
int max10by10(int a[N][N]) {
  int best = 0;
    for (int u_row = 0; u_row <= N-9; u_row++) {
      for (int u_col = 0; u_col <= N-9; u_col++) {
        int total = 0;
        for (int row = u_row; row < u_row+10; row++) {
          for (int col = u_col; col < u_col+10; col++) {
            total += a[row][col];
          }
        }
        best = max(best, total);
      }
    }
  return best;
}
```

$\mathbf{11}$ $\mathbf{10}$ $\mathbf{10}$

Approximate Method:

Count the *barometer instructions*, the instructions executed most frequently. Usually, in the innermost loop.

Innermost loop: $11 + 10 + 10 = 31$ ops

Total = $31 \times \mathbf{10} \times (\boldsymbol{N} - \mathbf{9}) \times (\boldsymbol{N} - \mathbf{9}) = \mathbf{310}\boldsymbol{N}^{\mathbf{2}}$

Which Performance Measurement?

- Empirical timings
  - run your code on a real machine with various input sizes
  - plot a graph to determine the relationship
- Operation counting
  - assumes all elementary instructions are created equal
- Actual performance can depend on much more than just your algorithm!

Running Time is Affected By . . .

- CPU speed
- Amount of main memory
- Specialized hardware (e.g., graphics card)
- Operating system
- System configuration (e.g., virtual memory)
- Programming Language
- Algorithm Implementation
- Other Programs
- . . .

Comparing Algorithm Performance

- There can be many ways to solve a problem, i.e., different algorithms that produce the same result
  - E.g., There are numerous sorting algorithms.
- Compare algorithms by their behaviour for large input sizes, i.e., as N gets large
  - On today's hardware, **most** algorithms perform quickly for small N
- Interested in growth rate as a function of N
  - E.g., Sum an array: *linear* growth $= O(N)$
  - E.g., Check for duplicates: *quadratic* growth $= O(N^2)$

Order Notation (the Big-*O*)

- Suppose we express the number of operations used in our algorithm as a function of N, the size of the problem.
- Intuitively, take the dominant term, remove the leading constant, and put $O(\ . \ . \ . \ )$ around it.

- E.g., $f(N) = 348N^2 - 6956N + 34762 \longrightarrow \boxed{O(N^2)}$

Formalities of the Big-*O*

- Given a function $T(N)$, we say $T(N) = O(f(N))$ if $T(N)$ is at most a constant times $f(N)$, except perhaps for some small values of N.
- Properties:
  - constant factors don't matter
  - low-order terms don't matter
- Rules:
  - For any k and any function $f(N)$, $k \cdot f(N) = O(f(N))$
    - E.g., $5N = O(N)$
    - E.g., $\log_a N = O(\log_b N)$. Why?
    - Q. Do leading constants really not matter?

$\log_a N = \frac{\log_b N}{\log_b a}$ for any b

constant

Leading Constants - Experiment

Of course, constant factors affect performance

- E.g., If two different algorithms run in $f_1(N) = 20N^2$ and $f_2(N) = 2N^2$, respectively, you would expect Algorithm 2 to run 10 times faster.
- E.g., Similarly, a 10x faster machine running Algorithm 1 would have the same running time.
- Big-*O* hides leading constants - *a hardware independent analysis.*

Cray Supercomputer

17.6×10^{15} instructions per second
runs unoptimized dup_chk( ) code from last time
$f(N) = 3/2\, N^2 + 5/2\, N + 3$

vs

iMac Desktop Personal Computer (2011)

40×10^9 instructions per second
runs an optimized, different dup_chk ( )
$f(N) = 30N \log N + 5N + 4$

Experimental Results

N	iMac	Cray
100,000	1.2 ms	850 ns
10^6	15 ms	85 μs
10^7	0.2 s	8.5 ms
10^8	2 s	0.85 s
10^9	22 s	1.75 min
10^{10}	4.2 min	2:22 hr
10^{11}	56 min	10 days
10^{12}	8:20 hr	2.7 years

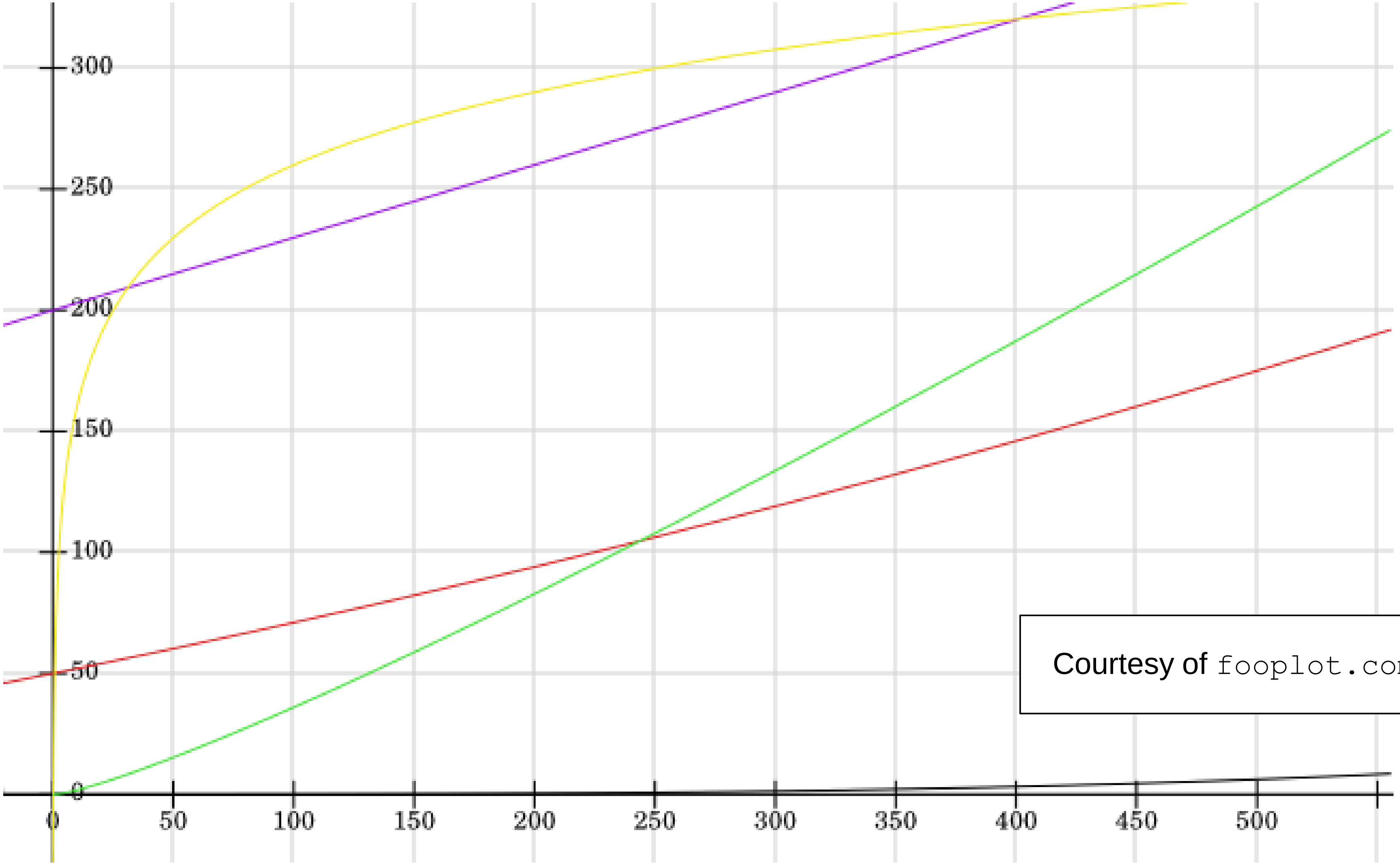
Conclusions:

- Cray runs $O(N^2)$ algorithm
- iMac runs $O(N \log N)$ algorithm which runs faster than Cray for large N (10^9 and beyond)
- Thus slow computer + $O(N \log N)$ >> fast computer + $O(N^2)$ algorithm
- **Rule of Thumb: The slower the function grows, the faster the algorithm.**

- For the $O(N^2)$ Cray, a 10x increase in N leads to roughly a 100x increase in running time.
- For the $O(N \log N)$ iMac, a 10x increase in N leads to roughly a 10x increase in running time (for the N), plus a little (for the $\log N$).

Some Plots

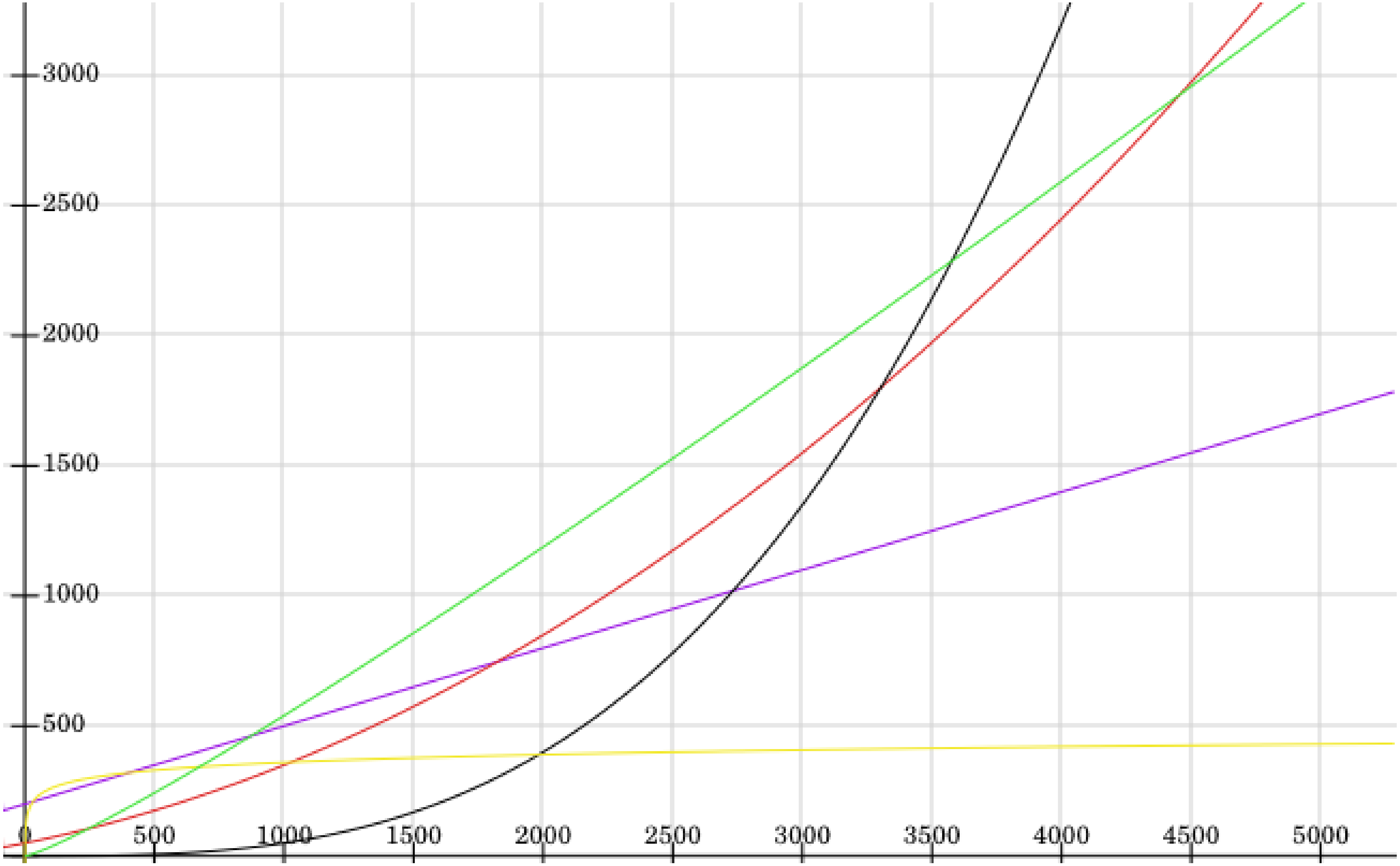
100**log(x)**+60 (yellow)
0.3**x**+200 (purple)
0.18**x*log(x)** (green)
0.0001**x^2**+0.2x+50 (red)
0.00000005**x^3** (black)



Courtesy of `fooplot.com`

Some Plots

100**log(x)**+60 (yellow)
0.3**x+**200 (purple)
0.18**x*log(x)** (green)
0.0001**x^2**+0.2x+50 (red)
0.00000005**x^3** (black)



Big-*O* and Barometer Instructions

Rule of thumb:

The frequency of the barometer instructions will be proportional to the big-O running time

So, find the most frequent operation(s) and count them!

Loops → Multiply

```
int max10by10(int a[N][N]) {
    int best = 0;
    for (int u_row = 0; u_row < N-9; u_row++) {
        for (int u_col = 0; u_col < N-9; u_col++) {
            int total = 0;
            for (int row = u_row; row < u_row+10; row++) {
                for (int col = u_col; col < u_col+10; col++) {
                    total += a[row][col];
                }
            }
            best = max(best, total);
        }
    }
    return best;
}
```

$\times (N-9)$

$\times (N-9)$

$\times 10$

$\times 10$

barometer instructions

$$T(N) = 3 \times 10 \times 10 \times (N-9) \times (N-9) = O(N^2)$$

Analysis of `dup_chk()`

Q. What is N?

- The number of elements in the array

```
     int dup_chk(int a[], int length) {
1      int i = length;
N + 1    while (i > 0) {
N          i--;
N          int j = i - 1;
i + 1      while (j >= 0) {
i            if (a[i] == a[j]) {
               return 1;
             }
i            j--;
           }
         }
1      return 0;
     }
```

Outside of loop: 2 (steps)

Outer loop: $3N + 1$

Inner loop: 3`i`$ + 1$ for all possible `i` from 0 to $N - 1$.

$= 3/2\ N^2 - 1/2\ N$

Grand total = $3/2\ N^2 + 5/2\ N + 3$

A *quadratic* function!

Some Math

Inner loop: $3\texttt{i} + 1$ for all possible `i` from 0 to N - 1.

= 3/2 N^2 - 1/2 N

$$1 + 4 + 7 + \cdots + 3(N-1) + 1 = (1 + \ 3N - 3 + 1) \times \frac{N}{2}$$

$$= \frac{1}{2} N(3N - 1)$$

$$= \frac{3}{2} N^2 - \frac{1}{2} N$$

Observation 1:

- If N is large, then the $\frac{1}{2}N$ term hardly matters

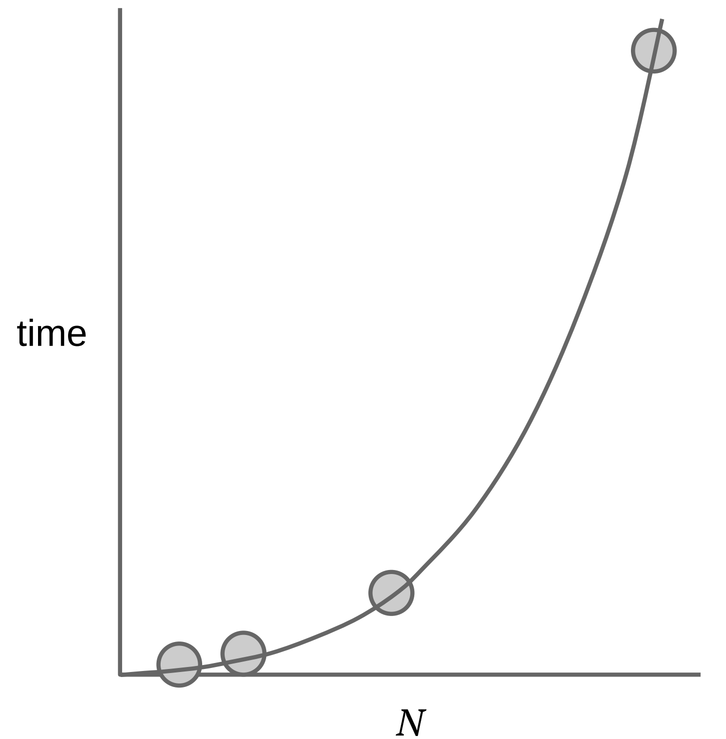
Observation 2: Intuition for arithmetic series, which have N^2 leading terms

- The sum of each pair increase linearly with N
- The number of pairs increases linearly with N
- Multiply two quantities that increase linearly with $N \rightarrow N^2$

Empirical Measurement

- Another graph - a quadratic this time!
- Confirms predictions: **doubling** (2x) the input size leads to a **quadrupling** (4x) of the running time.

N	time (in ms)
10000	89
20000	365
40000	1424
100000	9011



(Informal) Mathematical Definition

- Two functions $T(N), f(N) > 0$ for all large N
  - In this course, $T(N)$ is usually the running time of a program, and $f(N)$ is some well-known function
  - We want to find $f(N)$ that approximates how fast $T(N)$ increases as N becomes very large

- $T(N) = O\big(f(N)\big)$ if and only if there is some positive number M and N_0 such that

$$|T(N)| \leq Mf(N) \text{ for all } N \geq N_0$$

  - $O\big(f(N)\big)$ is an estimate of the **upper bound** of running time $T(N)$

- Many possible choices for $f(N)$ -- we want the **best** one
  - $5N^2 = O(N^3)$ is correct, but not the most useful.
  - $5N^2 = O(N^2)$ would be the best estimate of running time
- Many possibilities for $T(N)$ -- we want the **worst** one
  - When looking for an item in an array, you may find it right away
  - However, in the worst case you have to go through all elements

Polynomials

Rule:

The powers of N are ordered according to their exponents, i.e., $N^a = O(N^b)$ if and only if $a \leq b$

- E.g., $N^2 = O(N^3)$, but N^3 is not $O(N^2)$.

Why are lower-ordered terms not included?

- E.g., If your bank account followed $f(N) = N^2 + N + 1$, you would probably care a lot about the lower-ordered terms for small N, like N=5, as $f(5) = 5^2 + 5 + 1 = \$31$. You'll care about every dollar. But not for larger N, like N=1000, as $f(1000) = 1000^2 + 1000 + 1 = \$1{,}001{,}001$. You care most that you have that million bucks, and not much about the \$1000 or the \$1.

More Rules

3. A logarithm grows more slowly than any other positive power of N.
   - E.g., $\log_2 N = O(N^{1/2})$.
4. If $f(N) = O(g(N))$ and $g(N) = O(h(N))$ then $f(N) = O(h(N))$.
5. If both $f(N)$ and $g(N)$ are $O(h(N))$ then

   $f(N) + g(N) = O(h(N))$

3. If $f_1(N) = O(g_1(N))$ and $f_2(N) = O(g_2(N))$ then $f_1(N) \times f_2(N) = O(g_1(N) \times g_2(N))$

E.g., $(10 + 5N^2)(10\log_2 N + 1) + (5N + \log_2 N)(10N + 2N \log_2 N)$

Typical Growth Rate Functions

- *O*(1) – **constant** time
  - The time is independent of **N**, E.g., list look-up
- *O*(logN) – **logarithmic** time
  - Usually the log is to the base 2, E.g., binary search
- *O*(N) – **linear** time, E.g., linear search
- *O*(N logN) – E.g., quicksort, mergesort
- $O(N^2)$ – **quadratic** time, e.g. selection sort
- $O(N^k)$ – **polynomial** (where k is a constant)
- $O(2^N)$ – **exponential** time, very slow!