CMPT 125

Analysis and O Notation

Analysis

- Checking for duplicates
- Maximum density
- Battling computers and algorithms
- Barometer Instructions
- Big O expressions

Duplicates

Duplicate Function

- Write a function to determine if an array contains duplicates

```
int duplicate_check(int a[], int n) {
    int i = n;
    while (i > 0) {
        i--;
        int j = i - 1;
        while (j >= 0) {
            if (a[i] == a[j]) {
                return 1;
            }
            j--;
        }
    }
    return 0;
}
```

Which statements run most frequently?

The answer depends on the array elements

What is the worst case?

Worst Case Performance

- We often consider the worst case behaviour of an algorithm as a benchmark
  - Guarantees performance under all circumstances
  - Often, but not always, similar to average case behaviour
- Instead of timing an algorithm we can predict performance by counting the number of operations
  - Performed by the algorithm in the worst case
  - Derive total number of operations as a function of the input size (n)

Duplicate Function

```
int duplicate_check(int a[], int n) {
1     int i = n;
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 any loop: 2

outer loop: 3n + 1

inner loop: 3i + 1, for i from n-1 to 1

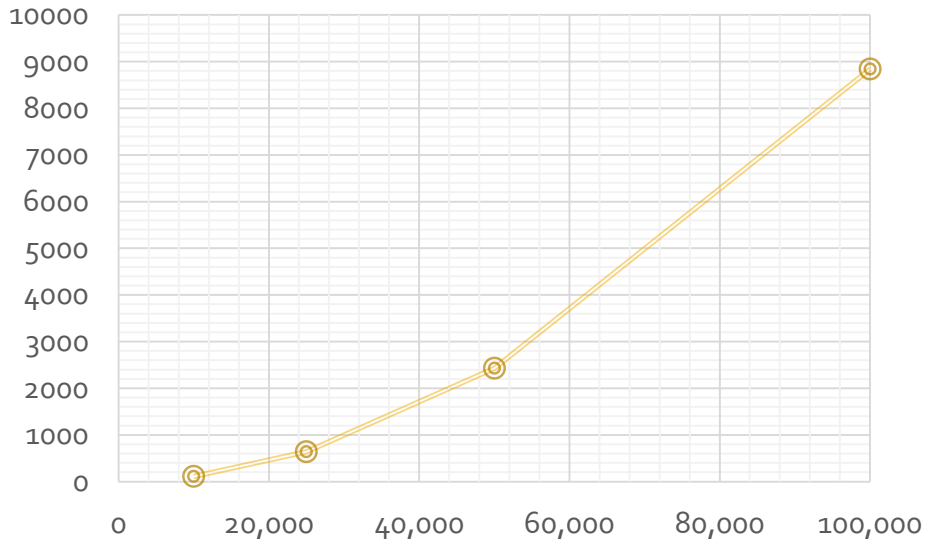
$= 3/2\ n^2 - 1/2n$

total: $3/2\ n^2 + 5/2n + 3$

Empirical Measurement

- Graph the check duplicate algorithm running times
  - Doubling the input size quadruples the running time
  - A quadratic function

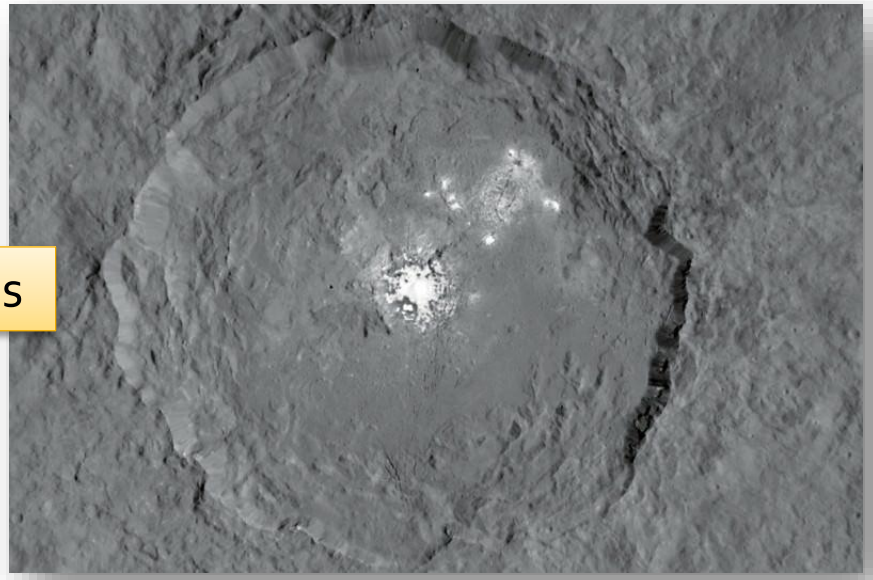
n	time
10,000	108
25,000	640
50,000	2,427
100,000	8,841



2D Maximum Density

Maximum Density

- Given a two dimensional, *n*x*n*, array of integers find the 10x10 swatch with the largest sum
  - Resource management
  - Finding brightest areas of photographs



bright spots on Ceres

Maximum Density Algorithm

- Simple approach
  - Try every possible position for the top-left corner of the 10x10 swatch
    - There are (*n*-10)x(*n*-10) of them
    - Use a nested loop
- The values in each 10x10 swatch can be summed
  - Retaining the largest
- A *brute-force* approach

Maximum Density Algorithm

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

N-10 N-10 10 11 10 10

Total: $348N^2 - 6956N + 34762$

Approximate method: count *barometer instructions*, the most frequently occurring instructions

Total = 31 x 10 x (N-10) x (N-10) = $310N^2$

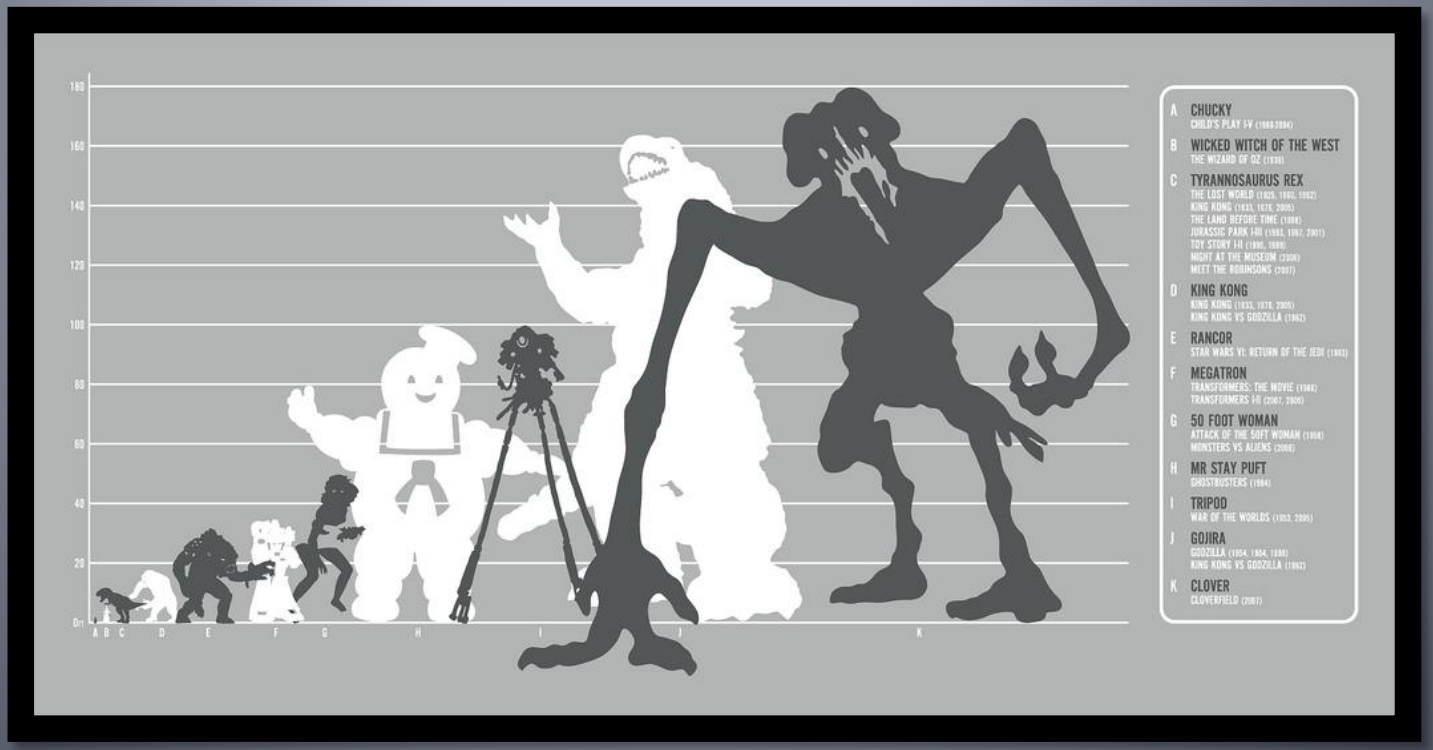
Which Measure?

- Empirical timing
  - Run program on a real machine with various input sizes
    - Plot a graph to determine the relationship
- Operation counting
  - Assumes all elementary instructions execute in approximately the same amount of time
- Actual performance can depend on much more than just the choice of algorithm

Running Time

- Actual running time is affected by
  - CPU speed
  - Amount of RAM
  - Specialized hardware (e.g., graphics card)
  - Operating system
  - System configuration (e.g., virtual memory)
  - Programming Language
  - Algorithm Implementation
  - Other Programs
  - …

Comparisons



Comparing Algorithm Performance

- There can be many ways to solve a problem
  - Different algorithms that produce the same result
    - 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*
  - Sum an array: *linear* growth O(n)
  - Check for duplicates: quadratic growth $O(n^2)$

Order Notation (Big O)

- Express the number of operations in an algorithm as a function of n, the problem size
- Briefly
  - Take the dominant term
  - Remove the leading constant
  - Put O(…) around it
- For example, $f(N) = 348n^2 - 6956n + 34762$
  - i.e. $O(n^2)$

O Notation, Formally

- Given a function $T(n)$
  - Say that $T(N) = O(f(n))$ 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 $f(n)$, $k*f(n) = O(f(n))$
    - e.g., $5n = O(n)$
    - e.g., $\log_a n = O(\log_b n)$

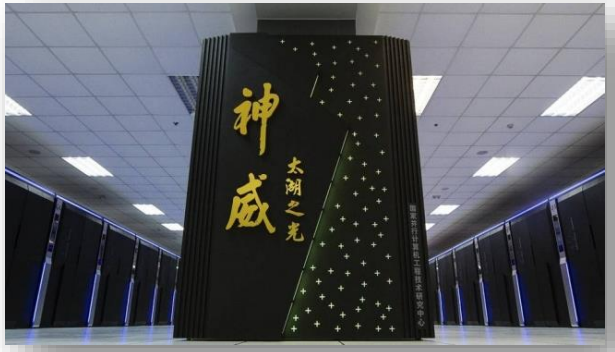
Do leading constants really not matter?

Leading Constants

- Of course leading constants matter
  - Consider two algorithms
    - $f_1(n) = 20n^2$
    - $f_2(n) = 2n^2$
  - Algorithm 2 runs ten times faster
- Let's consider machine speed
  - If machine 1 is ten times faster than machine 2 it will run the same algorithm ten times faster
- Big O notation ignores leading constants
  - It is a hardware independent analysis

Leading Constants

- Let's compare two algorithms on two computers
- Computer 1 – Sunway TaihuLight
  - Speed, 93 petaflops
    - 93 * 10^{15} floating point operations per second
    - 93,000,000 gigaflops
  - Fastest supercomputer 2016



- Computer 2 – Alienware Area 51
  - CPU – Intel Core i7 6950X , 98 gigaflops
    - 98 * 10^{9} floating point operations per second
  - 2016 Gaming PC

- The TaihuiLight is approximately 1,000,000 times faster

The Algorithms

- Let's compare two algorithms on two computers
- Algorithm 1 – nested loop duplicate check
  - $f1(n) = 3/2n^2 + 5/2n + 3$
- Algorithm 2 – a different duplicate check algorithm
  - $f2(n) = 30n * log_2(n) + 5n + 4$
- Not only is the TaihuLight much faster but it's algorithm has a much smaller leading constant
  - 3/2 versus 30

Results

n	TaihuLight	Area 51
100,000	161 ns	514 µs
10^6	16 µs	6 ms
10^7	1.61 ms	72 ms
10^8	161 ms	819 ms
10^9	16 s	9 s
10^{10}	27 mins	102 s
10^{11}	45 hrs	19 mins
10^{12}	187 days	3.4 hrs
10^{13}	51 years	1.5 days

- Conclusions
  - Area 51 running the algorithm with the smaller dominant term is faster
    - For large values of n
  - A slower computer with smaller O algorithm is faster
  - The $n\log(n)$ algorithm does not grow as fast as the n^2 algorithm
  - The slower the function grows the faster the algorithm
    - With n^2 an increase in size of 10x increases running time by 100x
    - With $n\log(n)$ an increase in size of 10x increases running time by just over 10x

Barometer Instructions

Review

- We often use the worst-case behavior as a benchmark
- Derive the number of instructions as a function of the input size, *n*
  - Can use the time command to measure for values of *n*
  - Or count the number of operations
- Use Big O to express the growth rate
  - Compare algorithms behaviour as *n* gets large
  - Remove leading constants
  - Hardware independent analysis

Leading Constants Review

- Leading constants are affected by
  - CPU speed
  - Other tasks performed by the system
  - Memory characteristics
  - Program optimization
- Regardless of leading constants
  - A O(n log(n)) algorithm will outperform a O(n^2) algorithm as n gets large

Algorithms Matter as *n* Grows

- A well designed and written algorithm can make the difference between software being usable or not
  - *n* may be very large
    - Google
  - Or the number of times a task has to be performed may be very large
    - Google again
  - Or it may be necessary to have near instanteous response
    - Real-time systems

Optimization

- Algorithms can be optimized
  - Implemented so as to improve performance
    - Reducing the number of operations
    - Replacing slower instructions with faster instructions
    - Making more efficient of memory
    - …
- Instead of improving an inherently slow algorithm
  - Consider if is there is a better algorithm
    - With a smaller Big O running time
  - Of course, no such algorithm might exist …

Big O

- Given an algorithm, how do you determine its Big O growth rate ?
  - The frequency of the algorithm's *barometer instructions* will be proportional to its Big-O running time
- Identify the most frequent instruction and count it
- Consider
  - Loops
  - Decisions
  - Function calls

Counting – Loops

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

Barometer instructions

$f(N) = 3 \times 10 \times 10 \times (N\text{-}10) \times (N\text{-}10) = O(N)^2$

Counting – Functions

- Function calls are not elementary instructions
  - They must be substituted for the Big O running time

```
int range(int arr[], int n) {
    int lo = min(arr, n);
    int hi = max(arr, n);
    return hi-lo;
}
```

O(n)

O(n)

O(1)

T(n) = O(n) + O(n) + O(1)

= O(n)

Counting – Decisions

- If … else is not an elementary operation
  - Take the larger of the running times
  - Remember … worst case analysis

```
int search(int arr[], int n, int key) {
   if (!sorted(arr, n)) {
     return lsearch(arr, n, key);
   } else {
      return bsearch(arr, n, key);
   }
}
```

O(n)

O(n)

O(log n)

T(n) = O(n) + max(O(n) + O(log n))

=O(n) + O(n)

=O(n)

Big O Notation

Big O Rules

- Take the dominant term, remove the leading constant and put O(…) around it
  - Lower order terms do not matter
  - Constants do not matter

Rules About Polynomials

- 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 $O(n^2)$
- A logarithm grows more slowly than any positive power of *n* greater than 1
  - e.g. $\log_2 n = O(n^{1/2})$
  - Though we say that $\log_2 n = O(\log_2 n)$
    - Note that if log n is referred to it is assumed that the base is 2

More Rules

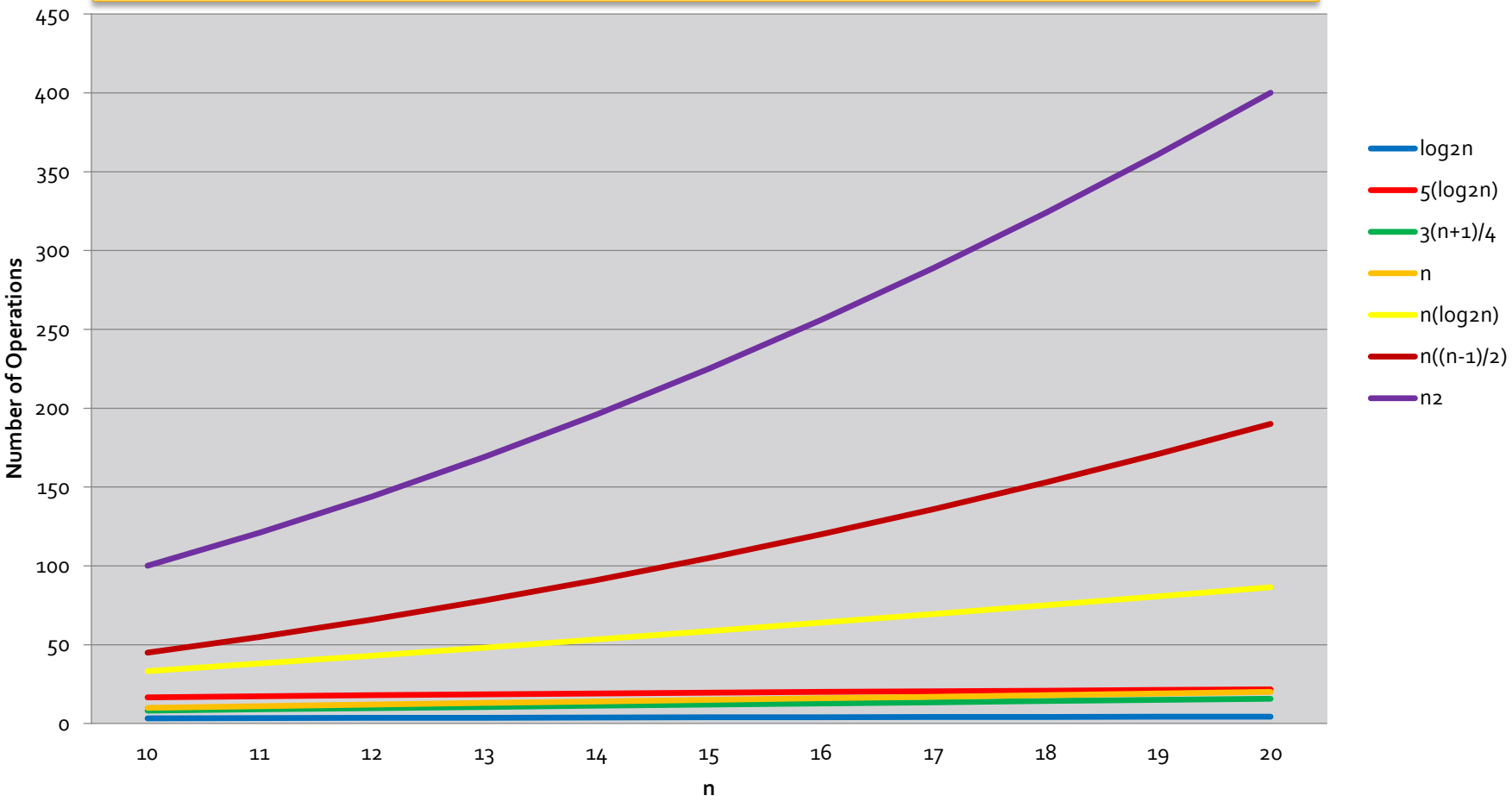
- Transitivity
  - If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$
- Addition
  - $f(n) + g(n) = O(\max(f(n), g(n)))$
- Multiplication
  - if $f1(n) = O(g1(n))$ and $f2(n) = O(g2(n))$
  - then $f1(n) * f2(n) = O(g1(n) * g2(n))$
- An example
  - $(10 + 5n^2)(10\log_2 n + 1) + (5n + \log_2 n)(10n + 2n \log_2 n)$

Common Growth Rates

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

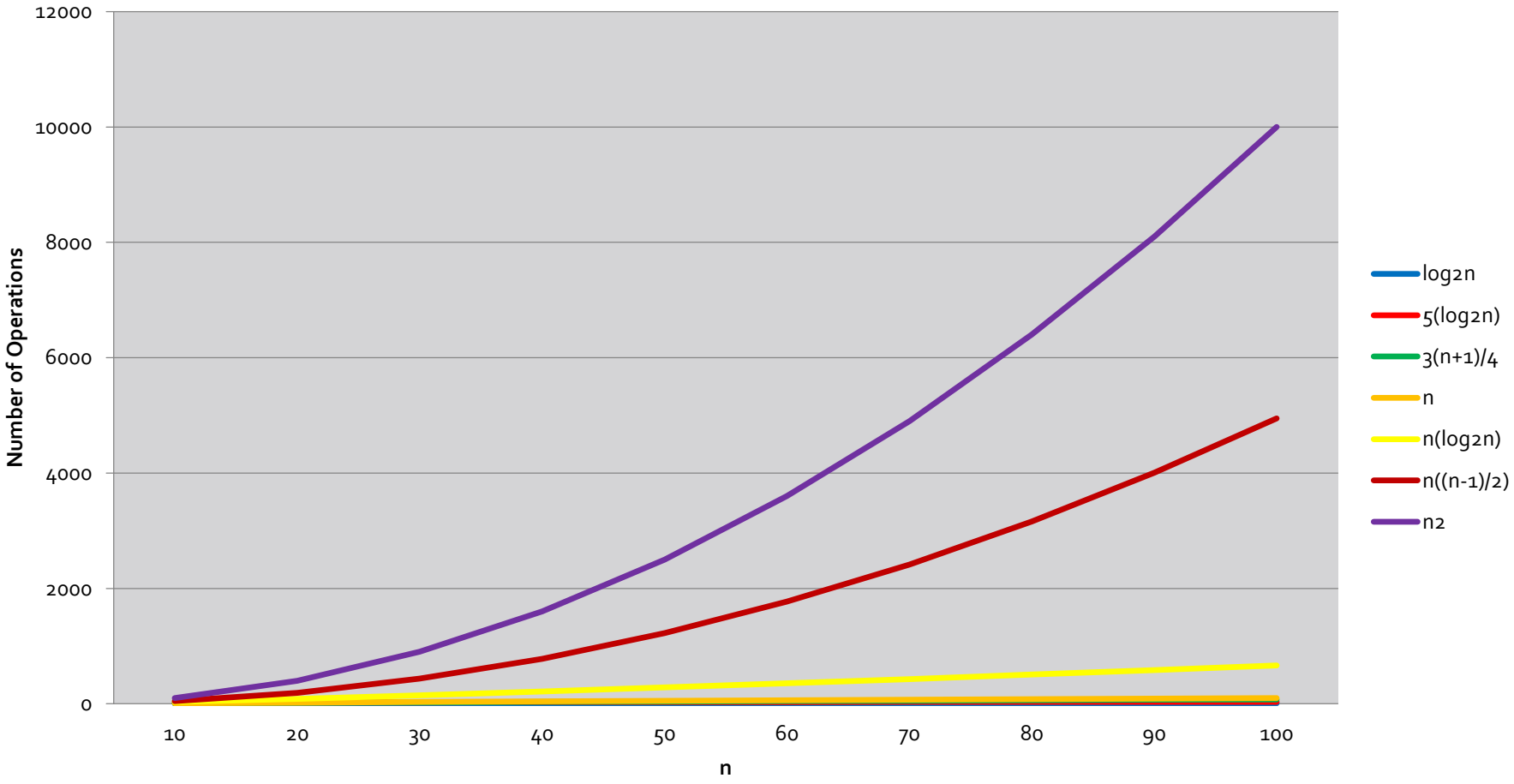
Small *n*

Hard to see what is happening with *n* so small …



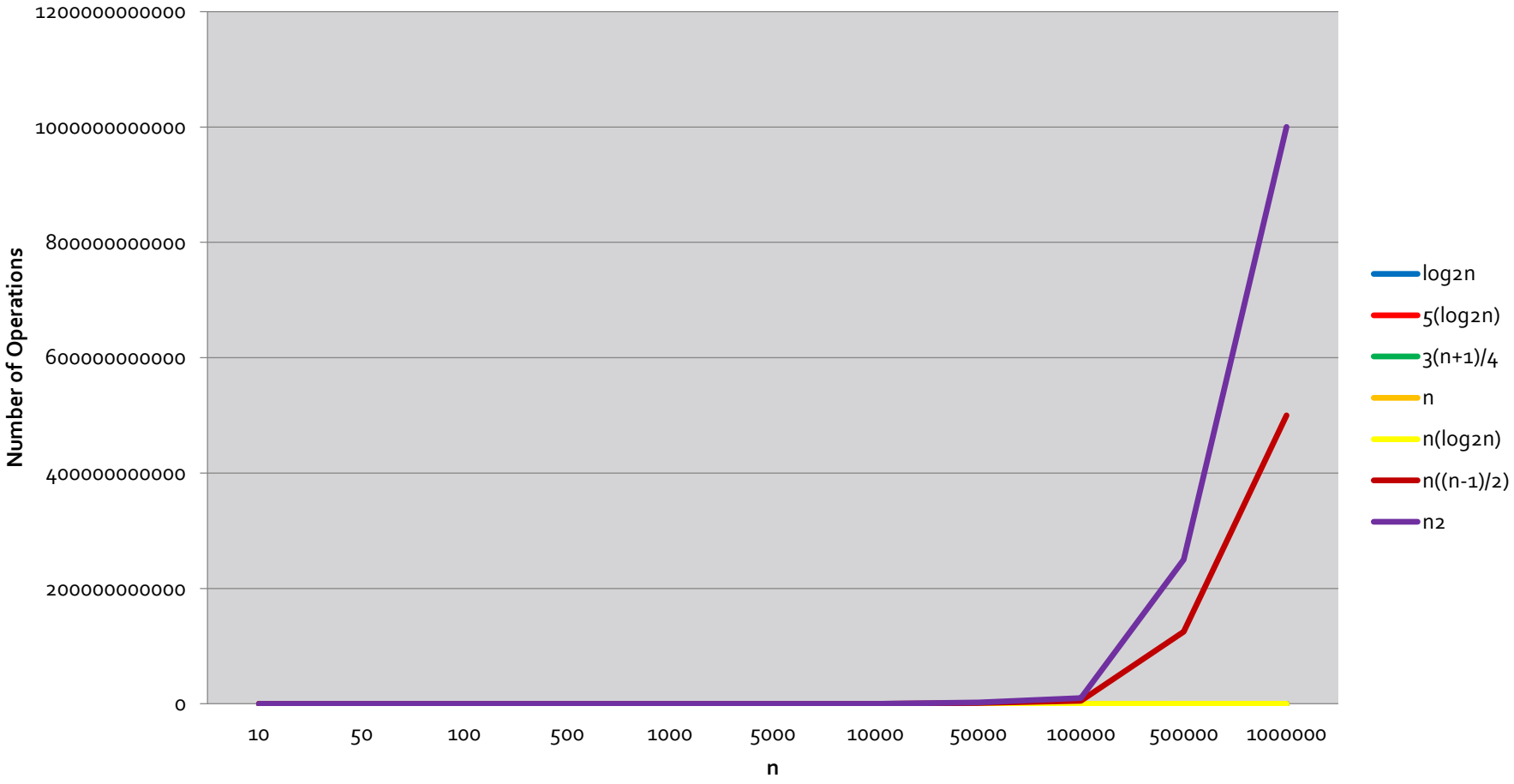
Not Much Bigger *n*

n^2 and $n(n-1)/2$ are growing much faster than any of the others



n from 10 to 1,000,000

Hmm! Let's try a logarithmic scale ...



n from 10 to 1,000,000

Notice how *clusters* of growth rates start to emerge