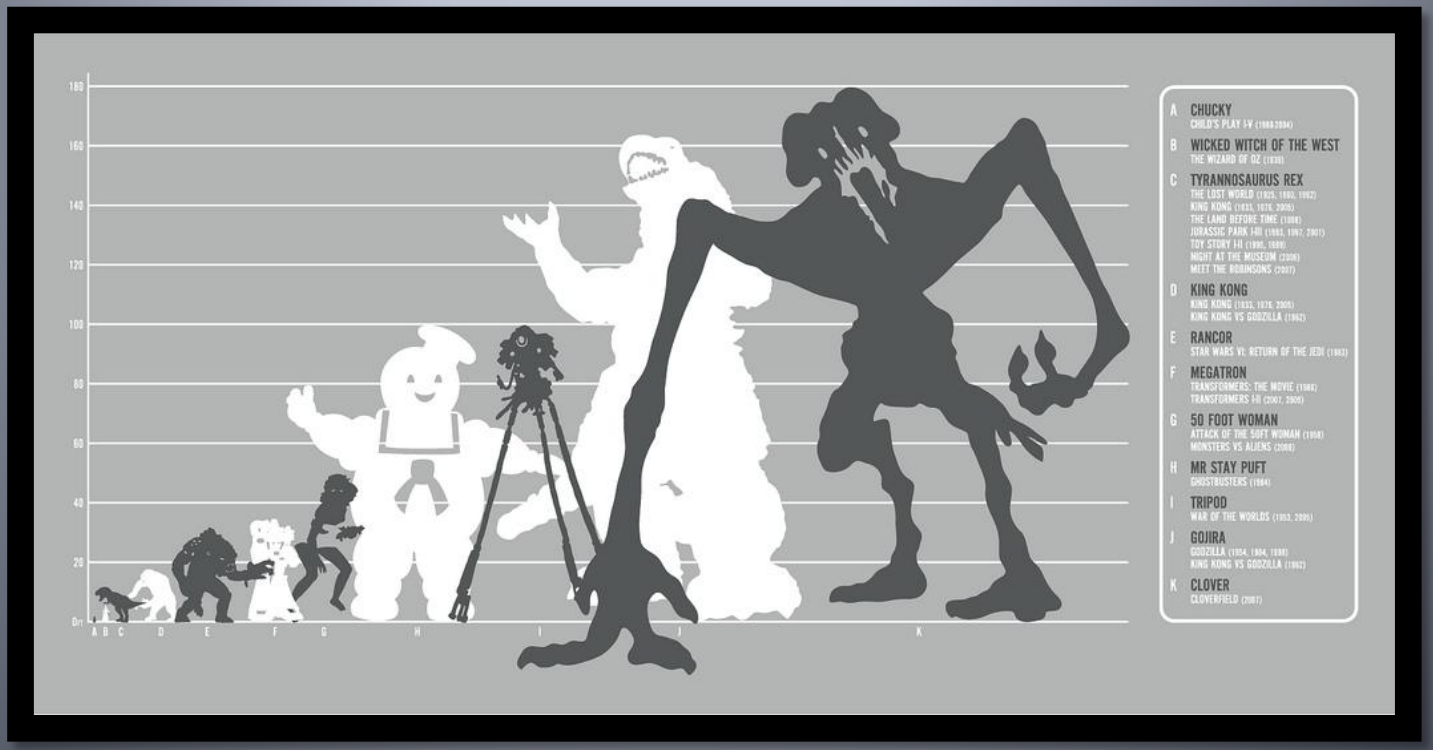
O Notation Part 5

O Notation

O Notation

- Counting comparisons
- O Notation
  - O notation's mathematical basis
  - O notation classes
  - Θ and Ω notations

Comparisons



Ignoring Leading Constants

- Calculation of a detailed cost function can be onerous and dependent on
  - Exactly how the algorithm was implemented
    - Implementing selection sort as a single function would have resulted in a different cost function
  - The definition of a single discrete operation
    - How many operations is this: *mid* = (*low* + *high*) / 2?
- We are often interested in how algorithms behave as the problem size increases

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
  - Sort with selection sort: *quadratic* growth

Comparing Algorithms

- Measuring the performance of an algorithm can be simplified by
  - Only considering the highest order term
    - i.e. only consider the number of times that the barometer instruction is executed
  - And ignoring leading constants
- Consider the selection sort algorithm
  - $t_{\text{selection sort}} = 2n^2 - 2n + 10(n-1) + 2$
  - Its cost function approximates to n^2

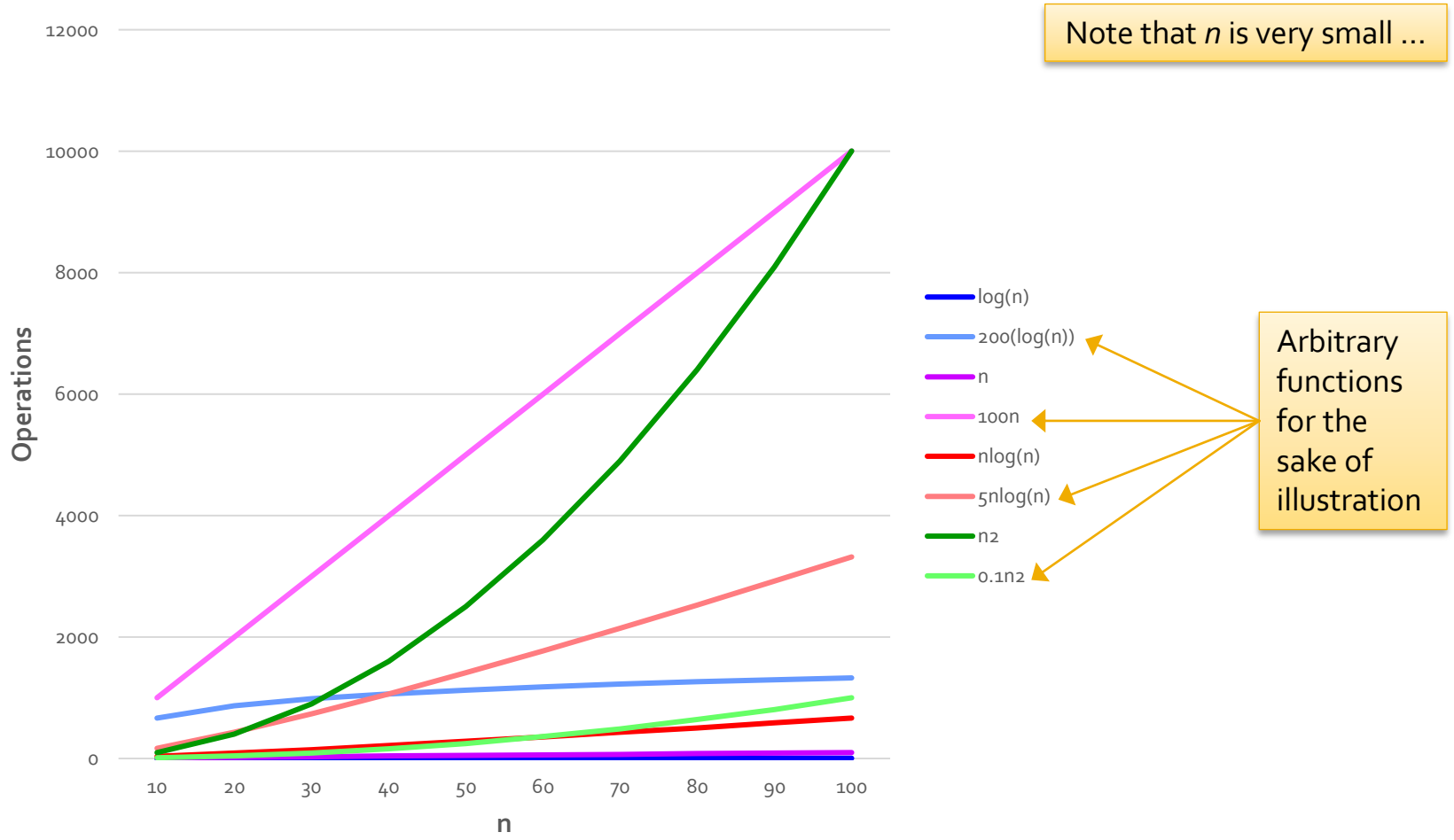
Algorithm Summary

- What are the approximate number of barometer operations for the algorithms we looked at?
  - Ignoring leading constants
- Linear search: n — worst and average case
- Binary search: $\log_2 n$ — worst and average case
- Selection sort: n^2 — all cases
- Insertion sort: n^2 — worst and average case
- Quicksort: $n(\log_2(n))$ — best and average case

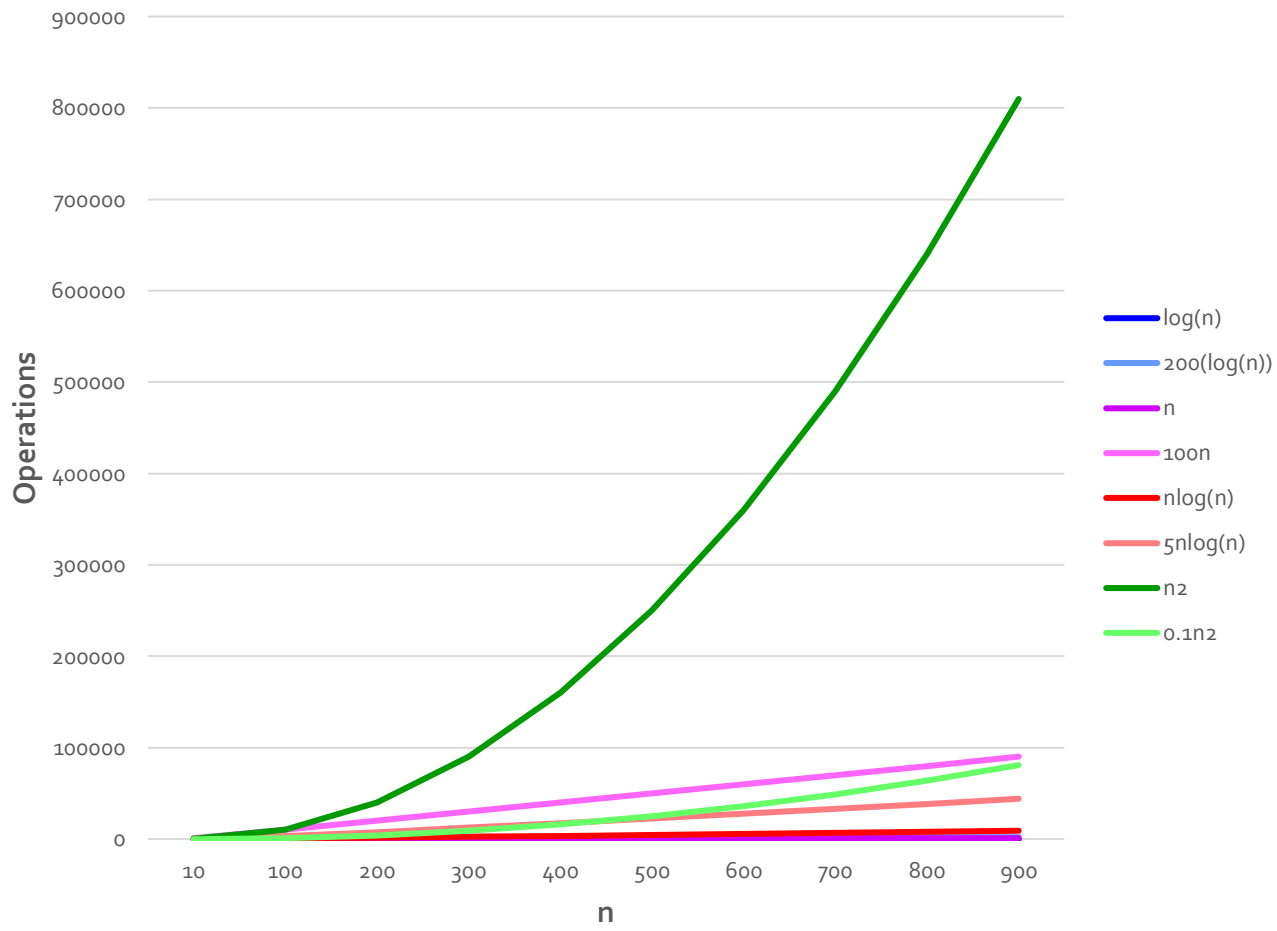
Algorithm Growth Rate

- What do we want to know when comparing two algorithms?
  - Often, the most important thing is how quickly the time requirements increase with input size
  - e.g. If we double the input size how much longer does an algorithm take?
- Here are some graphs …

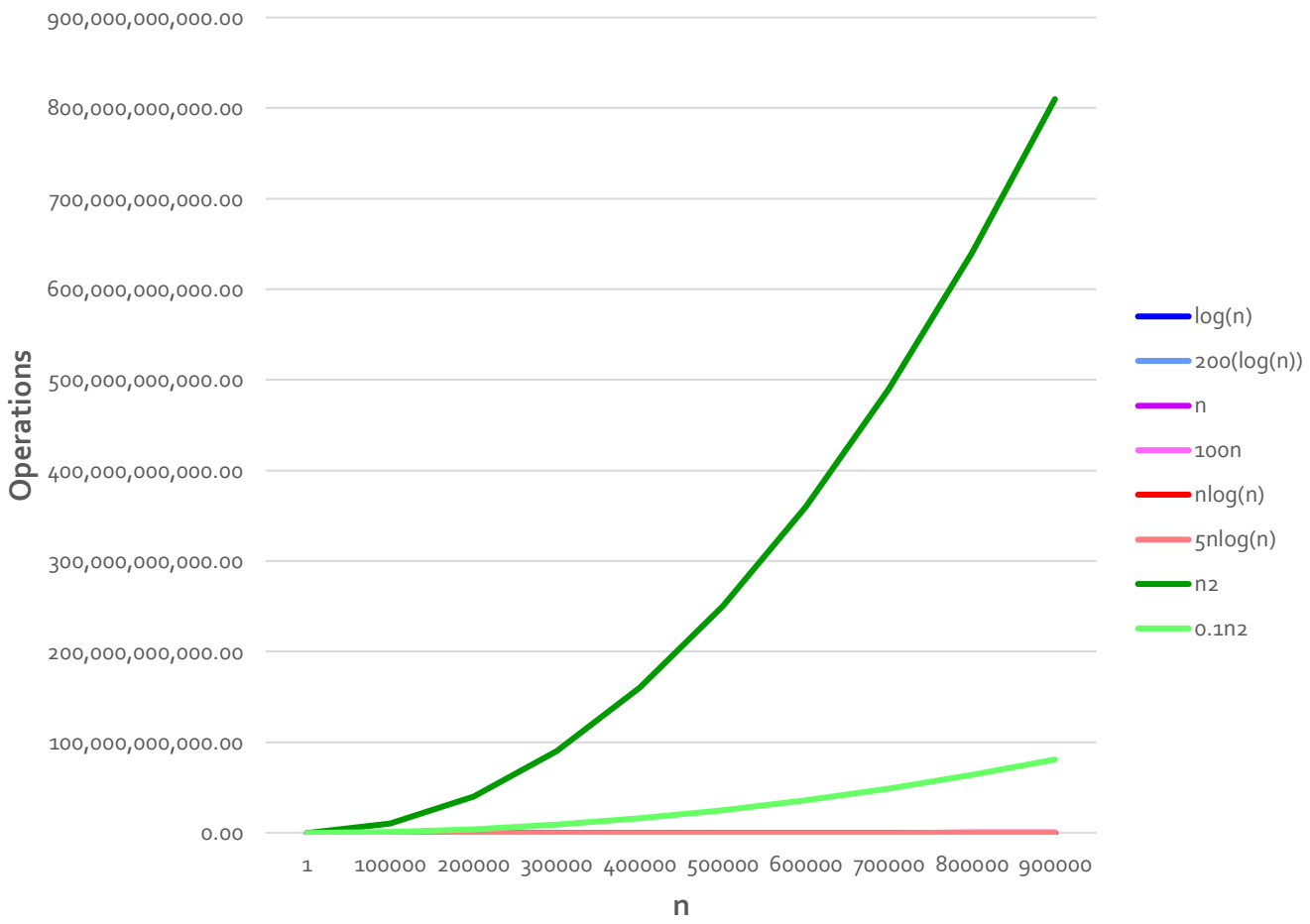
Small *n*


12000
10000
8000
6000
4000
2000
0
Operations
10
20
30
40
50
60
70
80
90
100
n
log(n)
200(log(n))
n
100n
nlog(n)
5nlog(n)
n2
0.1n2
Note that *n* is very small ...
Arbitrary functions for the sake of illustration

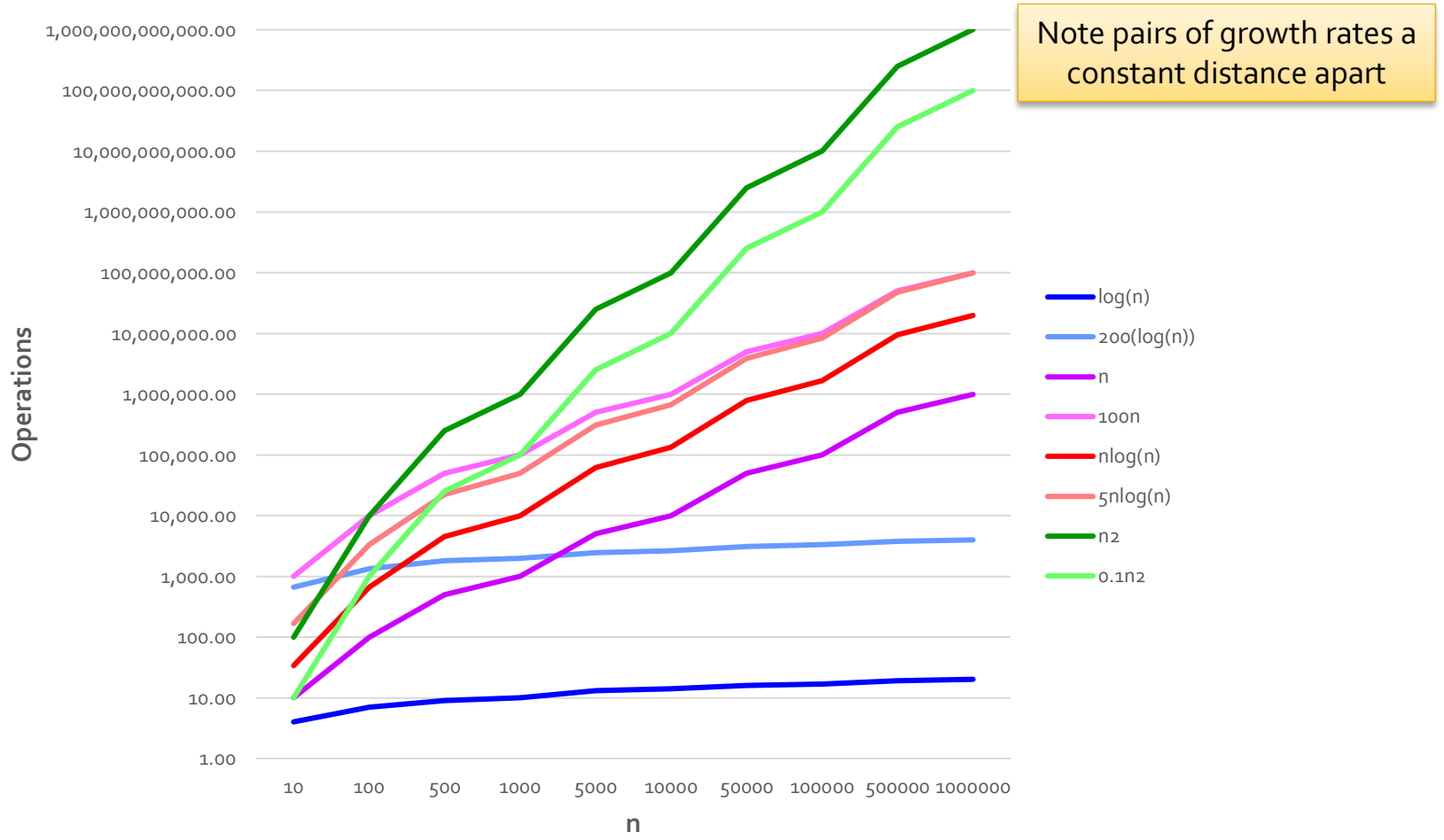
Larger *n*


900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Operations
10
100
200
300
400
500
600
700
800
900
n
log(n)
200(log(n))
n
100n
nlog(n)
5nlog(n)
n2
0.1n2

Much Larger *n*



Logarithmic Scale



Note pairs of growth rates a constant distance apart

O Notation

O Notation Introduction

- Exact counting of operations is often difficult (and tedious), even for simple algorithms
  - And is often not much more useful than estimates due to the relative importance of other factors
- *O Notation* is a mathematical language for evaluating the running-time of algorithms
  - O-notation evaluates the *growth rate* of an algorithm

Example of a Cost Function

- Cost Function: $t_A(n) = n^2 + 20n + 100$
  - Which term in the function is the most important?
- It depends on the size of n
  - $n = 2$, $t_A(n) = 4 + 40 + \underline{100}$
    - The constant, 100, is the dominating term
  - $n = 10$, $t_A(n) = 100 + \underline{200} + 100$
    - $20n$ is the dominating term
  - $n = 100$, $t_A(n) = \underline{10{,}000} + 2{,}000 + 100$
    - n^2 is the dominating term
  - $n = 1000$, $t_A(n) = \underline{1{,}000{,}000} + 20{,}000 + 100$
    - n^2 is still the dominating term

The general idea is …

- Big-*O* notation does not give a precise formulation of the cost function for a particular data size
- It expresses the general behaviour of the algorithm as the data size *n* grows very large so ignores
  - lower order terms and
  - constants
- A Big-*O* cost function is a simple function of *n*
  - n, n^2, $\log_2(n)$, etc.

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, $f(N) = 2n^2 - 2n + 10(n-1) + 2$
  - i.e. $O(n^2)$

But …

- 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

O Notation, More Formally

- Given some function $t(n)$ t(n) might be a detailed cost function …
  - Say that $t(n) = O(f(n))$ if $t(n)$ is at most a constant times $f(n)$ Where f(n) is a simple function: n^2, $n \log n$, etc.
    - 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 $g(n)$, $k*g(n) = O(f(n))$
    - e.g., $5n = O(n)$

Why Big O?

- An algorithm is said to be *order* $f(n)$
  - Denoted as $O(f(n))$
- The function $f(n)$ is the algorithm's growth rate function
  - If a problem of size n requires time proportional to n then the problem is $O(n)$
    - e.g. If the input size is doubled so is the running time

Big O Notation Definition

- An algorithm is *order* $f(n)$ if there are positive constants k and m such that
  - $t_A(n) \leq k * f(n)$ for all $n \geq m$
    - i.e. find constants k and m such that the cost function is less than or equal to k * a simpler function for all n greater than or equal to m
- If so, we would say that $t_A(n)$ is $O(f(n))$

Constants *k* and *m*

- Finding a constant $k \mid t_A(n) \leq k * f(n)$ shows that t is $O(f(n))$
  - e.g. If the cost function was $n^2 + 20n + 100$ and we believed this was $O(n)$
    - We claim to be able to find a constant $k \mid t_A(n) \leq k * f(n)$ for all values of n
    - Which is not possible
- For some small values of n lower order terms may dominate
  - The constant m addresses this issue

Or In English…

- The idea is that a cost function can be approximated by another, simpler, function
  - The simpler function has 1 variable, the data size n
  - This function is selected such that it represents an *upper bound* on the value of $t_A(n)$
- Saying that the time efficiency of algorithm A $t_A(n)$ is $O(f(n))$ means that
  - A cannot take more than $O(f(n))$ time to execute, and
  - The cost function $t_A(n)$ *grows at most as* fast as $f(n)$

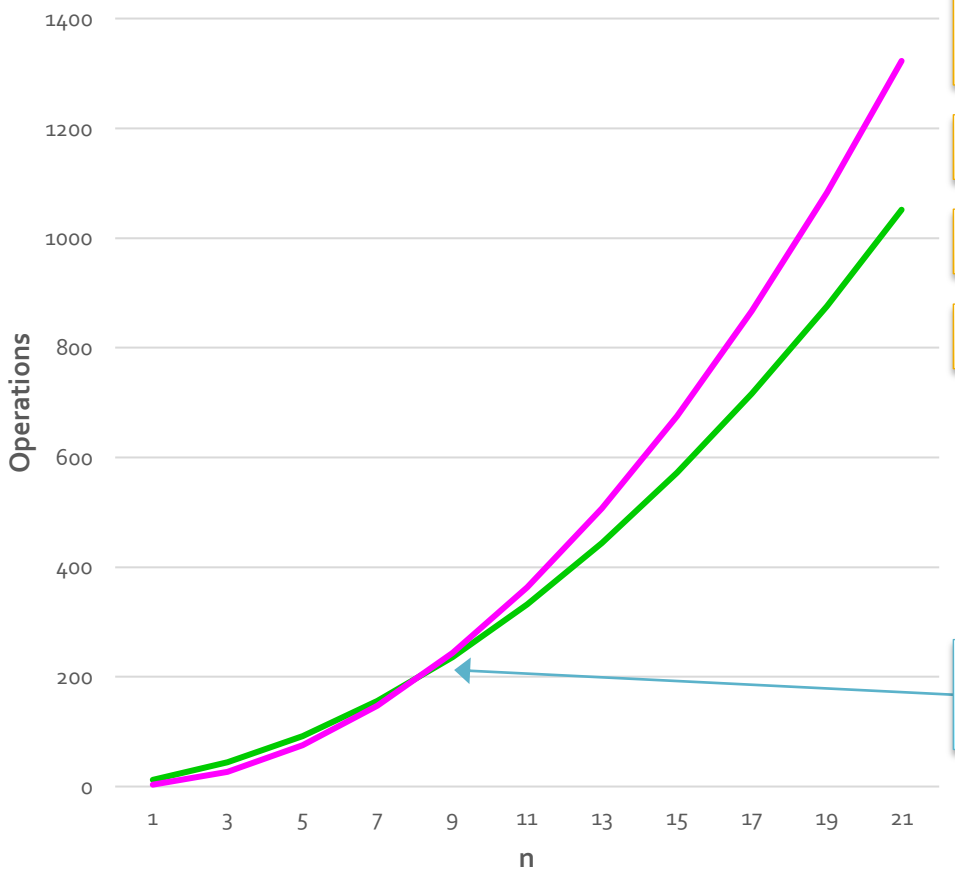
Big O Example

- An algorithm's cost function is $3n + 12$
  - If we can find constants m and k such that:
  - $k * n \geq 3n + 12$ for all $n \geq m$ then
  - The algorithm is $O(n)$
- Find values of k and m so that this is true
  - $k = 4$, and
  - $m = 12$ then
  - $4n \geq 3n + 12$ for all $n \geq 12$

Another Big O Example

- A cost function is $2n^2 - 2n + 10(n-1) + 2$
- If we can find constants m and k such that:
  - $k * n^2 \geq 2n^2 - 2n + 10(n-1) + 2$ for all $n \geq m$ then
  - The algorithm is $O(n^2)$
- Find values of k and m so that this is true
  - $k = 3$, and
  - $m = 9$ then
  - $3n^2 > 2n^2 - 2n + 10(n-1) + 2$ for all $n \geq 9$

And Another Graph



This is the graph of the selection sort cost function and $3n^2$

Demonstrating that selection sort is $O(n^2)$

For all $n \geq 9$, $3(n^2) \geq 2n^2 - 2n + 10(n-1) + 2$

i.e. $k = 3$ and $m = 9$

After this point $3n^2$ is always going to be larger than $2n^2 - 2n + 10(n-1) + 2$

O Notation Examples

- All these expressions are $O(n)$
  - n, $3n$
  - $61n + 5$
  - $22n - 5$
- All these expressions are $O(n^2)$
  - n^2
  - $9n^2$
  - $18n^2 + 4n - 53$
- All these expressions are $O(n \log n)$
  - $n(\log n)$
  - $5n(\log 99n)$
  - $18 + (4n - 2)(\log (5n + 3))$

Arithmetic and O Notation

- $O(k * f) = O(f)$ if k is a constant
  - e.g. $O(23 * O(\log n))$, simplifies to $O(\log n)$
- $O(f + g) = \max[O(f), O(g)]$
  - $O(n + n^2)$, simplifies to $O(n^2)$
- $O(f * g) = O(f) * O(g)$
  - $O(m * n)$, equals $O(m) * O(n)$
  - Unless there is some known relationship between m and n that allows us to simplify it, e.g. $m < n$

Typical Growth Rate Functions

- Growth rate functions are typically one of the following
  - $O(1)$
  - $O(\log n)$
  - $O(n)$
  - $O(n*\log n)$
  - $O(n^2)$
  - $O(n^k)$
  - $O(2^n)$

O(1) – Constant Time

- We write *O*(1) to indicate something that takes a constant amount of time
  - Array look up
  - Swapping two values in an array
  - Finding the minimum element of an *ordered* array takes *O*(1) time
    - The minimum value is either the first or the last element of the array
  - Binary or linear search best case
- *Important*: constants can be large
  - So, in practice *O*(1) is not *necessarily* efficient
  - It tells us is that the algorithm will run at the same speed no matter the size of the input we give it

O(log*n*) – Logarithmic Time

- *O*(log*n*) algorithms run in *logarithmic* time
  - Binary search average and worst case
  - The logarithm is assumed to be base 2 unless specified otherwise
- Doubling the size of *n* increases the number of operations by one
- Algorithms might be O(log*n*)
  - If they are divide and conquer algorithms that halve the search space for each loop iteration or recursive call

O(*n*) – Linear Time

- *O*(*n*) algorithms run in *linear* time
  - Linear search
  - Summing the contents of an array
  - Traversing a linked list
  - Insertion sort or bogo sort best case
- Doubling the size of *n* doubles the number of operations
- Algorithms might be O(*n*)
  - If they contain a single loop
    - That iterates from 0 to *n* (or some variation)
  - Make sure that loops only contain constant time operations
    - And evaluate any function calls

O(*n*log*n*)

- *O*(*n*log*n*)
  - Mergesort in all cases
  - Heap sort in all cases
  - Quicksort in the average and best case
  - O(*n*log*n*) is the best case for comparison sorts
    - We will not prove this in CMPT 225
- Growth rate is faster than linear but still slow compared to O(n^2)
- Algorithms are O(*n*log*n*)
  - If they have one process that is linear that is repeated O(log*n*) times

$O(n^2)$ – Quadratic Time

- $O(n^2)$ algorithms run in *quadratic* time
  - Selection sort in all cases
  - Insertion sort in the average and worst case
  - Bubble sort in all cases
  - Quicksort worst case
- Doubling the size of n quadruples the number of operations
- Algorithms might be $O(n^2)$
  - If they contain nested loops
    - As usual make sure to check the number of iterations in such loops
    - And that the loops do not contain non-constant function calls

$O(n^k)$ – Polynomial Time

- $O(n^k)$ algorithms are referred to as running in *polynomial* time
  - If k is large, they can be very slow
- Doubling the size of n increases the number of operations by 2^k

$O(2^n)$ – Exponential Time

- $O(2^n)$ or $O(k^n)$ algorithms are referred to as running in *exponential* time
- Very slow, and if there is no better algorithm implies that the problem is intractable
  - That is, problems of any reasonable size cannot be solved in a reasonable amount of time
    - Such as over the lifetime of the human race …
- $O(n!)$ algorithms are even slower
  - Traveling Salesman Problems (and many others)
  - Bogo sort in the average case

Worst, Average and Best Case

- The *O* notation growth rate of some algorithms varies depending on the input
- Typically, we consider three cases:
  - *Worst case*, usually (relatively) easy to calculate and therefore commonly used
  - *Average case*, often difficult to calculate
  - *Best case*, usually easy to calculate but less important than the other cases
    - Relying on the input having the best-case organization is not generally a good idea

Other Related Notations

- Ω (omega) notation
  - Gives a lower bound
    - Whereas O notation is an upper bound
  - There exists some constant k, such that $k * f(n)$ is a lower bound on the cost function $g(n)$
- Θ (theta) notation
  - Gives an upper and lower bound
  - $k_1 * f(n)$ is an upper bound on $g(n)$ and $k_2 * f(n)$ is a lower bound on $g(n)$

Upper and Lower Bound