Asymptotic Analysis

Dr. Mark Matthews

Algorithms

in which

Learn about order of growth classifications and we meet the three major notations: Big O, Big-Theta, and Big-Omega

Growth Rate

In complexity analysis we disregard the implementation and environmental details.

We only care about what happens when the input grows large

The hardware / software and any implementation details affects our algorithm's performance by a constant factor, but does NOT affect the growth rate.

How does our algorithm perform when put under pressure?

Asymptotic analysis

- Focusing on the largest growing term is what we call “asymptotic behaviour”
- By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—without getting pulled down into details that complicate our understanding.
- When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**.

- Mathematically speaking: we are interested in the limit of our function (f) as it moves towards infinity.

We only care about the dominant term that affects the growth

- Drop all the terms that grow slowly.
- Only keep the ones that grow fast as N becomes larger and therefore affect the performance of the algorithm.

```
linear_search (array, key){

(for i=0; i < array.length; i++){

        if([array[i] == key){
            return i;
             }
}
}
```

1 + N+1 + N

2N

T(N) = 4N + 2

Guessing Game: Let's look at an example

The maximum number of times the for-loop can run is N (worst case) when the value is not in the array

```
linear_search (array, guess){

(for i=0; i < array.length; i++){

        if([array[i] == guess){
            return i;
             }
}
Return -1;
}
```

Within the loop we perform:

- Comparisons
- Possibly value return
- Array access
- Increment

Guessing Game: Let's look at an example

The maximum number of times the for-loop can run is N (worst case) when the value is not in the array

```
linear_search (array, guess){

(for i=0; i < array.length; i++){

        if([array[i] == guess){
            return i;
             }
}
Return -1;
}
```

We can say these details within the loop = c1

We can add overhead for setting up the loop & returning the value = c2

This gives us: c1*N + c2

c1 & c2 depend on implementation details so we drop them

Guessing Game: Let's look at an example

The maximum number of times the for-loop can run is N (worst case) when the value is not in the array

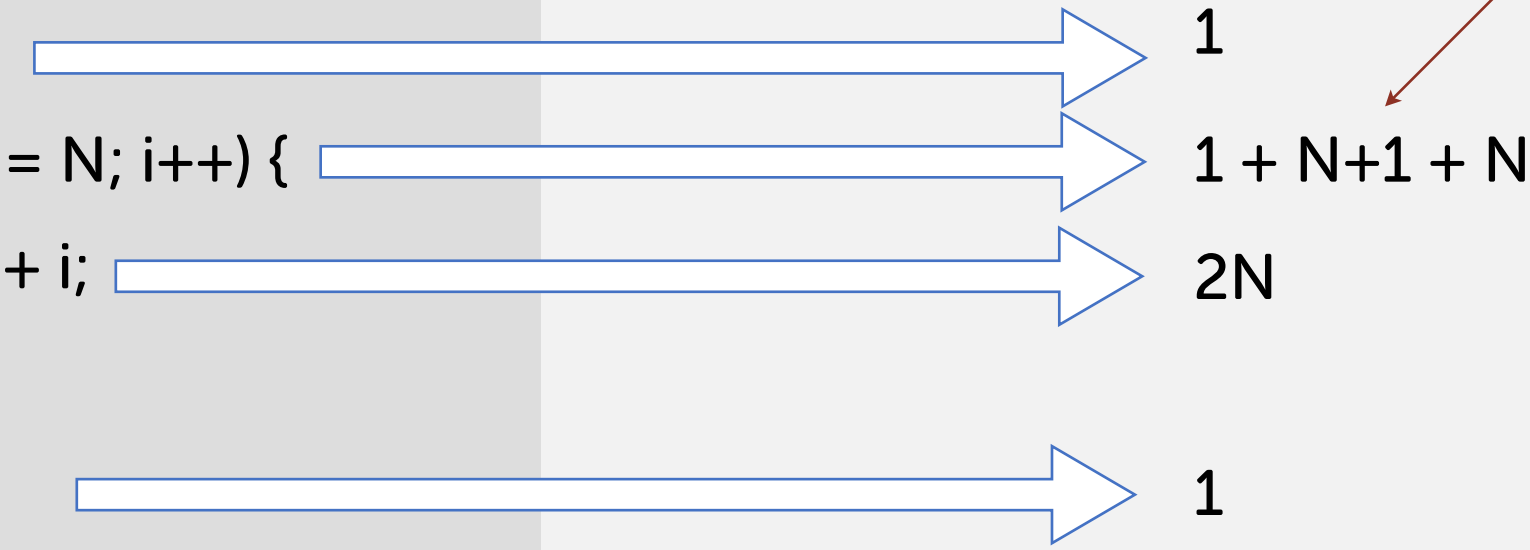
```
linear_search (array, guess){

(for i=0; i < array.length; i++){

        if([array[i] == guess){
            return i;
             }
}
Return -1;
}
```

The growth of our linear_search algorithm in its worst case grows depends on the array size (or the input).

We call this O(N)

sumUp: Implementation details give us different results

```
sum_upv1(N)
{
sum = 0;
for (i = 1; i <= N; i++) {
sum = sum + i;
}
return sum;
}
```



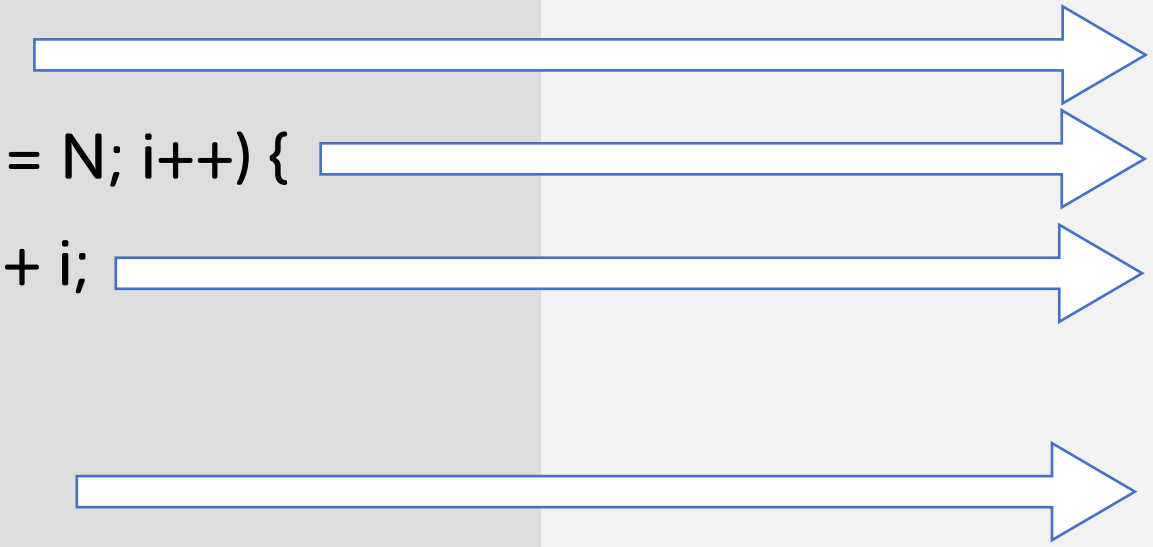
The condition in the for loop checks N+1 times counting the last time when it is no longer <= to n and exits

T(N) = 4N + 4

T(N) = time taken by the sum_up algorithm in the worst case

sumUp: Implementation details give us different results

```
sum_upv1(N)
{
sum = 0;
for (i = 1; i <= N; i++) {
sum = sum + i;
}
return sum;
}
```



1

1 + N+1 + N

2N

1

The condition in the for loop checks N+1 times counting the last time when it is no longer <= to n and exits

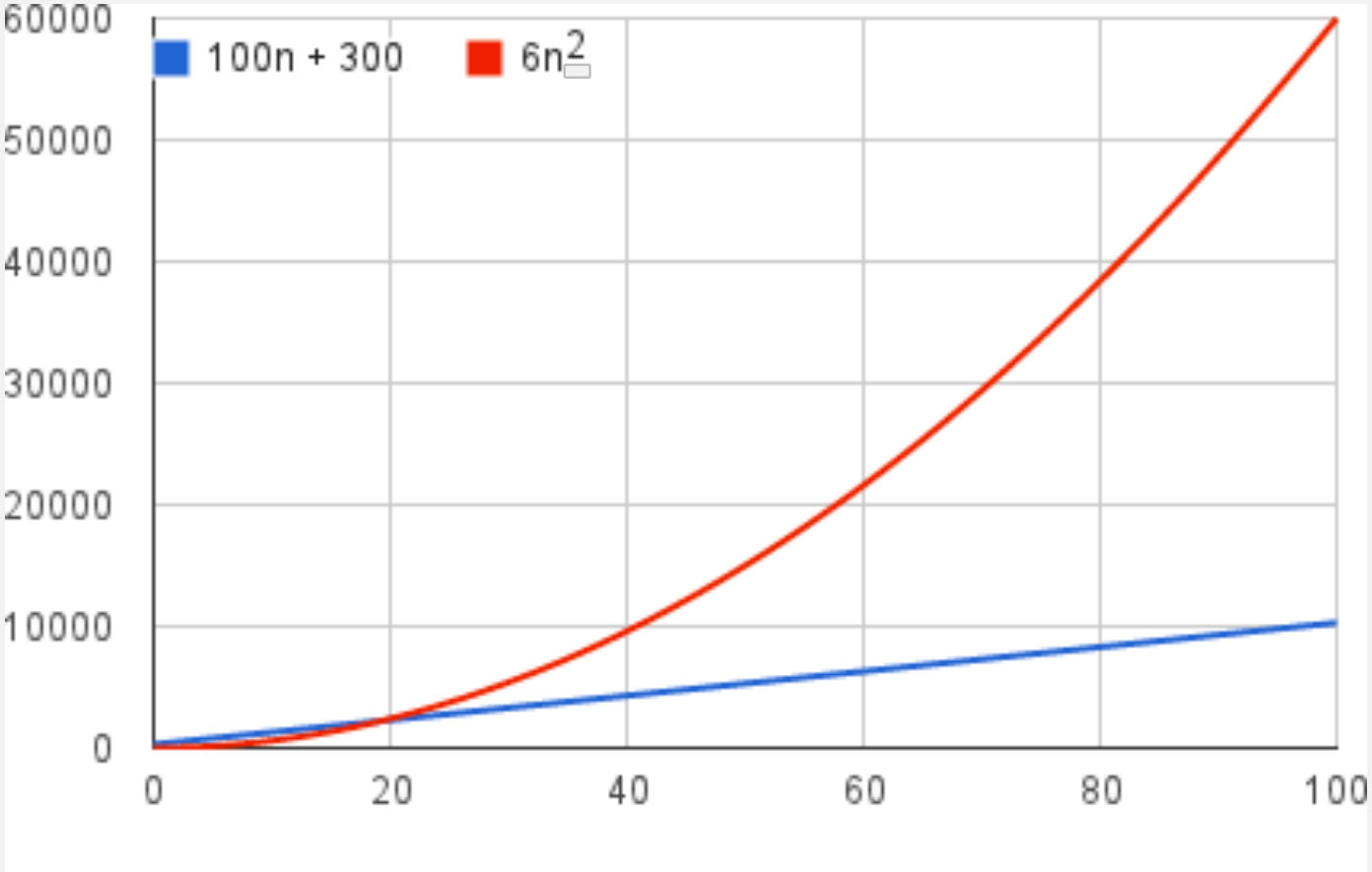
T(N) = 4N + 4

sumUp is O(N)

Sample Algorithm

Algorithm T(N): $6n^2 + 100n + 300$

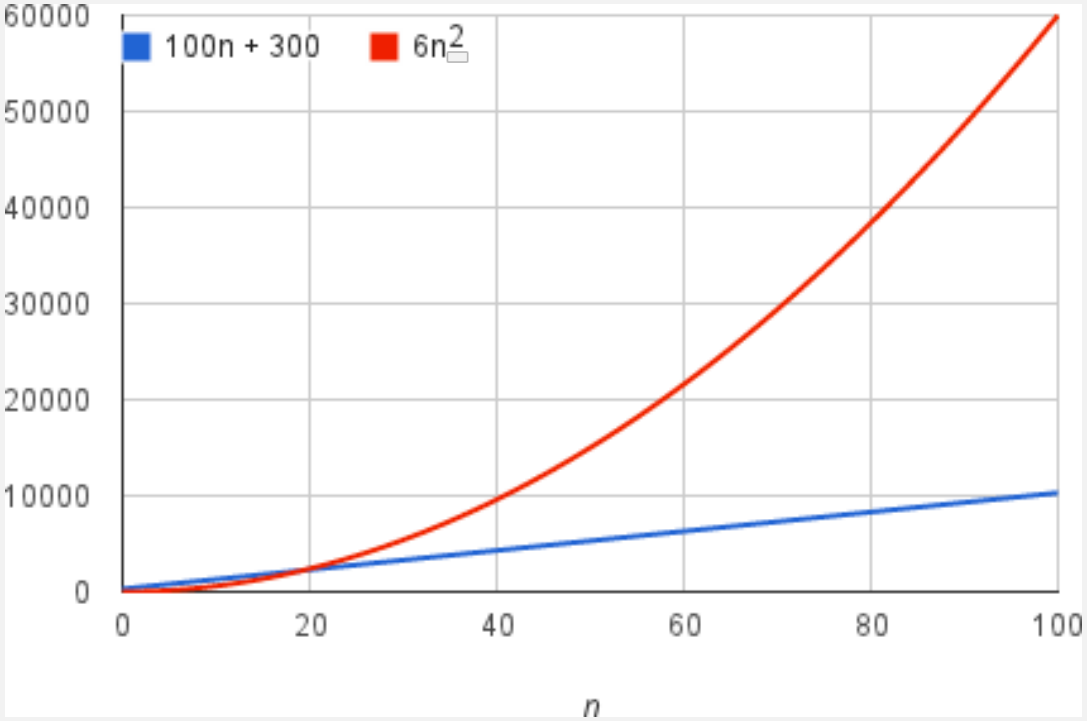
At some point (here it is 20) $6n^2$ becomes larger.



Sample Algorithm

Algorithm T(N): $6n^2$, + 100n + 300

In fact it doesn't matter what the coefficients are as long as the running time is $an^2 + bn + c$ for numbers a>0, b and c - there will always be a value where n^2 is greater than the other terms for ever more.



Summary

By dropping the less significant terms and constants we can measure the Algorithm's growth rate without needing to get into implementation details.

We call this **asymptotic analysis** and we use 3 main forms:

1. Big- Θ Notation
2. Big O Notation
3. Big Ω Notation

Asymptotic Algorithm Analysis

- We find the worst-case number of primitive operations executed as a function of the input size
- We express this function with big-Oh notation
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Aside: Pseudocode Analysis Rules

For loops

The running time of a for loop is at most the running time of the statements inside the loop times the number of loops

```
for (int I = 0; I < n; i++){
doSomething();
}
```

O(n)

Pseudocode Analysis Rules

Nested loops

Do the analysis inside out. The total running time of a statement inside a group of nested loops is the running time of the statement times the product of the sizes of all the loops

```
for (int i = 0; i < n; i++){
For (int j = 0; j <n; j++){
doSomething();
}
}
```

$O(n^2)$

Pseudocode Analysis Rules

Consecutive Statements

These simply add - the maximum is the one that counts.

```
for (int I = 0; I < n; i++){
doSomething();}
```

$O(n)$

```
for (int i = 0; i < n; i++){
For (int j = 0; j <n; j++){
doSomething();
} }
```

$O(n^2)$

Total = $O(n) + O(n^2) = O(n^2)$

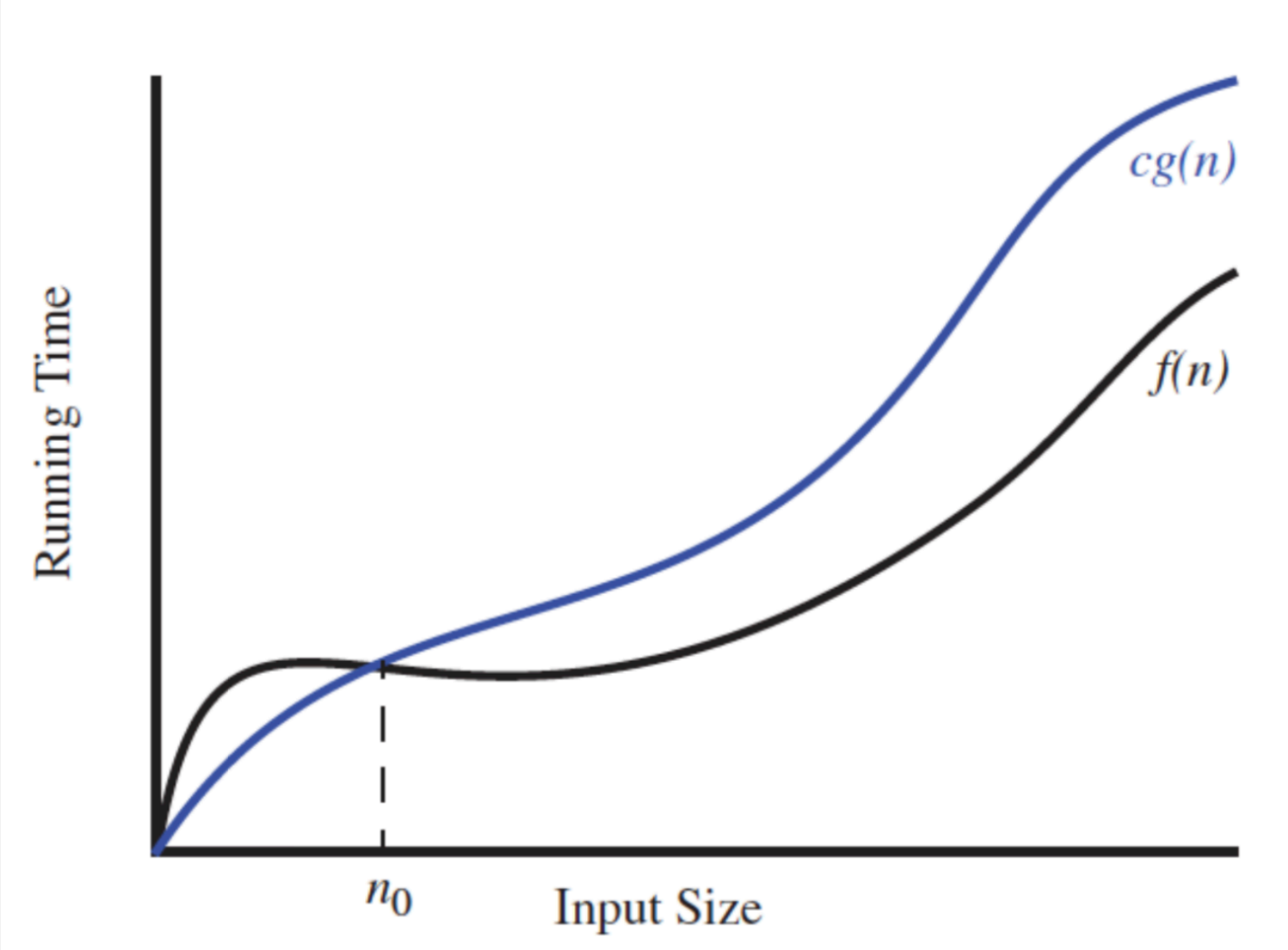
Big-Oh Definition

- T(N) = function on n = 1, 2, 3
- We focus usually on the worst-case running time of an algorithm
- We would like to have asymptotic notation that tells us that our algorithm's running time grows at most "**this much**" (but it could grow more slowly)

"Eventually, for all sufficiently large n, T(n), is bounded above by a constant multiple of f(n)"

“Big-Oh” Graph

f (n) is O(g(n))



“Big-Oh” Notation

What do we mean when we say T(N) = O(f(n)) or T(N) is Big O of f(N)?

Given two functions f(n) and g(n), we say that

$$f(n) \text{ is } O(g(n))$$

If and only if there are positive constants c and n0 such that

$$f(n) \leq cg(n) \ \forall \ n \geq n_0$$

We say “f(n) is Big-Oh of g(n)”

“Big-Oh” Example

Let’s say we have an algorithm with running time: 11n + 5

f(n) = 11n + 5

If we can show that g(n) = n satisfies:

f(n) <= c*g(n) for all n>= n0

Constants C and n0 need to be independent

* Not dependent on N

“Big-Oh” Example

$f(n) = 11n + 5 <= c*g(n)$

Let’s try C = 12

$11n + 5 <= 12n$

“Big-Oh” Example

$$f(n) = 11n + 5 <= c*g(n)$$

$$11n + 5 <= 12n$$

$$5 <= 12n - 11n$$

$$5 <= n$$

“Big-Oh” Example

$$f(n) = 11n + 5 <= c*g(n)$$

$$11n + 5 <= 12n$$

$$5 <= 12n - 11n$$

$$5 <= n$$

$$f(n) \leq 12g(n) \; \forall \; n \geq 5$$

“Big-Oh” Example

We chose C = 12 but we could choose any real number >= 12

and any integer greater than or equal to 5 works for N

$$f(n) \leq 12g(n) \; \forall \; n \geq 5$$

“Big-Oh” Rules

- Drop lower-order terms
- Drop constant factors
- Use the smallest possible class of functions
- Say “2n is O(n)” instead of “2n is $O(n^2)$”
- Use the simplest expression of the class
- Say “3n + 5 is O(n)” instead of “3n + 5 is O(3n)

Examples

We find the asymptotic behaviour of the following functions by dropping the constants and keeping the factor that grows the fastest.

$f(n) = 10n + 120$

$f(n) = 1028$

$f(n) = n^2 + 3n + 112$

$f(n) = n^3 + 1999n + 1337$

Examples

We find the asymptotic behaviour of the following functions by dropping the constants and keeping the factor that grows the fastest.

$f(n) = 10n + 120 \longrightarrow f(n) = O(n)$

$f(n) = 1028 \longrightarrow f(n) = O(1)$

$f(n) = n^2 + 3n + 112 \longrightarrow f(n) = O(n^2)$

$f(n) = n^3 + 1999n + 1337 \longrightarrow f(n) = O(n^3)$

Try these ones yourself

We find the asymptotic behaviour of the following functions by dropping the constants and keeping the factor that grows the fastest.

$f(n) = n^6 + 3n$

$f(n) = 2n + 12$

$f(n) = 3n + 2n$

$f(n) = n^n + n$

Common order of growth functions include..

An approximation is good if it is similar to the actual function, but does not include lower order terms.

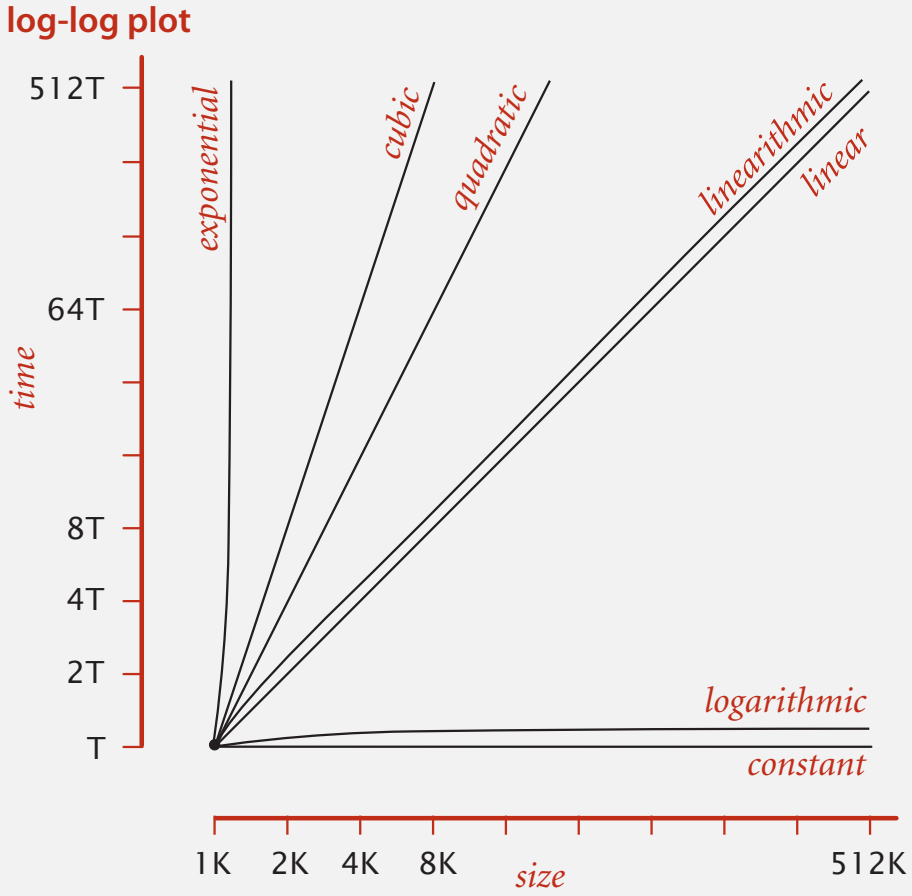
The hierarchy allows us to classify algorithms:

fixed: O(1)
logarithmic:O(log n)
linear: 0(n)
n log n: O(n log n)
quadratic: O ($_n$2)

polynomial:O(n^k), $k \geq 1$

exponential:O(a^n), $n > 1$

factorial:O(n!)

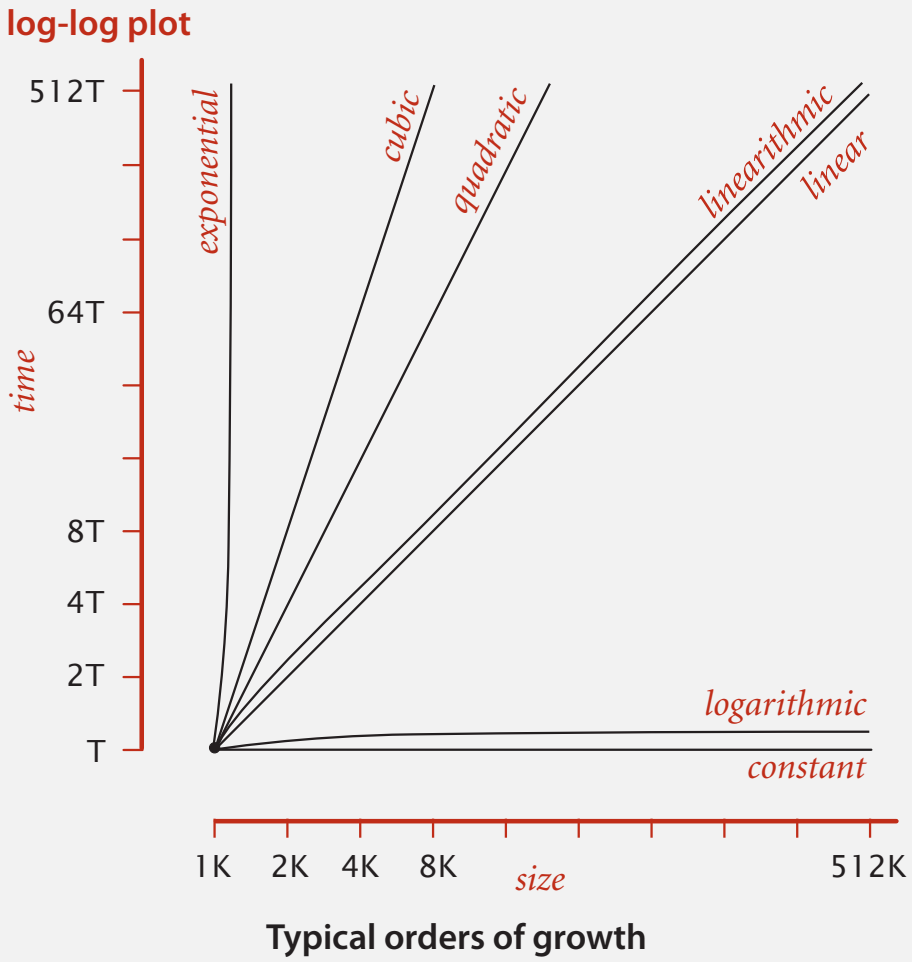


Typical orders of growth

Constant Time. O(N)

An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Example:

- Adding two numbers



Typical orders of growth

Logarithmic Time O(log n)

An algorithm is said to run in logarithmic time if its time execution is proportional to the square of the input size. Examples:

- binary search

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       if      (key < a[mid]) hi = mid - 1;
       else if (key > a[mid]) lo = mid + 1;
       else return mid;
   }
   return -1;
}
```

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's Arrays.binarySearch() discovered in 2006.

Ex 2. Compares for binary search.

Best: ~ 1

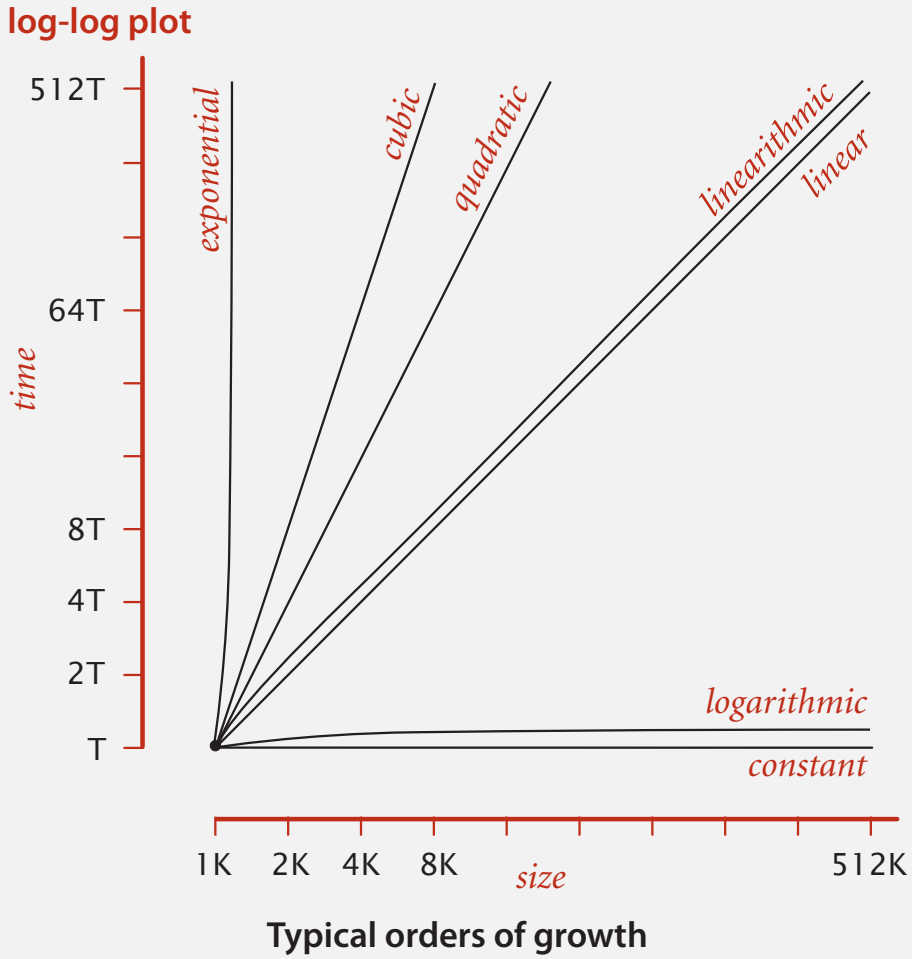
Average: $\sim \lg N$

Worst: $\sim \lg N$

Linear Time. O(n)

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases.
Example:

array: linear search, traversing, find minimum

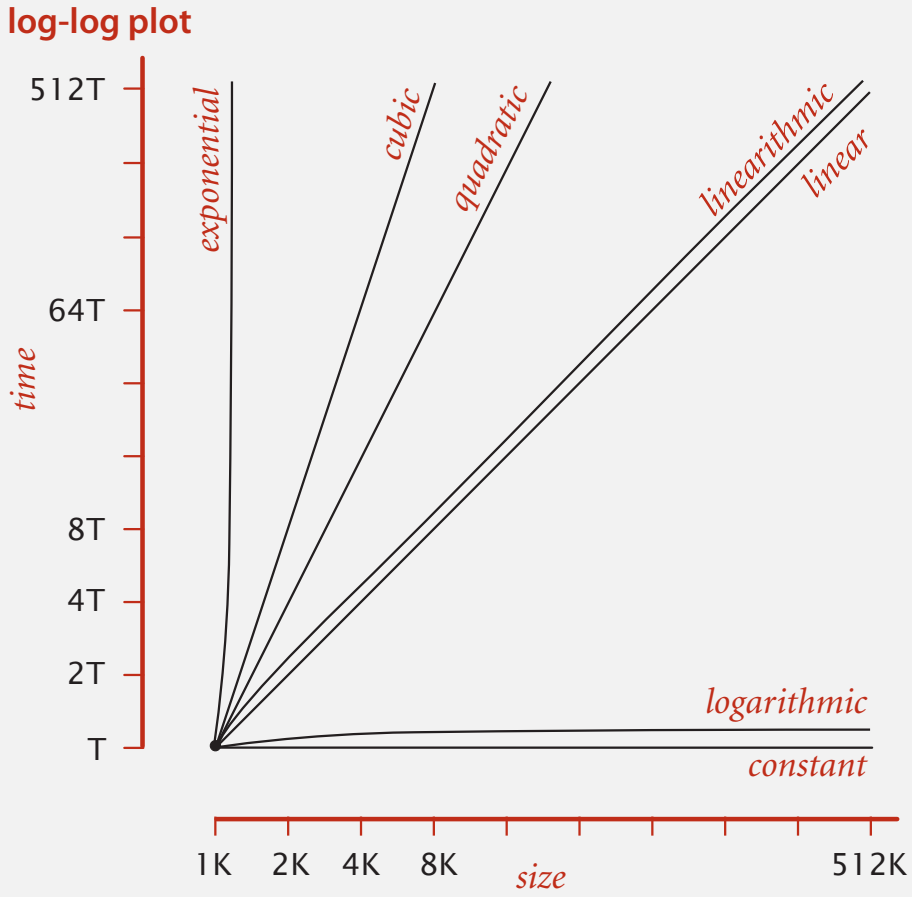


Typical orders of growth

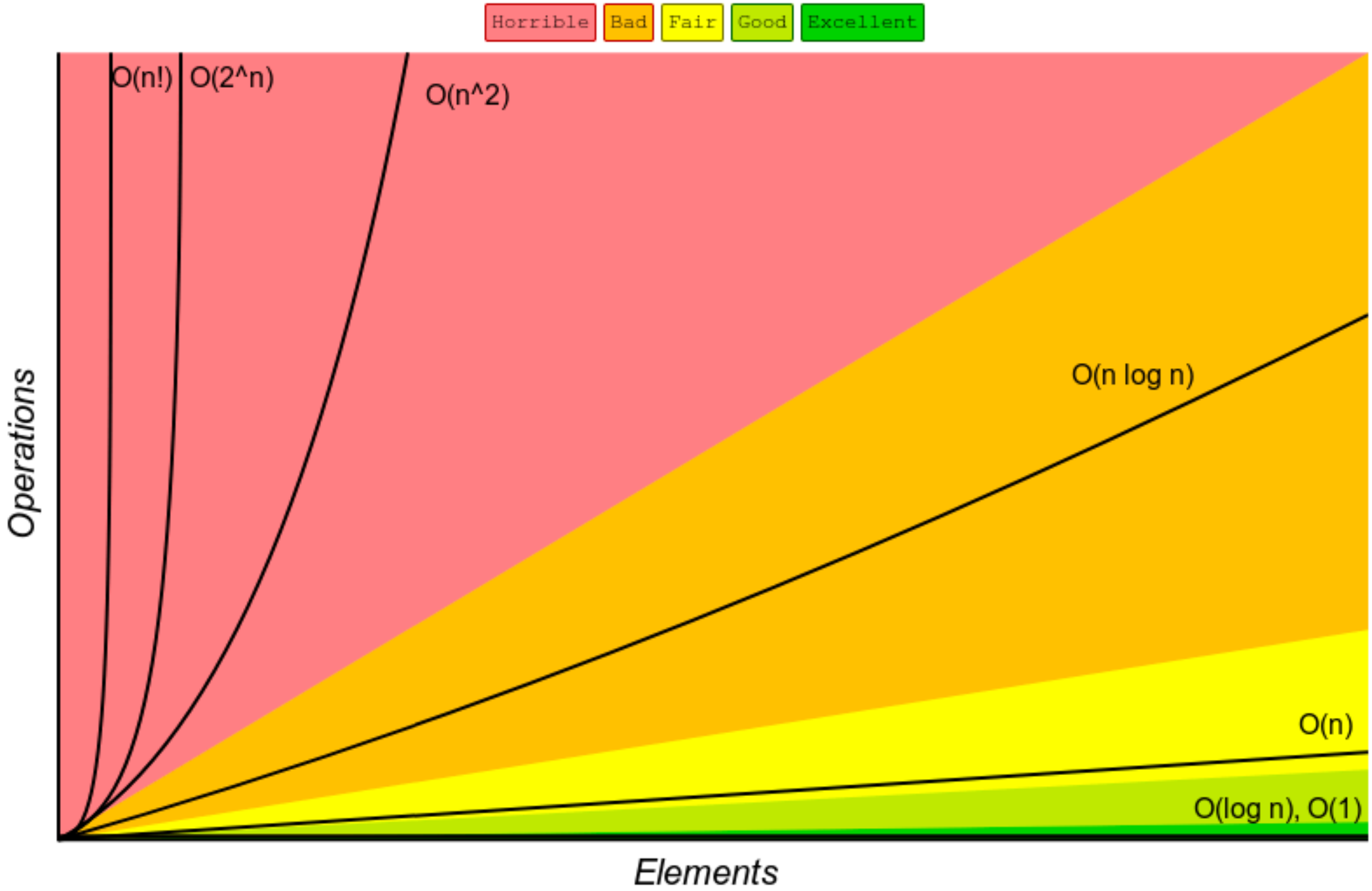
Quadratic function $O(n^2)$

An algorithm is said to run in quadratic time if its time execution is proportional to the square of the input size. Quadratic is part of the polynomial time functions O(n to power of k).

Example:
bubble sort, selection sort, insertion sort

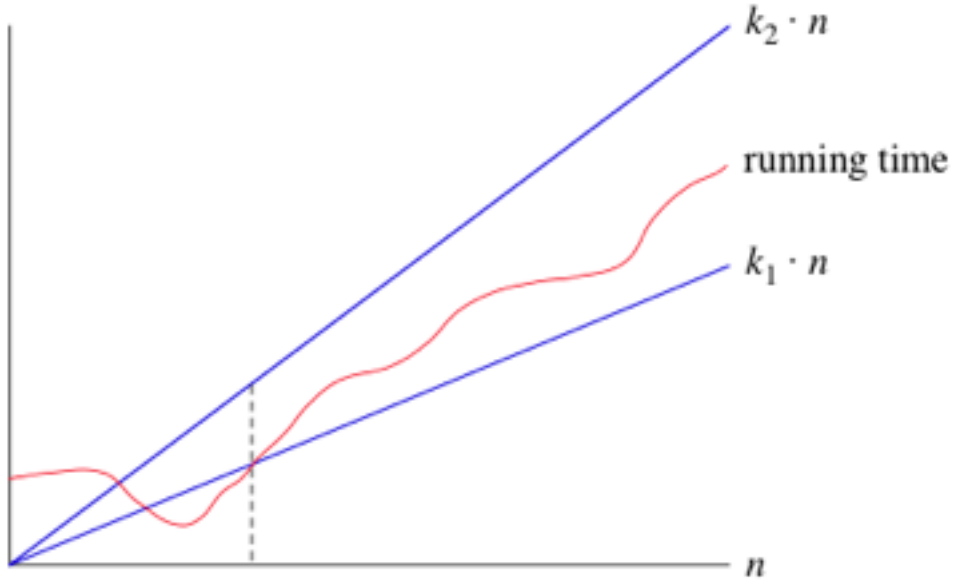


Typical orders of growth


Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Big-θ (Big-Theta) notation

- When we use big-Θ notation, we're saying that we have an asymptotically tight bound on the running time.
- "Tight bound" because we've nailed the running time to within a constant factor above and below



Running time: once gets large enough, the running time is at least k1 * N and at most K2 for some constants K1 and K2.

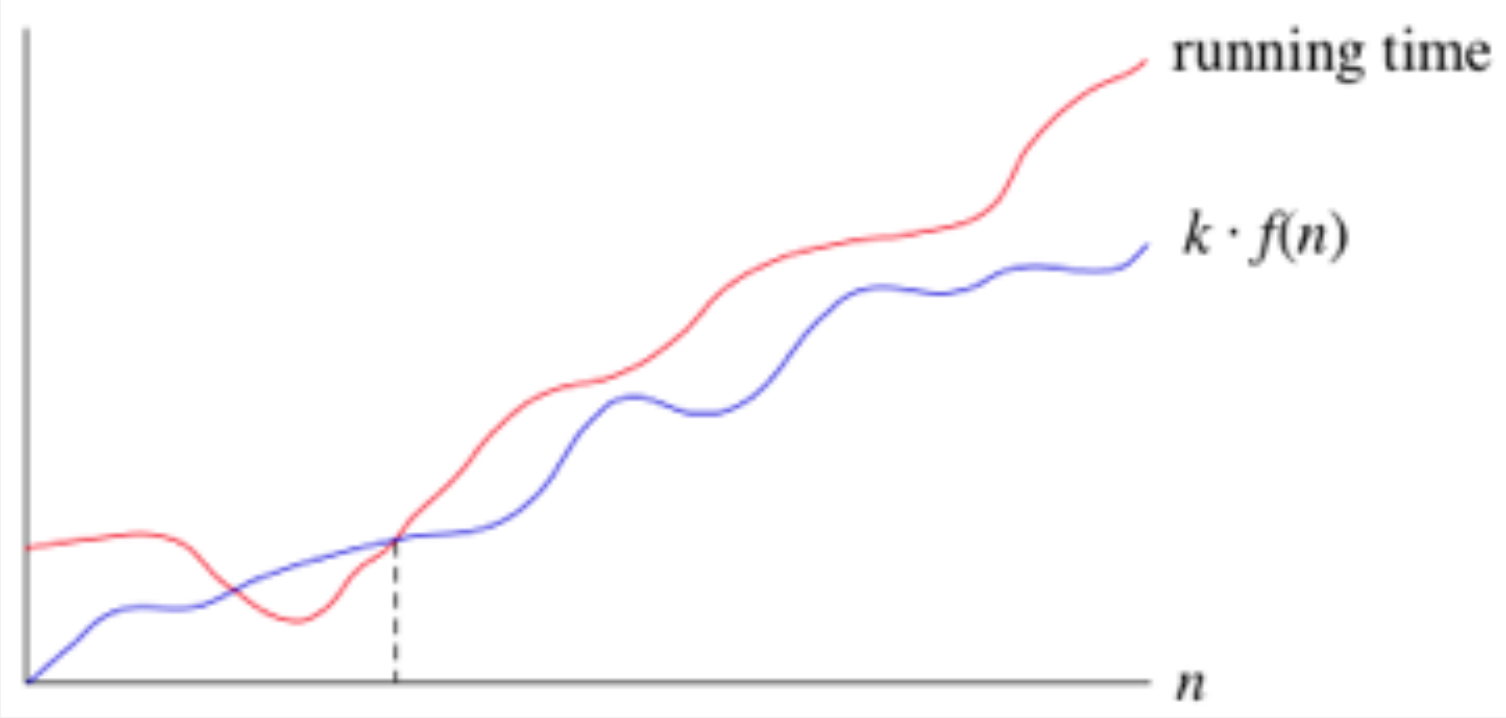
It's easier in practice to work out the O-complexity of an algorithm than its Θ-complexity.

Big-Ω (Big-Omega) notation

- Sometimes we want to be able to say that our algorithm takes at least a certain amount of time (it can't do better)
- Omega gives us the lower bound
- May be important that the time taken grows at least by this function

If running time is Ω(f(n)) then for large enough N it is at least k*f(n)

Big Ω provides asymptotic lower bounds



Commonly used asymptotic notations in classification of algorithms

notation	example	shorthand for	used to
Big Theta	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ ⋮	classify algorithms
Big Oh	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ ⋮	develop upper bounds
Big Omega	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ ⋮	develop lower bounds

Asymptotic comparison table

Algorithm is O(something)	A number is <= something
Algorithm is Θ (something)	A number is = something
Our algorithm is Ω (something)	A number is >= something

While all the symbols O, Ω, and Θ are useful at times, O is more commonly, as it's easier to determine than Θ and more practically useful than Ω.

Long Multiplication Algorithms

Long Multiplication

Long multiplication that is taught in school is $O(n^2)$

Input: two n-digit numbers x and y

Output: product of x and y

```
    23958233
×       5830
------------------
    00000000 (=     23,958,233 ×     0)
   71874699  (=     23,958,233 ×    30)
  191665864  (=     23,958,233 ×   800)
+ 119791165  (=     23,958,233 × 5,000)
------------------
 139676498390 (= 139,676,498,390       )
```

N = 3 requires 9 multiplications

N = 100 requires 10,000 mult.

Bottom line: the running times grows quadratically as the input integers grow

Multiplication algorithms is a very active area of development

Much faster multiplication algorithms than the standard algorithm have been developed

Karatsuba's algorithm (1960) has complexity:

$$O(n^{1.585})$$

Applications for cryptography and BigIntegers

For small n, long multiplication is faster but then Karatsuba faster

🤔 Java & Python use Karatsuba for integer multiplication

Complexity of Multiplication Algorithms

Multiplication	Two n-digit numbers	One $2n$-digit number	Schoolbook long multiplication	$O(n^2)$
			Karatsuba algorithm	$O(n^{1.585})$
			3-way Toom–Cook multiplication	$O(n^{1.465})$
			k-way Toom–Cook multiplication	$O(n^{\log(2k-1)/\log k})$
			Mixed-level Toom–Cook (Knuth 4.3.3-T)[2]	$O(n\, 2^{\sqrt{2\log n}} \log n)$
			Schönhage–Strassen algorithm	$O(n \log n \log\log n)$
			Fürer's algorithm[3]	$O(n \log n\, 2^{2\log^* n})$
			Harvey-Hoeven algorithm[4] [5]	$O(n \log n)$

https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations

The Perfect way to multiply

"On March 18, two researchers described the fastest method ever discovered for multiplying two very large numbers. The paper marks the culmination of a long-running search to find the most efficient procedure for performing one of the most basic operations in math."

Joris van der Hoeven, a mathematician at the French National Center for Scientific Research, helped create the fastest-ever method for multiplying very large integers.

https://www.quantamagazine.org/mathematicians-discover-the-perfect-way-to-multiply-20190411/

Russian Peasant's Algorithm

The Russian Peasant's Algorithm is O(logn)

Number doubled	Multiplications so far	Power of 2	Number halved	Division Problem	Remainder
~~13~~	13	2^0	~~12~~	12/2 = 6	0
~~26~~	13*2	2^1	~~6~~	6/2 = 3	0
52	13*2*2	2^2	3	3/2 = 1	1
104	13*2*2*2	2^3	1	1/2 = 0	1

Further reading

A Gentle Introduction to Algorithm Complexity Analysis by Dionysis "dionyziz" Zindros https://discrete.gr/complexity/

Cormen, Leiserson, Rivest, Stein. Introduction to Algorithms, MIT Press.

Mathematicians Discover the Perfect Way to Multiply
https://www.quantamagazine.org/mathematicians-discover-the-perfect-way-to-multiply-20190411/