



<http://algs4.cs.princeton.edu>

## 1.4 ANALYSIS OF ALGORITHMS

---

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

# Common order-of-growth classifications

---


**Definition.** If  $f(N) \sim c g(N)$  for some constant  $c > 0$ , then the **order of growth** of  $f(N)$  is  $g(N)$ .

- Ignores leading coefficient.
- Ignores lower-order terms.

**Ex.** The order of growth of the **running time** of this code is  $N^3$ .

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0)
                count++;
```

**Typical usage.** With running times.

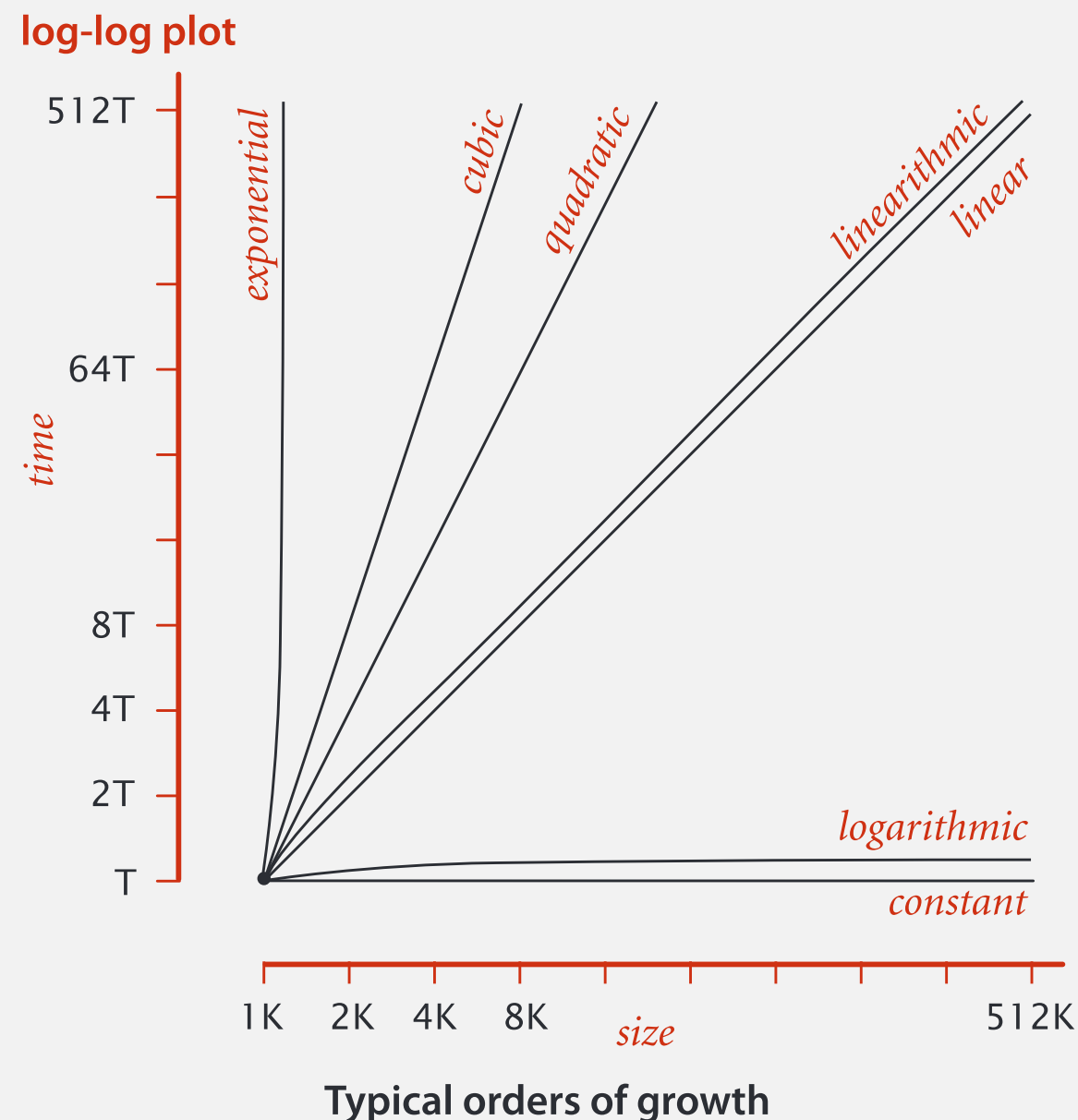
 where leading coefficient  
depends on machine, compiler, JVM, ...

# Common order-of-growth classifications

Good news. The set of functions

$1$ ,  $\log N$ ,  $N$ ,  $N \log N$ ,  $N^2$ ,  $N^3$ , and  $2^N$

suffices to describe the order of growth of most common algorithms.



# Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	<b>constant</b>	<code>a = b + c;</code>	statement	add two numbers	1
$\log N$	<b>logarithmic</b>	<pre>while (N &gt; 1) {     N = N / 2;  ... }</pre>	divide in half	binary search	$\sim 1$
$N$	<b>linear</b>	<pre>for (int i = 0; i &lt; N; i++) {     ... }</pre>	loop	find the maximum	2
$N \log N$	<b>linearithmic</b>	[see mergesort lecture]	divide and conquer	mergesort	$\sim 2$
$N^2$	<b>quadratic</b>	<pre>for (int i = 0; i &lt; N; i++)     for (int j = 0; j &lt; N; j++)     {         ...     }</pre>	double loop	check all pairs	4
$N^3$	<b>cubic</b>	<pre>for (int i = 0; i &lt; N; i++)     for (int j = 0; j &lt; N; j++)         for (int k = 0; k &lt; N; k++)         {             ...         }</pre>	triple loop	check all triples	8
$2^N$	<b>exponential</b>	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

# Binary search demo

---

**Goal.** Given a sorted array and a key, find index of the key in the array?

**Binary search.** Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.



**successful search for 33**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑
lo														hi

# Binary search: Java implementation

---

## Trivial to implement?

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

```
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;
}
```

← one "3-way compare"

**Invariant.** If key appears in the array `a[]`, then  $a[lo] \leq key \leq a[hi]$ .

# Binary search: mathematical analysis

---

**Proposition.** Binary search uses at most  $1 + \lg N$  key compares to search in a sorted array of size  $N$ .

**Def.**  $T(N)$  = # key compares to binary search a sorted subarray of size  $\leq N$ .

**Binary search recurrence.**  $T(N) \leq T(N/2) + 1$  for  $N > 1$ , with  $T(1) = 1$ .

$\uparrow$                        $\uparrow$   
left or right half      possible to implement with one  
(floored division)      2-way compare (instead of 3-way)

**Pf sketch.** [assume  $N$  is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && \text{[ given ]} \\ &\leq T(N/4) + 1 + 1 && \text{[ apply recurrence to first term ]} \\ &\leq T(N/8) + 1 + 1 + 1 && \text{[ apply recurrence to first term ]} \\ &\vdots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 && \text{[ stop applying, } T(1) = 1 \text{ ]} \\ &= 1 + \lg N \end{aligned}$$

# An $N^2 \log N$ algorithm for 3-SUM

## Algorithm.

- Step 1: Sort the  $N$  (distinct) numbers.
- Step 2: For each pair of numbers  $a[i]$  and  $a[j]$ , binary search for  $-(a[i] + a[j])$ .

**Analysis.** Order of growth is  $N^2 \log N$ .

- Step 1:  $N^2$  with insertion sort.
- Step 2:  $N^2 \log N$  with binary search.

**Remark.** Can achieve  $N^2$  by modifying binary search step.

**input**

30 -40 -20 -10 40 0 10 5

**sort**

-40 -20 -10 0 5 10 30 40

**binary search**

(-40, -20)	60
(-40, -10)	50
(-40, 0)	40
(-40, 5)	35
(-40, 10)	30
⋮	⋮
(-20, -10)	30
⋮	⋮
(-10, 0)	10
⋮	⋮
( 10, 30)	<del>-40</del>
( 10, 40)	-50
( 30, 40)	-70

only count if  
 $a[i] < a[j] < a[k]$   
to avoid  
double counting



# Comparing programs

---

**Hypothesis.** The sorting-based  $N^2 \log N$  algorithm for 3-SUM is significantly faster in practice than the brute-force  $N^3$  algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

**Guiding principle.** Typically, better order of growth  $\Rightarrow$  faster in practice.