

Course «Algorithms» · Lecture 3

The Analysis Framework: Time, Space and Order of Growth

Textbook: A. Levitin, Section 2.1

Today’s goals

- Name the two kinds of **efficiency**
- Choose the **input size** for a given problem
- Find the **basic operation** and estimate the running time
- Order the standard **growth functions**
- Explain the **worst, best and average case**

Last week: what an algorithm is, and Euclid’s algorithm. A program must be correct, but also efficient. How do we compare algorithms independently of the computer?

Two kinds of efficiency

- **Time efficiency:** how fast the algorithm runs
- **Space efficiency:** how much extra memory it needs, in addition to the input

Today memory is cheap, so in this course we mostly talk about time. Space still matters on phones and embedded devices, and for very big data.

Both are analyzed in the same way.

Measuring the input size

Efficiency is a function of the input size n. How to choose n depends on the problem:

Problem	Input size
Searching, sorting	n = number of elements
Polynomials	n = degree of the polynomial
n × n matrices	n = order of the matrix (or n^2 elements; say which)
Graphs	two numbers: vertices |V| and edges |E|
Numbers (is m prime?)	number of bits b, not m itself

$b = \lfloor \log_2 m \rfloor + 1$ 1,000,000 has only 20 bits.

Units: why not seconds?

Seconds depend on the computer, the compiler and the code. Instead, count the **basic operation**: the one that contributes the most to the running time, usually in the innermost loop.

Algorithm	**Basic operation**
Sequential search	comparing the key with an array element
Sorting by comparisons	comparing two keys, A[i] > A[j]
Matrix multiplication	multiplying two numbers
Euclid’s algorithm	the division m mod n

Estimating the running time

$$T(n) \approx c_{\text{op}}\, C(n)$$

- **c_op**: time of one basic operation; depends on the computer
- **C(n)**: how many times it runs; depends only on the algorithm

We can’t control the constant, so C(n) is what we analyze.

What if we double the input?

Suppose the count of basic operations is:

$$C(n) = \tfrac{1}{2}\, n(n-1) \approx \tfrac{1}{2}\, n^2$$

$$\frac{T(2n)}{T(n)} \approx \frac{c_{\text{op}} \cdot \tfrac{1}{2}(2n)^2}{c_{\text{op}} \cdot \tfrac{1}{2} n^2} = 4$$

The algorithm runs **4 times longer**, and we didn't need to know the computer's speed.

Both constants cancel. That's why we ignore constant factors and look at how C(n) grows.

Order of growth

For small inputs all algorithms are fast. What matters is how C(n) behaves for large n.

n	$C_1(n) = 100n$	$C_2(n) = n^2$	Which is faster?
10	1,000	100	the second
100	10,000	10,000	equal
1,000	100,000	1,000,000	the first, 10 times
1,000,000	10^8	10^{12}	the first, 10,000 times

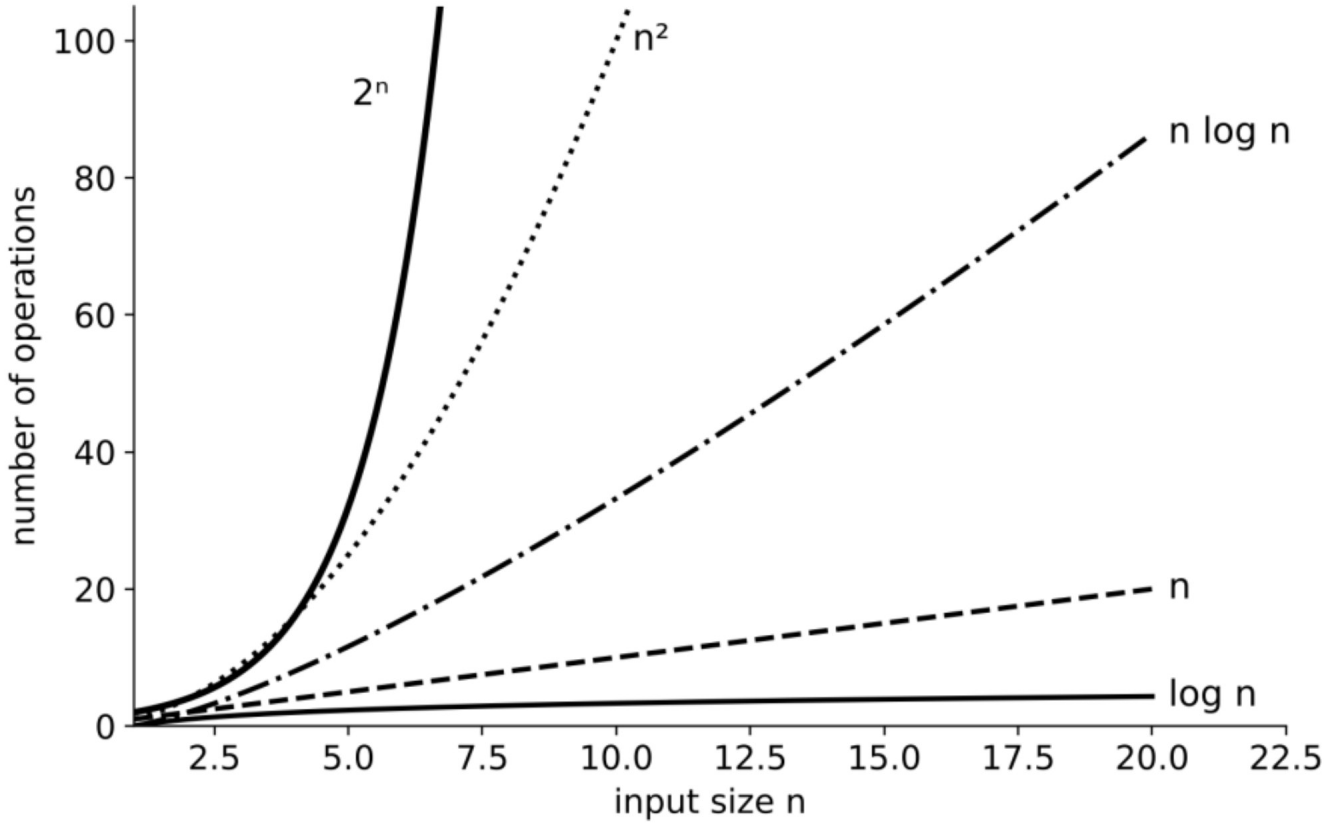
The constant 100 looked big, but the form of the function decides everything.

The common functions

$$1 < \log_2 n < n < n\log_2 n < n^2 < n^3 < 2^n < n!$$

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	33	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	660	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	$1.0 \cdot 10^4$	10^6	10^9	too big	too big
10^6	20	$2.0 \cdot 10^7$	10^{12}	10^{18}	too big	too big

How the common functions grow



What the numbers tell us

- **Logarithm:** very slow growth. For a million elements, $\log_2 n$ is only 20
- **2^n and $n!$:** terribly fast. For $n = 100$ and 10^{12} operations per second, 2^n takes about $4 \cdot 10^{10}$ years

When n doubles	$\log_2 n$	n	n^2	n^3	2^n
The count…	grows by 1	doubles	× 4	× 8	gets squared

Exponential algorithms are usable only for very small inputs.

Example: sequential search

```
SequentialSearch(A[0..n−1], K):
  i ← 0
  while i < n and A[i] ≠ K do
    i ← i + 1
  if i < n return i
  else return −1
```

Basic operation: the key comparison. Their number depends on **where K is**, not only on n, so we consider three cases.

Worst, best and average case

Case	Meaning	Sequential search
Worst	largest count over all inputs of size n: a guarantee	K is last or absent: C = n
Best	smallest count; not very useful on its own	K is first: C = 1
Average	count on a “typical” random input; needs an assumption	next slide

“Efficiency” without more details means the worst case.

Average case of sequential search

K is in the array with probability p, equally likely in any of the n positions. At position i: i comparisons. Not there: n comparisons.

$$C_{\text{avg}}(n) = (1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \ldots + n \cdot \frac{p}{n}) + n(1-p) = \frac{p(n+1)}{2} + n(1-p)$$

$$p = 1: \quad C_{\text{avg}}(n) = \frac{n+1}{2}$$

$$p = 0: \quad C_{\text{avg}}(n) = n$$

The average case is not the average of the best and the worst case.

Common mistakes

- Measuring efficiency in **seconds**: count basic operations instead
- Taking the **value of a number** as its size: the size is the number of bits
- Choosing a basic operation **outside the innermost loop**
- Thinking the average case is **(best + worst) / 2**: it needs an assumption about the inputs

Summary

- Count basic operations C(n) as a function of n, not seconds
- Time: how fast. Space: how much extra memory
- For large n only the order of growth matters
- $\log_2 n < n < n \log_2 n < n^2 < n^3 < 2^n < n!$
- Running time can depend on the input: worst, best, average
- The worst case gives a guarantee

Review questions

1. Why is the running time in seconds a bad measure of the efficiency of an algorithm?
2. What is the basic operation of sequential search? How many times is it executed in the best and in the worst case?
3. Put these functions in order of growth, from slowest to fastest: n^2, $\log_2 n$, 2^n, n, $n!$.

Reading: Levitin, Section 2.1; CLRS, Section 2.2.