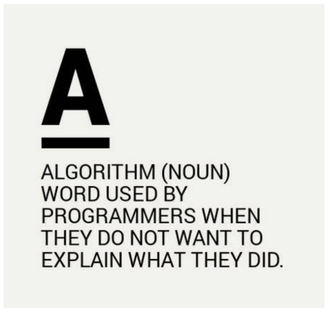
Intro to algorithms

CS Osvita

January 6, 2025



A

ALGORITHM (NOUN)
WORD USED BY
PROGRAMMERS WHEN
THEY DO NOT WANT TO
EXPLAIN WHAT THEY DID.

Why study algorithms?

1. **Algorithms are fundamental to all areas of CS** For example, operating systems and compilers leverage scheduling algorithms and efficient data structures, networking crucially uses shortest-path algorithms, machine learning leverages fast geometric algorithms and similarity search, cryptography leverages fast number theoretic and algebraic algorithms, and computational biology leverages algorithms that operate on strings and often employ the dynamic programming paradigm. We'll discuss applications to all these subjects in this class.

2. **Algorithms are practical** Much of the progress in the tech industry is due to the dual developments of improved hardware and algorithms. The industry needs to continue developing new algorithms for tomorrow's problems, and you can help contribute.

3. **Algorithms are** fun Back when I was a student, my favorite classes were always the challenging ones that, after I struggled through them, left me feeling a few IQ points smarter than when I started. I hope this material provides a similar experience for you. The design of algorithms requires a combination of creativity and mathematical precision. It is both an art and a science; hopefully, some of you will love this combination. When

you become good with Algo you also become good at a bunch of other super valuable skills which will be useful going forward. The side skills include – systems thinking, fast learning, problem statement deduction, and algorithmic thinking. It's not about the algorithms you learn, it's about the effect the whole learning and growth process will have on your brain/thinking.

Algorithm analysis

What makes one computer program better than another? If you were given two programs that solve the same problem, how would you decide between them? There are many valid criteria, which are often in conflict.

We typically want our program to be correct. In other words, we'd like the program's output to match our expectations. Unfortunately, correctness is not always clear. For instance, what does it mean for Google to return the "correct" top 10 search results for your search query?

Good software engineers often want their code to be *readable, reusable, elegant, or testable*. These are admirable goals, but you may not be able to achieve them all simultaneously. It's also not entirely clear what something like "elegance" looks like, and we certainly haven't been able to model it mathematically, so computer scientists haven't given these aspects of programs much consideration.

Two factors that computer scientists love to model mathematically, though, are how long a program will take to run and how much memory it will use. We call these *time and space efficiency*, and they'll be at the core of our study of algorithms. But please don't go away thinking they're always the most important factors. The only truly correct answer is: "It depends".

Another aspect of "it depends", even when we focus on just time or space, is the program's context. For instance, if you "grep" over many large files, it will take longer than if you "grep" over fewer, smaller files. This relationship between inputs and behavior will be an important part of our analysis. Beyond this, the exact time and space that your program uses will also depend on many other factors, some of which are:

- How long does it take your computer to execute every instruction?
- How many cores of your machine does the program use?
- What language is your program written in?
- How does your operating system choose to schedule processes?
- What other programs are running at the same time?

... and there are many more

All of these are important in practice, but none address whether an algorithm is generally better or worse than another. Sometimes, we'd like to be able to ask: generally speaking, irrespective of whether a program is written in Fortran for the IBM 704 or in Python running on a shiny new Macbook, will it be more time and/or space-efficient than an alternative? Will it use less space?

This is the crux of algorithm analysis: *Algorithm analysis is a way to compare the time and space efficiency of programs concerning their possible inputs, irrespective of other contexts.*

Time complexity

In the real world, we measure the time a program uses in some unit of time, such as seconds. Similarly, we measure space used in something like bytes. In analysis, this would be too specific. If we measure the time it takes to finish, this number will incorporate details like language choice and CPU speed. We will need new models and vocabulary to speak with the level of generality that we're seeking.

Let's explore this idea with an example. Say I wanted to calculate the sum of the first n numbers, and I'm wondering how long this will take.

```python
def sum(n: int) -> int:
    total = 0
    for i in range(n + 1):
        total += i
    return total
```

Let's now add some profiling code:

```python
import time
def profiler(n: int) -> tuple:
    start = time.time()
    total = sum(n)
    end = time.time()
    return total, end - start
```

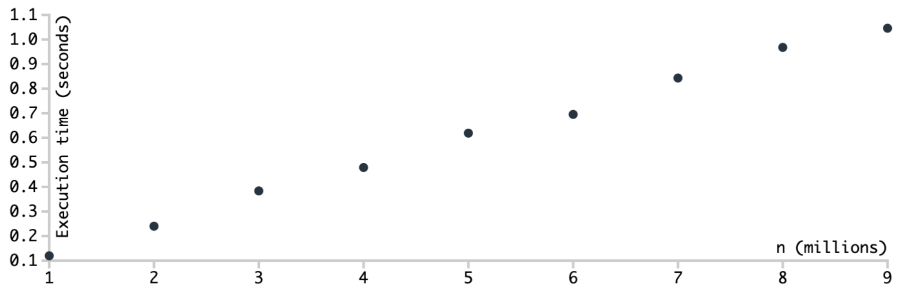
I ran this with $n = 1.000.000$ and noticed it took 0.11 seconds. What would you expect to see if I ran it 3 more times?

1. profiler(1000000) = 500000500000 (0.1209280 seconds)
2. profiler(1000000) = 500000500000 (0.1107872 seconds)
3. profiler(1000000) = 500000500000 (0.1187370 seconds)

Interestingly, each invocation takes a slightly different amount of time due to the slightly different state of my computer and the Python virtual machine each time. We'd generally like to ignore such small and random differences. What if we were to rerun it with a range of different inputs, say 1 million, 2 million, and so on up to 5 million? What would you expect to see?

1. profiler(1000000) = 500000500000 (0.1198549 seconds)
2. profiler(2000000) = 2000001000000 (0.2401729 seconds)
3. profiler(3000000) = 4500001500000 (0.3838110 seconds)
4. profiler(4000000) = 8000002000000 (0.4790699 seconds)
5. profiler(5000000) = 12500002500000 (0.6189690 seconds)

Do you see the general relationship between n and time elapsed?

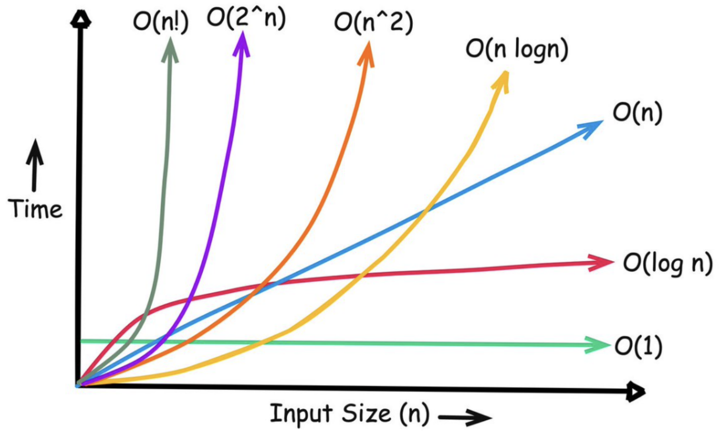


We describe $sum(n)$ as a "linear" algorithm. You can start to see why. Irrespective of the exact times that these functions take to execute, we can spot a general trend that the execution time for $sum(n)$ grows in proportion to n.

An algorithm's execution time can be expressed as the **number of steps** required to solve the problem. This abstraction is exactly what we need: it characterizes an algorithm's efficiency in execution time while remaining independent of any particular program or computer.

Let's see the correspondence between the number of steps and execution time. In particular, the picture below shows how long algorithms that use n operations take to run on a fast computer, where each operation costs one nanosecond.

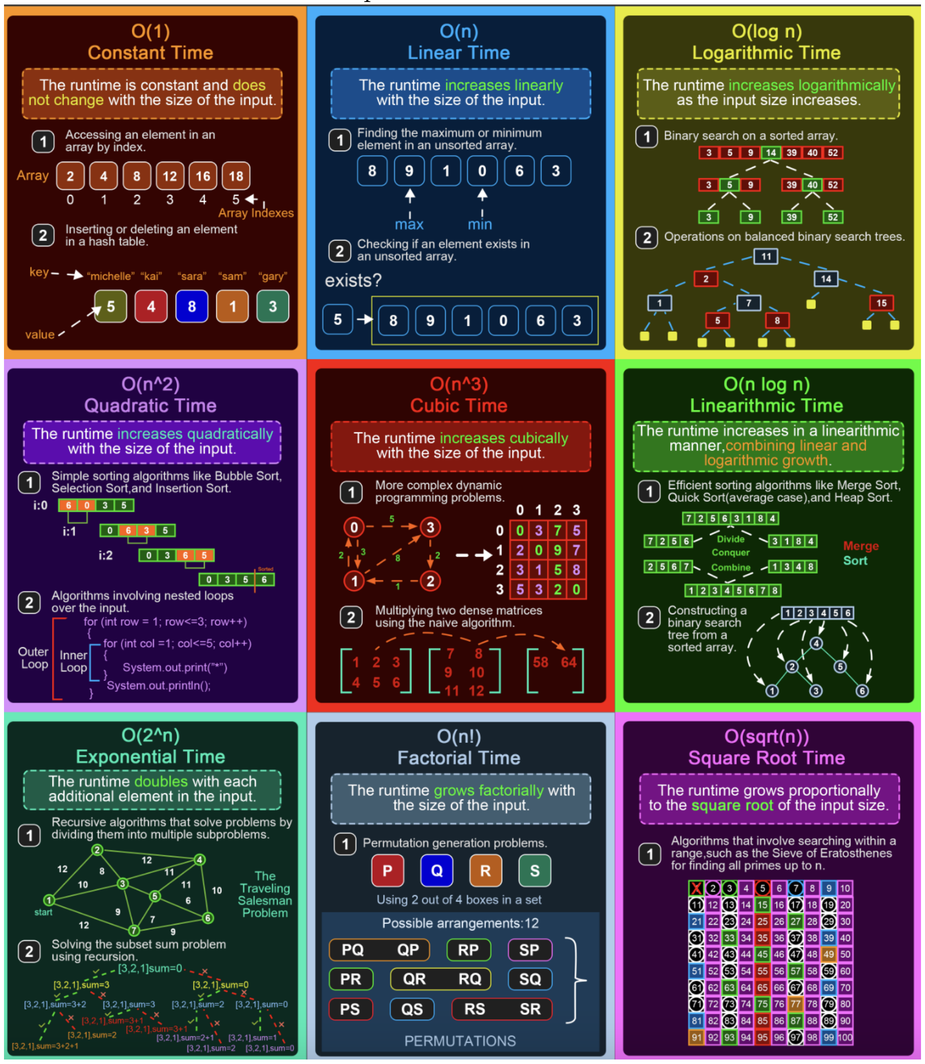
n	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		



The following conclusions can be drawn from this table:

1. All such algorithms take roughly the same time for $n = 10$.
2. Any algorithm with n! running time becomes useless for $n >= 20$.
3. Algorithms whose running time is 2^n have a greater operating range, but become impractical for $n > 40$.
4. Quadratic-time algorithms, whose running time is n^2, remain usable up to about $n = 10.000$, but quickly deteriorate with larger inputs. They will likely be hopeless for $n > 1.000.000$.
5. Linear-time and $nlgn$ algorithms remain practical on inputs of one billion items.
6. An $logn$ algorithm hardly sweats for any imaginable value of n.

And here is one more visual explanation for better intuition



O(1)
Constant Time
The runtime is constant and does not change with the size of the input.
1 Accessing an element in an array by index.
Array
Array Indexes
2 Inserting or deleting an element in a hash table.
key
"michelle" "kai" "sara" "sam" "gary"
value
O(n)
Linear Time
The runtime increases linearly with the size of the input.
1 Finding the maximum or minimum element in an unsorted array.
max
min
2 Checking if an element exists in an unsorted array.
exists?
O(log n)
Logarithmic Time
The runtime increases logarithmically as the input size increases.
1 Binary search on a sorted array.
2 Operations on balanced binary search trees.
O(n^2)
Quadratic Time
The runtime increases quadratically with the size of the input.
1 Simple sorting algorithms like Bubble Sort, Selection Sort,and Insertion Sort.
i:0
i:1
i:2
2 Algorithms involving nested loops over the input.
for (int row = 1; row<=3; row++)
for (int col =1; col<=5; col++)
System.out.print("*")
System.out.println();
Outer Loop
Inner Loop
O(n^3)
Cubic Time
The runtime increases cubically with the size of the input.
1 More complex dynamic programming problems.
2 Multiplying two dense matrices using the naive algorithm.
O(n log n)
Linearithmic Time
The runtime increases in a linearithmic manner,combining linear and logarithmic growth.
1 Efficient sorting algorithms like Merge Sort, Quick Sort(average case),and Heap Sort.
Divide
Conquer
Combine
Merge Sort
2 Constructing a binary search tree from a sorted array.
O(2^n)
Exponential Time
The runtime doubles with each additional element in the input.
1 Recursive algorithms that solve problems by dividing them into multiple subproblems.
start
The Traveling Salesman Problem
2 Solving the subset sum problem using recursion.
[3,2,1]sum=0
O(n!)
Factorial Time
The runtime grows factorially with the size of the input.
1 Permutation generation problems.
P Q R S
Using 2 out of 4 boxes in a set
Possible arrangements:12
PQ QP RP SP
PR QR RQ SQ
PS QS RS SR
PERMUTATIONS
O(sqrt(n))
Square Root Time
The runtime grows proportionally to the square root of the input size.
1 Algorithms that involve searching within a range,such as the Sieve of Eratosthenes for finding all primes up to n.