sourcemind

Data Structures and Algorithms

Lionel D. KITIHOUN

Objectives

Analyze algorithms

- How well does an algorithm perform? Speed, correctness, intuition.
- Compare algorithms and choose the most suitable.

Develop algorithm design skills (hopefully)

Learn common data structures and algorithms

Course format

10 sessions of 2 hours

- One topic per session, with examples and sample codes.
- 1 homework per session.
- 2 Quizzes of 2 hours, after sessions 4 and 10.

sourcemind

Topics

- Complexity analysis
- Recursion
- Sorting algorithms
- Binary search
- Basic data structures: linked list, stack, queue
- Maps, hash tables, and Bloom Filters
- Introduction to graphs
- Binary search trees
- Graphs
- Common graphs algorithms

sourcemind

Session 1
Complexity Analysis

sourcemind

A bit of history

When chess was presented to a great king, the king offered the inventor any reward that he wanted. The inventor asked that a single grain of wheat be placed on the first square of the chessboard. Then two grains on the second square, four grains on the third, and so on. Doubling each time.

Source: Wikimedia Commons

A bit of history

The king, baffled by such a small price for a wonderful game, immediately agreed, and ordered the treasurer to pay the agreed upon sum. A week later, the inventor went before the king and asked why he had not received his reward. The king, outraged that the treasurer had disobeyed him, immediately summoned him and demanded to know why the inventor had not been paid.

A bit of history

The treasurer explained that the sum could not be paid - by the time you got even halfway through the chessboard, the amount of grain required was more than the entire kingdom possessed.

sourcemind

A bit of story

The king took in this information and thought for a while. Then he did the only rational thing a king could do in those circumstances. He had the inventor killed, as an object lesson in the perils of trying to outwit the king.

Source: forbes.com

Explanation of the problem

The problem may be solved using simple addition. With 64 squares on a chessboard, if the number of grains doubles on successive squares, then the sum of grains on all 64 squares is: 1 + 2 + 4 + 8 + ... and so forth for the 64 squares. The total number of grains can be shown to be $2^{64}-1$ or 18,446,744,073,709,551,615.

Explanation of the problem

That is over 2,000 times the annual world production of wheat.

Source: Wikipedia

It is about growth

Our goal is to understand how the computation required by an algorithm increases relative to its input and select the best algorithm when have several choices.

Introductory problem

Given an integer N, compute the sum of the first N integers.

$$S = 1 + 2 + 3 + \ldots + N$$

sourcemind

Simple solution

Nothing complex.

We just need to go through the numbers with a counter and add them up.

Simple solution pseudo-code

```
sum = 0
for i = 1 to N:
   sum = sum + i
print(sum)
```

Performance of our solution

Computers can roughly perform one hundred million operations per second. Our program performs very well when N is not too large.

But what happens when we have big values for N? Let’s 1 billion or 2 billions.

Let’s run some tests.

sourcemind

Benchmark results

Value of N	Time
100	0.021s
10 000	0.024s
1 000 000	0.117s
100 000 000	0m9.952s
1 000 000 000	1m29.860s

Analysis

When N reaches a certain threshold, the running time of our algorithm grows exponentially.

This is not acceptable.

We need programs that do their job efficiently.

Gauss summation formula

There is a nice formula to compute the sum of the first N integers.

$$S = 1 + 2 + 3 + \ldots + N = N * (N + 1) / 2$$

Gauss formula algorithm

```
sum = n * (n + 1) / 2

print(sum)
```

Benchmark results for summation formula

Value of N	Time
100	0.017s
10 000	0.018s
1 000 000	0.017s
100 000 000	0.018s
1 000 000 000	0.018s

This example helps us introduce why it is important to analyze an algorithm behaviour and verify how it performs when its input grows.

Asymptotic analysis: first insight

What we need now is a convention to inform other people about how an algorithm performs to let them know what they're getting into so that they can decide if our solution fits their requirements.

It is very similar to a computer technical specs. You decide if a computer matches your requirements from its description.

Big O notation

It is a measure of the time complexity of an algorithm.

It helps us know how the time used by an algorithm grows relative to its input.

We will introduce it gradually through the course.

Determining algorithm complexity

No magic. It is very simple.

- Count the number of operations.
- Derive a formula.
- Choose a well-known function that is its upper bound.

First illustration

How many operations does this algorithm use?

```
sum = 0
for i = 1 to N:
  sum = sum + i
print(sum)
```

```
sum = 0               /* 1 */
for i = 1 to N:       /* 1 */
  sum = sum + i       /* 2: addition + assignment */
end
print(sum)            /* We can ignore this */
```

Result

It takes **3 * N + 1** operations. That’s the cost of our algorithm.

- But mathematicians are lazy.
- They will say the overall complexity is **O(N)**.
- The dominant element in the formula is N.

sourcemind

Gauss formula complexity

```
sum = n * (n + 1) / 2
print(sum)
```

sourcemind

Gauss formula complexity

- Whatever the value of N, we will always perform the same number of operations to find the answer.
- 3 operations.
- Since this number is constant, the complexity is **O(1)**.

Second example: exponentiation

Given two integers B and E, compute B^E

First attempt (naive)

```
pow = 1
for i = 1 to N:
    pow = pow * b
print(pow)
```

Analysis

How many operations does it use?

We can identity **1 + 3 * N** operations.

The overall complexity is O(N).

sourcemind

Exponentiation by squaring

One nice thing we learn from binary is that each positive integer can be written as the sum of powers of 2.

5	=	1	+	4					
12	=	4	+	8					
23	=	1	+	2	+	4	+	16	
100	=	4	+	32	+	64			

sourcemind

Exponentiation by squaring

We can use this property to speed up our exponentiation process.

$$x^5 = x^4 * x^1$$
$$x^{12} = x^8 * x^4$$
$$x^{23} = x^{16} * x^4 * x^2 * x^1$$
$$x^{100} = x^{64} * x^{32} * x^4$$

sourcemind

Fast exponentiation pseudocode

```
function pow(B, E)
    P = 1
    while E != 0
         if E % 2 == 1
              P = P * B
         end
         B = B * B
         E = E / 2
    end
    return P
end
```

sourcemind

Fast exponentiation cost

- 1 initial affectation.
- The while loop contains 9 operations.
- The total cost is then 1 + 9 * K.
- K is the number of times the while loop executes.

Question

What is the value of K?

Answer

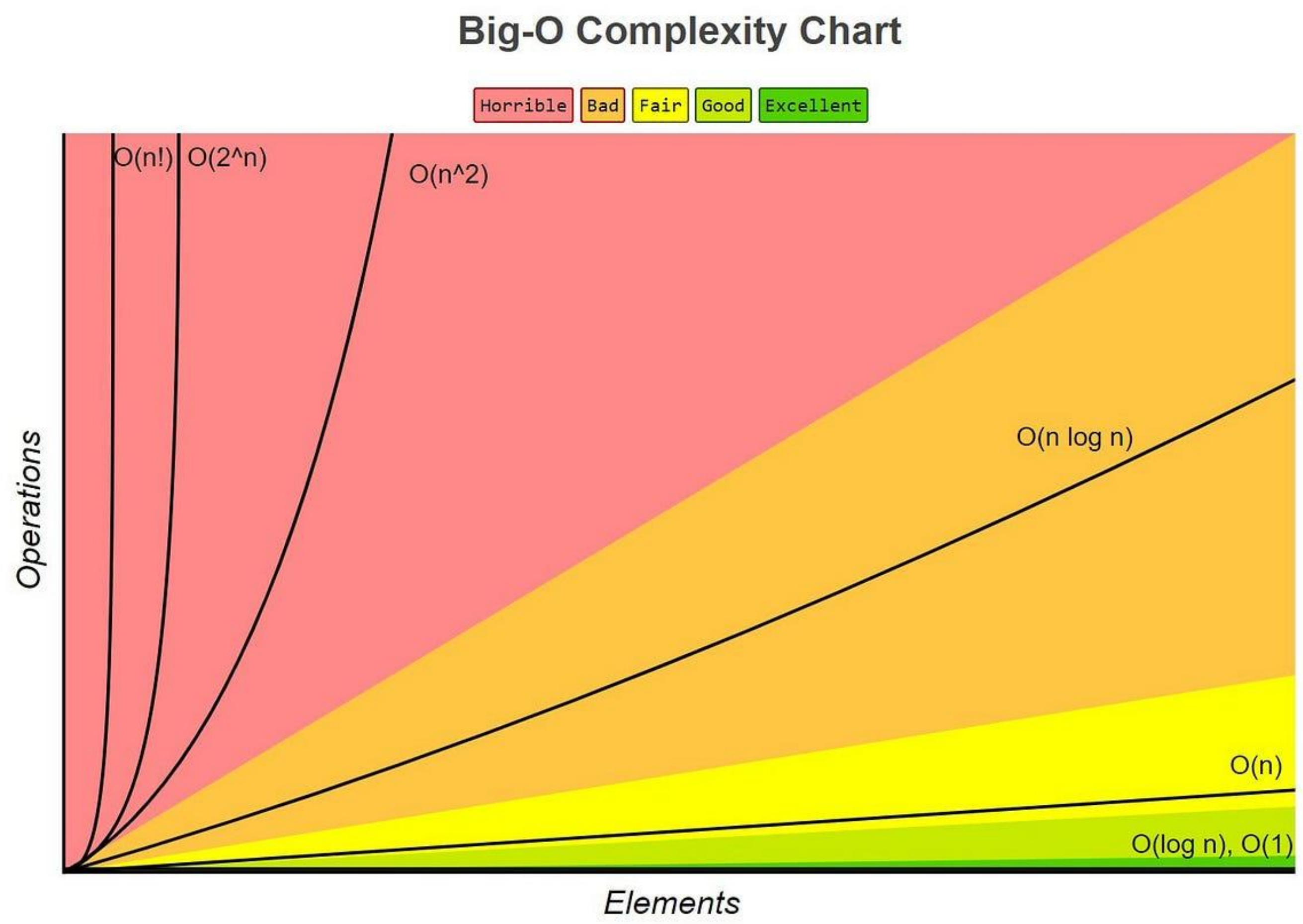
$$K = \ln_2 E + 1$$

sourcemind

- The while loop executes $\ln_2 E + 1$ times
- Cost is $1 + 9 * (\ln_2 E + 1)$
- In Big O notation, the complexity is **$O(\ln E)$**

Comparison of the two exp. algorithms

- The naive algorithm complexity is O(E).
- The fast algorithm complexity is O(ln E).
- Which one would you choose?



Big-O Complexity Chart: http://bigocheatsheet.com

sourcemind