Algorithms and complexity

Definitions, big O notation

beOI Training

October 21, 2022

Table of contents

What's an algorithm?

- A way to calculate a result
- An idea to solve a problem
- A series of instructions
- The description of a programme

What's a *good* algorithm?

For a normal programmer:

- Doesn't crash
- Halts
- Gives the right answer

For us:

- Is fast
- Uses little memory
- **Is accepted in competitions**

Table of contents

Measuring the efficiency

Ideas:

- Timing
- Measure the RAM usage

But this varies depending on:

- The language
- The implementation
- The machine
- The time of the day

Big O notation

Finding an *intrinsic* measure of efficiency:

- Grow the input
- Observe how the execution speed changes

Example: calculating $1 + \cdots + n$. If n is multiplied by 2, the execution time:

- Stays constant: $O(1)$
- Is multiplied by 2: $O(n)$
- Is multiplied by 4: $O(n^2)$

It doesn't depend on the constant factors!

Constant execution time

Task: compute the sum $1 + 2 + \cdots + n$.

Solution 1: A simple calculation

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

- Execution time doesn't change if n doubles
- "Constant" execution time
- $O(1)$ complexity
- Execution time "proportional to 1"

Linear execution time

Solution 2: A loop

```
int sum = 0;
for (int i = 1; i <= n; i++)
    sum += i;
```

- Execution time doubles if n doubles
- "Linear" execution time
- $O(n)$ complexity
- Execution time "proportional to n"

Quadratic execution time

Solution 3: Two nested loops (useless!)

```
int sum = 0;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= i; j++)
        sum++;
```

- Execution time quadruples if n doubles
- "Quadratic" execution time
- $O(n^2)$ complexity
- Execution time "proportional to n^2"

Powers

Definition:

- Successive multiplications
- "3 to the power of n"
- $3^n = \underbrace{3 \times \cdots \times 3}_{n \text{ times}}$

Examples:

- $3^0 = 1$ (by definition)
- $3^1 = 3$
- $3^2 = 3 \times 3 = 9$ (squared)
- $3^3 = 3 \times 3 \times 3 = 27$ (cubed)

Logarithms: intuition (1)

Game with two players:

- Alice chooses a number between 1 and 16
- During each turn:
  - Bob gives one or more numbers to Alice
  - Alice says if her chosen number is among those given by Bob
- Bob wins when he guesses the correct number
- How can he win in as few turns as possible?

Logarithms: intuition (2)

Strategy: Give a list with half the total available numbers

- First give 8 numbers among the 16
- Then 4 among the 8 remaining
- Then 2 among the 4 remaining
- Then 1 among the 2 remaining
- Found!

Thus, 4 questions suffice.

Logarithms: intuition (3)

Generally, if we start with n numbers, how may questions needed?

How many times can we cut in half?

- If $n = 2$, once
- If $n = 4$, twice
- If $n = 8$, three times
- If $n = 16$, four times

Logarithm in base 2

The function that answers that question is $\log_2$: the logarithm in base 2. For example

- $\log_2(2) = 1$
- $\log_2(4) = 2$
- $\log_2(8) = 3$
- $\log_2(16) = 4$

Bob's strategy is $O(\log_2(n)) = O(\log n)$.

In other words, the logarithm is the exponent we have to put on the 2 to reach n:

$$x = \log_2(n) \;\Leftrightarrow\; 2^x = n$$

General logarithm (bonus)

This can also apply for numbers other than 2! The logarithm in base a, is the number of times we can divide by a. For example:

- $\log_3(27) = 3$
- $\log_4(16) = 2$
- $\log_5(5) = 1$

In other words, it's the exponent we have to put on the a to reach n:

$$x = \log_a(n) \iff a^x = n$$

It's sort of the "inverse" of powers.

Searching in a sorted array (1)

We are given a sorted array (numbers in increasing order):

1	4	6	9	15	23	24

Our task is to check if x is in the array.

Solution 1: Go through all the elements: linear, $O(n)$

```
bool isIn(int tab[], int n, int x)
{
    for (int i = 0; i < n; i++)
        if (tab[i] == x)
            return true;
    return false;
}
```

Searching in a sorted array (2)

We are searching for 7.

Idea: check the middle element and compare:

1	4	6	**9**	15	23	24

Too big ($9 > 7$), go to the left:

1	**4**	6	9	15	23	24

To small ($4 < 7$), go to the right:

1	4	**6**	9	15	23	24

Only one element left, but $6 \neq 7$: hence, 7 is not in the array

Searching in a sorted array (3)

Each time, we cut in half $\Rightarrow \log_2(n)$ tries.

Solution 2: Dichotomous search: logarithmic, $O(\log n)$

```
bool isIn(int tab[], int n, int x)
{
    int left = 0, right = n-1;
    while (left <= right)
    {
        int mid = (left+right) / 2;
        if (x < tab[mid]) right = mid - 1;
        else if (x > tab[mid]) left = mid + 1;
        else return true;
    }
    return false;
}
```

Way faster!

Limits in practice

Limits on n so that the program can execute in a few seconds:

Complexity	Limit on n	Example
$O(1)$, $O(\log n)$	$\leq 10^{18}$	(Maximal integer size)
$O(n)$	$\leq 100\,\text{M}$	Going through an array
$O(n \log n)$	$\leq 1\,\text{M}$	Sorting an array
$O(n^2)$	$\leq 10\,\text{k}$	Loop nested in a loop

For contests: find the limit on n in the second column, and get the complexity in the first column.