Time and Space complexity

Sometimes, there are more than one way to solve a problem. We need to learn how to compare the performance of different algorithms and how to choose the best one to solve a particular problem. While analyzing an algorithm, we mostly consider time complexity and space complexity.

- ***Time complexity*** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- ***Space complexity*** of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

E-OLYMP **7829. Sum of array elements** You are given array, and you have to find the sum of its elements.

Simple solution to this problem is to traverse the whole array and sum up all its elements:

```
s = 0;
for(i = 0; i < n; i++)
  s = s + m[i];
```

If the size of array is n, we need to make n operations. The total time depends on the length of array: if $n = 10$, we should make 10 additions, if $n = 10^6$, we should make 10^6 additions.

Order of growth is how the *time of execution* depends on the length of the *input*. In the above example, we can clearly see that the time of execution is linearly depends on the length of the array. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

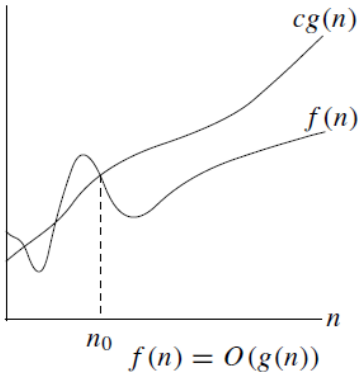
Big O-notation

Definition. Let f(n) and g(n) be two functions of n (n is usually the input size in algorithm analysis). We say that

$$f(n) = O(g(n))$$

if $\exists\ n_0 \in N$ and constant $c > 0$ such that $|f(n)| \leq c|g(n)|\ \forall n \geq n_0$.

O-notation gives an ***upper bound*** for a function to within a constant factor.



O-notation gives us an **upper limit** of the execution time i.e. the execution time in the worst case.

Let the size of linear array is *n*. The complexity of finding the sum of all elements is O(*n*).

The complexity of solution for each of the next problem is O(*n*):

E-OLYMP **904. Increase by 2** You are given an array, increase each its non-negative element by 2.

E-OLYMP **7831. Sum without maximal** Find the sum of all array elements, not equal to maximal.

E-OLYMP **928. The sum of the largest and the smallest** Find the sum of the smallest and the largest element in array.

Example. Find the complexity of the next code:

```
count = 0;
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
  count++;
```

Let's see how many times *count*++ will run. It will run n^2 times, so the time complexity is O(n^2).

Example. Find the complexity of the next code:

```
count = 0;
for(i = 0; i < n; i++)
for(j = 0; j < i; j++)
  count++;
```

When *i* = 0, *j* loop will run 0 times.
When *i* = 1, *j* loop will run 1 times.

When $i = 2$, j loop will run 2 times.
. . .
When $i = n - 1$, j loop will run $n - 1$ times.
The total number of times *count*++ will run is

$$0 + 1 + 2 + \ldots + (n - 1) = n * (n - 1) / 2$$

So the time complexity is $O(n^2)$.

Below we'll list the main classes used in the analysis of algorithms:

- $f(n) = O(1)$ constant
- $f(n) = O(\log(n))$ logarithmic growth
- $f(n) = O(n)$ linear growth
- $f(n) = O(n*\log(n))$ quasilinear growth
- $f(n) = O(n^m) = n^{O(1)}$ polynomial growth
- $f(n) = O(2^n)$ exponential growth

For exampe, the grows $O(n^2)$ is called quadratic, the grows $O(n^3)$ is called cubic.

The complexity of solution for each of the next problem is O(1):

E-OLYMP **519. Sum of squares** Find the sum of the squares of two numbers.

E-OLYMP **108. Median number** Three different numbers a, b, c are given. Print the median number.

Example. Find the complexity of the next code:

```
count = 0;
for(i = n; i > 0; i /= 2)
for(j = 0; j < i; j++)
  count++;
```

When $i = n$, j loop will run n times.
When $i = n / 2$, j loop will run $n / 2$ times.
When $i = n / 4$, j loop will run $n / 4$ times.
. . .
When $i = 1$, j loop will run 1 times.
The total number of times *count*++ will run is

$$n + n / 2 + n / 4 + \ldots + 1 = 2*n$$

The time complexity is $O(n)$.

E-OLYMP **2860. Sum of integers on the interval** Find the sum of all integers from a to b. Integers are no more than $2 * 10^9$ by absolute value.

► Let's solve the problem with ***for*** loop:

```
res = 0;
for(i = a; i <= b; i++)
  res = res + i;
```

Number of iterations is proportional to amount of numbers on the interval [*a*..*b*]. Let $n = b - a + 1$ be the size of the interval. To run a program, we must make n iterations in the ***for*** loop. For example, if $n = 2 * 10^9$, we must make $2 * 10^9$ iterations. Number of operations increase linearly with the value of n. So time complexity is $T(n) = O(n)$.

The speed of nowadays computers is approximately 10^9 operations per 2 seconds. So we can also estimate the running time of our programs in seconds.

Time limit for the problem **2860 (Sum of integers on the interval)** is 1 second. So *for loop* solution will give **Time Limit Exceeded** (TLE) on some test cases. We must find an algorithm faster than $O(n)$.

We can notice that numbers from a to b form an arithmetic progression with difference $d = 1$. And their sum according to the formula equals to

$$\frac{a+b}{2}(b-a+1)$$

Solution to the problem can be just one line:

```
res = (a + b) * (b - a + 1) / 2;
```

This formula has 5 arithmetic operations regardless the value of n. So complexity of this solution is $O(1)$ and it is accepted in 1 second.

Example. Consider the next triple loop with complexity $O(n^3)$.

time_t represents the system time and date as some sort of integer. Function time(0) or time(NULL) returns number of seconds since January 1, 1970.

Change the value of n and estimate the running time of the program.

```
#include <stdio.h>
#include <ctime>

int i, j, k, n;
long long cnt;

int main(void)
{
  // Number of sec since January 1,1970
  time_t start = time(0);
  printf("Number of seconds started: %lld\n", start);

  n = 1000; // 10^9 operations per 2 seconds, CORE i5
  for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
  for (k = 1; k <= n; k++)
    cnt++;

  printf("Counter = %lld\n", cnt);
  time_t finish = time(0);
```

```
  printf("Number of seconds finished: %lld\n", finish);

  printf("Running time of the program in seconds: %lld\n", finish -
start);
  return 0;
}
```

Using the function **clock**(), you can estimate the running time in milliseconds. The C library function **clock**(**void**) returns the number of clock ticks elapsed since the program was launched. To get the number of seconds used by the CPU, you will need to divide by CLOCKS_PER_SEC.

Try to run the program with $n = 1000$ and $n = 2000$.

```
#include <stdio.h>
#include <ctime>

int i, j, k, n;
long long cnt;

int main(void)
{
  clock_t start = clock();

  n = 1000;
  for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
  for (k = 1; k <= n; k++)
    cnt++;

  printf("Counter = %lld\n", cnt);
  clock_t finish = clock();
  // now you can see running time milliseconds
  printf("Running time of the program in seconds: %f\n",
(float)(finish - start) / CLOCKS_PER_SEC);
  return 0;
}
```

E-OLYMP **1616. Prime number?** Check if the given number n is prime. The number is ***prime*** if it has no more than two divisors: 1 and the number itself.

► If number n is composite, it has a divisor not greater than $\lfloor\sqrt{n}\rfloor$. To check if n is prime, we must check its divisibility by 2, 3, …, $\lfloor\sqrt{n}\rfloor$. Complexity $O(\sqrt{n})$.

```
int IsPrime(int n)
{
  for (int i = 2; i <= sqrt(n); i++)
    if (n % i == 0) return 0;
  return 1;
}
```