Running time calculations

1. Basic operations in algorithm

An algorithm to solve a particular task employs some set of basic operations. When we estimate the amount of work done by an algorithm we usually do not consider all the steps such as e.g. initializing certain variables. Generally, the total number of steps is roughly proportional to the number of the basic operations. Thus, we are concerned mainly with the basic operations - how many times the basic operations have to be performed depending on the size of input.

Typical basic operations for some problems are the following:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find x in an array</td>
<td>Comparison of x with an entry in the array</td>
</tr>
<tr>
<td>Multiplying two matrices with real entries</td>
<td>Multiplication of two real numbers</td>
</tr>
<tr>
<td>Sort an array of numbers</td>
<td>Comparison of two array entries plus moving elements in the array)</td>
</tr>
<tr>
<td>Traverse a tree</td>
<td>Traverse an edge</td>
</tr>
</tbody>
</table>

*The work done by an algorithm, i.e. its complexity, is determined by the number of the basic operations necessary to solve the problem.*

Note: The complexity of a program that implements an algorithm is roughly the same, but not exactly the same as the complexity of the algorithm. Here we are talking about algorithms independent on any particular implementation - programming language or computer.

2. Size of input

Some algorithms are not dependent on the size of the input - the number of the operations they perform is fixed. Other algorithms depend on the size of the input, and these are the algorithms that might cause problems. Before implementing such an algorithm, we have to be sure that the algorithm will finish the job in reasonable time.

What is size of input? We need to choose some reasonable measure of size. Here are some examples:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Size of input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find x in an array</td>
<td>The number of the elements in the array</td>
</tr>
<tr>
<td>Multiply two matrices</td>
<td>The dimensions of the matrices</td>
</tr>
<tr>
<td>Sort an array</td>
<td>The number of elements in the array</td>
</tr>
<tr>
<td>Traverse a binary tree</td>
<td>The number of nodes</td>
</tr>
<tr>
<td>Solve a system of linear equations</td>
<td>The number of equations, or the number of the unknowns, or both</td>
</tr>
</tbody>
</table>
3. Counting the number of operations

The core of the algorithm analysis: to find out how the number of the basic operations depends on the size of the input.

A. In our calculation we will use the following rules to manipulate Big-Oh expressions:

Let $T_1(N)$ and $T_2(N)$ be the number of operations necessary to run two program segments respectively, and let $T_1 = O(f_1(N))$, $T_2 = O(f_2(N))$.

Then:
- $T_1(N) + T_2(N) = \max(O(f_1(N)), O(f_2(N)))$
  
  where $\max(O(f_1(N)), O(f_2(N)))$ is determined in the following way:

  - If $f_1(N) = O(f_2(N))$, then $\max (O(f_1(N)), O(f_2(N))) = O(f_2(N))$
  - If $f_2(N) = O(f_1(N))$, then $\max (O(f_1(N)), O(f_2(N))) = O(f_1(N))$

  **Example:** $\max(O(n^2), O(n^3)) = O(n^3)$.

- $T_1(N) \times T_2(N) = O(f_1(N) \times f_2(N))$

B. There are four rules to count the operations:

**Rule 1: for loops - the size of the loop times the running time of the body**

The running time of a `for` loop is at most the running time of the statements inside the loop times the number of iterations.

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

- a. Find the running time of statements when executed only once:
  
  The statements in the loop heading have fixed number of operations, hence they have constant running time $O(1)$ when executed only once.
  
  The statement in the loop body has fixed number of operations, hence it has a constant running time when executed only once.

- b. Find how many times each statement is executed.

```plaintext
for( i = 0; i < n; i++) // i = 0; executed only once: O(1)
    // i < n; n + 1 times O(n)
    // i++ n times O(n)
    // total time of the loop heading: O(1) + O(n) + O(n) = O(n)

sum = sum + i; // executed n times, O(n)
```
The loop heading plus the loop body will give: \( O(n) + O(n) = O(n) \).

Loop running time is: \( O(n) \)

**If**

a. the size of the loop is \( n \) (loop variable runs from 0, or some fixed constant, to \( n \)) and
b. the body has constant running time (no nested loops)

**then** the time is \( O(n) \)

**Rule 2:** Nested loops – the product of the size of the loops times the running time of the body

The total running time is the running time of the inside statements times the product of the sizes of all the loops

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

Applying Rule 1 for the nested loop (the ‘j’ loop) we get \( O(n) \) for the body of the outer loop.

The outer loop runs \( n \) times, therefore the total time for the nested loops will be \( O(n) \times O(n) = O(n^2) \)

What happens if the inner loop does not start from 0?

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

Here, the number of the times the inner loop is executed depends on the value of \( i \)

- \( i = 0 \), inner loop runs \( n \) times
- \( i = 1 \), inner loop runs \( n-1 \) times
- \( i = 2 \), inner loop runs \( n-2 \) times
  ...
- \( i = n-2 \), inner loop runs 2 times
- \( i = n-1 \), inner loop runs once.

Adding the right column, we get: \( (1 + 2 + \ldots + n) = \frac{n(n+1)}{2} = O(n^2) \)

**General rule for loops:**

Running time is the product of the size of the loops times the running time of the body.
Example:

```c
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < 2n; j++)
        sum++;
```

We have one operation inside the loops, and the product of the sizes is $2n^2$
Hence the running time is $O(2n^2) = O(n^2)$

**Note:** if the body contains a function call, its running time has to be taken into consideration.

```c
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < n; j++)
        sum = sum + function(sum);
```

Assume that the running time of function(sum) is known to be $\log(n)$.
Then the total running time will be $O(n^2*\log(n))$

**Rule 3: Consecutive program fragments**

The total running time is the maximum of the running time of the individual fragments

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

```c
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < 2n; j++)
        sum++;
```

The first loop runs in $O(n)$ time, the second - $O(n^2)$ time, the maximum is $O(n^2)$

**Rule 4: If statement**

```c
if C
    S1;
else
    S2;
```

The running time is the maximum of the running times of S1 and S2.
Here are some examples:

3.1. **Search in an unordered array of elements.**

```plaintext
for (i = 0; i < n; i++)
    if (a[i] == x) return 1;  // 1 means succeed
return -1;  // -1 means failure, the element is not found
```

The basic operation in this problem is *comparison*, so we are interested in how the number of comparisons depends on *n*.

Here we have a loop that runs at most *n* times:
- If the element is not there, the algorithm needs *n* comparisons.
- If the element is at the end, we need *n* comparisons.
- If the element is somewhere in between, we need less than *n* comparisons.

So how do we determine the number of operations (the comparisons), since in this problem it depends on the particular input?

Here comes the notion of a **worst-case scenario and average-case scenario**.

In the **worst case** (element not there, or located at the end), we have *n* comparisons to make.

To find what is in the average case, we have to consider the possible cases:
- Element is at the first position: 1 comparison
- Element is at the second position: 2 comparisons
- ....

We compute the sum of the operations for the possible cases and then divide by the number of the cases, assuming that all cases are equally likely to happen:

\[ 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \]

Thus in the average case the number of comparisons would be \( \frac{n(n+1)}{2} \), which gives \( O(n^2) \) running time.

3.2. **Search in a table n x m**

```plaintext
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        if (a[i][j] == x) return 1;  // 1 means succeed
return -1;  // -1 means failure - the element is not found
```

Here the inner loop runs at most *m* times and it is located in another loop that runs at most *n* times, so in total there would be at most \( n*m \) operations, running time is \( O(n*m) \).
3.3. Finding the greatest element in an array

amax = a[0];
for (i = 1; i < n; i++)
    if (a[i] > amax) amax = a[i];

Here the number of comparisons is always n-1.
The amount of work depends on the size of the input, but does not depend on the particular values.

The running time here is O(n), we disregard "-1" because the difference for large n is negligible.

3.4. More examples

a. $O(n^3)$
   sum = 0;
   for( i = 0; i < n; i++)
       for( j = 0; j < n * n; j++)
           sum++;

b. $O(n^2)$
   sum = 0;
   for( i = 0; i < n; i++)
       for( j = 0; j < i; j++)
           sum++;

c. $O(n^3)$
   sum = 0;
   for( i = 0; i < n; i++)
       for( j = 0; j < i*i; j++)
           for( k = 0; k < j; k++)
               sum++;

d. $O(n^2)$
   sum = 0;
   for( i = 0; i < n; i++)
       sum++;

val = 1;
for( j = 0; j < n*n; j++)
    val = val * j;

e. $O(n^3\log n)$
   sum = 0;
   for( i = 0; i < n; i++)
       sum++;
   for( j = 0; j < n*n; j++)
       compute_val(sum, j);

   The complexity of the function compute_val(x,y) is given to be $O(n\log n)$