Algorithm Analysis

Objectives:

- to present **ways to measure the efficiency** of an algorithm
- to define the concept of **growth rate** of the cost of an algorithm depending on the growth of the problem size
- to define the notions of **lower bound** and **upper bound** for a growth rate
- review some **mathematical tools** needed for algorithm analysis

Measuring Algorithm Efficiency

Purpose:

- to estimate the resources needed by an algorithm
- to choose from several algorithm solving the same problem

Efficiency has to do with the use of resources.

Resources used by an algorithm:

- time
- storage
- effort required to code ...

We will concentrate on time efficiency:

The amount of time required by an algorithm

Ways to measure time efficiency

1. Implement the algorithm and **run it** on a computer

   depends on machine!

2. **Sum up the times required by all the commands** used to the algorithm has to perform

   depends on implementation!

3. **Count only the specific (basic) operations** performed by the algorithm

Example:

```
int largest (int* array, int n){
    int currlarge = 0;
    for (int i=0 ; i<n ; i++)
        if (array(i) > currlarge)
            currlarge = array (i);
    return currlarge;
}
```

Note. This program only works if at least one element of the array is greater or equal than 0.

- the basic operation is the comparison “**array(i) > currlarge**“ because it is the operation that is absolutely needed to solve this problem

The **basic operation is** usually **selected** to be the operation that:

- is needed to solve the problem
- is the most time consuming
- is the most frequently used

The **time** required by an algorithm usually **depends on the "size"** of the problem instance it is presented with:

- *it usually grows with the growth of the growth of the problem size*

We need to study this dependency!

Problem size: depends on the number of symbols needed to describe it.

Example:

- an instance of the maximum search problem is described by the elements of the array
- since there are n elements the description of an instance of this problem requires n times the number of symbols needed to describe an element

We will **specify this dependency** as a function defined over the problem sizes which gives the number of basic operation for each problem size

Best, Worst and Average Cases

Worst case:

- an upper bound on the number of basic operations that will be performed

Best case:

- a lower bound on the number of basic operations that will be performed

Average case:

- the average number of basic operations that will be performed over all the problems of a given size

Example:

Consider the following program for searching for a given number in an array of size *n* (assumes that the element is in the array!):

```
int search(int* array, int n, int x){
    int i = 0;
    while (array(i) != x)
        i++;
    return i;
}
```

Worst case: happens when x is in the last position - $f_b(n)=n$

Best case: happens when x is in the first position - $f_w(n)=1$

Average case: $\frac{1+2+3+\ldots+n}{n} = \frac{n(n+1)}{2n}$

Calculating the Running Time

1. **sequences:** the running time for a sequence of statements is the sum of the running times for each of the statements
2. **loops:** the running time for a loop is the sum over all the iterations of the running times of the body for those iterations
3. **binary decisions:** the worst(best)-case running time for a binary decision is the maximum(minimum) between the running times for its two branches
4. **recursion:** the running time for a recursive program is given by a recursion

Examples:

1.

```
x = a+b*(c+d);
y = (x+2*a*b)/3;
...
```

2.

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

3.

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

4.

```
sum = 0;
if(n%2)
    for (i=1 ; i<=n ; i++)
        sum += i;
else
    for (i=2 ; i<= n ; i+=2)
        sum += i;
...
```

5.

```
int Bsearch(int* array, int n, int x){
    if (n == 1)
        return array(0) == x;
    else
        k = n/2;
        if (array(k)<x)
            return Bsearch(&array(0),k,x);
        else
            return Bsearch(&array(k),k,x);
}
```

How do we compare two algorithms based on their running time expressed as a function of the problem size?

This reduces to the problem: **How do we compare two functions?**

Assume that the problem size is characterized by a non-negative integer n.

Definition 1. The running time $T(n)$ of an algorithm is $O(f(n))$ if there exist the positive constants c and n_0 such that

$$|T(n)| \le c \cdot |f(n)|$$

for all $n > n_0$.

Definition 2. The running time *T(n)* of an algorithm is Ω *(f(n))* if there exist the positive constants *c* and n_0 such that

$$(T(n)) \geq c \cdot |f(n)|$$

for all n>n_0.

Definition 3. The running time *T(n)* of an algorithm is Θ*(f(n))* if *T(n)* is both O(f(n)) and Ω *(f(n))*.