Analysis of algorithms

ALGORITHMS

An algorithm is a detailed sequence of actions to perform to accomplish some task. The name comes from an Iranian mathematician *Al-Khawarizmi*. A more precise definition :
– *A finite set of step-by-step instructions for a problem-solving or computation procedure, especially one that can be implemented by a computer.*
However, the steps of an algorithm are not dependent on any specific computer language. It gives an idea of the logical steps used, without bothering about the syntax of a language. It helps us in checking if the solution process is correct and efficient, even before it is run on a computer. An algorithm can be coded in any computer language. You might say that the algorithm can be described through a "pseudo code". Here are some examples of algorithms.

Finding minimum of 3 numbers:

1. Read the three numbers into a, b and c.
2. let min be the variable to hold the minimum value.
3. if (a < b) then set min = a
   else set min = b
4. if (c < min) then set min = c
5. Print min.

To find *sdiv* the smallest divisor of a number. (Alg. A)

1. If number is even, *sdiv* =2, Go to step 7.
2. Let divisor be 3.
3. remainder = number mod divisor
4. if remainder=0, *sdiv* = 3, Go to step 7.
5. As long as *remainder is not zero* & *divisor* < *number*
   keep on doing the following:

   5a. divisor = divisor +2.
   5b. remainder = number mod divisor.

6. if divisor = number , *sdiv* = 1, else *sdiv* = divisor
7. print *sdiv*.

To find *sdiv* smallest divisor of a number (Alg. B)

1. If number is even, *sdiv* =2, Go to step 8.
2. If number is divisible by 3, *sdiv* = 3, Go to step 8.
3. Let divisor be 3.

4. remainder = number mod 3
5. As long as *remainder is not zero* & *divisor * divisor* < *number*
   keep on doing the following:
   a. divisor = divisor +2.
   b. remainder = number mod divisor.
6. if number mod divisor is zero, *sdiv* = divisor
   Else, *sdiv* = 1.
7. print *sdiv*.

Time Complexity of Algorithms:

Counting the number of operations involved in the algorithm to handle *n* items. Meaningful comparison possible for very large values of n.

Algorithm Analysis: Loops

LOOP 1:

```
a= 0; b = 0;
for (k=0; k< n; k++)  {
     for (j = 0; j < m; j++)
          a= a + j;
}
```

It takes nm addition operations.

LOOP 2:

```
a= 0; b = 0;
for (k=0; k< n; k++)  {
     a= a+k;
     for (j = 0; j < m; j++)
          b= b + j;
}
```

It takes nm+ n addition operations.

LOOP 3:

```
total = 0;
for (k=0; k<n; ++k)  {
     rows[k] = 0;
     for (j = 0; j <n; ++j){
          rows[k] = rows[k] + matrix[k][j];
          total = total + matrix[k][j];
     }
}
```

It takes $2n^2$ addition operations

LOOP 4:

```
total =0;
for (k=0; k<n; ++k)
     rows[k] = 0;
     for (j = 0; j <n; ++j)
          rows[k] = rows[k] + matrix[k][j];
     total = total + rows[k];
}
```

This one takes $n^2 + n$ operations .

Complexity of testing elements of an array

Let ar[ ] be an array containing *n* integer values. It is desired to find the number of elements having value more than 50.

```
sum = 0;
for(k = 0; k<n;   k++)
     if(ar[k]>50)
          sum+ = ar[k];
```

for *each* value in the array, *one* comparison is being carried out, while the assignment statement may get executed only for some of the values.

Worst case: All values are more than 50, (all values are checked and total gets incremented every time).
no. of operations = 2n

Best case: When all values are less than 50. (all values are checked and none gets incremented
number of operations = n

Complexity of Linear Search:

searching for an element in an array

Consider the task of searching an array with n elements, to see if it contains a particular value.

```
i = 0;

while (i< n && array[i] != target)
      i = i + 1;

if (i < n)
     printf ("Yes, target found \n" );
else
     printf("No, target not found \n" );
```

The work involved : Checking target value with each of the n elements.

no. of operations:	1	(best case)
	n	(worst case)
	n/2	(average case)

Computer scientists tend to be more concerned about the worst time complexity.

Worst Case complexity.

The worst case guarantees that the performance of the algorithm will be at least as good as the analysis indicates.

Complexity of Selection Sort

The sorting process consists of reordering the elements in an array so that they are arranged in an increasing or decreasing sequence. One simple sorting method is known as the Selection sort. To sort the elements in ascending order, it examines each element in the list, locates the smallest element and swaps it with the first element in the array. Then it examines the array starting with the second element, locates the smallest element and swaps it with the second element. The steps are repeated till it has swapped the last element in place.

[56 25 37 58 95 19 73 30]

After one pass:

[19 25 37 58 95 56 73 30]

After second pass:

[19 25 37 58 95 56 73 30]

Algorithm for selection sort:

```
for lh = 0 to n
{
   rh = lh,
   for j = lh +1 to n
      if  element at j   < element at rh
           rh = j ;

  swap element at lh with element at rh
}
```

Analyzing performance of selection sort

In the first cycle, the inner loop examines n elements, so it takes n operations. Next cycle it examines n – 1 elements, then n – 2 elements and so on till it examines only one element in the end. Thus total number of operations is given by the expression

$T = n + n - 1 + n - 2 + \ldots\ldots + 3 + 2 + 1$

$= \Sigma j$

$= n (n + 1) / 2$

$= (n^2 + n)/ 2$

Is this term proportional to n or n^2 ? Study the following table

n	**T**
10	55
100	5,050
400	80,200
1000	500,500

It is obvious that T is more nearer to n^2 than n. Similarly for any algorithm we can work out a complicated expression for T, but we want a simpler *qualitative* answer, so that we can compare performance of two algorithms.

Time complexity measured in terms of order of n:

The time complexity of an algorithm is given by the function which counts the total number of operations for n elements of data. What is really of concern here is not the terms contained in the function, but rather a description of how the function grows with n. Concretely, consider the two functions:

$f(n) = n + 20$

$g(n) = n – 10$

Which function grows faster?

Try with small values of n say up to 100. It would show that f(n) grows faster. Now try values between 1000 and 10,000. What does it show? There is hardly any noticeable difference between values of f(n) and g(n). Both grow linearly with n.

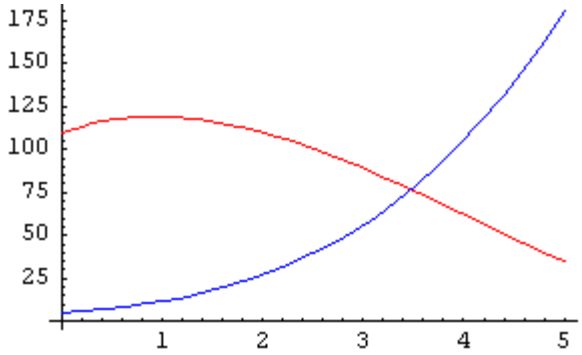
Let us now consider another pair of functions

$$f(n) = n^3 – 12n^2 + 20n + 110$$

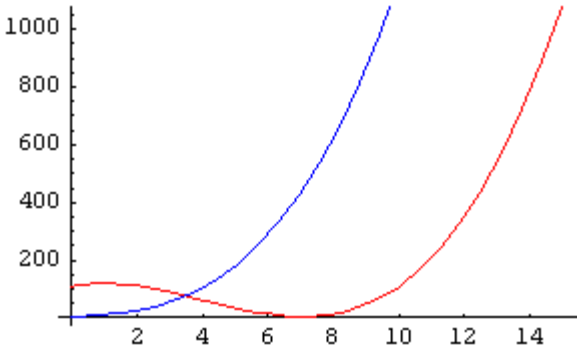
$$g(n) = n^3 + n^2 + 5n + 5$$

They look quite different, but how do they behave? Let's look at a few plots of the function

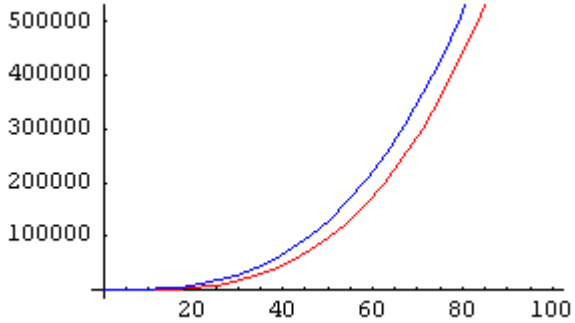
(f(n) is in red, g(n) in blue):



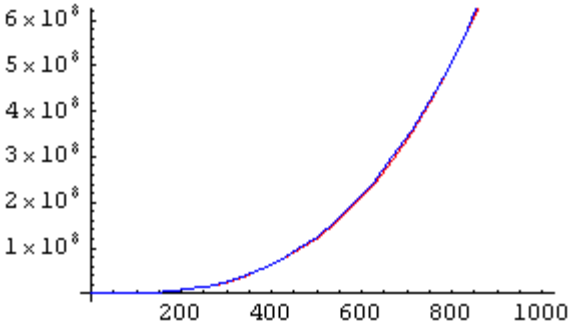
Plot of f and g, in range 0 to 5



Plot of f and g, in range 0 to 15



Plot of f and g, in range 0 to 100



Plot of f and g, in range 0 to 1000

In the first plot, which covers very small values of n, the curves appear somewhat different. In the second plot, they start going the same way *(sort of)*, in the third plot for moderate values of n, there is only a very small difference, and in the final plot for appreciable values of n, they are virtually identical. In fact, they approach n^3, the dominant term. *As n gets larger, the other terms become minuscule in comparison to* n^3.

As you can see, improving an algorithm's non-dominant terms doesn't help much. What really matters is the dominant term. This is why we adopt a new notation for this. We say that:

$f(n) = n^3 - 12n^2 + 20n + 110$ is of order n^3 *,that is* $O(n^3)$

The big-O Notation

We want to understand how the performance of an algorithm responds to changes in problem size. Basically the goal is to provide a *qualitative* insight on number of operations for a problem size of *n* elements. The total number of operations required by an algorithm can be described through a mathematical expression in *n*.

The big-O notation is a way of measuring the order of magnitude of a mathematical expression
O(n) means of the Order of n

Consider
$n^4 + 3n^2 + 10 = f(n)$

The idea is to reduce the expression so that it captures the qualitative behavior in simplest possible terms. We eliminate any term whose contribution to the total ceases to be significant as n takes on large values.
We also eliminate any constant factors, as these have no effect on the overall pattern with increase in n. Thus we may approximate f(n) above as

$O(n^4 + 3_1n^2 + 10) = O(n^{4)}$

The complexity of an expression

$O(3n^4 + 35n^2 + 100)$

is evaluated by first dropping the insignificant terms, to yield

$O(3n^4)$

and then dropping the constant factor to get

$O(n^4)$

Order Notation

O(1) or "Order One": Constant time

- does *not* mean that it takes only one operation
- *does* mean that the work *doesn't change* as n changes
- is a notation for "*constant work*"

O(n) or "Order n": Linear time

- does *not* mean that it takes n operations
- *does* mean that the work changes in a way that is *proportional* to n
- is a notation for "*work grows at a linear rate*"

O(n^2) or *"Order n^2 ": Quadratic time*
O(n^3) or *"Order n^3 ": Cubic time*

O(*log n*) or *"Order log n ": Log time*

Some algorithms even take less time than the number of elements in the problem. There is a notion of *logarithmic time* algorithms.

We know $10^3 = 1000$
Taking log of both sides to base 10 , we can write it as

$log_{10}1000 = 3$

Similarly suppose we have

$2^6 = 64$

Taking log of both sides *to base 2*

$log_2 64 = 6$

If the work of an algorithm can be reduced by half in every step, and in k steps we are able to solve the problem then, k and n can be related through

$2^k = n$

or in other words

$k = log_2 n$

This algorithm will be having **a logarithmic time** complexity, usually written as **O(ln n)** or **O(lg n).** Because $log_a n$ will increase much more slowly than n itself, logarithmic algorithms are generally very efficient.

Logarithms

$a^b = m$
Taking log of both sides with respect to base a,
$b \log_a a = \log_a m$
or
$b = \log_a m$
Thus knowing any two quantities, you can find the 3rd one.

There are a large number of algorithms whose complexity is **O(n $log_2 n$) .**
Note: O(n $log_2 n$) will be much less than **O(n^2).**

O(2^n) or *" Exponential Order "*

Finally note that in computer science, there are some algorithms whose efficiency is dominated by a term of the form a^n. These are called ***exponential algorithms***, and are of order **O(2^n).** They are of more theoretical rather than practical interest because they cannot reasonably run on typical computers for even for moderate values of n.

Comparison of N, $logN$ and N^2

N	O(LogN)	$O(N^2)$
16	4	256
64	6	4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12
1,073,741,824	30	1.15E+18

Complexity related Problems

1.Algorithm A runs in $O(N^2)$ time, and for an input size of 4, the algorithm runs in 10 milliseconds, how long can you expect it to take to run on an input size of 16?

$$\frac{4^2}{10} = \frac{16^2}{x}$$

$$\Rightarrow \quad x = 160\text{ms}$$

2. **Algorithm A runs in $O(N^3)$ time. For an input size of 10, the algorithm runs in 7 milliseconds. For another input size, the algorithm takes 189 milliseconds. What was that input size?**

$$\frac{10^3}{7} = \frac{N^3}{189} \quad \Rightarrow \quad N = 30$$

3. **Algorithm A runs in $O(\log_2 N)$ time, and for an input size of 16, the algorithm runs in 28 milliseconds, how long can you expect it to take to run on an input size of 64?**

$$\frac{\log 16}{28} = \frac{\log 64}{x}$$

How to find the log values?

$b = \log_a m$, $\quad a^b = m$

To find $\log_2 16$ use $2^b = 16$
This gives $b = 4$.

To find $\log_2 64$ use $2^b = 64$
This gives $b = 6$.

$$\frac{\log 16}{28\text{ ms}} = \frac{\log 64}{x} \quad \Rightarrow$$

$$\frac{4}{28\text{ ms}} = \frac{6}{x}$$

=> x = 42 ms

4.What is the computational complexity of the following algorithm?

```
sum = 0;
for(j=1; j <= N; j++)
       sum = sum + j ;
```

It has *one* arithmetic operation per loop. We can express this as a summation

$$\sum_{j=1}^{n} 1$$

What is the value of *sum* when the loop is done? It is

$$\text{sum} = \sum_{j=1}^{n} j$$

5.What is the computational complexity of the following algorithm?

```
sum = 0;
for(j= 10; j <= N; j++)
       sum = sum + j ;
```

Again it is one operation per loop, but now the loop does not start from 1. We can express the time complexity of the algorithm as a summation

$$\sum_{j=10}^{n} 1$$

while $\text{sum} = \sum_{j=10}^{n} j$

6. What is the computational complexity of the following algorithm?

```
sum = 0;
for(j=1; j <= N; j++){
   for(k=1; k <= j; k++)
       sum = sum + j * k;
 }
```

We can express this in summation form as

$$\sum_{j=1}^{N}\sum_{k=1}^{j} 2$$