Measuring Performance of Algorithms

There are *two aspects* of algorithmic performance:

- Time
  - Instructions take time.
  - How fast does the algorithm perform?
  - What affects its runtime?

- Space
  - Data structures take space
  - What kind of data structures can be used?
  - How does choice of data structure affect the runtime?

Algorithms can not be compared by running them on computers. Run time is system dependent. Even on same computer would depend on language Real time units like microseconds not to be used.

Generally concerned with how the amount of work varies with the data.

Measuring Time Complexity

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

Complexity of Linear Search

Consider the task of searching a list to see if it contains a particular value.

• A useful search algorithm should be *general.*

• Work done varies with the size of the list

• What can we say about the work done for list of *any* length?

```
i = 0;

while (i < MAX && this_array[i] != target)
     i = i + 1;

if (i <MAX)
    printf ( “Yes, target is there \n” );
else
    printf( “No, target isn’t there \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 concerned about the

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

Average Case Complexity:
It is the best statistical estimate of actual performance, and tells us how well an algorithm performs if you average the behavior over all possible sets of input data. However, it requires considerable mathematical sophistication to do the average case analysis.

Algorithm Analysis: Loops

Consider an n X n two dimensional array. Write a loop to store the row sums in a one-dimensional array `rows` and the overall total in `grandTotal`.

LOOP 1:

```
grandTotal = 0;
for (k=0; k<n-1; ++k)  {
    rows[k] = 0;
    for (j = 0; j <n-1; ++j){
        rows[k] = rows[k] + matrix[k][j];
        grandTotal = grandTotal + matrix[k][j];
    }
}
```

It takes $2n^2$ addition operations

LOOP 2:

```
grandTotal =0;
for (k=0; k<n-1; ++k)
    rows[k] = 0;
    for (j = 0; j <n-1; ++j)
        rows[k] = rows[k] + matrix[k][j];
    grandTotal = grandTotal + rows[k];
}
```

This one takes $n^2 + n$ operations

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. The Big-O notation is a way of measuring the order of magnitude of a mathematical expression

O(n) means on the Order of n

Consider

$n^4 + 3_1n^2 + 10 = f(n)$

The idea is to reduce the formula in the parentheses 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 becomes large.

We also eliminate any constant factors, as these have no effect on the overall pattern as n increases. Thus we may approximate f(n) above as

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

Let g(n) = n^4

Then the order of f(n) is O[g(n)].

Definition: f(n) is O(g(n)) if there exist positive numbers c and N such that f(n) < = c g(n) for all n >=N.

i.e. f is big –O of g if there is c such that f is not larger than cg for sufficiently large value of n (greater than N)

c g(n) is an upper bound on the value of f(n)

That is, the number of operations is at worst proportional to g*(n)* for all large values of *n*.

How does one determine c and N?

Let f(n) = $2\,n^2 + 3\,n + 1 = O(n^2)$

Now $2\,n^2 + 3\,n + 1 <= c\,n^2$

Or $2 + (3/n) + (1 / n^2) <= c$

You want to find c such that a term in f becomes the largest and stays the largest. Compare first and second term. First will overtake the second at N = 2,
so for N= 2, c >= 3.75,
for N = 5, c >= slightly more than 2,
for very large value of n, c is almost 2.

g is almost always > = f if it is multiplied by a constant c

Look at it another way : suppose you want to find weight of elephants, cats and ants in a jungle. Now irrespective of how many of each item were there, the net weight would be proportional to the weight of an elephant.

Incidentally we can also say f is big -O not only of n^2 but also of n^3 , n^4 , n^5 etc (HOW ?)

- Loop 1 and Loop 2 are both in the same big-O category: $O(n^2)$

Properties of Big-O notation:

$O(n) + O(m) = O(n)$ if $n >= m$

The function log n to base a is order of O(log n to base b) For any values of a and b (you can show that any log values are multiples of each other)

Linear search Algorithm:

Best Case - It's the first value
"order 1," O(1)
Worst Case - It's the last value, n
"order n," O(n)
Average - N/2 (if value is present)
"order n," O(n)

Example 1:

Use big-O notation to analyze the time efficiency of the following fragment of C code:

```
for(k = 1; k <= n/2; k++)
{
    .
    .
    for (j = 1; j <= n*n; j++)
    {
```

```
            .
            .
        }
}
```

Since these loops are nested, the efficiency is $n^3/2$, or O(n^3) in big-O terms.

Thus, for two loops with O[$f_1(n)$] and O[$f_2(n)$] efficiencies, the efficiency of the nesting of these two loops is O[$f_1(n) * f_2(n)$].

Example 2:

Use big-O notation to analyze the time efficiency of the following fragment of C code:

```
for (k=1; k<=n/2; k++)
{
    .
    .
}
for (j = 1; j <= n*n; j++)
{
    .
    .
}
```

The number of operations executed by these loops is the sum of the individual loop efficiencies. Hence, the efficiency is $n/2+n^2$, or O(n^2) in big-O terms.

Thus, for two loops with O[$f_1(n)$] and O[$f_2(n)$] efficiencies, the efficiency of the sequencing of these two loops is O[$f_D(n)$] where $f_D(n)$ is the dominant of the functions $f_1(n)$ and $f_2(n)$.

Complexity of Linear Search

In measuring performance, we are generally concerned with how the amount of work varies with the data. Consider, for example, the task of searching a list to see if it contains a particular value.

• A useful search algorithm should be *general*.

• Work done varies with the size of the list

• What can we say about the work done for list of *any* length?

```
i = 0;

while (i < MAX && this_array[i] != target)
     i = i + 1;

if (i <MAX)
    printf ( “Yes, target is there \n” );
else
    printf( “No, target isn’t there \n” );
```

Order Notation

How much work to find the target in a list containing N elements?
Note: we care here only about the *growth rate* of work. Thus, we *toss out all constant values.*

- Best Case work is *constant*; it does not grow with the size of the list.
- Worst and Average Cases work is *proportional* to the size of the list, N.

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*

Algorithms whose efficiency can be expressed in terms of a polynomial of the form

$$a_m n^m + a_{m-1} n^{m-1} + \ldots + a_2 n^2 + a_1 n + a_0$$

are called ***polynomial algorithms***. **Order O(n^m).**

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

We know $10^3 = 1000$
So we can write it as $\log_{10} 1000 = 3$

Similarly suppose we have

$2^6 = 64$

then we can write

$log_2 64 = 6$

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

$2^k = n$

or in other words

$log_2 n = k$

This algorithm will be having **a logarithmic time** complexity ,usually written as **O(ln n).**
Because $log_a n$ will increase much more slowly than n itself, logarithmic algorithms are generally very efficient. It also can be shown that it does not matter as to what base value is chosen.

Example 3:

Use big-O notation to analyze the time efficiency of the following fragment of C code:

```
k = n;
while (k > 1)
{
    .
```

```
    .
    k = k/2;
}
```

Since the loop variable is cut in half each time through the loop, the number of times the statements inside the loop will be executed is $\log_2 n$.
Thus, an algorithm that halves the data remaining to be processed on each iteration of a loop will be an O($log_2 n$) algorithm.

There are a large number of algorithms whose complexity is **O(n $log_2 n$) .**

Finally there are algorithms whose efficiency is dominated by a term of the form a^n
These are called ***exponential algorithms***. They are of more theoretical rather than practical interest because they cannot reasonably run on typical computers 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