Chapter 2 Algorithms and complexity analysis

2.1 Relationship between data structures and algorithms

The primary objective of programming is to efficiently process the input to generate the desired output. We can achieve this objective in an efficient and neat style if the input data is organized properly. Data Structures is nothing but ways and means of organizing data so that it can be processed easily and efficiently. Data structures dictate the manner in which the data can be processed. In other words, the choice of an algorithm depends upon the underlying data organization.

Let us try to understand the concept with a very simple example. Assuming that we have a collection of 10 integers and we want to find out whether a key is present in the collection or not.

Organization 1:
Data are stored in 10 disjoint variables A0, A2, A3, …, and A9

Algorithm

```
found = false;
if (key = = A0)        found = true;
else if (key = = A1)   found = true;
else if (key = = A2)   found = true;
else if (key = = A3)   found = true;
else if (key = = A4)   found = true;
else if (key = = A5)   found = true;
else if (key = = A6)   found = true;
else if (key = = A7)   found = true;
else if (key = = A8)   found = true;
else if (key = = A9)   found = true;
```

Figure 2-1 - A novice data organization and resulting algorithm

We can organize the data in many different ways. We start with putting the data in 10 disjoint variables and use simple if-else statements to

determine whether the given key is present in the collection or not. This organization and the corresponding algorithm are shown Figure 2-1.

A second possible data organization is to put the data in an array. We now have the choice of using if-else statements just like before or use a loop to iterate through the array to determine the presence of the key in our collection. This organization and the algorithm are presented in Figure 2-2.

Organization 2:
Data are stored in an array A of 10 elements

Algorithm

```
const int N = 10;
found = false;
for (int i = 0; i < N; i++)
   if (A[i] == key)
   {
      found = true;
      break;
   }
```

Figure 2-2 – Linear Search Algorithm

Yet another possibility is to store the data in an array in ascending (or descending) order. As shown in Figure 2-3, we now have a third possibility, the binary search, in our choice of algorithms.

Organization 3: Data are stored in an array A in ascending order

Algorithm

```
const int N = 10;
found = false;
low = 0;
high = N - 1;
while (( ! found) && ( low <= high))
{
   mid = (low + high)/2;
   if (A[mid] = = key)
      found =     true;
   else if (A[mid] > key)
      high   = mid - 1;
   else
      low    = mid + 1;
}
```

Figure 2-3 – Binary Search Algorithm

As can be clearly seen from these examples, choice of the algorithm depends upon the underlying data organization. With the first organization we had only one choice. With the second one we have two choices and the third one gives us the option to choose from any of the three algorithms.

Now the question is: which one of the three organizations is better and why?

We can easily see that the second one is better than the first. In both the cases the number of comparisons will remain the same but the second algorithm is more scalable as it is much easier to modify the second one to handle larger data sizes as compared to the first one. For example, if the data size was increased from 10 to 100, the first algorithm would require declaration of 90 more variable, and adding 90 new else-if clauses. Whereas, in the second case, all one has to do is to redefine N to be equal to 100. Nothing else needs to change.

However, the comparison between the second and third one is more interesting.

As can be easily seen, the second algorithm (linear search) is much simpler as compared to the third one (binary search). Furthermore, the linear search algorithm imposes less restriction on the input data as compared to the binary search algorithm as it requires data sorted in ascending or descending order. From an execution point of view, the body of the while loop in the linear search seems to be more efficient as compared to the body of the loop in the binary search algorithm. So, from the above discussion it appears that linear search algorithm is better than the binary search algorithm.

So, why should we at all bother about the binary search algorithm? Does it offer any real benefit over linear search? Actually it does and it is much superior to the linear search algorithm. In order to understand why it is so, we have to analyze a larger problem such as the one discussed next.

National Database and Registration Authority (NADRA), Government of Pakistan, maintains a database of the citizens of Pakistan which currently has over 80 million records and is growing incessantly.

Let us assume that we need to search for a record in this database. Let us also assume that we have a computer that can execute 100,000 iteration of the body of the while loop in the linear search algorithm in one second. It is also assumed that, on the average, each iteration of the body of the while loop in the binary search algorithm takes 3 times more than one iteration of the while loop in the linear search algorithm.

We now analyze the problem of searching a record in the NADRA database using these two algorithms and the result is summarized in Table 2-1. The picture looks obnoxiously in favor of binary search. How come binary search looks 1,000,000 times faster than the linear search? Why did we get these numbers? Is there anything wrong in our analysis or is it a tricky example?

	Linear Search	Binary Search
Number of iterations of the body of the loop in one second	100,000	~33000
Searching a record in 80 Million records		
Worst case – record is not found	800 seconds	~0.0008 seconds
Average case	400 seconds	~0.0004 seconds
Number of searches in one hour (average case)	9	~ 9 million
Number of searches per day	216	~ 213 Million

Table 2-1 - Comparison of Linear Search and Binary Search Algorithms

There is actually nothing wrong with this analysis – in general, binary search is much faster than the linear search and with increase in data size (e.g number of records in the database) the difference will become wider and even more staggering.

How do we get to that conclusion? The general question is: given two algorithms for the same task, how do we know which one will perform better? This question leads to the subject of complexity of algorithms which is the topic of our next discussion.

2.2 Algorithm Analysis

Efficiency of an algorithm can be measured in terms of the execution time and the amount of memory required. Depending upon the limitation of the technology on which the algorithm is to be implemented, one may decide which of the two measures (time and space) is more important. However, for most of the algorithms associated with this course, time requirement comparisons are more interesting than space requirement comparisons and hence we will concentrate more on the that aspect.

Comparing two algorithms by actually executing the code is not easy. To start with, it would require implementing both the algorithms on similar machines, using the same programming language, same complier, and same operating system. Then we need to generate a large number of data sets for profiling the algorithms. We can then try to measure the performance of these algorithms by executing them. These are no easy tasks and in fact are sometimes infeasible. Furthermore, a lot depends upon the input sequence. For the same amounts of input data with a different permutation, an algorithm may take drastically different execution times to solve the problem.

Therefore, with this strategy, whenever we have a new idea to solve the same problem, we cannot answer the question whether the program's running time would be acceptable for the task or not without implementing it and then comparing it with other algorithms by executing all these algorithms for a number of data sets. This is rather a very inefficient, if not absolutely infeasible, approach. We therefore need a mathematical instrument to estimate the execution time of the algorithm without having to implement the algorithm.

Unfortunately, in the real world, coming up with an exact and precise model of the system under study is an exception rather than a rule. For many problems that are worth studying it is not easy, if not impossible, to model the problem with an exact formula. In such cases we have to work with an approximation of the precise problem. Model for execution time for algorithms falls in the same category because:

1. each statement in an algorithm requires different execution time, and
2. presence of control flow statements renders it impossible to determine which subset of the statements in the algorithm will actually be executed.

We therefore need some kind of simplification and approximation for determining the execution time of an algorithm.

For this purpose, a technique called the time complexity analysis is used. In the time complexity analysis we define the time as a function of the problem size and try to estimate the growth of the execution time with the growth in problem size. For example, as we shall see a little later, the execution time of the linear search function grows linearly with the increase in data size. So, increasing the data size by a factor of 1000 will increase the execution time of the linear search algorithm also by a factor of 1000. On the other hand, execution time of the binary search algorithm increases logarithmically with the increase in the data size. So, increasing the data size by a factor of 1000 will increase the execution time by a factor of $\lceil \log_2 1000 \rceil$, that is, by a factor of 10 only. That is, the growth of Binary Search is much slower as compared to the growth of Linear Search. This tells us why Binary Search is better than Linear Search and why the difference between binary search and linear search widens with the increase in input data size.

2.3 The Big O

Many different notations and formulations can be used in the complexity analysis. Among these, Big O (the "O" stands for "order of") is perhaps the simplest and most widely used. It is used to describe the upper bound on growth rate of algorithms as a function of the problem size. Formally, Big O is defined as:

> $f(n) = O(g(n))$ iff there exists positive constants c and n_0 such that $f(n) \leq cg(n)$ for all values of $n \geq n_0$.

It is important to note that the equal sign in the expression $f(n) = O(g(n))$ does not denote mathematical equality. It is therefore incorrect to read $f(n) = O(g(n))$ as $f(n)$ *is equal to* $O(g(n))$ – instead, it is read as $f(n)$ is $O(g(n))$.

As depicted in Figure 2-6, $f(n) = O(g(n))$ basically means that $g(n)$ is a function which, if multiplied by an arbitrary constant, eventually becomes greater than or equal to $f(n)$ and then stays greater than it forever. In other words, $f(n) = O(g(n))$ means that *asymptotically* (as n gets really large), $f(n)$ cannot grow faster than $g(n)$. Hence $g(n)$ is an asymptotic upper bound for $f(n)$.

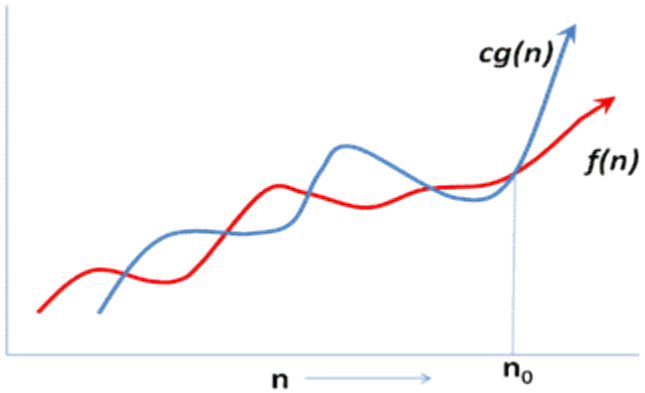

Figure 2- 4 - Big O

Big O is quite handy in algorithm analysis as it highlights the growth rate by allowing the user to simplify the underlying function by filtering out unnecessary details.

For example, if f(n) = 1000n, and g(n) = n^2, then $f(n)$ is $O(g(n))$ because for $n > 100$, and $c = 10$, $cg(n)$ is always greater than $f(n)$.

It can be easily seen that for a given function $f(n)$, its corresponding Big O is not unique - there are infinitely many functions $g(n)$ such that $f(n) = O(g(n))$. For example, $N = O(N)$, $N = O(N^2)$, $N = O(N^3)$, and so on.

We would naturally like to have an estimate which is as close to the actual function as possible. That is, we are interested in the tightest upper bound. For example, although $N = O(N), N = O(N^2),\ N = O(N^3)$ are all correct upper bounds for N, the tightest among these is $O(N)$ and hence that should be chosen to describe the growth-rate of the function N. Note that, choosing N^3 would also be correct but it would be like asking the question: how old is this man, and then getting the response: less than 1000 years. The answer is correct but it is not accurate and hence does not help in problems like comparing the age of two people and determining which one of the two is older. Therefore, the smallest of these functions should be chosen to express the growth rate. So the upper bound for the growth rate of N would be best described as $O(N)$and not by $O(N^2)$ or $O(N^3)$.

For large values of n, the growth rate of a function is dominated by the term with the largest exponent and the lower order terms become insignificant. For example, if we observe the growth of the function $f(n) = n^2 + 500n + 1000$ (Table 2-1), we see that the ratio $\frac{n^2}{f(n)}$ approaches 1 as n becomes really large. That is, $f(n)$ gets closer to n^2 as n becomes larger and other terms become insignificant in the process.

n	$f(n)$	n^2	$500n$	1000	$\frac{n^2}{f(n)}$	$\frac{500n}{f(n)}$	$\frac{1000}{f(n)}$
1	1501	1	500	1000	0.000666	0.333111	0.666223
10	6100	100	5000	1000	0.016393	0.819672	0.163934
100	61000	10000	50000	1000	0.163934	0.819672	0.016393
500	501000	250000	250000	1000	0.499002	0.499002	0.001996
1000	1501000	1000000	500000	1000	0.666223	0.333111	0.000666
5000	27501000	25000000	2500000	1000	0.909058	0.090906	3.64×10^{-5}
10000	105001000	100000000	5000000	1000	0.952372	0.047619	9.52×10^{-6}
100000	10050001000	10000000000	50000000	1000	0.995025	0.004975	9.95×10^{-8}

Table 2-2 - Domination of the largest term

In fact, it can be easily shown that if $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

That is, the Big O of a polynomial can be written by simply choosing the term with the largest exponent in the polynomial and omitting constant factors and lower order terms.

For example, if $f(x) = 10x^3 - 12x^2 + 3x - 20$, then $f(x) = O(x^3)$.

This is an extremely useful property of Big O as, for systems where derivation of the exact model is not possible, it allows us to concentrate on the most dominating term by filtering out the insignificant details (constants and lower order terms). So, if $f(n) = O(g(n))$, then instead of working out the detailed exact derivation of the function $f(n)$, we simply approximate it by $O(g(n))$. Thus Big O gives us a simplified approximation of an inexact function. As can be easily imagined, determining the Big O of a function is a much simpler task than determination of the function itself.

Even with such simplification, Big O is quite useful. It tells us about the scalability of an algorithm. Programs that are expected to process large amounts of data are usually tested for small inputs. The complexity analysis helps us in predicting how well and efficiently our algorithm will handle large real input data.

Since Big O denotes an upper bound for the function, it is used to describe the worst case scenario.

It is also important to note that if $f(n) = O(g(n))$ and $h(n) = O(g(n))$, then it does not follow that $f(n) = h(n)$.

For example, $N^2 + 2N = O(N^2)$ and $3N^2 + 20N + 50 = O(N^2)$, but $N^2 + 2N \neq 3N^2 + 20N + 50$.

This means that we can use Big O to comment about the growth rate of the functions and can use it to choose between two algorithms with different growth rates. However, if we have two algorithms with the same Big O, we cannot tell which one would actually be faster – all we can say is that both of these will grow at the same rate. Hence, Big O cannot be used to choose between such algorithms.

Although developed as a part of pure mathematics, it has been found to be quite useful in computer science and now it is the most widely used notation to describe an asymptotic upper bound of algorithms' need for computational resources (CPU time or memory) as a function of the data size. Omitting the constants and lower order terms makes Big O independent of any underlying architecture, programming language, compiler and operating environment. This enables us to predict the behavior of an algorithm without actually executing it. Therefore, given two different algorithms for the same task, we can analyze and compare their computing needs and decide which of the two algorithms will perform better and use lesser resources. It is also very easy to show that:

1. O(g(n))+O(h(n)) = max(O(g(n)),O(h(n))), and
2. O(g(n)).O(h(n)) = O(g(n).h(n))

Table 2-3 shows the growth for common growth-rate functions found in computer science.

Problem size n	Growth Rate						
	Constant $O(1)$	Logarithmic $O(\log n)$	Linear $O(n)$	Log-Linear $O(n \log n)$	Quadratic $O(n^2)$	Cubic $O(n^3)$	Exponential $O(2^n)$
1	1	1	1	1	1	1	2
2	1	2	2	4	4	8	4
3	1	2	3	6	9	27	8
4	1	3	4	12	16	64	16
5	1	3	5	15	25	125	32
6	1	3	6	18	36	216	64
7	1	3	7	21	49	343	128
8	1	4	8	32	64	512	256
9	1	4	9	36	81	729	512
10	1	4	10	40	100	1000	1,024
20	1	5	20	100	400	8000	1.5×10^6

30	1	5	30	150	900	27000	1.07×10^9
100	1	7	100	700	10000	10^6	1.26×10^{30}
1,000	1	10	1,000	10000	10^6	10^9	1.07×10^{301}
1,000,000	1	20	10^6	20×10^6	10^{12}	10^{18}	too big a number

Table 2-3 - Growth of common growth rate functions

2.4 Determining the Big O for an algorithm – a mini cookbook

Except for a few very tricky situations, Big O of an algorithm can be determined quite easily and, in most cases, the following guidelines would be sufficient for determining it. For complex scenarios a more involved formal mathematical analysis may be required but the following discussion should suffice for the analysis of algorithms presented in this book.

We shall first review some basic mathematical formulae needed for the analysis and then discuss how to find the Big O for algorithms.

2.4.1 Basic mathematical formulae

In the analysis of algorithms, sum of terms of a sequence is of special interest. Among them, arithmetic series, geometric series, sum of squares, and sum of logs are most frequently used. So, the formulae for these are given in the following subsections.

2.4.1.1 Arithmetic sequence

An arithmetic sequence is a sequence in which two consecutive terms differ by a constant value. For example, 1, 3, 5, 7, … is an arithmetic sequence. In general, the sequence is written as:

$$a_1, (a_1 + d), (a_1 + 2d), \dots, (a_1 + (n-2)d), (a_1 + (n-1)d)$$

where a_1 is the first term, d is the difference between two consecutive terms, and n is the number of terms in the sequence.

Sum of n terms of such a sequence is given by the formula:

$$S_n = a_1 + (a_1 + d) + (a_1 + 2d) + \cdots + (a_1 + (n-2)d) + (a_1 + (n-1)d)$$

$$= \frac{n(a_1 + a_n)}{2}$$

Where a_n is the nth term in the sequence and is equal to $(a_1 + (n-1)d)$.

2.4.1.2 Geometric sequence

In a geometric sequence, the two adjacent terms are separated by a fixed ratio called the common ratio. That is, given a number, we can find the next number by multiplying it with a fixed number. For example, 1, 3, 9, 27, 81, … is a geometric sequence where the common ratio is 3. So, if a is the first term in the sequence and the common ratio is r, then the nth term in the sequence will be equal to ar^{n-1} .

Sum of the n terms in the geometric sequence is given by the following formula:

$$S_n = ar^0 + ar^1 + ar^2 + \cdots + ar^{n-1} = \frac{a(r^n - 1)}{r - 1}$$

In computer science the number 2 is of special importance and for our purposes the most important geometric sequence is where the values of a and r are equal to 1 and 2 respectively. This gives us the sequence: $1, 2, 4, \ldots 2^{n-1}$. By putting the values of a and r in the formula, we get the sum of this sequence to be equal to $2^n - 1$. It would perhaps be easier to remember this if you recall that it is the value of the maximum unsigned integer that can be stored in n bits!

2.4.1.3 Sum of logarithms of an arithmetic sequence

Finding sum of logarithms of terms in an arithmetic sequence is also required quite frequently. That is, we need to find the value of

$$\log a_1 + \log(a_1 + d) + \log(a_1 + 2d) + \ldots + \log(a_1 + (n-1)d)$$

Putting the values of both a_1 and d equal to 1 gives us the series: $\log 1 + \log 2 + \cdots + \log n$. We may recall that $\log a + \log b = \log(a.b)$. Therefore

$$\log 1 + \log 2 + \cdots + \log n = \log(1.2.3 \ldots n) = \log n!$$

James Stirling discovered that factorial of a large number n can be approximated by the following formula:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Therefore

$$\log n! \approx \log\left(\sqrt{2\pi n}\ \left(\frac{n}{e}\right)^n\right) = \log\sqrt{2\pi n} + \log\left(\frac{n}{e}\right)^n$$

as $\log a^b = b \log a$, therefore

$$\log\sqrt{2\pi n} + \log\left(\frac{n}{e}\right)^n = \frac{1}{2}(\log 2\pi + \log n) + n\log n - n\log e$$

Since, for large values of n, the term $n \log n$ will dominate the rest of the terms, hence for large values of n,

$$\log 1 + \log 2 + \cdots + \log n = \log(1.2.3 \ldots n) = \log n! \approx n \log n$$

2.4.1.4 Floor and ceiling functions

Let us consider the following question:

> *If 3 people can do a job in 5 hours, how many people would be required to do the job in 4 hours?*

Application of simple arithmetic gives an answer of 3.75 people. As we cannot have people in fraction, we therefore say that we would need 4 people to complete the job in 4 hours.

In computer science, we often face such problems where it is required that the answer is given in integer values. For example, in binary search, the index of the middle element in the array is calculated by the formula: $mid = \frac{low + high}{2}$. So if $high = 10$ and $bot = 1$, mid will be calculated to be equal to 5.5, which is not a valid index. Therefore, in this case, we take 5 as the index of the middle value.

In order to solve such problems, the floor and ceiling functions can be used. Given a real number x, these functions are defined as follows:

1. `floor(x)` (mathematically written as $\lfloor x \rfloor$) is the largest integer not greater than x, and
2. `ceiling(x)` (written as $\lceil x \rceil$) is the smallest integer not less than x.

As can be easily seen, in the first example we used the ceiling function to compute the number of people ($\lceil 3.75 \rceil = 4$), and in the second example we used the floor function for finding the index of the middle element($\lfloor 5.5 \rfloor = 5$).

2.4.1.5 Recurrence relations

A recurrence relation defines the value of an element in a sequence in terms of the values at smaller indices in the sequence. For example, using recurrence, we can define the factorial of a number n as:

$$n! = \begin{cases} 1 & if\ n = 0 \text{ or } 1 \\ n(n-1)! & otherwise \end{cases}$$

That is, factorial of a number n is defined in terms of factorial of $n-1$. In recursive programs, the running time is often described as a recurrence equation. The most commonly used general form of such a recurrence equation for expressing the time requirement for a problem of size n is given as follows:

$$T(n) = \begin{cases} b & for\ n = 1\ (or\ some\ base\ case) \\ aT\big(f(n)\big) + g(n) & otherwise \end{cases}$$

where $f(n) < n$, a and b are constants, and $g(n)$ is a given function.

Taking the case of factorial, since the calculation of factorial of n requires calculation of factorial of $n-1$ and then multiplying it with n, its running time can be given as:

$$T(n) = \begin{cases} b & for\ n = 1\ (the\ base\ case) \\ T(n-1) + \mathrm{c} & otherwise \end{cases}$$

where b and c are some constant.

There are many methods that can be used to solve a recurrence relation. These include repeated substitution, Master theorem, change of variables, guess the answer and prove by induction, and linear homogeneous equations. We will only study repeated substitution as it is perhaps the simplest and most recurrence equations presented in this book can be solved by this method.

In the repeated substitution method we repeatedly substitute smaller values for larger ones until the base case is reached. For example, we can substitute $\mathrm{T}(n-2)+c$ for $T(n-1)$ in the above recurrence equation and get

$$T(n) = (T(n-2)+c)+c = T(n-2)+2c$$

The process is repeated until the base condition is reached. As can be easily seen, by doing that, we get

$$T(n) = b + nc$$

as the solution for the above recurrence relation.

Let us solve some more recurrence equations before concluding this section. For all these examples we will assume that n can be expressed as a power of 2. That is, we can write $n = 2^k$.

Let us now solve the recurrence

$$T(n) = \begin{cases} 1 & for\ n = 1\ (or\ some\ base\ case) \\ 2T\left(\frac{n}{2}\right) + n & otherwise \end{cases}$$

Now

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T\left(\frac{n}{4}\right) + 2n = 2^2T\left(\frac{n}{2^2}\right) + 2n$$

After ith substitution, we get

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + (i-1).$$

As $n = 2^k$, by taking log base 2 of both sides we get $k = \log n$. Therefore, after k substitution we get:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + k.n = nT(1) + n\log n = n + n\log n$$

When n cannot be expressed as a power of 2, we can calculate the value of k (the number of substitutions required to reach the trivial or the base case) by using the appropriate ceiling or floor function.

For example, if n is not a power of 2, the above recurrence relation can be written as

$$T(n) = 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n$$

and

$$k = \lceil\log n\rceil$$

For example, for n = 100, k will be equal to 8. Please remember that we are using log to the base 2 for our calculations.

As already mentioned, in the study of computer science, the number 2 has a special place. Therefore, in this book, unless otherwise stated, we will be using log of base 2. It may be noted that changing the base of log is a trivial exercise and is given by the following law of logarithm:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

For example, to change the base of log from 2 to 10, we simply need to divide it by $\log_{10} 2 \approx 0.30103$.

After reviewing the basic mathematical formulae used in our analyses, we are now ready to learn how to find the Big O of algorithms.

2.4.2 Calculating the Big O of Algorithms

A structured algorithm may involve the following different types of statements:

1. a simple statement
2. sequence of statements
3. block statement
4. selection
5. loop
6. non-recursive subprogram call
7. recursive subprogram call

Let us now discuss the Big O for each one of these.

2.4.2.1 Simple statement and O(1)

For the purpose of this discussion, we define a simple statement as the one that does not have any control flow component (subprogram call, loop, selection, etc.) in it. An assignment is a typical example of a simple statement.

Time taken by a simple statement is considered to be constant. Let us assume that a given simple statement takes k units of time. If we factor out the constant, we would be left with *1*, yielding $k = O(1)$. That is, constant amount of time is denoted by $O(1)$. It may be noted that we are not concerned with the value of the constant; as long as it is constant, we will have $O(1)$. For example, let us assume that we have two algorithmic steps, *a* and *b*, with constant time requirements of 10^{-5} and 10 units respectively. Although *b* consumes a million times more time than *a*, in both these cases we will have the time complexity of *O(1)*.

O(1) means that this algorithmic step (or algorithm) is independent of the data size. At first sight it looks strange – how can an algorithm be independent of the input data size. In fact there are many algorithms which fulfill this criterion and some of these will be discussed in the later chapters. For now, as a very simple example consider the statement:

```
a[i] = 1;
```

In this case, the address of the ith element in the array is calculated by the expression a+i (using pointer arithmetic). We can thus access the ith element of the array in constant amount of time, no matter what the array size is. That is, it will be same for an array of size 10 as well as for an array of size 10000. Hence this operation has a time complexity of O(1).

2.4.2.2 Sequence of simple statements

As stated above, a simple statement takes a constant amount of time. Therefore, time taken by a sequence of such statements will simply be the sum of time taken by individual statements, which will again be a constant. Therefore, the time complexity of a sequence of simple statements will also be *O(1)*.

2.4.3 Sequence of statements

Time taken by a sequence of statements is the sum of time taken by individual statements which is the maximum of these complexities. As an example, consider the following sequence of statements:

statement1	// $O(n^2)$
statement2	// O(1)
statement3	// O(n)
statement4	// O(n)

The time complexity of the sequence will be:

$$O(n^2) + O(1) + O(n) + O(n) = \max\big(O(n^2), O(1), O(n), O(n)\big) = O(n^2)$$

2.4.4 Selection

A selection can have many branches and can be implemented with the help of if or $switch$ statement. In order to determine the time complexity of an if or $switch$ statement, we first independently determine the time complexity of each one of the branches separately. As has already been mentioned, *Big O* provides us growth rate in the worst case scenario. Since, in a selection, each branch is mutually exclusive, the worst case

scenario would be the case of the branch which required the largest amount of computing resources. Hence the *Big O* for an if or a $switch$ statement would be equal to the maximum of the *Big O* of its individual branches. As an example, consider the following code segment:

```
if (cond1)        statement1    // O(n²)
else if (cond2)   statement2    // O(1)
else if (cond3)   statement3    // O(n)
else              statement4    // O(n)
```

The time complexity of this code segment will be:

$$\max\left(O(n^2), O(1), O(n), O(n)\right) = O(n^2)$$

2.4.5 Iteration

In C++ $for, while,$ and $do-while$ statements are used to implement an iterative step or loop. Complexity analysis of iterative statements is usually the most difficult of all the statements. The main task involved in the analysis of an iterative statement is the estimation of the number of iterations of the body of the loop. There are many different scenarios that need separate attention and are discussed below.

2.4.5.1 Simple loops

We define a simple loop as a loop which does not contain another loop inside its body (that is no nested loops). Analysis of a simple loop involves the following steps:

1. Determine the complexity of the code written in the loop body as a function of the problem size. Let it be $O(f(n))$.
2. Determine the number of iterations of the loop as a function of the problem size. Let it be $O(g(n))$.
3. Then the complexity of the loop would be:
   $O\big(f(n)\big).O\big(g(n)\big) = O(f(n).g(n))$

Simple loops come in many different varieties. The most common ones are the counting loops with uniform step size and loops where the step

size is multiplied or divided by a fixed number. These are discussed in the following subsections.

Loops with uniform step size

As stated above, we define a simple loop with uniform step size as the simple loop where the loop control variable is incremented/decremented by a fixed amount. That is, in this case, the values of the loop control variable follow an algebraic sequence. These are perhaps the most commonly occurring loops in our programs. These are usually very simple and have a time complexity of $O(n)$. Their analysis is also quite straight forwards.

The following code for Linear Search elaborates this in more detail.

```
index = -1;
for (i = 0; i < N; i++)
   if (a[i] == key) {
         index = i;
         break;
   }
```

We can easily see that:

1. Time complexity of the code in the body of the loop is $O(1)$
2. In the worst case (when key is not found), there will be n iterations of the for loop, resulting in $O(n)$.
3. Therefore the time complexity of the Linear Search algorithms is:
   $O(1).O(n) = O(n)$.

Simple loops with variable step size

In loops with variable step size, the loop control variable is increased or decreased by a variable amount. In such cases, the loop control variable is usually multiplied or divided by a fixed amount, thus it usually follows a geometric sequence. As an example, let us consider the following code for the Binary Search Algorithm:

```
high = N-1;
low = 0;
index = -1;
while(high >= low)
{
   mid = (high + low)/2;
   if (key == a[mid]) { index = mid; break;}
   else if (key > a[mid]) low = mid + 1;
   else high = mid – 1;
}
```

Once again we can easily see that the code written in the body of the loop has a complexity of $O(1)$. We now need to count the number of iterations.

For the sake of simplicity, let us assume that N is a power of 2. That is, we can write $N = 2^k$ where k is a non-negative integer. Once again, the worst case will occur if the key is not found. In each iteration, the search space spans from *low* to *high*. So, in the first iteration we have N elements in this range. After the first iteration, the range is halved as either the low or the high is moved to mid + 1 or mid -1 respectively, effectively reducing the search space to approximately half the original size. This pattern continues until we either find the key or the condition `high >= low` is false, which is the worst case scenario. This pattern is captured in Table 2-1.

Iteration	1	2	3	.	.	.	K+1
Search Size	N $= 2^k$	N/2 $= 2^{k-1}$	N/4 $= 2^{k-2}$				1 $= 2^{k-k}=2^0$

Table 2-1: The number of iterations in binary search

That is, after k+1 steps the loop will terminate. Since $N = 2^k$, by taking log base 2 of both sides we get $k = \log_2 N$. In general, when N cannot be written as power of 2, the number of iterations will be given by the

ceiling function $\lceil \log_2 N \rceil$. So, in the worst case, the number of iteration will be given by $O(\lceil \log_2 N \rceil)$.

logN).*O1=O(*logN).

A deceptive case

Before concluding this discussion on simple loops, let us look at a last, but quite deceiving, example. The following piece of code prints the table for number *m*.

```
for(int i=1; i<=10; i++)
   cout << m << " X " << i << " = " << m*i << endl;
```

This looks like an algorithm with time complexity $O(N)$. However, since the number of iterations is fixed, hence it is not dependent upon any input data size and therefore it will always take the same amount of time. Therefore, its time complexity is $O(1)$.

2.4.5.2 nested loop

For the purpose of this discussion, we can divide the nested loops in two categories:

1. Independent loops – the number of iterations of the inner loop are independent of any variable controlled in the outer loop.
2. Dependent loops – the number of iterations of the inner loop are dependent upon some variable controlled in the outer loop.

Let us now discuss these, one by one.

Independent loops

Let us consider the following code segment written to add two matrices a, and b and store the result in matrix c.

```
for (int i = 0; i < ROWS; i++)
      for (j = 0; j < COLS; j++)
            c[i][j] = a[i][j] + b[i][j];
```

For each iteration of the outer loop, the inner loop will iterate COLS number of times and is independent of any variable controlled in the outer loop. Since the outer loop iterates ROWS number of times, the statement in the body of the inner loop will be executed ROWS x COLS number of times. Hence the time complexity of the algorithms is $O(ROWS.COLS)$.

In general, in the case of independent nested loops, all we have to do is to determine the time complexity of each one of these loops independent of the other and then multiply them to get the overall time complexity. That is, if we have two independent nested loop with time complexity of O(x) and O(y), the overall time complexity of the algorithm will be O(x).O(y) = O(x.y).

Dependent Loops

When the number of iterations of the inner loop is dependent upon some variable controlled in outer loop, we need to do a little bit more to find out the time complexity of the algorithm. What we need to do is to determine the number of times the body of the inner loop will execute. As an example, consider the following selection sort algorithm:

```
for (i = 0; i < N ; i++) {
      min = i;
      for (j = i; j < N; j++)
            if (a[min] > a[j]) min = j;
      swap(a[i], a[min]);
}
```

As can be clearly seen, the value of j is dependent upon i, and hence the number of iterations of the inner loop depend upon the value of i which is controlled in the outer loop. In order to analyze such cases, we can use a table like the one given in Table 2-2.

Value of i	0	1	2	…	I	…	N – 1
Number of iterations of the inner loop	N	N – 1	N – 2		N – i		1
Total Number of times the body of the inner loop is executed	$N + (N-1) + (N-2) + \cdots + (N-i) + \cdots + 1$ $= \frac{N(N+1)}{2}$ $= O(N^2)$						

Table 2-2: Analysis of Selection sort

As shown in the table, the time complexity of selection sort is thus $O(N^2)$.

Let us look at another interesting example.

```
k = 0;
for (i = 1; i < N; i++)
      for (j = 1; j <= i; j = j * 2)
            k++;
```

Once again, the number of iterations in the inner loop depend upon that value of i which is controlled by the outer loop. Hence it is a case of dependent loops. Let us now use the same technique to find the time complexity of this algorithm. The first thing to note here is that the inner loop follows a geometric progression, going from 1 to i, with 2 as the common ratio between consecutive terms. We have already seen that in such cases the number of iterations is given by $\lceil \log_2(i+1) \rceil$. Using this information, the time complexity is calculated as shown in Figure…

Value of i	1	2	3	4	…	N – 1
Number of iterations of the inner loop	1 $= \lceil \log 2 \rceil$	2 $= \lceil \log 3 \rceil$	2 $= \lceil \log 4 \rceil$	3 $= \lceil \log 5 \rceil$		$\lceil \log N \rceil$
Total Number of times the body of the inner loop is executed	$\log 2 + \log 3 + \log 4 + \log 5 + \cdots + \log N$ $= \log(2.3.4 \ldots N)$ $= \log N!$ $= O(N \log N)$ by Stirling's formula					

Analyses of both the algorithms discussed above are little deceiving. It appears that in both these cases we can get the right answer by simply finding the complexity of each loop independently and then multiplying them just like we did in the case of independent loops. Hence this whole exercise seems to be an over kill. Although this statement is true to the extent that we will indeed get the same answer, however, this is not the right approach and will not always give the right answer in such cases. To understand why, let us consider the following code:

```
for (i = 1; i < N; i = i * 2)
      for (j = 0; j < i; j++)
            k++;
```

Now, in this case, if we calculated the complexity of each loop independently and then multiplied the results, we would get the time complexity as $O(N \log N)$. This is very different from what we get if we applied the approach used in the previous examples. The analysis is shown in Table 2-3

Value of i	1	2	4	8	…	N
Number of iterations of the inner loop	1 $= 2^0$	2 $= 2^1$	4 $= 2^2$	8 $= 2^3$		N $\approx 2^k$ Where $k = \lceil \log N \rceil$
Total Number of times the body of the inner loop will be executed	$2^0 + 2^1 + 2^2 + \cdots + 2^k$ $= 2^{k+1} - 1 = 2.2^k + 1$ $= O(2^k) = O(N)$					

Table 2-3: Analysis of a program involving nested dependent loop

2.4.6 non-recursive subprogram call

A non-recursive subprogram call is simply a statement whose complexity is determined by the complexity of the subprogram. Therefore, we simply determine the complexity of the subprogram separately and then use that value in our complexity derivations. The rest is as usual. For example, let us consider the following program:

```
float sum(float a[], int size)
                  // assuming size > 0
{
      float temp = 0;
      for (int i = 0; i < size; i++)
            temp = temp + a[i];
      return temp;
}
```

It is easy to see that this function has a linear time complexity, that is, $O(N)$.

Let us now use it to determine the time complexity of the following function:

```
float average(float data[], int size)
                  // assuming size > 0
{
      float total, mean, ;
      total = sum(data, size);    //O(N)
      mean = total/size;          //O(1)
      return mean;                //O(1)
}
```

We first determine the time complexity of each statement. Then we determine the complexity of the entire function by simply using the method discussed earlier to determine the complexity of a sequence of statements. This gives us the time complexity of O(n) for the function.

It is important to remember that if we have a function call inside the body of a loop, we need to apply the guidelines for the loops. In that case we need to carefully examine the situation as sometimes the time taken by the function call is dependent upon a variable controlled outside the function and hence gives rise to a situation just like dependent loops discussed above. In such cases, the complexity is determined with the help of techniques similar to the one used in the case of dependent loops. As an example, consider the following example:

```
int foo(int n)
{
      int count = 0;
      for (int j = 0; j < n; j++)
            count++;
      return count;
}
```

As can be easily seen, if considered independently, this function has linear time complexity, that is it is $O(N)$.

Let us now use it inside another function:

```
int bar(int n)
{
      temp = 0;
      for (i = 1; i < n; i = i * 2)
      {
            temp1 = foo(i);
            temp = temp1 + temp;
      }
}
```

If we are not careful, we can be easily deceived by it. If we treat it casually, we can put the complexity of the for loop to be $O(\log N)$ and we have already determined that the complexity of the function foo is $O(N)$. Hence the overall complexity would be calculated to be $O(N \log N)$. However, a more careful analysis would show that $O(N)$ is a tighter upper bound for this function and that should be used to describe the complexity of this function. The detailed analysis of this problem is left as an exercise for the reader.

2.4.6.1 Recursive subprogram calls

Analysis of recursive programs usually involves recurrence relations. We will postpone its discussion for the time being and revisit it in chapter ??.

2.5 Summary and concluding remarks

As mentioned at the start, for most cases, calculation of Big O should be straight forward and the examples and guidelines given above should provide sufficient background for its calculation. There are however complex cases that require more involved analysis but they do not fall within the scope of our discussion.

2.6 Problems:

1. If f(n) = O(g(n)) and h(n) = O(g(n)), which of the following are true:
   (a) f(n) = h(n)
   (b) $\frac{f(n)}{h(n)} = O(1)$
   (c) f(n) + h(n) = O(g(n))
   (d) g(n) = O(f(n))

2. Compute the Big O for the following:
   (a) $log^k n$
   (b) $\log n^k$
   (c) $\sum_{i=1}^{n} i^2$

3. Calculate the time complexity for the following:

(a)

```
k = 0;
for (i = 1; i < N; i = i * 2)
       for (j = 1; j <= i; j = j * 2)
              k++;
```

(b)

```
k = 0;
for (i = 1; i < N; i++)
       for (j = N; j >= i; j = j / 4)
              k++;
```

(c)

```
k = 0;
for (i = 1; i < N; i = i * 3)
       for (j = 1; j <= i; j = j * 4)
              k++;
```

(d)

```
k = 0;
for (i = 1; i < N; i = i * 2)
       for (j = 1; j <= N*N; j = j * 2)
              k++;
```

(e)

```
for(i = 1; i <= n; i++)
       for (j = 1; j <= i; j++)
              for (k = 1; k <= n; k++)
                     m++;
```

(f)

```
for(i = 1; i <= n; i++)
       for (j = 1; j <= i; j++)
              for (k = 1; k <= j; k++)
                     m++;
```

4. Given two strings, write a function in C++ to determine whether the first string is present in the second one. Calculate its time complexity in terms of Big O.

5. Write code in C++ to multiply two matrixes and store the result in the third one and calculate its Big O.
6. calculate the Big O for the following:

```
template <class T>
void swap(T &x, T &y) {
       T t;
       t = x;
       x = y;
       y = t;
}

void bubbleDown(int a[], int n)
{
       for (int i = 0; i < n-1, i++)
       {
              if (a[i] > a[i+1])
                     Swap(a[i], a[i+1]);
       }
}
```

i.

```
void bubbleSort1(int a[], int n) {
       for (int i = 0; i < n; i++)
              bubbleDown(a, n);
}
```

ii.

```
void bubbleSort2(int a[], int n) {
       for (int i = 0; i < n; i++)
              bubbleDown(a, n-i);
}
```

7. Compute the Big O for the following. Each part may require the previous parts.

(a)

```
template <class T>
T min(T x, T y) {
   if (x < y) return x;
   else return y;
}
```

(b)

```
template <class T>
void copyList(T * source, T *target, int size)
{
   for (int i =0; i < size; i++)
          target[i] = source[i];
}
```

(c)

```
template <class T>
void merge(T * list1, int size1, T *list2,
           int size2, T *target) {
   int i = 0, j = 0, k = 0;
   for (; i < size1 && j < size2; k++) {
          if (list1[i] < list2[j])
                { target[k] = list1[i]; i++; }
          else {target[k] = list2[j]; j++; }
   }
   if (i < size1)
      copyList(&list[i], &target[k], size1 - i);
   if (j < size2)
      copyList(&list[j], &target[k], size2 - j);
}
```

(d)

```
template <class T>
void mergeSublists(T * source, T* target,
                   int startIndex,
                   int sublistSize, int maxSize)
{
   T * subList1, * sublist2, *targetSublist;
   int size1 = min(sublistSize, maxSize – startIndex);
   int size2 = min(sublistSize, maxSize – startIndex +
                   sublistSize);
   if (size1 < 1) sublist1 = NULL;
   else sublist1 = &source[startIndex];
   if (size2 < 1) sublist1 = NULL;
   else sublist2 = &source[startIndex + size1];
   targetSublist = &target[startIndex];
   merge(sublist1, size1, sublist2, size2, targetSublist);
}
```

(e)

```
template <class T>
void mergeSort(T *list, int size) {
   T *temp = new T[size];
   T *source = list;
   for (int mergeSize=1; mergeSize < size; mergeSize *= 2)
   {
      for (int i = 0; i < size; i = i + mergeSize*2)
          mergeSublists(source, temp, i, mergeSize, size);
      swap(source, temp);
   }
   if (source == list) copyList(temp, list, size);
   delete []temp;
}
```

8. Carry out the detailed analysis for the bar function given in section 2.4.6.

9. Compute the complexity for the following:

```
template <class T>
int partition(T a[], int left, int right) {
   T pivot;
   if (a[left] > a[right]) swap(a[left], a[right]);
   pivot = a[0];
   int i = left+1, j = right;
   while (i < j) {
          for(; a[i] < pivot; i++);
          for(; j > left && a[j] >= pivot; j--);
          if (i < j)
                swap(a[i], a[j]);
   }
   swap(a[left], a[j]);
   return j;
}
```