Here $Time = (K1 + n * K2) \propto n$.

where $K1$ and $K2$ are two constants, $K1$ is the time taken for the assignment $T = 0$, and $K2$ is the time taken for the assignment $T = T + A(i, i)$. Thus, the algorithm takes linear time.

EXAMPLE 1.4: Maximum element of a matrix (square matrix).

```
Procedure Maxelement (A,n)
  Array A(n)
  Max = - ∞ /* initialization */
  For i = 1 to n Step 1 Do
    For j = 1 to n Step 1 Do
      If(A(i,j) > Max) Then Max = A(i,j);
    Endfor /* j */
  Endfor /* i */
End Maxelement
```

Here, the time required is of the form $(k_1 + k_2 * n * n) \propto n^2$, where k_1 and k_2 are constants; k_1 is the time for the assignment $Max = -\infty$ and k_2 is the time taken by the operations inside the loop. Thus, the algorithm takes quadratic time.

EXAMPLE 1.5: Product of two matrices A and B.

```
Procedure Matmul(A,B,n)
  Array A(n), B(n), C(n)
  For i = 1 to n Step 1
    For j = 1 to n Step 1
      C(i,j) = 0;     /* initialization */
      For k = 1 to n in Step 1
      C(i,j) = C(i,j) + A(i,k) * B(k,j);
      Endfor /* k */
    Endfor /* j */
  Endfor /* i */
End Matmul
```

Here the time required is of the form:

$\{(k_3 * n + k_2) * n\} * n$, where k_2 and k_3 are constants. This expression is proportional to n^3. Thus, the algorithm takes cubic time.

1.1 UPPER BOUND OF POLYNOMIAL FORM OF TIME COMPLEXITY

Let x be the size of data (according to the problem) and let the algorithm take time of the form:

$$T = A_n x^n + A_{n-1} x^{n-1} + \dots + A_0$$

$$\Rightarrow T \le |A_n|\, x^n + |A_{n-1}|\, x^{n-1} + |A_{n-2}|\, x^{n-2} + |A_0|$$

$$= \left(|A_n| + \frac{|A_{n-1}|}{x} + \frac{|A_{n-2}|}{x^2} + \dots + \frac{|A_0|}{x^n} \right) x^n$$

$$\le (|A_n| + |A_{n-1}| + |A_{n-2}| + \dots + |A_0|)x^n \quad (\because\ x \ge 1)$$

$$= Kx^n,\ K = (|A_n| + |A_{n-1}| + |A_{n-2}| + \dots + |A_0|)$$

That is, Kx^n bounds the time from the above. We say that the time requirement is of the order of x^n. For this, we use the '*Big Oh*' notation and write $T \in O(x^n)$.

Big O: $f(n)$ is said to be $O(g(n))$ iff there exist two constants c and n_o such that $f(n) \le c*g(n),\ \forall\ n \ge n_o$.

Let $f(n) = 4n^2 + 3n,\ g(n) = 2n^3$.

$g(n) > f(n),\ \forall\ n \ge 3 \Rightarrow f(n) \sim O(g(n)) \Rightarrow f(n) \sim O(n^3)$.

Algorithms taking constant time are said to be of $O(1)$. The order of dominance of some common time complexities is:

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(k^n)$, k is a constant.

Table 1.1 shows how a few of the common functions grow with the increase in argument values. We note that the growth rates of $T5$ and $T6$ are very very fast compared to the others. After some small values of N, these two functions become unmanageably large.

TABLE 1.1 Growth of functions

N	$T1 = N$	$T2 = N \log_2 N$	$T3 = N^2$	$T4 = N^3$	$T5 = 2^N$	$T6 = N!$
1	1	0	1	1	2	1
10	10	33.21928095	100	1000	1024	3628800
20	20	86.4385619	400	8000	1048576	2.4329E+18
30	30	147.2067179	900	27000	1073741824	2.65253E+32
40	40	212.8771238	1600	64000	1.09951E+12	8.15915E+47
50	50	282.1928095	2500	125000	1.1259E+15	3.04141E+64
60	60	354.4134357	3600	216000	1.15292E+18	8.32099E+81
70	70	429.0498112	4900	343000	1.18059E+21	1.1979E+100
80	80	505.7542476	6400	512000	1.20893E+24	7.1569E+118
90	90	584.2667787	8100	729000	1.23794E+27	1.4857E+138
100	100	664.385619	10000	1000000	1.26765E+30	9.3326E+157
110	110	745.9495685	12100	1331000	1.29807E+33	1.5882E+178
120	120	828.8268715	14400	1728000	1.32923E+36	6.6895E+198
130	130	912.9078157	16900	2197000	1.36113E+39	6.4669E+219
140	140	998.0996224	19600	2744000	1.3938E+42	1.3462E+241
150	150	1084.322804	22500	3375000	1.42725E+45	5.7134E+262

Problems whose best known algorithms require exponential (k^n) time or more, where k is a constant, is also known as *hard* or *intractable* problems.

There are other asymptotic notations like *Big Oh*. Some of these are:

Ω-notation: It deals with the minimum time (best cases) required by the algorithm. $f(n)$ is said to be $\Omega(g(n))$ iff there exist positive constants c and n_o such that $|f(n)| \geq c * |g(n)|$, $\forall\, n \geq n_o$.

Ω gives the lower bound while O gives the upper bound of. time required by the algorithm. We say an algorithm to be *optimal* if $f(n)$ is $O(g(n))$ as well as $f(n)$ is Ω $(g(n))$.

Θ-notation: $f(n)$ is $\Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_o such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$, $\forall\ n \geq n_o$. $\Theta(g(n))$ deals with the optimum time.

o-notation (small o): $f(n)$ *is* $o(g(n))$ iff:

$$\text{Limit}_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

Now we consider some more examples for evaluating time complexities.

EXAMPLE 1.6: Suppose we have a programme having outline as given below. Let one time execution of the statements between the i-loop and the j-loop require K_1 units of time and those within the j-loop require K_2 units of time.

```
Do 10 i = 1 to N
  ...
  K1
  ...
Do 10 j = 1 to i
  K2
  ...
10 Continue
```

The time-complexity of the above algorithm may be expressed as:

$$T(N) = K_1 * N + K_2 * 1 + K_2 * 2 + K_2 * 3 + \ldots + K_2 * N$$

$$= K_1N + K_2(1 + 2 + 3 + \ldots + N)$$

$$= K_1N + K_2N(N + 1)/2$$

So, $T(N)$ is $O(N^2)$.

EXAMPLE 1.7: Suppose we have a programme having outline as given below. Let one time execution of the statements between the i-loop and the j-loop require K_1 units of time and those within the j- and l-loops require K_2 units of time.

```
Do 10 i = 1 to N
  ...
  K1
  ...
Do 10 j = 1 to i
Do 10 l = 1 to i
  ...
  K2
  ...
10 Continue
```

The time complexity for the above algorithm may be written as:

$$
\begin{aligned}
T(N) &= K_1N + K_2 1^2 + K_2 2^2 + \dots + K_2N^2 \\
&= K_1N + K_2(1^2 + 2^2 + \dots + N^2) \\
&= K_1N + K_2 \frac{N(N+1)(2N+1)}{6}
\end{aligned}
$$

So, $T(N)$ is $O(N^3)$.

EXAMPLE 1.8: Suppose we have a programme having the outline given below. Let one time execution of the statements between the i- and j-loops require K_1 units of time and those within the j-, l- and m-loops require K_2 units of time.

```
Do 10 i = 1 to N
  ...
  K1
  ...
Do 10 j = 1 to i
Do 10 l = 1 to i
Do 10 m = 1 to i
  ...
  K2
  ...
10 Continue
```

The time complexity of the above algorithm may be written as:

$$
\begin{aligned}
T(N) &= K_1N + K_2(1^3 + 2^2 + 3^3 + \dots + N^3) \\
&= K_1N + K_2\left(\frac{N(N+1)}{2}\right)^2
\end{aligned}
$$

So, $T(N)$ is $O(N^4)$.

Some of the useful relations involving O-notation that help in finding time complexities are:

- $f(n) \sim O(f(n))$, that is, $f(n)$ is dominated by its own time.
- $C * O(f(n)) = O(f(n))$, that is, constants are absorbed in O-notation.