Software development

- ***Understand and define the problem***
  - what is it that needs to be solved

- ***Identify an approach***
  - the basic equations, relations, etc

- ***Develop and specify an algorithm***
  - Sequence of steps to be executed
  - Specify how data is organized

- ***Evaluate the algorithm***
  - always provide correct results ?
    is it efficient?
- ***Coding***
  - Develop functions for input/output or processing the data

  - **Test and validate**
    - Debug program
      Run various test cases
    - confirm -- gives expected results?
    - confirm efficiency

- ***Documentation***
  - Problem statement,
  - solution process ,
  - algorithm,
  - program code
  - testing procedure

- ***Program maintenance***
  - Make necessary changes if errors are found
  - Ensure that it meets user changing requirements

Algorithms for problem solving

ALGORITHMS

An algorithm is a detailed sequence of actions to perform to accomplish some task. The name comes from an Iranian mathematician *Al-Khawarizmi*.

Other definitions:
– A finite set of step-by-step instructions for a problem-solving or computation procedure, especially one that can be implemented by a computer.
- A mathematical procedure that can usually be explicitly encoded in a set of computer language instructions that manipulate data.
- An ordered sequence of well defined and effective operations which, when executed, will terminate in finite amount of time and produce correct result.

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.

Properties of good algorithms:

Precision:
- Each step must be clear and unambiguous.
- each step must be finite
- Number of steps must be finite

Simplicity:

- Each step must be simple enough to be easily understood
- Should translate to a few lines of code

Levels of Abstraction:

-steps should be grouped into related modules
-other algorithms are called by names instead of complete code
-details should be hidden at higher levels of abstraction
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 smallest divisor of a number

The smallest divisor of a number N is the smallest value which divides N exactly without leaving a remainder. The simplest way to find this would be to try dividing by 2,3,4,5,6,….N and stop when the division does not leave any remainder.

Algorithm A

1. Let divisor be 1.
2. while divisor < number keep on doing the following:

   2a. divisor = divisor +1.
   2b. remainder = number mod divisor.
   2c. if remainder = 0 go to step 3 else go to step 2

3. if divisor = number, smallest divisor = 1, else smallest divisor = divisor
4. print smallest divisor.

Algorithm B

1. If number is even, smallest divisor =2, Go to step 7.
2. Let divisor be 3.
3. remainder = number mod divisor
4. if remainder=0, smallest divisor = 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, smallest divisor = 1, else smallest divisor = divisor
7. print smallest divisor.

Algorithm C

1. If number is even, smallest divisor =2, Go to step 8.
2. If number is divisible by 3, smallest divisor = 3, Go to step 8.
3. Let divisor be 3.
4. Find factor = number/divisor (get nearest integer value)
5. remainder = number mod 3
6 As long as remainder is not zero & divisor < factor
   keep on doing the following:
      6a. divisor = divisor +2.
      6b. remainder = number mod divisor.
      6c. factor = number / divisor
7. if number mod divisor is zero, smallest divisor = divisor
   Else, smallest divisor = 1.
8. print smallest divisor.

Compare the number of operations involved in each of the algorithms.

Finding largest sequence of Ones

Consider a two dimensional array. Each row consists of sequence of Ones, followed by sequence of Zeros. The problem is to find the maximum number of Ones in any row.

```
11111100000000000000000
11111111111100000000000
11111000000000000000000
11111111111111111111000
11110000000000000000000
11111111111110000000000
11000000000000000000000
```

Algorithm 1:

Let any element of this 2 dimensional array of rows x cols be a(j,k)

```
max = 0;
for j = 1 to rows {
     find total number of Ones;
     if total >max
          set max = total;
}
```

How many operations involved?
= cols x rows

Algorithm 2:

Find the position of last 1 in first row.
Start examining second row from that position onward.
Get the position of last 1.
Start next row from that position.
If that position contains zero, skip that row.
Keep a record of row where change was made.
Stop when all rows examined.

How many operations now? (number of cols+ no. of rows).

Performance analysis of algorithms

Measuring Time Complexity

Counting number of operations involved in the algorithms 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;                                   ……(1)
     for (j = 0; j < m; j++)
          b= b + j;                            ……(2)
}
```

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];……(3)
          total = total + matrix[k][j];    ……(4)
     }
}
```

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];……(5)
     total = total + rows[k];            ……(6)
}
```

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 one comparison is carried out, and 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 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 && array[i] != target)
      i = i + 1;
if (i <MAX)
     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 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.

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.

Complexity of a sorting algorithm:
Selection Sort

A 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 (smallest to largest), it examines each element in the list, finds the smallest element and swaps it with the first element in the array. Then it examines the array starting with the second element, finds 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

Sorted array after pass 1: **19** 25 37 58 95 56 73 30

Algorithm for selection sort:

1. **for lh taking on values 0 to n-1 do steps 2-5.**
2. **set rh = lh as starting value of the new array.**
3. **Examine all elements of new array with index j by doing step 4.**
4. **If the element at j < element at rh, then set rh=j.(Find the smallest element).**
5. **swap element at index lh with element at rh.(smallest element stored in beginning of new array)**

Implementation:

```
void selectionsort(int array[],int n)
{
     int lh, rh, j, temp;
     for(lh=0; lh<n; lh++)  {
          rh = lh;
          for(j=lh+1; j<n; j++)
               if(array[j]<array[rh])   rh = j;

          temp = array[lh];
          array[lh]=array[rh];
          array[rh]=temp;
     }
}
```

Analyzing performance of selection sort

In the first cycle, the inner loop examines n elements, so it takes n operations. During the 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 T 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. Thus we can say that the selection sort algorithm is of order n^2.

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.

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

$O(n^4 + 3_1 n^2 + 10) = O(n^{4)}$

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 a *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?

Consider a specific case.

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

Let us assume $f(n) = O(n^2)$

So as per definition of Big-O, we must have

$f(n) <= c\ g(n)$

or $2n^2 + 3n + 1 <= cn^2$

To determine range of values for c, we divide both sides by n^2

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

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 values 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 was there, the net weight would be proportional to the weight of an elephant.

So here the first term $2n^2$ dominates the other terms, which can be neglected in comparison with the first term, and we can say

$f(n) = O(2n^2)$

The constant term has no significance, and the complexity can be finally expressed as

f(n) = O(n^2)

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 ?)

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

$O(n^4)$

Complexity of the Linear search Algorithm in terms of Big O:

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 2:

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

```
for (k=1; k<=n/2; k++)
{
     sum = sum + 5;

}
for (j = 1; j <= n*n; j++)
{
     delta = delta +1;
     .
}
```

The number of operations executed by these loops is the sum of the individual loop operations.

The complexity of total operations $n/2+n^2$ would be $O(n^2)$ in big-O terms.

Thus, for two loops with $O[f_1(n)]$ and $O[f_2(n)]$ complexities, the overall complexity of sequencing the two loops is $O[f_D(n)]$ where $f_D(n)$ is the dominant term corresponding to the functions $f_1(n)$ and $f_2(n)$.

Time Complexity meaningful for large value 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 what the function is exactly but a description of how the function grows. Concretely, consider the two functions:

f(n) = n + 20

g(n) = n – 10

Which function grows faster?

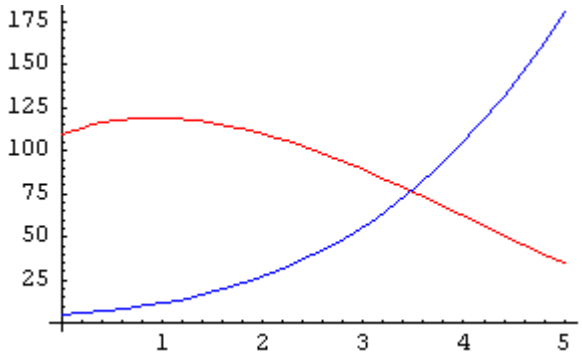
Try with small values of n say up to 100. It would show that f(n) grows faster. Now try values between 1000 and 10,000. What does it show? There is hardly any noticeable difference between values of f(n) and g(n). Both grow linearly with n. Thus both are of order n.

Let us now consider another pair of functions

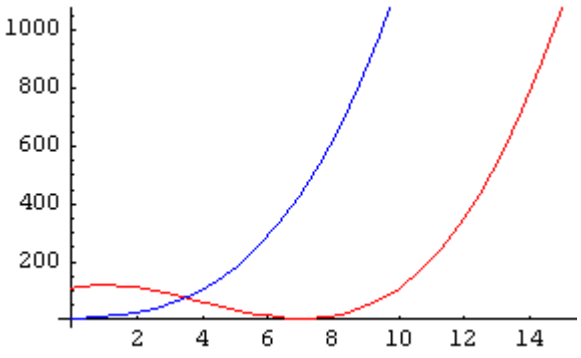
$$f(n) = n^3 - 12n^2 + 20n + 110$$

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

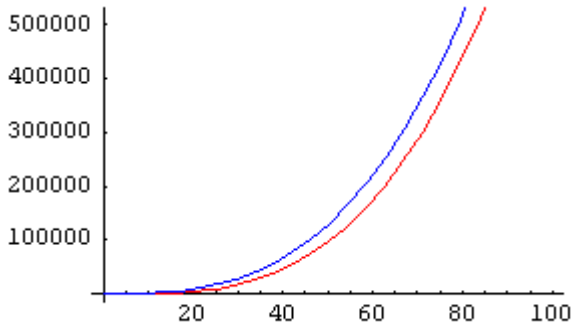
They look quite different, but how do they behave for different values of n? Let's look at a few plots of the function (f(n) is in red color , and g(n) is in blue color):

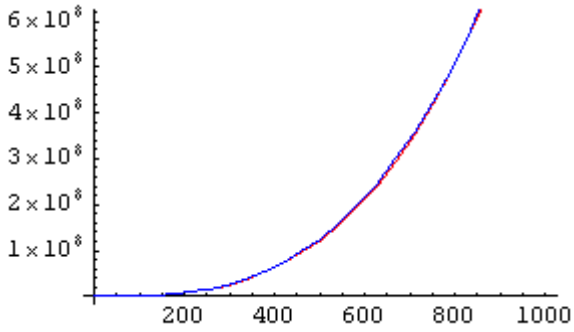


Plot of f and g, in range 0 to 5



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 graph, drawn for very limited values of n, the curves appear somewhat different. In the second graph, they sort of start moving somewhat similar, in the third graph for appreciable values of n, there is only a very small difference between f(n) and g(n), and in the last graph drawn for very large values of n, the plots 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 the big-O notation for this. We say that:

$$f(n) = n^3 - 12n^2 + 20n + 110$$

$$= O(n^3)$$

Or in other words, f(n) is of order n^3 .

Order notations used in computer science

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 complexity can be expressed in terms of a polynomial of the form

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

are called ***polynomial algorithms*** and are of **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 $10^3 = 1000$
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 invariably very efficient.

It can also be shown that the choice of base of the logarithm is immaterial. It does not matter as to what base value is chosen.

Thus $log_3 n$ can also be expressed as **O(ln n).**

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 $log_2 n$) .**

Remember that **O(n $log_2 n$)** will be much less than **O(n^2).**

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.

Evaluating 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.

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.

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

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

$$\Rightarrow \quad x = 42 \text{ ms}$$

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

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

        sum = sum + j ;

    }
```

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

$$\text{Number of operations} = \sum_{j=1}^{n} 1$$

What is the value of sum when the loop is finished? 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 this as a summation

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

What is the value of sum when the loop is finished?

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

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;

   }
}
```

Two operations are carried out each time the "k" loop is executed. The k loop runs for "j" times.

"j" loop runs N times

For N = 1 , j loop runs 1 time and k loop runs 1 time

For N = 2 , j loop runs 2 times and k loop runs 1+2 times

For N = 3 , j loop runs 3 times and k loop runs 1+2+3 times

For N = 4 , j loop runs 4 times and k loop runs 1+2+3+4 times

For any value of N, the number of operations are

$2(1 + 2 + 3 + 4 + \ldots\ldots\ldots + N)$

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

$= N^2 + N$

Thus the order of complexity is $O(N^2)$

We can also express this in summation form as

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

7. Consider the following code segment:

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

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

grandTotal = grandTotal + rows[k];
}
```

How many addition operations are involved in this case?

$$\sum_{k=1}^{n} (1 + \sum_{j=1}^{n} 1)$$

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

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

   for(k=1; k <= j; k++) {
   for(m=1; m <= j; m++) {

       sum = sum + a(m,k);

   }
}
```

Where a(m,k) is some combination of the running variables m and k. Let us say computation of the sum each time in the innermost loop involves 4 steps. Since the two nested loops involve j^2 operations, the overall complexity of the algorithm can be expressed as the summation

$$\sum_{j=1}^{n} 4j^2$$

Evaluating Summations

To work out the total number of operations in a program involving simple variables, arrays or linked lists, and being computed within a set of loops, find the number of times the loop is getting executed and the number of operations within the loop.

The number of operations can be found with the help of summations. There can be a single loop or nested loops. Start with the innermost loop and work outwards.

Let us consider various cases below:

Case 1: Summation of constant values

Consider a single loop with a single operation. The summation with the loop variable moving from 1 to n would be given by

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

$= 1 + 1 + 1 + \ldots + 1$ (n times)

$= n$

Consider another summation with 4 operations:

$$\sum_{k=1}^{n} 4$$

The constant term can be taken out of the summation.

$$= 4 \sum_{k=1}^{n} 1$$

$= 4n$

Here is another example, where the constant term n can be taken out of the summation;

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

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

$= n^2$

Take care if the loop starts from 0 instead of starting from 1. You would need to add one more term to the summation.

$$\sum_{k=0}^{n} 4$$

$$= 4 (n+1)$$

Now let us consider a summation, where the lower limit is different from 1

$$\sum_{k=5}^{10} 4$$

This can be broken down into two summations

$$\sum_{k=1}^{10} 4 \quad - \quad \sum_{k=1}^{4} 4$$

$$= 4(10) - 4(4)$$

$$=24$$

Case 2: Summation involving the loop variable

When the number of operations depends on the loop variable, then the summation can take the following form

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

$$= 1 + 2 + 3 + 4 + \ldots\ldots + n$$

This is a well known mathematical series whose sum is expressed through the formulae:

$$\mathbf{n(n+ 1)/ 2}$$

It can be shown by mathematical induction that this proposition is true.

Step 1

Prove the base case, for n = 1

Sum = 1(1+1) /2 = 1, which is true.

Step 2

Prove the inductive case, i.e. show that it is true for n+1 as well.
The goal is to show that

1 + 2 + 3 + 4 +……..+ n + (n+1) = (n+1)(n+2)/2
Since we already know the summation till nth term, the left hand side can be written as

n(n+1)/2 + (n+1)

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

= $(n^2 + 3n + 2) /2$

= (n+1) (n+2) /2

This completes the proof.

Case 3: Summation involving the loop variable & a constant

Now consider summation of a constant and a loop variable term

$$\sum_{j=1}^{10} (4 + j)$$

This summation can be split into two summations and evaluated separately

$$\sum_{j=1}^{10} 4 + \sum_{j=1}^{10} j$$

= 4 (10) + (10) (11)/2

= 40 + 55

What happens when the lower limit is not 1? In the following example the lower limit is zero.

$$\sum_{j=0}^{10} j$$

$$= 10\ (11)/\ 2$$

which is same as the summation with lower limit as 1, since the first term is zero and effectively the summation is being carried out for j values from 1 to 10.

When the lower limit is different from 1 or 0, such as in the following summation, we split the summation in two parts

$$\sum_{j=5}^{10} j$$

$$= \sum_{j=1}^{10} j \; - \; \sum_{j=1}^{4} j$$

$$= 10\ (11) / 2 \ - \ 4\ (5) / 2$$

$$= 45$$

Case 4: Summation involving a different variable

Let us now consider the summation where the loop variable is different from the variable inside the summation, as in

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

Since the variable k is not the loop variable j, it can be treated as a constant and can be moved out of the summation, leaving a 1 inside.

$$= 10\ k$$

Case 5: Double Summations

When you have two nested loops, you would get a summation inside another summation. These are solved by working out the summations right to left. Consider for example,

$$\sum_{k=1}^{10} \sum_{j=1}^{6} j$$

The summation on the right is a case of a loop variable, and this allows us to reduce the problem to

$$\sum_{k=1}^{10} 6(7)/2$$

Now, it is a simple case of summation over a constant, which can be worked out as follows:

$$\sum_{k=1}^{10} 21$$

$$= 21 \sum_{k=1}^{10} 1$$

$$= 210$$

Case 6: Summation involving square of the loop variable

Finally, note that the summation of square of the loop variable is given by the series expression

$$\sum_{k=1}^{n} k^2$$

$$= n(n+1)(2n+1)/6$$