2. ARRAY

2.1 Introduction

An array is linear collection of similar elements or data items or objects. The elements are stored consecutively in memory. Though array is simple linear data structure is has some advantages and disadvantages in using it.

2.1.1 Advantages

1. Simple and linear representation in memory.
2. Accessing (read, write, modify) is easier.

2.1.2 Disadvantages

1. Because of static memory allocation, memory may get wasted if not utilized fully.
2. The operation like deletion will cause the array elements to move to the left.
3. The operation like insertion will cause the array elements to move right.

An array as a data structure is defined as a set of pairs (*index, value*) such that with each index, a value is associated.
index—indicates the location of an element in an array
value—indicates the actual value of that data element

In general, one can define an array as a data structure as given below:
Type <array name> [Size of Array];
Where,
Type is the type of value you want to store in an array
<array name> is name of an array where you store the data.
[Size of Array] is the maximum number of elements you can store in an array.

Example:
INTEGER A[10];
This will create an array name ‘A’ of maximum size 10. The memory representation for this creation will be as shown below.

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
Index	1	2	3	4	5	6	7	8	9	10
Value	12	22	34	45	50	56	60	72	87	55
	2001	2005	2009	2013	2017	2021	2025	2029	2033	2037

The memory allocation for the array ‘A’ will be sequential and if it is assumed that an integer takes 4 bytes in memory and if the allocation starts from the fictious address 2001 then:

The first element will be given 4 bytes chunk at address 2001.
The second element will be given 4 bytes from address 2005 and so on, and last element will be at location 2035.

Mathematically, the array indexing will start from index 1 and ends with index value 10. This defines the lower index (Li) and higher index (Hi) of an array.
Thus the Li=1 and Hi=N

In almost all well known programming languages like C, C++, JAVA, etc. the indexing starts from '0' and ends with N-1.

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
Index	0	1	2	3	4	5	6	7	8	9
Value	12	22	34	45	50	56	60	72	87	55
	2001	2005	2009	2013	2017	2021	2025	2029	2033	2037

Thus, the Li=0 and Hi=N-1.

In the array, the first element's address is called as Base Address of an array. Base Address is an address from where the array starts storing its elements.

Any element of an array can be accessed as <array name>[index]. For instance, first element of an array 'A' can accessed as A[1], the second as A[2] and so on till A[10] if the lower index starts with '1'.

But if the lower index starts with '0' then the first element will be A[0], the second will be A[1] and so on till A[9] as a last element'

Thus, the address of first element and the Base Address is one and the same. That is;
Address of first element = BASE ADDRESS.
In the array shown above the address of first element A[0] = Base Address= 2001

The array can be represented as one dimensional array, 1-D array as shown in the above example or two-dimensional array, 2-D array and so on till N-dimensional array, N-D array.

2.2 Array as an Abstract Data Type

An array can be represented as an abstraction with it's ADT as give below. The various Operations on an array are given in section '**declare**' of an ADT.

structure *ARRAY*(*value, index*)

declare

CREATE( ) →*array*

This operation creates an array named "*array*"

RETRIEVE(*array, index*) → *value*

This operation retrieves/reads a *value* at *index* from an *array* and returns that *value*

STORE(*array, index, value*) →*array;*

This operation stores a *value* at *index* of an *array* and returns an updated *array*.

for all *A* € *array, i, j* € *index, x* € *value* **let**
RETRIEVE(*CREATE, i*) :: = **error**
This operation returns an **error** if one tries to retrieve or read something at *index* '*i*' from the array which is yet to be CREATED.

RETRIEVE(*STORE*(*A, i, x*),*j*) :: = **if** *EQUAL*(*i*,*j*) **then** *x* **else** *RETRIEVE*(*A*,*j*)
This operation is nested operation where first and element 'x' is stored at index 'i' of an array 'A'. [STORE(A, i, x)]
Then from this modified array 'A', a value is retrieved at index 'j'. if the index 'i' at which value 'x' was stored is same as index 'j' then this new value 'x' is return or some different value will be return. [*RETRIEVES*(A, j)]

end
end *ARRAY*

Merging of two arrays

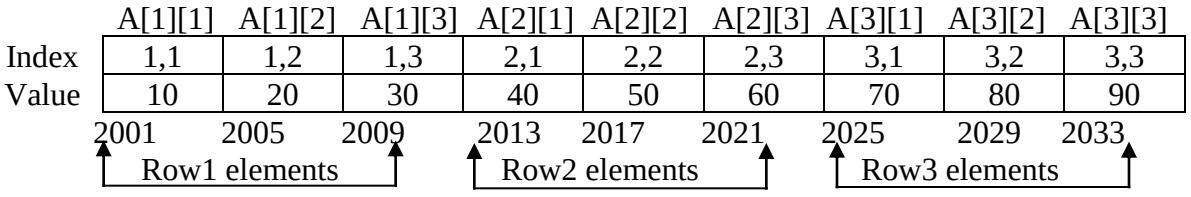
2.3 Multidimensional Arrays

Two-dimensional arrays or N-dimensional arrays, where 'N' can be any finite inter, are stored as matrix of rows and columns in the memory. The representation of rows and columns is sequential in the memory but the internal interpretation by computer languages is in the form of rows and columns.

The 2-D array can be defined as:
Type <array name> [Row Size][Column Size];
Where,
Type is the type of value you want to store in an array
<array name> is name of an array where you store the data.
[Row Size] is the maximum number of rows you can store in an array.
[Column Size] is the maximum number of columns you can store in an array.

Example:
Integer A[3][3];

This will create an array name 'A' of maximum row size 3 and column size also 3. The memory representation for this creation will be as shown below.



The mathematical interpretation of an above array is like a matrix of rows and columns as shown below. The elements in the array are viewed as matrix of order 3 X 3 with total 9 elements stored in row and column fashion. In general, the 2-D array has m X n number of elements if it has m-rows and n-columns.

		Col1	Col2	Col3
		1	2	3
Row1	1	1, 1	1, 2	1, 3
Row2	2	2, 1	2, 2	2, 3
Row3	3	2, 1	2, 2	2, 3

		Col1	Col2	Col3
		1	2	3
Row1	1	10	20	30
Row2	2	40	50	60
Row3	3	70	80	90

For programming languages like C, C++, JAVA, etc. the same 2-D will look like:

		Col1	Col2	Col3
		0	1	2
Row1	0	10	20	30
Row2	1	40	50	60
Row3	2	70	80	90

The actual memory representation of above array will be:

	A[0][0]	A[0][1]	A[0][2]	A[2][1]	A[1][1]	A[1][2]	A[2][0]	A[2][1]	A[2][2]
Index	0,0	0,1	0,2	1,0	1,1	1,2	3,1	3,2	3,3
Value	10	20	30	40	50	60	70	80	90
	2001	2005	2009	2013	2017	2021	2025	2029	2033

Row1 elements Row2 elements Row3 elements

In the array, the first element's address is called as Base Address of an array. Base Address is an address from where the array starts storing its elements.

Any element of an array can be accessed using notation of <array name>[row index] [column index]. For instance, first element of an array 'A' can accessed as A[1][1], the second as A[1][2] and so on till A[3][3] if the lower index of a row and column start with '1'.

But if the lower index of row and column start with '0' then the first element will be A[0][0], the second will be A[0][1] and so on till A[3][3] as a last element'

Thus, the address of first element and the Base Address is one and the same. That is;
Address of first element = BASE ADDRESS.
In the array shown above the address of first element A[0][0] = Base Address= 2001

2.4 Storage Representation and their Address Calculation

The 2-D or the N-D array is stored in computer memory and two different forms, mainly Row major and Column Major.

In row major format, the mathematically interpreted and represented 2-D array will be stored in memory by transferring the rows of the given matrix to the physical memory as shown below.

		Col1	Col2	Col3
		0	1	2
Row1	0	10	20	30
Row2	1	40	50	60
Row3	2	70	80	90

	A[0][0]	A[0][1]	A[0][2]	A[2][1]	A[1][1]	A[1][2]	A[2][0]	A[2][1]	A[2][2]
Index	0,0	0,1	0,2	1,0	1,1	1,2	3,1	3,2	3,3
Value	10	20	30	40	50	60	70	80	90
	2001	2005	2009	2013	2017	2021	2025	2029	2033

Row1 elements | Row2 elements | Row3 elements

In column major format, the mathematically interpreted and represented 2-D array will be stored in memory by transferring the columns of the given matrix to the physical memory as shown below.

		Col1	Col2	Col3
		0	1	2
Row1	0	10	20	30
Row2	1	40	50	60
Row3	2	70	80	90

	A[0][0]	A[0][1]	A[0][2]	A[2][1]	A[1][1]	A[1][2]	A[2][0]	A[2][1]	A[2][2]
Index	0,0	0,1	0,2	1,0	1,1	1,2	3,1	3,2	3,3
Value	10	40	70	20	50	80	30	60	90
	2001	2005	2009	2013	2017	2021	2025	2029	2033

Column 1 elements | Column2 elements | Column3 elements

The access of elements to the user will same for row major and column major format will be same.

The address calculation of each of the elements in 1-D array and 2-D array will be done based on the structure of an array.

The ***address calculation of 1-D array*** will be done using following formula.

Address of an i^{th} index element =
Base Address (BA) of an array + (i^{th} index of the element -1) * size of an element in bytes

The above formula works for an array with lower index value starts with '1'.

For an array with lower index value '0', the formula of address calculation will be:
Address of an i^{th} index element =
Base Address (BA) of an array + i^{th} index of the element * size of an element in bytes

Let now apply the above formula of address calculation for any element in the given array below.

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
Index	1	2	3	4	5	6	7	8	9	10
Value	12	22	34	45	50	56	60	72	87	55
	2001	2005	2009	2013	2017	2021	2025	2029	2033	2037

Let us calculate the address of 1^{st} element, A[1], assuming the Lower index of an array is 1.
Here, i =1
BA=2001
Size of an element=4 bytes

Address of A[1] = BA + (1-1)*4 = 2001 + 0 *4 = 2001+0 = 2001

For 3^{rd} element,
i =3
BA=2001
Size of an element=4 bytes

Address of A[3] = BA + (3-1) * 4 = 2001+ 2*4 = 2001+8 = 2009

Similarly, address of A[10] = BA + (10-1) * 4 = 2001 + 9*4 = 2001 + 36 = 2037

The ***address calculation of 2-D array*** will be done using following formula for ***row major representation***. Here, it is assumed that 2-D array has 'm' rows and 'n' columns.

Address of an i^{th} row and j^{th} column index element =
Base Address (BA) of an array + {(i^{th} row index of the element -1 * n) +(j^{th} row index of the element -1)}* size of an element in bytes

For ***column major representation***, the formulas of address calculation will be:

Address of an i^{th} row and j^{th} column index element =
Base Address (BA) of an array + {(i^{th} row index of the element -1) +(j^{th} row index of the element -1 * m)}* size of an element in bytes

Let now apply the above formula of address calculation for any element in the given array below.

	A[1][1]	A[1][2]	A[1][3]	A[2][1]	A[2][2]	A[2][3]	A[3][1]	A[3][2]	A[3][3]
Index	1,1	1,2	1,3	2,1	2,2	2,3	3,1	3,2	3,3
Value	10	20	30	40	50	60	70	80	90
	2001	2005	2009	2013	2017	2021	2025	2029	2033
	Row1 elements			Row2 elements			Row3 elements		

Let us calculate the address of 1^{st} element, i.e. first row first column element A[1][1], assuming the Lower index of an array is 1.
Here,
m=3
n=3
i =1
j=1
BA=2001
Size of an element=4 bytes

Address of 1^{st} element, A[1][1] = BA + {(1-1) * 3 +(1-1)}*4= 2001 +{0*3+0}*4 = 2001+0*4 = 2001

The address of 3^{rd} element, i.e. second row first column element A[2][1].
Here,
i=2
j=1
BA=2001
Size of an element=4 bytes
Address of 4^{th} element, A[2][1] = BA + {(2-1) * 3 +(1-1)}*4= 2001 +(1*3+0}*4
= 2001+(3+0)*4 = 2001+12=2013

2.5 Concept of Ordered List

One of the simplest and most commonly found data object is the ordered or linear list. Examples are the days of the week

(MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,

SATURDAY, SUNDAY)

or the values in a card deck

(2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)

or the floors of a building

(basement, lobby, mezzanine, first, second, third)

or the years the United States fought in World War II

(1941, 1942, 1943, 1944, 1945).

If we consider an ordered list more abstractly, we say that it is either empty or it can be written as

(a1,a2,a3, ...,an)

where the ai are atoms from some set S.

There are a variety of operations that are performed on these lists. These operations include:

(i) find the length of the list, n;

(ii) read the list from left-to-right (or right-to-left);

(iii) retrieve the i-th element, ;

(iv) store a new value into the i-th position, ;

(v) insert a new element at position causing elements numbered i,i + 1, ..., n to become numbered i + 1,i + 2, ...,n + 1;

(vi) delete the element at position causing elements numbered i + 1, ..., n to become numbered i, i + 1, ...,n - 1.

2.6 Single Variable Polynomial

A general polynomial $A(x)$ can be written as
$a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$
where $a_n \neq 0$ and we say that the degree of A is n. Then we can represent $A(x)$ as an ordered list of coefficients using a one dimensional array of length $n + 2$,
$A = (n, a_n, a_{n-1}, ..., a_1, a_0)$.

The first element is the degree of A followed by the $n + 1$ coefficients in order of decreasing exponent. This representation leads to very simple algorithms for addition and multiplication.
We have avoided the need to explicitly store the exponent of each term and instead we can deduce its value by knowing our position in the list and the degree.

The disadvantages to this representation is the large amount of wasted storage for certain polynomials. Consider $x^{1000} + 1$, for instance.
It will require a vector of length 1002, while 999 of those values will be zero. Therefore, we are led to consider an alternative scheme.

Suppose we take the polynomial $A(x)$ above and keep only its nonzero coefficients. Then we will really have the polynomial
$b_{m-1}\, x^{em-1} + b_{m-2}x^{em-2} + ... + b_o x^{eo}$ -------(1)

where each b_i is a nonzero coefficient of A and the exponents e_i are decreasing $e_{m-1} > e_{m-2} > \ldots > e_o \geqq 0$. If all of A's coefficients are nonzero, then $m = n + 1$, $e_i = i$, and $b_i = a_i$ for 0<=i<=n. Alternatively, only a_n may be nonzero, in which case $m = 1$, $b_o = a_n$, and $e_0 = n$.

In general, the polynomial in (1) could be represented by the ordered list of length $2m + 1$,
$(m, e_{m-1}, b_{m-1}, e_{m-2}, b_{m-2}, \ldots, e_0, b_0)$.

The first entry is the number of nonzero terms. Then for each term there are two entries representing an exponent-coefficient pair.

This method any better than the first scheme and certainly solves our problem with $x^{1000} + 1$, which now would be represented as (2,1000,1,0,1).

Basic algorithms will need to be more complex because we must check each exponent before we handle its coefficient, but this is not too serious.
As for storage, this scheme could be worse than the former.
For example, $x^4 + 10x^3 + 3x^2 + 1$ would have the two forms
(4,1,10,3,0,1) or (4,4,1,3,10,2,3,0,1).

In the worst case, scheme 2 requires less than twice as much storage as scheme 1 (when the degree = n and all $n + 1$ coefficients are nonzero). But scheme 1 could be much more wasteful, as in the case of $x^{1000} + 1$, where it needs more than 200 times as many locations. Therefore, we will prefer representation scheme 2 and use it.

2.6.1 Representation using arrays

The polynomial expression can be stored as 1-D array with index of the array serves as the degree of the variable term and the value in array at particular index will serve as its coefficient.
For example:
$P(x)=11x^8 + 5x^6 + x^5 + 2x^4 - 3x^2 + x + 10$

In above polynomial we have variable term ‘x’ and the highest degree of the term in ‘x’ in polynomial P(x) is 8, so we need an array large enough to hold the highest degree term 8.

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
Index	0	1	2	3	4	5	6	7	8	9
Value	10	1	-3	0	2	1	5	0	11	0

2.6.2 Polynomial as array of structure

The efficient way of storing the polynomial is to represent it as an array of structure wherein the coefficient and exponent are bound together in a structure to form one polynomial term, and then the array of ten such structures is used to represent a polynomial.

```
Struct Polynomial{
        Float COEF; //coefficient
        Int EXP;//exponent
};
```

Now create an array of structure as;
Struct Polynomial Poly[5];
The above declaration will create an array of structure to hold the polynomial terms.

Poly[0]		Poly[1]		Poly[2]		Poly[3]		Poly[4]	
COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP

The polynomial $3x^7 + x^3 - 2x + 5$ will be stored in above array of structure as shown below.

Poly[0]		Poly[1]		Poly[2]		Poly[3]		Poly[4]	
COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP
3	7	1	3	-2	1	5	0	-	-

2.6.3 Polynomial Addition

Let two polynomials A and B be
$A = 4x^9 + 8x^6 + 5x^3 + x^2 + 4x$
$B = 3x^7 + x^3 - 2x + 5$
Then,
$C = A + B = 4x^9 + 3x^7 + 8x^6 + 6x^3 + x^2 + 2x + 5$

The polynomial A & B will be stored as an array of structure as;
$A = 4x^9 + 8x^6 + 5x^3 + x^2 + 4x$

A[0]		A[1]		A[2]		A[3]		A[4]	
COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP
4	9	8	6	5	3	1	2	4	1

$B = 3x^7 + x^3 - 2x + 5$

B[0]		B[1]		B[2]		B[3]	
COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP
3	7	1	3	-2	1	5	0

$C = A + B = 4x^9 + 3x^7 + 8x^6 + 6x^3 + x^2 + 2x + 5$

A[0]		A[1]		A[2]		A[3]		A[4]		A[5]		A[6]	
COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP	COEF	EXP
4	9	3	7	8	6	6	3	1	2	2	1	5	0

The polynomials A and B are to be added to get the resultant polynomial C. Here, we assume that the two polynomials are in descending order of their exponents.

Let us revise the procedure of adding two polynomials. Let i, j, and k be the three indices to keep track of the current term of the polynomials A, B, and C, respectively, being processed. Initially, it tracks the fi rst term. The major steps involved can be listed as follows:

1. If the exponents of the two terms of polynomials A and B are equal, then the coefficients are added, and the new term is stored in the resultant polynomial C and advance i, j, and k to track to the next term.
2. If the exponent of the term indicated by i in A is less than the exponent of the current term specifi ed by j of B, then copy the current term of B pointed by j in the location pointed by k in polynomial C. The pointers j and k are advanced to the next term.

3. If the exponent of the term pointed by j in B is less than the exponent of the current term pointed by i of A, then copy the current term of A pointed by i in the location pointed by k in polynomial C. Advance the pointer i and k to the next term.
Each time a new term is generated, its coefficient and exponent fields are set accordingly. The resultant term then is attached to the end of the polynomial C. The current term of polynomial C is indicated by k.

```
procedure PADD(A,B,C)
1. Read two polynomials say A and B
2. Let M and N denote total terms in A and B respectively.
Here, C is resultant polynomial.
4. Let i = j = k = 0
5. while (i < M and j < N) do
       begin // repeat till one of the polynomials is copied
       if(A[i].Exp = B[j].Exp)
              begin
                     C[k].Coef = A[i].Coef+B[j].Coef
                     C[k].Exp = A[i].Exp;
                     i = i + 1; j = j + 1, k = k + 1
              end
       else
              if(A[i].Exp > B[j].Exp)
              begin
                     C[k].Coef = A[i].Coef;
                     C[k].Exp = A[i].Exp;
                     i = i + 1
                     k = k + 1
              end
       else
              begin
                     C[k].Coef = B[j].Coef;
                     C[k].Exp = B[j].Exp;
                     j = j + 1
                     k = k + 1
              end
 end
```

```
6. while(i < m) do
       begin // copy remaining terms
              C[k].Coef = A[i].Coef;
              C[k].Exp = A[i].Exp;
              i = i + 1
              k = k + 1
       end
7. while (j < n) do
       begin // copy remaining terms
              C[k].Coef = B[j].Coef;
              C[k].Exp = B[j].Exp;
              j = j + 1
              k = k + 1
       end
8) stop
```

2.6.4 Polynomial Multiplication

Let A = 4x9 + 3x6 + 5x3 + 1 and B = 3x6 + x2 - 2x be the two polynomials to be multiplied, and the resultant polynomial be C. Let us revise the paper-pencil method. The polynomial A is multiplied by each term of B. We get n partial products if B has n terms in it. Finally, we add all these partial products to get the resultant polynomial C.
This method generates partial products each of length m, where m is the length of the polynomial A. n such partial products are generated and stored and finally added to get the resultant polynomial. Here, m and n are input dependent. Let us devise a better approach where we need not generate, store, and then add all partial products. A better solution is to pick up a term of polynomial B and multiply it with each term of A. One term of B and one term of A when multiplied yield one resultant term. This term can be immediately added to the resultant polynomial C, and this process is to be repeated.

To add a resultant term to polynomial C, the resultant term is compared with each term of the resultant polynomial C. Then the new term is inserted at the appropriate location in polynomial C. If the new term with equal exponent is found, then the term is added, else it is inserted in the resultant polynomial at an appropriate position.
This process is repeated for each term of B with each term of A. The major steps are listed briefly as follows:
1. Let A and B be two polynomials.
2. Let the number of terms in A be M, and number of terms in B be N.
3. Let C be the resultant polynomial to be computed as C = A x B.
4. Let us denote the ith term of polynomial B as tBi. For each term tBi of polynomial B, repeat steps 5 to 7 where i = 1 to N.
5. Let us denote the jth term of polynomial A as tAj. For each term of tAj of polynomial A, repeat steps 6 and 7 where j = 1 to M.
6. Multiply tAj and tBi. Let the new term be tCk = tAj x tBi.
Linear Data Structure using Arrays 67
7. Compare tCk with each term of polynomial C. If a term with equal exponent is found, then add the new term tCk to that term of polynomial C, else search for an appropriate position for the term tCk and insert the same in polynomial C.
8. Stop.

Let
$A = 4x^9 + 3x^6 + 5x^3 + 1$,
$B = 3x^6 + x^2$, and C be the resultant polynomial.

Initially, C is an empty polynomial.

1. We multiply each term of A with the first term of B. To start with, multiply $4x^9$ with $3x^6$ and the result is $12x^{15}$. Currently, C is empty, so there is no term in it with the exponent 15; therefore, we insert it in polynomial C. Now, polynomial C is
$C = 12x^{15}$
Now, continue to multiply $3x^6$ with $3x^6$, and the result obtained is $9x^{12}$. There is no term in polynomial C with exponent 12, so we insert it in polynomial C at an appropriate location. Now, polynomial C is
$C = 12x^{15} + 9x^{12}$
Continuing in a similar manner for the remaining two terms of polynomial A, we get polynomial C as
$C = 12x^{15} + 9x^{12} + 15x^9 + 3x^6$

2. Now, multiply each term of A with the second term of B. Initially, multiply $4x^9$ with x^2 and the result is $4x^{11}$. There is no term in C with exponent 11, we insert it in polynomial C at an appropriate location. So now we get polynomial C as
$C = 12x^{15} + 9x^{12} + 4x^{11} + 15x^9 + 3x^6$

Continue to multiply $3x^6$ with x^2 and the result is $3x^8$. There is no term in polynomial C with exponent 8, so we add it at an appropriate place. Now, the polynomial C is
$C = 12x^{15} + 9x^{12} + 4x^{11} + 15x^9 + 3x^8 + 3x^6$

Let us now multiply $5x^3$ with x^2 and we get $5x^5$. There is no term in C with exponent 5, so we insert it in polynomial C at a proper location. Now,
$C = 12x^{15} + 9x^{12} + 4x^{11} + 15x^9 + 3x^8 + 3x^6 + 5x^5$

Let us now multiply the term 1 of A with x^2; we get x^2. There is no term in C with exponent 2, so we insert it in polynomial C at an appropriate location. Therefore,
$C = 12x^{15} + 9x^{12} + 4x^{11} + 15x^9 + 3x^8 + 3x^6 + 5x^5 + x^2$

This is the resultant polynomial C as a result of A x B.

2.7 Sparse Matrix

A matrix is a mathematical object which arises in many physical problems. As computer scientists, we are interested in studying ways to represent matrices so that the operations to be performed on them can be carried out efficiently.

A general matrix consists of m rows and n columns of numbers as shown below.

		Col1	Col2	Col3
		0	1	2
Row1	0	10	20	30
Row2	1	40	50	60
Row3	2	70	80	90

In many situations, the matrix size is very large but most of the elements in it are 0s (less important or irrelevant data). Only a small fraction of the matrix is actually used. A matrix of such type is called a *sparse matrix*, as the matrix is filled sparsely by data and most of the positions are empty or contain non-relevant data.

	Col1	Col2	Col3	Col4	Col5	Col6
Row1	0	0	6	1	1	2
Row2	0	1	0	0	9	9
Row3	4	0	0	0	0	0
Row4	0	0	0	0	0	0
Row5	7	0	0	0	0	6
Row6	0	2	0	0	0	8

The above matrix is sparse as there are only a few non-zero elements, so there is wastage of space in the above representation and storage of the values of 6 x 6 matrix.
This has created a need of the alternate but efficient representation of the above matrix.

2.7.1 Sparse matrix representation using array

Sparse matrix addition
Transpose of sparse
matrix- Simple and Fast Transpose
Time and Space trade-off