**Divide And Conquer**   
This technique can be divided into the following three parts:

1. **Divide:**This involves dividing the problem into smaller sub-problems.
2. **Conquer:**Solve sub-problems by calling recursively until solved.
3. **Combine:**Combine the sub-problems to get the final solution of the whole problem.

The following are some standard algorithms that follow Divide and Conquer algorithm.

1. [**Quicksort**](https://www.geeksforgeeks.org/quick-sort/) is a sorting algorithm. The algorithm picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.
2. [**Merge Sort**](https://www.geeksforgeeks.org/merge-sort/) is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.
3. [**Closest Pair of Points**](https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/) The problem is to find the closest pair of points in a set of points in the x-y plane. The problem can be solved in O(n^2) time by calculating the distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(N log N) time.
4. [**Strassen’s Algorithm**](https://www.geeksforgeeks.org/strassens-matrix-multiplication/) is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices needs 3 nested loops and is O(n^3). Strassen’s algorithm multiplies two matrices in O(n^2.8974) time.
5. [**Fast Fourier Transform (FFT) algorithm**](http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm) is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(N log N) time.

Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in the array. If the values match, return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs for the left side of the middle element, else recurs for the right side of the middle element. Contrary to popular belief, this is not an example of Divide and Conquer because there is only one sub-problem in each step (Divide and conquer requires that there must be two or more sub-problems) and hence this is a case of Decrease and Conquer.

**Divide And Conquer algorithm :**

DAC(a, i, j)

{

if(small(a, i, j))

return(Solution(a, i, j))

else

m = divide(a, i, j) // f1(n)

b = DAC(a, i, mid) // T(n/2)

c = DAC(a, mid+1, j) // T(n/2)

d = combine(b, c) // f2(n)

return(d)

}

**Recurrence Relation for DAC algorithm :**  
This is a recurrence relation for the above program.

O(1) if n is small

T(n) = f1(n) + 2T(n/2) + f2(n)

**Binary Search:** Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

**Recursive**implementation of Binary Search

* C++

|  |
| --- |
| // C++ program to implement recursive Binary Search  #include <bits/stdc++.h>  using namespace std;    // A recursive binary search function. It returns  // location of x in given array arr[l..r] is present,  // otherwise -1  int binarySearch(int arr[], int l, int r, int x)  {      if (r >= l) {          int mid = l + (r - l) / 2;            // If the element is present at the middle          // itself          if (arr[mid] == x)              return mid;            // If element is smaller than mid, then          // it can only be present in left subarray          if (arr[mid] > x)              return binarySearch(arr, l, mid - 1, x);            // Else the element can only be present          // in right subarray          return binarySearch(arr, mid + 1, r, x);      }        // We reach here when element is not      // present in array      return -1;  }    int main(void)  {      int arr[] = { 2, 3, 4, 10, 40 };      int x = 10;      int n = sizeof(arr) / sizeof(arr[0]);      int result = binarySearch(arr, 0, n - 1, x);      (result == -1) ? cout << "Element is not present in array"                     : cout << "Element is present at index " << result;      return 0;  } |

**Output :**

Element is present at index 3

Here you can create a check function for easier implementation.

Here is recursive implementation with check function which I feel is a much easier implementation:

* C++

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;    //define array globally  const int N = 1e6 +4;    int a[N];  int n;//array size    //elememt to be searched in array     int k;    bool check(int dig)  {      //element at dig position in array      int ele=a[dig];        //if k is less than      //element at dig position      //then we need to bring our higher ending to dig      //and then continue further      if(k<=ele)      {          return 1;      }      else      {      return 0;      }  }  void binsrch(int lo,int hi)  {      while(lo<hi)      {          int mid=(lo+hi)/2;          if(check(mid))          {              hi=mid;          }          else          {              lo=mid+1;          }      }      //if a[lo] is k      if(a[lo]==k)          cout<<"Element found at index "<<lo;// 0 based indexing      else          cout<<"Element doesnt exist in array";//element was not in our array    }      int main()  {      cin>>n;     for(int i=0; i<n; i++)     {      cin>>a[i];     }     cin>>k;       //it is being given array is sorted     //if not then we have to sort it       //minimum possible point where our k can be is starting index     //so lo=0     //also k cannot be outside of array so end point     //hi=n       binsrch(0,n);        return 0;  } |

**Iterative**implementation of Binary Search

* C++

|  |
| --- |
| // C++ program to implement recursive Binary Search  #include <bits/stdc++.h>  using namespace std;    // A iterative binary search function. It returns  // location of x in given array arr[l..r] if present,  // otherwise -1  int binarySearch(int arr[], int l, int r, int x)  {      while (l <= r) {          int m = l + (r - l) / 2;            // Check if x is present at mid          if (arr[m] == x)              return m;            // If x greater, ignore left half          if (arr[m] < x)              l = m + 1;            // If x is smaller, ignore right half          else              r = m - 1;      }        // if we reach here, then element was      // not present      return -1;  }    int main(void)  {      int arr[] = { 2, 3, 4, 10, 40 };      int x = 10;      int n = sizeof(arr) / sizeof(arr[0]);      int result = binarySearch(arr, 0, n - 1, x);      (result == -1) ? cout << "Element is not present in array"                     : cout << "Element is present at index " << result;      return 0;  } |

**Output :**

Element is present at index 3

**Time Complexity:**   
The time complexity of Binary Search can be written as

T(n) = T(n/2) + c

The above recurrence can be solved either using the Recurrence Tree method or Master method. It falls in case II of the Master Method and the solution of the recurrence is .  
**Auxiliary Space:** O(1) in case of iterative implementation. In the case of recursive implementation, O(Logn) recursion call stack space.  
**Algorithmic Paradigm:** [Decrease and Conquer](https://www.geeksforgeeks.org/decrease-and-conquer/).

**Note:**

Here we are using

*int mid = low + (high – low)/2;*

Maybe, you wonder why we are calculating the ***middle index***this way, we can simply add the *lower and higher index and divide it by 2.*

*int mid = (low + high)/2;*

But if we calculate the***middle index***like this means our code is not 100% correct, it contains bugs.

That is, it fails for larger values of int variables low and high. Specifically, it fails if the sum of low and high is greater than the maximum positive int value(231 – 1).

The sum overflows to a negative value and the value stays negative when divided by 2. In java, it throws *ArrayIndexOutOfBoundException.*

*int mid = low + (high – low)/2;*

*2)* ***MERGE SORT***

Like [QuickSort](https://www.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = l+ (r-l)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

The following diagram from [wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg" \t "_blank) shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



* C++
* A-> 1,2,3,4,21
* B-> 20,34,56,78,90
* C->1,2,3,4,20,21, 34,56,78,90

|  |
| --- |
| // C++ program for Merge Sort  #include <iostream>  using namespace std;    // Merges two subarrays of array[].  // First subarray is arr[begin..mid]  // Second subarray is arr[mid+1..end]  void merge(int array[], int const left, int const mid, int const right)  {      auto const subArrayOne = mid - left + 1;      auto const subArrayTwo = right - mid;        // Create temp arrays      auto \*leftArray = new int[subArrayOne],           \*rightArray = new int[subArrayTwo];        // Copy data to temp arrays leftArray[] and rightArray[]      for (auto i = 0; i < subArrayOne; i++)          leftArray[i] = array[left + i];      for (auto j = 0; j < subArrayTwo; j++)          rightArray[j] = array[mid + 1 + j];        auto indexOfSubArrayOne = 0, // Initial index of first sub-array          indexOfSubArrayTwo = 0; // Initial index of second sub-array      int indexOfMergedArray = left; // Initial index of merged array        // Merge the temp arrays back into array[left..right]      while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {          if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {              array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];              indexOfSubArrayOne++;          }          else {              array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];              indexOfSubArrayTwo++;          }          indexOfMergedArray++;      }      // Copy the remaining elements of      // left[], if there are any      while (indexOfSubArrayOne < subArrayOne) {          array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];          indexOfSubArrayOne++;          indexOfMergedArray++;      }      // Copy the remaining elements of      // right[], if there are any      while (indexOfSubArrayTwo < subArrayTwo) {          array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];          indexOfSubArrayTwo++;          indexOfMergedArray++;      }  }    // begin is for left index and end is  // right index of the sub-array  // of arr to be sorted \*/  void mergeSort(int array[], int const begin, int const end)  {      if (begin >= end)          return; // Returns recursively        auto mid = begin + (end - begin) / 2;      mergeSort(array, begin, mid);      mergeSort(array, mid + 1, end);      merge(array, begin, mid, end);  }    // UTILITY FUNCTIONS  // Function to print an array  void printArray(int A[], int size)  {      for (auto i = 0; i < size; i++)          cout << A[i] << " ";  }    // Driver code  int main()  {      int arr[] = { 12, 11, 13, 5, 6, 7 };      auto arr\_size = sizeof(arr) / sizeof(arr[0]);        cout << "Given array is \n";      printArray(arr, arr\_size);        mergeSort(arr, 0, arr\_size - 1);        cout << "\nSorted array is \n";      printArray(arr, arr\_size);      return 0;  }    // This code is contributed by Mayank Tyagi  // This code was revised by Joshua Estes |

**Output**

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

**Time Complexity:** Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

MergeSort(A, p, r) // T(n)

{

if p > r

return

q = (p+r)/2

mergeSort(A, p, q) // T(n/2)

mergeSort(A, q+1, r) // T(n/2)

merge(A, p, q, r) // O(n)

}

T(n) = 2T(n/2) + O(n)

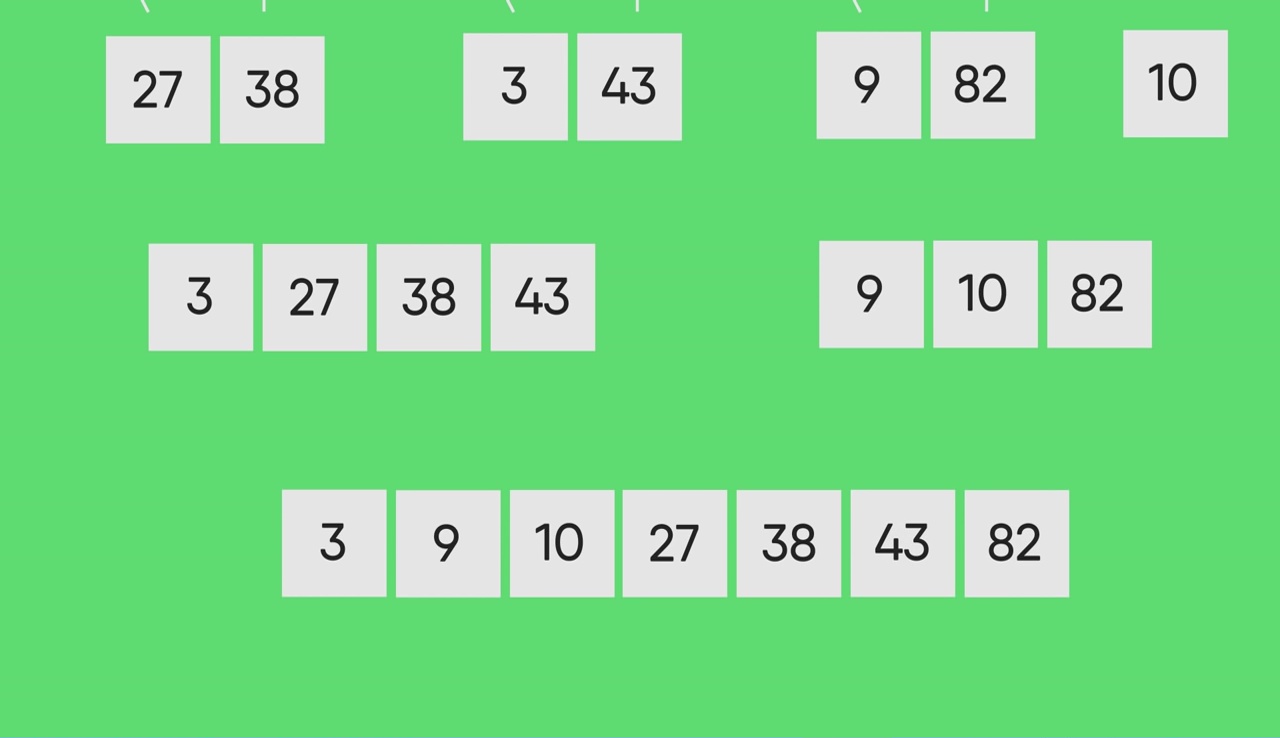
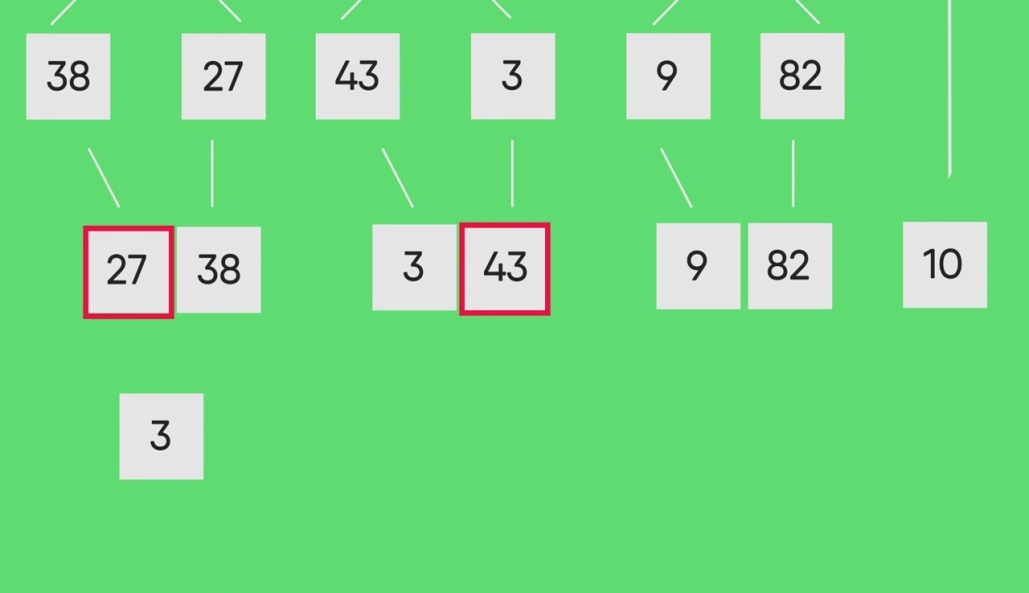
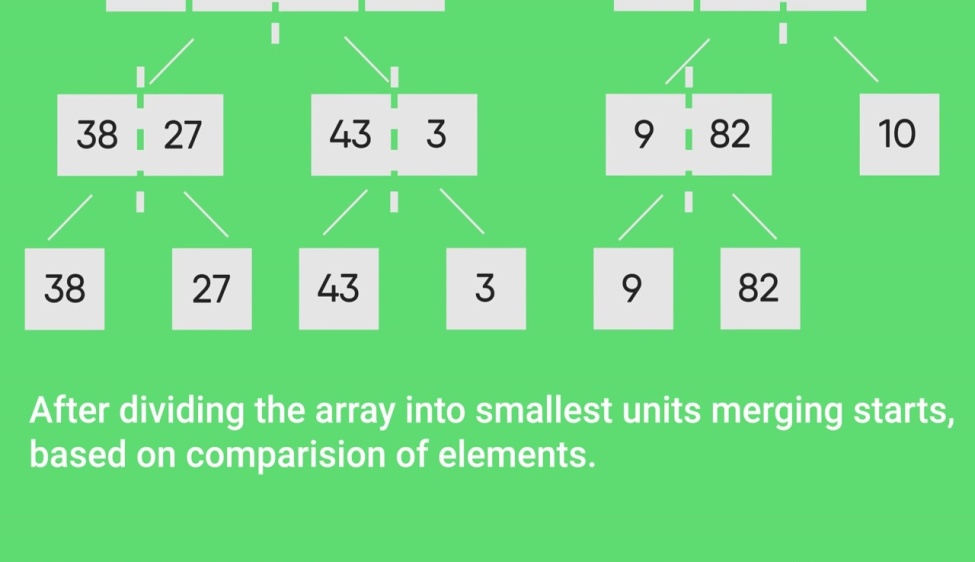
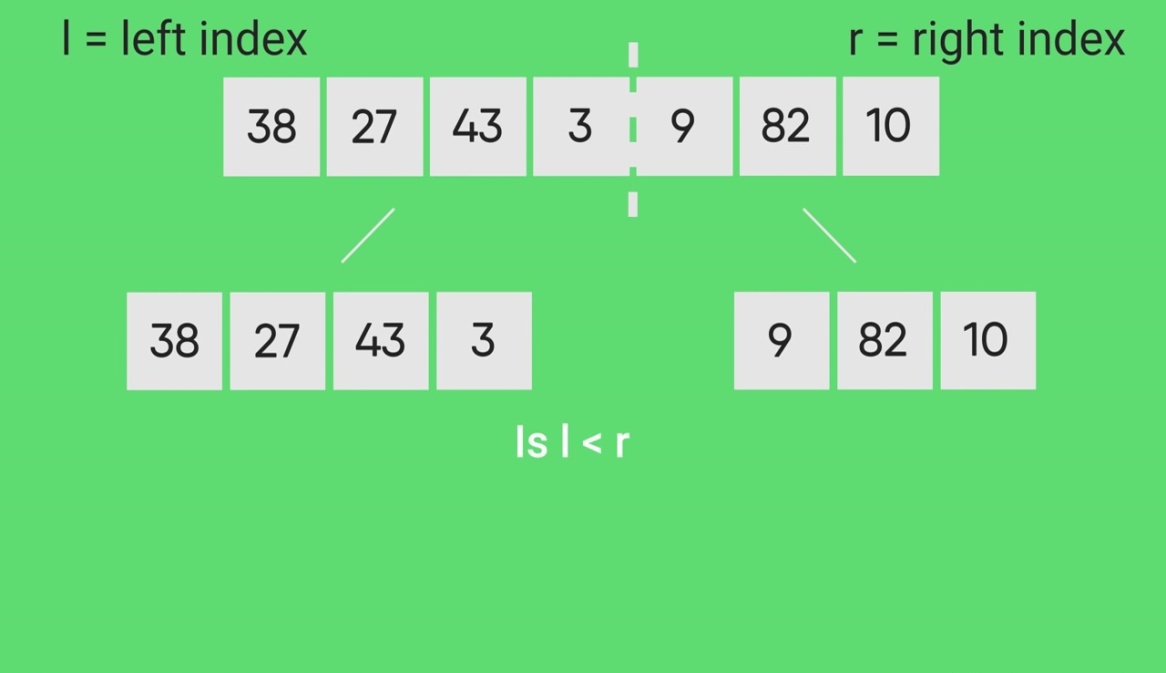
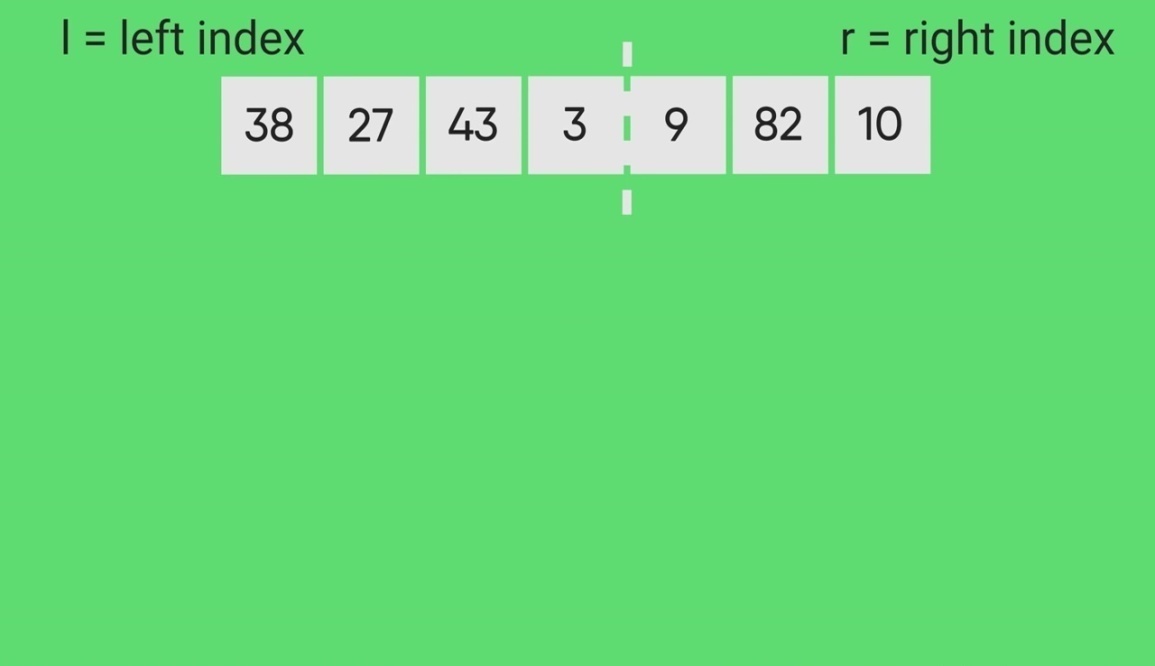
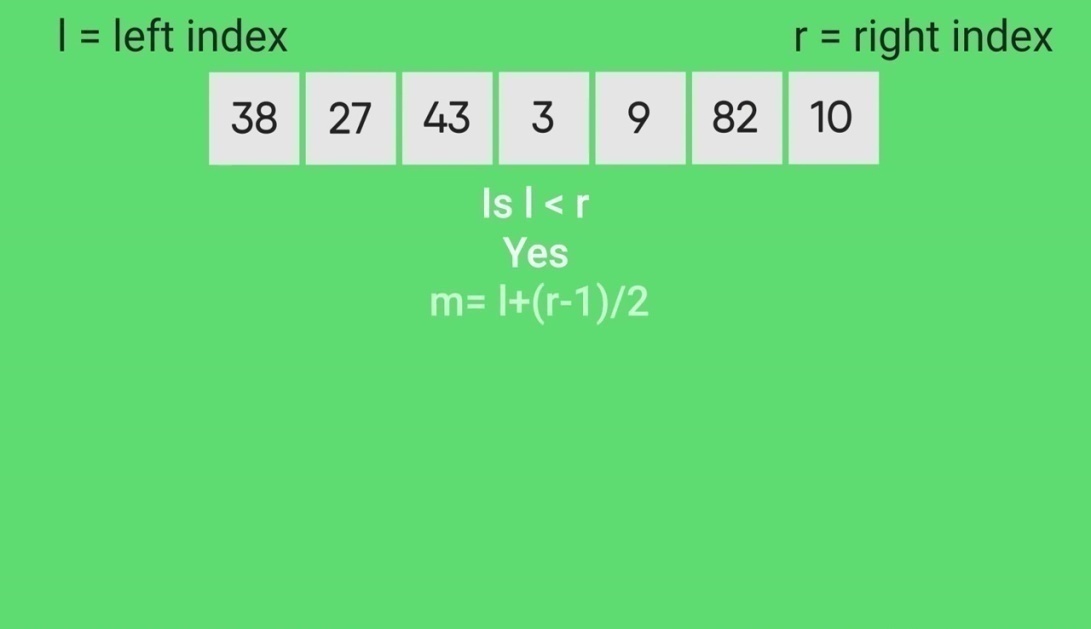
The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is θ(nLogn). Time complexity of Merge Sort is  θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.  
**Auxiliary Space:** O(n)  
**Algorithmic Paradigm (APPROACH):**Divide and Conquer  
**Sorting In Place:** No in a typical implementation  
**Stable:** Yes

**Applications of Merge Sort**

1. [Merge Sort is useful for sorting linked lists in O(nLogn) time](https://www.geeksforgeeks.org/merge-sort-for-linked-list/). In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore, the merge operation of merge sort can be implemented without extra space for linked lists.  
   In arrays, we can do random access as elements are contiguous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in the linked list. Quick Sort requires a lot of this kind of access. In a linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have a continuous block of memory. Therefore, the overhead increases for quicksort. Merge sort accesses data sequentially and the need of random access is low.
2. [Inversion Count Problem](https://www.geeksforgeeks.org/counting-inversions/)
3. Used in [External Sorting](http://en.wikipedia.org/wiki/External_sorting)

**Drawbacks of Merge Sort**

* Slower comparative to the other sort algorithms for smaller tasks.
* Merge sort algorithm requires an additional memory space of 0(n) for the temporary array.
* It goes through the whole process even if the array is sorted.



*Find min and max simultaneously is not in syllabus*

**Example:**   
To find the maximum and minimum element in a given array.

**Input:** { 70, 250, 50, 80, 140, 12, 14 }

**Output:** The minimum number in a given array is : 12

The maximum number in a given array is : 250

**Approach:** To find the maximum and minimum element from a given array is an application for divide and conquer. In this problem, we will find the maximum and minimum elements in a given array. In this problem, we are using a divide and conquer approach(DAC) which has three steps divide, conquer and combine.

**For Maximum:**   
In this problem, we are using the recursive approach to find maximum where we will see that only two elements are left and then we can easily using condition i.e. if(a[index]>a[index+1].)  
In a program line a[index] and a[index+1])condition will ensure only two elements in left.

*if(index >= l-2)   
{   
if(a[index]>a[index+1])   
{   
// (a[index]   
// Now, we can say that the last element will be maximum in a given array.   
}   
else   
{   
//(a[index+1]   
// Now, we can say that last element will be maximum in a given array.   
}  
}*

In the above condition, we have checked the left side condition to find out the maximum. Now, we will see the right side condition to find the maximum.   
Recursive function to check the right side at the current index of an array.

*max = DAC\_Max(a, index+1, l);   
// Recursive call*

Now, we will compare the condition and check the right side at the current index of a given array.   
In the given program, we are going to implement this logic to check the condition on the right side at the current index.

*// Right element will be maximum.   
if(a[index]>max)   
return a[index];  
// max will be maximum element in a given array.   
else   
return max;   
}*

**For Minimum:**   
In this problem, we are going to implement the recursive approach to find the minimum no. in a given array.

*int DAC\_Min(int a[], int index, int l)   
//Recursive call function to find the minimum no. in a given array.  
if(index >= l-2)   
// to check the condition that there will be two-element in the left   
then we can easily find the minimum element in a given array.   
{   
// here we will check the condition   
if(a[index]<a[index+1])   
return a[index];   
else   
return a[index+1];   
}*

Now, we will check the condition on the right side in a given array.

*// Recursive call for the right side in the given array.   
min = DAC\_Min(a, index+1, l);*

Now, we will check the condition to find the minimum on the right side.

*// Right element will be minimum   
if(a[index]<min)   
return a[index];   
// Here min will be minimum in a given array.   
else   
return min;*

**Implementation:**

* C++

|  |
| --- |
| // C++ code to demonstrate Divide and  // Conquer Algorithm#include<iostream>  # include<iostream>  using namespace std;    // function to find the maximum no.  // in a given array.  int DAC\_Max(int arr[], int index, int l)  {      int max;      if(index >= l - 2)      {          if(arr[index] > arr[index + 1])            return arr[index];          else            return arr[index + 1];      }      max = DAC\_Max(arr, index + 1, l);      if(arr[index] > max)         return arr[index];      else         return max;  }    // Function to find the minimum no.  // in a given array  int DAC\_Min(int arr[], int index, int l)  {      int min;      if(index >= l - 2)      {          if(arr[index] < arr[index + 1])            return arr[index];          else            return arr[index + 1];      }        min = DAC\_Min(arr, index + 1, l);      if(arr[index] < min)         return arr[index];      else         return min;  }    // Driver code  int main()  {      int arr[] = {120, 34, 54, 32, 58, 11, 90};      int n = sizeof(arr) / sizeof(arr[0]);      int max, min;      max = DAC\_Max(arr, 0, n);      min = DAC\_Min(arr, 0, n);      cout << "Maximum: " << max << endl;      cout << "Minimum: " << min << endl;      return 0;  }    // This code is contributed by probinsah. |

**Output**

Maximum: 120

Minimum: 11

***Divide and Conquer (D & C) vs Dynamic Programming (DP)***   
Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems. How do choose one of them for a given problem? Divide and Conquer should be used when the same subproblems are not evaluated many times. Otherwise Dynamic Programming or Memoization should be used. For example, Quicksort is a Divide and Conquer algorithm, we never evaluate the same subproblems again. On the other hand, for calculating the nth Fibonacci number, Dynamic Programming should be preferred (See [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)for details).

# QuickSort

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)

Like [Merge Sort](https://www.geeksforgeeks.org/merge-sort/), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

**Pseudo Code for recursive QuickSort function :**

/\* low --> Starting index, high --> Ending index \*/

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[pi] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}



**Partition Algorithm**   
There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

/\* low --> Starting index, high --> Ending index \*/

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[pi] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

**Pseudo code for partition()**

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

partition (arr[], low, high)

{

// pivot (Element to be placed at right position)

pivot = arr[high];

i = (low - 1) // Index of smaller element and indicates the

// right position of pivot found so far

for (j = low; j <= high- 1; j++)

{

// If current element is smaller than the pivot

if (arr[j] < pivot)

{

i++; // increment index of smaller element

swap arr[i] and arr[j]

}

}

swap arr[i + 1] and arr[high])

return (i + 1)

}

**Illustration of partition() :**

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1

**j = 0** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 0**

arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j

// are same

**j = 1** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 1**

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

**j = 3** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 4** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 2**

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

**j = 5** : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

**i = 3**

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

**Finally we place pivot at correct position by swapping**

**arr[i+1] and arr[high] (or pivot)**

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after

it.

**Implementation:**   
Following are the implementations of QuickSort:

* C++

|  |
| --- |
| /\* C++ implementation of QuickSort \*/  #include <bits/stdc++.h>  using namespace std;    // A utility function to swap two elements  void swap(int\* a, int\* b)  {      int t = \*a;      \*a = \*b;      \*b = t;  }    /\* This function takes last element as pivot, places  the pivot element at its correct position in sorted  array, and places all smaller (smaller than pivot)  to left of pivot and all greater elements to right  of pivot \*/  int partition (int arr[], int low, int high)  {      int pivot = arr[high]; // pivot      int i = (low - 1); // Index of smaller element and indicates the right position of pivot found so far        for (int j = low; j <= high - 1; j++)      {          // If current element is smaller than the pivot          if (arr[j] < pivot)          {              i++; // increment index of smaller element              swap(&arr[i], &arr[j]);          }      }      swap(&arr[i + 1], &arr[high]);      return (i + 1);  }    /\* The main function that implements QuickSort  arr[] --> Array to be sorted,  low --> Starting index,  high --> Ending index \*/  void quickSort(int arr[], int low, int high)  {      if (low < high)      {          /\* pi is partitioning index, arr[p] is now          at right place \*/          int pi = partition(arr, low, high);            // Separately sort elements before          // partition and after partition          quickSort(arr, low, pi - 1);          quickSort(arr, pi + 1, high);      }  }    /\* Function to print an array \*/  void printArray(int arr[], int size)  {      int i;      for (i = 0; i < size; i++)          cout << arr[i] << " ";      cout << endl;  }    // Driver Code  int main()  {      int arr[] = {10, 7, 8, 9, 1, 5};      int n = sizeof(arr) / sizeof(arr[0]);      quickSort(arr, 0, n - 1);      cout << "Sorted array: \n";      printArray(arr, n);      return 0;  } |

**Output**

Sorted array:

1 5 7 8 9 10

**Analysis of QuickSort**   
Time taken by QuickSort, in general, can be written as following.

T(n) = T(k) + T(n-k-1) + (n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.   
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

***Worst Case:*** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + (n)

which is equivalent to

T(n) = T(n-1) + (n)

The solution of above recurrence is  (n2).

***Best Case:*** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + (n)

The solution of above recurrence is (n logn). It can be solved using case 2 of [Master Theorem](http://en.wikipedia.org/wiki/Master_theorem).

***Average Case:***   
To do average case analysis, we need to [consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/).   
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + (n)

Solution of above recurrence is also O(nLogn)  
Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like [Merge Sort](https://www.geeksforgeeks.org/merge-sort/) and [Heap Sort](https://www.geeksforgeeks.org/heap-sort/), QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

**Is QuickSort**[**stable**](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/)**?**   
The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

**Is QuickSort**[**In-place**](https://www.geeksforgeeks.org/in-place-algorithm/)**?**

As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.

**What is 3-Way QuickSort?**   
In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.   
Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:   
a) arr[l..i] elements less than pivot.   
b) arr[i+1..j-1] elements equal to pivot.   
c) arr[j..r] elements greater than pivot.

# Divide and Conquer | Set 5 (Strassen’s Matrix Multiplication)

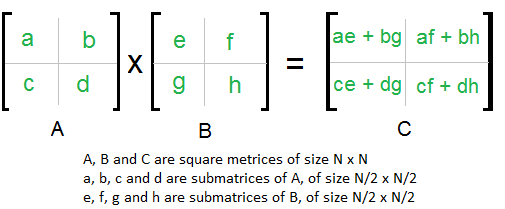
* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 12 Oct, 2021

Given two square matrices A and B of size n x n each, find their multiplication matrix.   
***Naive Method***   
Following is a simple way to multiply two matrices. 

|  |
| --- |
| void multiply(int A[][N], int B[][N], int C[][N])  {      for (int i = 0; i < N; i++)      {          for (int j = 0; j < N; j++)          {              C[i][j] = 0;              for (int k = 0; k < N; k++)              {                  C[i][j] += A[i][k]\*B[k][j];              }          }      }  } |

Time Complexity of above method is O(N3). 

***Divide and Conquer***  
Following is simple Divide and Conquer method to multiply two square matrices.   
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.   
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh. 



In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes O(N2) time. So the time complexity can be written as

T(N) = 8T(N/2) + O(N2)

From [Master's Theorem](https://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/), time complexity of above method is O(N3)

which is unfortunately same as the above naive method.

***Simple Divide and Conquer also leads to O(N3), can there be a better way?***   
In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of**Strassen’s method** is to reduce the number of recursive calls to 7. Strassen’s method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen’s method, the four sub-matrices of result are calculated using following formulae.



**Time Complexity of Strassen’s Method**   
Addition and Subtraction of two matrices takes O(N2) time. So time complexity can be written as 

T(N) = 7T(N/2) + O(N2)

From [Master's Theorem](https://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/), time complexity of above method is

O(NLog7) which is approximately O(N2.8074)

Generally Strassen’s Method is not preferred for practical applications for following reasons.   
1) The constants used in Strassen’s method are high and for a typical application Naive method works better.   
2) For Sparse matrices, there are better methods especially designed for them.   
3) The submatrices in recursion take extra space.   
4) Because of the limited precision of computer arithmetic on non integer values, larger errors accumulate in Strassen’s algorithm than in Naive Method (Source: [CLRS Book](http://www.flipkart.com/introduction-algorithms-3rd/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)) 

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Strassen’s Algorithm | Multiply two matrices in C++ Many times, during complex mathematical calculations, we require to multiply two matrices.  To implement the multiplication of two matrices, we can choose from the following techniques:   1. Basic Matrix multiplication 2. Strassen’s Algorithm  ****Technique 1: Basic Matrix multiplication**** In this method, we use the pen paper trick itself. The algorithm for the same is stated below:  **Logic:**  Multiply rows of first matrix with columns of second matrix. We take each row r at a time, take its first element r1 , then, we multiply it with all the elements of column C  c1,2,3,..n. We use this in an iterative manner and get the result.  **Algorithm:**   1. Input the no. of rows and columns of both the elements. 2. Check if the number of columns of first matrix is same as the rows  of second matrix(condition for matrix multiplication) 3. Applying proper loops, use the formula Cij = ∑(Aik \* Bik)  where, i,j,k are positive integers and i,j,k<=n 4. Next, we display the final matrix.   **Code:**  #include <iostream>  using namespace std;  void multiply(int[5][5], int[5][5], int, int, int);  int display(int[5][5], int, int);  int main()  {  int a[5][5], b[5][5], r1, c1, r2, c2;  cout<<"\n Enter rows for first matrix: ";  cin>>r1;  cout<<"\n Enter columns for second matrix: ";  cin>>c1;  cout<<"\n Enter rows for first matrix: ";  cin>>r2;  cout<<"\n Enter columns for second matrix: ";  cin>>c2;  // To check if columns of first matrix are equal to rows of second matrix  if (c1 != r2)  return 0;  // Storing elements of first matrix.  cout<<"\n Enter elements of first matrix \n";  for(int i=0; i<r1; i++)  {  for(int j=0; j<c1; j++)  cin>>a[i][j];  }  // Storing elements of second matrix.  cout<<"\n Enter elements of second matrix\n";  for(int i=0; i<r2; i++)  {  for(int j=0; j<c2; j++)  cin>>b[i][j];  }  display(a,r1,c1);  display(b,r2,c2);  //calling the function to multiply a and b. passing number of rows  //and columns in both of them  multiply(a, b, r1, c2, c1);  return 0;  }  void multiply(int a[5][5], int b[5][5], int row, int col, int c1)  {  int c[5][5];  //input 0 for all values of c, in order to remove  //the garbage values assigned earlier  for(int i=0; i<row; i++)  {  for(int j=0; j<col; j++)  c[i][j]=0;  }  //we apply the same formula as above  for(int i=0; i<row; i++)  {  for(int j=0; j<col; j++)  {  for(int k=0; k<c1; k++)//columns of first matrix || rows of second matrix  c[i][j]+=a[i][k]\*b[k][j];  }  }  //to display matrix  cout<<"\n Matrix c after matrix multiplication is:\n";  display(c, row, col);  }  int display(int c[5][5], int row, int col)  {  cout<<"\n Matrix is:\n";  for(int i=0; i<row; i++)  {  for(int j=0; j<col; j++)  cout<<c[i][j]<<" ";  cout<<"\n";  }  return 0;  }  **Output:**  Enter rows for first matrix: 2  Enter columns for second matrix: 3  Enter rows for first matrix: 3  Enter columns for second matrix: 2  Enter elements of first matrix  5 7 6  1 3 7  Enter elements of second matrix  6 2  8 9  3 6  Matrix is  5 7 6  1 3 7  Matrix is  6 2  8 9  3 6  Matrix c after matrix multiplication is:  Matrix is  104 109  51 71 ****Technique 2: Strassen’s Algorithm**** In this method, we use the algorithm given by [Strassen](https://en.wikipedia.org/wiki/Volker_Strassen). The advantage of this algorithm is, that it uses less number of operations then the naive method.  It uses divide and conquer strategy, and thus, divides the square matrix of size n to n/2.  It reduces the 8 recursive calls to 7.  In this program, we use a 4×4 matrix.  **Logic:**  Divide the matrix, then use the Strassen’s formulae:  p=(a11+a22)\*(b11+b22);  q=(a21+a22)\*b11;  r=a11\*(b12-b22);  s=a22\*(b21-b11);  t=(a11+a12)\*b22;  u=(a11-a21)\*(b11+b12);  v=(a12-a22)\*(b21+b22);  for two 2×2 matrices a and b, where,  A=   |  |  | | --- | --- | | a11 | a12 | | a21 | a22 |   B=   |  |  | | --- | --- | | b11 | b12 | | b21 | b22 |   Multiplied matrix will have  C=   |  |  | | --- | --- | | p+s-t+v | r+t | | q+s | p+r-q+u |   **Algorithm:**   1. Input the no. rows and columns of both the elements 2. Check ifthe number of columns of first matrix is same as the rows of second matrix(condition for matrix multiplication). 3. Use the strassen’s formulae. 4. Feeding the values in the final matrix. 5. Next, we display the final matrix.   **Code:**  #include<iostream>  using namespace std;  double a[4][4];  double b[4][4];  void insert(double x[4][4])  {  double val;  for(int i=0;i<4;i++)  {  for(int j=0;j<4;j++)  {  cin>>val;  x[i][j]=val;  }  }  }  double cal11(double x[4][4])  {  return (x[1][1] \* x[1][2])+ (x[1][2] \* x[2][1]);  }  double cal21(double x[4][4])  {  return (x[3][1] \* x[4][2])+ (x[3][2] \* x[4][1]);  }  double cal12(double x[4][4])  {  return (x[1][3] \* x[2][4])+ (x[1][4] \* x[2][3]);  }  double cal22(double x[4][4])  {  return (x[2][3] \* x[1][4])+ (x[2][4] \* x[1][3]);  }  int main()  {  double a11,a12,a22,a21,b11,b12,b21,b22,a[4][4],b[4][4];  double p,q,r,s,t,u,v,c11,c12,c21,c22;  //insert values in the matrix a  cout<<"\n a: \n";  insert(a);  //insert values in the matrix a  cout<<"\n b: \n";  insert(b);    //dividing single 4x4 matrix into four 2x2 matrices  a11=cal11(a);  a12=cal12(a);  a21=cal21(a);  a22=cal22(a);  b11=cal11(b);  b12=cal12(b);  b21=cal21(b);  b22=cal22(b);    //assigning variables acc. to strassen's algo  p=(a11+a22)\*(b11+b22);  q=(a21+a22)\*b11;  r=a11\*(b12-b22);  s=a22\*(b21-b11);  t=(a11+a12)\*b22;  u=(a11-a21)\*(b11+b12);  v=(a12-a22)\*(b21+b22);  //outputting the final matrix  cout<<"\n final matrix";  cout<<"\n"<<p+s-t+v<<" "<<r+t;  cout<<"\n"<<q+s<<" "<<p+r-q+u;  return 0;  }  **Output:**  a:  1 5 3 7  4 2 6 2  7 2 7 2  9 2 6 2  b:  5 4 2 6  4 6 6 1  5 4 2 6  7 1 4 7  Final matrix:  1440 2072  1680 1444 |