

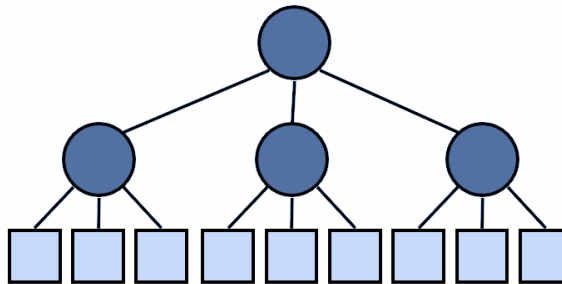
Divide and Conquer

In computer science, divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Divide and Conquer is an algorithmic approach that primarily employs recursion. Some can be solved using iteration.

Divide and conquer steps:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem
2. **Conquer** the subproblems by solving them recursively
3. **Combine** the solutions to the subproblems into the solution for the original problem
4. The **base case** for the recursion is subproblems of constant size



A common approach to solve a problem using divide and conquer

(Suppose, you want to find the count of even numbers in an array **Arr** of **N** integers)

- First, define the function similar to following

```
int countEven( int Arr[], int i, int j ) {
}
```

- Second, identify the **base case**. A base case is the given problem in a smaller size such that it can be solved without any computation.
 - For example, for the count-even-numbers problem, if you have to count the even numbers of an array of size 1 (e.g. **Arr**={3}), then it's easy. if that one element is an even then you return 1, else 0. Therefore, now the code looks like following

```
int countEven( int Arr[], int i, int j ) {
    if (i==j) { // array size 1
        if ( Arr[i] % 2 == 0 ) return 1;
        else return 0;
    }
}
```

- Next, for solving the problem when the base case scenario does not occur, suppose if you already know the solution of a problem of size $N/2$. Now, can you find the solution of a bigger problem of size N ?
 - For example, suppose, you already know the count of even numbers in the first half of the array which is 0, and you also know the count of the even numbers in the second half of the array which is 2. Then, can you tell me the count of even numbers in the whole array?
 - Yes, you can. It is $0+2=2$.
 - So, how can we actually know the count of even numbers in the first half and the second half? (**Divide and Conquer**)
 - Using recursion

```
int countEven( int Arr[], int i, int j ) {
    if (i==j) { // array size 1
        if ( Arr[i] % 2 == 0 ) return 1;
        else return 0;
    } else {
        int mid = (i+j)/2;
        int c1 = countEven(Arr, i, mid); // solution to the first half
        int c2 = countEven(Arr, mid+1, j); // solution to the second half
    }
}
```

- Now, you just have to **combine** the solutions together

```
int countEven( int Arr[], int i, int j ) {
1.  if (i==j) { // array size 1
2.      if ( Arr[i] % 2 == 0 ) return 1;
3.      else return 0;
4.  } else {
5.      int mid = (i+j)/2;
6.      int c1 = countEven(Arr, i, mid); // solution to the first half
7.      int c2 = countEven(Arr, mid+1, j); // solution to the second half
8.      return c1+c2;
9.  }
}
```

Note that not all divide and conquer algorithms follow the exact same pattern, e.g. quicksort.

Practice problems:

PROBLEM 01.

Write a function `print_odd` using divide-and-conquer algorithm to print the odd numbers of an array of n integers.

PROBLEM 02.

Write a function `calc_sum` using divide-and-conquer algorithm to calculate the sum of an array of n integers.

PROBLEM 03.

Write a function `calc_sum` using divide-and-conquer algorithm to calculate the sum of the even numbers of an array of n integers.

PROBLEM 04. X^Y

Write a program that takes X and Y as input and calculates the value of X^Y using *divide and conquer* and prints it.

sample input	sample output
3 7	2187

Hint:

$$3^{100} = 3^{50} \cdot 3^{50} = (3^{\frac{100}{2}}) \cdot (3^{\frac{100}{2}})$$

$$3^{50} = 3^{25} \cdot 3^{25}$$

$$3^{25} = 3^{12} \cdot 3^{12} \cdot 3$$

$$3^{12} = 3^6 \cdot 3^6$$

$$3^6 = 3^3 \cdot 3^3$$

$$3^3 = 3^1 \cdot 3^1 \cdot 3$$

$$3^1 = 3^0 \cdot 3^0 \cdot 3$$

PROBLEM 05. Find the max and min element of an array.

Write a program that does the following

- (i) take **N** numbers as input and store them in an array **A**
- (ii) write a function **findMaxMin** that returns the maximum and minimum elements of an array *using divide and conquer*.
- (iii) use the function **findMaxMin** to print the maximum and minimum elements of the array **A**

sample input	sample output
6 34 -1.5 5 6 -50.1 -6	max: 34.0 min: -50.1

Pseudocode:

Note that following pseudocode assumes that **indexing starts from 1**

For-loop version	Divide and Conquer version:
Function MaxMin(A): 1. fmax = A[1] 2. fmin = A[1] 3. for i=2 to n do 4. if A[i] > fmax 5. then fmax = A[i] 6. if A[i] < fmin 7. then fmin = A[i] 8. end for 9. return fmax, fmin	Function RMaxMin(A, i, j): 1. if i==j then 2. return A[i], A[i] 3. else 4. mid = (i+j)/2 5. max1, min1 = RMaxMin(A, i, mid) 6. max2, min2 = RMaxMin(A, mid+1, j) 7. fmax = max(max1, max2) 8. fmin = min(min1, min2) 9. end if 10. return fmax, fmin

PROBLEM 06. Binary Search

Write a function **binary_search** that finds the index of an element **X** in a sorted (ascending) array **A** of **N** integers *using divide and conquer*. If the element **X** is not present in the array, return -1.

Write a **main** that takes the array **A** and an integer **X** as input from the user. After that, sort the array **A** using the quicksort algorithm and find the index of **X** in **A** using the function **binary_search** and print it.

sample input	sample output
5 3 4 5 7 2 4	4 found in index 2
5 3 4 5 7 2 14	14 not found

Pseudocode:

Note that following pseudocode assumes that the array is sorted in ascending order.

```
Function BinarySearch( A, i, j, X) {
1.  if i==j then
2.      if X == A[i] then return i
3.      else return NOT_FOUND
4.  else
5.      mid = (i+j)/2
6.      if X == A[mid] then return mid
7.      else if X < A[mid] then
8.          return BinarySearch(A, i, mid-1, X)
9.      else // if X > A[mid]
10.         return BinarySearch(A, mid+1, j, X)
11.  end if
}
```

More concise version:

```
Function BinarySearch( A, start, end, X) {
1.  if start <= end then
2.      mid = (start + end)/2
3.      if x==A[mid] then
4.          return mid
5.      else if x < A[mid] then
6.          return BinarySearch(A, start, mid-1, X)
7.      else // if A[mid] < x
8.          return BinarySearch(A, mid+1, end, X)
9.  else
10.     return NOT_FOUND // base case
11.  end if
}
```

PROBLEM 07. Merge sort

Write a function **merge_sort** that sorts an array of **N** numbers in **descending** order *using merge sort*.

Write a **main** that takes **N** numbers as input from users into an array, sorts the array in **descending** order using the function **merge_sort**, and prints the sorted array.

sample input	sample output
4 3 7 5 -1	7 5 3 -1

Pseudocode (ascending order): Note that following pseudocode assumes that **indexing starts from 1**

MERGE-SORT(A, p, r)

```

1  if  $p < r$ 
2    then  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3         MERGE-SORT( $A, p, q$ )
4         MERGE-SORT( $A, q + 1, r$ )
5         MERGE( $A, p, q, r$ )

```

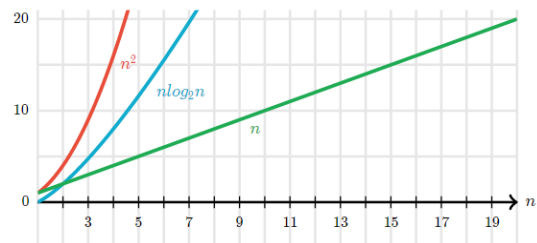
MERGE(A, p, q, r)

```

1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
4  for  $i \leftarrow 1$  to  $n_1$ 
5    do  $L[i] \leftarrow A[p + i - 1]$ 
6  for  $j \leftarrow 1$  to  $n_2$ 
7    do  $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow \infty$ 
9   $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13   do if  $L[i] \leq R[j]$ 
14     then  $A[k] \leftarrow L[i]$ 
15          $i \leftarrow i + 1$ 
16   else  $A[k] \leftarrow R[j]$ 
17          $j \leftarrow j + 1$ 

```

Algorithm	Worst-case running time	Best-case running time	Average-case running time
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$



[Divide and conquer algorithms \(article\) | Khan Academy](#)

PROBLEM 08. Count Inversion [Ref]

If $i < j$ and $A[i] > A[j]$, then the pair $(A[i], A[j])$ is called an **inversion** of an array **A**. The sequence 8, 4, -1, 2, 5 has 6 inversions: (8,4), (8,-1), (8,2), (8,5), (4,-1), (4,2). The sequence 2, 4, 1, 3, 5 has 3 inversions (2,1), (4,1), (4,3).

Write a function **count_inversion** that counts the inversions in an array of **N** numbers *using divide and conquer*. Write a main function that takes **N** numbers from users and uses the function **count_inversion** to count the number of inversions and print it.

sample input	sample output
5 8 4 -1 2 5	#inversions: 6
7 1 20 6 4 5 8 4	#inversions: 10
10 1 20 6 4 5 8 4 6 2 5	#inversions: 23

Hint: The solution is similar to merge-sort. Merge two sorted lists into one output list, but while doing so, we also count the inversion (in line 16 of the *merge* function in the pseudocode).

PROBLEM 09. Quick Sort

The quick sort uses divide and conquer just like merge sort but without using additional storage. The steps are following:

1. Select an element q , called a pivot, from the array. In this algorithm we have chosen the last index as the pivot.
2. The PARTITION function finds the location of the pivot in such a way that all the elements smaller than the pivot are on the left side and all the elements on the right-hand side of the pivot are greater in value. (Items with equal values can go either way).
3. Recursively call the QUICKSORT function which performs quicksort on the array on the left side of the pivot and then on the array on the right side, thus dividing the task into sub tasks. This is carried out until the arrays can no longer be split.

Pseudocode (ascending order):

```

QUICKSORT( $A, p, r$ )
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )

PARTITION( $A, p, r$ )
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 

```

Write a function **quick_sort** that sorts an array of **N** numbers in **descending** order *using quicksort*. Write a **main** that takes **N** numbers as input from users into an array, sorts the array in **descending** order using the function **quick_sort**, and prints the sorted array.

sample input	sample output
4 3 7 5 -1	7 5 3 -1
9 45 341 -1 45 3 31 -13 -134 5	341 45 45 31 5 3 -1 -13

PROBLEM 10. Maximum-sum subarray [\[Ref\]](#)

The maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array **A** of **N** numbers. Note that the array might contain negative or non-negative numbers. For example, for the array of values $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$, the contiguous subarray with the largest sum is $[4, -1, 2, 1]$, with sum 6.

Some properties of this problem are:

1. If the array contains all non-negative numbers, then the problem is trivial; a maximum subarray is the entire array.
2. If the array contains all non-positive numbers, then a solution is any subarray of size 1 containing the maximal value of the array (or the empty subarray, if it is permitted).
3. Several different sub-arrays may have the same maximum sum.

Write a function **find_max_sum_subarray** that finds the maximum sum subarray of an array **A** of **N** integers *using divide and conquer*. Write a **main** that takes the array **A** as input from the user and prints the maximum sum subarray and its sum using the function **find_max_sum_subarray**.

sample input	sample output
9 -2 1 -3 4 -1 2 1 -5 4	4 -1 2 1 sum 6
6 4 -10 6 -1 2 -3	6 -1 2 sum 7

Pseudocode:

```

1  Function FIND-MAXIMUM-SUBARRAY (A, low, high):
2
3      if high == low /// base case: only one element
4          return (low, high, A[low])
5      else
6          mid = (low+high)/2
7
8          left-low, left-high, left-sum = FIND-MAXIMUM-SUBARRAY(A, low, mid)
9          right-low, right-high, right-sum = FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
10         cross-low, cross-high, cross-sum = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
11
12         if left-sum >= right-sum and left-sum >= cross-sum
13             return (left-low, left-high, left-sum)
14         else if right-sum >= left-sum and right-sum >= cross-sum
15             return (right-low, right-high, right-sum)
16         else
17             return (cross-low, cross-high, cross-sum)

```

```
1  Function FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high):
2  |
3  //Find a maximum subarray of the form A[i..mid]
4  left-sum = -∞
5  sum = 0
6  for i = mid downto low
7      sum = sum + A[i ]
8      if sum > left-sum
9          left-sum = sum
10         max_left = i
11 //Find a maximum subarray of the form A[mid + 1 .. j ]
12 right-sum = -∞
13 sum = 0
14 for j = mid +1 to high
15     sum = sum + A[j]
16     if sum > right-sum
17         right-sum = sum
18         max_right = j
19 //Return the indices and the sum of the two subarrays
20 return (max_left, max_right, left-sum + right-sum)
```

PROBLEM 11. Longest common prefix of n strings

Write a program that takes **N** strings from the user and finds the longest common prefix of those strings using *divide and conquer*.

sample input	sample output
3 Algolab Algorithms Algeria	Alg
4 Algolab Algorithms Algeria UIU	No common prefix

PROBLEM 12. Closest pair of points [[Ref1](#), [Ref2](#)]

Write a program that takes N two-dimensional points' coordinates as input, and finds the pair of points that are closest together (according to euclidean distance).

sample input	sample output
6 2 3 12 30 40 50 5 1 12 10 3 4	The closest pair is point (2,3) and point (3,4) Their distance is 1.414
3 1 1 1 2 5 8	The closest pair is point (1,1) and point (1,2) Their distance is 1.000

Pseudocode:

```

Closest-Pair( $p_1, \dots, p_n$ ) {
    Compute separation line L such that half the points
    are on one side and half on the other side.  $O(n \log n)$ 

     $\delta_1 = \text{Closest-Pair}(\text{left half})$   $2T(n/2)$ 
     $\delta_2 = \text{Closest-Pair}(\text{right half})$ 
     $\delta = \min(\delta_1, \delta_2)$ 

    Delete all points further than  $\delta$  from separation line L  $O(n)$ 

    Sort remaining points by y-coordinate.  $O(n \log n)$ 

    Scan points in y-order and compare distance between
    each point and next 11 neighbors. If any of these
    distances is less than  $\delta$ , update  $\delta$ .  $O(n)$ 

    return  $\delta$ .
}

```

Running time:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n \log n)$$

$$T(n) = O(n \log^2 n)$$

Can we achieve $O(n \log n)$?

Yes. Don't sort points in the strip from scratch each time.

- Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
- Sort by merging two pre-sorted lists.
- $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$ $T(n) = O(n \log \log n)$

More practice problems

[Divide and Conquer - LeetCode](#)

Food for thought

1. How to change binary search for descending order? (related to problem 06)
2. How to change merge sort for descending order? (related to problem 07)
3. How to count good pairs? [Suppose, if $i < j$ and $A[i] < A[j]$, then the pair $(A[i], A[j])$ is called a **good pair** of an array A . The sequence 8, 4, -1, 2, 5 has 4 inversions: (4,5), (-1,2), (-1,5), (2,5).] (related to problem 08)
4. How to change quick sort for descending order? (related to problem 09)
5. How to find the minimum-sum subarray? (related to problem 10)
6. Can the furthest pair be found in $O(n \log n)$ time? (related to problem 12)
7. [Can divide and conquer algorithmic problems only be solved using recursion? - Quora](#)

Reference:

- Slides of Dr. Md. Abul Kashem Mia, Professor, CSE Dept, BUET
- [Divide and conquer algorithms \(article\) | Khan Academy](#)
- [Counting Inversions](#)
- [Maximum subarray problem - Wikipedia](#)
- [Chapter 5 - Divide and Conquer](#)
- [CMSC 451: Closest Pair of Points](#)