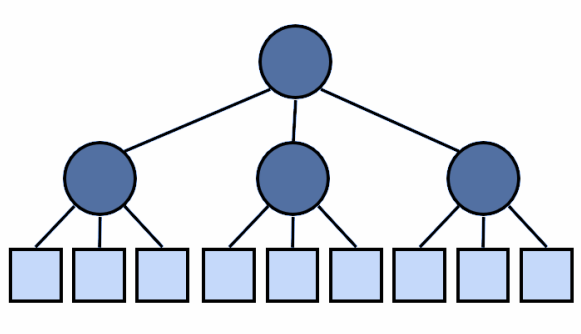
Divide and Conquer

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.

```
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?
    - 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*.

Problem-1: Write a function *print_odd* using divide-and-conquer algorithm to print the odd numbers of an array of n integers.

Problem-2 (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:

$$3^{100} = 3^{50}.3^{50} = (3^{\frac{100}{2}}).(3^{\frac{100}{2}})$$
$$3^{50} = 3^{25}.3^{25}$$
$$3^{25} = 3^{12}.3^{12}.3$$
$$3^{12} = 3^{6}.3^{6}$$
$$3^{6} = 3^{3}.3^{3}$$
$$3^{3} = 3^{1}.3^{1}.3$$
$$3^{1} = 3^{0}.3^{0}.3$$

The problem can be recursively defined by:

power(x, n) = power(x, n / 2) * power(x, n / 2); *// if n is even*

power(x, n) = x * power(x, n / 2) * power(x, n / 2); *// if n is odd*

```
long long power(int x, int y){
    1.  if (y==0) {
    2.          return 1;
    3.  } else {
    4.          int mid = y/2;
    5.          long long p = power(x, mid);
    6.          if (y%2==0) { // power even
    7.              return p*p;
    8.          } else {
    9.              return p*p*x;
    10.         }
    11. }
}
```

Problem-3: 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

For-loop version

```
Function MaxMin(A):
    1.  fmax = A[0]
    2.  fmin = A[0]
    3.  for i=1 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
```

Divide and Conquer version:

```
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-4: Binary Search

Write a function ***binarySearch*** 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 ***binarySearch*** 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

*Note that the 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-5: Merge Sort

Write a function ***mergeSort*** 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 ***mergeSort***, and prints the sorted array.

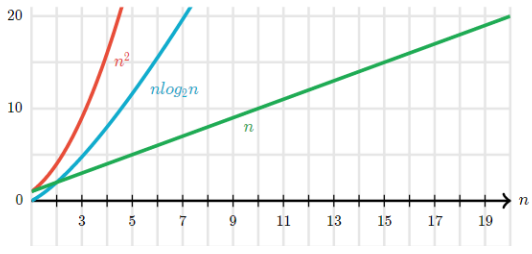
sample input	sample output
4 3 7 5 -1	7 5 3 -1

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 ← ⌊(p + r)/2⌋
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  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← 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

Practice problems:

PP1. Write a function calc_sum using divide-and-conquer algorithm to calculate the sum of an array of n integers.

PP2. Write a function calc_sum using divide-and-conquer algorithm to calculate the sum of the even numbers of an array of n integers.

PP3. 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).

PP4. 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 = 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] ≤ 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

PP5. 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

More practice problems

C/C++ Divide and Conquer Programs - GeeksforGeeks (**Practice Easy and Medium problems**)

Food for thought

1. How to change binary search for descending order?
2. How to change merge sort for descending order?
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).]
4. How to change quick sort for descending order?
5. How to find the minimum-sum subarray?

Can divide and conquer algorithmic problems only be solved using recursion? - Quora