Divide and Conquer

In computer science, *divide and conquer* is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type (similar to the original problem but smaller in size), until these become simple enough to be solved directly. The "**binary search**" and the "**merge sort**" are two well-known divide and conquer algorithms.

Broadly, we can understand divide-and-conquer approach in a three-step process: divide, conquer and combine.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one. In some problems, like binary search, combine part may not be necessary to find the solution.

Binary Search (reducing search space into half)

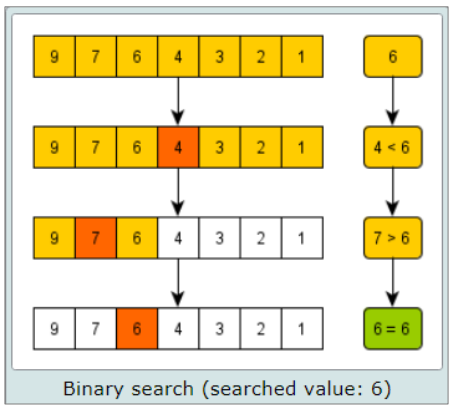
Binary search (half-interval search) is a search technique, which finds a position of an input element within a sorted array. Unlike sequential search, which needs at most O(n) operations, binary search operates in O(lgn) asymptotic complexity. It searches a sorted array by repeatedly dividing the search interval in half.

```
//Binary Seach
#include "pch.h"
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

vector<int> X{ 9, 7, 6, 4, 3, 2, 1};
int target = 6;
//----------------------------------------
//This method can be implemented non-recursively.
int binSearch(int left, int right)
{
	//No more items to search
	if (left > right)
		return -1;

	int middle = (left + right) / 2;  //left + (right-left)/2;
	//Target found. No need to continue.
```

```
        if (X[middle] == target)
                return middle;

        //Target is not on the right side.
        if (X[middle] < target)
                binSearch(left, middle - 1);
        //Target is not on the left side.
        else //if(X[middle] >= target)
                binSearch(middle + 1, right);
}
//----------------------------------------
int main()
{
        int n = X.size();
        int pos = binSearch(0,n-1);
        if (pos >= 0)
                cout << target << " found in the positon " << pos << "."<< endl;
        else
                cout << target << " not found." << endl;
        return 0;
}
```

Output

```
6 found in the positon 2.
```



Binary search (searched value: 6)

Merge Sort (Dividing search space into two halves and combining)

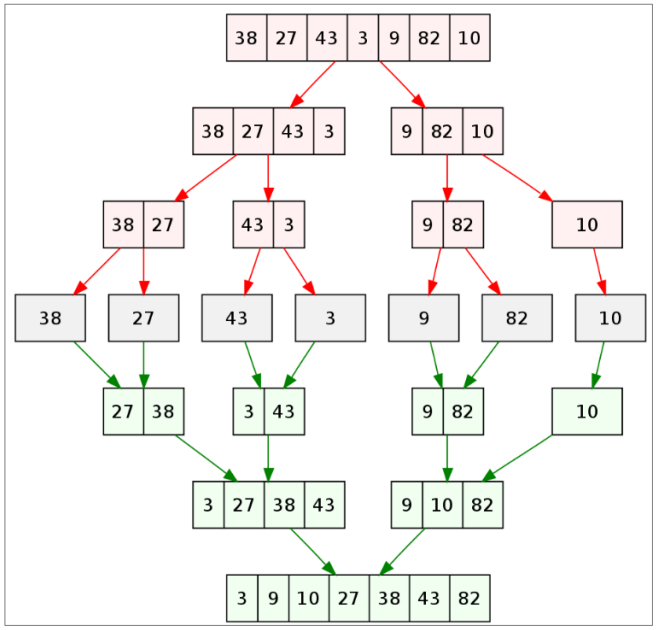
Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub-list consists of a single element and merging those sub-lists in a manner that results into a sorted list.

Idea:

- Divide the unsorted list into N sub-lists, each containing element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sub-lists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list.

This procedure is repeated until both the smaller sub-lists are empty and the new combined sub-list comprises all the elements of both the sub-lists.



A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

Mergesort recurrence.

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. T(n) = $O(n \log_2 n)$.

```
// Merge Sort
#include "pch.h"
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

vector<int> X{ 38, 27, 43, 3, 9, 82, 10 };
//----------------------------------------
//Merge two sorted parts of X
void merge(int left, int right)
{
	vector<int> temp;
	int middle = (left + right) / 2;
	int i = left, j = middle + 1;	//left + (right-left)/2;
	while (i <= middle && j <= right)
	{
		if (X[i] <= X[j])
		{
			temp.push_back(X[i]);
			i++;
		}
		else
		{
			temp.push_back(X[j]);
			j++;
```

```
			}
		}

		//Add the remainig parts. Only one of the loops will be executed.
		for (int k = i; k <= middle; k++)
			temp.push_back(X[k]);
		for (int k = j; k <= right; k++)
			temp.push_back(X[k]);

		//Copy the temp vector (combined part) into the original vector
		copy(temp.begin(), temp.end(), X.begin() + left);
}
//----------------------------------------
//Divide and Conquer mechanism.
void mergeSort(int left, int right)
{
		//base case: subproblem is small enough not to break.
		if (left == right)
			return;

		//Break the problem into two subproblems
		int middle = (left + right) / 2;
		mergeSort(left, middle);
		mergeSort(middle + 1, right);

		//Combine the solutions of subproblems
		merge(left, right);
}
//----------------------------------------
void printVector()
{
		for (int x : X)
			cout << x << " ";
}
//----------------------------------------
int main()
{
		int n = X.size();
		mergeSort(0, n - 1);
		printVector();
		return 0;
}
```

Output

```
3 9 10 27 38 43 82
```

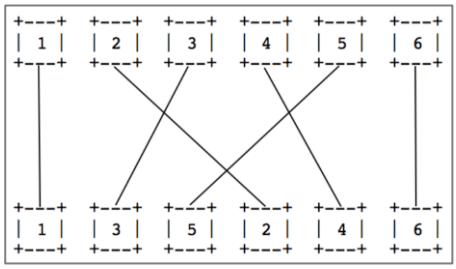
Sample Problem 1: Counting Inversions using Merge Sort

Inversion count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j

Example:

The sequence 1, 3, 5, 2, 4, 6 has three inversions (3, 2), (5, 2), (5, 4).



Merge Sort algorithm can be modified to calculate the number of the inversions in an array. Since the two halves are already sorted, in the combining part of the merge sort the number of the inversions can be calculated in a linear time. Thus, overall complexity will be the same with the merge sort. O(nlgn).

Code:

```
// Counting inversions using merge sort algorithm.
#include "pch.h"
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

vector<int> X{ 1, 3, 5, 2, 4, 6 };
int result;   //global int variables are initialized to zero.
//----------------------------------------
//Merge two sorted parts of X
void merge(int left, int right)
{
	vector<int> temp;
	int middle = (left + right) / 2;
	int i = left, j = middle + 1;			//left + (right-left)/2;
	while (i <= middle && j <= right)
	{
		if (X[i] <= X[j])
		{
			temp.push_back(X[i]);
			i++;
		}
		else
		{
			result += middle - i + 1;  //right element is smaller. Inversions.
			temp.push_back(X[j]);
			j++;
		}
	}

	//Add the remainig part. Only one of the loops will be executed.
	for (int k = i; k <= middle; k++)
		temp.push_back(X[k]);
	for (int k = j; k <= right; k++)
		temp.push_back(X[k]);

	//Copy the temp vector (combined part) into the original vector
	copy(temp.begin(), temp.end(), X.begin() + left);
}
//----------------------------------------
//Divide and Conquer mechanism.
void mergeSort(int left, int right)
{
	//base case: subproblem is small enough not to break.
	if (left == right)
		return;

	//Break the problem into two subproblems
```

```
        int middle = (left + right) / 2;
        mergeSort(left, middle);
        mergeSort(middle + 1, right);

        //Combine the solutions of subproblems
        merge(left, right);
}
//----------------------------------------
int main()
{
        int n = X.size();
        mergeSort(0, n - 1);
        cout << "Number of inversions is "<<result <<"."<< endl;
        return 0;
}
```

Output

```
Number of inversions is 3.
```

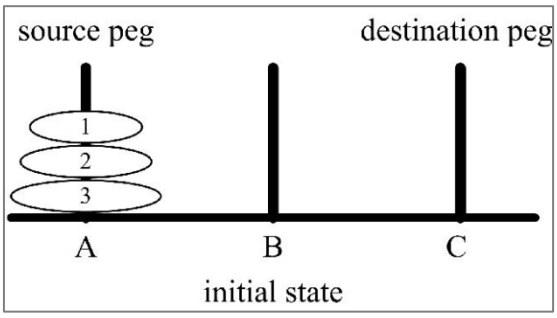
Tower of Hanoi

The Tower of Hanoi is a mathematical puzzle invented by the French mathematician Edouard Lucas in 1883. In the puzzle, we have three pegs and n disks. The objective of the puzzle is to move the entire stack to another peg, obeying the following simple rules:

1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

Solution for n=3:

```
Disk 1 moved from A to C
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C
```



To write a program to solve this problem with Divide and Conquer approach is very simple:

1) Move the top N-1 disks from peg A to peg B (using C as an auxiliary peg) (recursive call)
2) Move the bottom disk from peg A to peg C. (manual operation)
3) Move N-1 disks from Peg B to Peg C (using Peg A as an auxiliary peg). (recursive call)

Code:

```
// Tower of Hanoi.cpp
#include "pch.h"
#include <iostream>

using namespace std;
```

```
//-----------------------------------------------
void toh(char a, char b, char c, int n)
{
	if (n == 1)   //base case
		cout << a << " -> " << c << endl;
	else
	{
		//move n-1 disks from the source peg to the auxiliary peg.
		toh(a, c, b, n - 1);
		//Move the buttom disk from the source peg to the destination peg.
		cout << a << " -> " << c << endl;
		//Move n-1 disks from the auxilary peg to the destination peg.
		toh(b, a, c, n - 1);
	}
}
//-----------------------------------------------
int main()
{
	int N = 3;
	toh('A', 'B', 'C', N);
}
```

Output

```
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

Note: Every recursive program can be implemented without using recursion. Try to implement a non-recursive version of the Tower of Hanoi program.

Practice Problems

1. **Bitonic Point:** You are given a bitonic sequence, the task is to find Bitonic Point in it. A Bitonic Sequence is a sequence of numbers which is first strictly increasing then after a point strictly decreasing.
2. **Median:** Find the median of two sorted arrays.
3. **The Biggest Rectangle:** An axis-aligned rectangle is given. There are n points inside the rectangle. Find the area of the largest axis-aligned rectangle (inside the first rectangle) that doesn't contain any points. All coordinates are integer.
4. **Closest Pair of Points**: Given n points in the plane, find a pair with smallest Euclidean distance between them.
5. **Quick Sort:** Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Useful Links and References

https://en.wikipedia.org › wiki › Divide-and-conquer_algorithm
http://www.programming-algorithms.net/article/40119/Binary-search
https://homes.cs.washington.edu/~jrl/teaching/cse312au10/lec24.pdf
https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/tutorial/
https://www.geeksforgeeks.org/counting-inversions/
https://www.includehelp.com/data-structure-tutorial/tower-of-hanoi-using-recursion.aspx