**Department of Computer Science**

**Design and Analysis of Algorithm**

**CSCD 304**

****

**ASSIGNMENT**

**ID:10666676**

**Advantages of divide and conquer algorithms**

### **Solving difficult problems**

Divide is best in solving difficult problems, all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem.

**Efficiency**

The divide-and-conquer algorithm often helps in the discovery of efficient algorithms. It was the key to finding more efficient algorithms that exist these times which are efficient. The efficiency of divide and conquer is unique and for all distinct values, it works perfectly.

### **Parallelism**

Divide-and-conquer algorithms are naturally adapted for execution in multi-processor machines, especially [shared-memory](https://en.wikipedia.org/wiki/Shared-memory" \o "Shared-memory) systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

### **Memory access**

Divide-and-conquer algorithms naturally tend to make efficient use of [memory caches](https://en.wikipedia.org/wiki/Memory_cache" \o "Memory cache). The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory.

### **Round-off control**

In computations with rounded arithmetic, e.g. with [floating-point](https://en.wikipedia.org/wiki/Floating-point" \o "Floating-point) numbers, a divide-and-conquer algorithm may yield more accurate results than a superficially equivalent iterative method.

**Limitations of the divide and conquer algorithm.**

**Recursion slow rate**

One of the most common issues with this sort of algorithm is the fact that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process

**Iterative approach**

Sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n.  
In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together.

**Base case**

Choosing base cases is also a limitation as picking small cases may work if the cases are taken much higher than the capacity of the system than problems may occur.

**Time and space anomaly**

Sometimes a case where the problem when broken down results in same sub-problems which may needlessly increase the solving time and extra space may be consumed.

**Using the divide and conquer method to recursively implement the binary search.**

In the implementation, we will be using java to demonstrate how the implementation is done.

// Java implementation of recursive Binary Search

class BinarySearch {

// Returns index of x if it is present in arr[l..

// r], else return -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;

}

// Driver method to test above

public static void main(String args[])

{

BinarySearch ob = new BinarySearch();

int arr[] = { 2, 3, 4, 10, 40 };

int n = arr.length;

int x = 10;

int result = ob.binarySearch(arr, 0, n - 1, x);

if (result == -1)

System.out.println("Element not present");

else

System.out.println("Element found at index " + result);

}

}