1 ASSIGNMENT NO: A1

Author: Ameeth Kanawaday
Roll No: 4430

2 Problem Definition

Using Divide and Conquer Strategies design a function for Binary Search using C++/ Java/ Python/Scala.

3 Learning Objectives:

1. To understand Divide and Conquer strategy.
2. To use Divide and Conquer for implementing binary search.

4 S/W and H/W requirements:

1. Open source 64 bit OS.
2. Gedit text editor.
3. Scala

5 Theory

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, 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.

This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).

Understanding and designing Divide and Conquer algorithms is a complex skill that requires a good understanding of the nature of the underlying problem to be solved. As when proving a theorem by induction, it is often necessary

to replace the original problem with a more general or complicated problem in order to initialize the recursion, and there is no systematic method for finding the proper generalization. These Divide and Conquer complications are seen when optimizing the calculation of a Fibonacci number with efficient double recursion.
The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

Binary Search algorithm
Given an array A of n elements with values or records A0...An-1, sorted such that A0 ¡=...¡= An-1, and target value T, the following subroutine uses binary search to find the index of T in A.

1. Set L to 0 and R to n-1.
2. If L ¿ R, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the floor of (L+R)/2.
4. If Am ¡ T, set L to m+1 and go to step 3.
5. If Am ¿ T, set R to m-1 and go to step 3.
6. Now Am = T, the search is done; return m.

6 Related Mathematics

Let S be the solution perspective of the given problem.
The set S is defined as:
$S = \{ s, e, X, Y, F, DD, NDD | \varnothing_s \}$
Where,
s= Start point
e= End point
F= Set of main functions
DD= set of deterministic data
NDD= set of non deterministic data

X= Input Set.
$X = \{ < x_1, x_2, ..., x_n >, N \}$
where,
$x_i \in I$
N = no. to be searched

Y = $\{N, i\}$
where, i= index of N in array

s = $\{X\}$
A = $< x_1, x_2, ..., x_n >$
e = Search complete.

F = $\{f_{split}, f_{compare}, f_{findmid}\}$

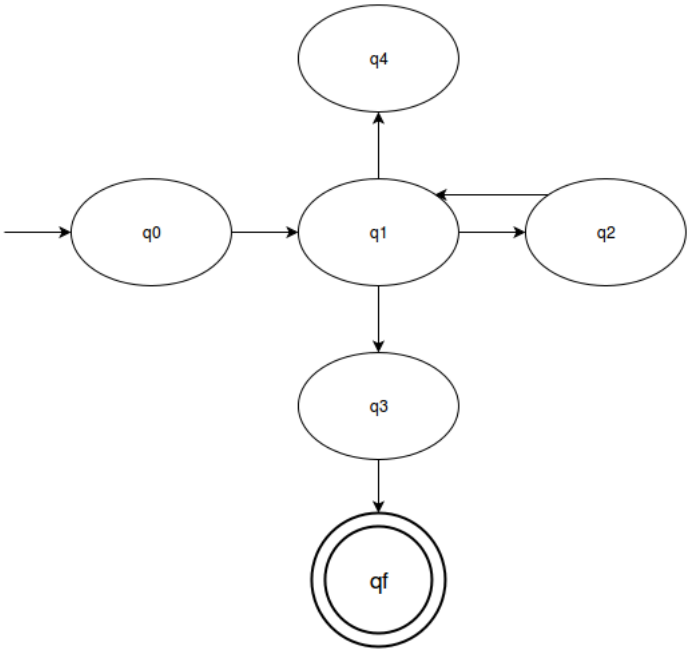
f_{split} :function to divide the array from mid element.

$f_{compare}$:function to compare the mid element with N.

$f_{findmid}$:function to find the mid element of the array or subarray.

DD = $\{A, N\}$
NDD = ϕ

7 State Diagram



q0 = input element state
q1 = comparision with mid element
q2 = array dividde state
q3 = element found state
q4 = element not found state
qf = final state

8 Program

```
import scala.annotation.tailrec
object BinarySearchApp{

  def main(args: Array[String]){
    val l = List(1,2,4,5,6, 8,9,25,31);
    if(search2(5, l) == Some(5))
println("Found 5");
    else
println("Not found!");
    if(search2(6, l) == Some(6))
println("Found 6");
    else
println("Not found!");
    if(search2(7, l) == Some(7))
println("Found 5");
    else
println("Not found!");
  }

  def search(target:Int, l:List[Int]) = {
    @tailrec
    def recursion(low:Int, high:Int):Option[Int] = (low+high)/2 match{
      case _ if high < low => None
      case mid if l(mid) > target => recursion(low, mid-1)
      case mid if l(mid) < target => recursion(mid+1, high)
      case mid => Some(mid)
    }
    recursion(0,l.size - 1)
  }

  def search2(target:Int, l:List[Int]) = {
    def recursion(mid:Int, list:List[Int]):Option[Int]=  list match {
      case tar::Nil if tar == target => Some(tar)
      case tar::Nil => None
      case ls => {
          val(lows,highs) = ls.splitAt(mid)
          if (ls(mid)>target)
            recursion((lows.size)/2, lows)
          else
            recursion((highs.size)/2, highs)
        }
    }

    recursion((l.size)/2, l);
```

```
  }
}
```

```
OUTPUT:
ameeth@ubuntu-16.0.4:~/CL1$ scala a1.scala
Found 5
Found 6
Not found!
ameeth@ubuntu-16.0.4:~/CL1$
```

9 Conclusion

Thus we have implemented the Divide and Conquer Binary Search function in Scala.