Assignment A1

Problem Statement:

Using Divide and Conquer Strategies and object-oriented software design technique using Modelio to design a software function for Binary Search for an unordered data stored in memory. Use necessary USE-CASE diagrams and justify its use with the help of mathematical modeling and related efficiency. Implement the design using Eclipse C++ or python.

Learning Objectives:

1. To understand Divide Conquer strategy.
2. Learn how to implement Binary Search as an example of Divide Conquer strategy.
3. Learn about UML modeling in Modelio Software.

Learning Outcomes:

1. Learnt about Divide Conquer strategy.
2. Implemented Binary Search algorithm.
3. Learnt UML modeling in Modelio Software.

Software and Hardware Requirements:

1. Latest 64-BIT Version of Linux Operating System
2. Eclipse with Python
3. Modelio Software

Theory

Divide and Conquer Strategies

- In computer science, divide and conquer (DC) 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 (divide), until these become simple enough to be solved directly (conquer). 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), syntactic analysis (e.g. top-down parsers) and computing the discrete Fourier transform (FFTs).

- Understanding and designing DC 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 DC complications are seen when optimizing the calculation of a Fibonacci number with efficient double recursion.

Advantages

1. Solving Difficult Problems : Divide and conquer is a powerful tool for solving conceptually 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. Similarly, decrease and conquer only requires reducing the problem to a single smaller problem.

2. Algorithm Efficiency : The divide and conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsubas fast multiplication method, the quick sort and merge sort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms. In all these examples, the DC approach led to an improvement in the asymptotic cost of the solution. For example, if the base cases have constant-bounded size, the work of splitting the problem and combining the partial solutions is proportional to the problems size n, and there are a bounded number p of subproblems of size n/p at each stage, then the cost of the divide and conquer algorithm will be O(n log n).

3. Parallelism : Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned 5 in advance, because distinct sub-problems can be executed on different processors.

4. Memory Access : Divide and conquer algorithms naturally tend to make efficient use of memory caches. 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. An algorithm designed to exploit the cache in this way is called cache-oblivious, because it does not contain the cache size(s) as an explicit parameter. Moreover, DC algorithms can be designed for important algorithms (e.g., sorting, FFTs, and matrix multiplication) to be optimal cache-oblivious algorithms they use the cache in a probably optimal way, in an asymptotic sense, regardless of the cache size. In contrast, the traditional approach to exploiting the cache is blocking, as in loop nest optimization, where the problem is explicitly divided into chunks of the appropriate size this can

also use the cache optimally, but only when the algorithm is tuned for the specific cache size(s) of a particular machine. The same advantage exists with regards to other hierarchical storage systems, such as NUMA or virtual memory, as well as for multiple levels of cache: once a sub-problem is small enough, it can be solved within a given level of the hierarchy, without accessing the higher (slower) levels.

5. Roundoff : Control In computations with rounded arithmetic, e.g. with floating point numbers, a divide and conquer algorithm may yield more accurate results than a superficially equivalent iterative method. For example, one can add N numbers either by a simple loop that adds each datum to a single variable, or by a DC algorithm called pairwise summation that breaks the data set into two halves, recursively computes the sum of each half, and then adds the two sums. While the second method performs the same number of additions as the first, and pays the overhead of the recursive calls, it is usually more accurate.

Binary Search

- In computer science, a binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomic divide-and-conquer search algorithm and executes in logarithmic time.

- The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle elements value, then the position is returned and the search is finished. If the target value is less than the middle elements value, then the search continues on the lower half of the array; or if the target value is greater than the middle elements value, then the search continues on the upper half of the array. This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and not found is returned).

Complexity Analysis

Worst case performance O(log n)
Best case performance O(1)
Average case performance O(log n)
Worst case space complexity O(1)

Mathematical Model

Let S be the system of solution set for given problem statement such that,
S = { s, e, X, Y, F, DD, NDD, Su, Fu }
where,

s = start state
such that, y = { }
e = end state
such that, y = { pos }
where, pos = position of found element & $0 \leq pos \leq n$
X = set of input
such that X = { X1, X2 }
where,
X1 = Set of integer elements
such that, X1 = { x1, x2, x3.... xn }
X2 = Integer key to be found
Y = set of output
such that Y = { pos }
where, pos = position of found element & $0 \leq pos \leq n$
F = set of function
such that F = { f1, f2, f3, f4 }
where,
f1 = function to take elements from user
f2 = function to sort entered elements
f3 = function to take search element from user
f4 = function to search element in sorted array using divide and conquer strategy
DD = Deterministic data
DD = |X|
NDD = Nondeterministic data
NDD = pos i.e value of position of search element
Su = Success case
If element is present in array then value of pos is in the range of 1 to n where, n is number of element present in the array
Fu = Failure case
If element is not present in array then value of pos is not defined

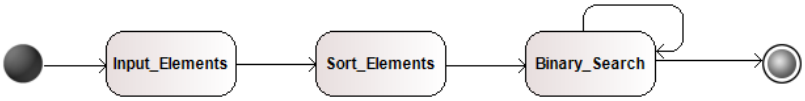
Pseudo code

```
Procedure binary_search
        A ← sorted array
        n ← size of array
        x ← value to be searched

        Set lowerBound = 1
        Set upperBound = n

        while x not found
                if upperBound < lowerBound
                        EXIT: x does not exists

                set midPoint=lowerBound + (upperBound−lowerBound)/2

                if A[midPoint] < x
                        set lowerBound = midPoint + 1

                if A[midPoint] > x
                        set upperBound = midPoint − 1

                if A[midPoint] = x
                        EXIT: x found at location midPoint
        end while

end procedure
```

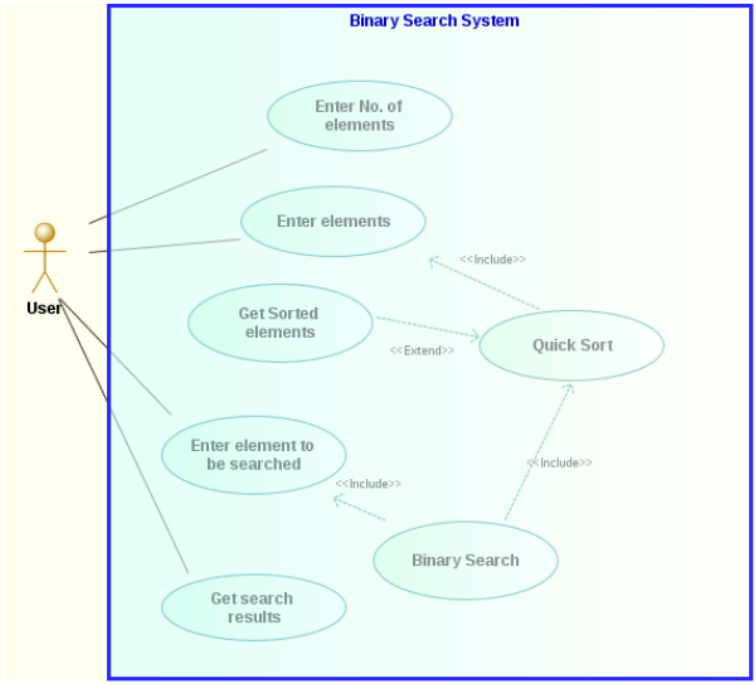
Algorithm

1. Start
2. Declare an array of a specified size.
3. Input the no. of array element.
4. Input the element of array.
5. Check the value of no. of element if value is less than 0 goto 3. otherwise goto 6.
6. Sort the element in array.
7. Enter the element which we want to search.
8. Search the element in array using Divide and Conquer strategies.
9. If the value of position is not -1 goto 10 else goto 11.
10. Print the search successfull.
11. Print the search unsuccessfull.

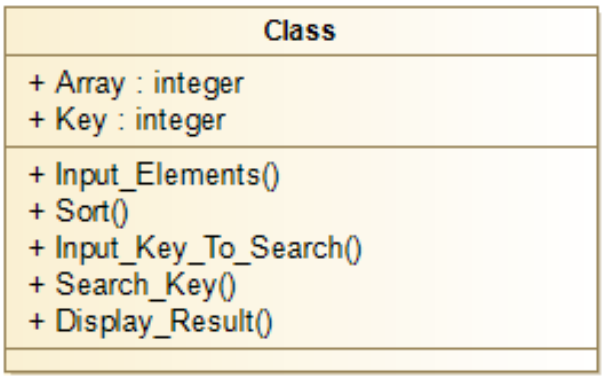
State diagram



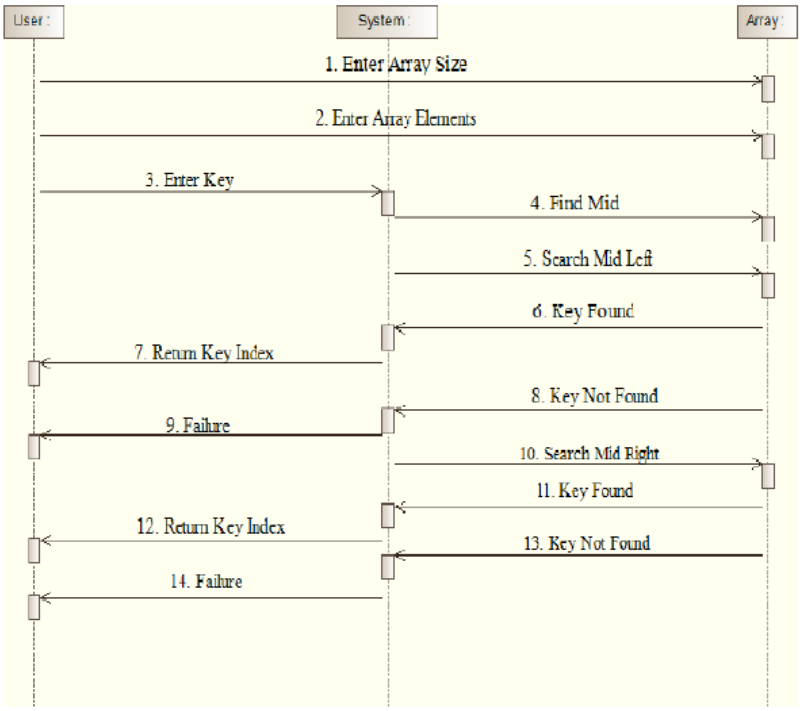
Use-case diagram



Class diagram



Sequence diagram



Positive Testing

Positive testing is a testing technique to show that a product or application under test does what it is supposed to do. Positive testing verifies how the application behaves for the positive set of data.

1. Array size to be specified in integer values, within specified limits.
2. Array elements should be integer values.

Sr. No.	Test Condition	Steps to be executed	Expected Result	Actual Result
1.	Enter the key to be find	Press Enter	Display	Display position of key
2.	Find the key at particular position	Press Enter	Position of key	Display position of key

Negative Testing

Negative testing ensures that your application can specifically handle invalid input or unexpected user behavior.

1. Size of array which exceeds its limit.
2. Any data type other than integer for array size and elements.

Sr. No.	Test Condition	Steps to be executed	Expected Result	Actual Result
1.	Entered key is in array or not	Give key	If not display error message	Same as Expected
2.	Press Enter without specifying key	Press Enter	Error message	Same as Expected

Conclusion

We have successfully implemented binary search using divide and conquer technique.