

Assignment No : Group A-1

Date: /06/2015 to /06/2015

Title : Using Divide and Conquer Strategies design a function for Binary Search using C++

Name : Aditi Tripathy

Roll no : 423069

Class : BE C Batch : C4

Remarks:

# 1 Problem Statement

Use Divide and Conquer Strategies to design a function for Binary Search using C++.

## 2 Objectives

1. To study Divide and Conquer strategies.
2. To implement binary search using recursive techniques

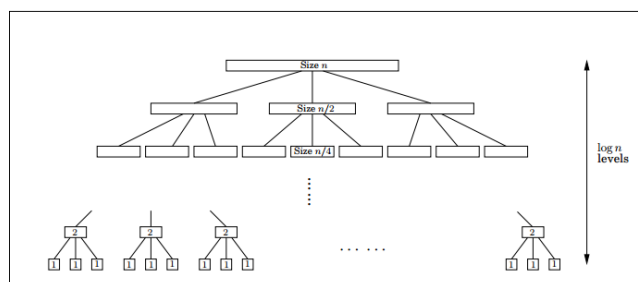
## 3 Theory

### 3.1 Divide And Conquer

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem.

Little more formally, divide-and-conquer paradigm consists of following major phases:

1. Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
2. Solve the sub-problem recursively (successively and independently), and then
3. Combine these solutions to subproblems to create a solution to the original problem.



Divide-and-conquer algorithms often follow a generic pattern: they tackle a problem of size  $n$  by recursively solving, say,  $a$  subproblems of size  $n/b$  and then combining these answers in  $O(n^d)$  time, for some  $a, b, d$  greater than 0 their running time can be captured by the equation

$$T(n) = aT(n/b) + O(n^d)$$

We next derive a closed-form solution to this general recurrence so that we no longer have to solve it explicitly in each new instance.

### 3.2 Binary Search

The binary search algorithm is a method of searching a sorted array for a single element by cutting the array in half with each recursive pass. The trick is to pick a midpoint near the center of the array, compare the data at that point with the data being searched and then, responding to one of three possible conditions:

1. the data is found at the midpoint
2. the data at the midpoint is greater than the data being searched for
3. the data at the midpoint is less than being searched

Recursion is used in this algorithm because with each pass, a new array is created cutting the old one in half. The binary search procedure is then called recursively, this time, on the new (and smaller) array. Typically the array size is adjusted by manipulating a beginning and ending index. The algorithm exhibits a logarithmic order of growth because essentially it divides the problem domain in half in each pass.

## 4 Mathematical Modelling

Let  $S$  be the solution perspective of the class BinarySearch such that

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

$s = \{\text{Initial state of the the Binary Search class}\}$

$e = \{\text{End state or destructor of the class}\}$

$i = \{\text{Input of the system}\}$

$o = \{\text{Output of the system}\}$

$DD = \{\text{Deterministic data: it helps identifying the load store functions or assignment functions}\}$

$NDD = \{\text{Non deterministic data: data of the system } S \text{ to be solved}\}$

$Success = \{\text{Desired outcome generated}\}$

$Failure = \{\text{Desired outcome not generated or forced exit due to system error}\}$

#### For class BinarySearch

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

$s = \{\text{Initial state of the the Binary Search class}\}$

$e = \{\text{End state or destructor of the class}\}$

$i = \{I1, I2, I3\}$

$I1 = \{x - x \text{ is the max size of the array}\}$

$I2 = \{x - x \text{ is the elements of the array}\}$

$I3 = \{x - x \text{ is the key to be searched}\}$

$o = \{o1, o2\}$

$o1 = \{x - x \text{ is the position of the element found}\}$

$o2 = \{x - x \text{ is the error message when element is not found}\}$

$f = \{\text{Sort}(), \text{BSearch}(), \text{Display}()\}$

$\text{Sort}() = \{x - x \text{ is the function to sort elements}\}$

$\text{BSearch}() = \{x - x \text{ is the function to search the key}\}$

$\text{Display}() = \{x - x \text{ is the function to display the elements}\}$

## 5 Algorithm

1. Start.
2. Initialise  $a[n]$ ,  $first=0$ ,  $last=n-1$
3. Get key to search
4. Set  $mid = (first+last)/2$
5. If first greater than last :  
    set  $mid = -1$   
    goto step 8
6. If  $a[mid]==key$   
    goto step 8
7. else if  $a[mid]$  greater than key  
    set  $last=mid-1$   
    goto step 4  
    else  
    set  $first=mid+1$   
    goto step 4
8. Print position as mid
9. Stop.

## 6 Code

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int size,excpption=0;
class Search{
int *num;
public:
Search();
void Sort();
int BSearch(int first,int last,int key);
void Display();
};
//
Search::Search(){
try{
int i,temp,j,flag = 0;
cout<<"\nPlease enter total number of elements : ";
cin>>size;
if(cin.fail())
throw size;
num = new int[size];
for(i=1;i<=size;i++){
flag=0;
temp = rand()%size;
for(j=1;j<i;j++){
if(temp==num[j]){
i--;
flag = 1;
break;
}
if(flag!=1)
num[i] = temp;
}
}
catch(...){
cout<<"\nException caught!Please enter only integers.\n";
excpption = 1;
}
}

void Search::Display(){
for(int i = 1;i<=size;i++)
cout<<"\t"<<num[i];
}

void Search::Sort(){
int i,j,temp;
for(i=1;i<=size;i++)
```

```

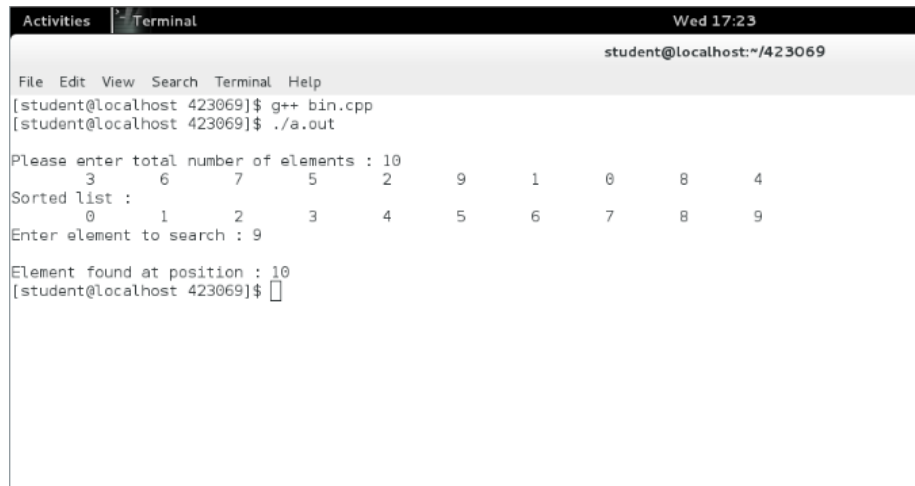
for(j=2;j<=size;j++)
if(num[j]<num[j-1]){
temp = num[j-1];
num[j-1] = num[j];
num[j] = temp;
}
}

int Search::BSearch(int first,int last,int key){
int mid = (last+first)/2;
if(num[mid]==key)
return mid;
else{
if(first>last)
return 0;
else{
if(num[mid]>key)
BSearch(first,mid-1,key);
else
BSearch(mid+1,last,key);
}
}
}

int main(){
try{
int key,pos=0;
Search S1;
if(excption)
return 0;
S1.Display();
S1.Sort();
cout<<"\nSorted list : \n";
S1.Display();
cout<<"\nEnter element to search : ";
cin>>key;
if(cin.fail())
throw key;
pos = S1.BSearch(1,size,key);
if(pos)
cout<<"\nElement found at position : "<<pos<<"\n";
else
cout<<"\nElement not found.\n";
return 0;
}
catch(...){
cout<<"\n Exception caught!Please enter only integers.\n";
}
}

```

## 7 Output



```
Activities Terminal Wed 17:23
student@localhost:~/423069

File Edit View Search Terminal Help
[student@localhost 423069]$ g++ bin.cpp
[student@localhost 423069]$ ./a.out

Please enter total number of elements : 10
3      6      7      5      2      9      1      0      8      4
Sorted list :
0      1      2      3      4      5      6      7      8      9
Enter element to search : 9

Element found at position : 10
[student@localhost 423069]$
```

## 8 Testing

### 8.1 Eliminating Redundant Conditional Statements

The range of the loop or the condition imposes a constraint for values of variables or expression in the control blocks enclosed within. Different views of the constraints can be computed by performing a different set of forward substitutions. By composing the different views of the loop headers and conditionals statements, we get different views of the dominating constraints. Consider any conditional statement for which the different views of the dominating constraints are available. By comparing the different views of this conditional with the different views of dominating constraints, we determine if this conditional is redundant. A redundant conditional is simply removed and the statements enclosed inside it are merged with the control block in which the conditional statement was initially placed. Redundant conditional statements can be removed by combining two conditions using logical operators such as OR and AND. Also, directly checking contents of some memory eliminates the need to add extra variables for copying and checking them instead. Removing redundant conditional statements reduces the branching factor of the code, as well as the lines.

### 8.2 Eliminating Redundant Iterative Statements

Consider any loop which encloses an original candidate for placement, and let this candidate be anticipable at the beginning of the first statement enclosed in the loop. We compare all the views of this loop against all anticipable views of the candidate for placement. If either the left-hand-side or the right-hand-side of the expression are identical or separated by a constant, we fold in the

candidate into this loop. That is to say, that if the particular operation can be done without introducing further iterations, then it is good practice to do so. Removal of redundant loops will reduce the time complexity of the program and lead to faster execution.

### 8.3 Black Box Testing

For BlackBox testing, several types of inputs were provided.

1. Even number of elements
2. Odd number of elements
3. Element in the middle of array
4. Element at boundary of array

```
File Edit View Search Terminal Help
aditi@aditi-VirtualBox:~/CL1$ g++ a1.cpp
aditi@aditi-VirtualBox:~/CL1$ ./a.out

Please enter total number of elements : 10
3      6      7      5      2      9      1      0      8      4
Sorted list :
0      1      2      3      4      5      6      7      8      9
Enter element to search : 3

Element found at position : 4
aditi@aditi-VirtualBox:~/CL1$ ./a.out

Please enter total number of elements : 9
1      7      0      5      3      6      4      8      2
Sorted list :
0      1      2      3      4      5      6      7      8
Enter element to search : 3

Element found at position : 4
aditi@aditi-VirtualBox:~/CL1$ ./a.out

Please enter total number of elements : 10
3      6      7      5      2      9      1      0      8      4
Sorted list :
0      1      2      3      4      5      6      7      8      9
Enter element to search : 0

Element found at position : 1
aditi@aditi-VirtualBox:~/CL1$ ./a.out

Please enter total number of elements : 9
1      7      0      5      3      6      4      8      2
Sorted list :
0      1      2      3      4      5      6      7      8
Enter element to search : 8

Element found at position : 9
aditi@aditi-VirtualBox:~/CL1$
```

### 8.4 White Box Testing

White box Testing (known as glass box testing), is used for testing the loops and conditions of the program. It focuses on correct execution rather than fulfilling of requirements. It ensures that each and every statement is executed atleast once. There are two types of white box testing :

1. Basis Path Testing
2. Control Structure Testing



Binary search has been executed using recursion technique. In the program given above, the function BSearch() employs recursion by calling itself over and over again, each time, reducing the search space by half. The code for the same is :

```
1.int BSearch(int first,int last,int key){
2. int mid = (last+first)/2;
3.  if(num[mid]==key)
4.  return mid;
5.  else{
6.  if(first>last)
7.  return 0;
8.  else{
9.  if(num[mid]>key)
10. BSearch(first,mid-1,key);
11. else
12. BSearch(mid+1,last,key);
13. }
14. }
15.}
```

line no. 2 initialises the variable mid, with the index value of middle element in the array.

line no. 3 checks the value of the mid element, if it matches the key element to be searched, it returns the position of mid.

line no. 6 checks for the end of the array, by comparing values of first and last. This is a stopping condition in case the value to be searched is not in the array. In that case, the function returns value 0.

line no. 9 compares the value of the middle element with the key.

If it is found to be greater, the function BSearch() calls itself again, with search space as the initial half of the original array.

If it is lesser, then the function is called with search space as the latter half of the array.

The whole function is executed recursively with three comparisons per recursion, and the sample space reduced to half at each step.

This ensures a logarithmic time complexity.

## 8.5 Positive-Negative Testing

### 1. Positive Testing

For positive testing, all the correct and expected inputs are provided and the result is checked i.e. only integers are provided as the input.

### 2. Negative Testing

For negative testing, unexpected input is provided to check the behaviour of the program. In this case, character is entered in place of integer and the exception is caught successfully, upon which, the program is exited with appropriate message.

```

aditi@aditi-VirtualBox:~/CL1$ g++ a1.cpp
aditi@aditi-VirtualBox:~/CL1$ ./a.out

Please enter total number of elements : 10
3      6      7      5      2      9      1      0      8      4
Sorted list :
0      1      2      3      4      5      6      7      8      9
Enter element to search : 2

Element found at position : 3
aditi@aditi-VirtualBox:~/CL1$ ./a.out

Please enter total number of elements : 10
3      6      7      5      2      9      1      0      8      4
Sorted list :
0      1      2      3      4      5      6      7      8      9
Enter element to search : 10

Element not found.
aditi@aditi-VirtualBox:~/CL1$ ./a.out

Please enter total number of elements : a

Exception caught!Please enter only integers.
aditi@aditi-VirtualBox:~/CL1$

```

## 9 Conclusion

Thus studied Divide and Conquer strategies using Binary Search as an example.