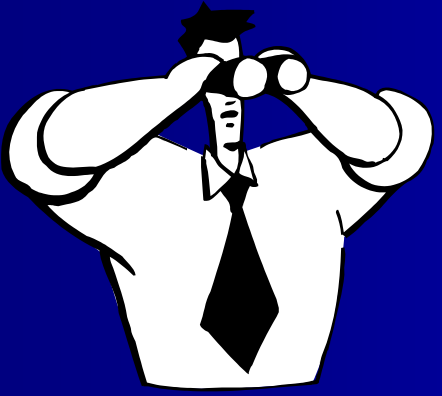

10001011110110000111

The Binary Search

by

Mr. Dave Clausen

Guessing Game

- Let's play a guessing game.
  - You will enter the largest number that you wish to guess, and
  - keep guessing until you find the random number between 1 and the largest number that you entered.
- This illustrates how the Binary Search finds an element in the list.

GuessingGame.cpp GuessingGame.exe

The Binary Search Description

The binary search consists of examining a middle value of a list to see which half contains the desired value. The middle value of the appropriate half is then examined to see which half of the half contains the value in question. This halving process is continued until the value is located or it is determined that the value is not in the list.

Binary Search Variations

- We will not use any variations of the Binary Search.
- We will only determine whether the target is in the list or not in the list.
- We will not find the index numbers of the target or how many occurrences there are in the list.

Binary Search Assumptions

- The list must be sorted!
  - This will allow us to find the middle data item in the list in constant time, by dividing the sum of the first index position and the last index position by two and using the subscript operation.

Binary Search Uses Recursion

- The basic idea of binary search can be expressed recursively.
  - If there are items in the list remaining to be examined, we compare the target value to the item at the middle position in the list.
  - If the target value equals this item, we return the index position of the item.
  - If the target value is greater than the item at the middle position, the target will be somewhere to the right of the middle position if it is in the list at all, so we recursively search the right half of the list.
  - Otherwise, the target value will be to the left of the middle position if it is in the list at all, so we recursively search the left half of the list.
  - If the target value is not in the list, we will run out of items to consider at the end of some recursive process, so we return the value -1.

Binary Search Parameters

- There are four input parameters to the problem:
  - the target item,
  - the list,
  - the index value of the first position in the list,
  - and the index value of the last position in the list.
- There is one value to be returned:
  - -1 indicating that we have not found the target item in the list
  - or an integer indicating its index position if we have found it.
- The initial value of the first position is zero. The initial value of the last index position is the number of data items in the list minus one.

A Binary Search Algorithm

//Using Iteration

If there are no more items to consider then

Return -1

Else

Set midpoint to (last + first) / 2

If the item at index midpoint = = the target item then

Return midpoint

Else if the item at index midpoint > the target item then

Return search the left half of the list (from indices first to midpoint - 1)

Else

Return search the right half of the list (from indices midpoint + 1 to last)

Binary Search Python Using Iteration

```
def binarySearch(target, listCopy):
    left = 0
    right = len(listCopy)-1
    while left <=right:
        midpoint = (left+right)//2
        if target == listCopy[midpoint]:
            return midpoint
        elif target < listCopy[midpoint]:
            right = midpoint - 1
        else:
            left = midpoint + 1
    return -1
```

Binary Search Python Using Recursion

```
def binarySearchRecursion(target, listCopy, first, last):
    if first > last:
        return -1
    else:
        midpoint = (first + last) // 2
        if listCopy[midpoint] == target:
            return midpoint
        elif listCopy[midpoint] > target:
            return binarySearchRecursion(target, listCopy, first, midpoint - 1)
        else:
            return binarySearchRecursion(target, listCopy, midpoint + 1, last)
```

The Binary Search C ++

```
//Using Recursion
int Binary_Search(int target, const apvector<int> &list, int first, int last)
{
   if (first > last)
      return -1;
   else
   {
      int midpoint = (first + last) / 2;
      if (list[midpoint] == target)
         return midpoint;
      else if (list[midpoint] > target)
         return Binary_Search(target, list, first, midpoint - 1);
      else
         return Binary_Search(target, list, midpoint + 1, last);
   }
}
```

Interface for Binary Search C++

- For many of the recursive functions it is customary to start with an “interface” function, since recursive functions call themselves.
- Here is the “interface” for the Binary Search:

```
int Bin_Search(int target, const apvector<int> &list)
{
   return Binary_Search(target, list, 0, list.length() - 1);
}
```

- You can use logical size - 1 instead of list.length() –1 if the logical and physical sizes are not the same. Don’t forget to pass logical size to the Bin_Search function.

Driver Program

- Here is a driver program to test the Binary Search source code:

binsearch.cpp binsearch.txt

Binary Search Walk Through

- Before continuing, let's walk through a binary search to better understand how it works. Assume **`list`** is the vector with values as indicated.

4	7	19	25	36	37	50	100	101	205	220	271	306	321
list[0]													list[13]

- Furthermore, assume **`target`** contains the value 205. Then initially, **`first`**, **`last`**, and **`target`** have the values

0	13	205
first	last	target

Walk Through 2

- A listing of values by each call of **`binsearch`** produces:

	first	last	midpoint	list[midpoint]
After initial call	0	13	6	50
After second call	7	13	10	220
After third call	7	9	8	101
After fourth call	9	9	9	205

- Note that we need only four comparisons to find the target at the ninth position in the list.

Walk Through 3

- To illustrate what happens when the value being looked for is not in the vector, suppose **`target`** contains 210. The listing of values then produces:

	first	last	midpoint	list[midpoint]
After initial call	0	13	6	50
After second call	7	13	10	220
After third call	7	9	8	101
After fourth call	9	9	9	205
After fifth call	10	9	9	205

- At this stage, **`first > last`** and the recursive process terminates.

Big - O Notation

Big - O notation is used to describe the efficiency of a search or sort. The actual time necessary to complete the sort varies according to the speed of your system. Big - O notation is an approximate mathematical formula to determine how many operations are necessary to perform the search or sort. The Big - O notation for the Binary Search is O($\mathbf{\log_2 n}$), because it takes approximately $\mathbf{\log_2 n}$ passes to find the target element.