

On the Science Behind Computing (Part I)

Samir Khuller *

1 Arrays

Often we need to store a large collection of values, and not just a single value. Say for example we wish to compute the average height of the students in the class. To do this, we first need to add up the heights of all the students in the class. We use something called an “Array” to store the heights. Let A be the name of the array. The array essentially allows us to create a large set of variables that we can access by referring to $A[0]$, $A[1]$ etc. Each $A[i]$ is actually an integer variable storing the height of the i^{th} student.

Suppose we have a set of students with heights 62, 65, 64, 68, 63, 62 (in inches). The following Ruby program will compute the average height.

```
sum=sum+A[0]
sum=sum+A[1]
sum=sum+A[2]
sum=sum+A[3]
sum=sum+A[4]
sum=sum+A[5]
```

```
A=Array[62,65,64,68,63,62]
sum=0
A.length.times{|i| sum=sum+A[i] }
print(‘#{sum/A.length}’)
```

The main idea is to create a variable called “sum”. In this variable we store the sum of all the elements seen so far as we scan the array. We initialize the value of sum to 0, and then first add $A[0]$ to sum, then we add $A[1]$ to sum, then $A[2]$ to sum etc.

$A.length$ tells us the value of the length of the array. In this example, this will be 6. We will iterate over the array with $i = 0, 1, 2, 3, 4, 5$ and essentially do the following commands.

At the end, sum contains the sum of the six elements in the Array A . To compute the average height, we divide by the number of entries in the array, which is given by $A.length$.

2 Looking up information

There are many many applications where we need to look up information in extremely large tables. Consider the following simple problem faced by a cellular provider that has millions of

*Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail : samir@cs.umd.edu.

customers. Cell phone towers have a small coverage range – as we travel by car or by train, we move from connecting to one cell tower to another. How do calls then get routed to a user who is moving around from one region to another? If I call the person’s number – how does the phone network “know” which cell tower to route the call to? One simple solution would be to create two large “tables”. The first table (an array) A stores all the phone numbers in order. In other words, $A[0]$ is the first phone number (smallest phone number say 2022132637), $A[1]$ may be the next one, 2022132641 etc. The second table stores the cell phone towers that users are currently close to. For example if $A[j] = 3012373100$ then in $B[j]$ we store the cell tower number that this phone number is currently connected to. If this phone moves from cell tower 317 to cell tower 419, then we change $B[j]$ from 317 to 419. It is partly due to these rapid changes that cellular companies disallow the use of cell phones on airplanes, while they are in flight.

How do we look up the location of a particular phone? Given the phone number Y , we want to “find” i such that $A[i] == Y$. Once we have i , $B[i]$ gives us the location of phone Y . As the customer moves from region to region, the phone sends messages whenever it registers with a new tower to update the table so that calls can be routed to the correct cell tower.

This motivates the need to look up information in large tables. The most amazing thing about the searching method is that even in a table with one million entries, by considering only 20 entries, we can find the correct location where a given phone number is stored. This is simply amazing! Later on we will discuss why we only need to look up no more than 20 entries. We accomplish this using a simple but powerful idea called “binary search”.

The key idea behind binary search is to order the entries and then consider the “middle” entry. If the value we are searching for is exactly the middle entry, then we are done - we have found the value. If it is smaller than the middle entry then we need to search for the value in an array of half the size (the left half) and if the value we are searching for is larger than the middle entry then we need to search in the right half of the array.

We define two indexes into the array – called **low** and **high**. The portion of the array we are searching in is the portion going from $A[\text{low}]$ to $A[\text{high}]$. Initially **low** =0 and **high**= A.length. At each step, by making a single comparison we are able to narrow down the search by either increasing **low** to **mid**+1, or by decreasing **high** to **mid**-1. The binary search repeatedly compares y to $A[\text{mid}]$ until we find it, or **low** becomes equal to **high**. In this case, our array has only one element – if this element is not y then it does not exist in the array. (As an exercise you should try to run binary search to look for entries not in the array.)

To do a binary search in an array containing N elements, it takes $\log_2 N$ comparisons. Thus if we do binary search in an array containing 1000 elements it only takes 10 comparisons. For a million elements it takes 20 comparisons. You may want to at least think about why this is the case.

```

# Binary Search
A=Array[3,7,9,11,15,19,27,29,31,45,56,78,99]

def find(y)
  low=0
  high=A.length

  while (low < high)
    mid = (low+high)/2 #define the middle index
    print ("#{mid}\n") #this is just a check to see what comparison was done
    if (A[mid]==y)
      return mid
    elsif (A[mid]<y)
      low=mid+1
    elsif (A[mid]>y)
      high=mid-1
    end
  end

  if (A[low]==y) then
    return low
  else
    return -1 #return a special code if y was not found
  end
end

#the following lets us provide the input argument from the command line
#ruby find.rb 9 will search for element 9 in the input array, and return 2
if (ARGV.length == 1)
  p = Integer(ARGV[0])
end

print("The index of #{p} is #{find(p)}\n");

```