Searching Elements in an Array: Linear and Binary Search

Searching

- Check if a given element (called ***key***) occurs in the array.
  - Example: array of student records; rollno can be the key.
- Two methods to be discussed:
  - If the array elements are unsorted.
    - Linear search
  - If the array elements are sorted.
    - Binary search

Linear Search

Basic Concept

- **Basic idea:**
  - Start at the beginning of the array.
  - Inspect elements one by one to see if it matches the key.
- **Time complexity:**
  - A measure of how long an algorithm takes to run.
  - If there are ***n*** elements in the array:
    - **Best case:**

      match found in first element (1 search operation)
    - **Worst case:**

      no match found, or match found in the last element (n search operations)
    - **Average case:**

      (n + 1) / 2 search operations

```
#include <stdio.h>

int linear_search (int a[], int size, int key)
{
   for (int i=0; i<size; i++)
     if  (a[i] == key)  return i;
   return -1;
}

int main()
{
   int x[]={12,-3,78,67,6,50,19,10}, val;
   printf ("\nEnter number to search: ");
     scanf ("%d", &val);
   printf ("\nValue returned: %d \n", linear_search (x,8,val);
}
```

- **What does the function `linear_search` do?**
  - It searches the array for the number to be searched element by element.
  - If a match is found, it returns the array index.
  - If not found, it returns -1.

Contd.

```
int x[]= {12, -3, 78, 67, 6, 50, 19, 10};
```

- **Trace the following calls :**

  search (x, 8, 6) ; ← **Returns 4**

  search (x, 8, 5) ; → **Returns -1**

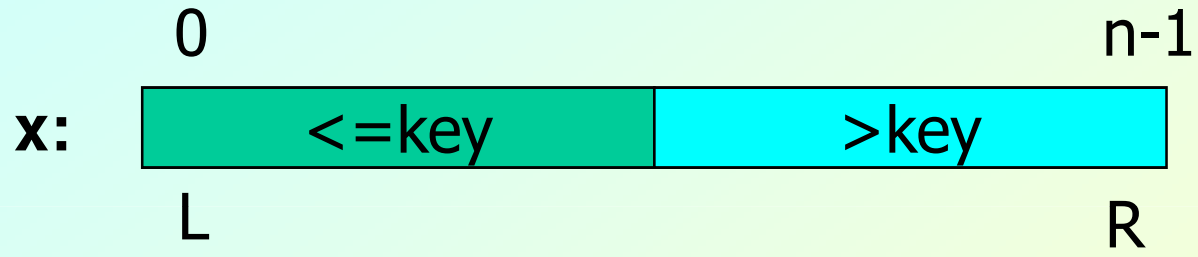
Binary Search

Basic Concept

- Binary search works if the array is *sorted*.
  - Look for the target in the middle.
  - If you don't find it, you can ignore half of the array, and repeat the process with the other half.
- In every step, we reduce the number of elements to search in by half.

The Basic Strategy

- **What we want?**
  - **Find split between values larger and smaller than key:**



  - **Situation while searching:**
    - **Initially L and R contains the indices of first and last elements.**
  - **Look at the element at index [(L+R)/2].**
    - **Move L or R to the middle depending on the outcome of test.**

Iterative Version

```
#include <stdio.h>

int bin_search (int a[], int size, int key)
{
   int  L, R, mid;
   L = 0;  R = size - 1;

   while  (L <= R)  {
     mid = (L + R) / 2;
     if  (a[mid] < key)  L = mid + 1;
     else if (a[mid] > key)  R = mid -1;
          else  return mid;   /* FOUND AT INDEX mid */
   }

  return  -1;     /* NOT FOUND */
}
```

```
int main()
{
   int x[]={10,20,30,40,50,60,70,80}, val;

   printf ("\nEnter number to search: ");
     scanf ("%d", &val);

   printf ("\nValue returned: %d \n", bin_search (x,8,val);
}
```

Recursive Version

```
#include <stdio.h>

int bin_search (int a[], int L, int R, int key)
{
   int  mid;

   if  (R < L)  return -1;     /* NOT FOUND */
   mid = (L + R) / 2;
   if  (a[mid] < key)  return (bin_search(a,mid+1,R,key));
   else if (a[mid] > key) return (bin_search(a,L,mid-1,key));
        else  return mid;   /* FOUND AT INDEX mid */
}
```

```
int main()
{
   int x[]={10,20,30,40,50,60,70,80}, val;

   printf ("\nEnter number to search: ");
     scanf ("%d", &val);

   printf ("\nValue returned: %d \n", bin_search (x,0,7,val);
}
```

Is it worth the trouble ?

- Suppose that the array **x** has 1000 elements.
- Ordinary search
  - If ***key*** is a member of **x**, it would require 500 comparisons on the average.
- Binary search
  - after 1st compare, left with 500 elements.
  - after 2nd compare, left with 250 elements.
  - After at most 10 steps, you are done.

Time Complexity

- If there are n elements in the array.
  - Number of searches required: $\log_2 n$
- For n = 64 (say).
  - Initially, list size = 64.
  - After first compare, list size = 32.
  - After second compare, list size = 16.
  - After third compare, list size = 8.
  - …….
  - After sixth compare, list size = 1.

$2^k = n$, where k is the number of steps.

$\log_2 64 = 6$

$\log_2 1024 = 10$