CS19001: Programming and Data Structures Laboratory

Soumyajit Dey, Aritra Hazra;
CSE, IIT Kharagpur

http://cse.iitkgp.ac.in/~aritrah/course/lab/PDS/Autumn2018/CS19101_PDS-Lab_Autumn2018.html

29-Sep-2018

Sorting

- Suppose you have an array A[ ] of n elements (say, integers). They are stored in the array locations,

$$A[0], A[1], \ldots, A[i], \ldots, A[n-1]$$

- We want to rearrange these integers in such a way that after the rearrangement, we have either of the following:

$$A[0] \leq A[1] \leq \cdots \leq A[i] \leq \cdots \leq A[n-1]$$

$$A[0] \geq A[1] \geq \cdots \geq A[i] \geq \cdots \geq A[n-1]$$

- Then, the resultant array is called **sorted** in either **ascending** or **descending** order, respectively.
- There are many such sorting methods. *Bubble-sort* and *Selection-sort* are two among them.

Bubble-sort (in ascending order)

Code

```
for (i=n-2; i>=0; --i)
{
   for (j=0; j<=i; ++j)
   {
      if (A[j] > A[j+1])
      {
         t = A[j];
         A[j] = A[j+1];
         A[j+1] = t;
      }
   }
}
```

Working Principle

A[4] = {4,3,2,1}
i,j: $A \rightarrow A'$

bubble till position i=4-2=2.
2,0: 4,3,2,1 → 3,4,2,1
2,1: 3,4,2,1 → 3,2,4,1
2,2: 3,2,4,1 → 3,2,1,4
bubble till position i=1
1,0: 3,2,1,4 → 2,3,1,4
1,1: 2,3,1,4 → 2,1,3,4
bubble till position i=0
0,0: 2,1,3,4 → 1,2,3,4

Selection-sort (in ascending order)

Code

```
for (i=n-1; i>0; --i)
{
  m = i;
  for (j=0; j<i; ++j)
  {
    if (A[j] > A[m])
      m = j;
  }
  t = A[i];
  A[i] = A[m];
  A[m] = t;
} // Why swap if i=m?
```

Working Principle

$A[4] = \{4,3,2,1\}$

$i = 3 \rightarrow m = 3$

$j = 0$: $m = 0$

$j = 1$: $m = 0$

$j = 2$: $m = 0$

$A[4] = \{1,3,2,4\}$

$i = 2 \rightarrow m = 2$

$j = 0$: $m = 2$

$j = 1$: $m = 1$

$A[4] = \{1,2,3,4\}$

$i = 1 \rightarrow m = 1$

$j = 0$: $m = 1$

$A[4] = \{1,2,3,4\}$

Searching

- Suppose you have an array A[ ] of n elements (say, integers). They are stored in the array locations,

$$A[0], A[1], \ldots, A[i], \ldots, A[n-1]$$

- We want to search/report the location/index of a particular value, say v, from this array of integers.
- We report the index 'k' ($0 \leq k < n$), if $A[k] = v$. Otherwise, we may report '-1' to denote that the searched element, v, is not found.
- Given an *unordered* array, you have to compare each element of the array **sequentially** to find the index,

$$\textit{For all } i \ (0 \leq i < n), \ \textit{whether } A[i] = v?$$

- However, for *ordered* (ascending / descending) arrays, things are more exciting! *We shall study these variants.*

Searching in an Unordered Array

Forward-Iteration

```
for (i=0; i<n; ++i)
  if(A[i] == v)
    break;
// Answer: found at i
if(i == n) i = -1;
// Answer: not found
```

Backward-Iteration

```
for (i=n-1; i>=0; --i)
  if(A[i] == v)
    break;
// Answer: found at i
```

Recursive-Code

```
int seqSr ( int A[], int n,
                     int v )
{
  if (n > 0)
  {
    if(A[n-1] == v)
      return n-1;
    else
      return (seqSr(A,n-1,v));
  }
  else
  {
    return -1;
  }
}
```

Searching in an Ordered Array

Idea of Binary Search:

Consider a sorted array A and an element (say v) as input. The goal is to report whether the element is present in the array and in that case what is the corresponding array index.

- Choose the middle element $A[n/2]$
- If $v == A[n/2]$, we are done
- If $v < A[n/2]$, search for v between $A[0]$ and $A[n/2-1]$
- If $v > A[n/2]$, search for v between $A[n/2+1]$ and $A[n-1]$
- Repeat until v is found or no more elements remain to be searched.

We consider three variables first, last and mid pointing to array beginning, end and middle respectively. We keep on updating these three elements in each iteration recursively.

Searching in an Ordered Array

Binary Search

```
int binSr ( int A[], int v,
            int si, int ei )
{
 int mi;
 if (si <= ei)
 {
  mi = (si+ei)/2;
  if(A[mi] > v)
   return binSr(A,v,si,mi-1);
  else if (A[mi] < v)
   return binSr(A,v,mi+1,ei);
  else
   return mi;
 }
 else
  return -1;
}
```

Working Principle

Ex-1: A[5] = {1, 2, 3, 4, 5}; v = 2

si=0, ei=4; mi=(0+4)/2=2
A[ ] = {1, 2, 3, 4, 5}; A[2]=3(>2)
si=0, ei=mi-1=1; mi=0
A[ ] = {1, 2}, 3, 4, 5}; A[0]=1(<2)
si=mi+1=1, ei=1; mi=1
A[ ] = {1, {2}, 3, 4, 5}; A[1]=2

Ex-2: A[5] = {1, 2, 3, 4, 5}; v = 0

si=0, ei=4; mi=(0+4)/2=2
A[ ] = {1, 2, 3, 4, 5}; A[2]=3(>0)
si=0, ei=mi-1=1; mi=0
A[ ] = {1, 2}, 3, 4, 5}; A[0]=1(>0)
si=0, ei=mi-1=-1
A[ ] = {1, 2, 3, 4, 5}; not-found(-1)

Programming Assignments

Complete and submit during lab

Assignment 1 [MinMax-Sort]

Write a C-program to perform MinMax-sort over an unordered n-element integer array to make the elements ascending-ordered.

Procedure

The working of the MinMax-sort is somewhat similar to that of selection sort. Here, the outer loop runs over (i, j) together, where i ranges from 0 up to $(\lfloor \frac{n}{2} \rfloor - 1)$ and j ranges from $(n-1)$ down to $\lceil \frac{n}{2} \rceil$. For given i, j, largest and smallest elements in the sub-array $A[i], A[i+1], \ldots, A[j-1], A[j]$ are found out (both together) and are swapped with the elements $A[j]$ and $A[i]$, respectively. Thus, during the first iteration of the outer loop $A[n-1]$ and $A[0]$ receives the largest and smallest element in the array, respectively; in the second iteration $A[n-2]$ and $A[1]$ receives the second-largest and second-smallest element, respectively and so on.

Example

$\{4,5,6,3,1,2\} \longmapsto$ after iteration 1 of outer loop $\longmapsto \{1,5,2,3,4,6\}$
$\{1,5,2,3,4,6\} \longmapsto$ after iteration 2 of outer loop $\longmapsto \{1,2,4,3,5,6\}$
$\{1,2,4,3,5,6\} \longmapsto$ after iteration 3 of outer loop $\longmapsto \{1,2,3,4,5,6\}$

Assignment 2 [Biparted-Ternary-Search]

Procedure

Consider a variation of binary search where the sorted array of size n is divided into two parts, but everytime by choosing the $n/3$-th element instead of the middle elements. The algorithm is as follows:

- Compare v (the searched element) with the $n/3$-th element
- If equal, v found – return
- If v is smaller, search first sub-array (0 to $n/3 - 1$)
- If v is greater, search middle sub-array ($n/3 + 1$ to $n - 1$)

Recursive-Function

Write a recursive C-function
int BiTernarySearch (int A[ ], int v, int low, int high)
which takes as parameters a sorted array A of integers, two indices *low* and *high* ($low \leq high$) in A and the element to be searched for v. The function returns the index, k ($low \leq k \leq high$), of A if v is found within the indices *low* and *high* (both included) of A, otherwise it returns -1.

Assignment 2 [Biparted-Ternary-Search]

Main-Program

Write a main C-function that

1. reads from user an integer n ($n \leq 100000$) and then takes from user n integers in an array (may be unordered);
2. reads another integer x, which is the element being searched;
3. sort the array elements in ascending order using previous **MinMax-Sort** program (Refer to *Assignment-1*);
4. checks whether x resides in the array or not, by using **BiTernarySearch** function;
5. prints the location/index where the element x resides in the array, otherwise print -1 in case it is not found.

Assignment 3 [Triparted-Ternary-Search]

Consider a variation of binary search where the sorted array of size n is divided into three parts instead of two parts by choosing the $n/3$-th and $2n/3$-th elements instead of only the middle elements. The algorithm is as follows:

- Compare v (the element being searched for) with the $n/3$-th element
- If equal, v found – return
- If v is smaller, search first sub-array (0 to $n/3 - 1$)
- If v is greater, compare with $2/3$-th element
- If equal, v found – return
- If v is smaller, search middle sub-array ($n/3 + 1$ to $2n/3 - 1$)
- If v is greater, search third sub-array ($2n/3 + 1$ to $n - 1$)

Assignment 3 [Triparted-Ternary-Search]

Soumyajit Dey, Aritra Hazra; CSE, IIT Kharagpur

Recursive-Function

Write a recursive C-function
int TriTernarySearch (int A[ ], int v, int low, int high)
which takes as parameters a sorted array A of integers, two indices *low* and *high* (*low* $\leq$ *high*) in A and the element to be searched for v. The function returns the index, k (*low* $\leq k \leq$ *high*), of A if v is found within the indices *low* and *high* (both included) of A, otherwise it returns -1.

Main-Program

Write a main C-function that

1. reads from user an integer n ($n \leq 100000$) and then takes from user n integers in an array (must be in ascending order);
2. reads another integer x, which is the element being searched;
3. checks whether x resides in the array or not, by using **TriTernarySearch** function;
4. prints the location/index where the element x resides in the array, otherwise print -1 in case it is not found.

You do not have to sort the array. Just enter the numbers in sorted order directly from the keyboard.

Thank You