CS11001/CS11002 Programming and Data Structures (PDS) (Theory: 3-0-0)

Teacher: Sourangshu Bhattacharya
sourangshu@gmail.com
http://cse.iitkgp.ac.in/~sourangshu/

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

Sorting: the basic problem

- Given an array

  `x[0], x[1], ... , x[size-1]`

  reorder entries so that

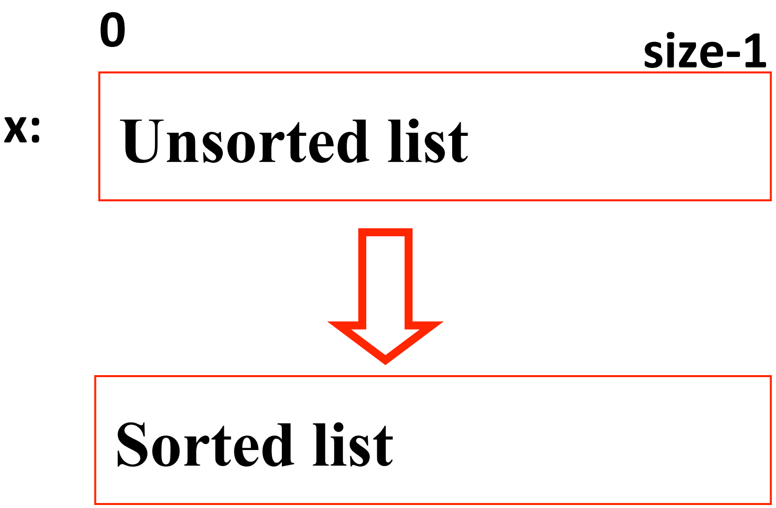
  `x[0] <= x[1] <= . . .  <= x[size-1]`

  `List is in non-decreasing order.`

- We can also sort a list of elements in non-increasing order.

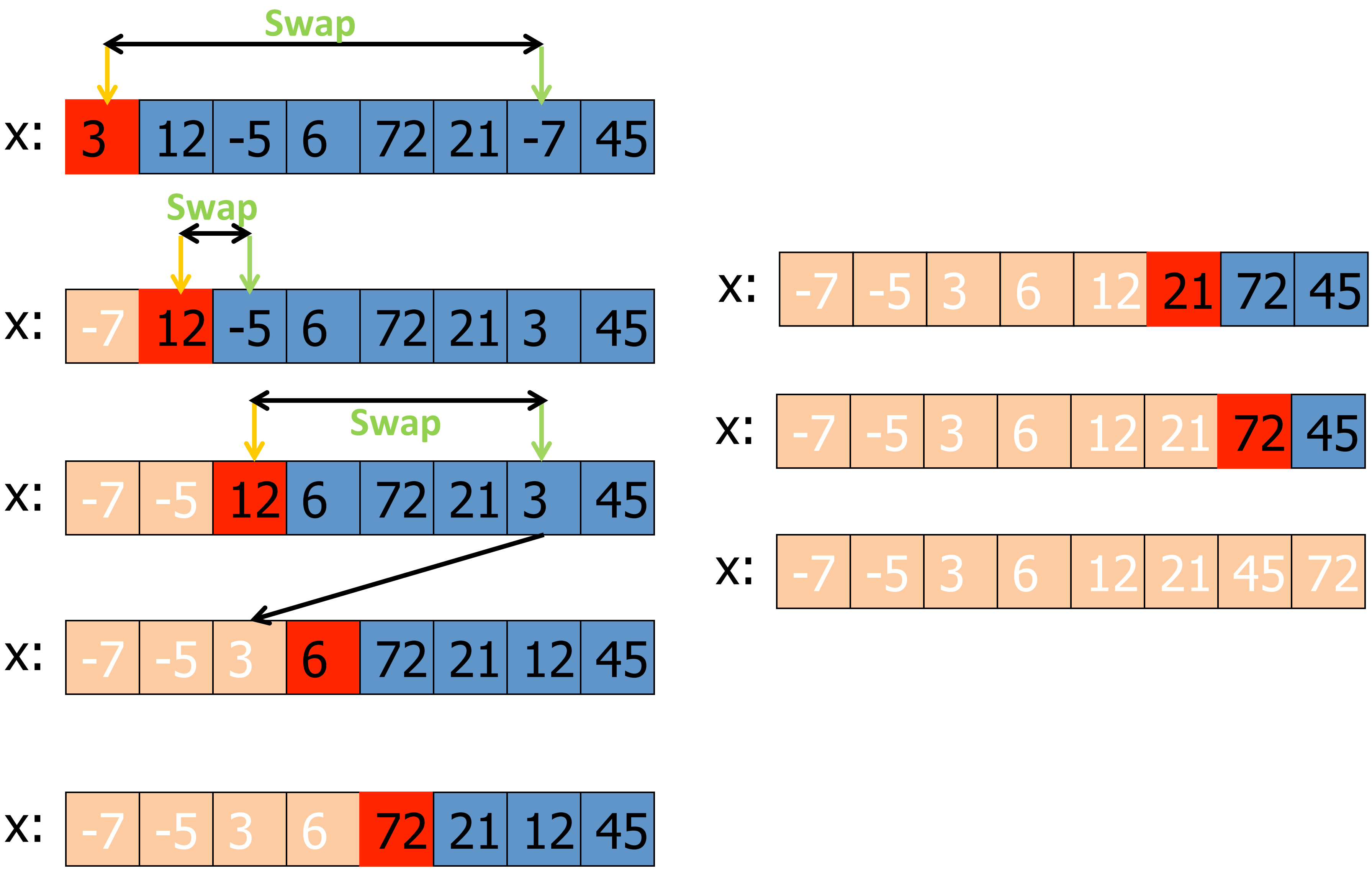
Sorting Problem

- What we want : Data sorted in order
- Input: A list of elements
- Output: A list of elements in sorted (non-increasing/non-decreasing) order



- Original list:
  - 10, 30, 20, 80, 70, 10, 60, 40, 70
- Sorted in non-decreasing order:
  - 10, 10, 20, 30, 40, 60, 70, 70, 80
- Sorted in non-increasing order:
  - 80, 70, 70, 60, 40, 30, 20, 10, 10

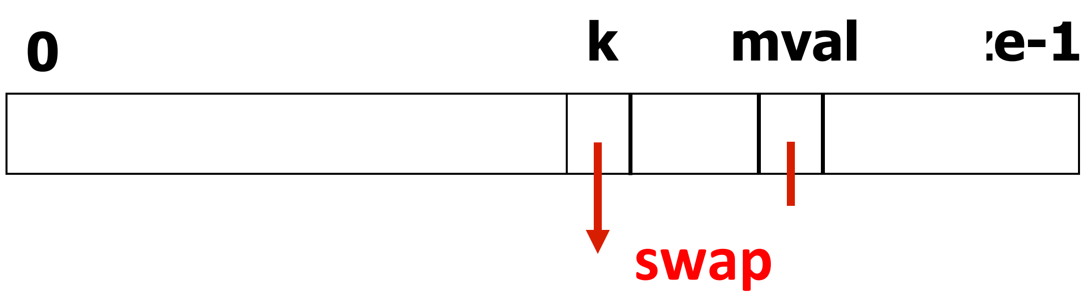
Example


Swap
x: 3 12 -5 6 72 21 -7 45
Swap
x: -7 12 -5 6 72 21 3 45
Swap
x: -7 -5 12 6 72 21 3 45
x: -7 -5 3 6 72 21 12 45
x: -7 -5 3 6 72 21 12 45
x: -7 -5 3 6 12 21 72 45
x: -7 -5 3 6 12 21 72 45
x: -7 -5 3 6 12 21 45 72

Selection Sort

- General situation :

- **Steps :**
  - `Find smallest element, mval, in x[k+1..size-1]`
  - `Swap smallest element with x[k-1],`
  - `Increase k.`



Subproblem: Find smallest element

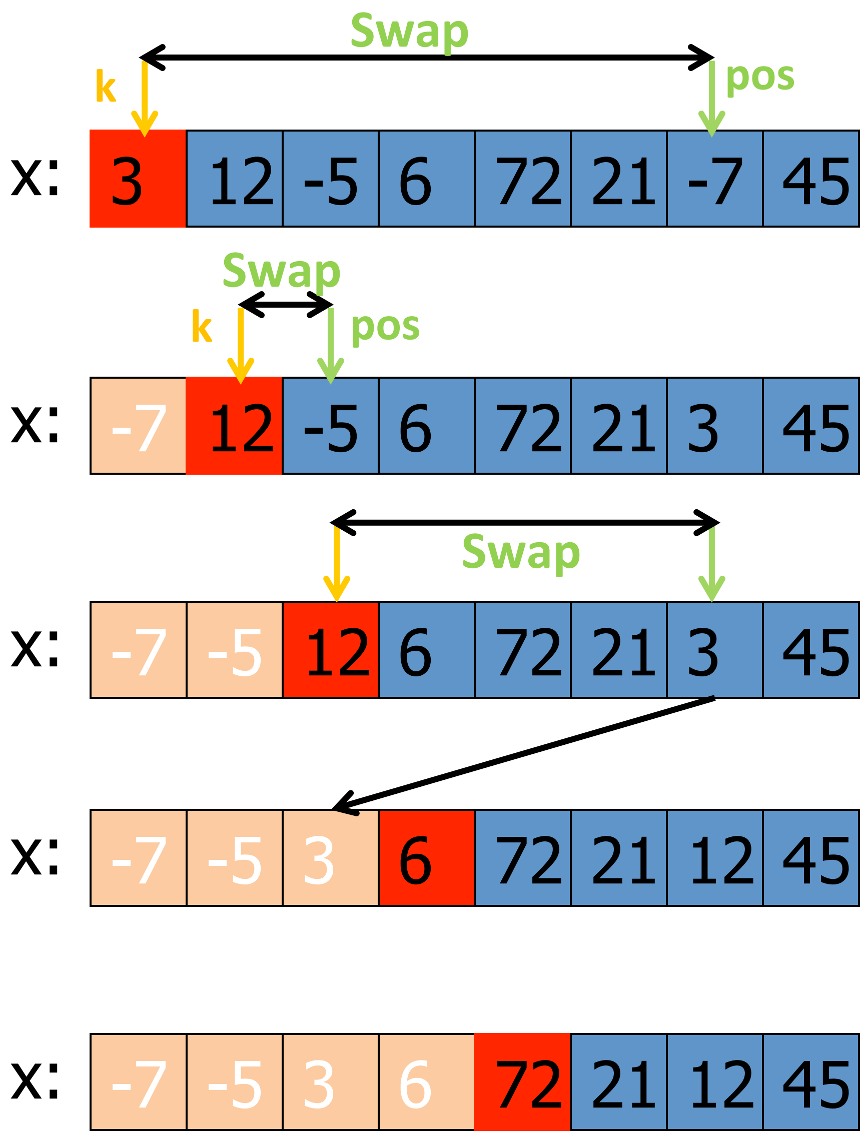
```
/* Yield location of smallest element in x[k ..
   size-1] and store in pos*/

int j, pos;
pos = k;      /* assume first element is the smallest element */
for (j=k+1; j<size; j++) {
    if (x[j] < x[pos]) { /* x[pos] is the smallest element as
   of now */
        pos = j;
    }
}
printf("%d",pos);
```

Subproblem: Swap with smallest element

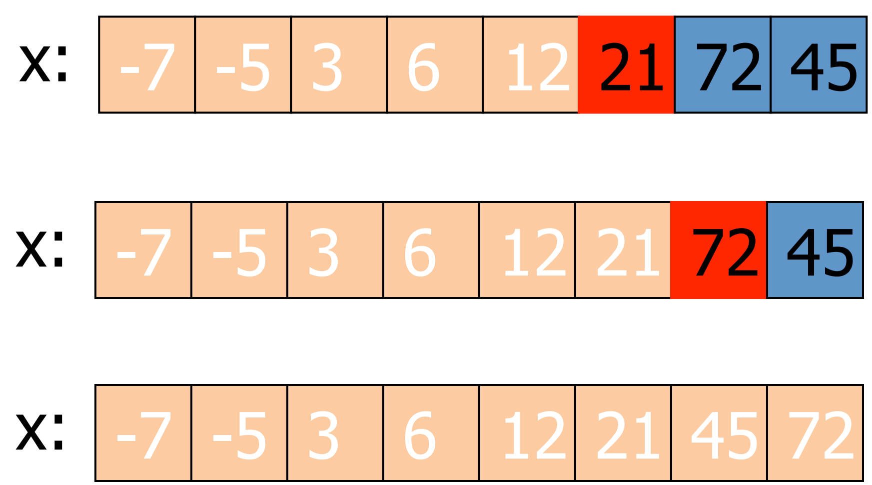
```
/* Yield location of smallest element in x[k .. size-1] and
   store in pos*/

int j, pos;
pos = k;        /* assume first element is the smallest
   element */
for (j=k+1; j<size; j++) {
    if (x[j] < x[pos]) {    /* x[pos] is the smallest element
   as of now */
        pos = j;
    }
}
printf("%d",pos);
if (x[pos] < x[k]) {
   temp = x[k];       /* swap content of x[k] and x[pos] */
   x[k] = x[pos];
   x[pos] = temp;
}
```

```
#include <stdio.h> /* Sort x[0..size-1] in non-decreasing order */
int main()
{
    int k,j,pos,x[100],size,temp
    printf("Enter the number of elements: ");
    scanf("%d",&size);
    printf("Enter the elements: ");
    for(k=0;k<size;k++) scanf("%d",&x[k]);
    for(k=0; k<size-1; k++)  {
       pos = k; /* assume first element is the smallest element */
       for(j=k+1; j<size; j++) {
          if (x[j] < x[pos]) /*x[pos] is the smallest element as of now*/
              pos = j;
       }

       temp = x[k];           /* swap content of x[k] and x[pos] */
       x[k] = x[pos];
       x[pos] = temp;
     }

     for(k=0;k<size;k++)/* print the sorted (non-decreasing) list */
              printf("%d ",x[k]);
      return 0;
}
```

Example




```
for (k=0; k<size-1; k++)  {
        pos = k;          for
    (j=k+1; j<size; j++) {
            if (x[j] < x[pos])
    pos = j;
        }
        temp = x[k];          x[k]
    = x[pos];
        x[pos] = temp;
}
```



Analysis

How many steps are required?

Let us assume there are n elements (size=n).
Each statement executes in constant time.

To read

for loop will take of the **order of n** time

Sorting

When k=0, in worst case

(n-1) comparisons to find minimum.

When k=1, in worst case

(n-2) comparisons to find minimum.

…..

(n-1)+(n-2)+……+1= n(n-1)/2

To print

for loop will take of the **order of n** time

```
for(k=0;k<size;k++)

   scanf("%d",&x[k]);
for (k=0; k<size-1; k++)  {
   pos = k;
   for(j=k+1; j<size; j++)
   {
         if (x[j] < x[pos])
            pos = j;
    }
    temp = x[k];
    x[k] = x[pos];
    x[pos] = temp;
}
 for(k=0;k<size;k++)
      printf("%d ",x[k]);
```

Total time= 2 × order of n + order of n^2 = order of n^2

Analysis

How many steps are required?

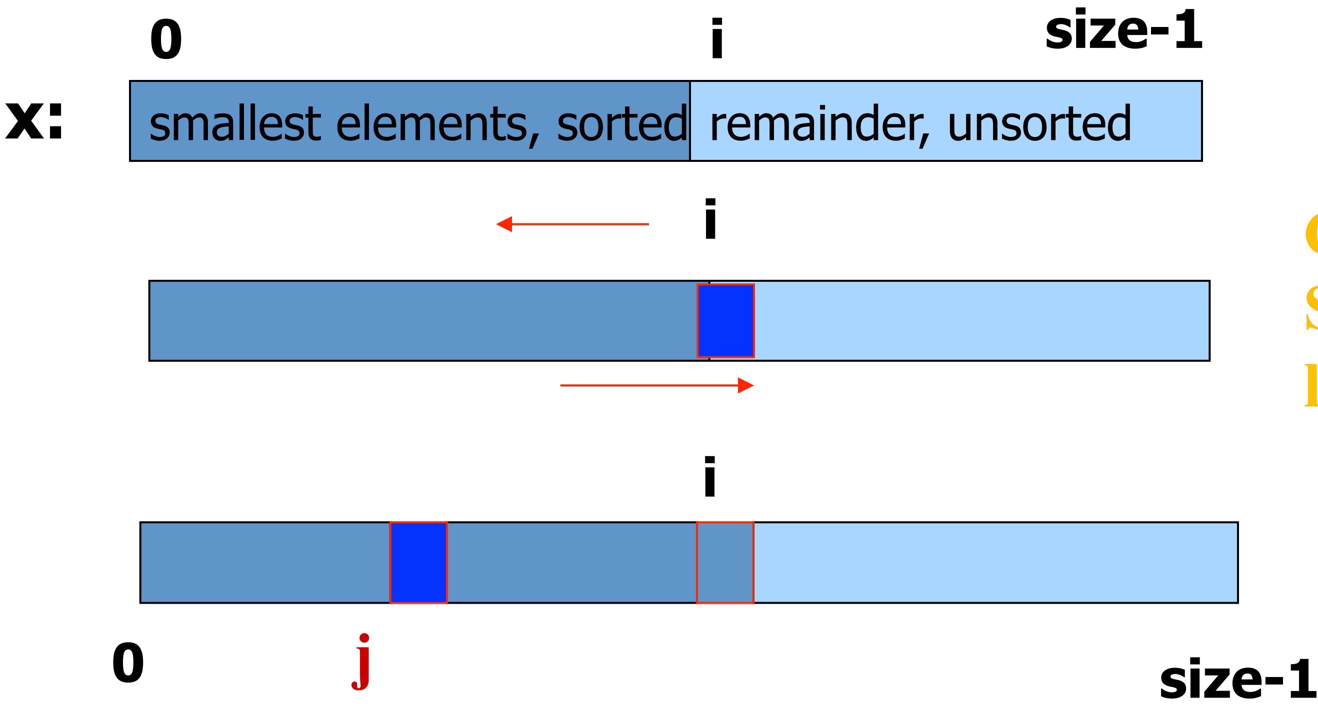
Let us assume there are n elements (size=n).
Each statement executes in constant time.

Best Case?
Worst Case?
Average Case?

```
for(k=0;k<size;k++)

   scanf("%d",&x[k]);
for (k=0; k<size-1; k++)  {
    pos = k;
    for(j=k+1; j<size; j++)
   {
        if (x[j] < x[pos])
            pos = j;
     }
     temp = x[k];
     x[k] = x[pos];
     x[pos] = temp;
}
 for(k=0;k<size;k++)
      printf("%d ",x[k]);
```

Insertion Sort

- General situation :



Compare and Shift till x[i] is larger.

Insertion Sorting

```
int list[100], size;
____________________;

for (i=1; i<size; i++) {
   item = list[i] ;
   for (j=i-1; (j>=0)&& (list[j] > item); j--)
       list[j+1] = list[j];
   list[j+1] = item ;
}
____________________;
```

Complete Insertion Sort

```
#include <stdio.h> /* Sort x[0..size-1] in non-decreasing order */
#define SIZE 100
int main()
{
   int i,j,x[SIZE],size,temp;
   printf("Enter the number of elements: ");
   scanf("%d",&size);
   printf("Enter the elements: ");
   for(i=0;i<size;i++)
           scanf("%d",&x[i]);
   for (i=1; i<size; i++) {
    temp = x[i] ;
    for (j=i-1; (j>=0)&& (x[j] > temp); j--)
        x[j+1] = x[j];
    x[j+1] = temp ;
   }
   for(i=0;i<size;i++) /* print the sorted (non-decreasing) list */
                printf("%d ",x[i]);
        return 0;
}
```

Insertion Sort Example

```
for (i=1; i<size; i++) {
        temp = x[i] ;
        for (j=i-1; (j>=0)&& (x[j] > temp); j--)
                x[j+1] = x[j];
        x[j+1] = temp ;
}
```

Input:

3 **12** -5 6 72 21 -7 45

3 **12** **-5** 6 72 21 -7 45

-5 3 12 **6** 72 21 -7 45

-5 3 **6** 12 **72** 21 -7 45

-5 3 6 12 **72** **21** -7 45

-5 3 6 12 **21** 72 **-7** 45

-7 -5 3 6 12 21 72 **45**

-7 -5 3 6 12 21 **45** 72

Output:

-5 3 6 12 **21** 72 **-7** 45

-7

-5 3 6 12 21 72 ___ 45

-5 3 6 12 21 ___ 72 45

-5 3 6 12 ___ 21 72 45

-5 3 6 ___ 12 21 72 45

-5 3 ___ 6 12 21 72 45

-5 ___ 3 6 12 21 72 45

___ -5 3 6 12 21 72 45

-7 -5 3 6 12 21 72 45

Time Complexity

- Number of comparisons and shifting:

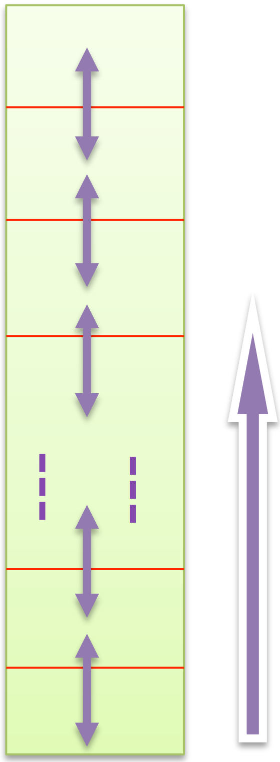
  - Worst Case?

$$1+2+3+ \ldots\ldots +(n-1) = n(n-1)/2$$

  - Best Case?

$$1+1+\ldots\ldots \; (n-1) \text{ times} = (n-1)$$

Bubble Sort



In every iteration heaviest element drops at the bottom.

The bottom moves upward.

Example

Pass: 1

x:	3	12	-5	6	72	21	-7	45

x:	3	12	-5	6	72	21	-7	45

x:	3	-5	12	6	72	21	-7	45

x:	3	-5	6	12	72	21	-7	45

x:	3	-5	6	12	72	21	-7	45

x:	3	-5	6	12	21	72	-7	45

x:	3	-5	6	12	21	-7	72	45

x:	3	-5	6	12	21	-7	45	72

Bubble Sort

```
int pass,j,a[100],temp;

/* read number of elements as  n and elements as a[] */

for(pass=0; pass<n; pass++) {
     /* sub pass */
     for(j=0; j<n-1; j++) {
          if(a[j]>a[j+1]) {
               temp = a[j+1];
               a[j+1] = a[j];
               a[j] = temp;
          }
     }
}

/* print sorted list of elements */
```

Example

Pass: 1

x:

3	12	-5	6	72	21	-7	45

x:

3	12	-5	6	72	21	-7	45

x:

3	-5	12	6	72	21	-7	45

x:

3	-5	6	12	72	21	-7	45

x:

3	-5	6	12	72	21	-7	45

x:

3	-5	6	12	21	72	-7	45

x:

3	-5	6	12	21	-7	72	45

x:

3	-5	6	12	21	-7	45	72

Example

Pass: 2

x:	3	-5	6	12	21	-7	45	72

x:	-5	3	6	12	21	-7	45	72

x:	-5	3	6	12	21	-7	45	72

x:	-5	3	6	12	21	-7	45	72

x:	-5	3	6	12	21	-7	45	72

x:	-5	3	6	12	-7	21	45	72

x:	-5	3	6	12	-7	21	45	72

Time Complexity

- Number of comparisons :
  - Worst Case?

    $1+2+3+ \ldots\ldots +(n-1) = n(n-1)/2$

  - Best Case?

    Same

How do you make best case with (n-1) comparisons only?

Bubble Sort

```
int pass,j,a[100],temp,swapflag;
/* read number of elements as  n and elements as a[] */

for(pass=0; pass<n; pass++) {

     swapflag=0;
     for(j=0; j<n-1; j++) {
          if(a[j]>a[j+1]) {
             temp = a[j+1];
             a[j+1] = a[j];
             a[j] = temp;
             swapflag=1;
          }
     }
     if(swapflag==0)
          break;
}
/* print sorted list of elements */
```

Can we improve the sorting time?

Basis of efficient sorting algorithms

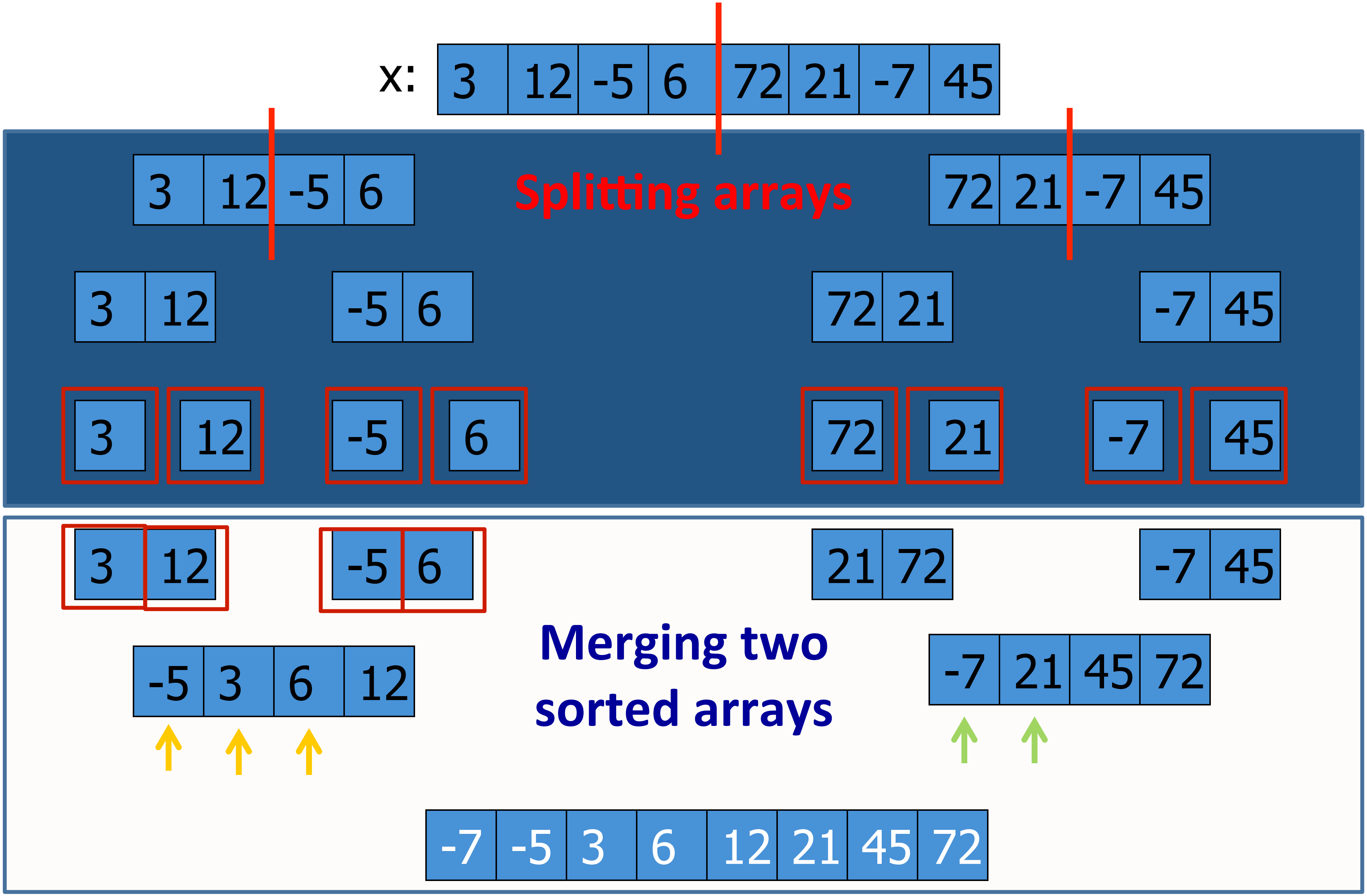
- Two of the most popular sorting algorithms are based on divide-and-conquer approach.
  – Quick sort
  – Merge sort

- Basic concept:

```
sort (list)
{
   if the list has length greater than 1
   {
      Partition the list into lowlist and highlist;
      sort (lowlist);
      sort (highlist);
      combine (lowlist, highlist);
   }
}
```

Merge Sort

Example


x:
3 12 -5 6 72 21 -7 45
Splitting arrays
3 12 -5 6
72 21 -7 45
3 12
-5 6
72 21
-7 45
3
12
-5
6
72
21
-7
45
3 12
-5 6
21 72
-7 45
Merging two sorted arrays
-5 3 6 12
-7 21 45 72
-7 -5 3 6 12 21 45 72

Merge Sort C program

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
    int a[30],n,i;
    printf("Enter no of elements:");
    scanf("%d",&n);
    printf("Enter array elements:");

    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    mergesort(a,0,n-1);

    printf("\nSorted array is :");
    for(i=0;i<n;i++)
        printf("%d ",a[i]);

    return 0;
}
```

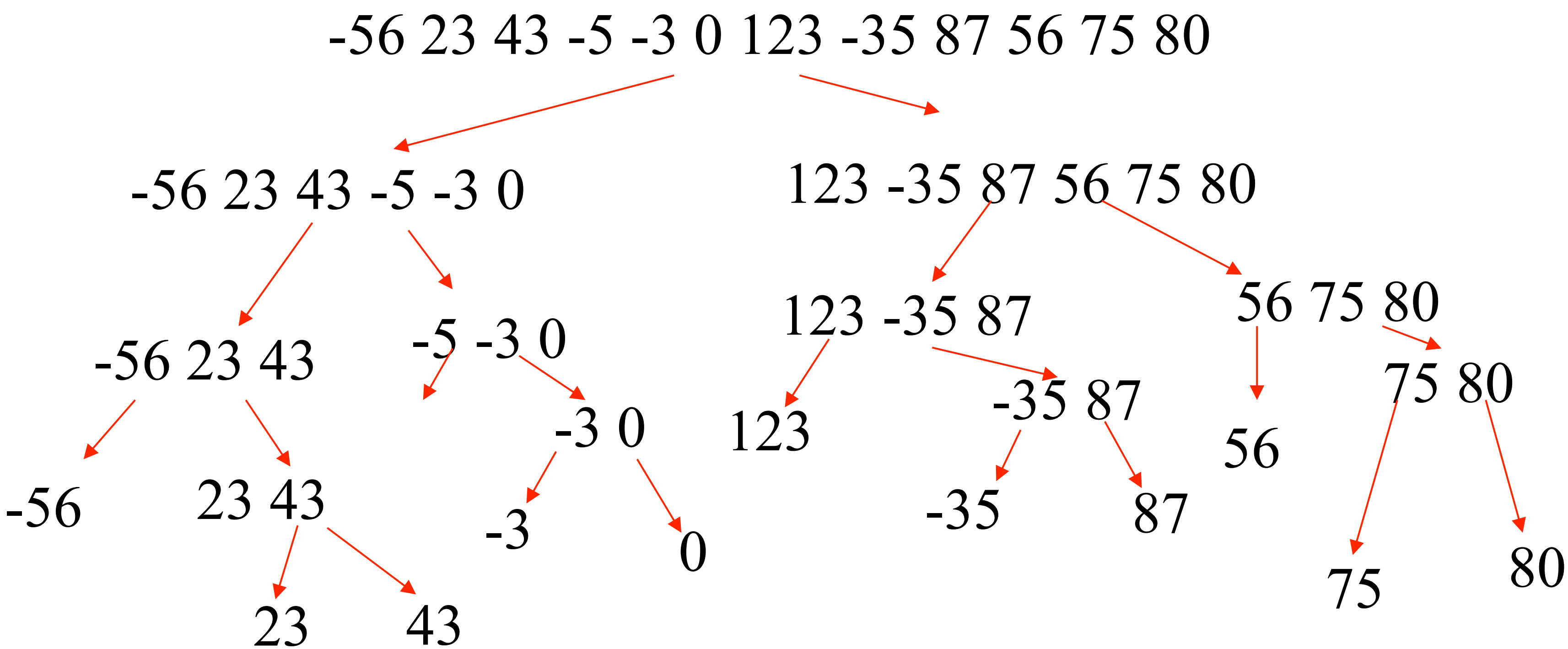
Merge Sort C program

```
void mergesort(int a[],int i,int j)
{
    int mid;

    if(i<j)     {
        mid=(i+j)/2;
    /* left recursion  */
        mergesort(a,i,mid);
    /* right recursion */
        mergesort(a,mid+1,j);
   /* merging of two sorted sub-arrays */
        merge(a,i,mid,mid+1,j);
     }
}
```

Merge Sort C program

```
void merge(int a[],int i1,int i2,int j1,int j2)
{
    int temp[50];    //array used for merging
    int i=i1,j=j1,k=0;

    while(i<=i2 && j<=j2)    //while elements in both lists
    {
        if(a[i]<a[j])
            temp[k++]=a[i++];
        else
            temp[k++]=a[j++];
    }

    while(i<=i2)    //copy remaining elements of the first list
        temp[k++]=a[i++];

    while(j<=j2)    //copy remaining elements of the second list
        temp[k++]=a[j++];

        for(i=i1,j=0;i<=j2;i++,j++)
        a[i]=temp[j];        //Transfer elements from temp[] back to a[]
}
```

Splitting Trace



-56 -35 -5 -3 0 23 43 56 75 80 87 123

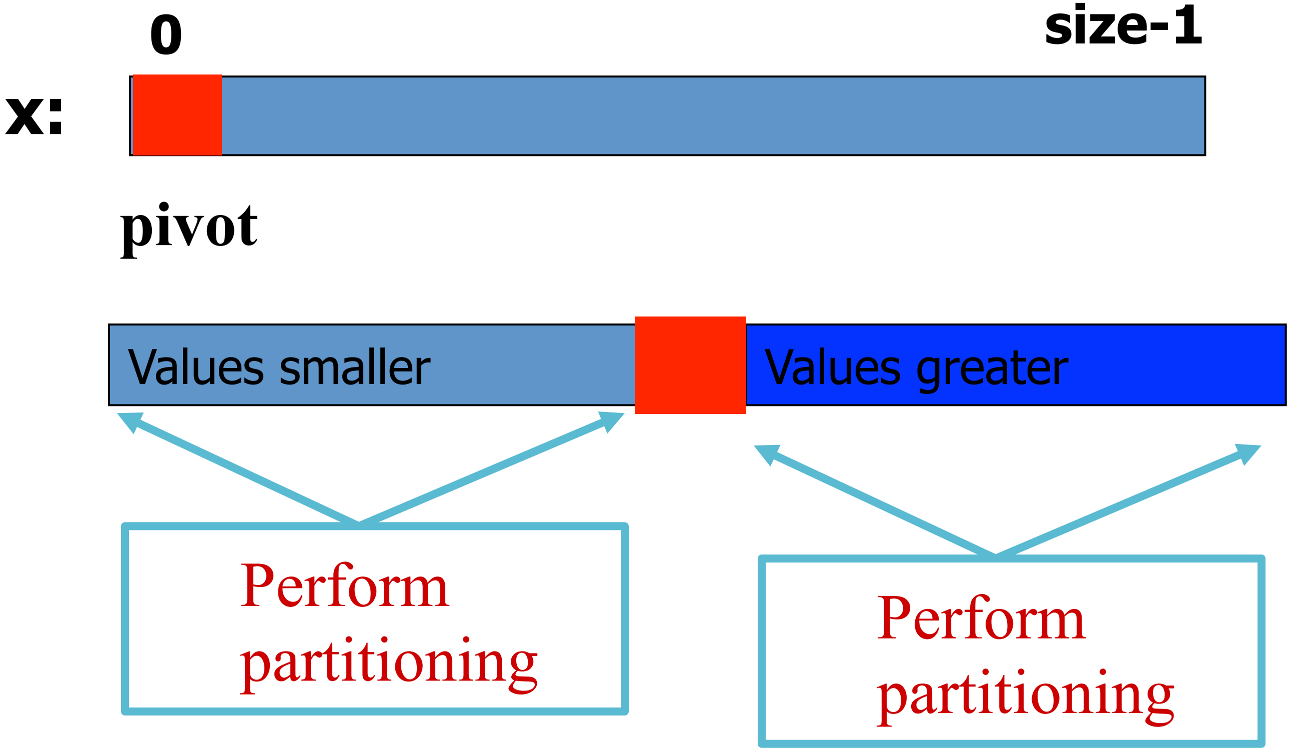
Space Complexity??

Worst Case: O(n.log(n))

Quicksort

- At every step, we select a ***pivot element*** in the list (usually the first element).
  - We put the pivot element in the final position of the sorted list.
  - All the elements less than or equal to the pivot element are to the left.
  - All the elements greater than the pivot element are to the right.

Partitioning



Quick Sort program

```
#include <stdio.h>
void quickSort( int[], int, int);
int partition( int[], int, int);

void main()
{
    int i,a[] = { 7, 12, 1, -2, 0, 15, 4, 11, 9};
    printf("\n\nUnsorted array is: ");
    for(i = 0; i < 9; ++i)
        printf(" %d ", a[i]);
    quickSort( a, 0, 8);
    printf("\n\nSorted array is: ");
    for(i = 0; i < 9; ++i)
        printf(" %d ", a[i]);
}
void quickSort( int a[], int l, int r)
{
    int j;
    if( l < r ) { // divide and conquer
        j = partition( a, l, r);
        quickSort( a, l, j-1);
        quickSort( a, j+1, r);
    }
}
```

Quick Sort program

```
int partition( int a[], int l, int r)
{
    int pivot, i, j, t;
    pivot = a[l];
    i = l;
    j = r+1;
    while( 1) {
            do {
                 ++i;
            } while(a[i]<=pivot && i<=r);

            do {
                 --j;
            } while( a[j] > pivot );
            if( i >= j ) break;
            t = a[i];
           a[i] = a[j];
           a[j] = t;
    }
    t = a[l];
    a[l] = a[j];
    a[j] = t;
    return j;
}
```

Trace of Partitioning

Input: 45 -56 78 90 -3 -6 123 0 -3 45 69 68

45 -56 78 90 -3 -6 123 0 -3 45 69 68

-6 -56 -3 0 -3 45 123 90 78 45 69 68

-56 -6 -3 0 -3 68 90 78 45 69 123

-3 0 -3 45 68 78 90 69

-3 0 69 78 90

Output: -56 -6 -3 -3 0 45 45 68 69 78 90 123

Time Complexity

- Partitioning with n elements.
  - No. of comparisons:

    n-1

Choice of pivot element affects the time complexity.

- Worst Case Performance:

  (n-1)+(n-2)+(n-3)+………. +1= n(n-1)/2

- Best Case performance:

  (n-1)+2((n-1)/2-1)+4(((n-1)/2-1)-1)/2-1) .. k steps

  = O(n. log(n))

$2^k=n$

Searching an Array:
Linear and Binary Search

Searching

- Check if a given element (key) occurs in the array.

Linear Search

- Basic idea:
  - Start at the beginning of the array.
  - Inspect every element to see if it matches the key.

```c
/* If key appears in a[0..size-1], print its location pos, where a[pos] == key. Else print unsuccessful search */
#include <stdio.h>
int main()
{
	int size,a[100],key,i,pos;

	printf("Enter the number of elements: ");
	scanf("%d",&size);
	printf("Enter the elements: ");
	for(i=0;i<size;i++)
		scanf("%d",&a[i]);
	printf("Enter the key element: ");
	scanf("%d",&key);
	for(pos=-1,i=0;i<size;i++) {    /* initializing pos as unsuccessful search*/
		if(a[i]==key) {
			pos=i;
			break;
		}
	}
	if(pos==-1)
	  printf("Unsuccessful search\n");
	else
	  printf("The element is present at %d position\n",pos+1);
	return 0;
}
```

Linear Search

```c
/* If key appears in a[0..size-1], print its location pos, where a[pos] == key. Else print unsuccessful search */
#include <stdio.h>
#include <stdlib.h>          /* for exit() function */
#define SIZE    100

void main()
{
    int size,a[SIZE],key,i,pos;

    printf("Enter the number of elements: ");
    scanf("%d",&size);
    if(size>SIZE) {      /* size is a variable, SIZE is not!! */
        printf("Array Size error!!! I am exiting .... \n");
        exit(0);
    }
    printf("Enter the elements: ");
    for(i=0;i<size;i++)a
        scanf("%d",&a[i]);
    printf("Enter the key element: ");
    scanf("%d",&key);
    for(pos=-1,i=0;i<size;i++) {    /* initializing pos as unsuccessful search*/
        if(a[i]==key) {
            pos=i;
            break;
        }
    }
    (pos==-1)? printf("Unsuccessful search\n"):printf("The element is present at %d position\n",pos+1);
}
```

1. If function type is void then nothing is to be returned from the function.
2. It is always good to assign a return data type (including void) with each function.
3. exit() will exit from the program without executing remaining statements of the program. This needs stdlib.h as header file.

Linear Search

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

- Trace the following calls :

  search 6;

  search 5;

Returns 5

Unsuccessful search

Linear Search

- Basic idea:
  - Start at the beginning of the array.
  - Inspect every element 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:
      (n + 1) / 2 search operations

Search on Sorted List

~~int x[ ]= {12,-3,78,67,6,50,19,10,11} ;~~

int x[ ]= {-3,6,10,~~11~~,12,~~19,50,67,78}~~ ;

- Trace the following calls :

  search 6;

  search 5;

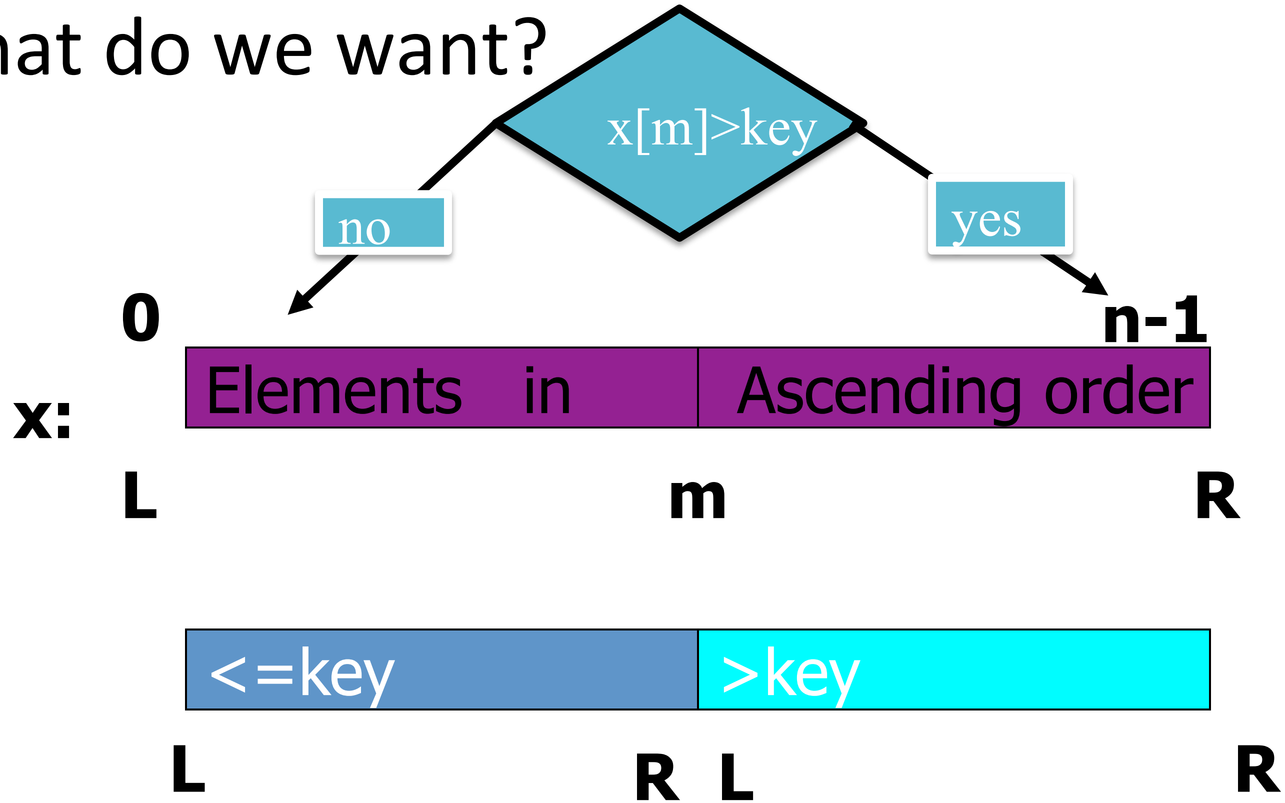
Binary Search

- 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 do we want?



  – Look at [(L+R)/2]. Move L or R to the middle depending on test.
  – Repeat search operation in the reduced interval.

Binary Search

```
/* If key appears in x[0..size-1], prints its location pos where
     x[pos]==key. If not found, print -1 */

int  main ()
{
     int x[100],size,key;
     int L, R, mid;
     ____________________;

     while ( ______________ )
     {
          _____________________;
     }
     ____________________ ;
}
```

The basic search iteration

```
/* If key appears in x[0..size-1], prints its location pos where x[pos]==key. If
     not found, print -1 */

int  main ()
{
     int x[100],size,key;
     int L, R, mid;
     ___________________;
     while ( _____________ )
     {
            mid = (L + R) / 2;
       if  (x[mid] > key)
            R = mid;
       else L = mid;
     }
     ___________________ ;
}
```

Loop termination

```
/* If key appears in x[0..size-1], prints its location pos where x[pos]==key. If
     not found, print -1 */

int  main ()
{
     int x[100],size,key;
     int L, R, mid;
     ____________________;
     while ( L+1 != R )
     {
             mid = (L + R) / 2;
       if  (x[mid] <= key)
             L = mid;
       else R = mid;
     }
     ____________________ ;
}
```

Print result

```
/* If key appears in x[0..size-1], prints its location pos where x[pos]==key. If
     not found, print -1 */

int  main ()
{
     int x[100],size,key;
     int L, R, mid;
     __________________;
     while ( L+1 != R )
     {
             mid = (L + R) / 2;
       if  (x[mid] <= key)
             L = mid;
       else R = mid;
     }
     if   (L >= 0  &&  x[L] = = key)   printf("%d",L);
     else  printf("-1");
}
```

Initialization

```
/* If key appears in x[0..size-1], prints its location pos where x[pos]==key. If not
      found, print -1 */

int  main ()
{
      int x[100],size,key;
      int L, R, mid;
_________________;
      L = -1;   R = size;
      while ( L+1 != R )
     {
             mid = (L + R) / 2;
        if  (x[mid] <= key)
              L = mid;
        else R = mid;
      }
      if   (L >= 0  &&  x[L] = = key)   printf("%d",L);
     else  printf("-1");
}
```

Complete C Program

```
/* If key appears in x[0..size-1], prints its location pos where x[pos]==key. If not found, print -1 */

void main ()
{
      int x[100],size,key;
      int L, R, mid;
      printf("Enter the number of elements: ");
      scanf("%d",&size);
      printf("Enter the elements: ");
      for(i=0;i<size;i++)
           scanf("%d",&a[i]);
      printf("Enter the key element: ");
      scanf("%d",&key);

      L = -1;   R = size;
      while ( L+1 != R )    {
            mid = (L + R) / 2;
        if  (x[mid] <= key)
             L = mid;
        else R = mid;
      }
      if   (L >= 0  &&  x[L] = = key)   printf(“%d”,L);
      else  printf(“-1”);
}
```

Binary Search Examples

Sorted array

-17	-5	3	6	12	21	45	63	50

Trace :

binsearch 3;

binsearch 145;

binsearch 45;

L= -1; R= 9; x[4]=12;
L= -1; R=4; x[1]= -5;
L= 1; R=4; x[2]=3;
L=2; R=4; x[3]=6;
L=2; R=3; return L;

Is it worth the trouble ?

- Suppose there are 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.

What is best case?
What is worst case?

Time Complexity

- If there are n elements in the array.
  - Number of searches required:

    $\log_2 n$

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

- 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.

$\log_2 64 = 6$

Homework

Modify the algorithm by checking equality with x[mid].