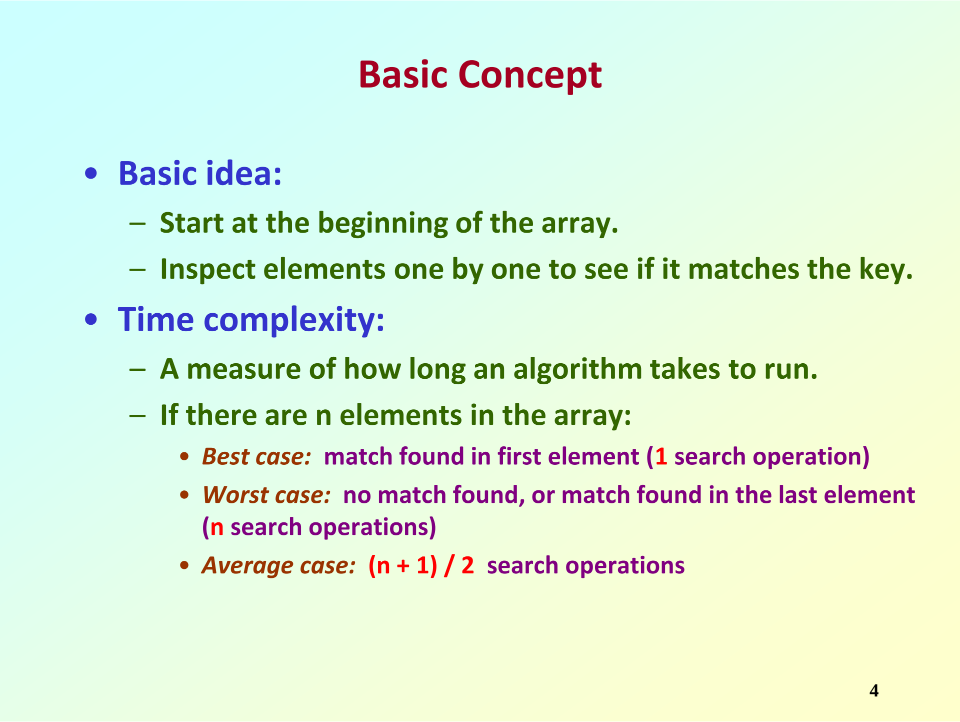
Searching Elements in an Array:

Linear and Binary Search

Searching

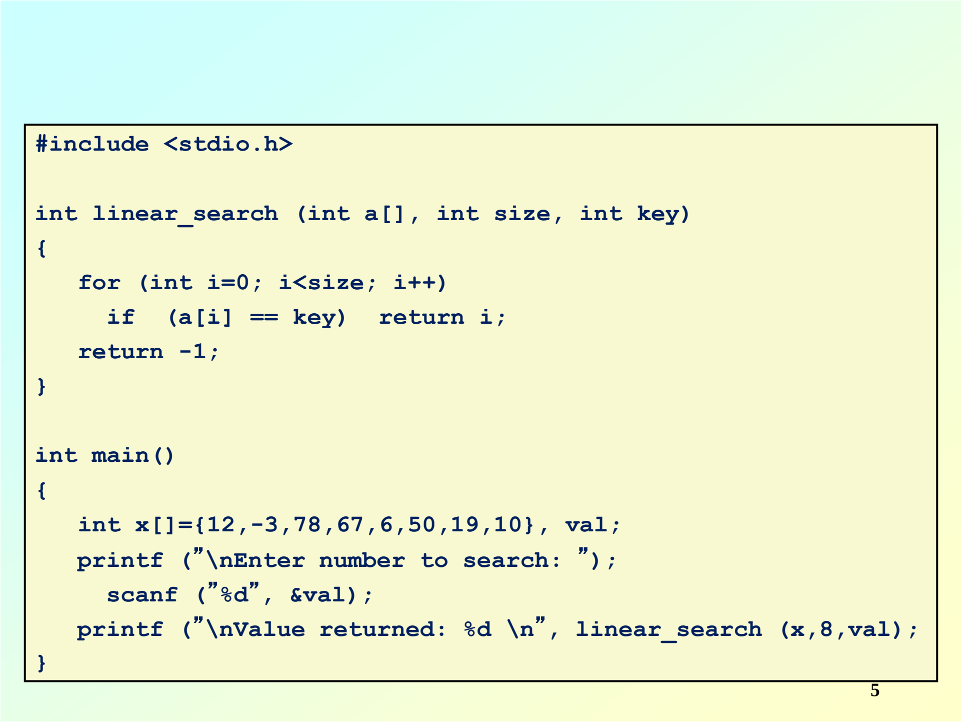
- Check if a given element (called ***key***) occurs in the array.
  - Example: array of student records; ***rollno*** can be the key.
- Two methods to be discussed:
  - If the array elements are unsorted.
    - Linear search
  - If the array elements are sorted.
    - Binary search

Linear Search

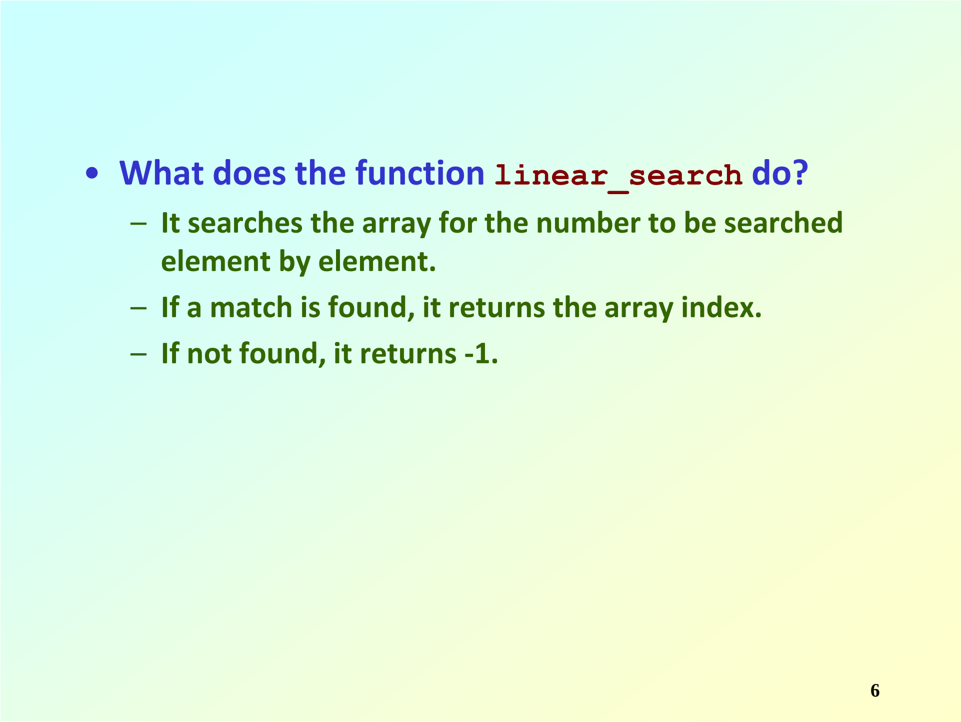


Basic Concept

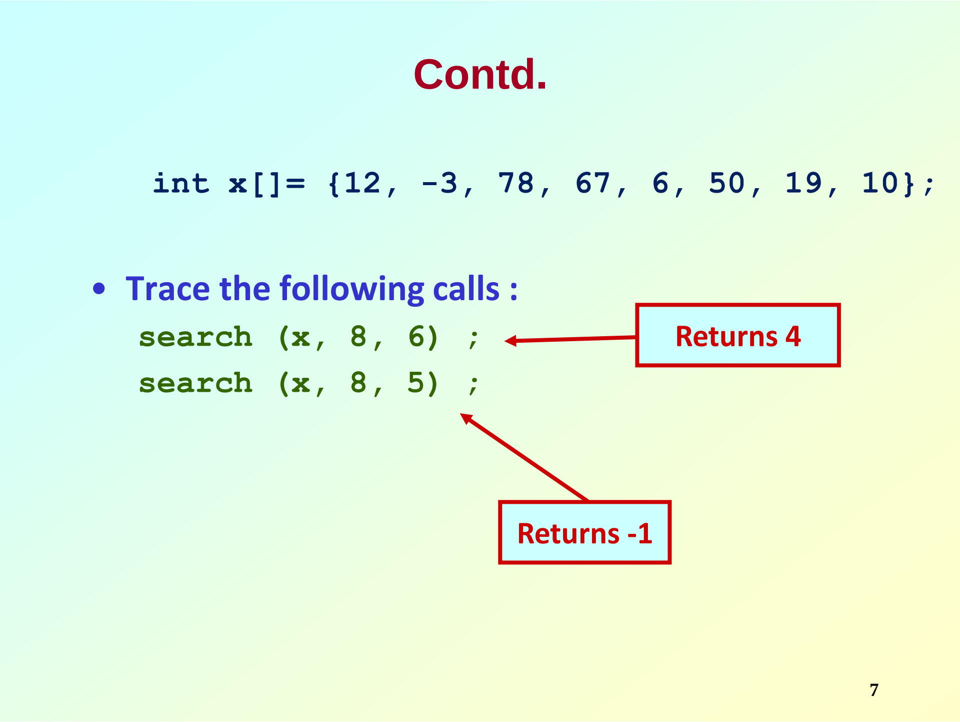
- **Basic idea:**
  - Start at the beginning of the array.
  - Inspect elements one by one to see if it matches the key.
- **Time complexity:**
  - A measure of how long an algorithm takes to run.
  - If there are n elements in the array:
    - *Best case:* match found in first element (1 search operation)
    - *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};
```

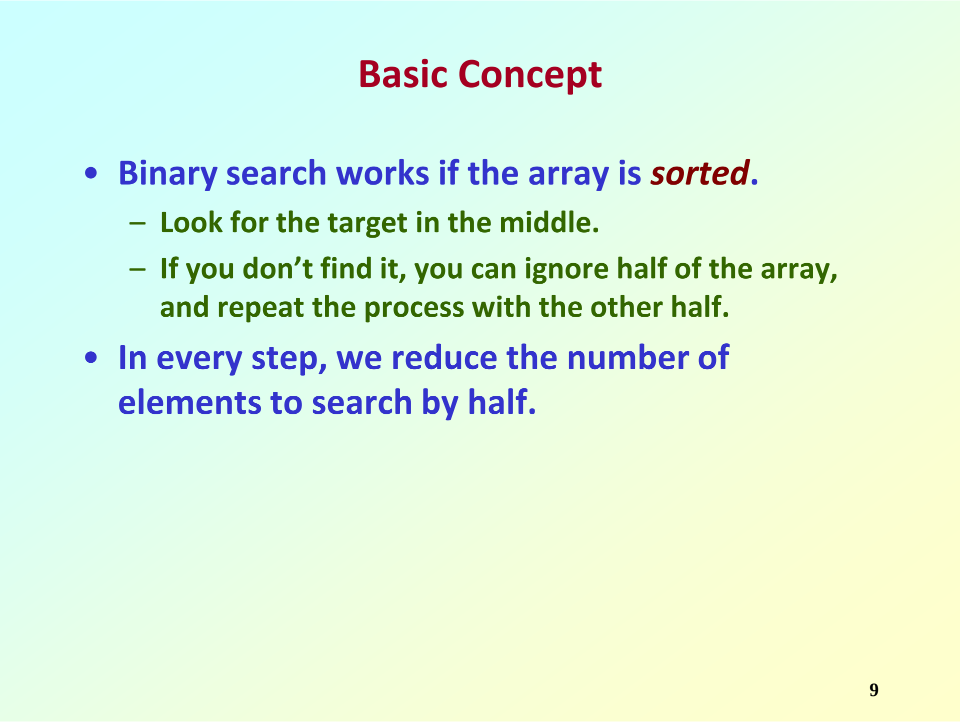
- Trace the following calls :

```
search (x, 8, 6) ;
search (x, 8, 5) ;
```

Returns 4

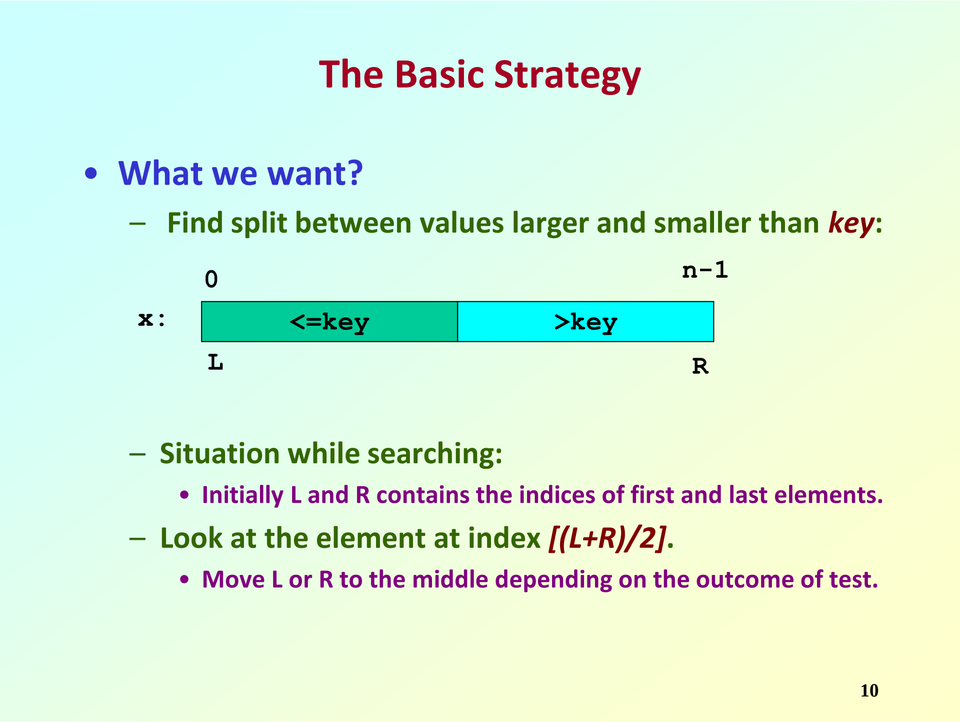
Returns -1

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 by half.

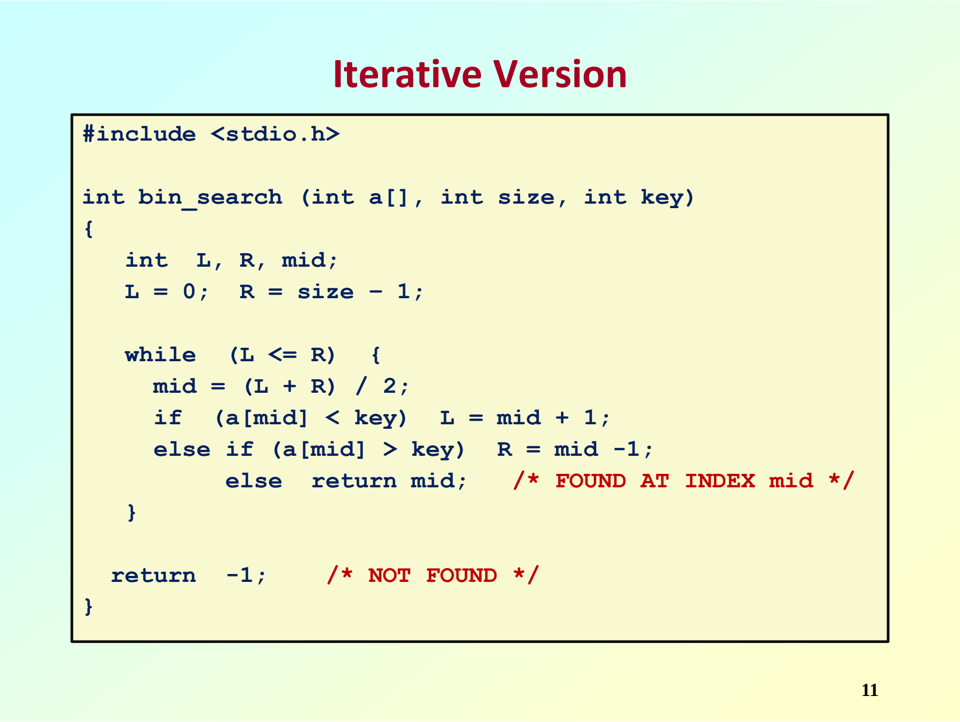


The Basic Strategy

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

0
n-1
x:
<=key
>key
L
R

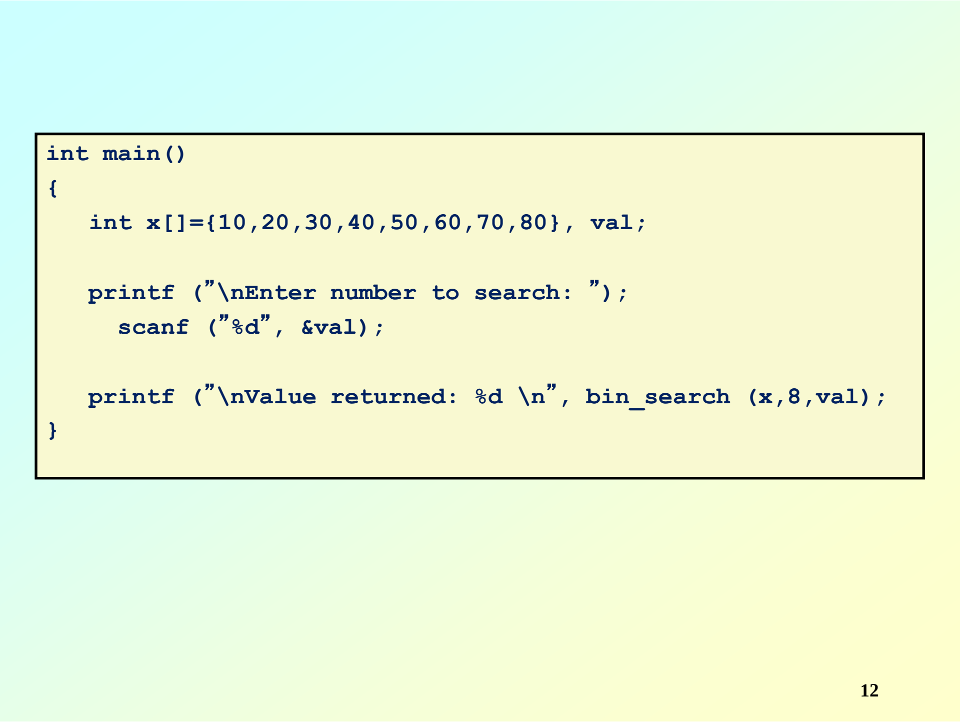
  - Situation while searching:
    - Initially L and R contains the indices of first and last elements.
  - Look at the element at index ***[(L+R)/2]***.
    - Move L or R to the middle depending on the outcome of test.



Iterative Version

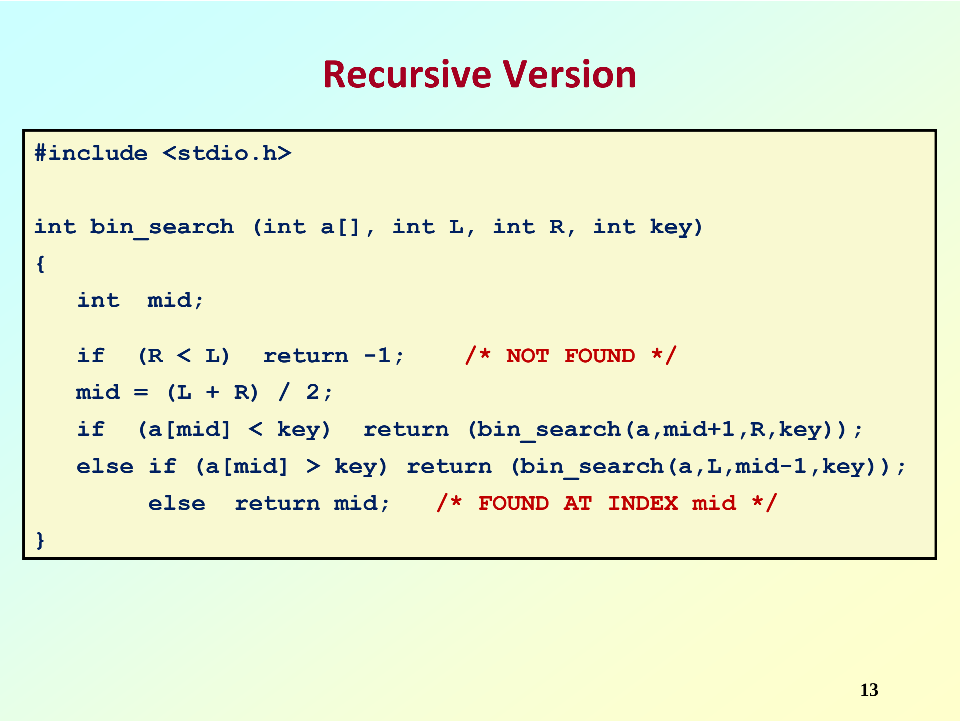
```
#include <stdio.h>

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 */
}
```

11



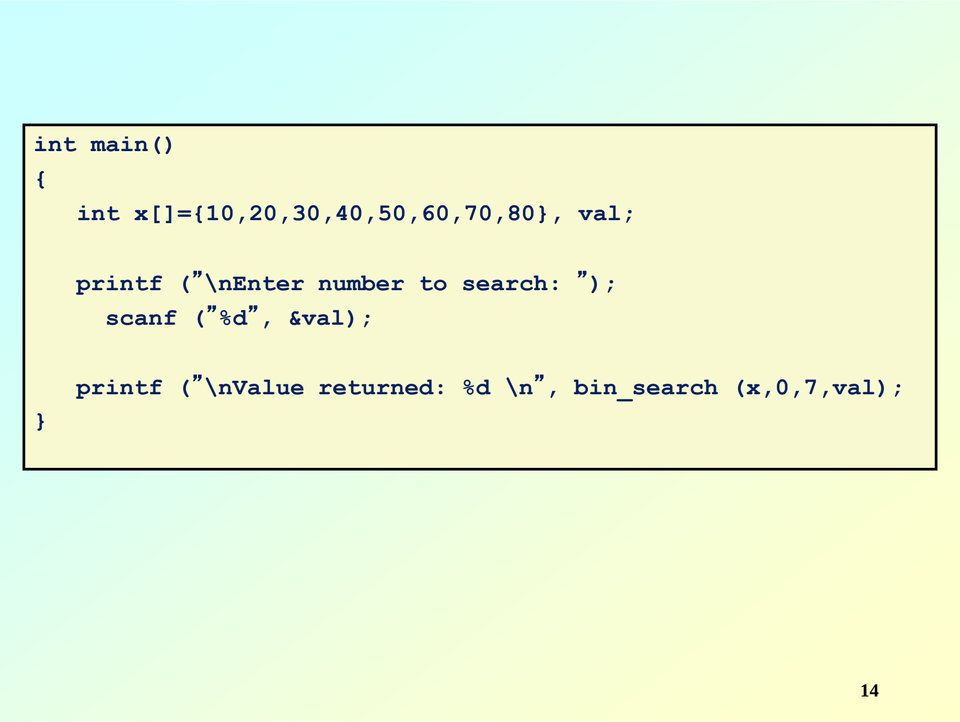
```
int main()
{
   int x[]={10,20,30,40,50,60,70,80}, val;

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

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

12



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 present in *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 3rd compare, left with 125 elements.
  - After at most 10 steps, you are done.

Time Complexity

- If there are n elements in the array.
  - Number of searches required in the worst case: $\log_2 n$
- 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.

$k = \log_2 n$

$\log_2 64 = 6$
$\log_2 1024 = 10$

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.

Example

- **Original list:**

  `10, 30, 20, 80, 70, 10, 60, 40, 70`

- **Sorted in non-decreasing order:**

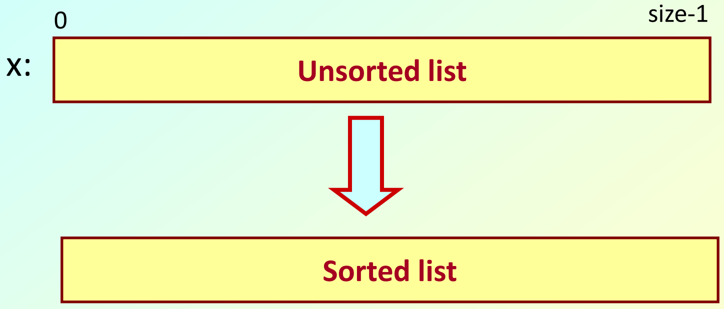
  `10, 10, 20, 30, 40, 60, 70, 70, 80`

- **Sorted in non-increasing order:**

  `80, 70, 70, 60, 40, 30, 20, 10, 10`

Sorting Problem

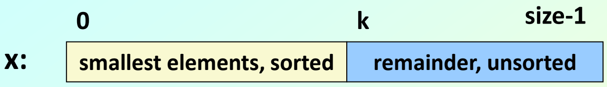
- **What we want ?**
  - **Sort the given data sorted in the specified order**



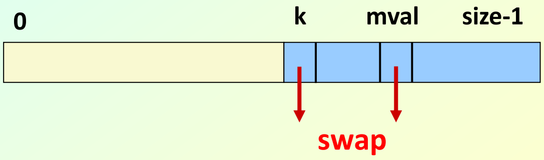
Selection Sort

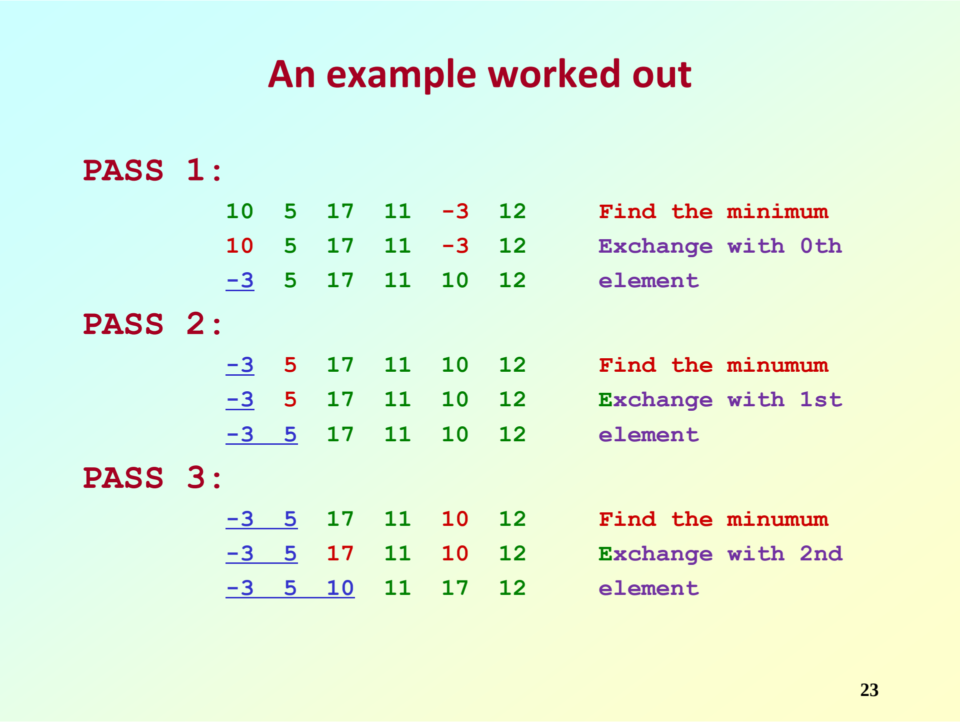
How it works?

- **General situation :**


0
k
size-1
x:
smallest elements, sorted
remainder, unsorted

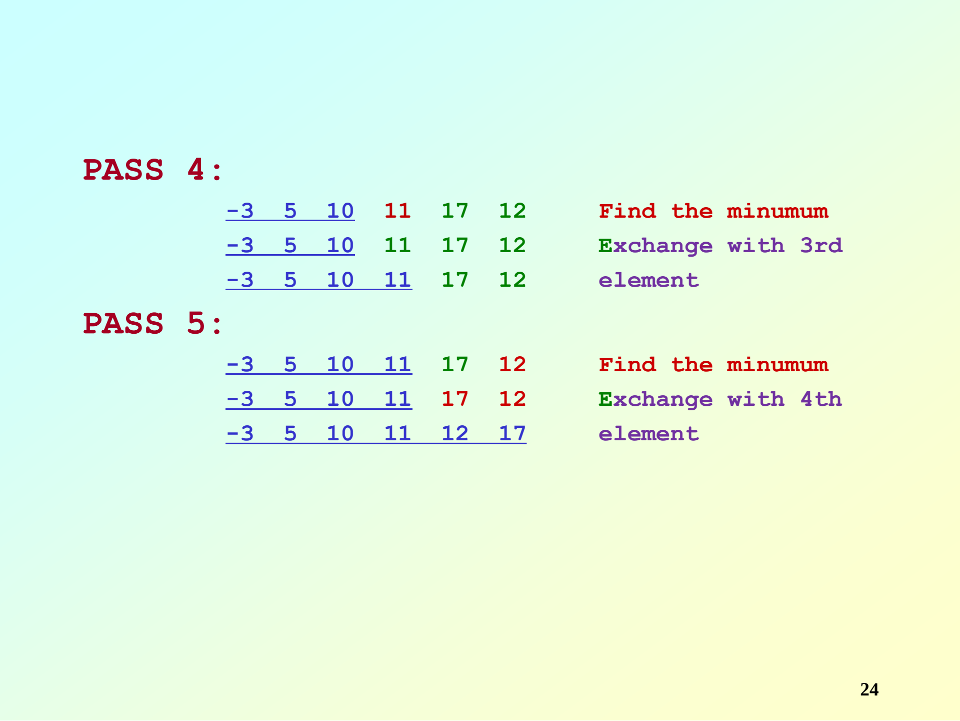
- **Steps :**
  - Find smallest element, `mval`, in `x[k..size-1]`
  - Swap smallest element with `x[k]`, then increase `k` by 1


0
k
mval
size-1
swap

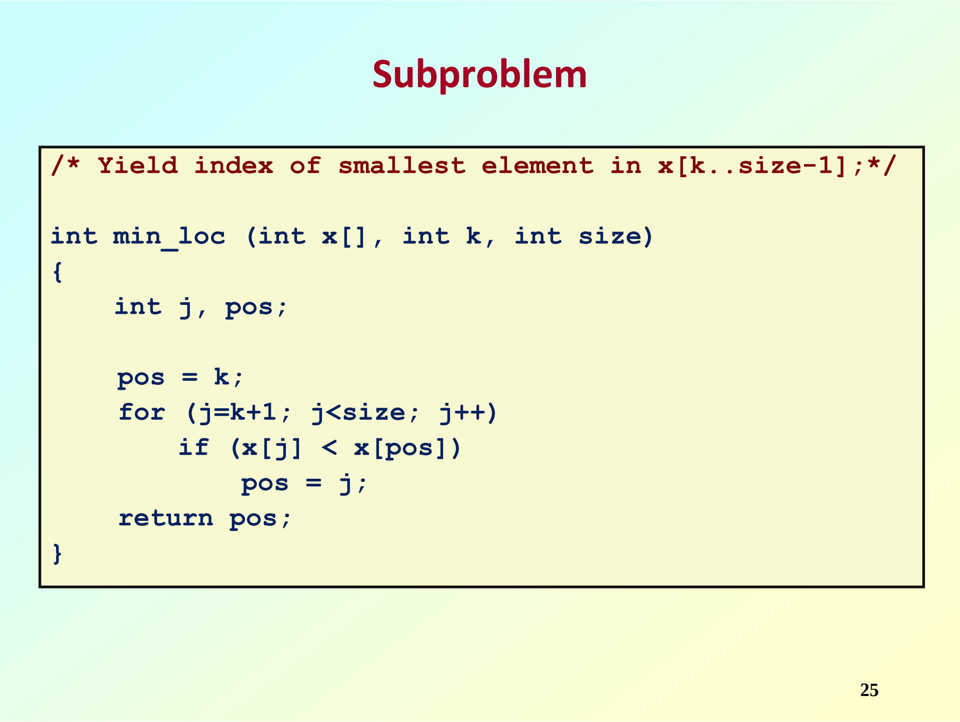


An example worked out

```
PASS 1:
          10   5  17  11  -3  12     Find the minimum
          10   5  17  11  -3  12     Exchange with 0th
          -3   5  17  11  10  12     element
PASS 2:
          -3   5  17  11  10  12     Find the minumum
          -3   5  17  11  10  12     Exchange with 1st
          -3   5  17  11  10  12     element
PASS 3:
          -3   5  17  11  10  12     Find the minumum
          -3   5  17  11  10  12     Exchange with 2nd
          -3   5  10  11  17  12     element
```

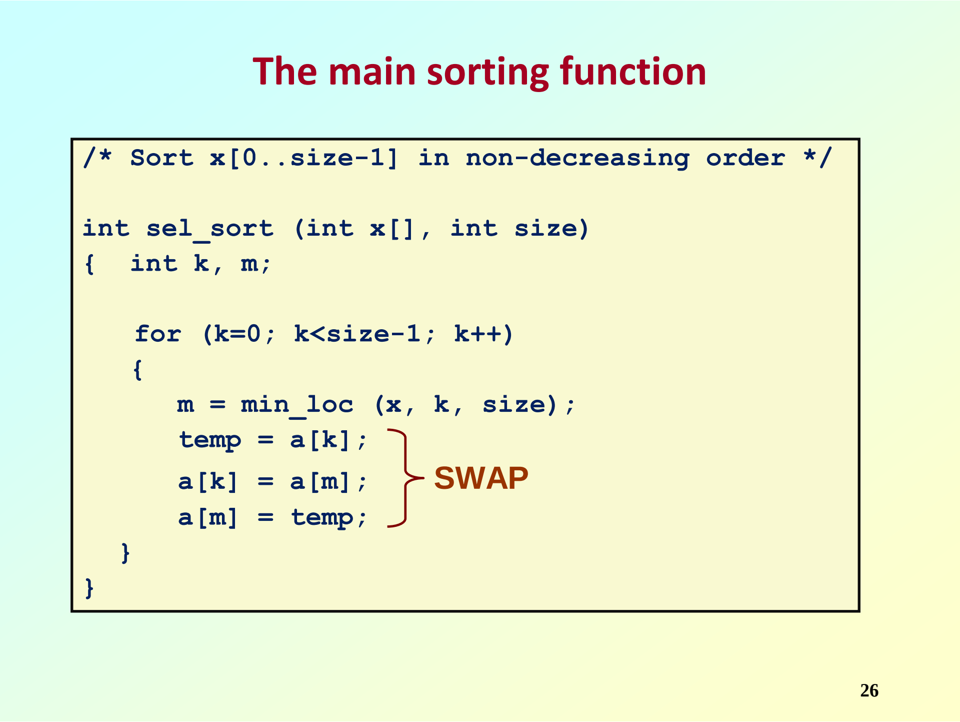


```
PASS 4:
          -3   5  10  11  17  12     Find the minumum
          -3   5  10  11  17  12     Exchange with 3rd
          -3   5  10  11  17  12     element
PASS 5:
          -3   5  10  11  17  12     Find the minumum
          -3   5  10  11  17  12     Exchange with 4th
          -3   5  10  11  12  17     element
```



Subproblem

```
/* Yield index of smallest element in x[k..size-1];*/

int min_loc (int x[], int k, int size)
{
    int j, pos;

    pos = k;
    for (j=k+1; j<size; j++)
        if (x[j] < x[pos])
            pos = j;
    return pos;
}
```



The main sorting function

```
/* Sort x[0..size-1] in non-decreasing order */

int sel_sort (int x[], int size)
{   int k, m;

    for (k=0; k<size-1; k++)
    {
       m = min_loc (x, k, size);
       temp = a[k];
       a[k] = a[m];    } SWAP
       a[m] = temp;
    }
}
```

```
int main()
{
  int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
  int i;
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
  sel_sort(x,12);
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
}
```

```
-45 89 -65 87 0 3 -23 19 56 21 76 -50
-65 -50 -45 -23 0 3 19 21 56 76 87 89
```

Analysis

- How many steps are needed to sort n items ?
  - Total number of steps proportional to n^2.
  - No. of comparisons?

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

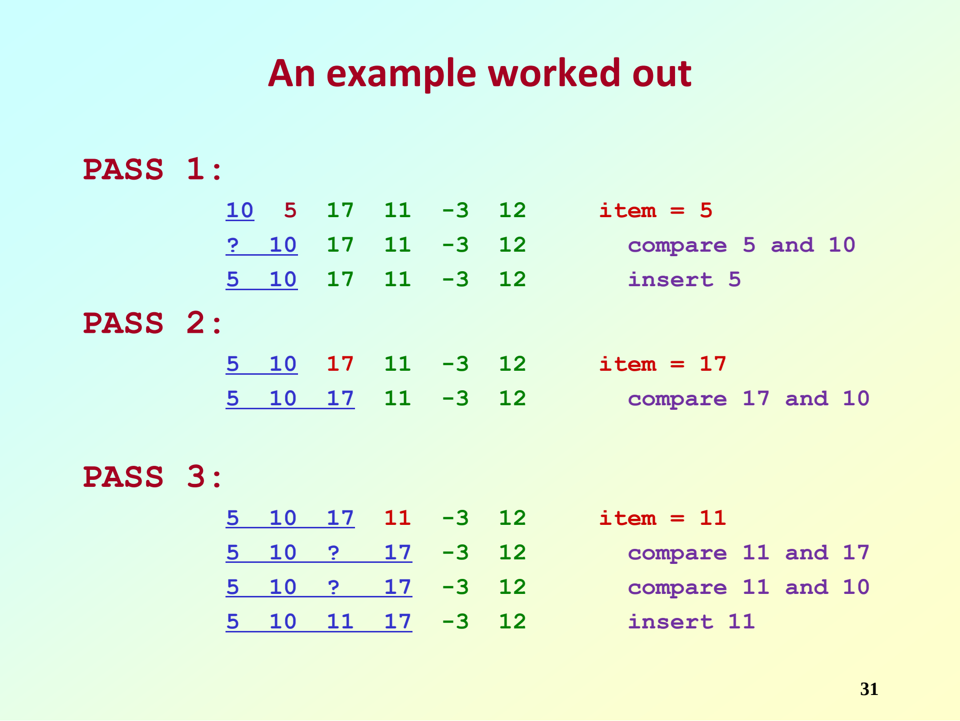
    Of the order of n^2

  - Worst Case? Best Case? Average Case?

Insertion Sort

Basic Idea

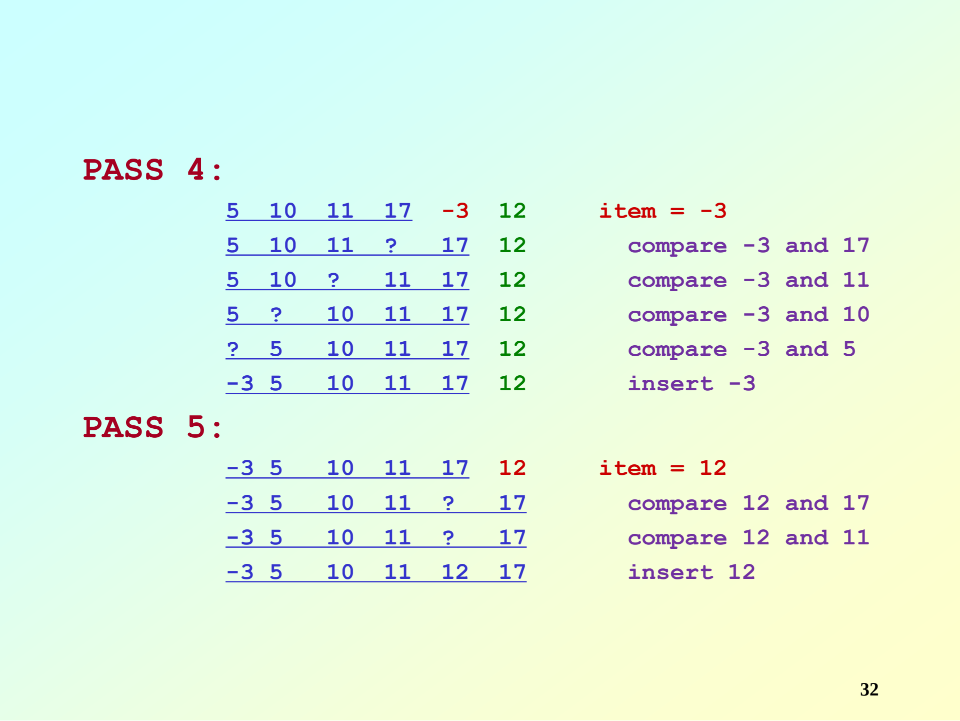
- Insert elements one at a time, and create a partial sorted list.
  - Sorted list of 2 elements, 3 elements, 4 elements, and so on.
- In general, in every iteration an element is compared with all the elements before it.
- After finding the position of insertion, space is created for it by shifting the other elements and the desired element is then inserted at the suitable position.
- This procedure is repeated for all the elements in the list.



An example worked out

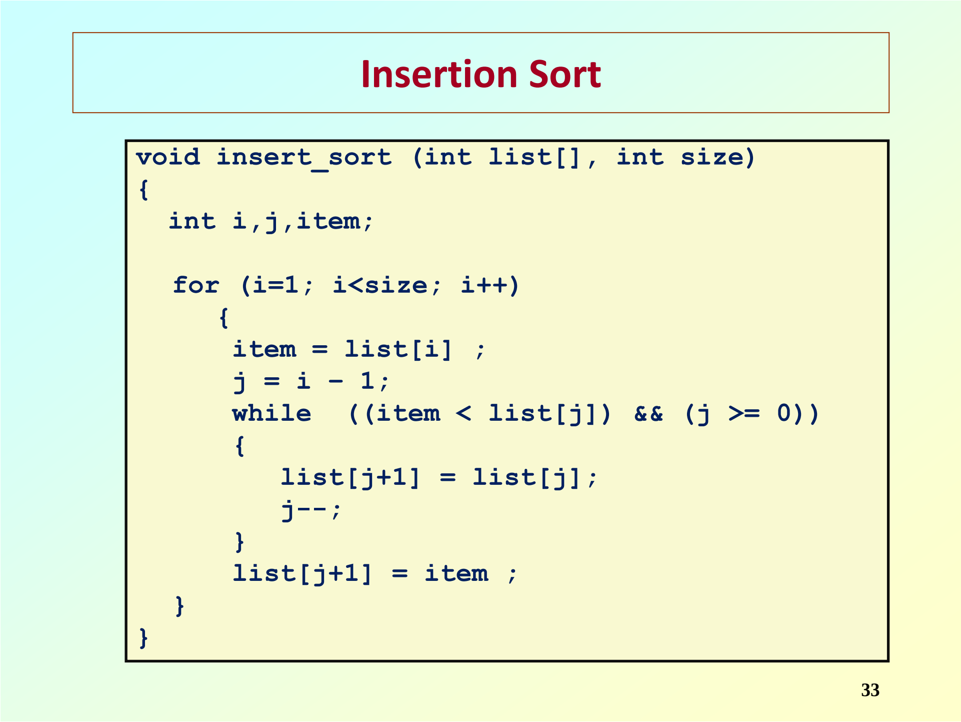
```
PASS 1:
           10  5  17  11  -3  12      item = 5
           ?  10  17  11  -3  12        compare 5 and 10
           5  10  17  11  -3  12        insert 5
PASS 2:
           5  10  17  11  -3  12      item = 17
           5  10  17  11  -3  12        compare 17 and 10

PASS 3:
           5  10  17  11  -3  12      item = 11
           5  10  ?   17  -3  12        compare 11 and 17
           5  10  ?   17  -3  12        compare 11 and 10
           5  10  11  17  -3  12        insert 11
```

31



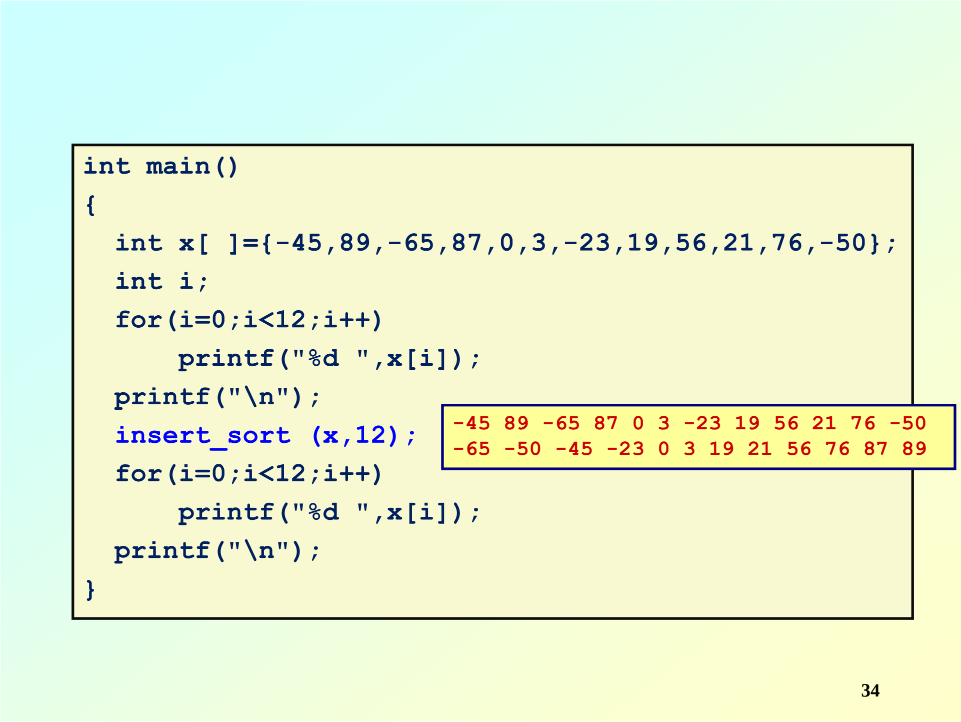
```
PASS 4:
           5  10  11  17  -3  12      item = -3
           5  10  11  ?   17  12        compare -3 and 17
           5  10  ?   11  17  12        compare -3 and 11
           5  ?   10  11  17  12        compare -3 and 10
           ?  5   10  11  17  12        compare -3 and 5
           -3 5   10  11  17  12        insert -3
PASS 5:
           -3 5   10  11  17  12      item = 12
           -3 5   10  11  ?   17        compare 12 and 17
           -3 5   10  11  ?   17        compare 12 and 11
           -3 5   10  11  12  17        insert 12
```

32



Insertion Sort

```
void insert_sort (int list[], int size)
{
  int i,j,item;

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



```
int main()
{
  int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
  int i;
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
  insert_sort (x,12);
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
}
```

```
-45 89 -65 87 0 3 -23 19 56 21 76 -50
-65 -50 -45 -23 0 3 19 21 56 76 87 89
```

Time Complexity

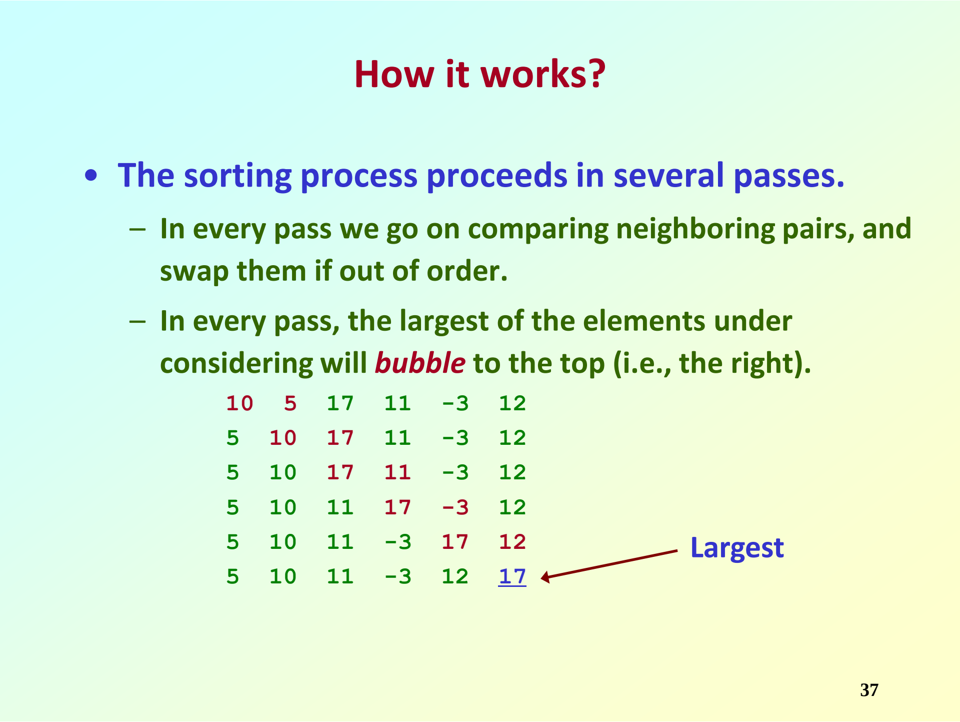
- **Number of comparisons and shifting:**
  - **Worst case?**

    `1 + 2 + 3 + …… + (n-1) = n(n-1)/2`

  - **Best case?**

    `1 + 1 + …… to (n-1) terms = (n-1)`

Bubble Sort

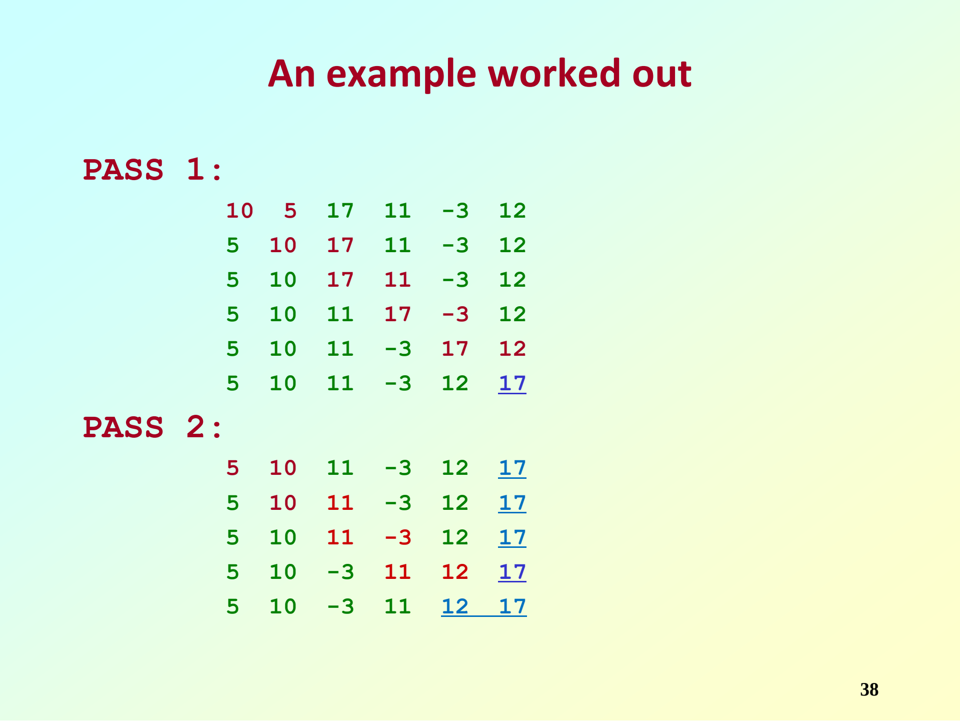


How it works?

- The sorting process proceeds in several passes.
  - In every pass we go on comparing neighboring pairs, and swap them if out of order.
  - In every pass, the largest of the elements under considering will *bubble* to the top (i.e., the right).

```
10   5   17   11   -3   12
5   10   17   11   -3   12
5   10   17   11   -3   12
5   10   11   17   -3   12
5   10   11   -3   17   12
5   10   11   -3   12   17
```

Largest



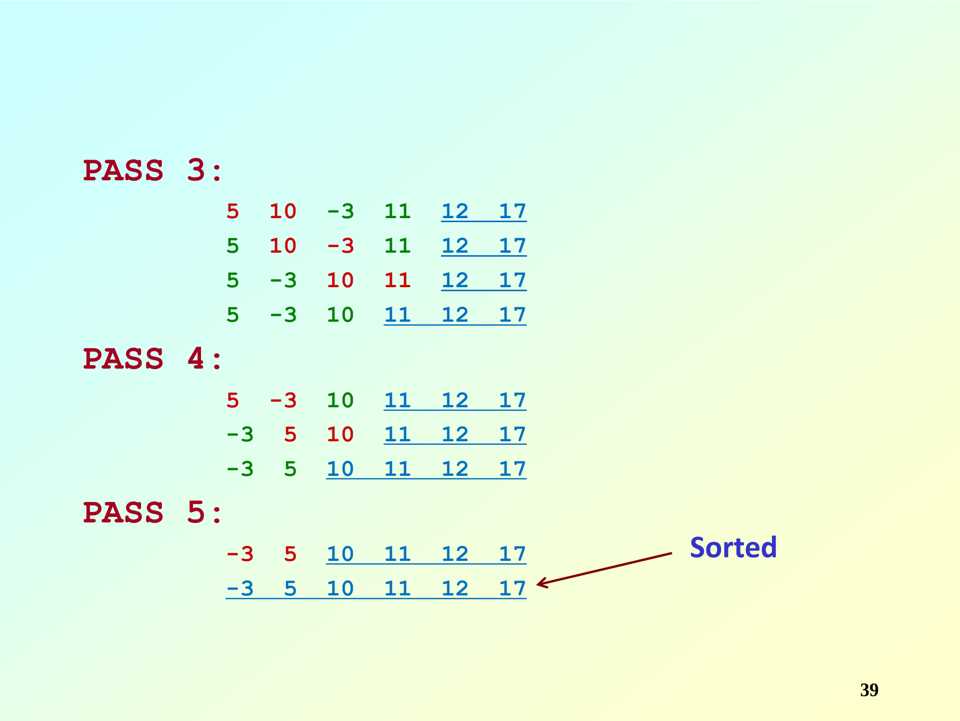
An example worked out

PASS 1:

```
10   5   17   11   -3   12
5   10   17   11   -3   12
5   10   17   11   -3   12
5   10   11   17   -3   12
5   10   11   -3   17   12
5   10   11   -3   12   17
```

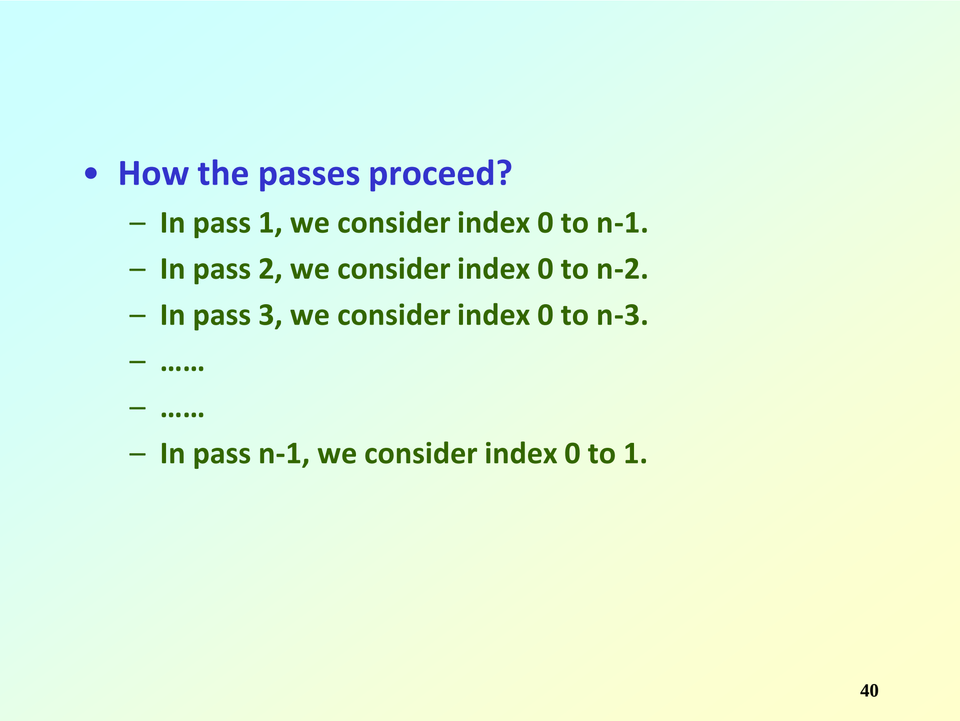
PASS 2:

```
5   10   11   -3   12   17
5   10   11   -3   12   17
5   10   11   -3   12   17
5   10   -3   11   12   17
5   10   -3   11   12   17
```



```
PASS 3:
          5   10   -3   11   12   17
          5   10   -3   11   12   17
          5   -3   10   11   12   17
          5   -3   10   11   12   17
PASS 4:
          5   -3   10   11   12   17
         -3    5   10   11   12   17
         -3    5   10   11   12   17
PASS 5:
         -3    5   10   11   12   17
         -3    5   10   11   12   17
```

Sorted



- **How the passes proceed?**
  - In pass 1, we consider index 0 to n-1.
  - In pass 2, we consider index 0 to n-2.
  - In pass 3, we consider index 0 to n-3.
  - ……
  - ……
  - In pass n-1, we consider index 0 to 1.

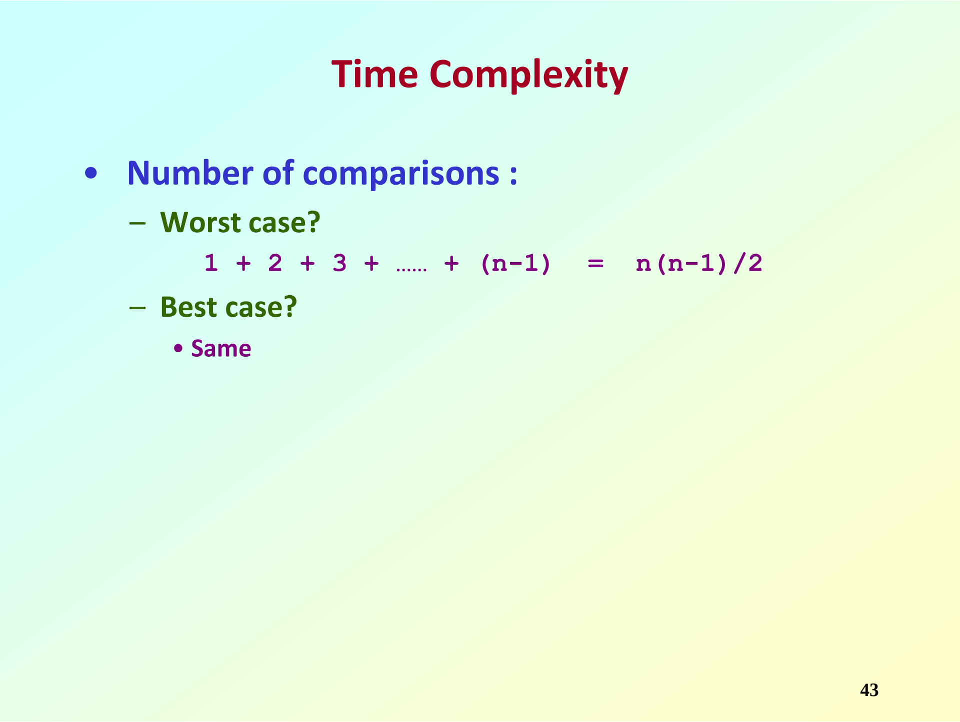
Bubble Sort

```
void swap(int *x, int *y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

```
void bubble_sort
         (int x[], int n)
{
  int i,j;

  for (i=n-1; i>0; i--)
    for (j=0; j<i; j++)
      if (x[j] > x[j+1])
        swap(&x[j],&x[j+1]);
}
```

```
int main()
{
  int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
  int i;
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
  bubble_sort (x,12);
  for(i=0;i<12;i++)
      printf("%d ",x[i]);
  printf("\n");
}
```

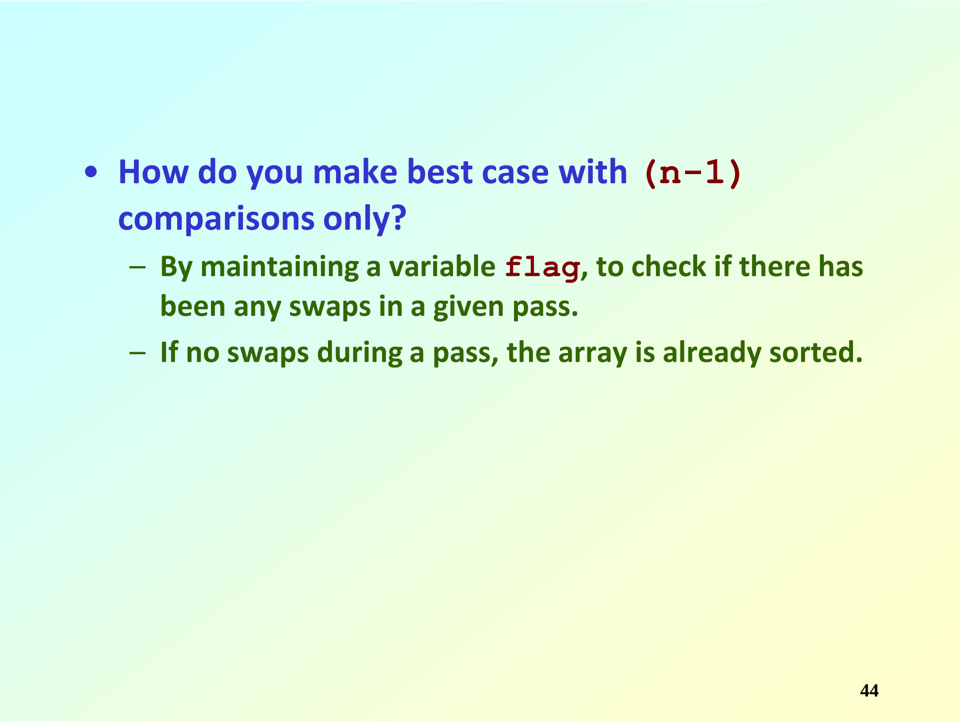
```
-45 89 -65 87 0 3 -23 19 56 21 76 -50
-65 -50 -45 -23 0 3 19 21 56 76 87 89
```



Time Complexity

- Number of comparisons :
  - Worst case?

    `1 + 2 + 3 + …… + (n-1)   =  n(n-1)/2`
  - Best case?
    - Same



- How do you make best case with `(n-1)` comparisons only?
  - By maintaining a variable `flag`, to check if there has been any swaps in a given pass.
  - If no swaps during a pass, the array is already sorted.

```
void bubble_sort (int x[], int n)
{
  int i, j, flag;

  for (i=n-1; i>0; i--)
  {
      flag = 0;
      for (j=0; j<i; j++)
         if (x[j] > x[j+1])
         {
            swap(&x[j],&x[j+1]);
            flag = 1;
         }
      if (flag == 0) return;
  }
```

Some Efficient Sorting Algorithms

- Two of the most popular sorting algorithms are based on divide-and-conquer approach.
  - Quick sort
  - Merge sort
- Basic concept (divide-and-conquer method):

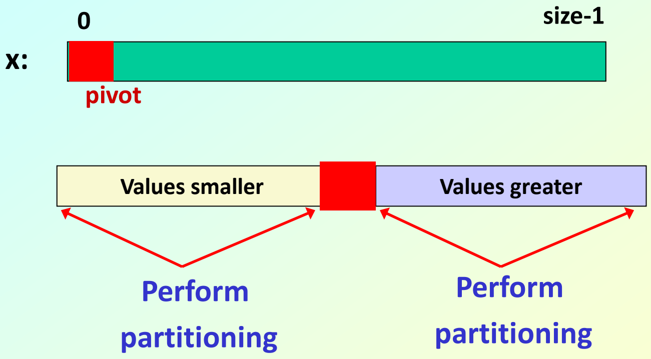
```
sort (list)
{
   if the list has greater than 1 elements
   {
      Partition the list into lowlist and highlist;
      sort (lowlist);
      sort (highlist);
      combine (lowlist, highlist);
   }
}
```

Quick Sort

How it works?

- 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



0

size-1

x:

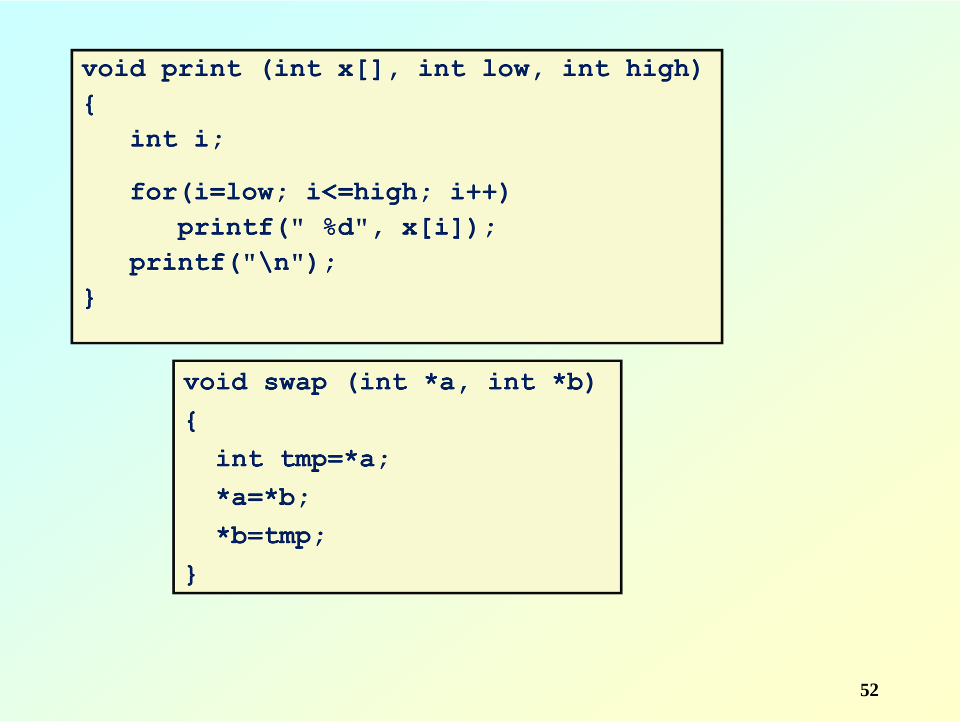
pivot

Values smaller

Values greater

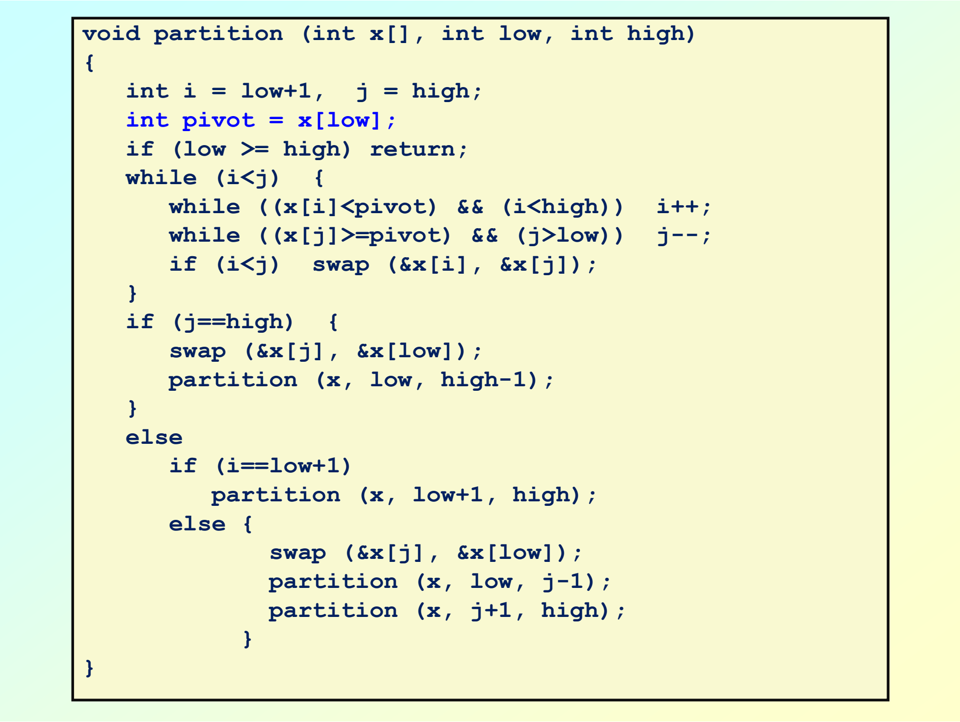
Perform partitioning

Perform partitioning



```c
void print (int x[], int low, int high)
{
   int i;

   for(i=low; i<=high; i++)
      printf(" %d", x[i]);
   printf("\n");
}
```

```c
void swap (int *a, int *b)
{
   int tmp=*a;
   *a=*b;
   *b=tmp;
}
```



```c
void partition (int x[], int low, int high)
{
   int i = low+1,  j = high;
   int pivot = x[low];
   if (low >= high) return;
   while (i<j)  {
      while ((x[i]<pivot) && (i<high))  i++;
      while ((x[j]>=pivot) && (j>low))  j--;
      if (i<j)  swap (&x[i], &x[j]);
   }
   if (j==high)  {
      swap (&x[j], &x[low]);
      partition (x, low, high-1);
   }
   else
      if (i==low+1)
         partition (x, low+1, high);
      else {
             swap (&x[j], &x[low]);
             partition (x, low, j-1);
             partition (x, j+1, high);
           }
}
```

```
int main (int argc, char *argv[])
{
   int x[] = {-56,23,43,-5,-3,0,123,-35,87,56,75,80};
   int i=0;
   int num;

   num = 12;   /* Number of elements */

   partition(x,0,num-1);

   printf("Sorted list: ");
   print (x,0,num-1);
}
```

Trace of Partitioning: an example

```
        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

   Sorted list:  -56 -6 -3 -3 0 45 45 68 69 78 90 123
```

Time Complexity

- Worst case:

  n^2 ==> list is already sorted

- Average case:

  $n \log_2 n$

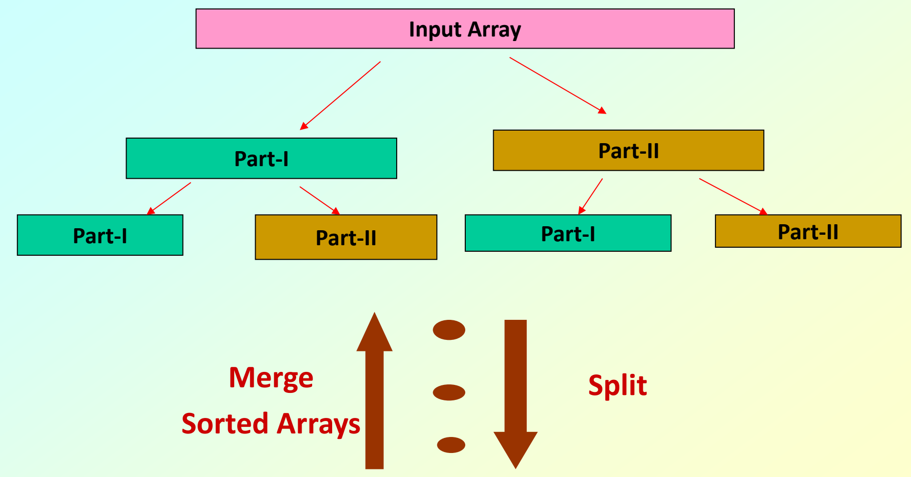
- Statistically, quick sort has been found to be one of the fastest algorithms.

56

Merge Sort

57

Merge Sort



Input Array

Part-I

Part-II

Part-I

Part-II

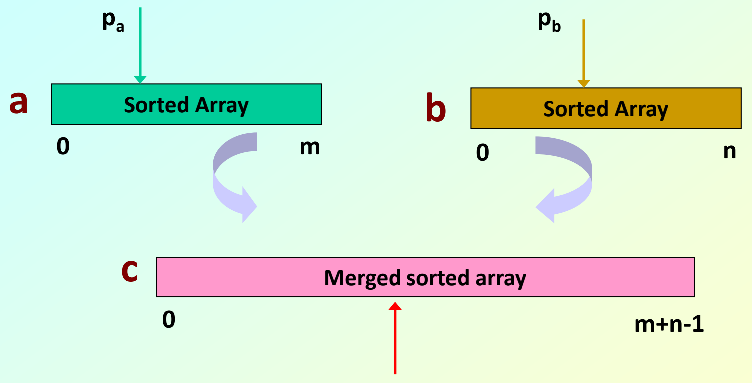
Part-I

Part-II

Merge
Sorted Arrays

Split

Merging two sorted arrays



p_a

p_b

a Sorted Array

0 m

b Sorted Array

0 n

c Merged sorted array

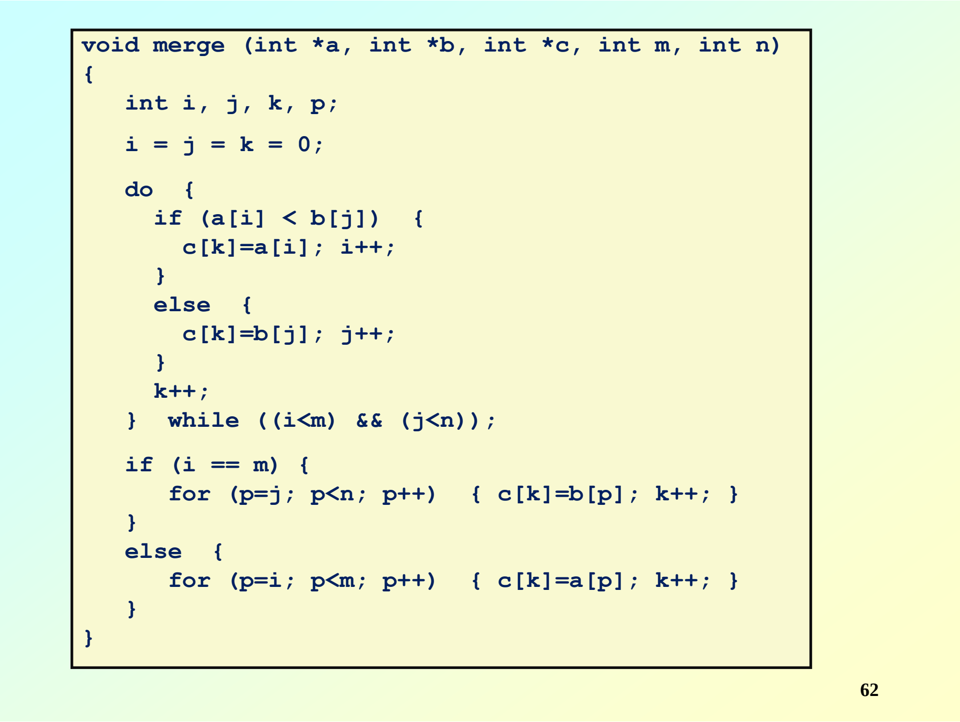
0 m+n-1

Move and copy elements pointed by p_a if its value is smaller than the element pointed by p_b in (m+n-1) operations; otherwise, copy elements pointed by p_b .

Example

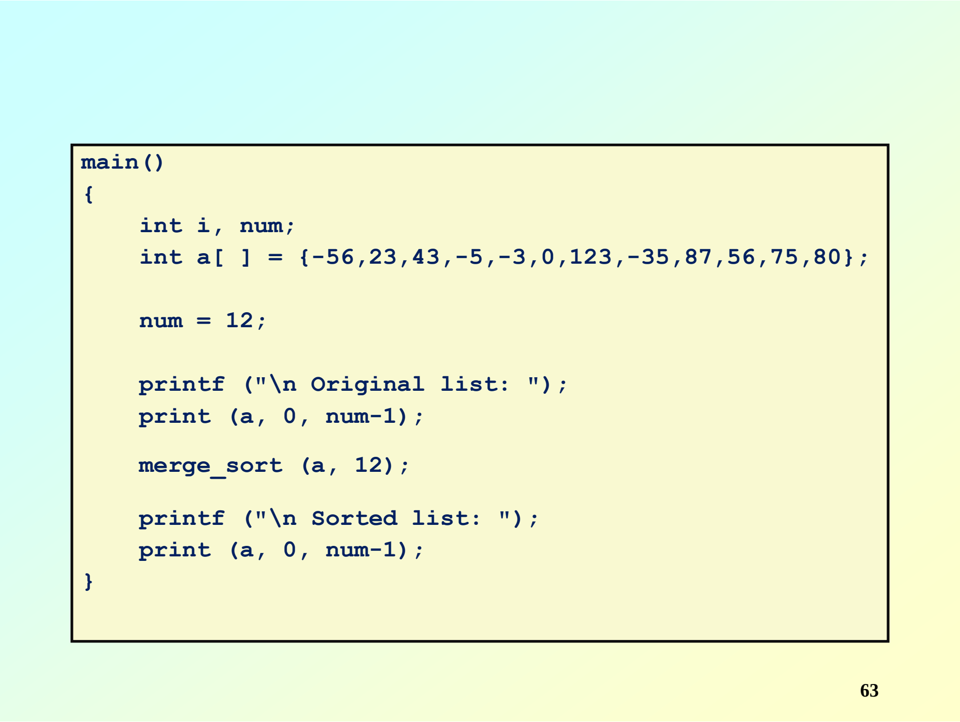
- Initial array A contains 14 elements:
  - 66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30
- Pass 1 :: Merge each pair of elements
  - (33, 66) (22, 40) (55, 88) (11, 60) (20, 80) (44, 50) (30, 70)
- Pass 2 :: Merge each pair of pairs
  - (22, 33, 40, 66) (11, 55, 60, 88) (20, 44, 50, 80) (30, 77)
- Pass 3 :: Merge each pair of sorted quadruplets
  - (11, 22, 33, 40, 55, 60, 66, 88) (20, 30, 44, 50, 77, 80)
- Pass 4 :: Merge the two sorted subarrays to get the final list
  - (11, 20, 22, 30, 33, 40, 44, 50, 55, 60, 66, 77, 80, 88)

```
void merge_sort (int *a, int n)
{
   int i, j, k, m;
   int *b, *c;

   if (n>1)  {
      k = n/2;
      m = n-k;
      b = (int *) malloc(k*sizeof(int));
      c = (int *) malloc(m*sizeof(int));

      for (i=0; i<k; i++)
         b[i]=a[i];
      for (j=k; j<n; j++)
         c[j-k]=a[j];

      merge_sort (b, k);
      merge_sort (c, m);
      merge (b, c, a, k, m);
      free(b); free(c);
   }
}
```



```
void merge (int *a, int *b, int *c, int m, int n)
{
   int i, j, k, p;
   i = j = k = 0;
   do  {
     if (a[i] < b[j])  {
       c[k]=a[i]; i++;
     }
     else  {
       c[k]=b[j]; j++;
     }
     k++;
   }  while ((i<m) && (j<n));
   if (i == m) {
      for (p=j; p<n; p++)  { c[k]=b[p]; k++; }
   }
   else  {
      for (p=i; p<m; p++)  { c[k]=a[p]; k++; }
   }
}
```

62



```
main()
{
    int i, num;
    int a[ ] = {-56,23,43,-5,-3,0,123,-35,87,56,75,80};

    num = 12;

    printf ("\n Original list: ");
    print (a, 0, num-1);

    merge_sort (a, 12);

    printf ("\n Sorted list: ");
    print (a, 0, num-1);
}
```

63

Time Complexity

- Best/Worst/Average case:

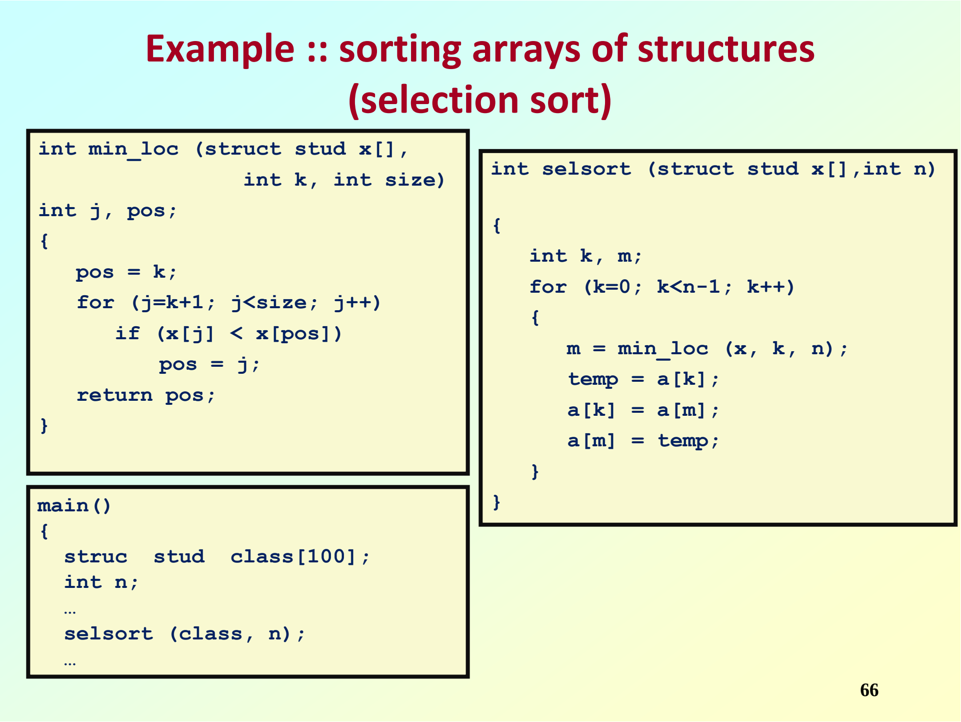
  n $\log_2 n$

- Drawback:
  - Needs double amount of space for storage.
  - For sorting *n* elements, requires another array of size *n* to carry out merge.

Example :: sorting arrays of structures (bubble sort)

```
#include <stdio.h>
struct stud
{
    int  roll;
    char  dept_code[25];
    float  cgpa;
};

main()
{
  struc  stud  class[100], t;
  int  j, k, n;

  scanf  (”%d”, &n);
        /* no. of students */
```

```
for (k=0; k<n; k++)
  scanf (”%d %s %f”, &class[k].roll,
               class[k].dept_code,
               &class[k].cgpa);
for (j=0; j<n-1; j++)
  for (k=1; k<n-j; k++)
  {
    if (class[k-1].roll >
                   class[k].roll)
    {
       t = class[k-1];
       class[k-1] = class[k];
       class[k] = t;
     }
   }
    <<<< PRINT THE RECORDS >>>>
}
```

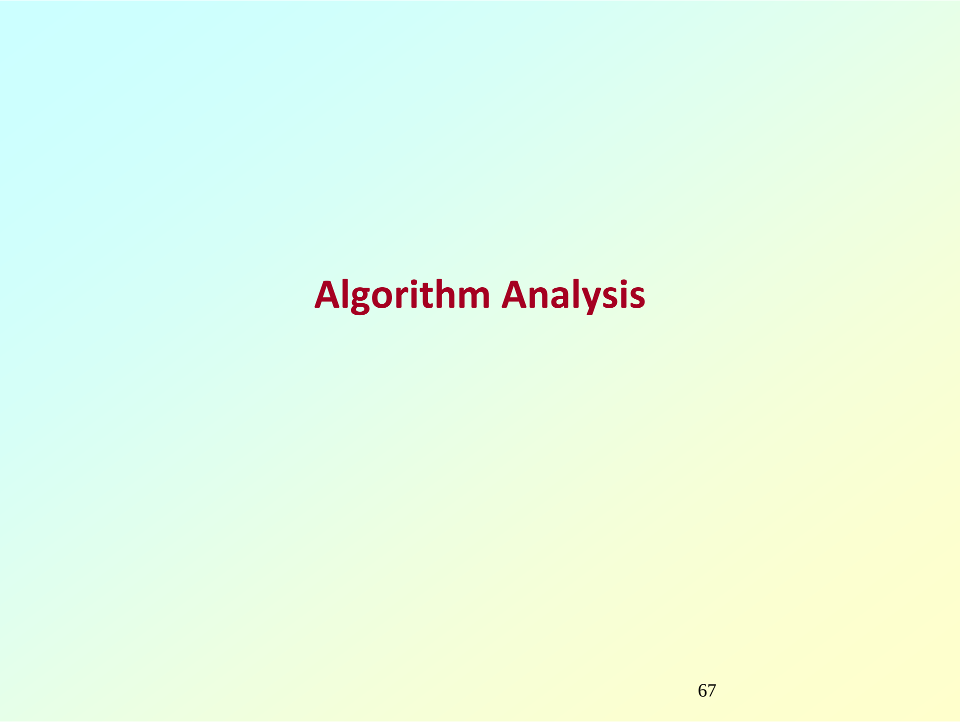


Example :: sorting arrays of structures (selection sort)

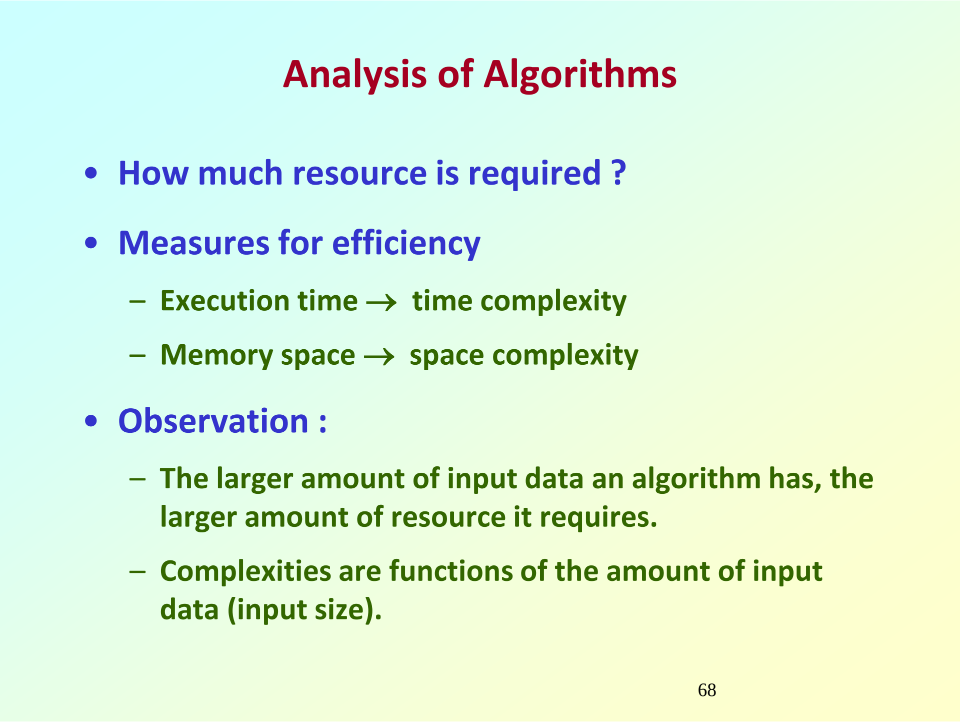
```
int min_loc (struct stud x[],
                int k, int size)
int j, pos;
{
   pos = k;
   for (j=k+1; j<size; j++)
      if (x[j] < x[pos])
          pos = j;
   return pos;
}
```

```
main()
{
  struc  stud  class[100];
  int n;
  …
  selsort (class, n);
  …
```

```
int selsort (struct stud x[],int n)

{
   int k, m;
   for (k=0; k<n-1; k++)
   {
      m = min_loc (x, k, n);
      temp = a[k];
      a[k] = a[m];
      a[m] = temp;
   }
}
```

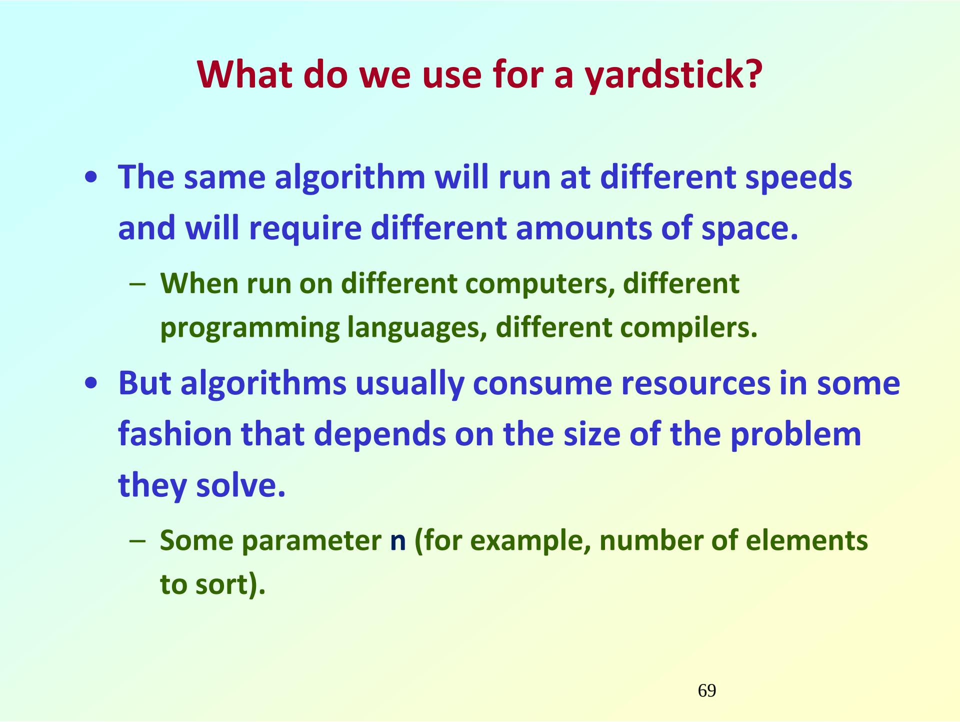


Algorithm Analysis



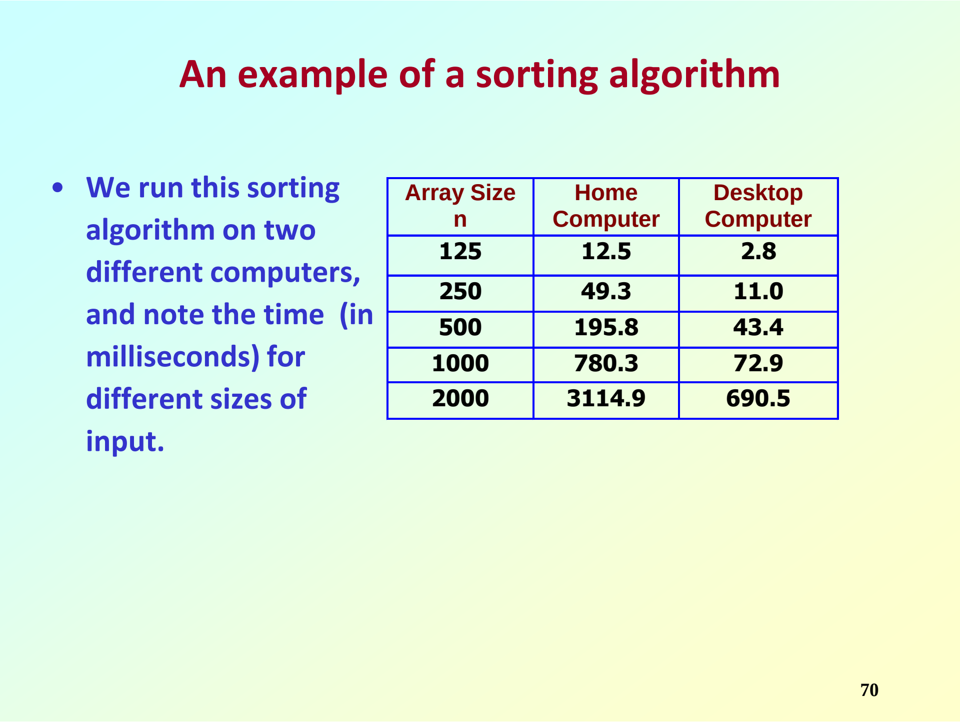
Analysis of Algorithms

- How much resource is required ?
- Measures for efficiency
  - Execution time → time complexity
  - Memory space → space complexity
- Observation :
  - The larger amount of input data an algorithm has, the larger amount of resource it requires.
  - Complexities are functions of the amount of input data (input size).



What do we use for a yardstick?

- The same algorithm will run at different speeds and will require different amounts of space.
  - When run on different computers, different programming languages, different compilers.
- But algorithms usually consume resources in some fashion that depends on the size of the problem they solve.
  - Some parameter **n** (for example, number of elements to sort).

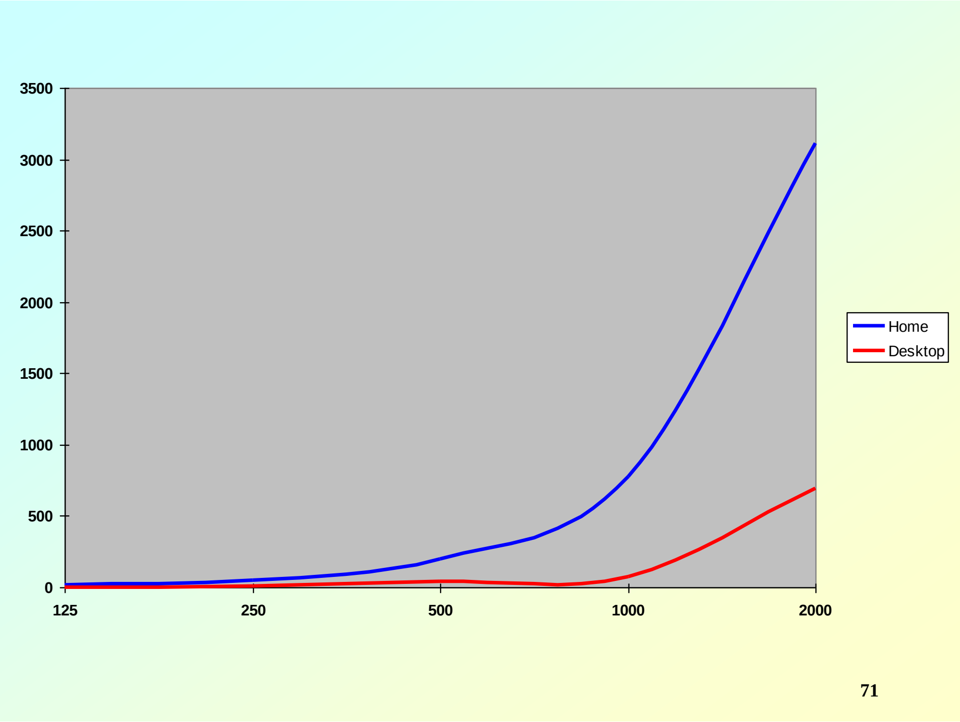


An example of a sorting algorithm

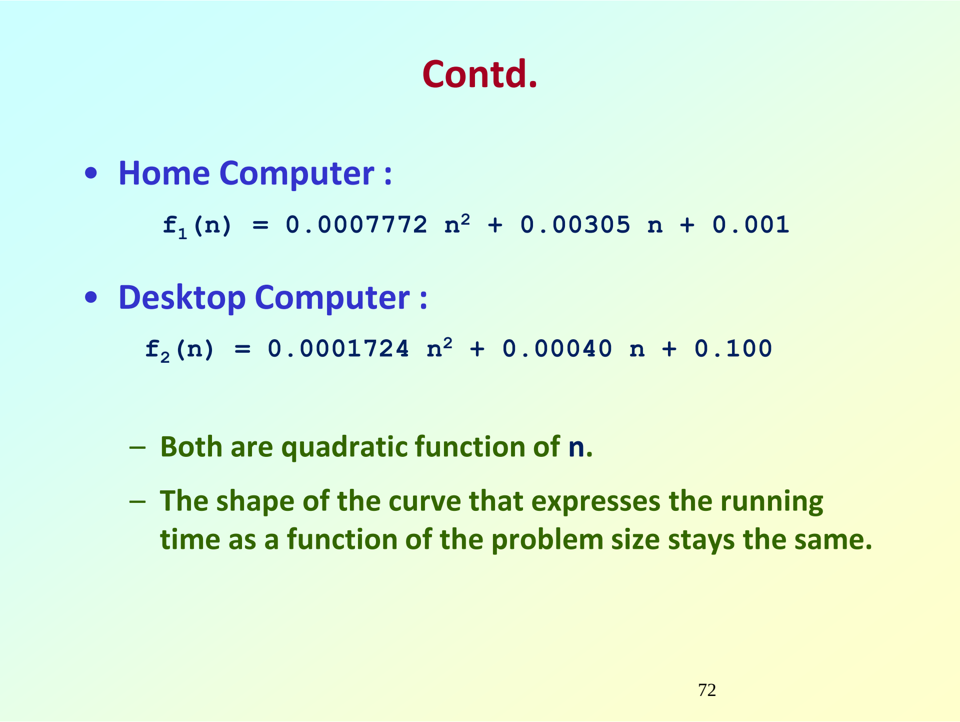
- We run this sorting algorithm on two different computers, and note the time (in milliseconds) for different sizes of input.

Array Size n	Home Computer	Desktop Computer
125	12.5	2.8
250	49.3	11.0
500	195.8	43.4
1000	780.3	72.9
2000	3114.9	690.5

70



71



Contd.

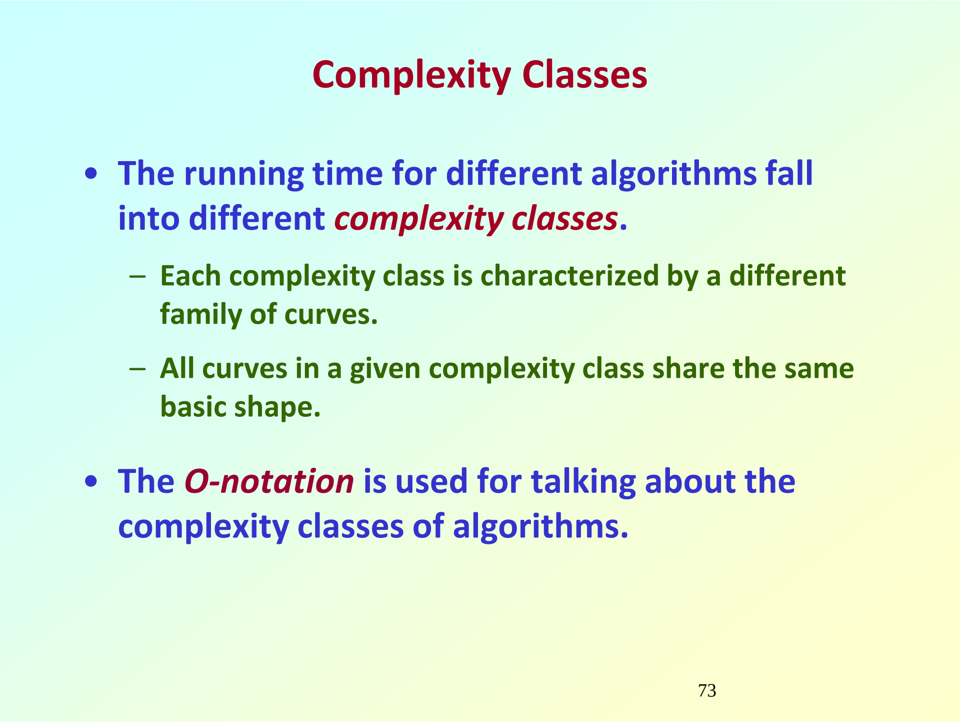
- **Home Computer :**

  $f_1(n) = 0.0007772\ n^2 + 0.00305\ n + 0.001$

- **Desktop Computer :**

  $f_2(n) = 0.0001724\ n^2 + 0.00040\ n + 0.100$

  - **Both are quadratic function of n.**
  - **The shape of the curve that expresses the running time as a function of the problem size stays the same.**



Complexity Classes

- **The running time for different algorithms fall into different *complexity classes*.**
  - **Each complexity class is characterized by a different family of curves.**
  - **All curves in a given complexity class share the same basic shape.**
- **The *O-notation* is used for talking about the complexity classes of algorithms.**

Running time of algorithms

Assume speed is 10^7 instructions per second.

size n	10	20	30	50	100	1000	10000
n	.001 ms	.002 ms	.003 ms	.005 ms	.01 ms	.1 ms	1 ms
nlogn	.003 ms	.008 ms	.015 ms	.03 ms	.07 ms	1 ms	13 ms
n^2	.01 ms	.04 ms	.09 ms	.25 ms	1 ms	100 ms	10 s
n^3	.1 ms	.8 ms	2.7 ms	12.5 ms	100 ms	100 s	28 h
2^n	.1 ms	.1 s	100 s	3 y	$3x10^{13}$ c	inf	inf

- The complexity classes:

$\log_2 n$

n

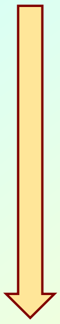
$n \log_2 n$

n^2

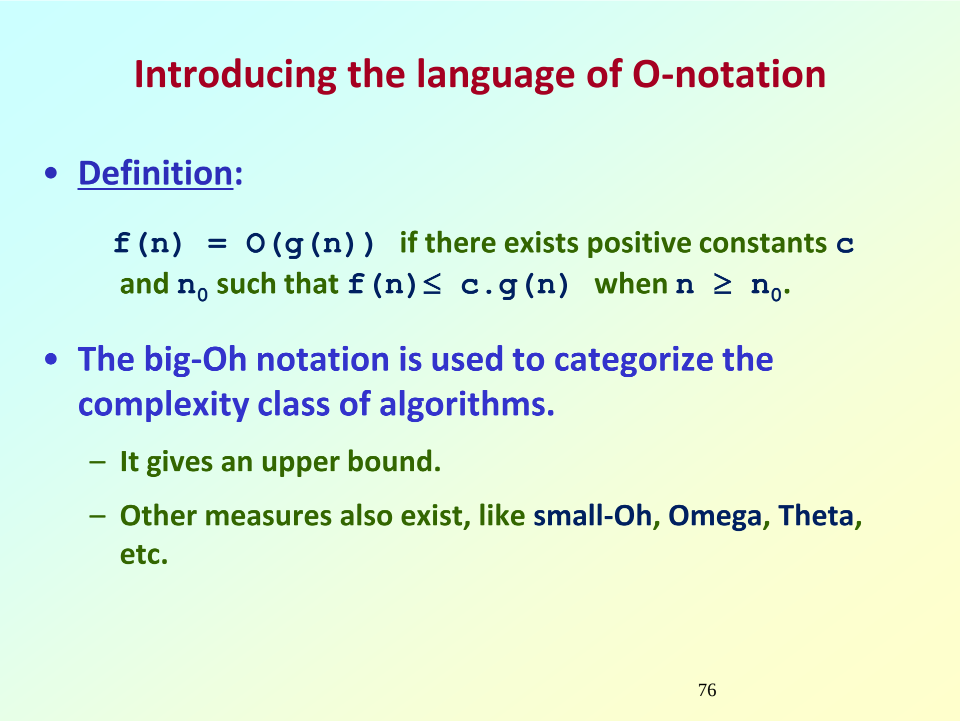
n^3

2^n

$n!$



Complexity increases

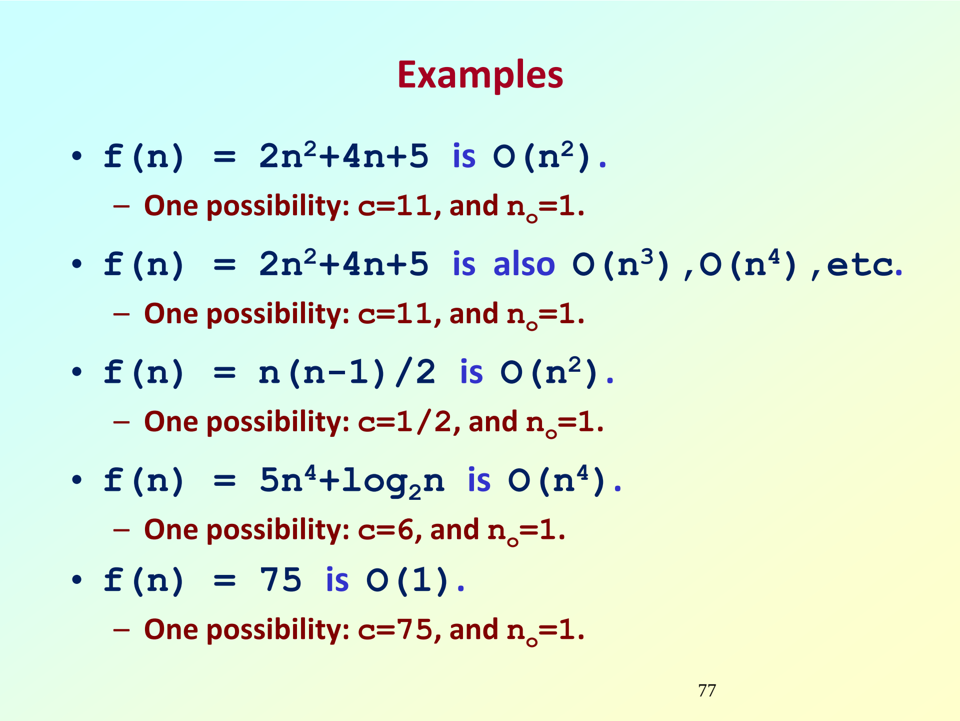


Introducing the language of O-notation

- **Definition:**

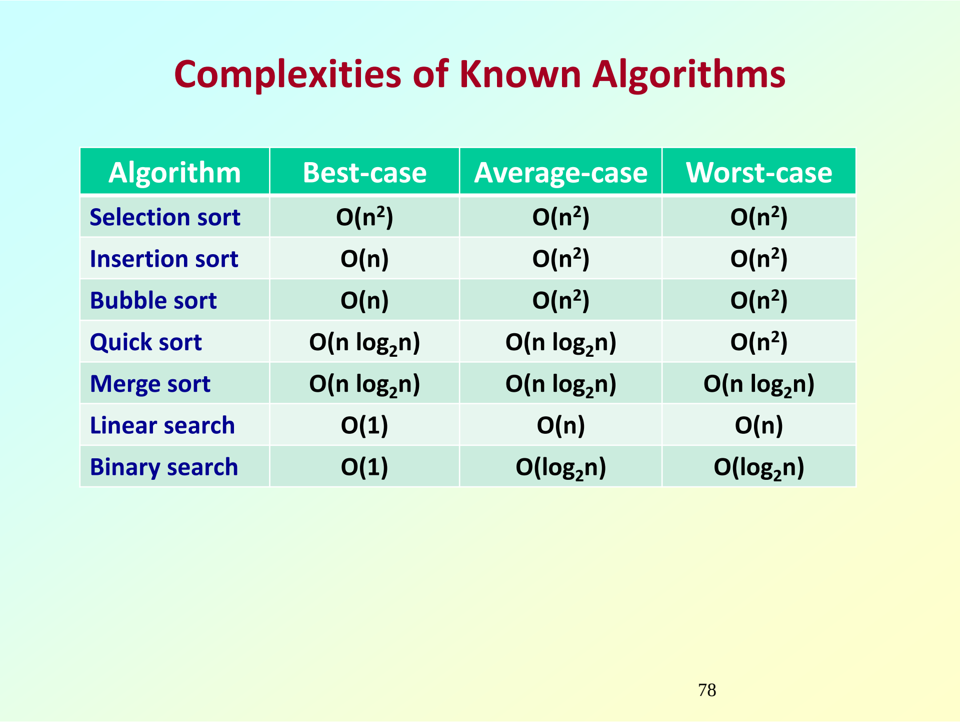
  $f(n) = O(g(n))$ if there exists positive constants c and n_0 such that $f(n) \leq c.g(n)$ when $n \geq n_0$.

- The big-Oh notation is used to categorize the complexity class of algorithms.
  - It gives an upper bound.
  - Other measures also exist, like small-Oh, Omega, Theta, etc.



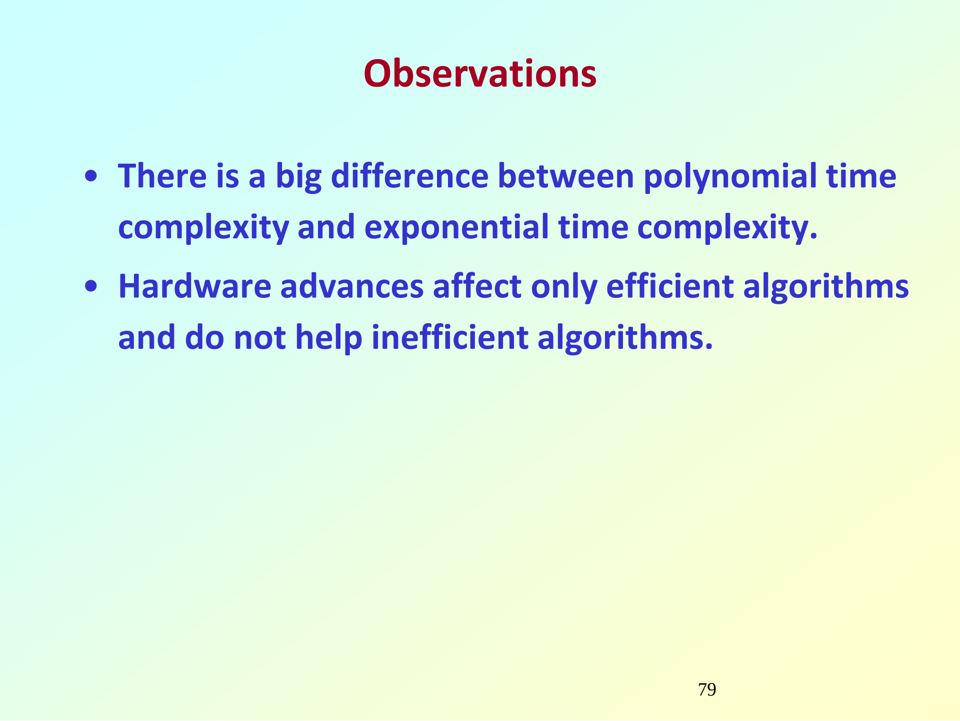
Examples

- $f(n) = 2n^2+4n+5$ is $O(n^2)$.
  - One possibility: $c=11$, and $n_o=1$.
- $f(n) = 2n^2+4n+5$ is also $O(n^3), O(n^4)$, etc.
  - One possibility: $c=11$, and $n_o=1$.
- $f(n) = n(n-1)/2$ is $O(n^2)$.
  - One possibility: $c=1/2$, and $n_o=1$.
- $f(n) = 5n^4+\log_2 n$ is $O(n^4)$.
  - One possibility: $c=6$, and $n_o=1$.
- $f(n) = 75$ is $O(1)$.
  - One possibility: $c=75$, and $n_o=1$.



Complexities of Known Algorithms

Algorithm	Best-case	Average-case	Worst-case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Linear search	$O(1)$	$O(n)$	$O(n)$
Binary search	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$



Observations

- There is a big difference between polynomial time complexity and exponential time complexity.
- Hardware advances affect only efficient algorithms and do not help inefficient algorithms.