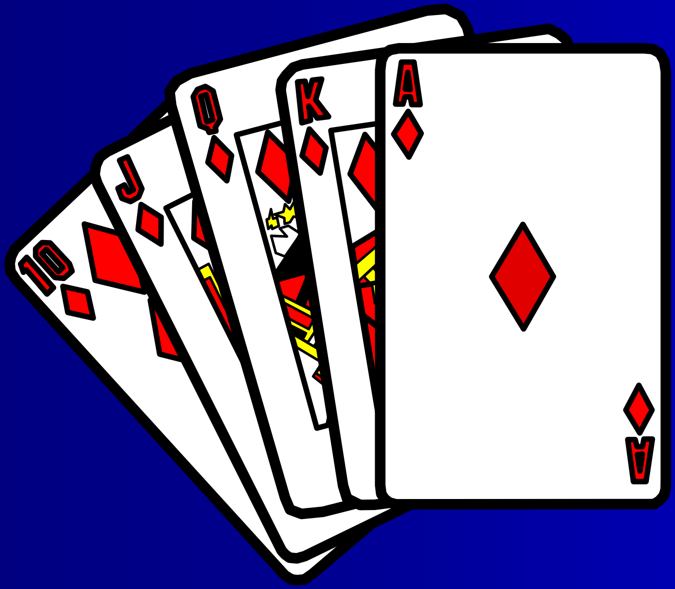
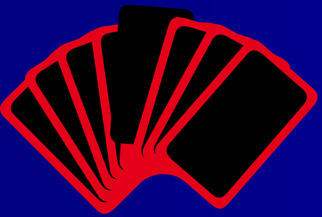


The Insertion Sort

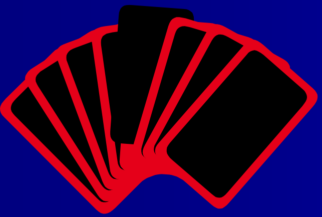
Mr. Dave Clausen

La Cañada High School



Objectives

- Understand and use the Insertion Sort to sort data in a program.
- Understand and know Big-O notation for the Insertion Sort.

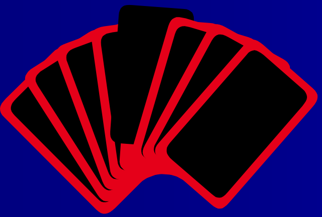


Insertion Sort Description

The *insertion sort* uses a vector's partial ordering. On the *k*th pass, the *k*th item should be inserted into its place among the first *k* items in the vector.

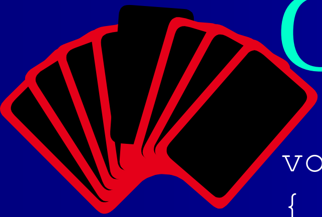
After the *k*th pass (*k* starting at 1), the first *k* items of the vector should be in sorted order.

This is like the way that people pick up playing cards and order them in their hands. When holding the first (*k* - 1) cards in order, a person will pick up the *k*th card and compare it with cards already held until its sorted spot is found.



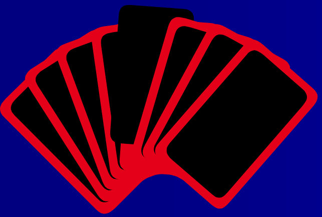
Insertion Sort Algorithm

```
For each k from 1 to n - 1 (k is the index of vector element to
   insert)
    Set item_to_insert to v[k]
    Set j to k - 1
    (j starts at k - 1 and is decremented until insertion position
   is found)
    While (insertion position not found) and (not beginning of
   vector)
        If item_to_insert < v[j]
            Move v[j] to index position j + 1
            Decrement j by 1
        Else
            The insertion position has been found
      item_to_insert should be positioned at index j + 1
```



C + + Code For Insertion Sort

```
void Insertion_Sort(vector<int> &v)
{
   int item_to_insert, j; // On the kth pass, insert item k into its correct
   bool still_looking;      // position among the first k entries in vector.
   for (int k = 1; k < v.size(); ++k)
   {   // Walk backwards through list, looking for slot to insert v[k]
      item_to_insert = v[k];
      j = k - 1;
      still_looking = true;
      while ((j >= 0) && still_looking )
         if (item_to_insert  < v[j])
         {
             v[j + 1] = v[j];
             --j;
          }
         else
            still_looking = false;   // Upon leaving loop, j + 1 is the index
         v[j + 1] = item_to_insert;   // where item_to_insert  belongs
     }
     }
```



Insertion Sort Example

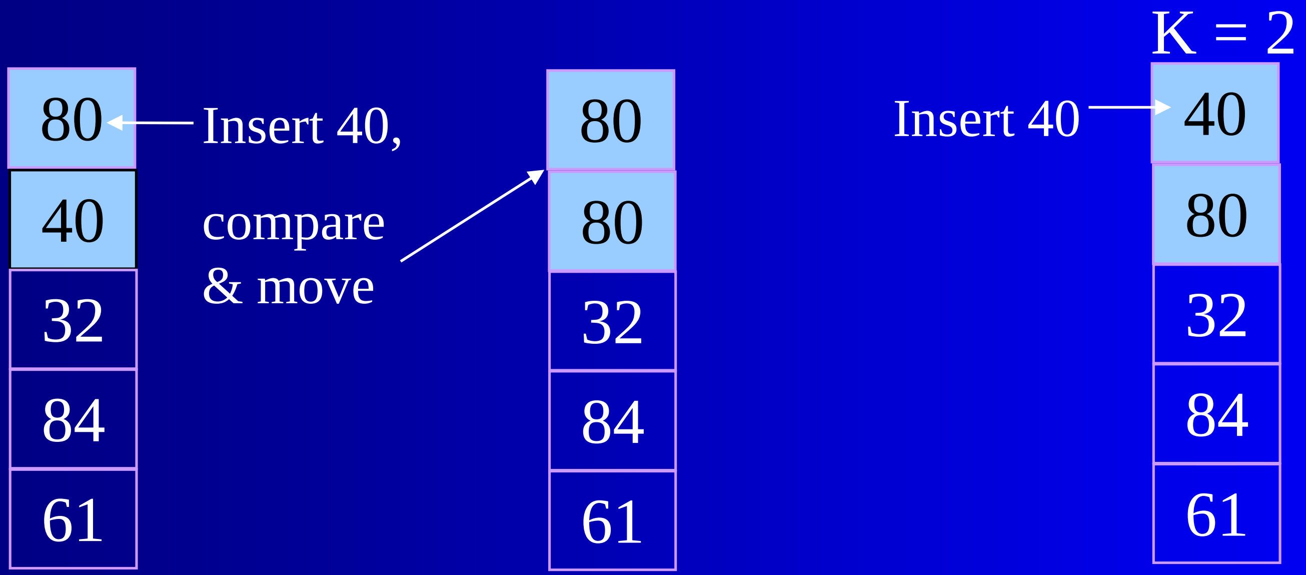
The Unsorted Vector:

80
40
32
84
61

For each pass, the index *j* begins at the (*k* - 1)st item and moves that item to position *j* + 1 until we find the insertion point for what was originally the *k*th item.

We start with k = 1
and set j = k-1 or 0 (zero)

The First Pass



80
40
32
84
61

Insert 40,

compare
& move

80
80
32
84
61

K = 2

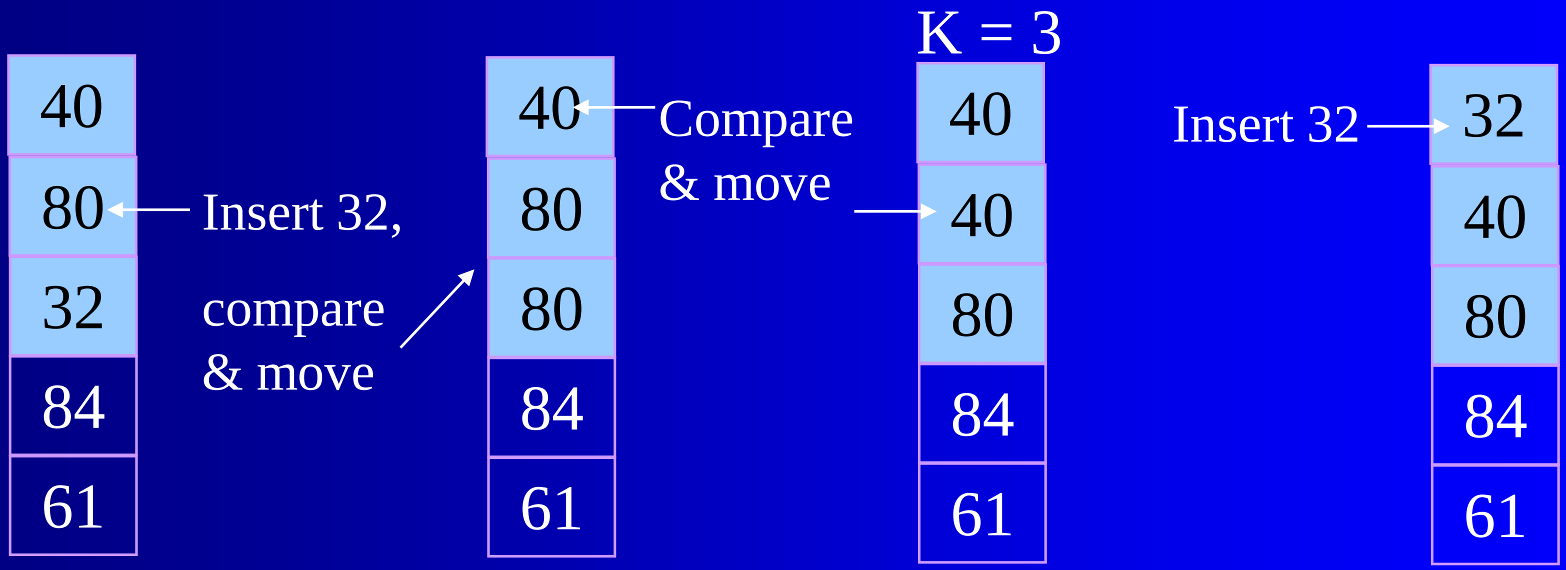
Insert 40

40
80
32
84
61

item_to_insert

40

The Second Pass



40
80
32
84
61

Insert 32,

compare
& move

40
80
80
84
61

Compare
& move

K = 3

40
40
80
84
61

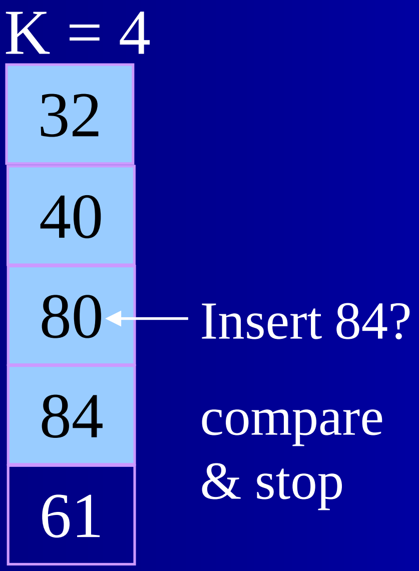
Insert 32

32
40
80
84
61

item_to_insert

32

The Third Pass



K = 4

32

40

80

84

61

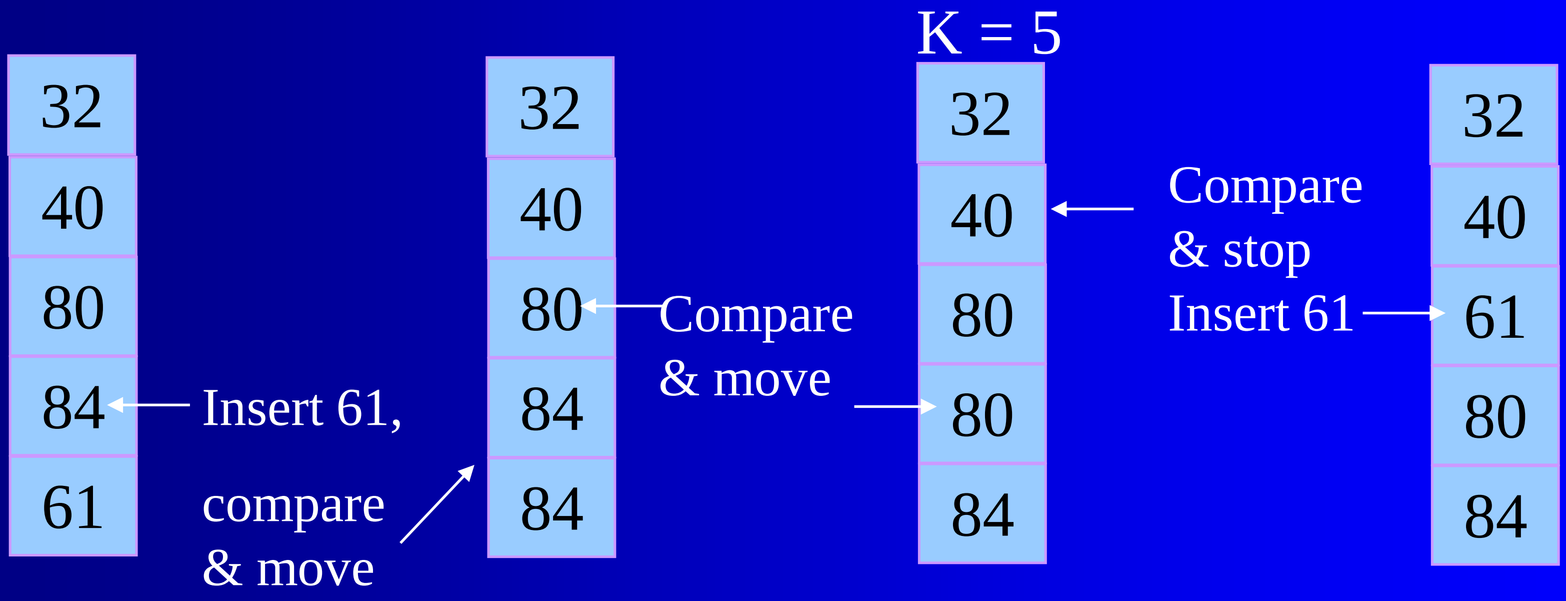
Insert 84?

compare
& stop

item_to_insert

84

The Fourth Pass



K = 5

32	32	32	32
40	40	40	40
80	80	80	61
84	84	80	80
61	84	84	84

Insert 61,

compare
& move

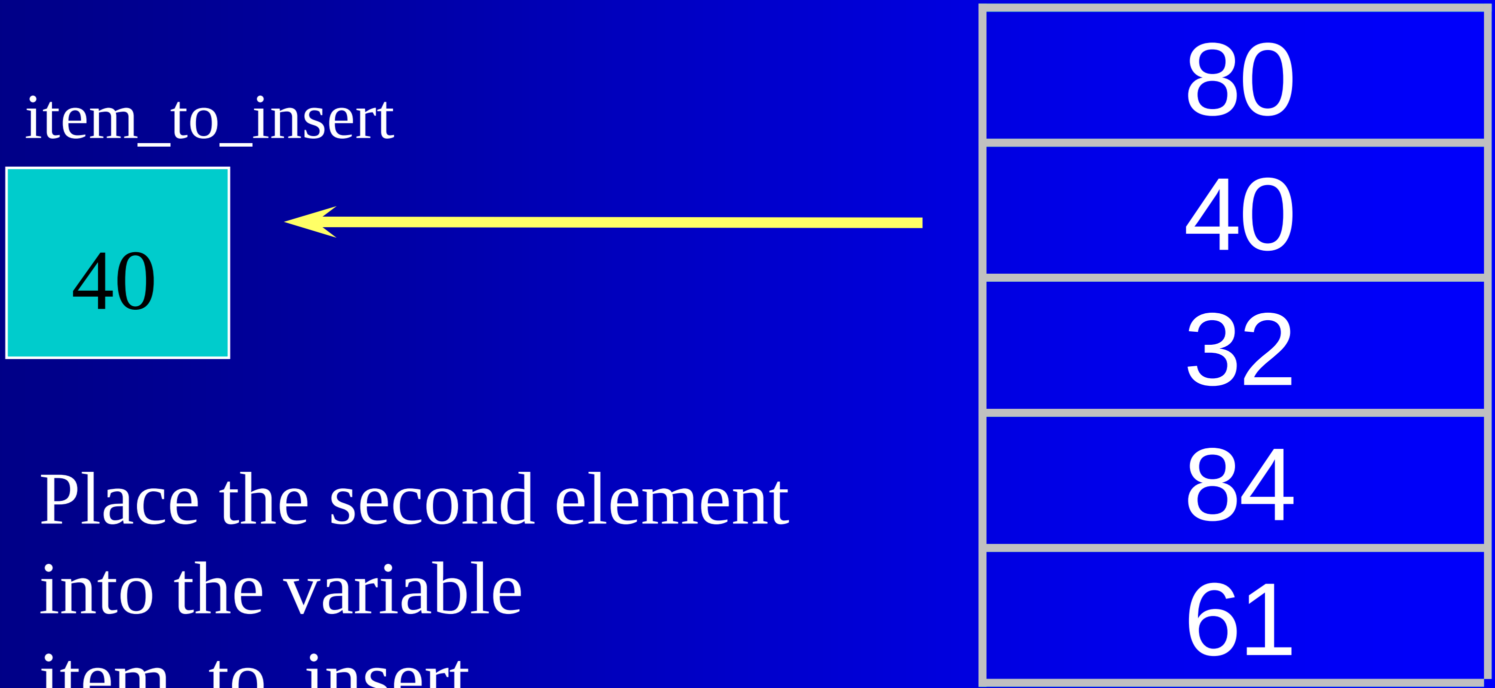
Compare
& move

Compare
& stop
Insert 61

item_to_insert

61

What "Moving" Means



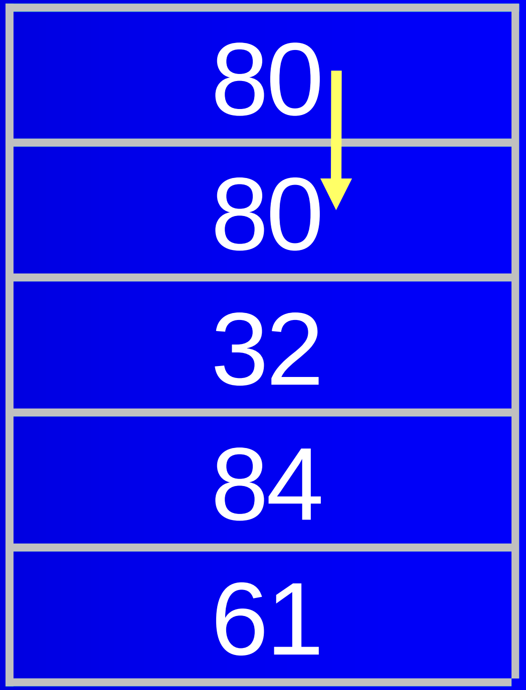
Place the second element into the variable item_to_insert.

What “Moving” Means

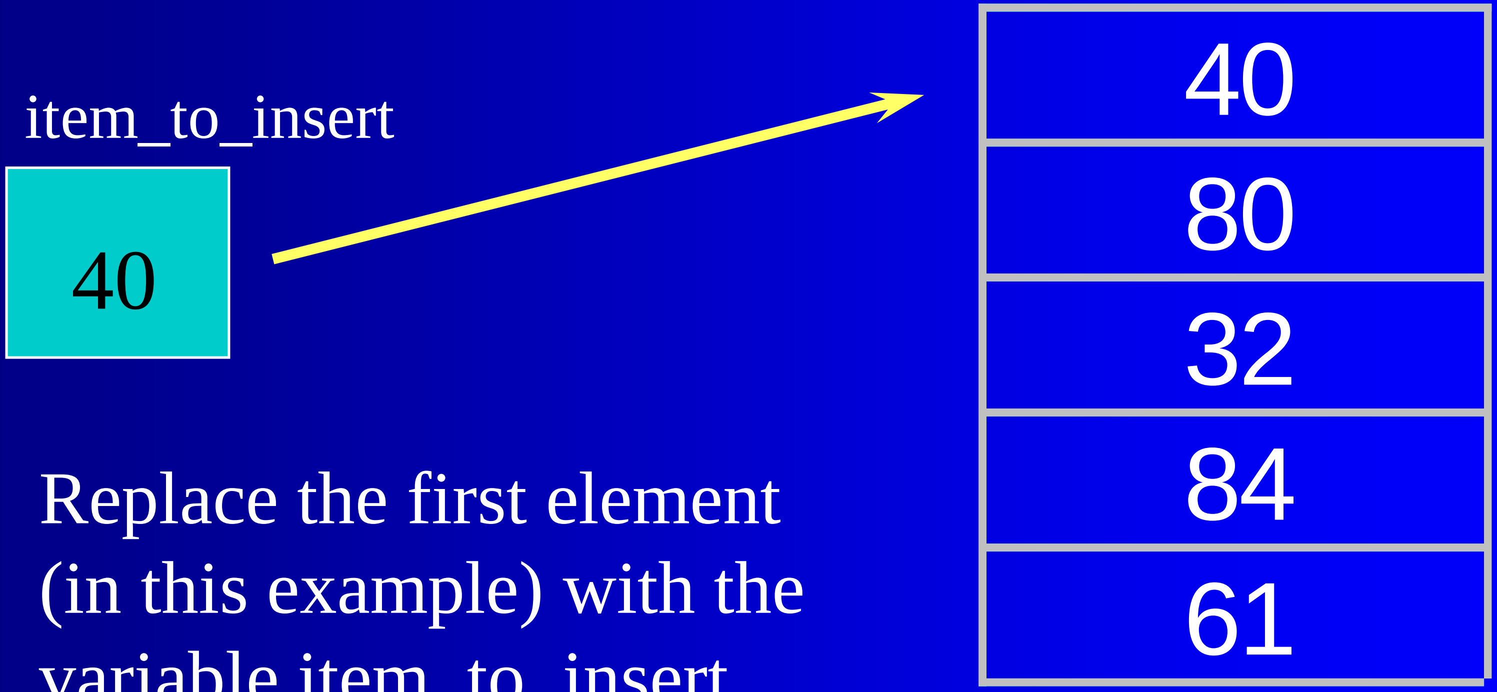
item_to_insert

40

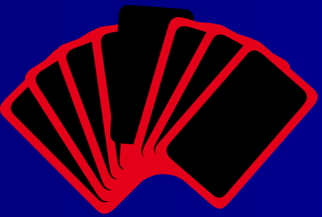
Replace the second element with the value of the first element.



What “Moving” Means



Replace the first element (in this example) with the variable item_to_insert.



C + + Examples of The Insertion Sort

On the Net:

http://compsci.exeter.edu/Winter99/CS320/Resources/sortDemo.html

http://www.aist.go.jp/ETL/~suzaki/AlgorithmAnimation/index.html

Big - O Notation

Big - O notation is used to describe the efficiency of a search or sort. The actual time necessary to complete the sort varies according to the speed of your system. Big - O notation is an approximate mathematical formula to determine how many operations are necessary to perform the search or sort. The Big - O notation for the Insertion Sort is $O(n^2)$, because it takes approximately n^2 passes to sort the “n” elements.