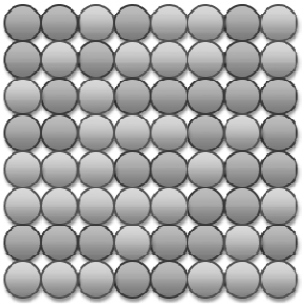
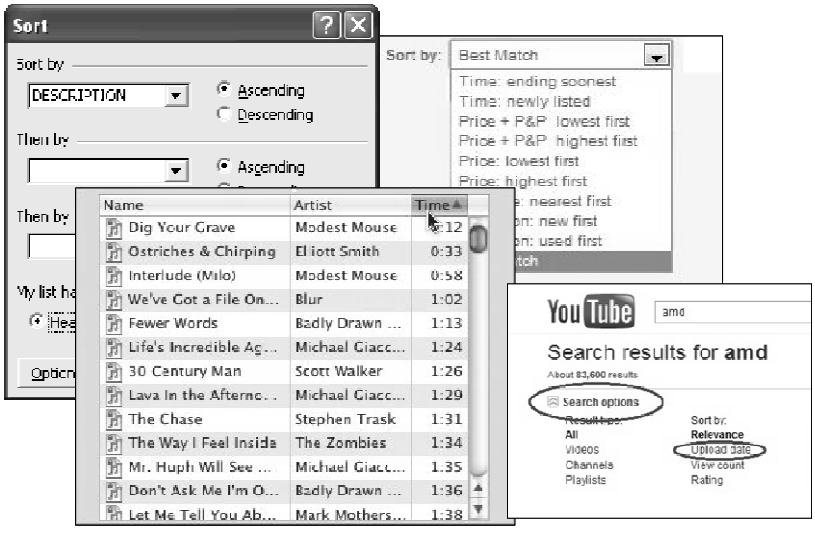


UNIT 4B
Iteration: Sorting

Sorting



Insertion Sort

Given an array a of length n, n > 0.

1. Set i = 1.
2. While i is not equal to n, do the following:
   a. Insert a[i] into its correct position in a[0..i].
   b. Add 1 to i.
3. Return the array a which will now be sorted.

Example

```
a = [53, 26, 76, 30, 14, 91, 68, 42]
i =       1
```

Insert a[1] into its correct position in a[0..1]
and then add 1 to i:

53 moves to the right,

26 is inserted back into the array

```
a = [26, 53, 76, 30, 14, 91, 68, 42]
i =           2
```

Example

```
a = [26, 53, 76, 30, 14, 91, 68, 42]
i =           2
```

Insert a[2] into its correct position in a[0..2]
and then add 1 to i:

76 is already in the correct place in a[0..2]

```
a = [26, 53, 76, 30, 14, 91, 68, 42]
i =               3
```


Example

```
a = [26, 53, 76, 30, 14, 91, 68, 42]
i =               3
```

Insert a[3] into its correct position in a[0..3]
and then add 1 to i:

76 moves to the right, then 53 moves to the right,

now 30 is inserted back into the array

```
a = [26, 30, 53, 76, 14, 91, 68, 42]
i =                   4
```

Look Closer at Insertion Sort

Given an array a of length n, n > 0.

1. Set i = 1.
2. While i is not equal to n, do the following:

   Precondition for each iteration: a[0..i-1] is sorted

   a. Insert a[i] into its correct position in a[0..i].

   b. Add 1 to i.

   Postcondition for each iteration: a[0..i-1] is sorted
3. Return the array a which will now be sorted.

Look Closer at Insertion Sort

Given an array a of length n, n > 0.

1. Set i = 1.
2. While i is not equal to n, do the following:

   Loop invariant: a[0..i-1] is sorted

   a. Insert a[i] into its correct position in a[0..i].

   b. Add 1 to i.
3. Return the array a which will now be sorted.

A <u>loop invariant</u> is a condition that is true at the start and end of each iteration of a loop.

Example (cont'd)

```
a = [26, 30, 53, 76, 14, 91, 68, 42]
i =                   4
```

Insert a[4] into its correct position in a[0..4]
and then add 1 to i:

76 moves to the right, then 53 moves to the right,

then 30 moves to the right, then 26 moves to the right,

now 14 is inserted back into the array

```
a = [14, 26, 30, 53, 76, 91, 68, 42]
i =                        5
```


Example

```
a = [14, 26, 30, 53, 76, 91, 68, 42]
i =                        5
```

Insert a[5] into its correct position in a[0..5]
and then add 1 to i:

91 is already in its correct position

```
a = [14, 26, 30, 53, 76, 91, 68, 42]
i =                             6
```

Example

```
a = [14, 26, 30, 53, 76, 91, 68, 42]
i =                              6
```

Insert a[6] into its correct position in a[0..6]
and then add 1 to i:

91 moves to the right,

76 moves to the right,

now 68 is inserted back into the array

```
a = [14, 26, 30, 53, 68, 76, 91, 42]
i =                                 7
```


Example

```
a = [14, 26, 30, 53, 68, 76, 91, 42]
i =                                 7
```

Insert a[7] into its correct position in a[0..7]
and then add 1 to i:

91 moves to the right, then 76 moves to the right,

then 68 moves to the right, then 53 moves to the right,

then 42 is inserted back into the array

```
a = [14, 26, 30, 42, 53, 68, 76, 91]
i =                                    8
```

Example

```
a = [14, 26, 30, 42, 53, 68, 76, 91]
i = 8
```

The array is sorted.

But how do we know that the algorithm always sorts correctly?

Reasoning with the Loop Invariant

The loop invariant is true at the end of each iteration, including the last iteration. After the last iteration, when we go to step 3:

a[0..i-1] is sorted AND i is equal to n

These 2 conditions imply that a[0..n-1] is sorted, but this range covers the entire array, so the array must always be sorted when we return our final answer!

Insertion Sort in Ruby

```
def isort(list)
     a = list.clone
     i = 1
     while i != a.length do
          move_left(a, i)
          i = i + 1
     end
     return a
end
```

insert a[i] into a[0..i] in its correct sorted position

Moving left

To move the element x at index i "left" to its correct position, start at position i-1, and search left until we find the first element that is less than x.

Then insert x back into the array to the right of the first element that is less than x when you searched from right to left in the sorted part of the array.

(The insert operation does not overwrite. Think of it as "squeezing into the array".)

Can you think of a special case for the step above?

Moving left: examples

Insert 68:

a = [14, 26, 30, 53, 76, 91, 68, 42]

Searching from right to left starting with 91, the first element less than 68 is 53.

Insert 68 to the right of 53.

Insert 76:

a = [26, 53, 76, 30, 14, 91, 68, 42]

Searching from right to left starting with 53, the first element less than 76 is 53.

Insert 76 to the right of 53 (where it was before).

Insert 14: SPECIAL CASE

a = [26, 30, 53, 76, 14, 91, 68, 42]

Searching from right to left starting with 76, all elements left of 14 are greater than 14. Insert 14 into the position 0.

The **`move_left`** algorithm

Given an array a of length n, n > 0 and a value at index i to be "moved left" in the array.

1. Remove a[i] from the array and store in x.
2. Set j = i-1.
3. While j >= 0 and a[j] > x, do the following:
   a. Subtract 1 from j.
4. Reinsert x into position a[j+1].

How is the special case handled here?

`move_left` in Ruby

```
def move_left(a, i)
    x = a.slice!(i)
    j = i-1
    while j >= 0 and a[j] > x do
        j = j - 1
    end
    a.insert(j+1, x)
end
```

remove the item at position i in array a and store it in x

logical operator AND: both conditions must be true for the loop to continue

insert x at position j+1 of array a, shifting all elements from j+1 and beyond over one position