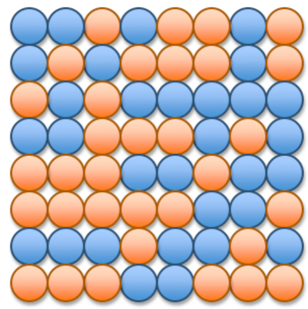
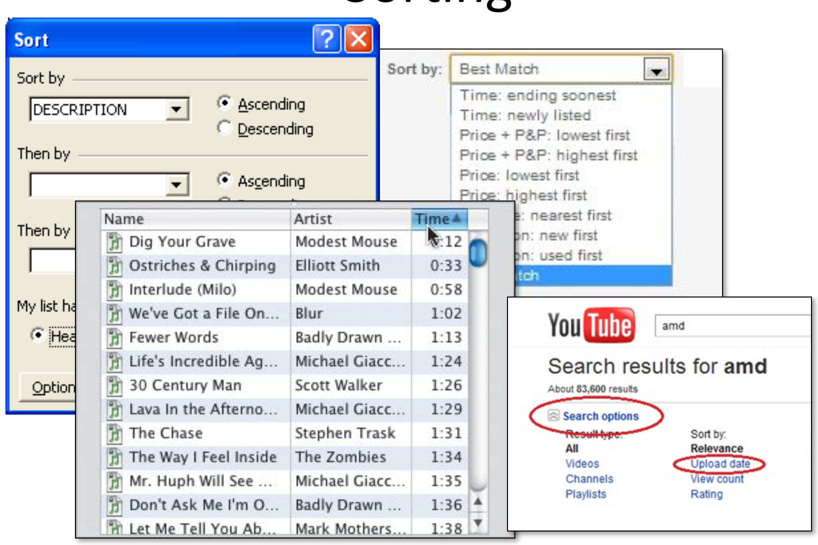


UNIT 4B
Iteration: Sorting

Sorting



Insertion Sort Outline

```
def isort(datalist):
    result = []
    for value in datalist:
        # insert value in its
        # proper place in result
    return result
```


insert function

```
datalist.insert(position, value)

>>> a = [10, 30, 20]
>>> a
[10, 30, 20]
>>> a.insert(0, 15)
>>> a
[15, 10, 30, 20]
```

insert function (cont'd)

```
>>> a.insert(2, 50)
>>> a
[15, 10, 50, 30, 20]
>>> a.insert(5, 40)
>>> a
[15, 10, 50, 30, 20, 40]
```

The insert function just inserts at the given index. It doesn't assume the list is sorted.

Insertion Sort, Refined

```
def isort(datalist):
    result = []
    for value in datalist:
        # compute place to insert
        result.insert(place, value)
    return result
```

How do we find the right place to insert?

gr_index

Compute the index of first element greater than item

```
def gr_index(datalist, item):
# precondition: datalist is sorted!
    index = 0
    while index < len(datalist) and datalist[index] < item:
        index = index + 1
    return index
```


Testing gr_index

```
>>> a = [10, 20, 30, 40, 50]
>>> a
[10, 20, 30, 40, 50]
>>> gr_index(a, 3)
0
>>> gr_index(a, 14)
1
>>> gr_index(a, 37)
3
>>> gr_index(a, 99)
5
```

Insertion Sort, Complete

```
def isort(datalist):
  result = [ ]
  for value in datalist:
    place = gr_index(result, value)
    result.insert(place, value)
  return result
```


Debugging Insertion Sort

```
def isort(list):
  result = [ ]
  print(result)     # for debugging
  for val in list:
    place = gindex(result, val)
    result.insert(place, val)
    print(result)   # for debugging
  return result
```

Testing isort

```
  >>> isort([3, 1, 4, 1, 5, 9, 6, 2])
  []
  [3]
  [1, 3]
  [1, 3, 4]
  [1, 1, 3, 4]
  [1, 1, 3, 4, 5]
  [1, 1, 3, 4, 5, 9]
  [1, 1, 3, 4, 5, 6, 9]
  [1, 1, 2, 3, 4, 5, 6, 9]
=> [1, 1, 2, 3, 4, 5, 6, 9]
```


Can We Do Better?

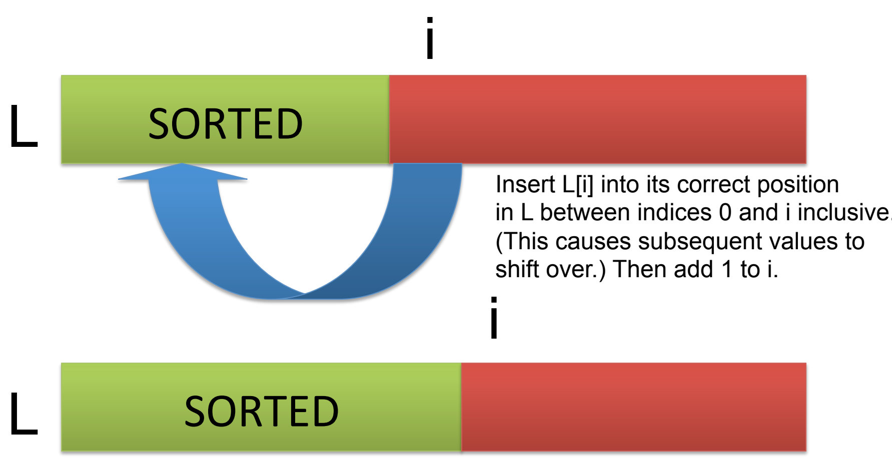
- isort doesn't change its input list.
- Instead it makes a new list, called result.
- This takes twice as much memory.
- Can we write a destructive ("in place") version of the algorithm that doesn't use extra memory?

Destructive (In Place) Insertion Sort

Given a list L of length n, n > 0.

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

General Idea: Any one Iteration



Same basic picture!

Insertion Sort in Python
("in place")

```
def isort(datalist):
    i = 1
    while i < len(datalist):
        x = datalist.pop(i)
        index = 0
        while index < i and datalist[index] < x:
            index = index + 1
        datalist.insert(index, x)
        i = i + 1
    return datalist
```

pop: remove the item at position i in list and store it in x

Look familiar? This is essentially gr_index, except it stops at index i rather than scanning the whole list!

Look Closer at Insertion Sort

L[0..i) means:
List L from index 0
up to but not including i

Given a list L of length n, n > 0.

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

   Precondition for each iteration: L[0..i) is sorted

   a. Insert L[i] into its correct position in L between index 0 and index i inclusive.

   b. Add 1 to i.

   Postcondition for each iteration: L[0..i) is sorted
3. Return the list L which will now be sorted.

Look Closer at Insertion Sort

Given a list L of length n, n > 0.

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

   Loop invariant: L[0..i) is sorted

   a. Insert L[i] into its correct position in L between index 0 and index i inclusive.

   b. Add 1 to i.
3. Return the list L which will now be sorted.

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

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:

L[0..i) is sorted (from the last iteration)

AND

i is equal to n (due to the while loop terminating)

These 2 conditions imply that L[0..n) is sorted, but this range is the entire list, so the list must always be sorted when we return our final answer!