(03) Iteration & Decomposition Part 2a (Sorting)

Video (13 mins):

https://youtu.be/AOqN6GNHQSU?list=PLi1cUmnkDnZvpLI1NPYxmq1Jnd7LAGCaa&t=18

Sorting

- The search algorithms shown on the previous slides are examples of ***linear*** algorithms
  - start at the beginning of a collection
  - systematically progress through the collection, all the way to the end if necessary
- A similar strategy can be used to sort the items in an array
- The next set of slides will introduce a very simple sorting algorithm known as ***insertion sort***

Insertion Sort

- The important property of the insertion sort algorithm: at any point in this algorithm ***left part of the array is already sorted***
- The item we currently want to find a place for will be called the ***key***
  - items to the left of the key are already sorted
  - the goal on each iteration is to insert the key at its proper place in the sorted part

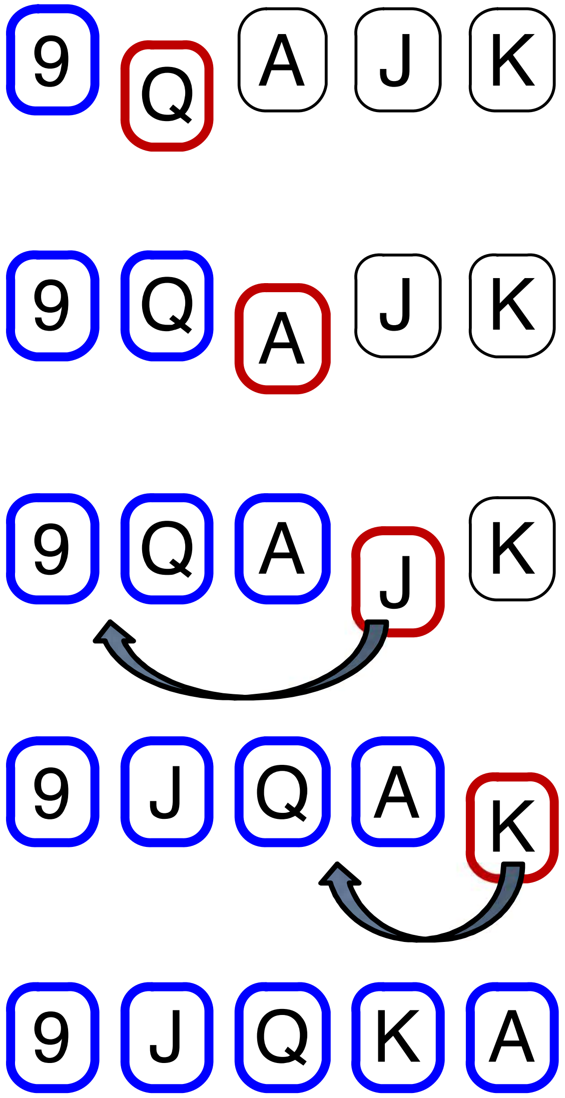
- Basic idea:
  - Pick up an item, find the place it belongs, insert it back into the array.
  - Move to the next item and repeat

Insertion Sort – Formal Definition

- Insertion sort algorithm
  1. the initial key is the second item in the array (the Q in this example)
  2. use your left hand to pick up the key
  3. Try to move the key to as left as possible, until you find an item lower than the one in your left hand, or the front of the array, whichever comes first
  4. the new key is the item to the right of the location of the previous key
  5. go back to step 2

- This new version is precise enough that we can write a Python function that implements this algorithm

Insertion Sort – Example



Insertion Sort in Python

- The algorithms have all been fully implemented in a PythonLabs module

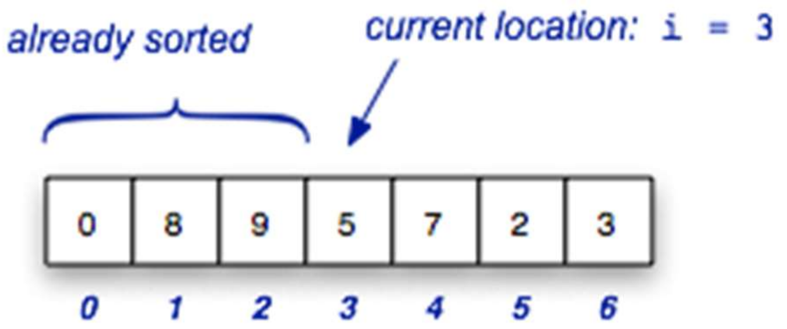
```
def iSort(array):
    i = 1
    while i < len(array):
        move_left(array, i)
        i += 1
    return array
```

see later slides

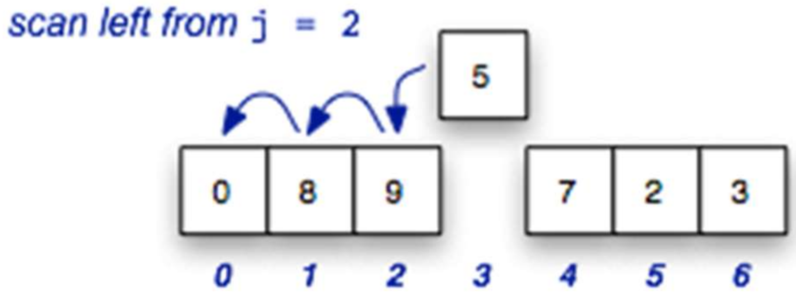
Insertion Sort Example

- The following pictures show an example of how insertion sort works (using a list of numbers instead of cards)

  - when `i` is 3 the positions to the left (0 through 2) have been sorted

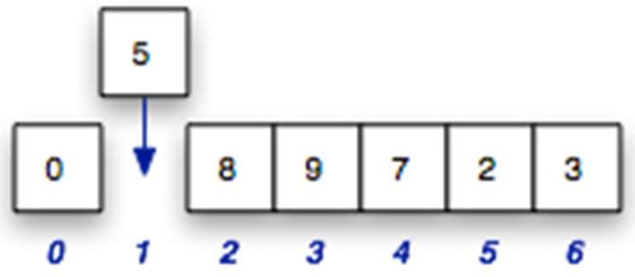


  - the first statements in the body of the loop set `key` to 5 and remove it from `a`

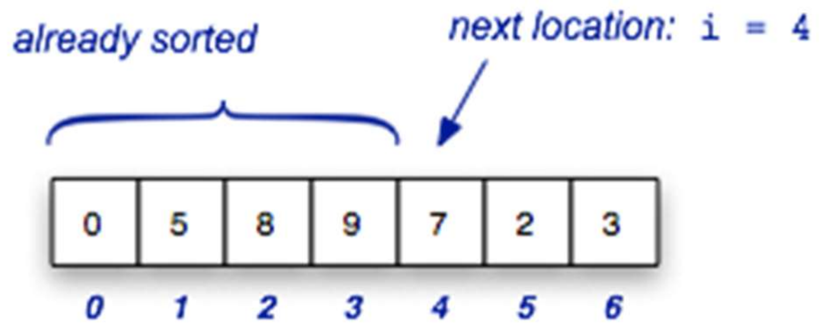


Insertion Sort Example (cont'd)

- ❖ the algorithm looks to the left for a location to put the 5
- ❖ `j` is set to 0 since `a[0]` is the first item smaller than 5



- ❖ on the next iteration, `i` will be 4, and the sorted region has grown by 1 to include all items in locations 0 to 3



Visualization:

- http://www.sorting-algorithms.com/
- https://visualgo.net/en/sorting

Helper Function

- move_left: The operations of moving item at a[i], to as left as possible.
  - programmers call these special-purpose functions “helper functions”
  - not intended to be used on its own, but only during a sort

```
def move_left(array,i):
    while i > 0 and array[i] < array[i-1]:
        # swap elements at i-1 and i
        array[i],array[i-1] = array[i-1], array[i]
        i -= 1 # move left
```

*The **Big Picture**:*

- *a call to* move_left(array,i) *moves* a[i] *somewhere to the left*
- *this function uses iteration – in fact it’s a type of linear search*

Nested Loops

- At first glance it might seem that insertion sort is a "linear" algorithm like `search` and `max`
    - it has a `while` loop that progresses through the array from left to right
- But it's important to note what is happening in `move_left`
    - the step that finds the proper location for the current item is also a loop
    - it scans left from location `i`, going all the way back to 0 if necessary

```
def isort(a)
  i = 1
  while i < len(a)
    ...
    while i > 0 && less(a[i], a[i-1])
      ...
    end
    ...
  end
  return a
end
```

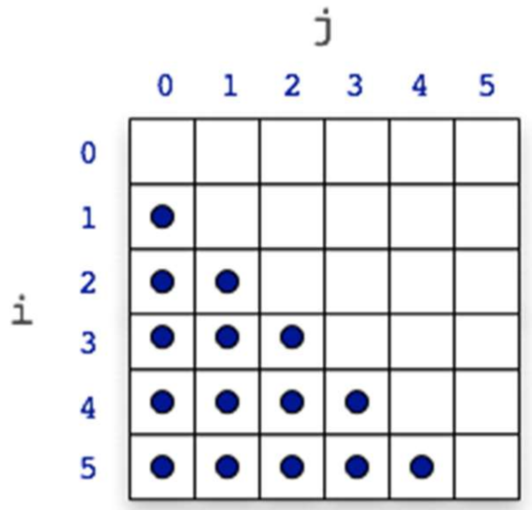
The `while` *loop in* `move_left`

- An algorithm with one loop inside another is said to have ***nested loops***

In the Worst Case ...

- The diagram at right helps visualize how many comparisons are in `isort`
  - a dot in a square indicates a potential comparison
  - for any value of `i`, the inner loop might have to compare key to values from `i` - 1 all the way down to 0
- The number of dots in this diagram is: (6 x 5) / 2 = 15
- In general, for an array with n items, the potential number of comparisons is:

$$\frac{n \times (n-1)}{2} \approx \frac{n^2}{2}$$



"Snap off" the empty top row, leaving `n*(n-1)` cells, half of which contain dots...

When does this worst case happen?

Animated Demo: `iSort`

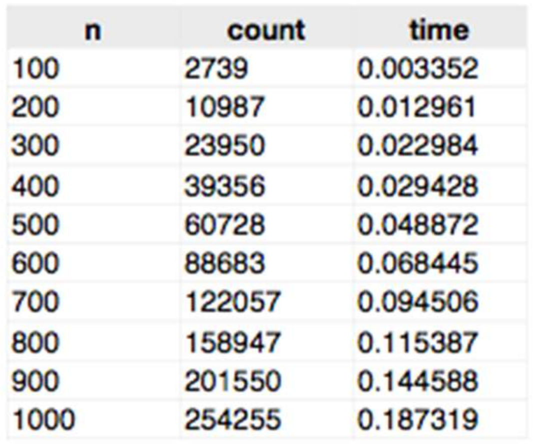
- An example of how to call `iSort` animation

```
>> from SearchSortAnimation import *

>> arr = [5, 9, 8, 2, 11, 3, 6, 1]

>> iSort(arr, True)
```

Scalability

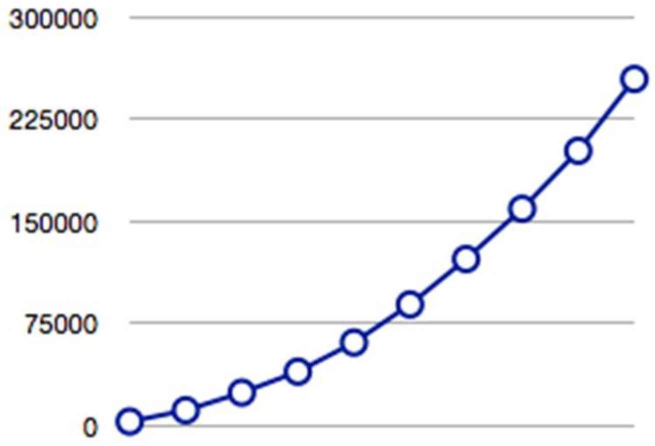
- The fact that the number of comparisons grows as the square of the array size may not seem important
  - for small to moderate size arrays it is not a big deal
  - but execution time will start to be a factor for larger arrays

(a)



n	count	time
100	2739	0.003352
200	10987	0.012961
300	23950	0.022984
400	39356	0.029428
500	60728	0.048872
600	88683	0.068445
700	122057	0.094506
800	158947	0.115387
900	201550	0.144588
1000	254255	0.187319

(b)



In-Class Exercises

How many comparison operations will <u>insertion sort</u> make for each input array below?

(a) When `x = [93, 85, 22, 69, 73, 59]`

(b) When `x = [24, 56, 30, 28, 47, 91, 60, 36]`

Divide and Conquer Sorting Algorithms

- The divide and conquer strategy used to make a more efficient search algorithm can also be applied to sorting
- Two well-known sorting algorithms:

QuickSort

- divide a list into big values and small values, then sort each part

Merge Sort

- sort subgroups of size 2, merge them into sorted groups of size 4, merge those into sorted groups of size 8, ...

- We first study the *iterative* version of Merge Sort
- In the next part of the class, we will study the *recursive* version of Merge Sort