Introduction to Algorithms & Analysis

Instructor: Krishna Venkatasubramanian

CSC 212

Announcements

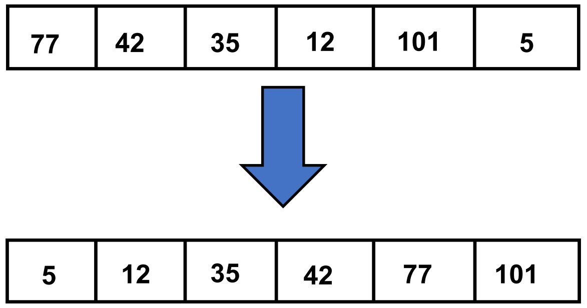
- Reminders:
  - Go to office hours
  - Self-advocacy --- you have to bring issues are you having with the course to us
- Go to Python tutorial on Fridays @2pm (Library 130)
- Next Quiz September 24
  - Quizzes now move to **Tuesdays** for the rest of the semester

Algorithms

- **Definition**:
  - Any well-developed computational procedure that takes some value or set of values as **input** and produces some value, or set of values, as **output**
- A **tool** to solve computational problems
  - Given a desired input and output relationship, an algorithm specifies a step-by-step procedure to make that happen!
- Example --- Sorting!!!
  - (One of the most common tasks a computer performs)

Sorting

- **Input**
  - A list of unsorted numbers A = <a_1, a_2, ….a_n>
- **Output**
  - A permutation (re-ordering) of A like <a'_1,a'_2,….a'_n>, where $a'_1 <= a'_2 <= …. <= a'_n$



Generally speaking, can be used for any sorting any set of values. The algorithm must know how to compare values (<, =, or >)

Insertion Sort

Same idea as **sorting cards as they are dealt.**

Example

Dealing order: 6, 8, 4, 1, 3

(1)	6				
(2)	6	8			
(3)	4	6	8		
(4)	1	4	6	8	
(5)	1	3	4	6	8

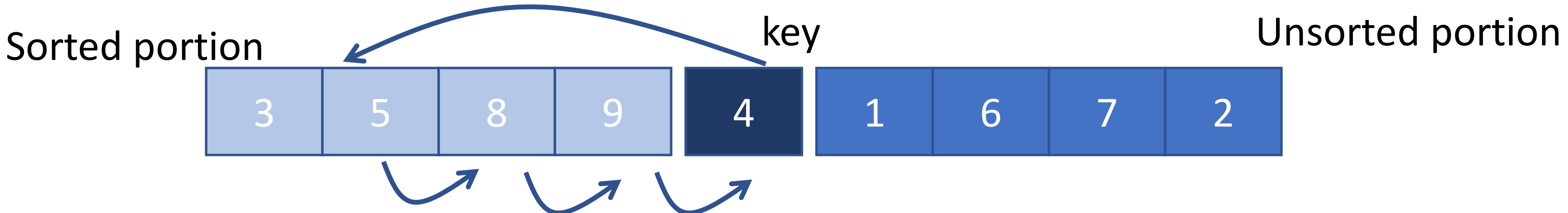
Card state as each new card is dealt

Python code

REMEMBER:
Python uses 0 index arrays

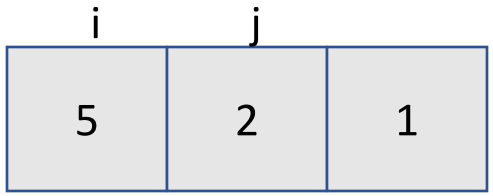
```
def InsertionSort(A)
 for j in range(1,len(A))
    key = A[j]
    i = j-1
    while i >= 0  and   A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

Ignore the gaps in the array
(they are shown for convenience)
It's one long array



Example

key = 2

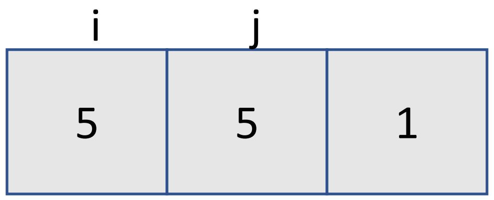


j = 1	A[j] = 2
i = 0	A[i] = 5

```
def InsertionSort(A)
  for j in range(1,len(A))
        key = A[j]
        i = j-1
        while i >= 0  and   A[i] > key
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

Example

key = 2

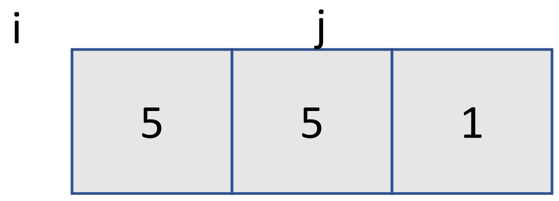


j = 1	A[j] = 5
i = 0	A[i] = 5

```
def InsertionSort(A)
 for j in range(1,len(A))
     key = A[j]
     i = j-1
     while i >= 0  and   A[i] > key
         A[i+1] = A[i]
         i = i - 1
     A[i+1] = key
```

Example

key = 2

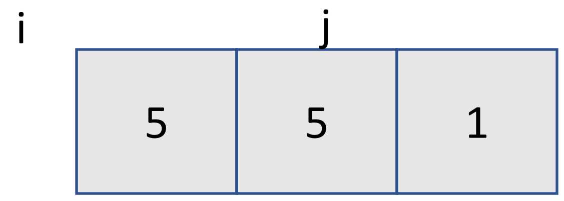


j = 1	A[j] = 5
i = -1	A[i] = N/A

```
def InsertionSort(A)
 for j in range(1,len(A))
    key = A[j]
    i = j-1
    while i >= 0  and   A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

Example

key = 2

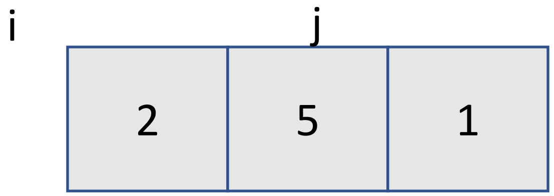


j = 1 A[j] = 5
i = -1 A[i] = N/A

```
def InsertionSort(A)
 for j in range(1,len(A))
     key = A[j]
     i = j-1
     while i >= 0  and   A[i] > key
         A[i+1] = A[i]
         i = i - 1
     A[i+1] = key
```

Example

key = 2

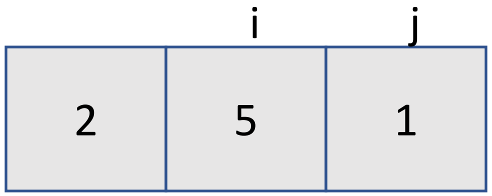


j = 1 A[j] = 5
i = -1 A[i] = N/A

```
def InsertionSort(A)
 for j in range(1,len(A))
     key = A[j]
     i = j-1
     while i >= 0  and   A[i] > key
         A[i+1] = A[i]
         i = i - 1
     A[i+1] = key
```

Example

key = 1

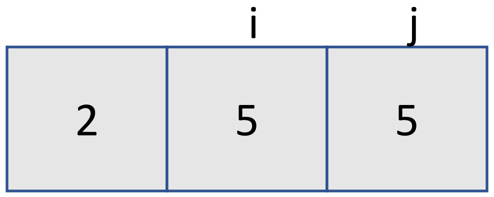


j = 2	A[j] = 1
i = 1	A[i] = 5

```
def InsertionSort(A)
  for j in range(1,len(A))
      key = A[j]
      i = j-1
      while i >= 0  and   A[i] > key
          A[i+1] = A[i]
          i = i - 1
      A[i+1] = key
```

Example

key = 1

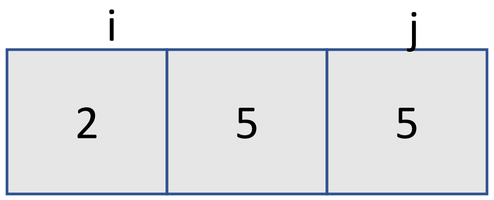


j = 2   A[j] = 5
i = 1   A[i] = 5

```
def InsertionSort(A)
 for j in range(1,len(A))
    key = A[j]
    i = j-1
    while i >= 0  and   A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

Example

key = 1

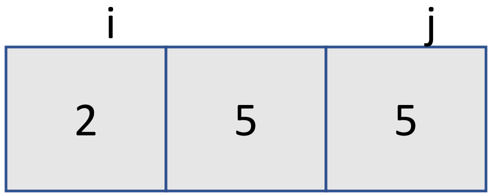


j = 1   A[j] = 5
i = 0   A[i] = 2

```
def InsertionSort(A)
 for j in range(1,len(A))
    key = A[j]
    i = j-1
    while i >= 0  and   A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

Example

key = 1

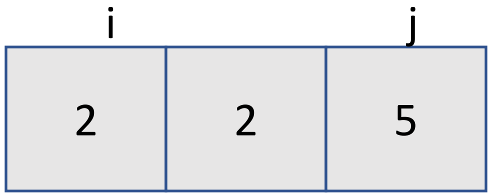


j = 1	A[j] = 5
i = 0	A[i] = 2

```
def InsertionSort(A)
 for j in range(1,len(A))
     key = A[j]
     i = j-1
     while i >= 0  and   A[i] > key
         A[i+1] = A[i]
         i = i - 1
     A[i+1] = key
```

Example

key = 1

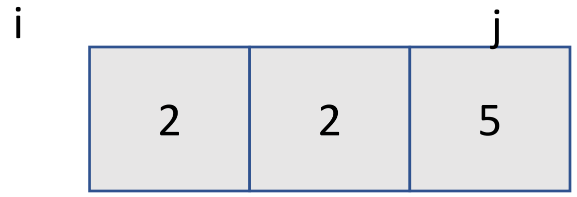


j = 2   A[j] = 5
i = 0   A[i] = 2

```
def InsertionSort(A)
 for j in range(1,len(A))
     key = A[j]
     i = j-1
     while i >= 0  and   A[i] > key
          A[i+1] = A[i]
          i = i - 1
     A[i+1] = key
```

Example

key = 1

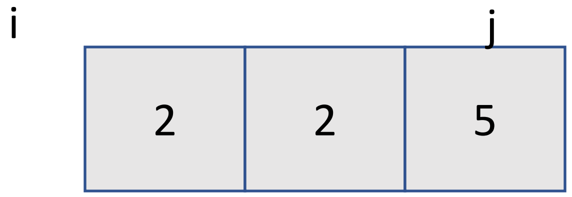


j = 2	A[j] = 5
i = -1	A[i] = N/A

```
def InsertionSort(A)
 for j in range(1,len(A))
     key = A[j]
     i = j-1
     while i >= 0  and   A[i] > key
         A[i+1] = A[i]
         i = i - 1
     A[i+1] = key
```

Example

key = 1

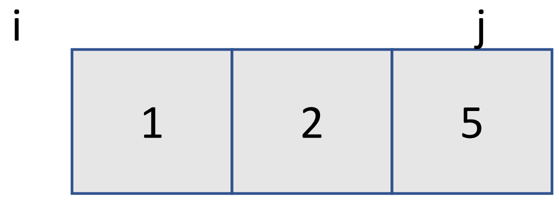


j = 2 A[j] = 5
i = -1 A[i] = N/A

```
def InsertionSort(A)
 for j in range(1,len(A))
     key = A[j]
     i = j-1
     while i >= 0  and   A[i] > key
         A[i+1] = A[i]
         i = i - 1
     A[i+1] = key
```

Example

key = 1

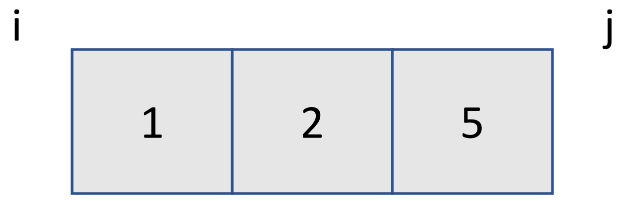


j = 2 A[j] = 5
i = -1 A[i] = N/A

```
def InsertionSort(A)
 for j in range(1,len(A))
     key = A[j]
     i = j-1
     while i >= 0  and   A[i] > key
         A[i+1] = A[i]
         i = i - 1
     A[i+1] = key
```

Example

key =



j = 3 A[j] = out of range
i = -1 A[i] = N/A

```
def InsertionSort(A)
    for j in range(1,len(A))
        key = A[j]
        i = j-1
        while i >= 0  and   A[i] > key
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

WE ARE DONE!

Analysis

- **Termination**
  - We terminate this case when *j* **goes out of bounds**

- **Correctness**
  - beginning of for-loop: if *A[1.. i]* sorted, then
  - end of for-loop: *A[1.. i+1]* sorted.

- **Efficiency: time/space**
  - Depends on input size *n*
  - Space: roughly *n*

Running Time

- In **general time taken by algorithm grows with the size of the input**
- So, traditionally, **running time is defined as a function of the size of the input**
- Input size depends upon the problem
  - For sorting problem it depends on number of values being sorted (e.g., size of input array)
  - For graph algorithms input depends on two values, # of vertices and # of edges of the network

Running Time Assumptions

- Running time of an algorithm in the number of primitive (basic) operations – steps --- executed
- Typically, we say each basic operation *i* takes a constant amount of time c_i
  - Note, <u>different primitive step may take a different amount of time</u>
  - But the time for that step is always the same, a constant

[IMPORTANT] 0-Index & 1-index

- Python uses zero index for its code. So do many programming languages

- <u>Cormen et al. uses one-index in its pseudocode</u>

- Therefore, you might notice some of differences in the code you see on the slides and in the textbook!!

Running Time of Insertion Sort

Assume 0 index

Assume ***n*** elements

	Cost	Times
`def InsertionSort(A)`		
`    for j in range(1,len(A))`	c1	n
`        key = A[j]`	c2	n-1
`# insert A[j] into the sorted portion of the A`	c3	0
`        i = j -1`	c4	n -1
`        while i > 0  and   A[i] > key`	c5	WHY? $\sum_{j=2}^{n} t_j$
`            A[i+1] = A[i]`	c6	$\sum_{j=2}^{n}(t_j -1)$
`            i = i - 1`	c7	$\sum_{j=2}^{n}(t_j -1)$
`        A[i+1] = key`	c8	n-1

Running time for Insertion Sort

$$T(n) = c1 * n + c2(n-1) + c4(n-1) + c5\sum_{j=2}^{n} t_j + c6\sum_{j=2}^{n} (t_j - 1) + c7\sum_{j=2}^{n} (t_j - 1) + c8(n-1)$$

Best Case Analysis

- When will the algorithm take the least amount of time?
    - When the array is already sorted (t_j = 1)
    - So the T(n) will be

$$\boxed{T(n) = c1 * n + c2(n-1) + c4(n-1) + c5(n-1) + c8(n-1)}$$

- Or T(n) is of the form **An +B**
- **Linear function** of input size, which is **n**

Worst Case Analysis

- When will the algorithm take the most amount of time?
  - When the array inverse sorted ($t_j = j$)
  - So the T(n) will be

WHY?

$$T(n) = c1 * n + c2(n-1) + c4(n-1) + c5\overbrace{\sum_{j=2}^{n} t_j}^{[n(n+1)/2] - 1} + c6\overbrace{\sum_{j=2}^{n} (t_j - 1)}^{n(n-1)/2} + c7\overbrace{\sum_{j=2}^{n} (t_j - 1)}^{n(n-1)/2} + c8(n-1)$$

  - Or T(n) is of the form **$An^2 + Bn + C$**
  - **Quadratic function** of input size, which is **n**

More on Worst Case Analysis

- Gives the upper-bound on the running time for **ANY** input
  - We cannot do any worse than this! IT will never take any longer.

- Worst case for an algorithm occurs fairly often --- example search algorithms --- which don’t find an entry in a database

- **Average case analysis** --- this computes on average how much running time of an algorithm

- This is useful sometimes, but most often it takes the same ball-park amount as the worst case.
  - What’s the average case T(n) for Insertion sort?
  - Depends on how many times the while loop executes on average.

Growth Functions

- We have used some simplifying abstractions to ease our analysis
  - replaced individual constants in the final value of T(n)
- Actually, we will use even more simplifications and and just focus on the leading terms of the formula
  - like ***an^2*** for the worst-case analysis
- This is because the order terms ***bn*** and ***c*** (lower-order terms) will always be < **an^2**
- *Next time we shall see how to represent the running time using what's called the **Big-O and Big-Theta** notations!*

That's all Folks!
Any Question?