Algorithmic Approaches for Biological Data, Lecture #13

Katherine St. John

City University of New York
American Museum of Natural History

7 March 2016

Outline

- Recap: Lists and Arrays

Outline

- Recap: Lists and Arrays
- Sorting:

Outline

- Recap: Lists and Arrays
- Sorting:
  - Insertion

Outline

- Recap: Lists and Arrays
- Sorting:
  - Insertion
  - Bubble

Outline

- Recap: Lists and Arrays
- Sorting:
  - Insertion
  - Bubble
  - Merge Sorts

Outline

- Recap: Lists and Arrays
- Sorting:
  - Insertion
  - Bubble
  - Merge Sorts
- *Break*

Outline

- Recap: Lists and Arrays
- Sorting:
  - Insertion
  - Bubble
  - Merge Sorts
- *Break*
- Analyzing sorts by running time on varying data sets (sorted, almost sorted, and random data)

Outline

- Recap: Lists and Arrays
- Sorting:
  - Insertion
  - Bubble
  - Merge Sorts
- *Break*
- Analyzing sorts by running time on varying data sets (sorted, almost sorted, and random data)
- Recursion: functions that call themselves

Recap: Lists

Creating lists: `a = []`

Recap: Lists

Creating lists:

```
a = []
b = ["hello", 3, 5]
```

Recap: Lists

Creating lists:

```
a = []
b = ["hello", 3, 5]
c = range(-10,11,2)
```

Recap: Lists

Creating lists:

```
a = []
b = ["hello", 3, 5]
c = range(-10,11,2)
```

Adding to lists:

```
d = a+b
```

Recap: Lists

Creating lists:

```
a = []
b = ["hello", 3, 5]
c = range(-10,11,2)
```

Adding to lists:

```
d = a+b
a.append["new"]
```

Recap: Lists

Creating lists:

```
a = []
b = ["hello", 3, 5]
c = range(-10,11,2)
```

Adding to lists:

```
d = a+b
a.append["new"]
```

Accessing elements:

```
b[2]
```

Recap: Lists

Creating lists:

```
a = []
b = ["hello", 3, 5]
c = range(-10,11,2)
```

Adding to lists:

```
d = a+b
a.append["new"]
```

Accessing elements:

```
b[2]
c[-3:-1]
```

Recap: Lists

Creating lists:	`a = []`
	`b = ["hello", 3, 5]`
	`c = range(-10,11,2)`
Adding to lists:	`d = a+b`
	`a.append["new"]`
Accessing elements:	`b[2]`
	`c[-3:-1]`
Deleting elements:	`del a[1]`

Recap: Lists

Creating lists:

```
a = []
b = ["hello", 3, 5]
c = range(-10,11,2)
```

Adding to lists:

```
d = a+b
a.append["new"]
```

Accessing elements:

```
b[2]
c[-3:-1]
```

Deleting elements:

```
del a[1]
del c[4:6]
```

Recap: Arrays

Assumes: `import numpy as np`

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays: `a = np.zeros(10)`

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
```

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
c = np.arange(-10,11,2)
```

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
c = np.arange(-10,11,2)
d = np.arange([3.4,5.555])
```

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
c = np.arange(-10,11,2)
d = np.arange([3.4,5.555])
e = np.linspace(0,100,50)
```

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
c = np.arange(-10,11,2)
d = np.arange([3.4,5.555])
e = np.linspace(0,100,50)
```

Adding to lists:

```
f = a+b
```

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
c = np.arange(-10,11,2)
d = np.arange([3.4,5.555])
e = np.linspace(0,100,50)
```

Adding to lists:

```
f = a+b
a.append["new"]
```

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
c = np.arange(-10,11,2)
d = np.arange([3.4,5.555])
e = np.linspace(0,100,50)
```

Adding to lists:

```
f = a+b
a.append["new"]
```

Accessing elements:

```
b[2]
```

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
c = np.arange(-10,11,2)
d = np.arange([3.4,5.555])
e = np.linspace(0,100,50)
```

Adding to lists:

```
f = a+b
a.append["new"]
```

Accessing elements:

```
b[2]
c[-3:-1]
```

Recap: Arrays

Assumes: `import numpy as np`

Creating arrays:

```
a = np.zeros(10)
b = np.ones(100)
c = np.arange(-10,11,2)
d = np.arange([3.4,5.555])
e = np.linspace(0,100,50)
```

Adding to lists:

```
f = a+b
a.append["new"]
```

Accessing elements:

```
b[2]
c[-3:-1]
```

Deleting elements:

np arrays are immutable, but can do:

```
newA = np.delete(a, 4)
```

In Pairs

In pairs/triples, work out (and then try at the shell or `pythonTutor`):

1. What does the following code do?

```
a = [12,3,1,50,18,6,15,34]
print a, a[::-3]
print max(a), max(a[::-3])
```

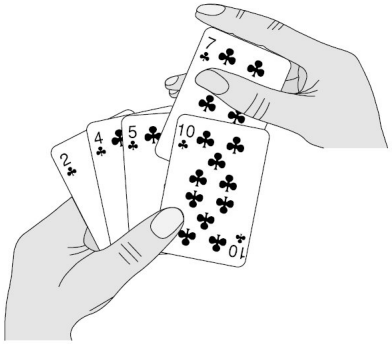
2. What does the following code do?

```
def mystery(a):
for i in range(1,len(a)):
    if a[i-1] > a[i]:
        a[i-1], a[i] = a[i], a[i-1]
a = [11,34,1,20,18,6,5,3]
print a
mystery(a)
print a
mystery(a)
print a
```

3. Describe how to find the largest card in a hand.
4. Describe how to sort a hand of cards.
5. What does the following code do?

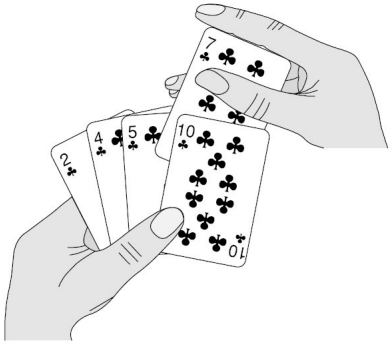
```
def mystery2(a,b):
    c = []
    ai = 0
    bi = 0
    while ai < len(a) and bi < len(b):
        if a[ai] < b[bi]:
            c.append(a[ai])
            ai += 1
        else:
        c.append(b[bi])
        bi += 1
    if ai < len(a):
        c = c + a[ai:]
    if bi < len(b):
        c = c + b[bi:]
    return c
x = [1,3,5,6,7,9]
y = [2,4,8,10,12,15]
z = mystery2(x,y)
print z
```

Sorting



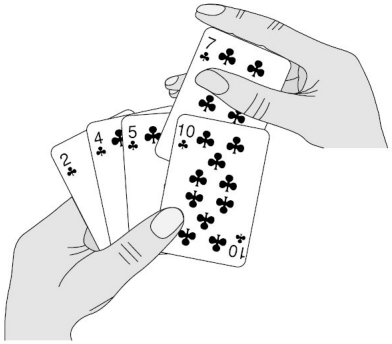
- How do you sort a hand of cards?

Sorting



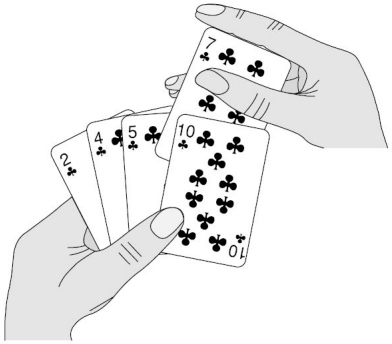
- How do you sort a hand of cards?
- Some common approaches:

Sorting



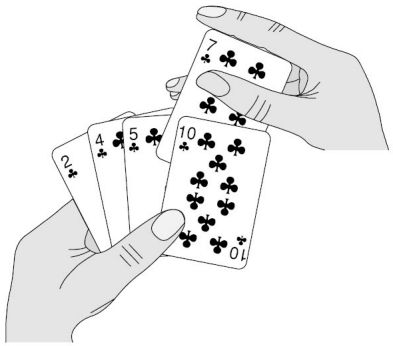
- How do you sort a hand of cards?
- Some common approaches:
  - Take the largest card and move to the end. Repeat with next largest...

Sorting



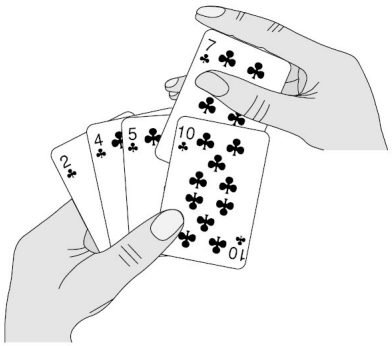
- How do you sort a hand of cards?
- Some common approaches:
  - Take the largest card and move to the end. Repeat with next largest...
  - Start a new list and insert each card into it to keep in order...

Sorting



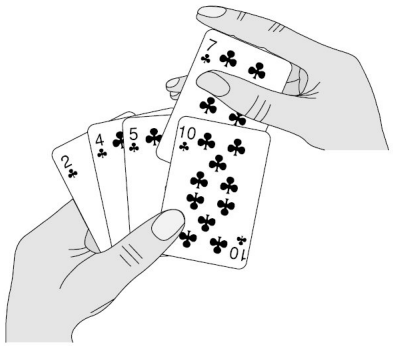
- How do you sort a hand of cards?
- Some common approaches:
  - Take the largest card and move to the end. Repeat with next largest...
  - Start a new list and insert each card into it to keep in order...
  - Divide the cards in half. Sort each half and merge sorted results together....

Sorting



- How do you sort a hand of cards?
- Some common approaches:
  - Take the largest card and move to the end. Repeat with next largest...
  - Start a new list and insert each card into it to keep in order...
  - Divide the cards in half. Sort each half and merge sorted results together....
- All need to compare values and re-order in some way

Sorting



- How do you sort a hand of cards?
- Some common approaches:
  - Take the largest card and move to the end. Repeat with next largest... BubbleSort
  - Start a new list and insert each card into it to keep in order...InsertionSort
  - Divide the cards in half. Sort each half and merge sorted results together....MergeSort
- All need to compare values and re-order in some way

BubbleSort

- Take the largest card and move to the end. Repeat with next largest...

BubbleSort

- Take the largest card and move to the end. Repeat with next largest...
- ```
  def mystery(a):
  for i in range(1,len(a)):
      if a[i-1] > a[i]:
          a[i-1], a[i] = a[i], a[i-1]
  ```
  ```

BubbleSort

- Take the largest card and move to the end. Repeat with next largest...
- ```
  def mystery(a):
  for i in range(1,len(a)):
      if a[i-1] > a[i]:
          a[i-1], a[i] = a[i], a[i-1]
  ```
- Add an outer loop to repeat:

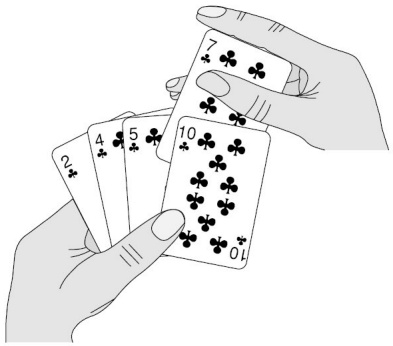
  ```
  def bubbleSort(a):
  for j in range(1,len(a)):
      for i in range(1,len(a)):
          if a[i-1] > a[i]:
              a[i-1], a[i] = a[i], a[i-1]
  ```
  ```

BubbleSort

- Take the largest card and move to the end. Repeat with next largest...
- ```
  def mystery(a):
  for i in range(1,len(a)):
      if a[i-1] > a[i]:
          a[i-1], a[i] = a[i], a[i-1]
  ```
- Add an outer loop to repeat:

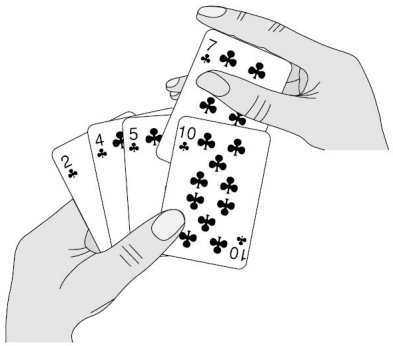
  ```
  def bubbleSort(a):
  for j in range(1,len(a)):
      for i in range(1,len(a)):
          if a[i-1] > a[i]:
              a[i-1], a[i] = a[i], a[i-1]
  ```
- *Python Tutor demo*
  ```

Insertion Sort



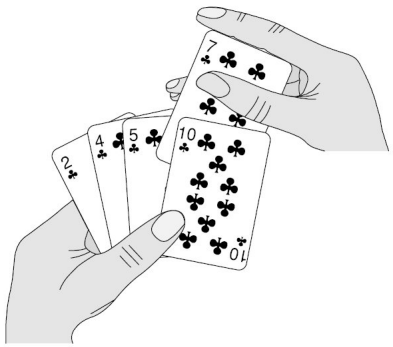
- Start a new list and insert each card into it to keep in order...

Insertion Sort



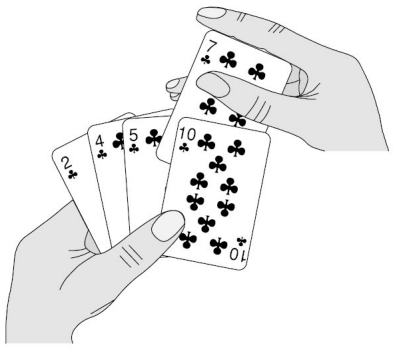
- Start a new list and insert each card into it to keep in order...
  - Start with the first element.

Insertion Sort



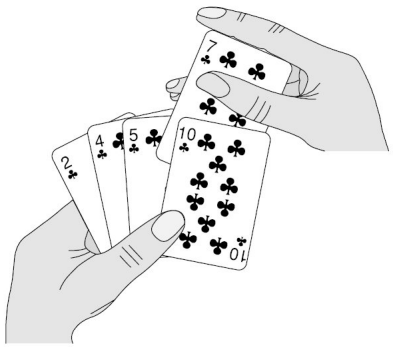
- Start a new list and insert each card into it to keep in order...
  - Start with the first element.
  - Compare the second one to it. If smaller, swap.

Insertion Sort



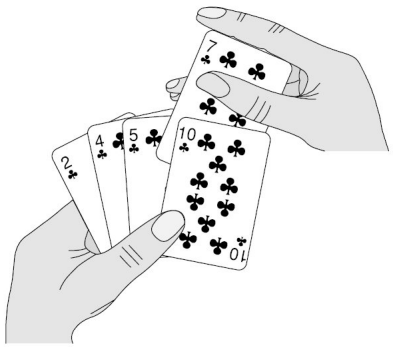
- Start a new list and insert each card into it to keep in order...
  - Start with the first element.
  - Compare the second one to it. If smaller, swap.
  - Add the third element to the list, checking it against the first two.

Insertion Sort



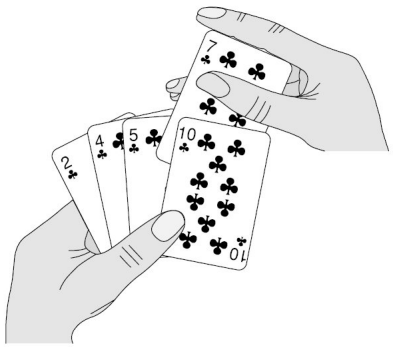
- Start a new list and insert each card into it to keep in order...
  - Start with the first element.
  - Compare the second one to it. If smaller, swap.
  - Add the third element to the list, checking it against the first two.
  - Add the fourth element, checking it against the first three...

Insertion Sort



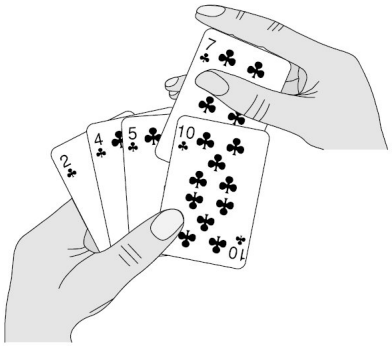
- Start a new list and insert each card into it to keep in order...
  - Start with the first element.
  - Compare the second one to it. If smaller, swap.
  - Add the third element to the list, checking it against the first two.
  - Add the fourth element, checking it against the first three...
- When adding a new element to the sublist, start at the top and move elements to make room.

Insertion Sort



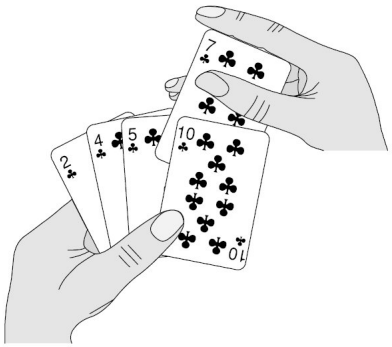
- Start a new list and insert each card into it to keep in order...
  - Start with the first element.
  - Compare the second one to it. If smaller, swap.
  - Add the third element to the list, checking it against the first two.
  - Add the fourth element, checking it against the first three...
- When adding a new element to the sublist, start at the top and move elements to make room.
- *wiki demo*

Insertion Sort



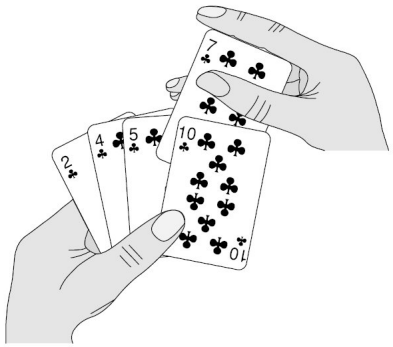
- Start a new list and insert each card into it to keep in order...

Insertion Sort



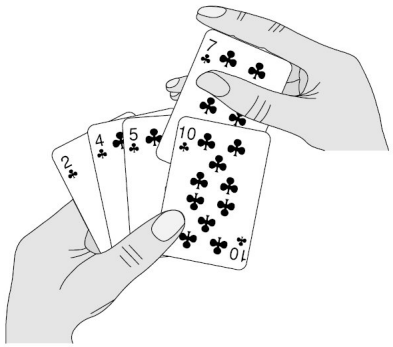
- Start a new list and insert each card into it to keep in order...
- When adding a new element to the sublist, start at the top and move elements to make room.

Insertion Sort



- Start a new list and insert each card into it to keep in order...
- When adding a new element to the sublist, start at the top and move elements to make room.
- ```
  def insertionSort(a):
      for i in range(1,len(a)):
          j = i
          while j > 0 and a[j-1] > a[j]:
              a[j], a[j-1] = a[j-1], a[j]
              j = j-1
  ```
  ```

Insertion Sort



- Start a new list and insert each card into it to keep in order...
- When adding a new element to the sublist, start at the top and move elements to make room.
- ```
  def insertionSort(a):
      for i in range(1,len(a)):
          j = i
          while j > 0 and a[j-1] > a[j]:
              a[j], a[j-1] = a[j-1], a[j]
              j = j-1
  ```
- *PythonTutor demo*
  ```

In Pairs

In pairs/triples, work through

1. *Problem Solving with Algorithms and Data Structures*, BubbleSort chapter (link on webpage)
2. *Problem Solving with Algorithms and Data Structures*, InsertionSort chapter (link on webpage)
3. If time, *Problem Solving with Algorithms and Data Structures*, MergeSort chapter (link on webpage)

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....
- Idea:

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....
- Idea:
  `mergeSort(a):`

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....
- Idea:

```
mergeSort(a):
1.  If len(a) > 1, then
```

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....
- Idea:

```
mergeSort(a):
1.  If len(a) > 1, then
2.  mid = len(a)/2
```

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....
- Idea:

```
mergeSort(a):
1.  If len(a) > 1, then
2.  mid = len(a)/2
3.  mergeSort(a[:mid])
```

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....
- Idea:

```
mergeSort(a):
1.  If len(a) > 1, then
2.  mid = len(a)/2
3.  mergeSort(a[:mid])
4.  mergeSort(a[mid:])
```

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....
- Idea:

```
mergeSort(a):
1.  If len(a) > 1, then
2.  mid = len(a)/2
3.  mergeSort(a[:mid])
4.  mergeSort(a[mid:])
5.  merge lists together
```

MergeSort

- Divide the cards in half. Sort each half and merge sorted results together....
- Idea:

```
mergeSort(a):
1.  If len(a) > 1, then
2.  mid = len(a)/2
3.  mergeSort(a[:mid])
4.  mergeSort(a[mid:])
5.  merge lists together
```

- We did the "merge" in pairs.
- *ProblemSolving book's demo*

Break

Comparing Algorithms

- Measure the size of the problem, usually called n.

Comparing Algorithms

- Measure the size of the problem, usually called n.
- Example: for sorting cards, n is the number of cards.

Comparing Algorithms

- Measure the size of the problem, usually called n.
- Example: for sorting cards, n is the number of cards.
- Different approaches can take different amounts of time.

Comparing Algorithms

- Measure the size of the problem, usually called n.
- Example: for sorting cards, n is the number of cards.
- Different approaches can take different amounts of time.
- How long does the algorithm take proportional to n?

Comparing Algorithms

- Measure the size of the problem, usually called n.
- Example: for sorting cards, n is the number of cards.
- Different approaches can take different amounts of time.
- How long does the algorithm take proportional to n?
- *Sorting Algorithms demo*

  Not in demo is the built-in Python sort: timSort (invented by Tim Peters in 2002) that is hybrid of merge sort and insertion sort.

Analysis of Algorithms

- How long does the algorithm take proportional to n?

Analysis of Algorithms

- How long does the algorithm take proportional to n?
- If an algorithm looks at each element once (or a constant number of times), the running time is proportional to n, the number of elements.

Analysis of Algorithms

- How long does the algorithm take proportional to n?
- If an algorithm looks at each element once (or a constant number of times), the running time is proportional to n, the number of elements.
- Then, the algorithm runs in linear time.

Analysis of Algorithms

- How long does the algorithm take proportional to n?
- If an algorithm looks at each element once (or a constant number of times), the running time is proportional to n, the number of elements.
- Then, the algorithm runs in linear time.
- Would write "the running time is $O(n)$." ("big-Oh" notation).

Analysis of Algorithms

- How long does the algorithm take proportional to n?
- If an algorithm looks at each element once (or a constant number of times), the running time is proportional to n, the number of elements.
- Then, the algorithm runs in linear time.
- Would write "the running time is $O(n)$." ("big-Oh" notation).
- Usually measure the worst-case running time.

Analysis of Algorithms

- How long does the algorithm take proportional to n?
- If an algorithm looks at each element once (or a constant number of times), the running time is proportional to n, the number of elements.
- Then, the algorithm runs in linear time.
- Would write "the running time is $O(n)$." ("big-Oh" notation).
- Usually measure the worst-case running time.

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

```
def bubbleSort(a):  #Let n be # of elements in a.
```

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

```
def bubbleSort(a):  #Let n be # of elements in a.
for j in range(1,len(a)):  #Will go through this loop n times.
```

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

```
def bubbleSort(a):  #Let n be # of elements in a.
for j in range(1,len(a)):  #Will go through this loop n times.
    for i in range(1,len(a)):  #Will go through this loop n times.
```

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

```
def bubbleSort(a):  #Let n be # of elements in a.
for j in range(1,len(a)):  #Will go through this loop n times.
    for i in range(1,len(a)):  #Will go through this loop n times.
        if a[i-1] > a[i]:  #Takes constant time.
```

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

```
def bubbleSort(a):  #Let n be # of elements in a.
for j in range(1,len(a)):  #Will go through this loop n times.
    for i in range(1,len(a)):  #Will go through this loop n times.
        if a[i-1] > a[i]:  #Takes constant time.
            a[i-1], a[i] = a[i], a[i-1] #Takes constant time.
```

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

```
def bubbleSort(a):  #Let n be # of elements in a.
for j in range(1,len(a)):  #Will go through this loop n times.
    for i in range(1,len(a)):  #Will go through this loop n times.
        if a[i-1] > a[i]:  #Takes constant time.
            a[i-1], a[i] = a[i], a[i-1] #Takes constant time.
```

- The lines in the if statement take constant time, but are performed $n \cdot n$ time.

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

```
def bubbleSort(a):  #Let n be # of elements in a.
for j in range(1,len(a)):  #Will go through this loop n times.
    for i in range(1,len(a)):  #Will go through this loop n times.
        if a[i-1] > a[i]:  #Takes constant time.
            a[i-1], a[i] = a[i], a[i-1] #Takes constant time.
```

- The lines in the if statement take constant time, but are performed $n \cdot n$ time.
- Upper bound on running time is $O(c \cdot n \cdot n) = O(n^2)$.

Analyzing Sorts by Running Times

- The sorting algorithms vary in running time, depending on number of elements and type of data.
- *Sorting Demo*
- Thinking about the worst-case, how many operations are performed in bubbleSort?

```
def bubbleSort(a):  #Let n be # of elements in a.
for j in range(1,len(a)):  #Will go through this loop n times.
    for i in range(1,len(a)):  #Will go through this loop n times.
        if a[i-1] > a[i]:  #Takes constant time.
            a[i-1], a[i] = a[i], a[i-1] #Takes constant time.
```

- The lines in the if statement take constant time, but are performed $n \cdot n$ time.
- Upper bound on running time is $O(c \cdot n \cdot n) = O(n^2)$.

(For big-Oh notation, drop constants and keep only largest terms.)

Recursion

- In many programming languages, functions can call themselves.

Recursion

- In many programming languages, functions can call themselves.
- This is called recursion (versus iteration).

Recursion

- In many programming languages, functions can call themselves.
- This is called recursion (versus iteration).
- Can be a bit confusing to trace through, but Python treats like any other function call.

In Pairs

In pairs/triples, work out (and then try at the shell, `nested.py` on webpage):

1. What does the following do:

```
def nested(t, k):
    for i in range(3):
        t.left(120)
        t.forward(k)
    if k > 10:
        nested(t, k/2)
tess = turtle.Turtle()
tess.shape("turtle")
nested(tess, 320)
```

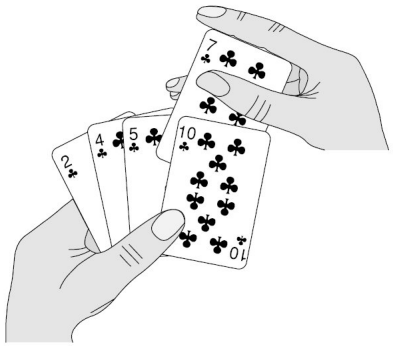
2. Modify the program to draw nested squares.

3. What does the following do:

```
def nested2(t, k):
    for i in range(3):
        t.left(120)
        t.forward(k)
        if k > 10:
            nested2(t, k/2)
tess = turtle.Turtle()
tess.shape("turtle")
nested2(tess, 320)
```

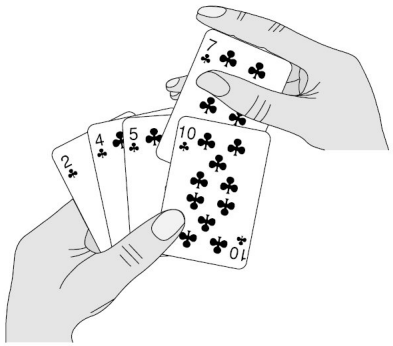
4. Modify the program to draw nested squares.

Recap



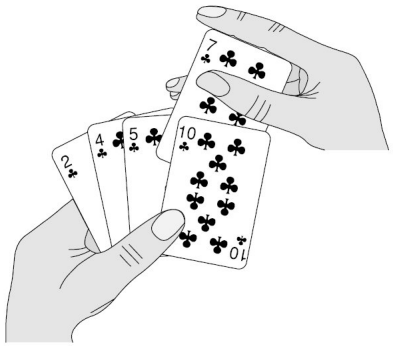
- No class next week (Spring Break).

Recap



- No class next week (Spring Break).
- Email lab reports to `kstjohn@amnh.org`

Recap



- No class next week (Spring Break).
- Email lab reports to `kstjohn@amnh.org`
- Challenges available at `rosalind.info`