

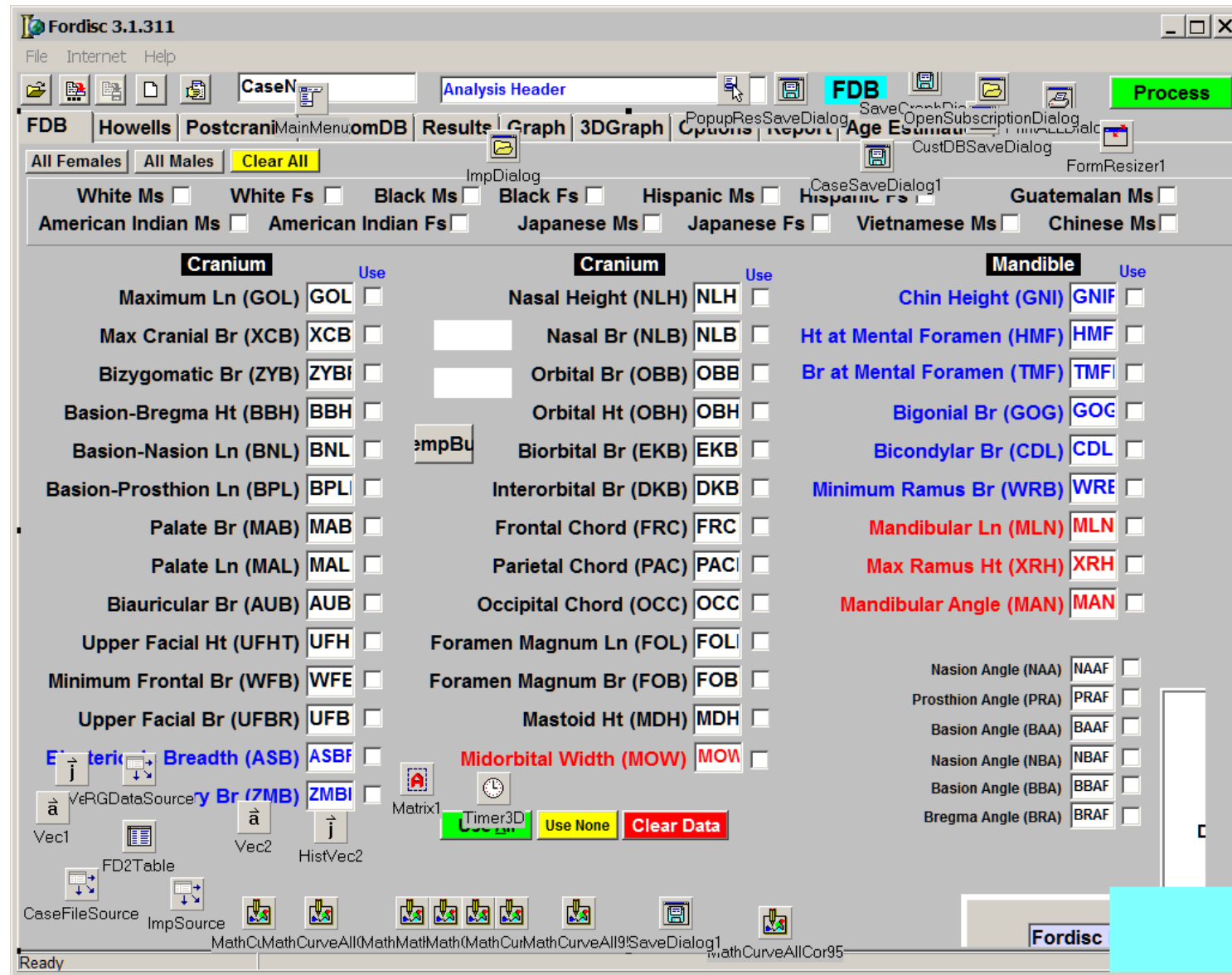
DATA 520

Lecture 19

Search and Sort Algorithms II

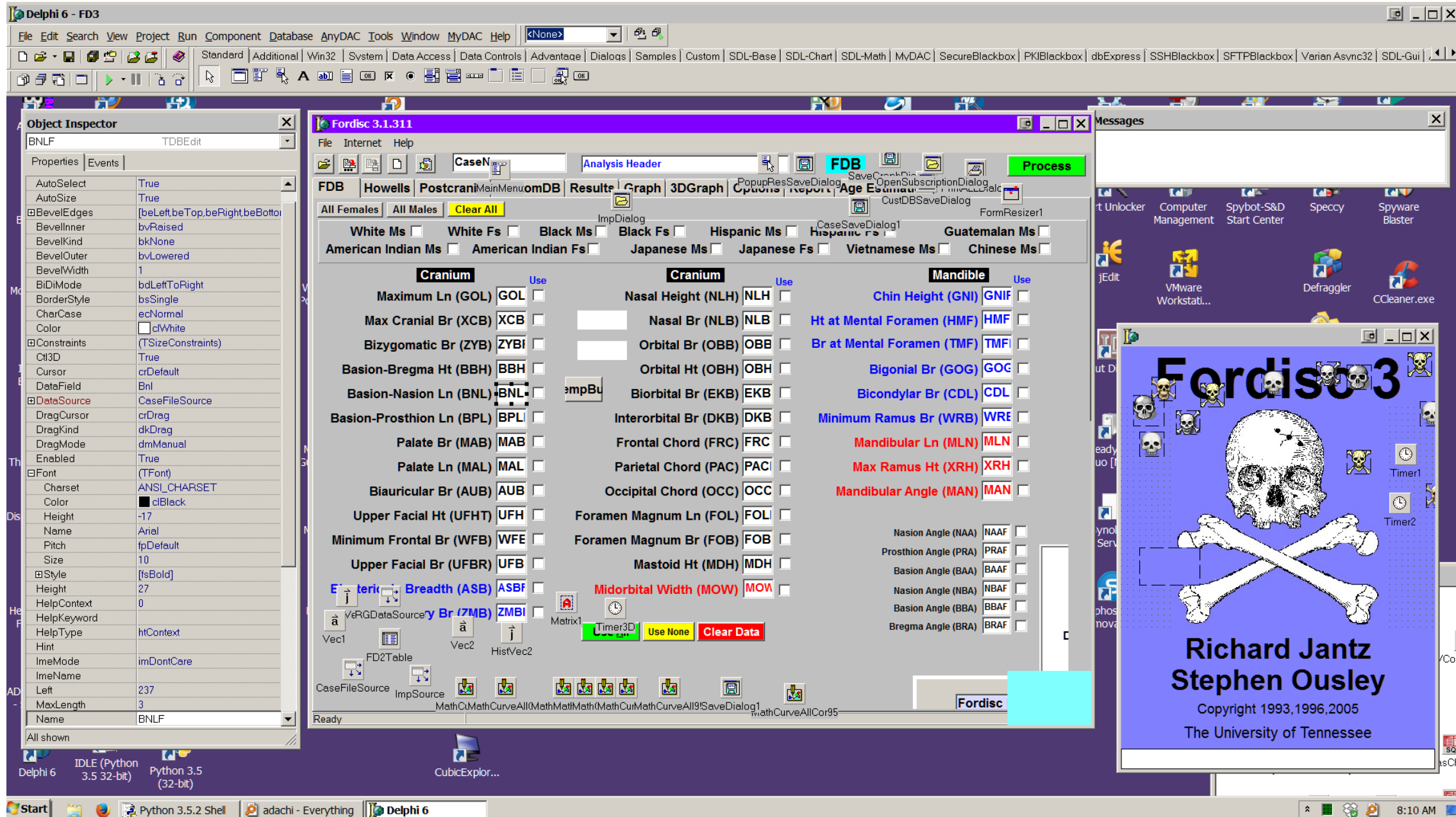
Application Monday

Delphi 6: GUI and IDE (Fordisc)



Delphi 6: GUI and IDE

Application Monday



Application Monday

Delphi 6: Pascal

```
procedure TFD3Frm1.ChkForFsClick(Sender: TObject);
var i: integer;
begin
    DM1.casetable.edit;    //choose all females    code completion
    try
        For i := 0 to FD3Frm1.ComponentCount -1 do

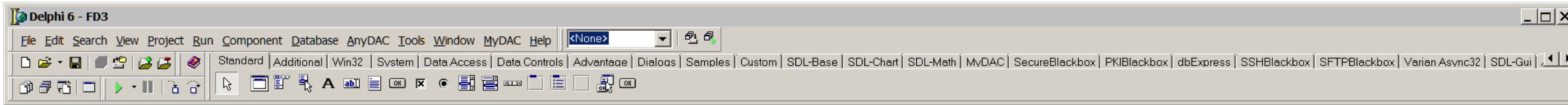
            If (FD3Frm1.Components[i].tag = 4) and (FD3Frm1.Components[i] is TDBCcheckbox)
            and (pos(' Fs',TDBCcheckbox(FD3Frm1.Components[i]).caption) > 0)
            // or (pos('Indian Fs',TDBCcheckbox(FD3Frm1.Components[i]).caption) > 0)
            then
                begin

                    DM1.casetable[TDBCcheckbox(FD3Frm1.Components[i]).datafield] := 1; //True;
                    Tcheckbox(FD3Frm1.Components[i]).CHECKED := true;

                end;
            except
            end;

        end;
end;
```

Application Monday

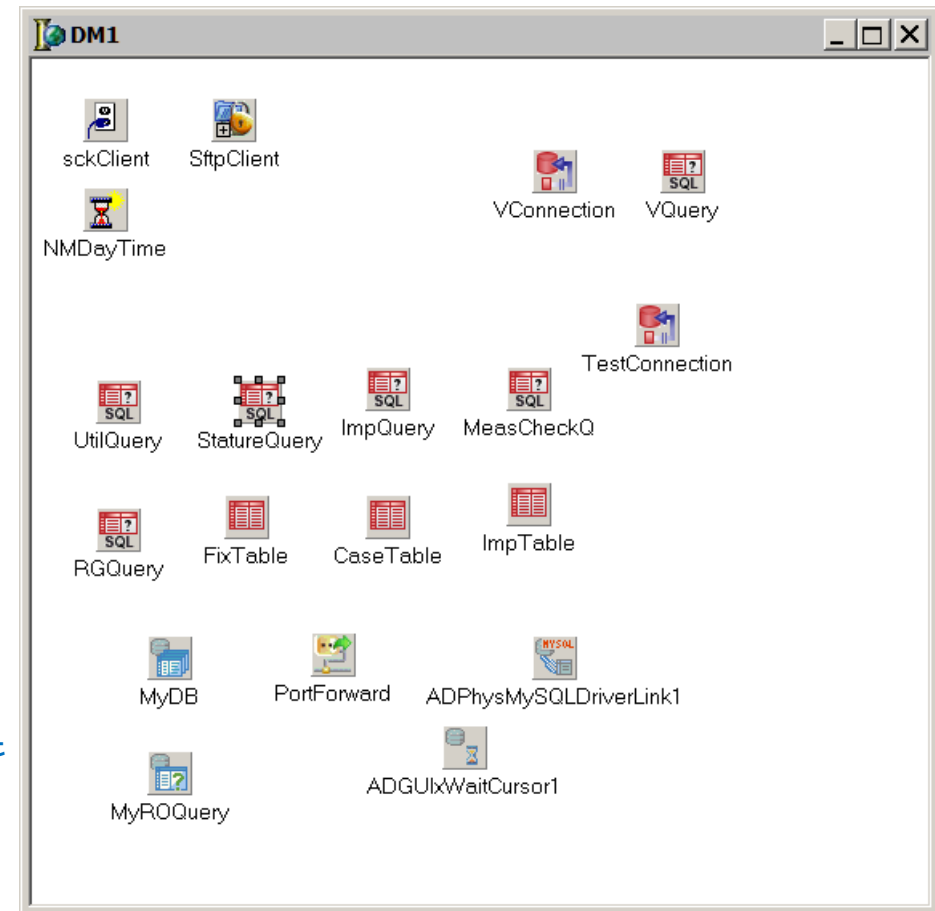


Delphi 6

Objects have properties and **methods**

```
object BNLF: TDBEdit
  Tag = 7
  Left = 237
  Top = 241
  Width = 41
  Height = 27
  Color = clWhite
  DataField = 'Bnl'
  DataSource = CaseFileSource
  Font.Charset = ANSI_CHARSET
  Font.Color = clBlack
  Font.Height = -17
  Font.Name = 'Arial'
  Font.Style = [fsBold]
  MaxLength = 3
  ParentFont = False
  TabOrder = 4
  OnChange = GOLFFChange
  OnEnter = GOLFFEnter
  OnExit = GOLFFExit
  OnKeyPress = GOLFFKeyPress
end
```

```
object MyDB: TADConnection
  Params.Strings = (
    'Compress=False'
    'Host=math.mercyhurst.edu'
    'AllowReconnect=True'
    'AutoCommit=False'
    'DriverID=MySQL')
  LoginPrompt = False
  BeforeConnect = MyDBBeforeConnect
  Left = 72
  Top = 360
end
```



Application Monday

Delphi 6: Create Fordisc installation files with serial numbers (DLL files):

- for PC, USB, Site License, Subscription (for secure labs)

Create Fordisc Installation Files 1.32

☐ Mercyhurst from to

☐ UTK 3839 3839

3839

Type

☐ PC ☐ USB ☒ Site ☐ Subscription

Do it!

- creates BOTH SL files
- uses relocated ZEOSLib

- 001-200 November 2005
- 201-300 USB 2006
- 101-200 REDone Sep 2006
- 301-400 USB done Feb 2007
- 401-500 PC Feb 2008
- 501-600 USB May 2008
- 601-700 PC June 2009
- 701-800 USB Dec 2010
- 800-899 Site License
- 900-999 PC Feb 2011
- 4001-4030 Sub: JPAC1 9/2011
- 4031-4040 Sub: CMP Cypress
- 4041-4060 Sub: JPAC2

Application Monday

Delphi 6:
program to keep track of
Fordisc subscription
information
(MySQL)

AnyDAC-SSH Connect to Mercyhurst anthfs server 1.15B

Query | Insert | Update | Delete

```
SELECT * FROM fd3sl.fd3sli /* a comment */  
-- SELECT * from radiographic.Demographics  
# another comment
```

Print ResultSet
DISConnect SSH
Disconnect
Execute
☐ Filter Location
University
Copy Info
Backup
☒ Top 100

FD3SerNo	Institution	ServerName
3389	Mercyhurst University	math.mercyhurst.edu
3800	University of Tennessee	webreports2.dii.utk.edu
3801	Western Washington University	west.wvu.edu
3802	Baylor University	sts01.baylor.edu
3803	University of Wisconsin-Madison	sscweb1.ads.ssc.wisc.edu
3804	University of Florida	ns.ufl.edu
3805	Universitaet Mainz	pc05-479.biologie.uni-mainz.de
3806	Chico State University	stat2.csuchico.edu
3807	Liverpool John Moores University	jmudas3.jmu.ac.uk
3808	Texas A + M University	libarts.tamu.edu
3809	University of Lincoln	cslic01.lincoln.ac.uk
3810	SUNY Oswego	ls1.oswego.edu
3811	University of Montana	um-licensing.umt.edu
3812	Florida International University	wr2-v200.fiu.edu
3813	University of Michigan-Dearborn	cicero.its.umd.umich.edu
3814	George Mason University	www.gmu.edu
3815	Bond University	gw-14.mgt.bond.edu
3816	University of West Georgia	www.westga.edu
3817	Drew University	oz.drew.edu
3818	Valdosta State University	li.valdosta.edu
3819	Academic Medical Center, U. of Amsterdam	proxy.amc.nl

SSH port 22 validated.
Database connected.

Protocol: mysql
Host Name: anthfs.mercyhurst.edu
Database: fd3sl
User Name: fd3sluser
Password: readonly123

Binary Search Times

Remember, these were on sorted lists

time_searches2.py output

Search times in a list of 10,000,001

Index	while	for	Sent	BinS	.index
10	0.00	0.00	0.00	0.01	0.00
5000000	897.03	349.20	472.75	0.01	68.98
9999951	1807.62	706.44	942.70	0.01	139.35

Sorting

What if we want to find the largest n or smallest n values in a list?

```
>>> CBA=[563,7590,1708,2142,3323,6197,1985,1316,1824,472,  
1346,6029,2670,2094,2464,1009,1475,856,3027,4271,  
3126,1115,2691,4253,1838,828,2403,742,1017,613,  
3185,2599,2227,896,975,1358,264,1375,2016,452,  
3292,538,1471,9313,864,470,2993,521,1144,2212,  
2212,2331,2616,2445,1927,808,1963,898,2764,2073,  
500,1740,8592,10856,2818,2284,1419,1328,1329,1479]
```

```
>>> scopy = sorted(CBA)
```

```
>>> scopy[-3:] # return largest 3
```

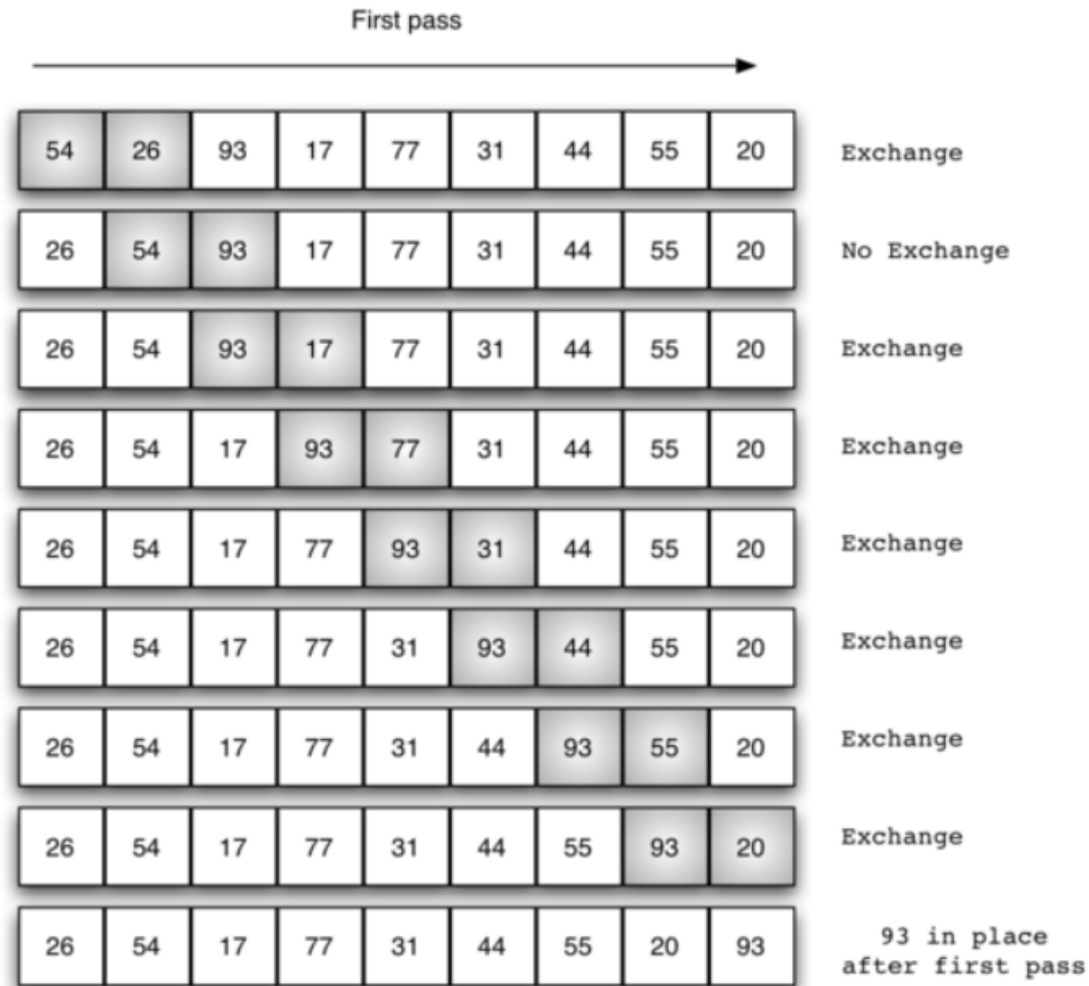
```
>>> scopy[0:3] # return smallest 3
```

But how does Python sort?

Bubble Sort

interactive sorting

<https://interactivepython.org/runestone/static/pythonds/SortSearch/toctree.html>



Bubble Sort

Must go through n times

- after first pass: largest value last ($n-1$ comparisons)
- after second, last 2 are the largest ($n-2$ comparisons)
- so we will need $n-1$ passes to get all in place and a lot of comparisons (n^2)

Big $O(n^2)$: how an algorithm scales to

On average, we trade places half of the time.

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1): # n times ; here: = range(8,0,-1)
        for i in range(passnum): # n times
            if alist[i] > alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

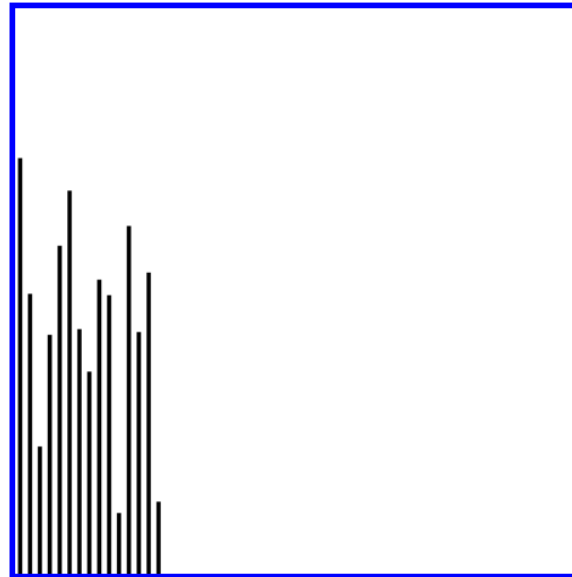
Bubble Sort

interactive sorting

<https://interactivepython.org/runestone/static/pythonds/SortSearch/toctree.html>

ActiveCode: 1 The Bubble Sort (lst_bubble)

The following animation shows `bubbleSort` in action.



Initialize Run Stop
Beginning Step Forward Step Backward End

To analyze the bubble sort, we should note that regardless of how the items are arranged in the initial list, $n - 1$ passes will be made to sort a list of size n . Table 1 shows the number of comparisons for each pass. The total number of comparisons is the sum of the first $n - 1$ integers. Recall that the sum of the first n integers is $\frac{1}{2}n^2 + \frac{1}{2}n$. The sum of the first $n - 1$ integers is $\frac{1}{2}n^2 + \frac{1}{2}n - n$, which is $\frac{1}{2}n^2 - \frac{1}{2}n$. This is still $O(n^2)$ comparisons. In the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.

Bubble Sort

If no exchanges, we know the list is sorted.

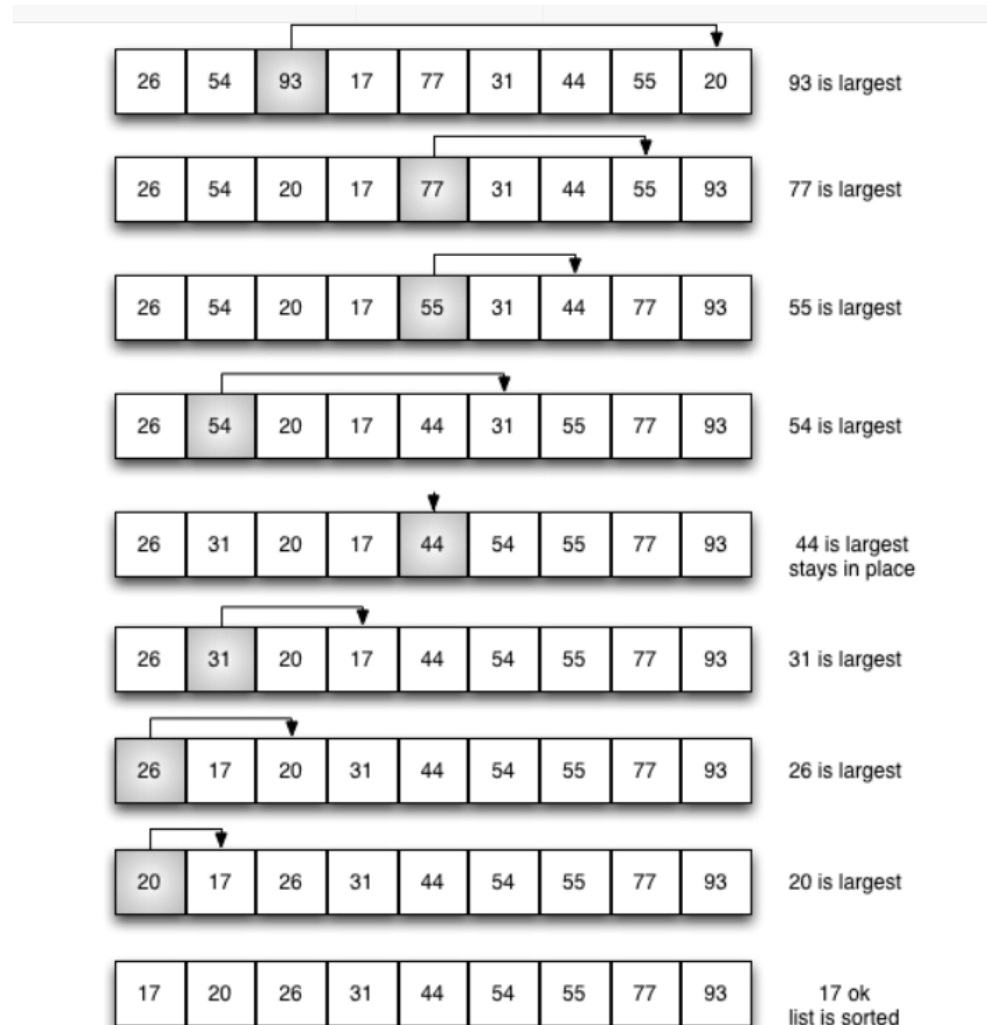
- will work quickly if only a few unsorted individuals
- so we can look for that happening

```
def shortBubbleSort(alist):  
    exchanges = True  
    passnum = len(alist)-1  
    while passnum > 0 and exchanges:  
        exchanges = False  
        for i in range(passnum):  
            if alist[i]>alist[i+1]:  
                exchanges = True  
                temp = alist[i]  
                alist[i] = alist[i+1]  
                alist[i+1] = temp  
        passnum = passnum-1  
  
alist=[20,30,40,90,50,60,70,80,100,110]  
shortBubbleSort(alist)  
print(alist)
```

Let's try some sorting using volunteers ...

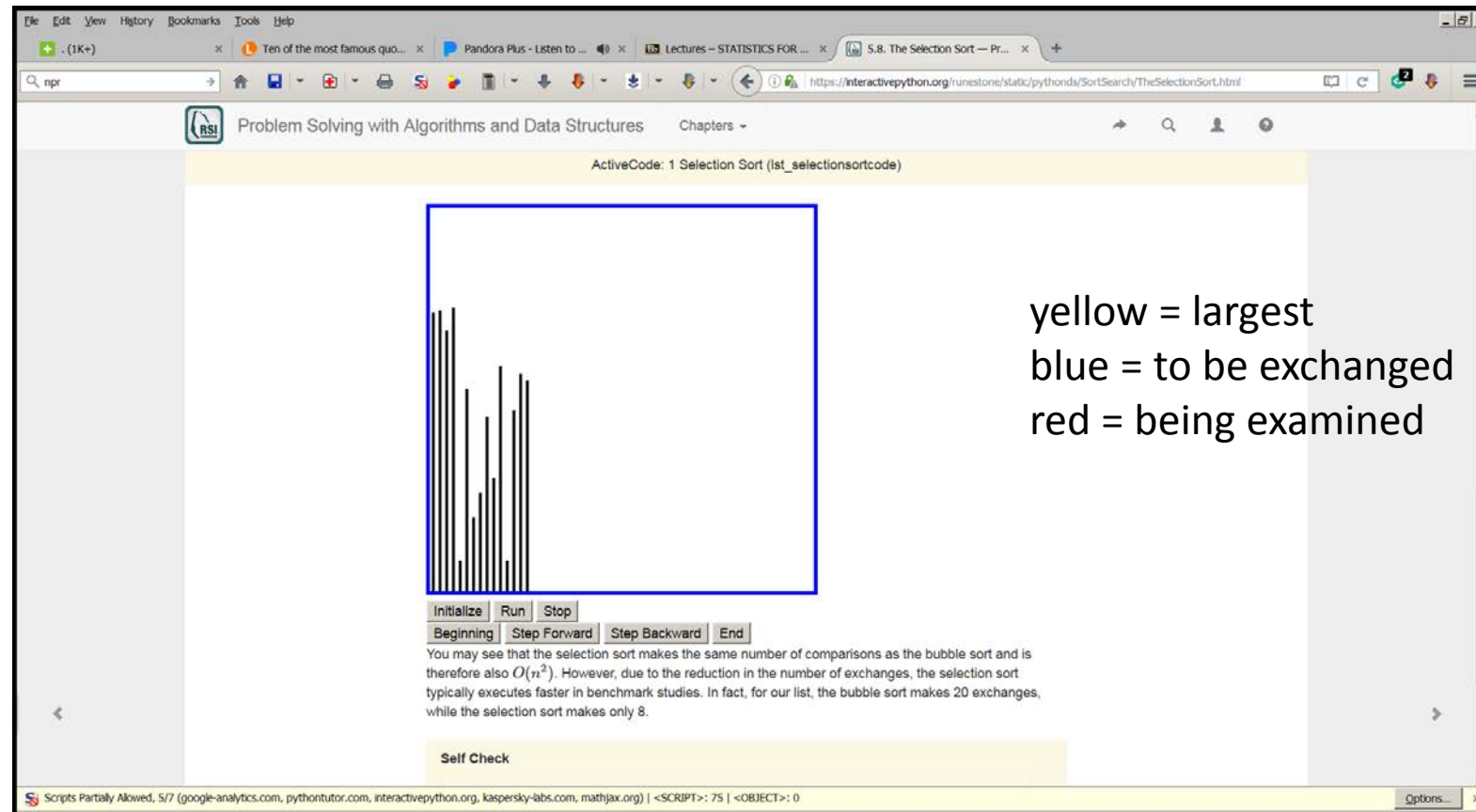
Selection Sort

Find the smallest item in the list and swap places with value in first position (0)
(or find the largest and swap with the last position)



Selection Sort

Find the smallest item in the list and swap places with value in first position (0)
(or find the largest and swap with the last position)
it does n^2 comparisons (looks at every one (n , $n-1$, $n-2$, etc.) to find the max)
- but it performs fewer exchanges than bubble sort



Selection Sort

Find the smallest item in the list and swap places with value in first position (0)
(or find the largest and swap with the last position)
Python has a special one-line method for swapping
(usually: `a,b; temp = a; a = b; b = temp;`)

```
def selection_sort(L):
    i = 0
    while i != len(L):
        # Find the index of the smallest item in L[i:]
        # a fast way: L.index(min(L))
        # without using min() function:
        smallest = find_min(L, i)
        L[i], L[smallest] = L[smallest], L[i]
        i = i + 1
```

```
def find_min(L,b):
    smallest = b # The index of the smallest so far.
    i = b + 1
    while i != len(L):
        if L[i] < L[smallest]:
            # We found a smaller item at L[i].
            smallest = i

        i = i + 1

    return smallest
```

doctest:

```
""" (list) -> NoneType
Reorder the items in L from smallest to largest.
>>> L = [3, 4, 7, -1, 2, 5]
>>> selection_sort(L)
>>> L
[-1, 2, 3, 4, 5, 7]
"""
```


Selection Sort

Find the smallest item in the list and swap places with value in first position (0)

```
def selection_sort(L): # we need expanded doctests
    """ (list) -> NoneType
    Reorder the items in L from smallest to largest.
    >>> L = [3, 4, 7, -1, 2, 5]
    >>> selection_sort(L)
    >>> L
    [-1, 2, 3, 4, 5, 7]
    >>> L = []
    >>> selection_sort(L)
    >>> L
    []
    >>> L = [1]
    >>> selection_sort(L)
    >>> L
    [1]
    >>> L = [2, 1]
    >>> selection_sort(L)
    >>> L
    [1, 2]
    >>> L = [1, 2]
    >>> selection_sort(L)
    >>> L
    [1, 2]
    >>> L = [3, 3, 3]
    >>> selection_sort(L)
    >>> L
    [3, 3, 3]
```

```
>>> L = [-5, 3, 0, 3, -6, 2, 1, 1]
>>> selection_sort(L)
>>> L
[-6, -5, 0, 1, 1, 2, 3, 3]
"""
i = 0
while i != len(L):
    # Find the index of the smallest item in L[i:]
    # a fast way: L.index(min(L))
    # without using min() function:
    smallest = find_min(L, i)
    L[i], L[smallest] = L[smallest], L[i]
    i = i + 1
```

```
def find_min(L, b):
    smallest = b # The index of the smallest so far.
    i = b + 1
    while i != len(L):
        if L[i] < L[smallest]:
            # We found a smaller item at L[i].
            smallest = i

        i = i + 1

    return smallest
```

```
# doctest with verbose = True
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Selection Sort

```
def selection_sort(L):
    """ (list) -> NoneType
    Reorder the items in L from smallest to largest.
    >>> L = [3, 4, 7, -1, 2, 5]
    >>> selection_sort(L)
    >>> L
    [-1, 2, 3, 4, 5, 7]
    >>> L = []
    >>> selection_sort(L)
    >>> L
    []
    >>> L = [1]
    >>> selection_sort(L)
    >>> L
    [1]
    >>> L = [2, 1]
    >>> selection_sort(L)
    >>> L
    [1, 2]
    >>> L = [1, 2]
    >>> selection_sort(L)
    >>> L
    [1, 2]
    >>> L = [3, 3, 3]
    >>> selection_sort(L)
    >>> L
    [3, 3, 3]
    >>> L = [-5, 3, 0, 3, -6, 2, 1, 1]
    >>> selection_sort(L)
    >>> L
    [-6, -5, 0, 1, 1, 2, 3, 3]
    """
    i = 0
    while i != len(L):
        # Find the index of the smallest item in L[i:]
        # a fast way: L.index(min(L))
        # without using min() function:
        smallest = find_min(L, i)
        L[i], L[smallest] = L[smallest], L[i]
        i = i + 1

def find_min(L, b):
    smallest = b # The index of the smallest so far.
    i = b + 1
    while i != len(L):
        if L[i] < L[smallest]:
            # We found a smaller item at L[i].
            smallest = i

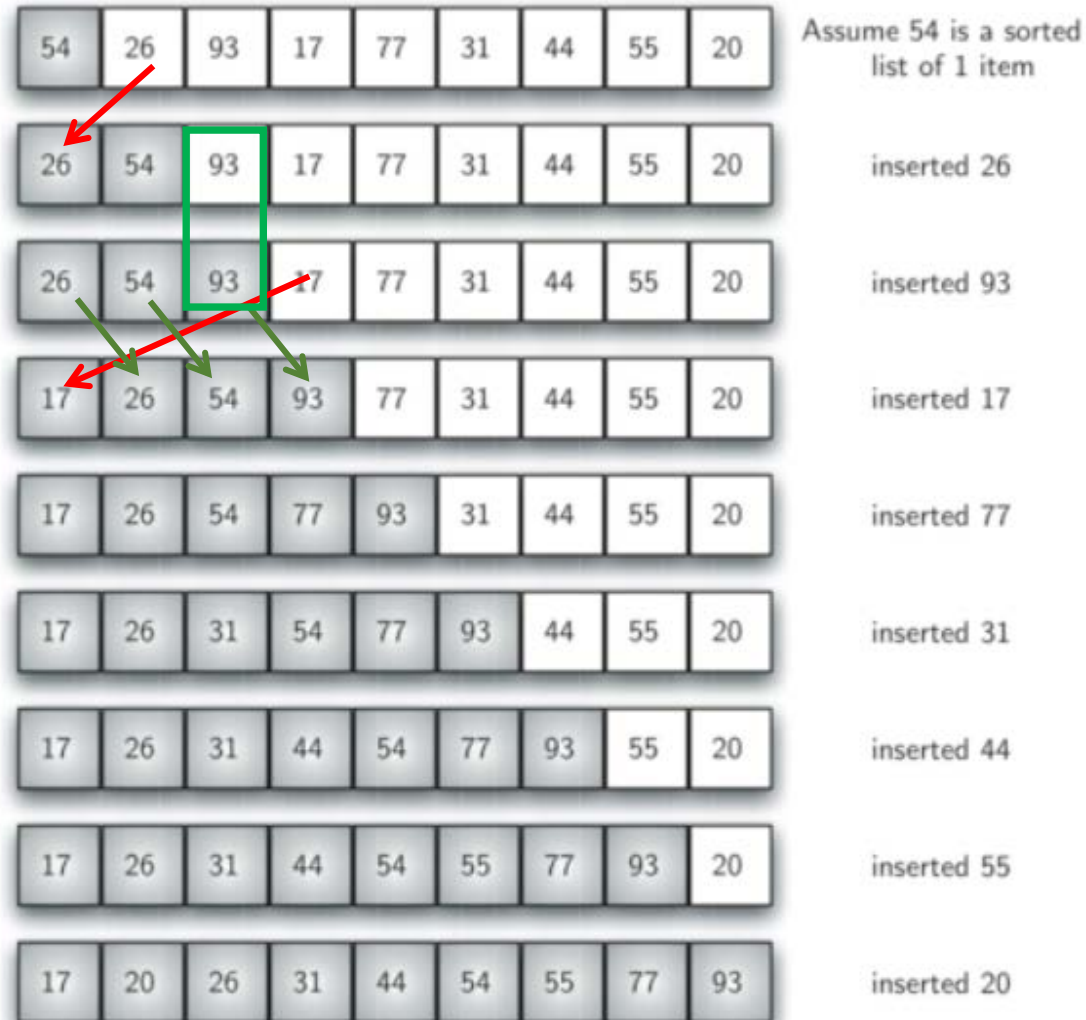
        i = i + 1

    return smallest

# doctest with verbose = True
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Insertion Sort

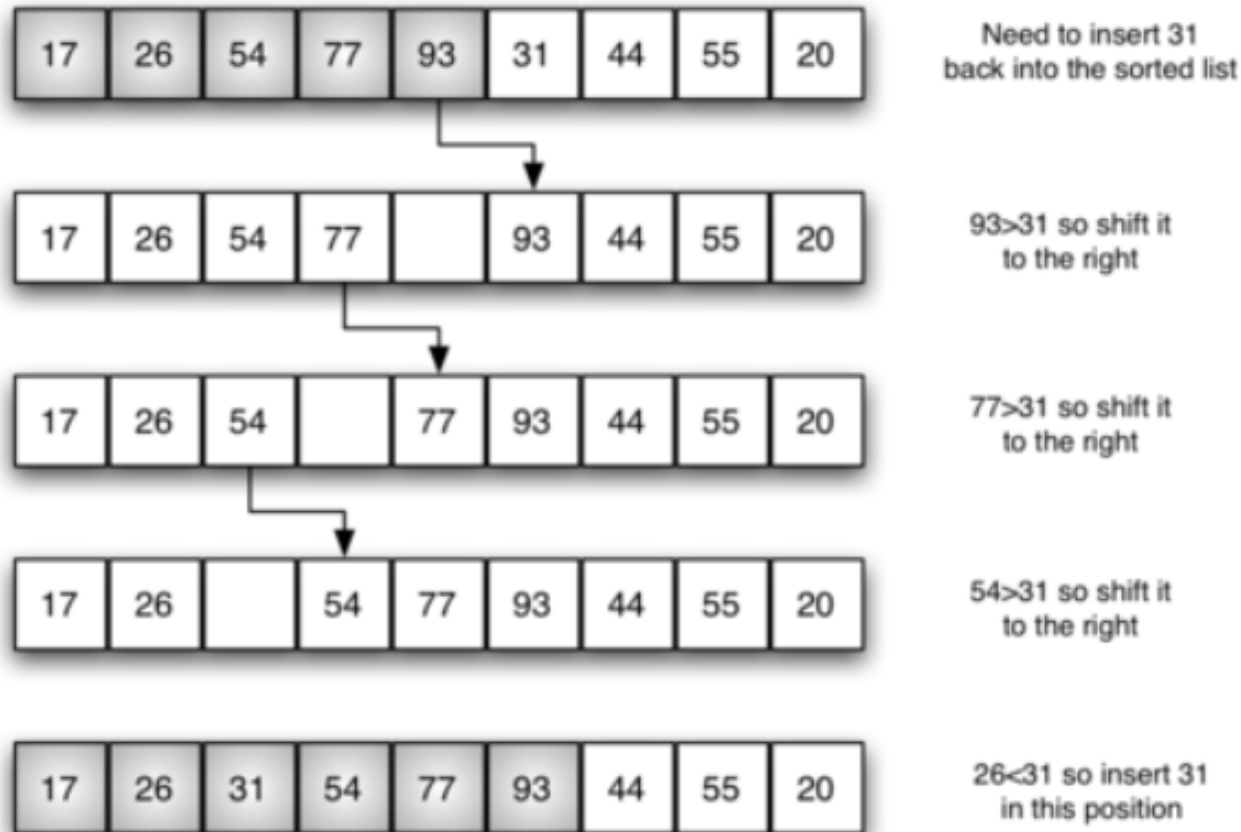
Go through every item, building a sorted list from the beginning



Insertion Sort

Go through every item, building a sorted list from the beginning
Fifth step: fewer moves/switches than selection sort

The first two do not
need to be moved



Insertion Sort

Go through every item, building a sorted list from the beginning
it does n^2 comparisons (looks at every one (n , $n-1$, $n-2$, etc.) to find the max)
- but it performs fewer exchanges than selection sort

The screenshot shows a web browser window with multiple tabs. The active tab is titled "5.9. The Insertion Sort — Pr..." and the address bar shows the URL <https://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html#lst-insertion>. The page content includes a header "Problem Solving with Algorithms and Data Structures" and a sub-header "Chapters". Below this, there is a text input field containing "16" and a label "ActiveCode: 1 Insertion Sort (lst_insertion)". The main area displays a bar chart with 16 vertical bars of varying heights, representing an array. A blue rectangular box highlights the first 10 bars of the chart. Below the chart, there are several control buttons: "Initialize", "Run", "Stop", "Beginning", "Step Forward", "Step Backward", and "End". At the bottom, there is a "Self Check" section with a text input field. The footer of the page contains a copyright notice: "Scripts Partially Allowed, 4/6 (google-analytics.com, interactivepython.org, kaspersky-labs.com, mathjax.org) | <SCRIPT>: 79 | <OBJECT>: 0" and an "Options..." button.

Insertion Sort

```
# add to sort.py
def insert(L,b):
    """ (list, int) -> NoneType
    Precondition: L[0:b] is already sorted.
    Insert L[b] where it belongs in L[0:b + 1].
    [-1, 2, 3, 4, 7, 5]
    """
    # Find where to insert L[b] by searching backwards from L[b]
    # for a smaller item.
    i = b
    while i != 0 and L[i - 1] >= L[b]:
        i = i - 1

    # Move L[b] to index i, shifting the following values to the right.
    value = L[b]
    del L[b]
    L.insert(i, value)

def insertion_sort(L):
    i = 0
    while i != len(L):
        insert(L, i)
        i = i + 1
```

Insertion Sort

add to sort.py

```
def insert(L,b):
    """ (list, int) -> NoneType
    Precondition: L[0:b] is already sorted.
    Insert L[b] where it belongs in L[0:b + 1].
    >>> L = [3, 4, -1, 7, 2, 5]
    >>> insert(L, 2)
    >>> L
    [-1, 3, 4, 7, 2, 5]
    >>> insert(L, 4)
    >>> L
    [-1, 2, 3, 4, 7, 5]
    """
    # Find where to insert L[b] by searching backwards from L[b]
    # for a smaller item.
    i = b
    while i != 0 and L[i - 1] >= L[b]:
        i = i - 1

    # Move L[b] to index i, shifting the following values to the right.
    value = L[b]
    del L[b]
    L.insert(i, value)

def insertion_sort(L):
    """ (list) -> NoneType
    Reorder the items in L from smallest to largest.
    >>> L = [3, 4, 7, -1, 2, 5]
    >>> insertion_sort(L)
    >>> L
    [-1, 2, 3, 4, 5, 7]
    >>> L = []
    >>> insertion_sort(L)
    >>> L
    []
    >>> L = [1]
    >>> insertion_sort(L)
    >>> L
    [1]
    >>> L = [2, 1]
    >>> insertion_sort(L)
    >>> L
    [1, 2]
    >>> L = [1, 2]
    >>> insertion_sort(L)
    >>> L
    [1, 2]
    >>> L = [3, 3, 3]
    >>> insertion_sort(L)
    >>> L
    [3, 3, 3]
    >>> L = [-5, 3, 0, 3, -6, 2, 1, 1]
    >>> insertion_sort(L)
    >>> L
    [-6, -5, 0, 1, 1, 2, 3, 3]
    """
    i = 0
    while i != len(L):
        insert(L, i)
        i = i + 1

# doctest with verbose = True
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Binary Sort

Phonebook - should be $N (\log_2 N) \log_2 N = N_2?$ (N base 2))

N values have to be inserted, so actually $N(N + N_2) \sim N^2$

add to sort.py

Mergesort

Along with quicksort and heapsort, divide and conquer algorithms
merge sorted lists, should be $N \log_2 N$
Compare a pair of sorted lists and add to another after comparing values
(Compare lists: L1 and L2)

Merge sorted lists L1 and L2 into a new list and return that new list.

```
>>> merge([1, 3, 4, 6], [1, 2, 5, 7])
```

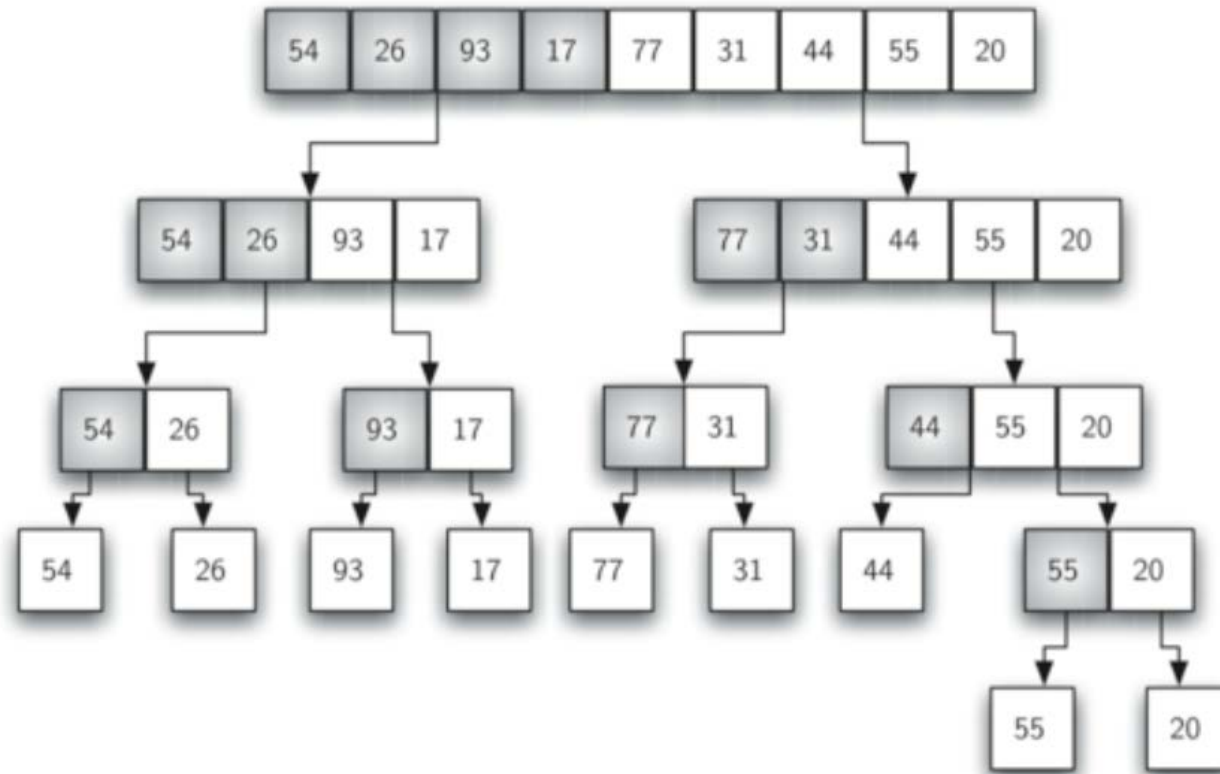
```
[1, 1, 2, 3, 4, 5, 6, 7]
```

Mergesort

Along with quicksort and heapsort, divide and conquer algorithms merge sorted lists recursively, should be $N \log_2 N$

Compare a pair of sorted lists and add to another after comparing values (L1 or L2)

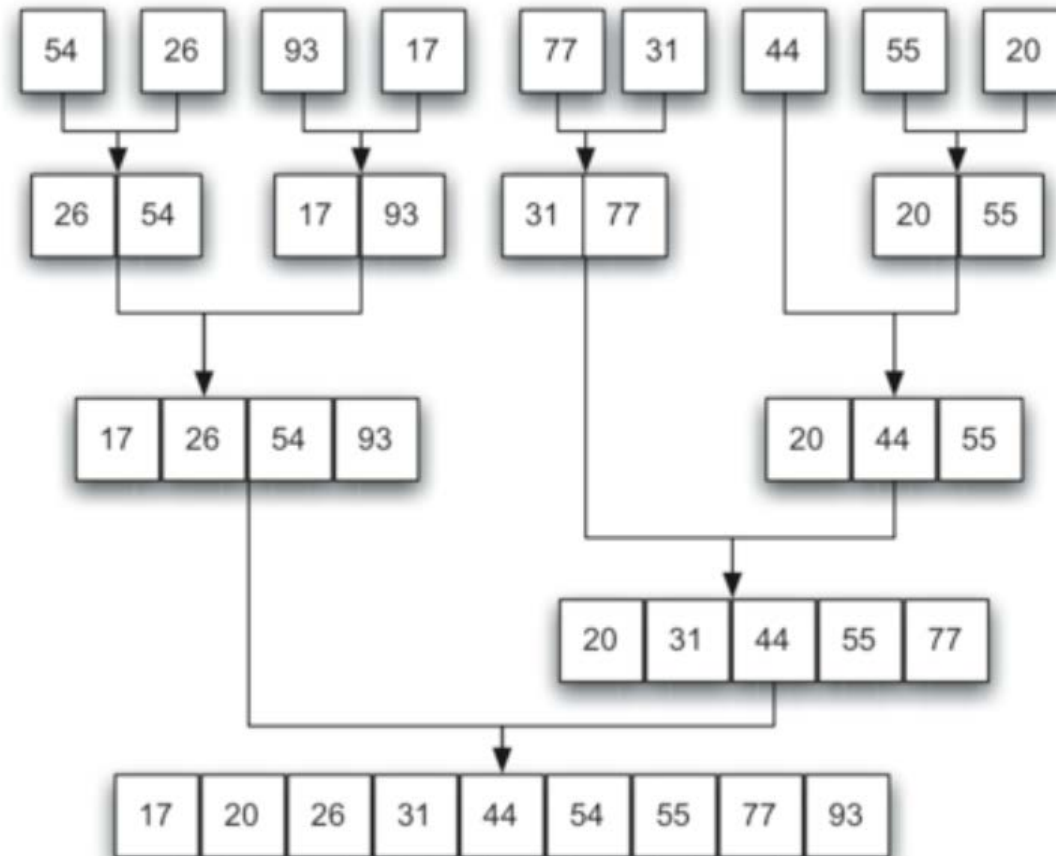
Divide first



Mergesort

Along with quicksort and heapsort, divide and conquer algorithms
merge sorted lists, should be $N \log_2 N$

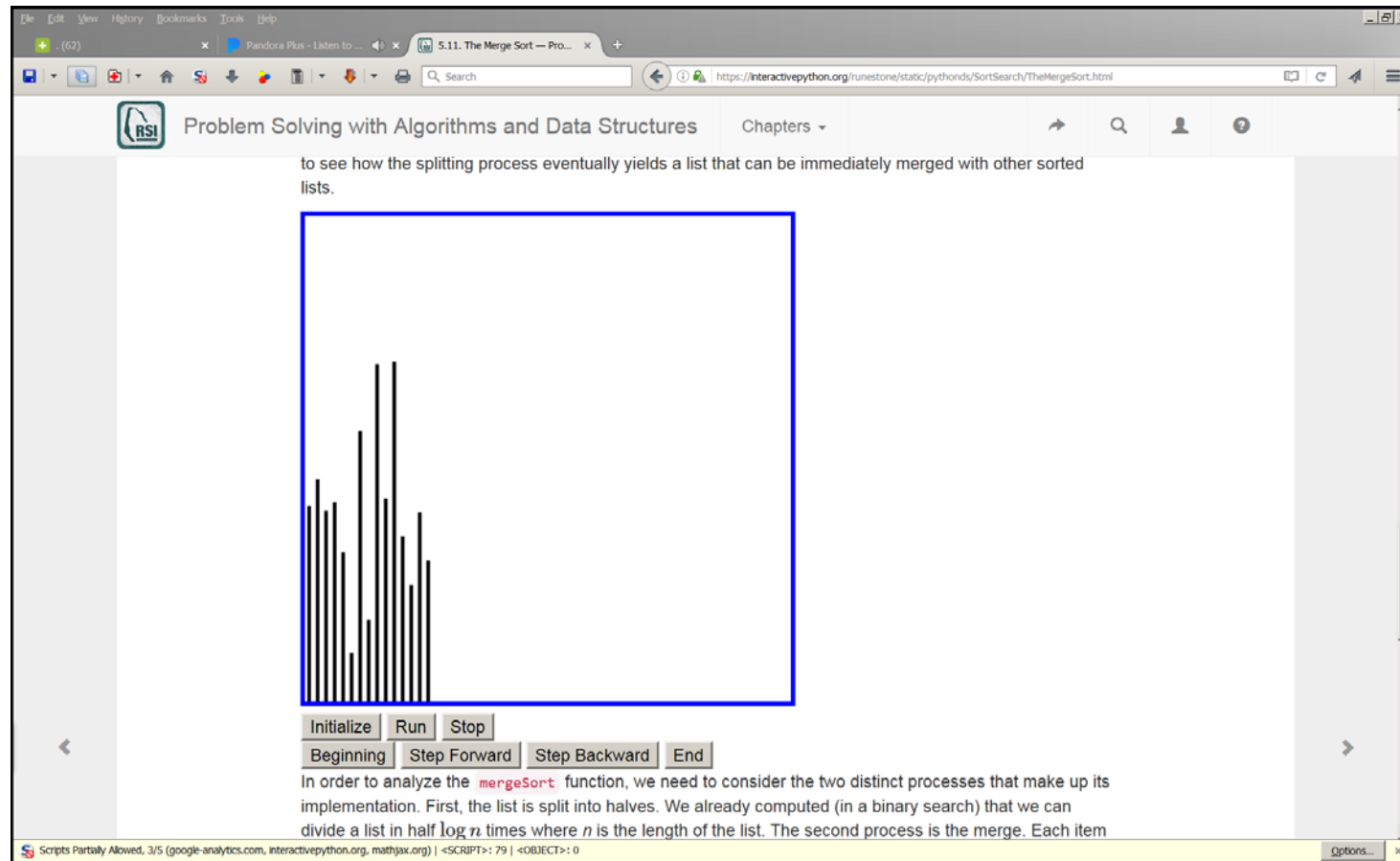
Compare a pair of sorted lists and add to another after comparing values (L1, L2)
Then merge



Mergesort

Along with quicksort and heapsort, divide and conquer algorithms merge sorted lists, should be $N \log_2 N$

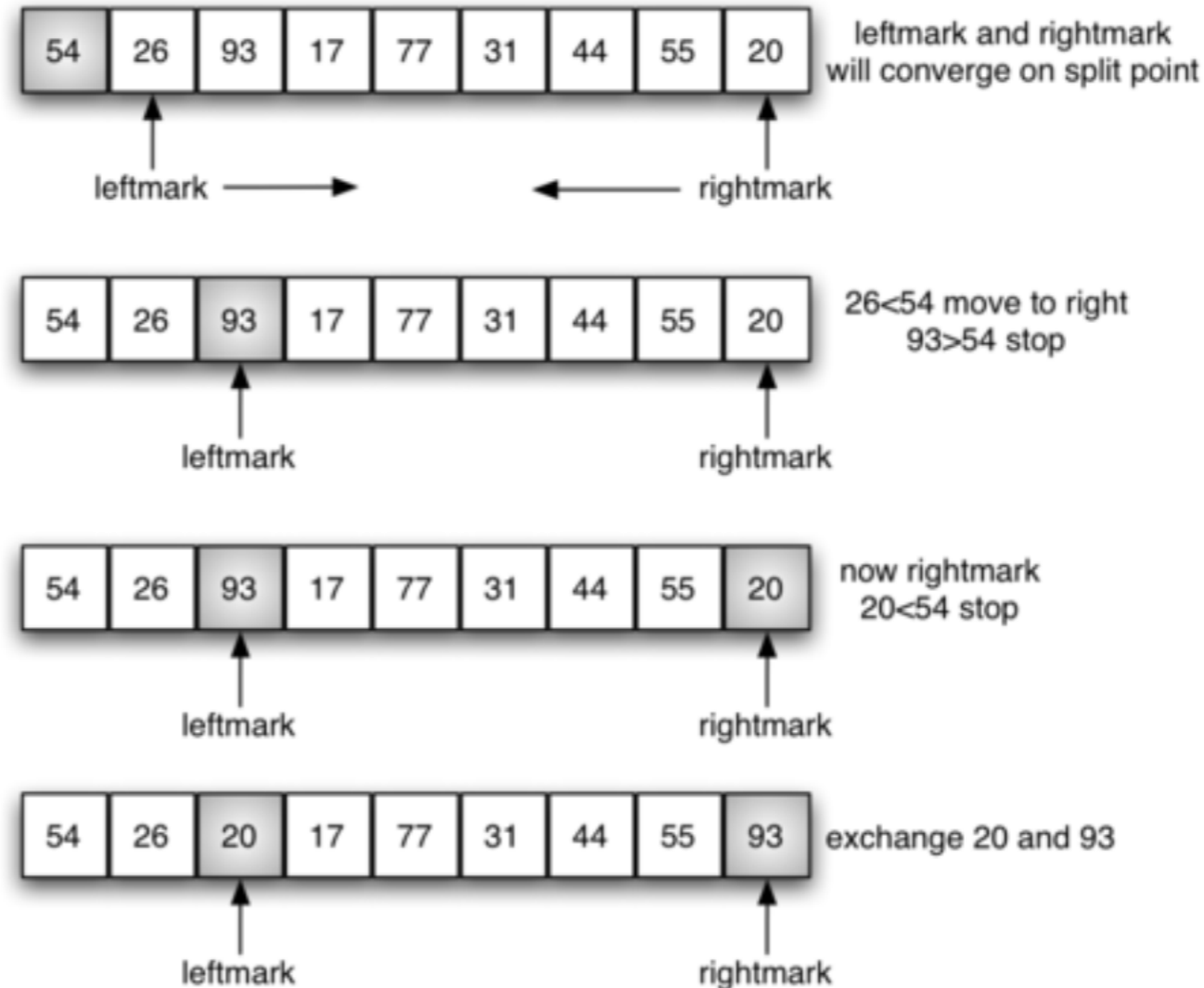
Compare a pair of sorted lists and add to another after comparing values (L1 or L2)



Quicksort

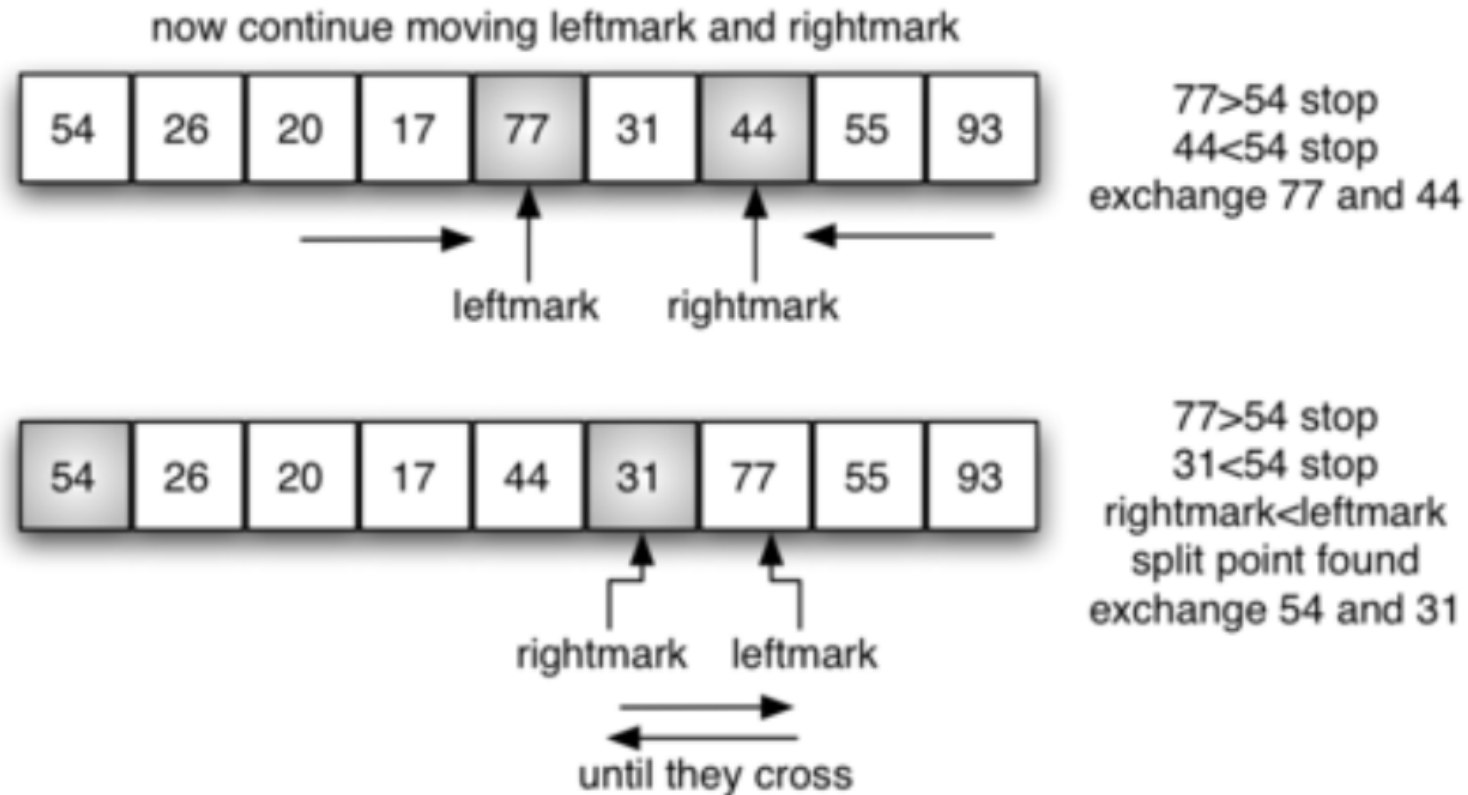
Along with mergesort and heapsort, a divide and conquer algorithm

Merge sorted lists, should be $N \log_2 N$
Exchange positions within a list based on a "pivot value"



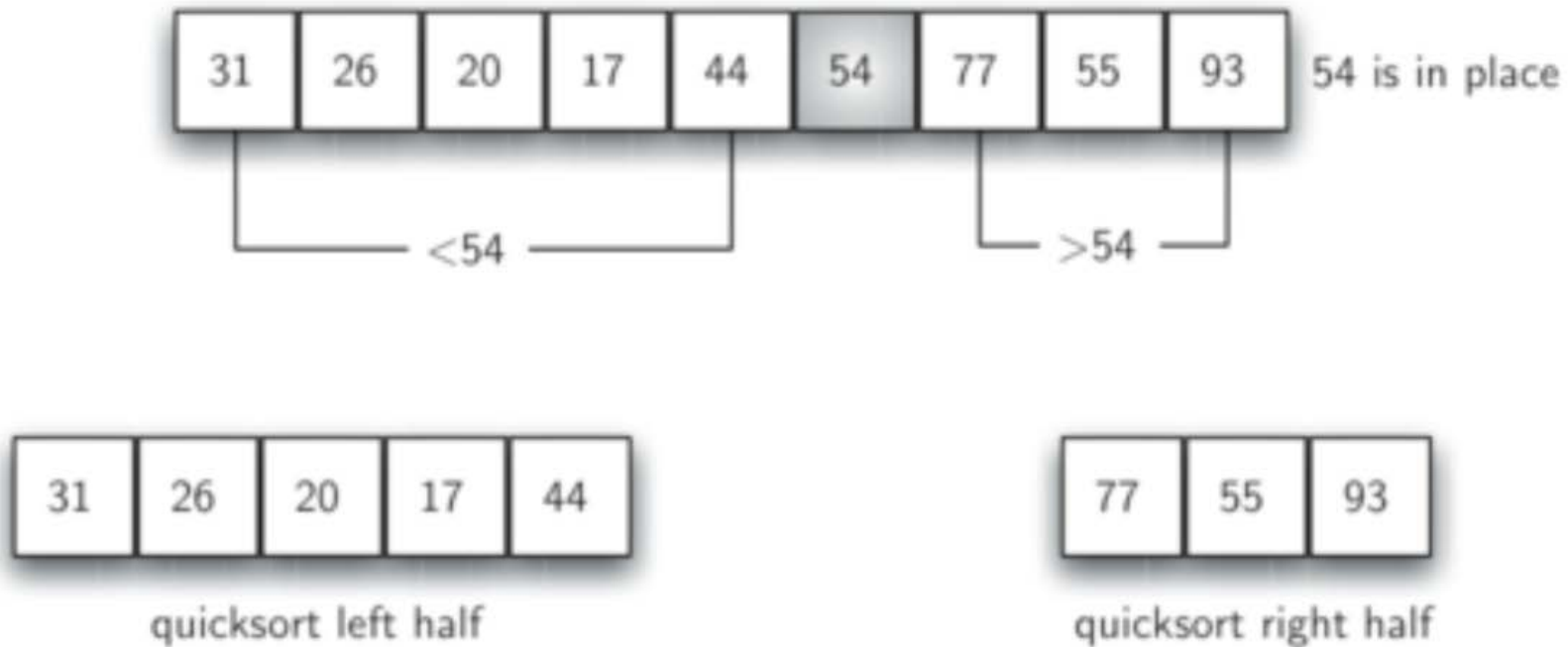
Quicksort

Keep exchanging positions within a list based on the "pivot value"



Quicksort

Keep exchanging positions within a list based on the "pivot value"
- then split into two lists and quicksorts them



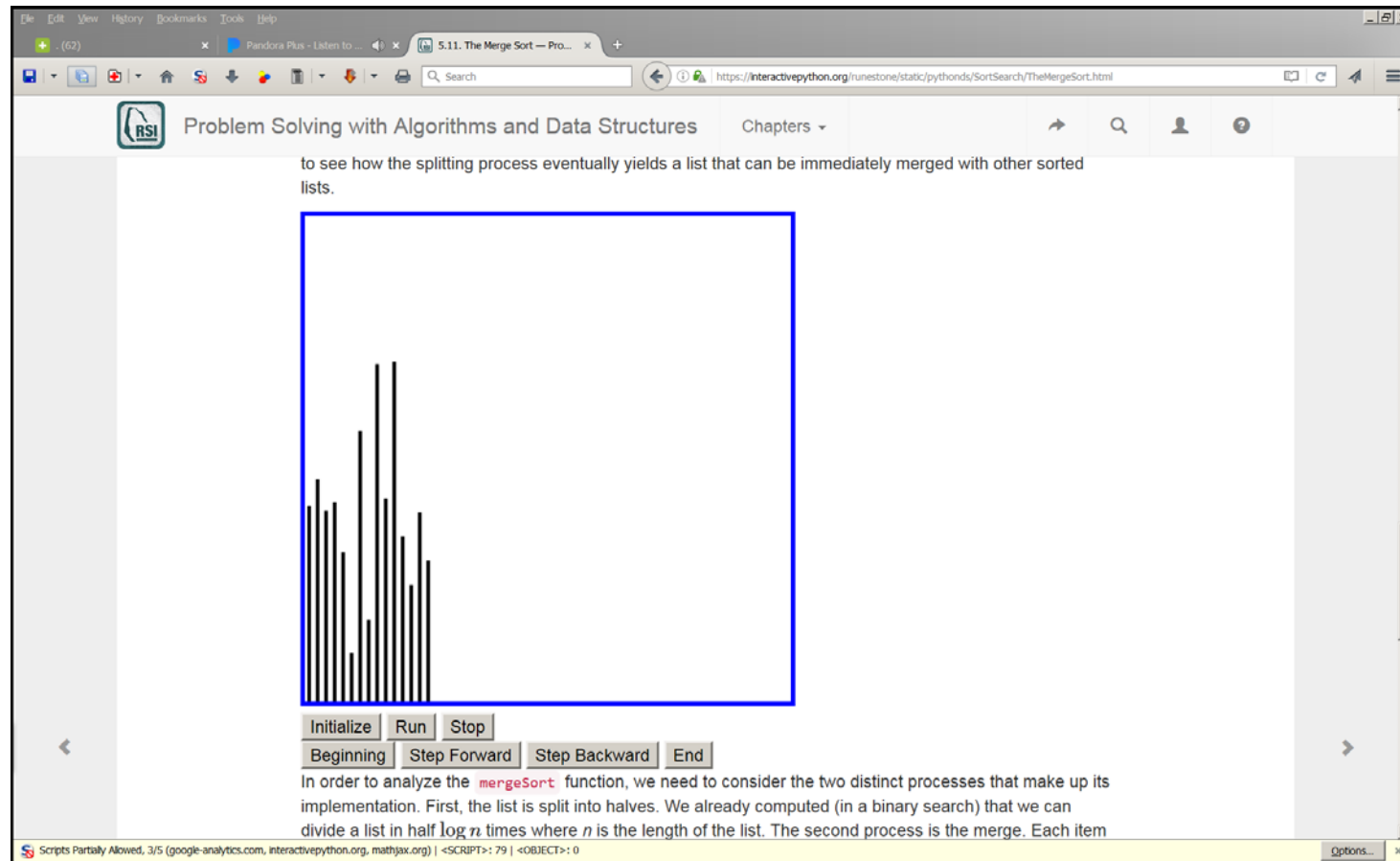
Quicksort

<https://interactivepython.org/runestone/static/pythonds/SortSearch/TheQuickSort.html>

Along with mergesort and heapsort, a divide and conquer algorithm

Merge sorted lists, should be $N \log_2 N$

Exchange positions within a list based on a "pivot value"



The screenshot shows a web browser window displaying the "Merge Sort" page on the interactivepython.org website. The page title is "Problem Solving with Algorithms and Data Structures" and the chapter is "5.11. The Merge Sort". The main content area features a diagram of a list of numbers represented by vertical bars of varying heights. A blue rectangular box highlights a portion of this list. Below the diagram, there are several interactive buttons: "Initialize", "Run", "Stop", "Beginning", "Step Forward", "Step Backward", and "End". The text below the buttons explains the merge sort process: "In order to analyze the `mergeSort` function, we need to consider the two distinct processes that make up its implementation. First, the list is split into halves. We already computed (in a binary search) that we can divide a list in half $\log n$ times where n is the length of the list. The second process is the merge. Each item

Quicksort

```
# quicksort
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
    if first<last:

        splitpoint = partition(alist,first,last)

        quickSortHelper(alist,first,splitpoint-1) # recursive
        quickSortHelper(alist,splitpoint+1,last)  # recursive

def partition(alist,first,last):
    pivotvalue = alist[first]

    leftmark = first+1
    rightmark = last

    done = False
    while not done:

        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1

        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark -1

        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp

    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp

    return rightmark

alist = [54,26,93,17,77,31,44,55,20] # list(range(900,0,-1))
quickSort(alist)
print(alist)
```

Homework 14

A. Gries 13.7 (page 266)

1, 3, 10 <<<< **problem with question 10, this is the correct text;**

10. In the function *merge* on page 261, there are two calls to `extend`. They are there because when the preceding loop ends, one of the two lists still has items in it that haven't been processed. Rewrite that loop so that these `extend` calls aren't needed.

AND:

B. Write a python program to sort a list by magnitude (absolute value). Ties are okay, but negatives should come before positives.

You can use any Python built-in methods (`find`, `min`, `max`, etc.)