Sorting and Searching

- **Sorting**
  - **Simple:** *Selection Sort* and *Insertion Sort*
  - **Efficient:** *Quick Sort* and *Merge Sort*
- **Searching**
  - Linear
  - Binary
- **Reading for this lecture:**
  http://introcs.cs.princeton.edu/python/42sort/

Sorting

- *Sorting* is the process of arranging a list of items in a particular order
- The sorting process is based on specific value(s)
  - Sorting a list of test scores in ascending numeric order
  - Sorting a list of people alphabetically by last name
- There are many algorithms, which vary in efficiency, for sorting a list of items
- We will examine <u>four</u> specific algorithms:

`Selection Sort`	**`Quicksort`**
`Insertion Sort`	**`Merge Sort`**

Selection Sort

- The approach of Selection Sort:
  – Select a value and put it in its final place in the list
  – Repeat for all other values
- In more detail:
  – Find the smallest value in the list
  – Switch it with the value in the first position
  – Find the next smallest value in the list
  – Switch it with the value in the second position
  – Repeat until all values are in their proper places

Selection Sort

- An example:

```
original:          3   9   6   1   2
smallest is 1:     1   9   6   3   2
smallest is 2:     1   2   6   3   9
smallest is 3:     1   2   3   6   9
smallest is 6:     1   2   3   6   9
```

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Selection Sort

- Algorithm:

```
def selection_sort (in_list):
    for index in range(len(in_list)-1):
        min = index
        for scan in range(len(in_list)):
            if in_list[scan] < in_list[min]:
                min = scan
        temp = in_list[min]
        in_list[min] = in_list[index]
        in_list[index] = temp
```

Swapping Two Values

- The processing of the selection sort algorithm includes the *swapping* of two values
- Swapping requires three assignment statements and a temporary storage location of the same type as the data being swapped:

```
first = 35
second = 53
temp = first
first = second   # 53 now
second = temp # 35 now
```

Polymorphism in Sorting

- Recall that a class can have comparison functions that establish the relative order of its objects
- We can use polymorphism to develop a generic sort for any list of comparable objects
- The list can sort itself using its `sort` function
- That way, one method can be used to sort a group of `Person` objects, `Book` objects, or whatever -- as long as the class implements the appropriate comparison functions for that type

Polymorphism in Sorting

- The sorting method doesn't "care" what type of object it is sorting, it just needs to be <u>able to compare</u> it to other objects in the list
- That is guaranteed by putting in the appropriate comparison functions so that the sorting method can compare the individual objects to one another – where they are *mutually comparable*
- We can define these functions for a class in order to determine what it means for one object of that class to be “less than another” – or "equal to", "greater than", etc.

Insertion Sort

- The approach of Insertion Sort:
  – Pick any item and insert it into its proper place in a sorted sublist
  – Repeat until all items have been inserted
- In more detail:
  – Consider the first item to be a sorted sublist (of one item)
  – Insert the second item into the sublist, shifting the first item as needed to make room to insert the new addition
  – Insert the third item into the sublist (of two items), shifting items as necessary
  – Repeat until all values are in their proper positions

Insertion Sort

- An example:

original:	**3**	9	6	1	2
insert 9:	**3**	**9**	6	1	2
insert 6:	**3**	**6**	**9**	1	2
insert 1:	**1**	**3**	**6**	**9**	2
insert 2:	**1**	**2**	**3**	**6**	**9**

Insertion Sort

- Algorithm:

```
def insertion_sort (in_list):
    for index in range(1, len(in_list)):
        key = in_list[index]
        position = index

        # Shift larger values to the right
        while position > 0 and key < in_list[position-1]:
            in_list[position] = in_list[position-1]
            position -= 1

        in_list[position] = key
```

Comparing Sorts

- The Selection and Insertion sort algorithms are similar in efficiency
- They both have *outer* loops that scan all elements, and *inner* loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts are of *order* n^2
- Other sorts are more efficient: *order* $n \log_2 n$

Quicksort

- The approach of Quicksort:
  - Reorganize the list into two partitions
  - Recursively call Quicksort on each partition
- In more detail:
  - Choose a "pivot" value from somewhere in the list
  - Move values in the list so all elements smaller than the pivot come before it, and all elements larger than the pivot come after it
  - Make recursive calls to Quicksort for the both partitions
  - Keep doing this so long as partitions are of length > 1

Quicksort

- Main algorithm:

```
def quicksort (in_list, start, end):

    if start < end:
        # partition the list around a pivot
        p = partition (in_list, start, end)

        # sort the items less than the pivot
        quicksort (in_list, start, p-1)

        # sort the items greater than the pivot
        quicksort (in_list, p+1, end)
```

Quicksort

```
def partition (in_list, start, end):
    pivot = in_list[end]
    i = start
    for j in range (start, end):
        if in_list[j] <= pivot:
            temp = in_list[i]
            in_list[i] = in_list[j]
            in_list[j] = temp
            i += 1
    temp = in_list[i]
    in_list[i] = in_list[end]
    in_list[end] = temp
    return i
```

Merge Sort

- The approach of Merge Sort:
  - Divide the list into two halves
  - Sort each half, and then merge the two back together
- In more detail:
  - So long as the input list has more than one item...
    - Divide the list into (roughly) equal halves
    - Call Merge Sort recursively on each half
    - Merge the two (sorted) halves into a single sorted list of items
  - A list of length 1 is considered "sorted" so it is returned with no need for further recursive calls.

Merge Sort

- Algorithm:

```
def merge_sort (in_list):
    # Trivial : it is considered "sorted"
    if len(in_list) <= 1:
        return in_list

    # Sort each half of the list
    first_half = merge_sort (in_list[:len(in_list)//2])
    second_half = merge_sort (in_list[len(in_list)//2:])

    # Merge the two sorted halves
    return merge (first_half, second_half)
```

Merge Sort

```
def merge (first, second):

    result = []

    while first and second:
        if first[0] < second[0]:
            result.append(first.pop(0))
        else:
            result.append(second.pop(0))

    return result + max(first, second)
```

Comparing Sorts

- The Quicksort and Merge Sort algorithms are similar in efficiency
- They both divide the list into two components and then recursively call themselves on each component.
- In the best case, approximately $n \log_2 n$ number of comparisons are made to sort a list of size n
- Therefore, we say these sorts are *order* $\mathbf{n \log_2 n}$
- Although there are exception, these sorts are considered much more efficient than *order* n^2

Searching

- Searching is the process of finding a target element within a group of items called the search pool
- The target may or may not be in the search pool
- We want to perform the search efficiently, minimizing the number of comparisons
- Let's look at two classic searching approaches: linear search and binary search
- As we did with sorting, we'll implement the searches with polymorphic comparability

Linear Search

- A linear search begins at one end of a list and examines each element in turn
- Eventually, either the item is found or the end of the list is encountered
- See the **`linear_search`** method in **search_code.py**
- At worst, you may examine *every single item* in the list!

Linear Search

- Algorithm:

```
def linear_search (in_list, target):
    for item in in_list:
        if item == target:
            return item
    return None
```

Binary Search

- A *binary search* **assumes** the list of items in the search pool is sorted
- It eliminates a large part of the search pool with a single comparison
- A binary search first examines the middle element -- if it matches the target, the search is over
- If it doesn't, only **half** of the remaining elements need be searched
- Since they are sorted, the target can only be in one half of the other

Binary Search

- The process continues by recursively searching one – and only one – half of the list
- Each comparison eliminates approximately half of the remaining data
- Eventually, the target is found or there are no remaining *viable candidates* (and the target has not been found)
- At most, there will be $\boldsymbol{log_2\ n}$ recursive calls
- See the `binary_search` method in **search_code.py**

Binary Search

- Algorithm:

```
def binary_search (in_list, target):
  if len (in_list) < 1:
    return None
  else:
    mid = len(in_list) // 2
    if in_list[mid] == target:
      return in_list[mid]
    elif in_list[mid] > target:
      return binary_search(in_list[:mid], target)
    else:
      return binary_search(in_list[mid:], target)
```

Binary Versus Linear Search

- The efficiency of binary search is good for the retrieval of data from a sorted group
- However, the group must be sorted initially, and as items are added to the group, it must be ***kept*** in sorted order
- The repeated sorting creates inefficiency
- If you add data to a group much more often than you search it, it may actually be **<u>worse</u>** to use binary searches rather than linear