CSc 110, Autumn 2017

Lecture 37: searching and sorting
Adapted from slides by Marty Stepp and Stuart Reges
Using `binary_search`

```
# index 0  1  2  3   4   5   6   7   8   9  10  11  12  13  14  15
a = [-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92]

index1 = binary_search(a, 42)
index2 = binary_search(a, 21)
index3 = binary_search(a, 17, 0, 16)
index2 = binary_search(a, 42, 0, 10)
```

- `binary_search` returns the index of the number
  or
- (index where the value should be inserted + 1)
Write the following two functions:

```python
# searches an entire sorted list for a given value
# returns the index the value should be inserted at to maintain sorted order
# Precondition: list is sorted
binary_search(list, value)

# searches given portion of a sorted list for a given value
# examines min_index (inclusive) through max_index (exclusive)
# returns the index of the value or -(index it should be inserted at + 1)
# Precondition: list is sorted
binary_search(list, value, min_index, max_index)
```
Binary search code

# Returns the index of an occurrence of target in a, # or a negative number if the target is not found. # Precondition: elements of a are in sorted order
def binary_search(a, target, start, stop):
    min = start
    max = stop - 1

    while min <= max:
        mid = (min + max) // 2
        if a[mid] < target:
            min = mid + 1
        elif a[mid] > target:
            max = mid - 1
        else:
            return mid  # target found

    return -(min + 1)  # target not found
Sorting

- **sorting**: Arranging the values in a list into a specific order (usually into their "natural ordering").
  - one of the fundamental problems in computer science
  - can be solved in many ways:
    - there are many sorting algorithms
    - some are faster/slower than others
    - some use more/less memory than others
    - some work better with specific kinds of data
    - some can utilize multiple computers / processors, ...

- **comparison-based sorting**: Determining order by comparing pairs of elements:
  - <, >, ...
Bogo sort

• **bogo sort**: Orders a list of values by repetitively shuffling them and checking if they are sorted.
  • name comes from the word "bogus"

  The algorithm:
  • Scan the list, seeing if it is sorted. If so, stop.
  • Else, shuffle the values in the list and repeat.

• This sorting algorithm (obviously) has terrible performance!
# Places the elements of `a` into sorted order.
def bogo_sort(a):
    while (not is_sorted(a)):
        shuffle(a)

# Returns true if `a`'s elements
# are in sorted order.
def is_sorted(a):
    for i in range(0, len(a) - 1):
        if (a[i] > a[i + 1]):
            return False
    return True

# Swaps `a[i]` with `a[j]`.
def swap(a, i, j):
    if (i != j):
        temp = a[i]
        a[i] = a[j]
        a[j] = temp

# Shuffles a list by randomly swapping each
# element with an element ahead of it in the list.
def shuffle(a):
    for i in range(0, len(a) - 1):
        # pick a random index in [i+1, a.length-1]
grange = len(a) - 1 - (i + 1) + 1
        j = (random() * range + (i + 1))
        swap(a, i, j)
Selection sort

- **selection sort**: Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

  The algorithm:
  - Look through the list to find the smallest value.
  - Swap it so that it is at index 0.
  - Look through the list to find the second-smallest value.
  - Swap it so that it is at index 1.
  ...
  - Repeat until all values are in their proper places.
**Selection sort example**

- **Initial list:**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | 22| 18| 12|-4| 27| 30| 36| 50| 7 | 68| 91 | 56  | 2  | 85 | 42 | 98 | 25 |

- **After 1st, 2nd, and 3rd passes:**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| 18| 12| 22| 27| 30| 36| 50| 7 | 68| 91 | 56  | 2  | 85 | 42 | 98 | 25 |

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| 2 | 12| 22| 27| 30| 36| 50| 7 | 68| 91 | 56  | 18 | 85 | 42 | 98 | 25 |

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| 2 | 7 | 22| 27| 30| 36| 50| 12| 68| 91 | 56  | 18 | 85 | 42 | 98 | 25 |
Selection sort code

# Rearranges the elements of a into sorted order using
# the selection sort algorithm.
def selection_sort(a):
    for i in range(0, len(a) - 1):
        # find index of smallest remaining value
        min = i
        for j in range(i + 1, len(a)):
            if (a[j] < a[min]):
                min = j
        # swap smallest value its proper place, a[i]
        swap(a, i, min)
Selection sort runtime (Fig. 13.6)

- How many comparisons does selection sort have to do?

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>16</td>
</tr>
<tr>
<td>4000</td>
<td>47</td>
</tr>
<tr>
<td>8000</td>
<td>234</td>
</tr>
<tr>
<td>16000</td>
<td>657</td>
</tr>
<tr>
<td>32000</td>
<td>2562</td>
</tr>
<tr>
<td>64000</td>
<td>10265</td>
</tr>
<tr>
<td>128000</td>
<td>41141</td>
</tr>
<tr>
<td>256000</td>
<td>164985</td>
</tr>
</tbody>
</table>

![Input size (N) vs. Runtime (ms) graph](image-url)
Similar algorithms

- **bubble sort**: Make repeated passes, swapping adjacent values
  - slower than selection sort (has to do more swaps)

- **insertion sort**: Shift each element into a sorted sub-list
  - faster than selection sort (examines fewer values)