Introduction to Programming

Spring 2022

Algorithm Design and Recursion

- Searching
- Recursive Problem Solving
- **Sorting Algorithms**
- Hard Problems

Sorting Algorithms

- The basic sorting problem is to take a list and rearrange it so that the values are in
  - increasing (or ascending) order
  - decreasing (or descending) order

Selection Sort

- To start out, pretend you’re the computer, and you’re given a shuffled stack of index cards, each with a number.
  - How would you put the cards back in order?

Selection Sort

- One simple method is to look through the deck to find the smallest value and place that value at the front of the stack.
- Then go through, find the next smallest number in the remaining cards, place it behind the smallest card at the front.
- Repeat, until the stack is in sorted order!
- Example

Selection Sort

- We already have an algorithm to find the smallest item in a list.
  - As you go through the list, keep track of the smallest one seen so far
  - updating it when you find a smaller one.
- Now place that smallest number at the front of the list.
- This sorting algorithm is known as a selection sort.

Selection Sort

- The algorithm has a loop, and each time through the loop the smallest remaining element is selected and moved into its proper position.
  - For n elements, we find the smallest value and put it in the 0th position.
  - Then we find the smallest remaining value from position 1 – (n-1) and put it into position 1.
  - The smallest value from position 2 – (n-1) goes in position 2.
  - Etc.

Selection Sort

- When we place a value into its proper position, we need to be sure we don't accidentally lose the value originally stored in that position.
  - If the smallest item is in position 10, moving it into position 0 involves the assignment:
    `nums[0] = nums[10]`
  - This wipes out the original value in `nums[0]`!

Selection Sort

- We can use simultaneous assignment to swap the values between `nums[0] and nums[10]`:

```
nums[0],nums[10] = nums[10],nums[0]
```

- Using these ideas, we can implement our algorithm, using variable `bottom` for the currently filled position, and `mp` is the location of the smallest remaining value.

Naive Sorting: Selection Sort

```
def selSort(nums):
 # sort nums into ascending order

 n = len(nums)

 # For each position in the list (except the very last)

 for bottom in range(n-1):
     # find the smallest item in nums[bottom]..nums[n-1]

     mp = bottom                         # bottom is smallest initially
     for i in range(bottom+1, n):        # look at each position
         if nums[i] < nums[mp]:          # this one is smaller
             mp = i                      # remember its index

     # swap smallest item to the bottom
     nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

Selection Sort

- Rather than remembering the minimum value scanned so far, we store its position in the list in the variable `mp`.
- New values are tested by comparing the item in position `i` with the item in position mp.
- `bottom` stops at the second to last item in the list. Why? Once all items up to the last are in order, the last item must be the largest!

Selection Sort

- The selection sort is easy to write and works well for moderate-sized lists, but is not terribly efficient. We’ll analyze this algorithm in a little bit.