

Searching and Sorting

April 29, 2020

Administrative Notes

A word about Lab 10

Project 3

Final Exam and Grading

The final is Friday, May 15 from 6 - 8 pm. That's the CMSC 201 common final time.

- It will be given on Blackboard, just like Exam 2.
 - That seemed to work okay
 - Seems to cut down on cheating
 - Yes, there will be a practice exam available no later than May 8.
- If you need accommodations, please contact Student Disability Services so they can work with me to make alternate arrangements
- You will get your grade no later than May 27 - I hope long before that
- AFTER you see your letter grade, you have the option to take that grade or switch to a PASS/FAIL

Virtual Environments

venv - see <https://docs.python-guide.org/dev/virtualenvs/> and <https://docs.python.org/3/library/venv.html>

Conceptually, you write a program to run directly on a computer's hardware. You compute on its CPU chip; you communicate over its network interfaces; you read from and write to its RAM and hard drives/SSDs

There are a lot of problems with that:

- Security: exposes everything on the computer to your code
- Safety: even if it's not malware, your bugs can mess up the system
- Inefficiency: makes it very hard to keep multiple threads or programs running at the same time

So you create a “virtual environment”

Sort of a sandbox; a limited world in which you operate by yourself and can't harm anybody else

- But it sure looks like a real computer to your code

Real-world examples:

- Virtual machines: VMWare; Microsoft Hyper-V
<https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/>
- Containers: Docker; Kubernetes; OpenShift

Tradeoffs:

- efficiency (more code running; fewer resources available) vs. safety/security

Lab 10

So the purpose of Lab 10 is to give you a little exposure to and practice with creating and using virtual environments on a shared server

- Because in your future careers, you'll likely be doing a lot of that

The other part of Lab 10: How many links on www.umbc.edu? Let's walk through it.

Now, let's talk about sorting lists of elements

All of our examples tonight use integers, but it all works the same way. As long as all elements in the list are of the same type, you can sort them into an order.

An easy one to understand - bubble sort.

Suppose you have a list of integers:

[4, -2, 19, 944, 27, 3]

You can go through the list and compare each pair of numbers. If the first one is larger than the second one, swap them.

When you have gone through the list one time, you will have “bubbled” the largest

Element up to the end of the list

An example - first, on the slide; then some coding

Original list :[4, -2, 19, 944, 27, 3]

Bubble sort - “bubble” the largest number left each time to the end of the list

-2, 4, 19, 944, 27, 3

-2, 4, 19, 27, 944, 3

-2, 4, 19, 27, 3, 944

-2, 4, 19, 3, 27, 944

We can write bubble sort as an iterative function, or as a recursive one

```
def iterative_bubble_sort (numbers):  
    # go through the entire list of numbers  
    for i in range(len(numbers)-1,0, -1):  
        for j in range(i):
```

temp

```
        if (numbers[j]>numbers[j+1]):  
            #swap the numbers - we'll use a  
            temp = numbers[j]  
            numbers[j] = numbers [j+1]  
            numbers[j+1] = temp  
        return(numbers)
```

Recursive Bubble Sort

```
def recursive_bubble_sort (numbers):  
    if len(numbers) == 1:  
        return(numbers)  
    else:  
        #bubble the largest number to  
the end  
        for j in range(len(numbers)-1):  
            if numbers[j] > numbers[j+1]:  
                temp = numbers[j]  
                numbers[j] = numbers[j+1]  
                numbers[j+1] = temp
```

*#then recursively call the function
with*

*#all but the last element of the
list*

```
        new_nums =  
        recursive_bubble_sort(numbers[:-1])  
        #add the last element back on  
        sorted_list = new_nums +  
        numbers[-1:]  
        return(sorted_list)
```

Stopping the sort

The list is sorted if there are no swaps made during a pass through the list

- Think about why

The previous code continues through even after the list is sorted.

We can be more efficient by converting the outer “for” loop into a “while” loop and using a boolean flag

- Let's look at the code

Selection Sort

Now, a different type of sorting.

This time, we're going to search through the entire list to find the smallest element, and then swap it with the first element in the list.

We then go through the rest of the list and select the smallest remaining element. We put this into the next available slot, and repeat until the list is supported

We “select” the smallest element from the list, and thus this called a
...selection sort

Selection sort, on paper and as an iterative function

Original list :[4, -2, 19, 944, 27, 3]

-2, 4, 19, 944, 27, 3

-2, 3, 19, 944, 27, 4

-2, 3, 4, 944, 27, 19

-2, 3, 4, 19, 27, 944

```
def iterative_selection_sort(numbers):  
    for i in range(len(numbers)):
```

```
        #find the smallest element remaining in the  
        #unsorted list. Start by presuming it's the  
        #first element
```

```
        smallest = i
```

```
        for j in range(i + 1, len(numbers)):  
            if numbers[smallest] > numbers[j]:  
                smallest = j
```

```
        # When the loop is done, we know that smallest is  
        # the index of the smallest value. Swap it  
        # with the first element
```

```
        temp = numbers[smallest]  
        numbers[smallest] = numbers[i]  
        numbers[i] = temp
```

Can we implement *this* as a recursive function?

```
def recursive_selection_sort(nums):  
    #The base case is: a list of length 0 or 1 is sorted  
    if len(nums) <= 1:  
        return nums  
    else:  
        #the core of the algorithm is the same  
        #as the iterative routine  
        index_of_smallest = 0  
        for j in range(1, len(nums)):  
            if nums[j] < nums[index_of_smallest]:  
                index_of_smallest = j
```

```
        # now swap the smallest element found with the  
        first  
        temp = nums[0]  
        nums[0] = nums[index_of_smallest]  
        nums[index_of_smallest] = temp  
        #now make the recursive call with the first  
        #element stripped out  
        r = recursive_selection_sort(nums[1:])  
        results = nums[:1] + r  
        return results
```

One more sorting algorithm: QuickSort

The idea here: pick an element in the list. Call this the “pivot”

Sort the list so that every item less than the pivot is before the pivot - “to the left of” the pivot, if you will

Every item greater than the pivot will be to the right - after the pivot in the list.

Note that there is no guarantee the items to the left and to the right of the pivot will be in any order at all.

So you have to recursively call the quicksort routine on the left side of the pivot, and then on the right.

Spoiler alert: this is called quicksort because it works faster than the other algorithms

A paper example before the code

Original list :[4, -2, 19, 944, 27, 3]

Less: -2, 3

Equal: 4

Greater: 19, 944, 27

Recursive call: Less; Greater

Quicksort code - this works best as a recursive function

```
def quicksort(list_of_nums):  
    #base case - a list of length one is sorted  
    if len(list_of_nums) <= 1:  
        return(list_of_nums)  
    #recursive case  
    else:  
        #pick a pivot - the first element  
        pivot = list_of_nums[0]
```

#define three empty lists, for elements greater than the pivot, less than the pivot, and equal to the pivot

```
less = []  
equal = []  
greater = []
```

Quicksort (continued)

*# go through the list and put each
element in the proper list*

```
for i in range(len(list_of_nums)):
```

```
    if list_of_nums[i] > pivot:
```

```
        greater.append(list_of_nums[i])
```

```
    elif list_of_nums[i] == pivot:
```

```
        equal.append(list_of_nums[i])
```

```
    else:
```

```
        less.append(list_of_nums[i])
```

```
        results = quicksort(less) +
```

```
        equal + quicksort(greater)
```

```
    return(results)
```

Searching - linear and binary

Linear search is what python uses as a default

Go through the list, one item at a time, in order from first element to last

Stop when you find the right element, or return failure if you never find it

It works, but it's not efficient

Binary Search

If the list is already sorted, we can search much more efficiently.

Go to the middle element of the list. List `[len(list)//2]`

Is this greater than the element we're searching for? If it is, just look at the first half of the original list. If not, just look at the right half. If this element is exactly what we're looking for, stop.

So now we have a recursive call to a list half the size of the original list

We'll look at the code in a minute, but we'll find the element we're looking for, or find it isn't there, in \log_2 of the length of the original list operations.

Code for binary search

```
def binary_search (numbers, target):  
    if len(numbers)//2 == 0:  
        return -1  
    if numbers[len(numbers)//2] == target:  
        return len(numbers)//2  
    elif numbers[len(numbers)//2] > target:  
        return binary_search(numbers[:len(numbers)//2], target)  
    else:  
        return binary_search(numbers[len(numbers)//2:], target)
```