Searching and Sorting, Part 2

November 18, 2024

Administrative notes

Project 2 is due tonight!!

Project 3 is out today

- It is due Monday, December 9
  - That’s the last day of class
  - Extensions will NOT be granted - we need to finish the class and get the grades in
- We’ll start talking about it today, and every lecture until the due date

Part 2 of our lecture

This code demonstration follows up on what we talked about last Wednesday

- First we'll describe the Quicksort algorithm
  - See the following slides

The file November_18_coding.py in the "code_samples" repo on the class GitHub repo contains the code I'll work from in this lecture

I have implemented bubble sort and selection sort using iterative versions of the algorithms. I will count:

- The number of comparisons required to sort a list
- The number of swaps required to sort a list
- The amount of time (in seconds) it took my computer to sort that list

I have implemented quicksort using a recursive version of the algorithm. I will count:

- The number of comparisons required to sort a list
- The number of times the function was called recursively

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)
```

Quicksort - Performance

On average: depending on your luck, the pivot will be in about the middle of the elements after they're sorted

- Half the elements go into less
- Half the elements go into greater
- First time: n comparison
- Second time: n/2 + n/2
- Third time: n/4 + n/4 + n/4 + n/4
- How many times do you recursively call the function?

Binary search:

N, n/2, n/4 - stop when you get to 1

log(base2) n is when you get to 1

On average: this takes n * log(n) comparisons

It is theoretically possible: you pick the worst possible pivot every time; and everything goes into less; nothing greater

P(worst pivot) = 1/n

P(worst pivot every single time) = (1/n)**n

If that happens: n**2 comparisons