Searching & Sorting & Analysis of Algorithms

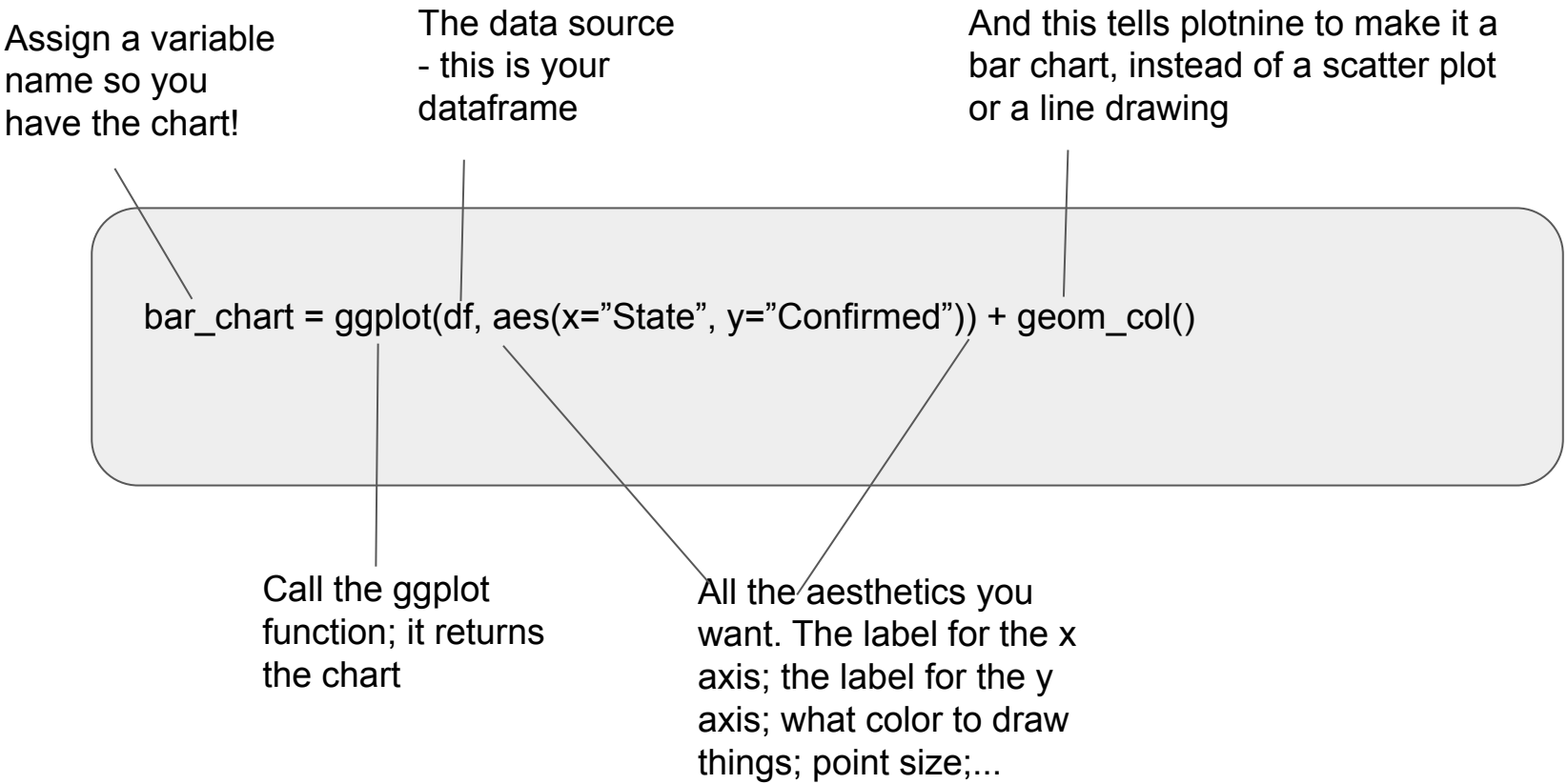
December 6, 2021

Administrative Notes

- Stand by for word about class on Monday, December 13
  - I may have to be in Colorado that day
  - It's the last class; there's no new material; it's the review for the final
  - The final is December 20, 3:30 - 5:30
  - Possibilities:
    - I may hold a live Webex session; you'll receive an invite if I do
    - I may record a video and put it on my YouTube channel
    - I'll provide answers to the sample final
- Sample final to be posted prior to Wednesday's class (December 8)
  - Final is worth 160 points per the syllabus - similar to previous exams
    - 15 T/F or MC - 3 points each (45 points)
    - 10 short answer - 7 points each (70 points)
    - 3 programming questions - 15 points each(45 points)
- Project 3 due next Monday, December 13, before midnight
  - Submit project on gl.umbc.edu as normal
    - submit cmsc201 PROJECT3 project3.ipynb

Notes on Project 3

- I posted corrected code on the github site that shows how to produce the 2D list from the 14 data files instead of the 3D we got in class
- General note on PATH variables: if you're still having trouble getting pip or jupyter to work correctly - you get an error that the command is not found - you may need to add the directory where they're installed to your PATH variable
- Reminder about how to create a graphic using plotnine/ggplot:


Assign a variable name so you have the chart!
The data source - this is your dataframe
And this tells plotnine to make it a bar chart, instead of a scatter plot or a line drawing
bar_chart = ggplot(df, aes(x="State", y="Confirmed")) + geom_col()
Call the ggplot function; it returns the chart
All the aesthetics you want. The label for the x axis; the label for the y axis; what color to draw things; point size;...

And now some Markdown examples

A good cheat sheet:

https://www.ibm.com/docs/en/watson-studio-local/1.2.3?topic=notebooks-markdown-jupyter-cheatsheet

The formal documentation:

https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Working%20With%20Markdown%20Cells.html

Back to sorting and searching

Last Wednesday we talked about ***bubble sort*** and ***selection sort***. Today we'll talk about ***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)
```

Now, searching

Mostly, the reason we sort lists is so that we can efficiently search through them later on.

Sorting is something we do once, or once in a great while. The useful operation is searching through the list to find the element you're looking for - or determine that it's not there.

Think about your project 2, where you had to find the correct student in the database. You didn't sort; you just used linear search which worked but took a while.

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(base2) 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)
```

Analysis of Algorithms: Sorting and Searching

We covered two ways to search through a list for a specific value: linear search, which always works because we search every element of a list until we find what we want; and binary search, which works if the list is already sorted

We covered three algorithms for sorting a list into order:

- Bubble sort
- Selection sort
- Quick sort

I said during that lecture that quick sort is the fastest of those algorithms. What does that mean?

Linear Search:

We have the following 120-element list:

[22, 75, 67, 24, 49, 65, 96, 81, 96, 36, 66, 100, 73, 30, 23, 32, 89, 5, 8, 70, 71, 9, 71, 77, 48, 45, 6, 73, 42, 71, 55, 98, 19, 47, 71, 21, 43, 75, 5, 72, 78, 53, 72, 89, 60, 79, 43, 89, 84, 81, 14, 31, 44, 54, 41, 91, 78, 71, 24, 24, 42, 30, 57, 55, 26, 26, 48, 65, 28, 95, 74, 93, 89, 49, 92, 86, 14, 62, 36, 15, 51, 27, 36, 6, 24, 41, 69, 54, 14, 24, 50, 6, 27, 58, 100, 45, 35, 9, 91, 57, 22, 3, 50, 72, 89, 13, 64, 0, 68, 52, 20, 16, 52, 40, 6, 74, 34, 34, 15, 71]

How many comparisons does it take to see if the number 83 is in the list, using linear search? 120, because it's not there and we have to check each element to confirm that.

How many comparisons does it take to see if the number 22 is in the list? 1, because we find it on the first comparison

Notation:

“Big O” notation: use a capital “O” to describe how long it takes for an algorithm to execute in the worst case. E.g., how many comparisons it takes.

We usually express this in terms of “n” because we assume a list of size n.

In the previous example: linear search is O(n). Because in the worst case - like with 83 - you have to check every element in the list. All n of them

Big Omega - **Ω**(n) - describes how long the algorithm takes to run in the best case. Linear search is **Ω**(1) because in the best case - like with 22 - it only takes one comparison

Binary search

Worst case - $O(\log_2 n)$ - you might not find it

Best case - $\Omega(1)$ - you might find it on the first value

Explanation:

$\log_2 n$ is the power that you have to raise 2 to in order to get n. Also, the number of times you can successively divide n in half.

If n = 256, log 256 = 8. N = 1024, log 1024 = 10

Now, the sorting algorithms

Bubble sort:

- List is n elements long
- You have to go through all remaining unsorted elements of the list each time:
  - n elements the first time; n-1 the second time; and so on.
  - This equals n * (n-1) / 2 from your calculus classes. Or, $(n^2 - n) / 2$
- We round this off to n^2.
- Bubble sort is $O(n^2)$ - WORST CASE BEHAVIOR
- Bubble sort is $\Omega(n)$ - if the list is already sorted and you stop when you don't swap anything, you only have to go through the list once

SIgma(i) from 1 to n -the sum of the first n numbers = n *(n-1) /2

Why do we round off?

120 element list - 120 squared = 14,400 so (14,400 -120) / 2 = 7140

Suppose n = 1,000,000

N squared = 1 000 000 000 000 - now

1 trillion minus 1 million = 999,999,000,000 even dividing by 2 I have half a trillion

- That’s close enough; round to 1 trillion

Don’t sweat the small stuff

Bubble sort is $O((n^2 - n)/2)$. How come we rounded that off to $O(n^2)$?

Think if n is really, really large. Say 1 million.

1 million squared is 1 trillion - 10 to the 12th power.

When n = 1 million, $(n^2 - n)/2$ = (1 trillion - 1 million)/2. Or 999 billion, 999 million /2. We just round that off to 1 trillion - it’s close enough.

Selection sort

Selection sort is almost like bubble sort

- List is n elements long
- You have to go through all remaining unsorted elements of the list each time:
  - n elements the first time; n-1 the second time; and so on.
  - This equals n * (n-1) / 2 from your calculus classes. Or, $(n^2 - n) / 2$
- We round this off to n^2.
- Selection sort is $O(n^2)$
- Selection sort is $\Omega(n^2)$
- So we say that selection sort is $\Theta(n^2)$ - if O and Ω are equal, Θ is the same value

What about quicksort? It's different

There's an area of risk. When we pick the pivot, we have no idea whether the pivot is somewhere in the middle of the values to be sorted.

If we get really, really unlucky, each time we pick a pivot it's the smallest number left, or the largest. All remaining values go to one side, and we just make the list one element smaller.

In that case, quicksort is $O(n^2)$ just like the other two algorithms.

You will often quicksort is $O(n * \log_2 n)$ - that's the more likely, average behavior

And the best case is $\Omega(n)$

But realistically...

On the average, you're not going to randomly pick the worst possible value for the pivot every time. Sometimes you're going to have good luck.

In that case, you'll divide the list to be sorted into halves, and this becomes like binary search.

So you'll often see that quicksort is O(n * log n)

If the list has n elements, n/2 will be less and n/2 will be greater

So now your recursive calls each have n/2

- n = 120; recursive calls will have 60 and 60. Then call with 30 and 30. Then 15 and 15; then 7 and 7; then 3 and 3; then 2 and 2; then 1.

What's 2 to the 7th power? 128 - approximately 120. $\text{Log}_2 120$ is approximately 7

Times n recursive calls - that's where you get (n * log n)

N * log n is always less than n squared - quicksort is faster.