Searching and Sorting In Python

November 13, 2024

Administrative Notes

Project 2 is due next Monday night at midnight

- Submit: `submit cmsc201 PROJ2 project2.py`

I

Searching and sorting

Why do we care about them so much?

- They’re common problems that happen a lot when you’re dealing with giant datasets - which is happening more and more
- They’ve been studied a lot, for decades, and we now know that there are good solutions and not-so-good solutions
-

First, Searching

This happens *a lot* in computer science - in a way, it’s how Google got rich

We’ll simplify the process to illustrate the principles:

- We’re going to search a Python list for a specified value
- All lists are going to contain only integers
    - But the principles apply regardless of your data

A long list of integers

Code:

```
import random
nums = []
length = int(input("how many numbers in the list?"))
for i in range(length):
   nums.append(random.randint(1,1000000))
print(nums)
```

The List:

```
[63235, 909657, 213738, 524781, 782837, 538957, 826029, 569339, 787505, 607449, 323143, 950157, 364441, 5508, 683898, 821711, 850789, 123268, 86392, 893582, 514801, 901787, 516386, 291951,
621488, 84650, 192842, 555013, 88127, 860928, 445408, 486647, 266054, 265450, 937358, 159121, 477167, 260612, 521104, 512932, 113106, 399472, 810371, 983559, 970939, 771883, 188522,
322171, 46881, 555802, 456408, 485306, 241569, 213156, 892385, 199540, 593192, 187611, 295321, 216583, 613342, 186072, 759204, 592838, 20678, 586948, 183504, 329238, 617203, 448421,
191400, 39777, 525165, 536724, 10812, 208995, 983209, 361426, 13724, 686053, 263874, 54074, 124680, 936770, 237193, 217521, 696018, 683375, 864473, 517445, 926260, 994578, 90109, 653833,
993103, 293910, 621710, 193828, 696601, 864474, 632079, 342640, 282573, 71425, 488586, 347545, 332389, 553196, 865441, 879996, 116220, 808194, 374743, 872214, 45740, 905528, 178274,
135424, 540837, 440221, 415130, 765922, 268481, 230925, 300634, 867538, 426981, 219531, 158665, 425201, 320926, 624786, 14771, 343009, 598511, 777464, 31521, 909069, 229042, 724522,
267154, 300803, 533873, 527568, 120905, 536946, 665060, 871403, 105264, 881816, 815959, 359837, 585848, 844930, 937431, 721043, 86641, 929996, 90942, 553128, 428913, 197676, 299399,
981245, 329788, 295315, 426409, 201905, 736769, 179395, 774971, 952509, 491852, 540079, 912848, 634250, 992521, 495394, 295918, 436589, 778505, 805887, 69627, 983871, 528763, 372299,
804158, 885350, 904318, 145038, 240707, 15564, 788024, 312137, 783727, 666788, 26733, 856713, 243607, 926752]
```

Question: is 50000 in the list?

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

Linear search: code

```
comparisons = 0

found = False

while (comparisons < len(nums)) and not found:

   if nums[comparisons] == 50000:

       found = True

   comparisons += 1

print("it took ", comparisons, " comparisons to find the answer ")
```

But maybe you’re going to be lucky:

```
comparisons = 0

found = False

while (comparisons < len(nums)) and not found:

   if nums[comparisons] == :  #whatever number is the first one in the list

       found = True

   comparisons += 1

print("it took ", comparisons, " comparisons to find the answer ")
```

Linear search: the bottom line

In the best case, you will find the number you’re looking for in ONE comparison

“Do you feel lucky?”

In the worst case, you will have to examine every number in the list before you can find the number you’re looking for, or know that it’s not there

Probability theory - on average, you will look at HALF the values in the list before you find what you’re looking for.

If there are n values in the list, you will make n/2 comparisons to search

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.

Binary search - a simple example

[1,3,5,7,9, 11,13,15]

Searching for 12
- is this element(9) the one I’m looking for? No - 9 != 12

- So take the right half of the list
- Element halfway through - 13
  - Is it == 12? No, it’s greater than 12
  - Take the left half of the list
- Element halfway through 11 - not == 12, check 9 - not == 12, end
- We did not find the number
- 4 comparisons to get our answer
  - This is the worst case!!!

A minor variation

[1,3,5,7,9]

Look for 7

Middle element: 5. 5!= 7; 7 > 5; take the right half of the list [7,9]

Worst case: pick 9; 9 != 7; pick 7 - success

3 comparisons

[1,5,3,9,0] search for 0

Code for binary search

```
def binary_search (numbers, target):
  if len(numbers)//2 == 0:  # len = 0 or 1
    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)
```

Note that this is a recursive function. This is one of the algorithms that's easier to understand in a recursive implementation

Results

On average: linear search will find the value in n/2 comparisons

Binary search will find the value in log(base2) of n comparisons

Sorting

So, now we know why we want to sort the values in a list

- Because it makes searching for a specific value so much cheaper and more efficient
- We only have to sort once, then we can search for the life of the program

TANSTAAFL - There Ain't No Such Thing As A Free Lunch

- *Heinlein, "The Moon is a Harsh Mistress", 1966*
- In computer science: if you want the algorithm to be easy to understand, it's going to cost more in time, memory etc.
- If you want fast & small, it's going to be more complex
- "Fast, cheap, good - pick two."

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):
 #makes sure you leave off the last value
#every time because it’s already got the
#biggest value
           if (numbers[j]>numbers[j+1]):
               #swap the numbers - we'll use a
temp
               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

[4, -2, 5, 19, 27, 944]

[-2, 4, 5, 19, 27, 944]

In general: n*n/2 comparisons to sort the list

1,000,000 items in the list

n*n/2 = 500,000,000,000!!!

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 sorted

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

...selection sort

Selection sort example

[4, -2, 5, 944, 27, 3]

Swap -2 with the first element

[-2, 4, 5, 944, 27, 3]

[-2, 3, 5, 944, 27, 4]

[-2, 3, 4, 944, 27, 5]

[-2, 3, 4, 5, 27, 944]

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

```python
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
```

You cannot stop selection sort early

You can stop bubble sort early BECAUSE you can keep track of how many swaps you've made each time through and you KNOW that if go through without making any swaps, you're done.

Not true for selection sort. All you can do is go through the list, and know that the first value is the smallest. You don't know anything about any other values in the list.

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