Complexity and Analysis of Algorithms

November 25, 2020

Administrative Notes

Project 3 - we'll talk about it first

The data file is encoded as ISO-8859-1, or 'latin' instead of utf8. So what you want to do is

dataframe = pd.read_csv("Project3_data.csv", encoding="ISO-8859-1")

Also, you do have to import the time module to use time.time_ns()

We'll use that in today's lecture

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?

Why do we care?

Hardware is cheap - clouds are cheaper

- If a program isn't running fast enough, buy more/faster hardware - couple of extra GPUs
- Buy an EC2 instance on AWS. Still not fast enough? Buy a bigger EC2 instance. Or go with Azure or Google Cloud instead

But some problems are so intractable that you can't solve them by buying hardware or using bigger cloud instances.

So we want to have a better algorithm that works faster and costs less. But we need to define what that means

Cost

When we talk about the “cost” of a Python program or more generally of an algorithm, do we mean:

- How much money it takes to run it?
- How much memory it takes to run it?
- How much CPU time it takes to run it? And what kind of CPU?

None of these is a very useful measure, so we generally talk about “cost” in terms of the number of operations performed.

- What operations? A comparison of two list entries; a swap of two list entries; a comparison of a list entry with a target value - something like that

Cost

For algorithms like searching for a value in a list, and sorting a list into ascending or descending order, we care about things like comparisons and swaps. Consider the iterative bubble sort algorithm we talked about on Monday:

```
def iterative_bubble_sort(numbers):
    swaps = True
    i = len(numbers) - 1
    while i > 0 and swaps:
        swaps = False
        for j in range(i):
            if numbers[j] > numbers[j+1]:
                temp = numbers[j+1]
                numbers[j+1] = numbers[j]
                numbers[j] = temp
                swaps = True
        i -= 1

    return numbers
```

This is a comparison

This is a swap

We're going to instrument this code to actually count comparisons and swaps, with lists of different lengths

Cost - what about time?

We counted the number of operations it takes to run the algorithm. What if we measured time instead?

Use the Python time module. Measure in nanoseconds, because measuring in seconds is not likely to be very useful

Reporting Algorithm Cost

In general, the number of operations (comparisons, swaps, etc.) is a function of the length of the list being operated upon. Call this length “n” - that is, a list has “n” elements.

If n is small - a few hundred, a few thousand - we’re not going to care much about algorithm cost. It’s when n starts to get to be in the millions, billions or quintillions that cost becomes important

Don’t sweat the small stuff

Now, here’s an important point: we don’t sweat the small stuff.

In some cases, we can tell that the number of operations is a function of say n^2 plus some function of n plus some other number. In math terms,

$f(n) = an^2 + bn + c.$

If n = 1 million, then n^2 = 1 trillion. And you’re never going to notice the bn or c terms when you’re running it, because 1 trillion is so much bigger than 1 million.

So we report algorithm cost only in terms of the highest power of n in the function.

We also usually get rid of *a*, the multiplier of the highest term. Why? Because it’s often not relevant.

Best case/worst case/average case

With many algorithms, the number of operations it takes is probabilistic.

If you search for an item in a list, you stop when you find the item. It makes no sense to keep searching, right?

So the “best case” might be “1 operation.” Found it the first time!!

You might have to search every doggone item in the list. Either it’s not there at all, or you find it in the very last element. That would be “worst case.”

Apply laws of probability. On average, you’ll find the item halfway through, if there’s a true random distribution of items in the list & items you want to find.

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.

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 $O(n^2)$
- 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

Remember - don’t sweat the small stuff

Bubble sort is $O((n^2 - n)/2)$ or $O(0.5n^2 - 0.5n)$. 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.

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)