The Python debugger *and* Analysis of Algorithms

December 3, 2019

Administrative Notes

Welcome Back! Hope everyone enjoyed the Thanksgiving holiday

You should have submitted the Project 3 Design last Friday

Project 3 is due this Friday, December 6!!!

    Don’t wait until the last minute to go to office hours if you need help

The Final Exam is NEXT Friday, December 13, from 6 to 8 pm

A quick review

It’s been almost two weeks. What did we do last time?

Binary and hexadecimal numbering systems

- Binary means base 2 - the only valid digits are 0 and 1
    - Every place to the left is a power of 2 more than the previous place
- Hexadecimal (“hex”) means base 16 - the valid digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F
    - Every place to the left is a power of 2 more than the previous place
    - Hex digits are preceded by “0x” in most computer languages

Review, continued

What is 10011000 binary in decimal? In hex?

What is BED hex in decimal? In binary

On to the Python debugger

A nice little tool that lets you examine your code while it’s running, so you can see where your errors are

PyCharm has its own debugger, but pdb is generic and works across pretty much all python implementations

import pdb #now you can use it

The “official” documentation is at https://docs.python.org/3/library/pdb.html

Using the python debugger

In a python console:

```
>>> pdb.run(‘program_name.py”)   # runs the program under the control of pdb
```

In a terminal:

```
python3  -m pdb program_name.py
```

(some examples of this usage)

Breaking into the debugger at a specific point in the code

Sometimes you know the error isn’t at the beginning of your code; it’s occurring later

There’s no need to run the first part of your program in the debugger

So you set a break point in your code where you want to enter the debugger

```
>>>import pdb
```

and then `pdb.set_trace()`

right before you want to break into the code.

“Single stepping”

“Single stepping” means to execute your program one statement at a time

- You can see exactly what happens when each statement is executed

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

Best case - **Ω**(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.

A coding example

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

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 just 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 $\Omega(n)$ - if the list is already sorted and you stop when you don’t swap anything, you only have to go through the list once

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)