Pokerbots 2021

Lecture 4: Performance

Announcements

Libratus Creator:
Noam Brown

This Friday @ 1:00pm EST!
pkr.bot/class

Poker Night Study Break!

(with prizes + sponsors)

Wed 01/20 after OH

Details TBA

Final Tournament & Sponsor Networking: Friday, January 29th

C++ Skeleton Updates!

```
$ git pull
```

Special Prize Announcement

New Prize: Strongest Alliance

Pokerbots is better with more teammates! Improve your bot and collaborate more effectively with an *Alliance.*

Alliance: A new team formed from two or more existing teams

Learn more and form your alliances at pkr.bot/alliance!

Weekly Tournament 1

WINNER:

AzureFractal

BIGGEST UPSET:

after midnight strip poker

Giveaway

Guess the runtime of Tournament 1:

pkr.bot/time

Agenda

- “What happens when I run my code?”
- Algorithmic Considerations
- Asymptotic Complexity
- Python Data Structures
- Live Coding Session

What happens when I run code?

Short answer: **it depends**

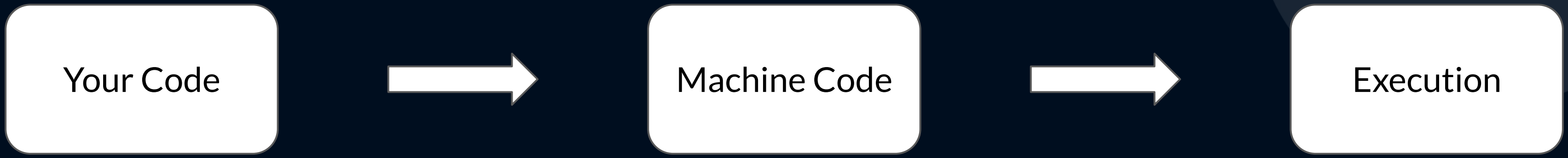
- What language?
- What version?
- What operating system?
- What processor?

But it basically goes like this...

Your computer can’t understand your code...It needs to be translated!

Things called compilers and interpreters translate your code into **machine code**

Machine code tells your computer what to do in a way it can understand



Your Code

Machine Code

Execution

Your computer's core parts

Machine Code invokes some of your computer's lowest level operations. These things are controlled by…

- Processor (CPU)
- Arithmetic Logic Unit (ALU)
- Registers
- Memory/Storage
- Caches

What takes so long?

“If electrons travel at the speed of light, then why is my code so slow?”

Again, **it depends**…

But, practically speaking, it comes down to two things:

1. Your language’s “translator”
2. The instructions you’re giving your computer

Compilers vs Interpreters

Compilers and interpreters share the same goal: convert your code into a format that your processor can understand. They achieve this goal in in different ways.

Compilers:

- Translate your code all at once, **before** it’s run by your computer
- Examples: C, C++, Java, Go

Interpreters:

- Translate your code a bit at a time, **when it is being run.**
- Examples: Python, MATLAB

Pros and Cons

Compilers:

- Compiling the code all at once can be slow :(
- Actually running the code is typically very **fast** :)
- Usually limited to static typing :(
  - ex: List<Integer> only lets us use integers in this list!

Interpreters:

- Interpreting code as it’s run is **slow** :(
- We have an easy time writing the code with dynamic typing :)
  - ex: Python lets us put anything in our list! [5, 6.176, True, None, Player(), ‘hello world’]

Choosing your language

Beyond swapping languages, there’s not much that can be done to help your code’s ‘translation’ process. (Besides rewriting it or precompiling or something else)

Here’s what we recommend:

- Choose the language you’re most comfortable with!
- If you need **really fast code**, try Java or C++
- If you’re new and want an easier time programming, try Python
- Trying to learn a new language? Give it a shot!

Your instructions

Regardless of language, the core instructions you give to your computer really impact your code's speed.

- Addition, subtraction, and other math operations (ALU)
  - Typically **fast**
- Loading and moving variables in your code
  - Registers - **fast,** Main Memory - **slower**, Disk/Storage - **slowest**

We need to be clever with the way we use these instructions. Efficient **algorithms** will help our code run faster!

Algorithms

Algorithms are sequences of instructions we give to computers to solve a particular problem

Problem solving

Whenever we’re faced with a problem in computer science, we typically make an algorithm to solve that problem!

A ‘recipe’ we can follow to solve a type of problem every single time we need to. Your Pokerbot is an algorithm for Blotto Hold’em!

Some algorithms are designed better than others...

Algorithm example

Classic example: Given a list of numbers, sort the numbers

There are dozens of algorithms that solve this problem. Some do it more efficiently than others.

- ○ *Merge Sort, Quick Sort, Heap Sort, Insertion Sort, Bubble Sort, Counting Sort, Selection Sort, Radix Sort, Bogo Sort, Timsort…*

Does this sound efficient?

- ○ *Randomly shuffle the list. If it’s sorted, return the list. Otherwise, shuffle it again…*

I don’t think so...

Complexity

Algorithmic efficiency

How do we measure the efficiency of an algorithm? We typically measure two things:

- How much time does it take to run in the worst case?
- How much space (memory) does it take?

We'll focus on time for now, but you have to balance both in Pokerbots!

Runtime complexity

“My algorithm runs in 1 second on my new computer, but it takes 10 seconds on my old computer! What’s going on here?”

- Advances in computing make wall-clock time a bad measure of run time
- Our analysis of an algorithm should be agnostic to the machine we use to run it
- We’ll use *asymptotic complexity* to measure time efficiency!

Main Idea: How many operations does our algorithm perform **as a function** of our input size?

Complexity example

Searching: Given a list of numbers of size ***n***, determine if the number ***k*** is in the list

- Example: Is 7 in the following list? [1, 6, 3, 29, 12, 7, 8, 10]
    - Yes! How did you do it?

Potential algorithm: *Check each element of the list. If the element is equal k, then we are done! If none of the numbers are equal to k, then the number is not in the list.*

How good is this? We check each element and compare it to our value *k*. In the worst case, we need to check *every element* in the list! Thus our algorithm might make ***n*** operations!

Other cases

Imagine another algorithm where we perform a few operations (say, 5) for every item in the list. Thus we would have *5n* operations in the worst case!

In other cases, we could perform *7n* operations for every element in the list. This would be $7n^2$ *total operations!*

We care about what happens when *n* gets arbitrarily large. This is called *asymptotic* complexity.

Asymptotic complexity

We use limits and asymptotic behavior of our complexity functions to classify our algorithms. *Big* - *O* notation captures this behavior in the worst case.

A function $f(x) = O(g(x))$ *if*

$$|f(x)| \leq C * g(x) \text{ for all } x > x' \text{ for some } x' \text{ and some constant } C$$

Essentially, $f(x) = O(g(x))$ if $g(x)$ grows at least as fast as $f(x)$ up to a constant.

Back to our examples

In our sorting example, we had about *n* operations to perform. Thus our algorithm is O(n)

If we had *5n* operations, it would still be O(n), because they are the same up to a constant (5 in this case)

The example with $7n^2$ operations is O(n^2) for the same reason!

Constant time operations

There are a few operations that computers perform incredibly fast. These don’t depend on size (for most uses), and are called *constant time operations.*

- Example: to your computer, adding 5 + 10 takes basically the same time as adding 500 + 1000

We refer the the complexity of constant time operations as *O(1)*

Applications to Python

Complexity of Python

How does this analysis translate into Python? Each Python operation is associated with some cost!

Constant time *O(1)* operations:

- Math on numbers (`+`, `-`, `/`, `*`, etc)
- Adding things to the end of a list (`append`)
- Assigning a new variable (`x = 5`)
- Getting a value at a list index (`x = list[i]`)

Complexity of Python

Other methods are not so fast:

- `in` operator is slow when used with lists - *O(size)*
- `for` and `while` loops can be big time hogs (remember Monte Carlo?) - *O(loops)*
- copying things (especially lists) - *O(size)*

Can we do any better?

Better Python data structures

Python has a few built-in data structures that make code much more efficient!

Notably, Python has two main *hash*[1] table objects (more details in classes like 6.006)

- *dictionaries* map *keys* to *values*
    - we can look up a value using its key in *O(1)*
- *sets* keep track of a collection of items
    - The *in* operator searches sets in constant time! Much faster than with lists!

[1] *There are some caveats in these cases (Google expected and amortized complexity). This shouldn't matter for our purposes, though.*

Python libraries

Many Python libraries utilize faster coding techniques “under the hood” to achieve speed increases in practice!

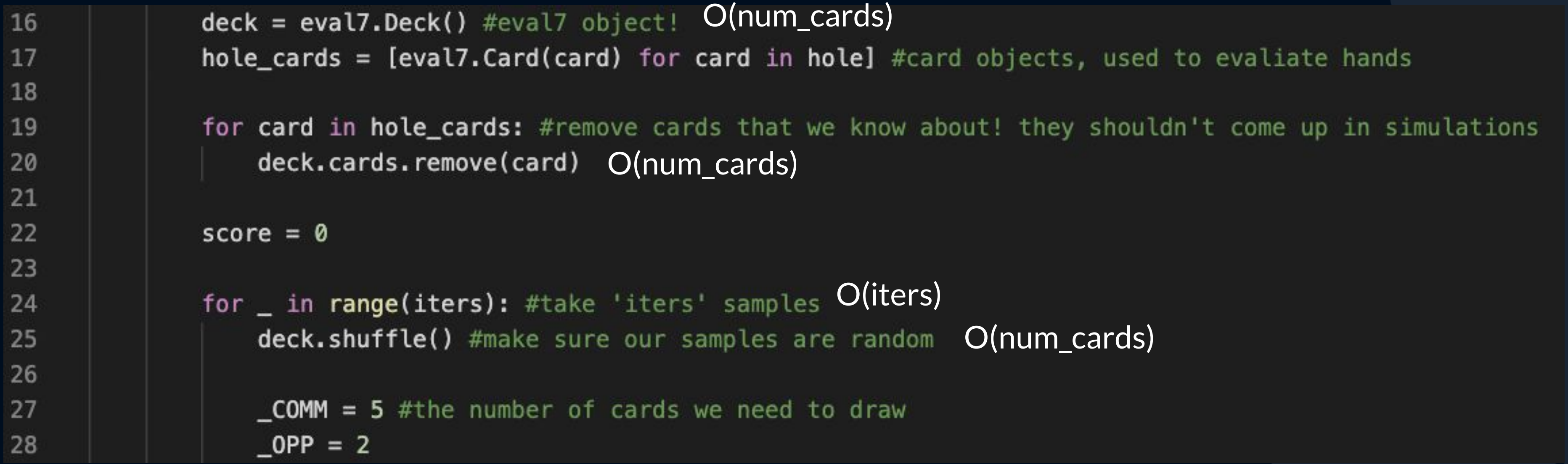
- `import pandas as pd`
  - Useful for storing, loading, and manipulating spreadsheets of data in Python
- `import numpy as np`
  - Great for operating on large arrays of numbers with amazing speed. Written in C behind the scenes!
- `import itertools`
  - Used for efficient looping - Combinations, Permutations, Counting, etc.

reference-lecture-4-2021

Analyzing our Monte Carlo

- Run 3 times per round…
- Run 500 rounds per game…
- 1500 runs for every single game!
- ...and it’s slow

Analyzing our Monte Carlo



```
        deck = eval7.Deck() #eval7 object!
        hole_cards = [eval7.Card(card) for card in hole] #card objects, used to evaliate hands

        for card in hole_cards: #remove cards that we know about! they shouldn't come up in simulations
            deck.cards.remove(card)

        score = 0

        for _ in range(iters): #take 'iters' samples
            deck.shuffle() #make sure our samples are random

            _COMM = 5 #the number of cards we need to draw
            _OPP = 2
```

O(num_cards)

O(num_cards)

O(iters)

O(num_cards)

O(num_cards * iters)

Fixing Monte Carlo

We have (52 choose 2) = 1326 possible hole cards to calculate!

- But we’ve been doing 1500 calculations per game….oh no

It actually gets much worse than that!

- Ah, Kd is basically the same as Ad, Kh….
- As, Ts is basically the same as Ah, Th….
- Which means we only need to consider “suited” or “off-suited” versions of the hole
- Which brings our number down to below 169 possible holes….wow

Our Plan

- Make all of the pairs of ranks we can encounter in a game (sorted by rank)
  - AA, AK, AQ, AJ, AT…. KK, KQ, KJ, KT …. QQ, QJ, QT ….
- For each of these holes, we'll consider them either "suited" or "off-suited"
  - AKs, AKo, AQs, AQo …. Pocket pairs are always "off-suited": AAo, KKo ….
- Calculate strengths for an example of each of these
  - Only 169 simulations!
- Save them all in a .csv for us to use later!

Giveaway Winners!

Tournament Runtime:
849 minutes