INFO 450 Fall 2020

Week 10

Oct 22, 2020

Agenda

1. Problem Solving
2. Homework Review
3. Data structures
4. Big O / Complexity
5. Sorting

More Problem Solving

- Let's continue to figure out 'how' to program logic, not just the programming part.

Instructions

Your roommate has invented a really cool robot. Yes, a REAL robot that does things at your command!

This robot is super smart and is instructed by interacting with the human voice. No programming required!

The robot can understand simple instructions. Grab something, turn something, squeeze, punch, lift, all kinds of commands.

If I wanted the robot to do a jumping jack, I'd tell it something like:

Professors note: I don't think those instructions are detailed enough, but hopefully you get the point.

Oh dang. I'm hungry

Can anyone make me a peanut butter and jelly sandwich?

- Everyone take 10 minutes
- Use notepad, email, piece of paper, whatever
- Write down your instructions
- I will randomly pick people to read their instructions
- We will all critique them.

Homework Review

Two approaches to solutions:

- Brute Force
- Smart Math Patterns

Data Structure

List

Queues

Stacks

Lists

- Linear list of objects/values.
- Size of the list is unknown, vs an array of a fixed size.
- We covered these extensively

```
my_list = []
my_list.append("one")
my_list.append(2)
my_list.append("foo")

for x in my_list:
    print(x)
```

Queue

Standing in line at a roller coaster.

First in, first out (FIFO)

This is a slow implementation, but using concepts we know already

```
my_queue = []
my_queue.append("first")
my_queue.append("second")
my_queue.append("third")
my_queue.append("fourth")


print(my_queue.popleft())
print(my_queue.popleft())
print(my_queue.popleft())
print(my_queue.popleft())
```

Better implementation

Professor note: pronunced 'deck', double ended queue

Collections package, deque object

```
from collections import deque
queue = deque()
queue.append("first in")
queue.append("second in")
queue.append("third in")

print(queue.popleft())
print(queue.popleft())
print(queue.popleft())
```

Stack

First in, Last Out (FILO)

Last In, First Out (LIFO)

- Coins in the car coin slot holder.
- Text editor, undo function

```
my_stack = []
my_stack.append("first in")
my_stack.append("second in")
my_stack.append("third in")

print(my_stack.pop())
print(my_stack.pop())
print(my_stack.pop())
```

Traditional methods for stacks: push, pop

Big O

Complexity of an algorithm

We use Big-O notation to asymptotically bound the growth of a running time to within constant factors above and below. Sometimes we want to bound from only above.

Asymptotic: so defined that their ratio approaches unity as the independent variable approaches a limit or infinity. (CHF: I liked this one the best)

What the heck does this mean? In programming, we measure the efficiency of an algorithm on a collection of data, in relation to the number of items in the collection.

Example

Consider a sorting algorithm on an array:

```
my_numbers = [5,1,6,3,2]
```

There are 5 elements in the array.

We write a sorting algorithm that puts them in the correct order, and it takes 5 units of time.

(computers are fast, so this could be 5 * 100 nanoseconds, 5 * 1 second, whatever.

That is why we talk about "units of time".

Now what?

```
my_numbers = [5, 1, 6, 3, 2, 8, 3, 7, 8, 2]
```

Now we are up to 10 items in our array, or double the number of items.

We measure the efficiency of our algorithm by the units of time, or set of operations, that get executed dependant on the number of items in the array.

In this case, if the 10 elements take 10 units of time, then we know our algorithm has a linear relationship to the number of elements.

This is labeled as O(n).

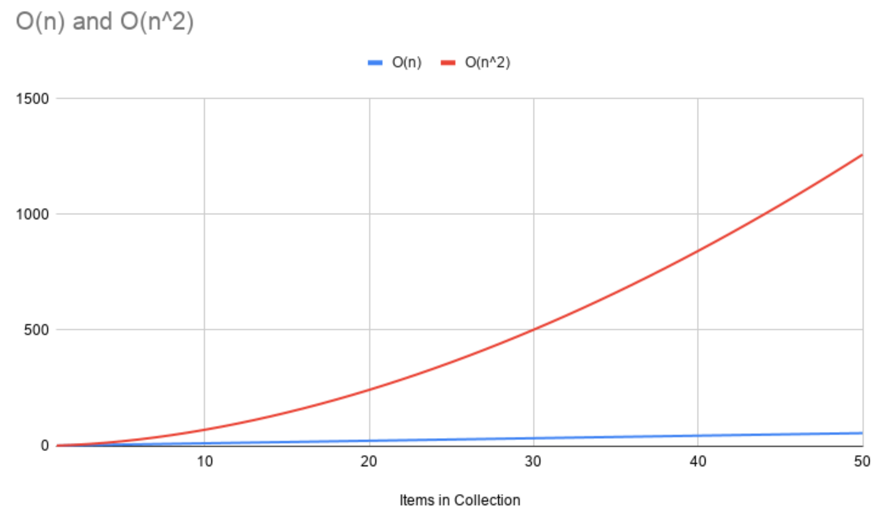
Where O is the notation of complexity, and represents the linear relationship.

This is similar to plotting x=y on a graph.

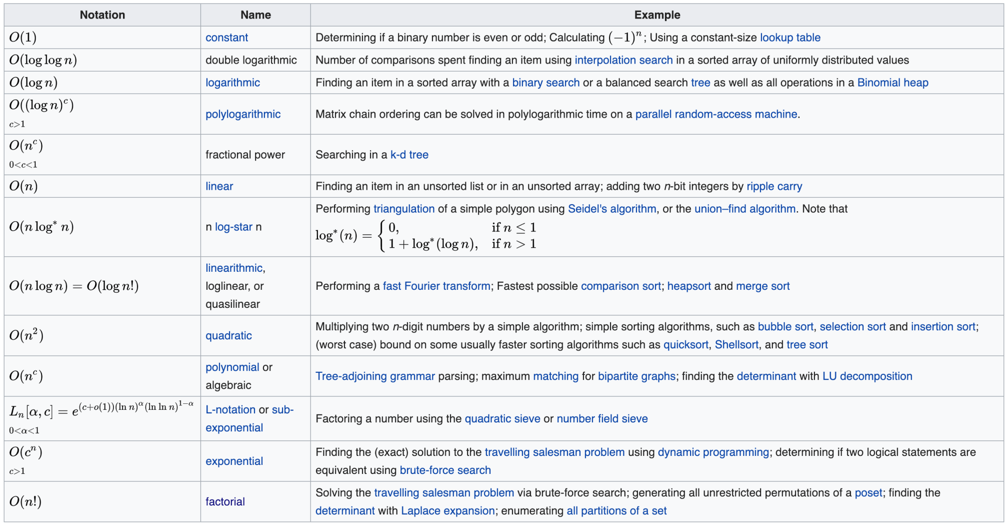
Can it be worse?

Using the same 10 items in our array, if we plot the time to sort the algorith, and we see an exponential relationship, then we have a problem.

The Red line indicates O(n^2)



List of Big O Complexities



Notation	Name	Example
$O(1)$	constant	Determining if a binary number is even or odd; Calculating $(-1)^n$; Using a constant-size lookup table
$O(\log\log n)$	double logarithmic	Number of comparisons spent finding an item using interpolation search in a sorted array of uniformly distributed values
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap
$O((\log n)^c)$ $_{c>1}$	polylogarithmic	Matrix chain ordering can be solved in polylogarithmic time on a parallel random-access machine.
$O(n^c)$ $_{0<c<1}$	fractional power	Searching in a k-d tree
$O(n)$	linear	Finding an item in an unsorted list or in an unsorted array; adding two *n*-bit integers by ripple carry
$O(n\log^* n)$	n log-star n	Performing triangulation of a simple polygon using Seidel's algorithm, or the union–find algorithm. Note that $\log^*(n) = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log n), & \text{if } n > 1 \end{cases}$
$O(n\log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear	Performing a fast Fourier transform; Fastest possible comparison sort; heapsort and merge sort
$O(n^2)$	quadratic	Multiplying two *n*-digit numbers by a simple algorithm; simple sorting algorithms, such as bubble sort, selection sort and insertion sort; (worst case) bound on some usually faster sorting algorithms such as quicksort, Shellsort, and tree sort
$O(n^c)$	polynomial or algebraic	Tree-adjoining grammar parsing; maximum matching for bipartite graphs; finding the determinant with LU decomposition
$L_n[\alpha, c] = e^{(c+o(1))(\ln n)^\alpha(\ln\ln n)^{1-\alpha}}$ $_{0<\alpha<1}$	L-notation or sub-exponential	Factoring a number using the quadratic sieve or number field sieve
$O(c^n)$ $_{c>1}$	exponential	Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search
$O(n!)$	factorial	Solving the travelling salesman problem via brute-force search; generating all unrestricted permutations of a poset; finding the determinant with Laplace expansion; enumerating all partitions of a set

Source: https://en.wikipedia.org/wiki/Big_O_notation

Sorting

- Bubble Sort
- Selection Sort
- Merge Sort

Bubble Sort

O(n^2) Video

Bubble Sort

Wikipedia: Bubble Sort

Bubble Sort (code)

```python
import random

def bubble(inbound):
    outbound = inbound.copy()
    n = len(outbound)
    for i in range(n):
        for j in range(0, n - i - 1):
            if outbound[j] > outbound[j + 1]:
                outbound[j], outbound[j+1] = outbound[j + 1], outbound[j]

    return outbound

if __name__ == "__main__":
    my_list = []
    for x in range(20):
        my_list.append(random.randint(0, 1000))
    print(my_list)

    sorted_list = bubble(my_list)

    print(sorted_list)
```

Selection Sort

O(n^2)

Video

Selection Sort

Wikipedia: Selection Sort

Selection Sort (Code)

```python
import random

def selection(inbound):
    outbound = inbound.copy()
    for i in range(len(outbound)):
        min_idx = i
        for j in range(i + 1, len(outbound)):
            if outbound[min_idx] > outbound[j]:
                min_idx = j
        outbound[i], outbound[min_idx] = outbound[min_idx], outbound[i]

    return outbound

if __name__ == "__main__":
    my_list = []
    for x in range(20):
        my_list.append(random.randint(0, 1000))
    print(my_list)

    sorted_list = selection(my_list)

    print(sorted_list)
```

Recursion

is the process of defining something in terms of itself, circular definitions.

e.g.

In programming (not just C++), recursion is the process of a function calling itself.

A function that calls itself is said to be .

Classic Factorial Example

The factorial of a number is the product of all the whole numbers between and

For Example, 4 factorial is 1x2x3x4 = 24

Let's compare the 'iterative' implementation of factorial to the recursive implementation

Iterative Implementation

```python
def fact(n):
    if n <= 0:
        raise Exception("Can't factorial a number less than 0")
    answer = 1
    for x in range(1, n + 1):
        answer *= x
    return answer

if __name__ == "__main__":
    for x in range(1, 21):
        print(f"Factorial of {x} is {fact(x)}")
```

Iterative Factorial

Recursive Implementation

```python
def recursive_fact(n):
    if n <= 0:
        raise Exception("Can't factorial a number less than 0")
    if n == 1:
        return n
    answer = recursive_fact(n - 1) * n
    return answer


if __name__ == "__main__":
    for x in range(1, 21):
        print(f"Recursive Factorial of {x} is {recursive_fact(x)}")
```

Recursive Factorial

Breaking it down

recursive_fact is the function with an integer

```
def recursive_fact(n):
```

Quick business rule, can't factorial a number less than 0

```
if n <= 0:
    raise Exception("Can't factorial a number less than 0")
```

CRITICAL to   have an 'exit' point of a recursive function, otherwise, you'll hit an infinite loop, blowing your stack

```
if n == 1:
    return n
```

The recursion. call factr with one less than n. Which will then call factr with another one less than n, until n is 1. Then, returns it to by multipled against the 'next' number

```
answer = recursive_fact(n - 1) * n
```

Return the answer

```
return answer
```

Merge Sort

Divide and Conquer

O(n log n)

Better than O(n^2)

Video

Merge Sort

Wikipedia: Merge Sort

Merge Sort

```python
import random

def merge_sort(inbound):
    if len(inbound) >1:
        mid = len(inbound)//2 # Finding the mid of the array
        left_array = inbound[:mid] # Dividing the array elements
        right_array = inbound[mid:] # into 2 halves

        merge_sort(left_array) # Sorting the first half
        merge_sort(right_array) # Sorting the second half

        i = j = k = 0

        # Copy data to temp arrays left_array[] and right_array[]
        while i < len(left_array) and j < len(right_array):
            if left_array[i] < right_array[j]:
                inbound[k] = left_array[i]
                i+= 1
            else:
                inbound[k] = right_array[j]
                j+= 1
            k+= 1

        # Checking if any element was left
        while i < len(left_array):
            inbound[k] = left_array[i]
            i+= 1
            k+= 1

        while j < len(right_array):
            inbound[k] = right_array[j]
            j+= 1
            k+= 1

if __name__ == '__main__':
    my_list = []
    for x in range(20):
        my_list.append(random.randint(0, 1000))
    print(my_list)

    merge_sort(my_list)

    print(my_list)
```