CS 142

Lecture 2
Abstraction & Analysis

Rhodes College

Practice from Last Time

- Write a program where the computer picks a random number from 1 to 100 and you have to guess what it is.
  - The computer will report whether each guess is too high, too low, or correct.
  - Report the number of guesses it takes to get it right.
- Write a program to simulate a turn of the game "One is Zero".
  - During a turn, you roll a six-sided die.
  - If you roll a 2-6, you get that number of points and may roll again to get more points, or you may choose to end your turn.
  - As soon as you roll a 1, your turn ends, you lose any points you already received for that turn, and get zero points for the turn.

Announcements

Program 1 has been assigned
- Due on 1/27 by 11:55pm

Abstraction

To tackle a large software project, it is essential to break it into smaller pieces.

One idea: divide the problem into a set of cooperating functions (divide-and-conquer)
-Also referred to as *functional abstraction*.

Abstraction

When you utilize the `mylist.index(item)` function you are using abstraction

- You don't know how it is implemented, but you do know that it will return an integer (either a -1 to tell you the item is not in the list, or a number from 0 to len(lst)-1 indicating the index of that item.
- "Black-box" – you don't know how it is implemented, but you know how it works

- Always write your code in such a way that you can achieve *implementation independence*
  - You want each function to work independently, so that if you changed the implementation of 1 function, it would not disrupt the rest of your code, or your co-worker's code
- Design programs using a *top-down design*

Measuring Efficiency

Analysis of algorithms: area of Computer Science which provides tools for comparing the efficiency of different methods of solving a problem.

What criteria might we use to analyze the performance of an algorithm?

Space or Time?

Mainly interested in how long our programs take to run, as time is generally a more precious resource than space.

How long does it take to run?

- Three difficulties with comparing the times taken for programs to run:
  - How are the algorithms code?
  - What computer should you use?
  - What data should the programs use?
- Our analysis should be independent of specific:
  - Coding,
  - Computers,
  - data

Measuring Algorithm Efficiency

How do we achieve an analysis of algorithms that is independent of specific implementations, computer used, and data?

Count the number of basic operations of an algorithm, and **generalize** the count.

What are basic operations?

Read, write, compare, assign, mathematical operations (add, subtract, divide, multiply, increment, decrement), open, close, logical operations (and, or, not), ….

Counting Operations Example

Example: calculating the sum of the first 10 elements in a list

```
def count_operations(items):
    total = 0
    index = 0
    while index < 10:
        total = total + items[index]
        index += 1

    return total
```

<- 1 assignment
<- 1 assignment
<- 11 comparisons
<- 10 plus/assignments
<- 10 plus/assignments

<- 1 return

Total: 34 operations

Counting Operations Example

Calculating the sum of the elements in a list.

```
def count_operations2(items):
    n = len(items)
    i = 0
    total = 0
    while i < n:
        total = total + items[i]
        i += 1

    return total
```

<- 1 assignment
<- 1 assignment
<- n+1 comparisons
<- n plus/assignments
<- n plus/assignments

<- 1 return

Total: 3n + 5 operations

We need to measure an algorithm's time requirement as a function of the **problem size**, e.g. in the example above, the problem size is n, the number of elements in the items list.

Problem Size

- Performance is usually measured by the rate at which the running time increases as the problem size gets bigger.
  - Looking at relationship between run-time and problem size
  - Need to first identify problem size
- Example: Analyzing an algorithm that processes a list
  - Problem size = size of list
- In many cases, the problem size will be the value of a particular variable, where the running time of the program depends on how big that value is.

Counting operations - Exercise

How many operations are required to do the following tasks?

1. Adding an element to the beginning of a list containing n elements.
2. Printing each element in a list containing n elements,
3. Adding a single element to a list using the append() function
4. Performing a nested loop where the outer loop is executed n times, and the inner loop is executed 9 times, e.g.,

```
items = [  [1, 2, 3, 4, 5, 6, 7, 8, 9],
           [1, 2, 3, 4, 5, 6, 7, 8, 9],
           [1, 2, 3, 4, 5, 6, 7, 8, 9],
           [1, 2, 3, 4, 5, 6, 7, 8, 9], … , [ … ]  ]
```

```
sum = 0
for i in items:
    for j in i:
        sum += j

print("sum: ", sum)
```

or

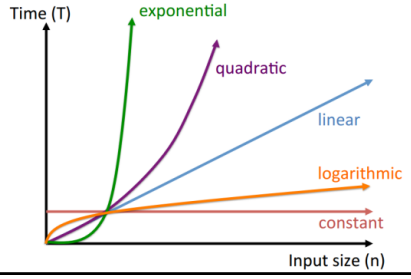
```
sum = 0
for i in range(len(items)):
    for j in range(9):
        sum += items[i][j]

print("sum: ", sum)
```


Algorithm Growth Rate Function

How quickly does time taken by the algorithm increase as a function of the size of the problem?

The growth rate function is a mathematical function used to specify an algorithm's run-time in terms of the size of the problem.



Growth rate function – A or B?

- Consider the following 2 algorithms.

ALGORITHM A

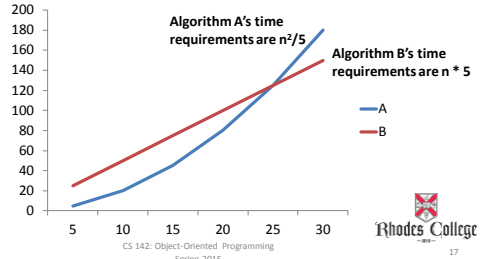
```
n = …
for i in range(n):
    j = 0
    while j < n:
        process(i, j)
        j = j + 5
```

ALGORITHM B

```
n = …
j = 0
for i in range(n):
    while j < 5:
        process(i, j)
        j += 1
```

Growth rate function – A or B?

n	5	10	15	20	24	25	26	30
A	5	20	45	80	115.2	125	135.2	180
B	25	50	75	100	120	125	130	150



Growth rate function – A or B?

For smaller values of n, the differences between algorithm A ($n^2/5$) and algorithm B(5* n) are not very big. But the differences are very evident for large problem sizes such as when n > 1,000,000

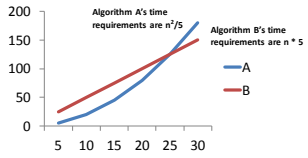
In n is 10^6, then Algorithm A's time is $(10^6)^2 / 5 = (10^{12})/5 = 2 * 10^{11}$
Algorithm B's time is $5 * 10^6$

Bigger problem size produces bigger differences

Algorithm efficiency is a concern for **LARGE** problem sizes.

Big-O Notation

- We use Big-O notation (capital letter O) to specify the complexity of an algorithm, e.g. O(n), $O(n^2)$, $O(n^3)$.
- If a problem of size *n* requires time that is directly proportional to N, the problem is O(n) **(Algorithm B)**
- If time requirement is directly proportional to n^2, the problem is $O(n^2)$ **(Algorithm A)**



Big-O Notation

Only interested in the performance of the program when n is LARGE.

Therefore, we ignore:

- Added constants
- Constant multipliers
- Smaller terms

Constants do not matter in terms of how an algorithm scales.

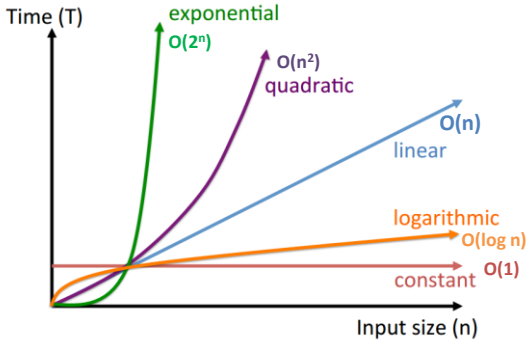
Big-O Notation

- Compactly describes run-time of an algorithm.
- If your algorithm for sorting a list of *n* numbers takes roughly n^2 operations for the most difficult dataset, then we say that the running time of your algorithm is $O(n^2)$.
- Actual Runtime (depends on your implementation):
  - $-1.5n^2$
  - $-n^2+n+2$
  - $-0.5n^2+1$

Big-O Formal Definition

- When we write that the running time of an algorithm is $O(n^2)$ mean that it does not grow faster than a function with a leading term of $c \cdot n^2$, for some constant c.
- Formally, a function f(n)is Big-O of function g(n), or O(g(n)), when $f(n) \leq c \cdot g(n)$ for some constant c and sufficiently large n.

Comparison of Growth Rates



Examples of Big-O complexities

- **Linear Time - O(n)** – finding the max in a list
- **Logarithmic Time - O(log n)** – Binary search
- **Quadratic Time – $O(n^2)$** – Finding closest pair of points
- **Cubic Time – $O(n^3)$** – Enumerate all triples of elements.
- **Polynomial Time: $O(n^k)$** - Enumerate all subsets of k nodes.
- **Exponential Time – $O(2^{n^k})$** - Enumerate all subsets.

Big-O Notation Practice

What is the Big-O of the following growth functions?

a. T(n) = n + log(n)
b. T(n) = n^4 + n*log(n) + 3000n^3
c. T(n) = 300n + 60n*log(n) + 342

Worst-case and average-case analysis

An algorithm can require different times to solve different problems of the same size. For example, a search for an item in a list.

Best-case analysis: the minimum amount of time that an algorithm requires to solve problems of size n

Worst-case analysis: the maximum amount of time that an algorithm requires to solve problems of size n

Average-case analysis: the average amount of time that an algorithm requires to solve problems of size n

Average performance and worst-case performance are the most commonly used in algorithm analysis.

More Practice

What is the Big-O of the following functions?

```
def exampleA(n):
  s = "PULL FACES"

  for i in range(n):
    print("I must not ", s)

  for j in range(n, 0, -1):
    print("I must not ", s)
```

```
def exampleB(n):
      s = "JUMP ON THE BED"

      for i in range(n):
            for j in range(i):
                  print("I must not ", s)
```

```
def exampleF(n):
  s = "FORGET MY MOTHER'S BIRTHDAY"
  i = n
  while i > 0:
    outF(s)
    i = i // 2

def outF(s):
  for i in range(25, 0, -1):
    print(i, "I must not ", s)
```

```
def exampleD(n):
      s = "PROCRASTINATE"

      for i in range(n):
            for j in range(n, 0, -1):
                  outD(s, n / 2)

def outD(s, b):
      number_of_times = int(b % 10)
      for i in range(number_of_times):
            print(i, "I must not ", s)
```


Next Time

No class on Monday

Read Sections 2.1-2.3