A Simple Introduction to Algorithmic Complexity

Chapter 11

Priorities

- What is the most important consideration when designing and developing a program or system?
  - The result is correct!

- What else is important?
  - Performance
    - Real Time
    - On Demand
    - Periodic
  - -ilities

What are –ilities

Non-functional requirements

- Maintainability
- Reliability
- Usability
- Adaptability
- Availability
- Security
- Portability
- Scalability
- Testability
- Reusability
- Sustainability
- Efficiency
- Safety
- Fault tolerance

https://ieeexplore.ieee.org/document/1353217

One more -ility

- Readability
  - Remember: You read code more than you write it

Back to performance

- What effects performance?
  - I/O throughput
  - Database systems
  - Server performance
  - Data volume
  - Computational complexity
- **Conceptual Complexity**
  - How difficult is it to understand an algorithm
- **Computational Complexity**
  - How hard does the computer actually have to work

Remember the Fibonacci sequence

- f(0) = 1
- f(1) = 1
- f(n) = f(n-1)+f(n-2)

```
def fib(n):
    if n == 0 or n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Thinking about computational complexity

- How long will our fib function take?
- It depends:
  - How big is n
  - How long does it take to for each statement
  - How fast is the computer
  - How fast is the Python interpreter

What’s a body to do?

- Theoretical computer **random access model**
  - Each **step** is processed sequentially
  - Each **step** takes the same amount of time
- We abstract the size of the inputs
  - **Best Case** – fib(0) or fib(1)
  - **Worst Case** – fib(maxInt)
  - **Average (Expected) Case** – Average of best and worst, or can use *a priori* knowledge of how the system is usually used

  - **Murphy’s Law**

Watch your step

```
def fact(n):
    """ computes the factorial of n (i.e. n!)
    Assumes n is an integer > 1"""
    answer = 1
    while n > 1:
        answer = answer * n  # answer *= n
        n = n - 1            # n -= 1
    return answer
```

- How many steps?
    - 5n + 2

Watch your step (cont.)

- 5n+2
- The +2 is only significant for the very smallest values of n (best case) – and even then not very significant
- What about the 5*
  - When testing for worst case even the 5 becomes less significant if not insignificant
  - When comparing implementations the constant is usually similar
  - We will ignore the constants

Searching a list

```
def linear_search(L, x):
    for e in L:
        if e == x:
            return True
    return False

my_list = []
for i in range(100):
    my_list.append(i+1)
```

```
print(linear_search(my_list, 1))
print(linear_search(my_list, 100))
print(linear_search(my_list, 0))
```

Calculating square roots

```
def square_root_exhaustive(x, epsilon):
    step = epsilon**2
    ans = 0.0
    while abs(ans**2 - x) >= epsilon and
                 ans*ans <= x:
        ans += step
    if ans*ans > x:
       raise ValueError
    return ans
```

```
square_root_exhaustive(25, 0.001)
4999900
```

```
def square_root_bi(x, epsilon):
    low = 0.0
    high = max(1.0, x)
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= epsilon:
      if ans**2 < x:
         low = ans
      else:
         high = ans
      ans = (high + low)/2.0
    return ans
```

```
square_root_exhaustive(25, 0.001)
15
```

Asymptotic notation

- Describes the relationship between the time an algorithm completes and the size of the input
- Best case is not very interesting
- Worst case – on the **asymptote** is where the action is
- **Big O Notation**
    - Defines an **Upper Bound** for run time
    - **Order of growth**
    - f(x) ∈ O(x^2)
- The **Average Case** is called **Big Theta** $\boldsymbol{\Theta}$

“Is” versus “In”

- What’s the big difference
- “In” implies the function will take at most O time
- “Is” implies the function will always take O time
    - Upper and **lower bound** are the same
    - **Tightly bound**

Complexity Classes

- Most common instances of Big O
  - O(1) – Constant
  - O(log n) – Logarithmic
  - O(n) – Linear
  - O(n log n) – Log-linear
  - O(n^k) – Polynomial
  - O(c^n) - Exponential

Constant Complexity

- What types of algorithms might have constant complexity?
  - Simple math
  - Some simple graphics
    - Consider drawing a checkerboard

Logarithmic Complexity

- Performance grows a logarithm of one of the inputs
- The base of the log doesn’t matter
  - Simple multiplication can convert

```python
def int_to_str(i):
    """Assumes i is a nonnegative int
    Returns a decimal string representation of i"""
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        i = i//10
    return result
```

Let’s count some steps:

$4 + 6\log(i)$

$O(\log(i))$

$\Theta(\log(i))$

```
def add_digits(n):
   """Assumes n is a nonnegative int
      Returns the sum of the digits in n"""
   string_rep = int_to_str(n)
   val = 0
   for c in string_rep:
       val += int(c)
   return val
```

Let’s count some steps:

$\Theta(\log(i) + \log(i))$

$\Theta(\log(i))$

Linear Complexity

- Usually relates to collections where each element could be touched
    - `add_digits(s)`
        - Bottom portion of `add_digits(n)`
    - Linear search
    - List comprehension

Recursion

```
def factorial(x):
    """Assumes that x is a positive int
       Returns x!"""
    if x == 1:
       return 1
    else:
       return x*factorial(x-1)
```

- O(n)
- Space consumption is also O(n)

Log Linear Complexity

- O(n log(n))
- Typical in sorting algorithms
- Longest Increasing Subsequence
  - Consider: 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15
    - Van der Corput sequence – first 16 elements
    - `0, 8, 12, 14, 15`
    - `0, 4, 12, 14, 15`
    - `0, 4, 10, 13, 15`
    - `0, 2,  6,  9, 13, 15`
    - `0, 2,  6,  9, 11, 15`

    - https://en.wikipedia.org/wiki/Longest_increasing_subsequence

Polynomial Complexity

- Most common is *quadratic* complexity – O(x^2)
- What is the complexity of `is_subset()`?

```
def is_subset(L1, L2):
    """Assumes L1 and L2 are lists.
      Returns True if each element in L1 is also in L2
      and False otherwise."""
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

```
def is_subset2(L1, L2):
    """Assumes L1 and L2 are lists.
    Returns True if each element in L1 is
also in L2
    and False otherwise."""
    for e1 in L1:
        if e1 not in L2:
            return False
    return True
```

O(len(L1)*len(L2))

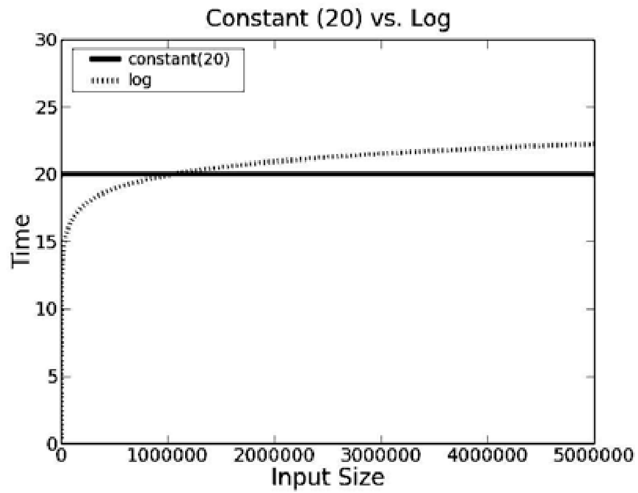
```python
def intersect(L1, L2):
    """Assumes: L1 and L2 are lists
      Returns a list without duplicates that is the intersection of
      L1 and L2"""
    #Build a list containing common elements
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
                break
    #Build a list without duplicates
    result = []
    for e in tmp:
        if e not in result:
            result.append(e)
    return result
```

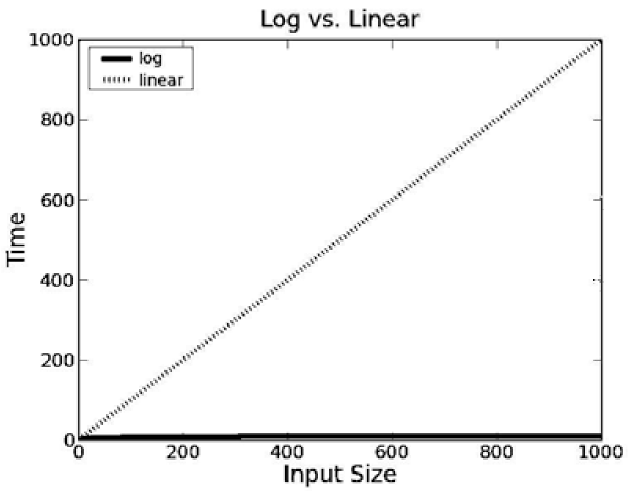
Exponential Complexity

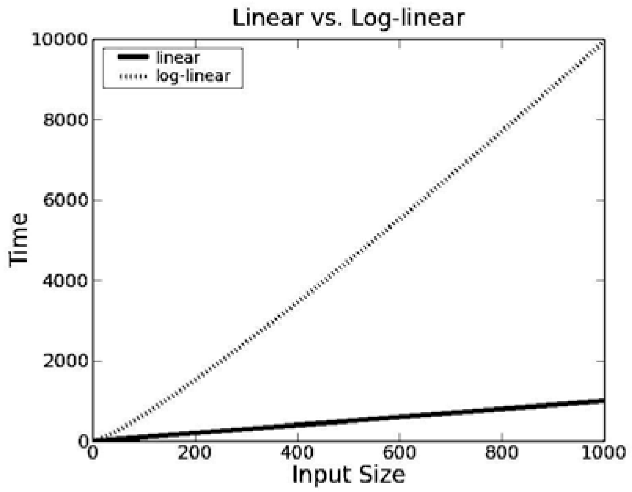
- Run time increases where the size of the input because the *exponent*
- O(c^n)
- What is this code doing?
  - gen_powerset(L)

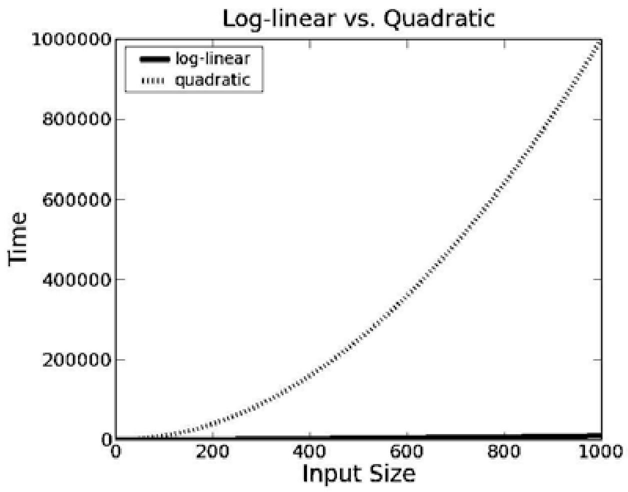
Comparing Complexity

Constant, Log, Linear, Log Linear vs Quadratic (Polynomial)









Comparing Complexity

Quadratic v Exponential