The Art of Data Structures

Runtime Analysis

Alan Beadle
CSC 162: The Art of Data Structures

Acknowledgment:
Slides and course materials originally prepared by Richard E Sarkis for the previous offering of this course
Agenda

• Why algorithm analysis is important
• Use Big-O to describe execution time
• Examine the Big-O execution time of common operations on Python lists and dictionaries
• Understand how the implementation of Python data impacts algorithm analysis
• Benchmark simple Python programs
Algorithm Analysis
Algorithm Analysis

- Given an algorithm can you estimate how much time and memory it will take to process a given amount of data?
Algorithm Analysis

• I have two programs
• They both do the same thing
• I measure their run time
Algorithm Analysis

• prog1 takes 3 sec to process a 1GB array and 3.585 sec to process a 1.5GB array.
• prog2 takes 1 sec to process a 1GB array and 2.25 sec to process a 1.5GB array.

• Which should I use on my 100GB array?
Algorithm Analysis

![Graph showing the running time of two programs vs input size. The graph compares prog1 (logN) and prog2 (cubic). The running time increases significantly with input size for prog2 compared to prog1.](image)
Algorithm Analysis

- This example captures the notion of **asymptotic complexity**
- how much time (and sometimes space) it takes to solve a problem of a given input size
- If you keep taking CS, you'll learn a lot more about this in 173, 280, or 282
Algorithm Analysis

• Sorting is a canonical example
• For lists of length \( N \), we'll look at sorts that take time proportional to \( 2^N \), \( N^2 \), \( N\log(N) \), and \( N \) (in special cases)
  • Note that if \( N \) is big enough, \( 1000000N\log(N) \) is still less than \( N^2 \)
• We will cover sorting later in the course
Algorithm Analysis

- The following solves a familiar problem, computing the sum of the first $n$ integers

```python
def loop_sum(n):
    the_sum = 0
    for i in range(1, n+1):
        the_sum = the_sum + i

    return the_sum
```
import time

def loop_sum(n):
    start = time.time()

    the_sum = 0
    for i in range(1, n+1):
        the_sum = the_sum + i

    end = time.time()

    return the_sum, end-start
def formula_sum(n):
    start = time.time()

    the_sum = (n*(n+1))//2

    end = time.time()
    return the_sum, end - start
Algorithm Analysis

- Let’s evaluate the execution (running) time of these algorithms as a good benchmarking of whether an algorithm is “good”
Algorithm Analysis

- Notice how the time it takes for `loop_sum()` to run when we scale the value of $n$ to larger values
- Compare that with the results for `formula_sum()` using the same scaled values of $n$
- It seems to be unaffected the the size of our input
Algorithm Analysis

- While this is illustrative, it isn’t concretely useful for analysis
- Many factors could affect the performance these same algorithms
  - A different (faster, slower) computer
  - A different language is used
Algorithm Analysis

- A characterization is needed that can describe algorithm performance regardless of these kinds of variability
Big-O
Big-O

- Try to quantify an algorithm on the number of operations, or steps taken
- A basic unit of computation needs to be considered
For example, we can consider assignment statements as a basic unit of computation.

In our function `sum_of_n2`, the number of assignments is 1 plus the value of `n`

Call this $T(n) = 1 + n$

"$T(n)$ is the time it takes to solve a problem of size $n$, name 1+n steps"
Big-O

- The characterization of the algorithm using a $T(n)$ relation shows something interesting
- If the problem size increases, certain terms dominate
The order of magnitude function describes the part of $T(n)$ that increases the fastest as $n$ increases.

**Big-O** notation, written as $O(f(n))$

This is a useful approximation.

$f(n)$ is a simple representation of the dominate part of $T(n)$.
<table>
<thead>
<tr>
<th>Mathematical Expression</th>
<th>Relative Rates of Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(N) = O(F(N))$</td>
<td>Growth of $T(N)$ is $\leq$ growth of $F(N)$.</td>
</tr>
<tr>
<td>$T(N) = \Omega(F(N))$</td>
<td>Growth of $T(N)$ is $\geq$ growth of $F(N)$.</td>
</tr>
<tr>
<td>$T(N) = \Theta(F(N))$</td>
<td>Growth of $T(N)$ is $\approx$ growth of $F(N)$.</td>
</tr>
<tr>
<td>$T(N) = o(F(N))$</td>
<td>Growth of $T(N)$ is $&lt;$ growth of $F(N)$.</td>
</tr>
</tbody>
</table>
• Some examples:

• $10n^2 + 50n + 100$ is $O(n^2)$

• $\log(n)+7$ is $O(\log(n))$

• We only care about the term that increases the fastest, since eventually it will take most of our resources!
• Any Polynomial is big-O of its leading term with coefficient of 1
• The base of a logarithm doesn’t matter.
• \( \log_a(n) \) is \( O(\log_b n) \) for any bases \( a \) and \( b \) because
• \( \log_a(n) = (\log_b n)(\log_a b) \)
• Logs grow slower than powers: \( \log(n) \) is \( O(n^{1/10}) \)

• Exponentials \( (c^n, c>1) \) grow faster than \( \text{poly } n^{10} \) is \( O(1.0001^n) \)

• Generally, polynomial time is tolerable

• Generally, exponential time is intolerable
<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>Constant</td>
</tr>
<tr>
<td>$\log N$</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>$\log^2 N$</td>
<td>Log-squared</td>
</tr>
<tr>
<td>$N$</td>
<td>Linear</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>$N^2$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$N^3$</td>
<td>Cubic</td>
</tr>
<tr>
<td>$2^N$</td>
<td>Exponential</td>
</tr>
</tbody>
</table>
The Tale of Two Algorithms
• Slow Algorithm, Fast Computer
• Fast Algorithm, Slow Computer

DEC Alpha workstation 500/400 (1995)
400 MHz Alpha 21164A

Radio Shack TRS-80 (1977)
1.78 MHz Zilog Z80
## Two Algorithms

<table>
<thead>
<tr>
<th>n</th>
<th>Alpha21164A, “C”, O(n^3)</th>
<th>TRS-80, “BASIC”, O(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.6 microsec</td>
<td>200 millisecs</td>
</tr>
<tr>
<td>100</td>
<td>0.6 millisecs</td>
<td>2.0 sec</td>
</tr>
<tr>
<td>1000</td>
<td>0.6 sec</td>
<td>20 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>10 min</td>
<td>3.2 min</td>
</tr>
<tr>
<td>100,000</td>
<td>7 days</td>
<td>32 min</td>
</tr>
<tr>
<td>1,000,000</td>
<td>19 years</td>
<td>5.4 hrs</td>
</tr>
</tbody>
</table>
Programming Exercise
Programming Exercise

• Write a Python function to convert ints to strings containing the binary representation
  • if (n == 5) return “101”;
  • if (n==12) return “1100”;
• def int2bin( n )
def int2bin(n):
    rval = ""
    while (n>0):
        if((n%2) == 1):
            rval += “1"
        else:
            rval +=“0"
        n = n // 2
    return rval
def int2bin(n):
    rval = ""
    while (n>0):
        if((n%2) == 1):
            rval += "1"
        else:
            rval +="0"
        n = n // 2
    return rval
```python
1  def int2bin(n):
2      rval = ""
3      while (n>0) :
4          if((n%2) == 1) :
5              rval += "1"
6          else :
7              rval +="0"
8          n = n // 2
9
10         return rval
```
def int2bin(n):
    rval = ""
    while (n>0):
        if((n%2) == 1):
            rval += "1"
        else:
            rval +="0"
        n = n // 2
    return rval
def int2bin(n):
    rval = ""
    while (n>0):
        if((n%2) == 1):
            rval += "1"
        else:
            rval +="0"
        n = n // 2
    return rval
def int2bin(n):
    rval = ""
    while (n>0):
        if((n%2) == 1):
            rval += "1"
        else:
            rval +="0"
        n = n // 2
    return rval
def int2bin(n):
    rval = ""
    while (n>0):
        if((n%2) == 1):
            rval += "1"
        else:
            rval +="0"
        n = n // 2
    return rval
def int2bin(n):
    rval = ""
    while (n>0):
        if((n%2) == 1):
            rval += "1"
        else:
            rval +="0"
        n = n // 2
    return rval
def int2bin(n):
    rval = ""
    while (n>0):
        if((n%2) == 1):
            rval += "1"
        else:
            rval +="0"
        n = n // 2
    return rval
• If we have time, discuss anagram examples in textbook

• We have a workshop to do during class, due date is a couple of days from now just in case

• There will also be a new lab assignment on Blackboard after class, due in 1 week
Questions?