CS 260: Foundations Of Computer Science

Class 11 – September 17, 2012

1

Thought for the Day

Life is what happens while you're making other plans.

2

Today's Agenda

- **Project 1 –**
  - Code & final diagrams - Due Wednesday, but grace period through Friday.
- **Exam 1 –Friday, September 21st**
  - Exam 1 will cover material through chapter 3, and some of chapter 4.
- **Read chapter 4 – Algorithms for today.**

3

Algorithm Analysis

Chapter 4

Algorithms

- Algorithms are designed to solve problems.
- A problem can have multiple solutions.

How do we determine which solution is the most efficient?

Execution Time

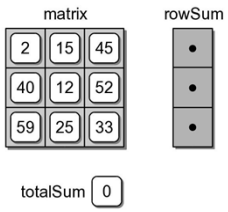
- Measure execution time:
  - construct a program for a given solution.
  - execute the program.
  - time it using a "wall clock".
- Dependent on:
  - amount of data
  - type of hardware and time of day
  - programming language and compiler

Complexity Analysis

- What if we examine the solution itself and measure critical operations:
  - logical comparisons
  - assignments
  - arithmetic operations

Example Algorithm

- Given a matrix of size *n x n*, compute the:
  - sum of each row of a matrix.
  - overall sum of the entire matrix.



Version 1

```
totalSum = 0
   for i in range(n) :
      rowSum[i] = 0
      for j in range(n) :
         rowSum[i] = rowSum[i] + matrix[i,j]
         totalSum = totalSum + matrix[i,j]
```


Version 2

```
totalSum = 0
   for i in range(n) :
      rowSum[i] = 0
      for j in range(n) :
         rowSum[i] = rowSum[i] + matrix[i,j]
      totalSum = totalSum + rowSum[i]
```


Compare the Results

- Number of additions: v_1: $2n^2$ v_2: $n^2 + n$
- Second version has fewer additions (n > 1)
  - Will execute faster than the first.
  - Difference will not be significant.

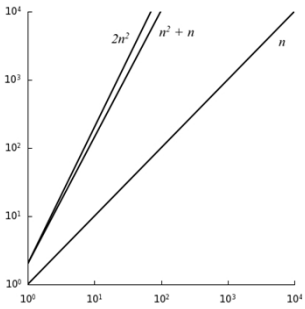
Both algorithms execute on the same order of magnitude, n^2

Growth Rates

- As n increases, both algorithms increase at approx the same rate:

n	$2n^2$	$n^2 + n$
10	200	110
100	20,000	10,100
1000	2,000,000	1,001,000
10,000	200,000,000	100,010,000
100,000	20,000,000,000	10,000,100,000

Growth Rates



Big-O Notation

- No need to count precise number of steps.
- Classify algorithms by order of magnitude.
  - execution time
  - space requirements

Approximates actual number of steps or actual storage in terms of variable-sized data sets.

Big-O Definition

- Given a function *T(n)*
  - # of steps required for an input of size *n*.
  - Ex: $T_2(n) = n^2 + n$
- Suppose there exist a function *f(n)* for all integers $n \geq 0$ such that

$$T(n) \leq c\ f(n)$$

for some constant *c* and for all large values of $n \geq m$ (a constant).

Big-O Definition

- Then, the algorithm has a **time-complexity** of or executes "on the order of" *f(n)*
  - We use the notation: *O(f(n))*
  - Big-O is intended for large values of *n*.

f(n) indicates the rate of growth at which the run time increases as the input size increases.

Big-O Example (v.1)

- Consider the previous sample algorithms.
- Version 1: $T_1(n) = 2n^2$

$T_1(n) \leq c\ f(n)$ Let c = 2

$2n^2 \leq 2n^2$

O(n^2)

Big-O Example (v.2)

- Consider the previous sample algorithms.
- Version 2: $T_2(n) = n^2 + n$

$T_2(n) \leq c\ f(n)$ Let c = 2

$n^2 + n \leq n^2 + n^2$

$n^2 + n \leq 2n^2$

O(n^2)

Upper Bound

- There is more than one *f(n)* for an algorithm.

$$n^2 + n \leq c\ f(n)$$

  - n^2 is not the only choice
  - *f(n)* could be n^2, n^3, n^4

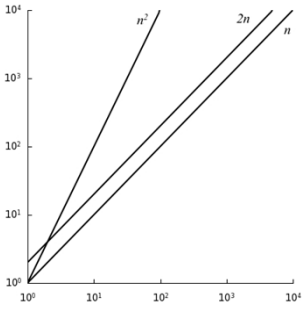
*Objective: find an f(n) that provides the **tightest** (lowest) **upper bound**.*

Constant of Proportionality

- Is it important?
- Consider two algorithms:
  - $O(n^2)$, with c = 1
  - O(2n), with c = 2

n	n^2	2n
10	100	20
100	10,000	200
1000	1,000,000	2,000
10,000	100,000,000	20,000
100,000	10,000,000,000	200,000

Constant of Proportionality



Constructing T(n)

- We don't count total number of specific instructions (math operations, comparisons, etc.)
  - Assume each basic statement takes the same time, **constant time**.
  - Total number of steps required:

$$T(n) = f_1(n) + f_2(n) + \cdots + f_k(n)$$

Constructing T(n)

```
1       totalSum = 0
        for i in range( n ) :
1         rowSum[i] = 0
n
        for j in range( n ) :
n         rowSum[i] = rowSum[i] + matrix[i,j]
          totalSum = totalSum + matrix[i,j]
```

Markup – All operations are marked with appropriate time: 1 or n

Constructing T(n)

```
n     for i in range( n ) :
          ...
n       for j in range( n ) :
            ...
```

Markup – Only the non-constant operations are marked: n

Choosing the Function

- Given *T(n)*, choose the **dominant term**.

$$T(n) = n^2 + \log_2 n + 3n$$

***n²** dominates the other terms (for n ≥ 3)*

$$n^2 + \log_2 n + 3n \leq n^2 + n^2 + n^2$$
$$n^2 + \log_2 n + 3n \leq 3n^2$$

Choosing the Function

- What is the dominant term for the following expression?

$$T(n) = 2n^2 + 15n + 500$$

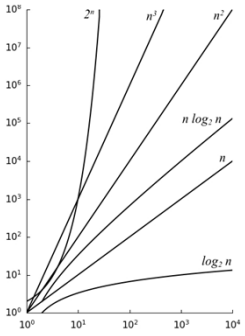
*When n < 16, **500** dominates.*
*When n > 16, **n²** is the dominant term.*

Classes of Algorithms

- Many algorithms have a time-complexity selected from a common set of functions.

f()	Common Name
1	constant
log n	logarithmic
n	linear
n log n	log linear
n^2	quadratic
n^3	cubic
a^n	exponential

Classes of Algorithms



Evaluating Python Code

- Basic operations only require constant time:
  - `x = 5`
  - `z = x + y * 6`
  - `if x > 0 and x < 100`
- What about function calls?

```
y = ex1(n)
```


Code Evaluation #1

```
def ex1( n ):
  count = 0
  for i in range( n ):
    count += i
  return count
```

Code Evaluation #2

```
def ex2( n ):
  count = 0
  for i in range( n ):
    count += 1
  for j in range( n ):
    count += 1
  return count
```


Code Evaluation #3

```
def ex3( n ):
  count = 0
  for i in range( n ):
    for j in range( n ):
      count += 1
  return count
```


Code Evaluation #3b

```
def ex3b( n ):
  count = 0
  for i in range( n ):
    count += ex2( n )
  return count
```


Code Evaluation #4

```
def ex4( n ):
  count = 0
  for i in range( n ):
    for j in range( 25 ):
      count += 1
  return count
```


Code Evaluation #5

```
def ex5( n ):
  count = 0
  for i in range( n ):
    for j in range( i+1 ):
      count += 1
  return count
```


Code Evaluation #6

```
def ex6( n ):
  count = 0
  i = n
  while i >= 1 :
    count += 1
    i = i // 2
  return count
```

Code Evaluation #7

```
def ex7( n ):
  count = 0
  for i in range( n ) :
    count += ex6( n )
  return count
```


Different Cases

- Some algorithms have different run times for different sets of inputs of the same size.
  - best case
  - worst case
  - average case

Different Cases

```
def findNeg( intSeq ):
  n = len( intSeq )
  for i in range( n ) :
    if intSeq[i] < 0 :
      return i
  return None
```

```
L = [ 72, 4, 90, 56, 12, 67, 43, 17, 2, 86, 33 ]
p = findNeg( L )

L = [ -12, 50, 4, 67, 39, 22, 43, 2, 17, 28 ]
p = findNeg( L )
```


The Python List

- We used the list to implement many of our ADTs.
- Their efficiency depends on the efficiency of Python's list.

Python List: Traversal

- Iterates over the contiguous elements of the underlying array.

```
# Sum the elements of a list.
sum = 0
for value in valueList :
  sum = sum + value

# Alternate version.
sum = 0
n = len(valueList)
for i in range( n ) :
  sum = sum + valueList[i]
```


Python List: Allocation

- Creating a non-empty list is not constant.

```
temp = list()
listX = [ 0 ] * n
valueList = [ 4, 8, 20, 2, 15, 89, 60, 75 ]
```

Python List: Appending

- When space is available, the item is stored in the next slot.

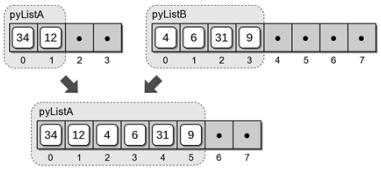
What if the underlying array is full?

Python List: Expansion

- Expanding the underlying array:
  - **Step 1:** create a new array, double the size.
  - **Step 2:** copy the items from original array to the new array.
  - **Step 3:** replace the original array with the new array.
  - **Step 4:** store the new value the next slot of the new array.

Python List: Extending

- Adds the contents of a source list to the end of the destination list.



Python List: Time-Complexities

List Operation	Worst Case
v = list()	O(1)
len(v)	O(1)
v = [0] * n	O(n)
v[i] = x	O(1)
v.append(x)	O(n)
v.extend(w)	O(n)
v.insert(x)	O(n)
v.pop()	O(n)
traversal	O(n)

Amortized Cost

- Consider a sequence of *n* append operations:

```
L = list()
for i in range( 1, n+1 ):
  L.append( i )
```

- What is the worst-case running time?

Special Case

- The `append()` method introduces a special case.
  - available capacity: O(1)
  - expansion required: O(n)

How many times does append require O(n) time?

Amortized Analysis

- Given a sequence of operations, compute the time-complexity by computing the **average cost** over the entire sequence.
  - Cost per operation must be known.
  - Cost must vary, with
    - many ops contributing little cost.
    - only a few ops contributing high cost.

Aggregate Method

- Determine upper bound total cost: T(n)
- Calculate average cost: T(n) / n

- **Example:** sequence of *n* append operations
  - Storage of a single item: O(1)
  - Expansion only occurs when ($i - 1$) is a power of 2.
  - Cost of the expansion based on current array size.

Amortized Cost

- The append() operation:
  - worst-case time: O(n)
  - amortized cost: O(1)

- Can only be used for a long sequence of append operations.