CSCI-UA 480.4: APS
Algorithmic Problem Solving

Fundamentals

Instructor: Joanna Klukowska

created based on materials for this class by
Bowen Yu and materials shared by the authors of
the textbook Steven and Felix Halim

This Course

- Course website: https://cs.nyu.edu/~joannakl/aps_s19/

  This page contains the syllabus and daily summaries as well as loads of links to all other resources and services you will need for this class.

- Recitations (required):
  - Fridays 5:10 - 7:00pm
  - plan to bring your laptop
- Course message board / discussion: Piazza
  - you can self-sign up at https://piazza.com/nyu/spring2019/aps
- Online judge: Vjudge
- Grades posted on NYU Classes
- (Possibly Gradescope - not certain yet)

Operations Count

- How fast your program runs depends on how many things it does

  things == operations that the CPU performs on behalf of the program

- How many things the computer does depends on how you (the programmer) write the code, what algorithm you chose, etc

Operations Count

- How fast your program runs depends on how many things it does

  things == operations that the CPU performs on behalf of the program

- How many things the computer does depends on how you (the programmer) write the code, what algorithm you chose, etc

So, how many operations does this program perform:

Operations Count

- How fast your program runs depends on how many things it does

  things == operations that the CPU performs on behalf of the program

- How many things the computer does depends on how you (the programmer) write the code, what algorithm you chose, etc

So, how many operations does this program perform:

???

Counting operations and not easy (possible)

- hard to figure out what to count, some operations in the high level programming language may be actualy multiple operations on the CPU
- some CPU operations are faster then others
- some operations require I/O and that slows them down

Counting operations and not easy (possible)

- hard to figure out what to count, some operations in the high level programming language may be actualy multiple operations on the CPU
- some CPU operations are faster then others
- some operations require I/O and that slows them down

we need a simplified way of deciding how fast the program runs

Asymptotic Analysis

Asymptotic Analysis

There exist positive constants c and such that for all

Asymptotic Analysis

There exist positive constants c and such that for all

There exist positive constants and such that for all

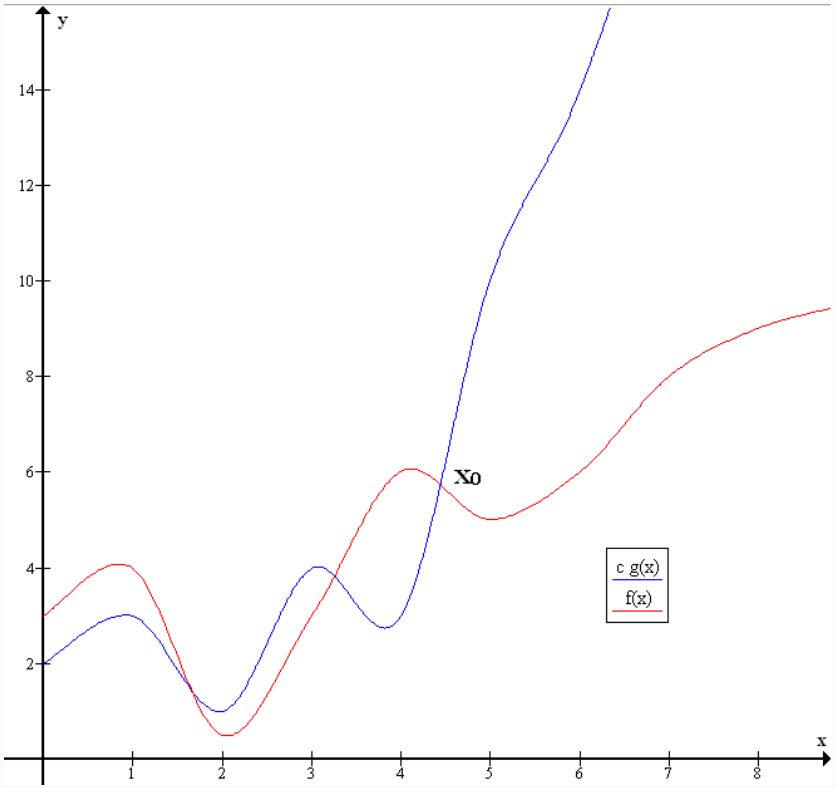
Asymptotic Analysis

There exist positive constants c and such that for all

There exist positive constants and such that for all

There exist positive constants , and such that for all

Asymptotic Analysis: Big O



Example of Big O notation:

$f(x) \in O(g(x))$ as there exists $c > 0$ (e.g., $c = 1$) and $x0$ (e.g., $x0 = 5$) such that $f(x) \leq cg(x)$ whenever $x \geq x0$.

Image and descriptions in Public Domain. Retrieved from Wikipedia: https://commons.wikimedia.org/wiki/File:Big-O-notation.png

Challenge

For each of the following functions state if it is :

-
-
-
-
-
-

"Abuse" of Big O

People often say "The algorithm is - this is slow".

An algorithm that performs in constant time is so the above does not make much sense.

But this is a common abuse of the terminology. We really mean that the algorithm is , even though we use the Big O description.

From Big O to Actual Execution Time

- a typical machine executes approximately operations per second (on average, since some are slower than others)

From Big O to Actual Execution Time

- a typical machine executes approximately operations per second (on average, since some are slower than others)
- to determine how long it might take the program to execute on the largest possible input size (this is specified as part of the constrains for the problem)
    - determine the the Big O term for the maximum N
    - divide the result by

  (this will give you the number of seconds it should take your program to solve the problem - again, this is an approximation, but it gives us the sense of what might happen)

From Big O to Actual Execution Time

Example: check all pairs of value in the input set

- Pwerformance is

From Big O to Actual Execution Time

Example: check all pairs of value in the input set

- Pwerformance is
- when , , time = => program finishes immediately

From Big O to Actual Execution Time

Example: check all pairs of value in the input set

- Pwerformance is
- when , , time = => program finishes immediately
- when , , time = => program finishes in 0.001s (practically instantly)

From Big O to Actual Execution Time

Example: check all pairs of value in the input set

- Pwerformance is
- when , , time = => program finishes immediately
- when , , time = => program finishes in 0.001s (practically instantly)
- when , , time = => program finishes in 10s (too long for the types of problem that we will be looking at)

What's the performance of ...

What's the performance of ...

What's the performance of ...

What's the performance of ...

Challenge

Write a function that checks if a sting Y is a substring of another string X. The function should return or .

Ex.

X = 'abcdaaaabbbbcddd'

- Y = 'ab' => function returns
- Y = 'abab' => function returns

What's the performance of ...

check if a string Y is a substring of another string X

What's the performance of ...

check if a string Y is a substring of another string X

O(X.length * Y.length)

(Note: there exists a more efficient solution.)

What's the performance of ...

check if a string Y is a substring of another string X

O(X.length * Y.length)

(Note: there exists a more efficient solution.) Can we reduce the time further? - constant factor optimization

What's the performance of ...

check if a string Y is a substring of another string X

O(X.length * Y.length)

(Note: there exists a more efficient solution.) Can we reduce the time further? - constant factor optimization

add

Constant factor optimization

- good in practice
- will not make significant improvments for most problems you encounter in this class

Challenge

given a sorted array of values, create a new sorted array that contains only the unique elements

ex.

given array: [1, 1, 4, 5, 5, 5, 6, 7, 7, 9, 9, 9, 9]

new array: [1, 4, 5, 6, 7, 9]

What's the performance of ...

given a sorted array of values, create a new sorted array that contains only the unique elements

What's the performance of ...

given a sorted array of values, create a new sorted array that contains only the unique elements

O(length of oldArray)

(even though we have nested loops in the code)

Rule of Thumb About Complexity

N	worst algorithm to pass on OJ
<= 10	,
<=[15 .. 18]	
<=[18 .. 22]	
<= 100	!!!
<= 400	
<= 2K	
< 10K	!!!
<= 1M	
<= 100M	!!! , ,

!!! but getting close to 10^8 may be dangerous

For examples of algorithms that perform with those complexities, see Table 1.4 in the book. (But do not worry if you are not familiar with all of those algorithms.)

Data Types and Their Representation

Data Types

All data represented as binary in hardware.

Primitive types:

type	size
(in C), (in C/C++)	1 byte
, (in Java)	2 bytes
,	4 bytes
,	8 bytes

Data Types

All data represented as binary in hardware.

Primitive types:

type	size
(in C), (in C/C++)	1 byte
, (in Java)	2 bytes
,	4 bytes
,	8 bytes

String

- not a primitive type
- stored as an array of characters
- WARNING: comparing two strings for equality is NOT constant time

Binary Representation of Integers

- use of base-2
- binary to decimal
  -
  -
- decimal to binary: mod by 2, record the remainder, divide by 2, and finally reverse the resulting string
  -
  -
  -
  -
  -

Use of other bases

Base 3

-

Base 9

-

Base 16

- ...

Largest/Smallest value with N bits

- A type using 4-bits can represent integers to
- **Non-negative numbers**: a type using N-bits can represent values in the range
  - upperbound is NOT included
  - this assumes that we represent only non-negative values
- **Non-negative numbers**: a type using N-bits can represent values in the range
  - upperbound is NOT included
  - the leading bit is associated with a negative multiplier

For signed numbers the limit on the is
(or approximately .

ASCII/UTF-8 and type

- characters are just numbers that use fewer bits (8 bits or 16 bits)
- each character has a corresponding numerical value
  - 'a' = 97
  - 'A' = 65
  - '2' = 50
  - '!' = 33
- this comes in handy when comparing strings or processing characters for other purposes
- but be careful about lexicographical ordering
  - "aaa" < "ab"
  - "AAA" < "aaa"
  - "ZZZ" < "aaa"
  - "10" < "100"
  - "2000" < "30"

Challenge: Split the number

In pairs,

- make sure you understand how the output values are obtained from the input
- create two new test cases (input and output pairs), restriction: they need to be values > 1000