01 B—Algorithm Analysis II

CS1102S: Data Structures and Algorithms

Martin Henz

January 15, 2010

Generated on Friday 15th January, 2010, 09:43

1. Review: Growth of Functions
   - Big Oh and Friends
   - Examples
2. Comparing Running Times
3. Model
4. Running Time Calculations

Motivation

Which functions grows faster?

$f(x) = 1000x$, or $g(x) = x^2$

Motivation

Which functions grows faster?

$f(x) = 1000x$, or $g(x) = x^2$

Intuition

g grows faster than f because *eventually* it will return larger values.

Motivation

Which functions grows faster?

$f(x) = 1000x$, or $g(x) = x^2$

Intuition

g grows faster than f because *eventually* it will return larger values.

No worries about constants

We would like to “overlook” when functions differ only by a constant factor.

Motivation

Which functions grows faster?

$f(x) = 1000x$, or $g(x) = x^2$

Intuition

g grows faster than f because *eventually* it will return larger values.

No worries about constants

We would like to “overlook” when functions differ only by a constant factor.
Example: $f(x) = 1000x$ grows in the same way as $g(x) = 2000x$.

Big Oh!

Definition

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Big Oh!

Definition

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Example

$T(N) = 1000N$
$f(N) = N^2$
$T(N) = O(f(N))$

Big Oh!

Definition

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Example

$T(N) = 1000N$
$f(N) = N^2$
$T(N) = O(f(N))$

Notation

We often simply use the function definitions as in:

$$1000N = O(N^2)$$

Some more definitions

Big Oh

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Some more definitions

Big Oh

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Omega

$T(N) = \Omega(f(N))$ if there are positive constants c and n_0 such that $T(N) \geq cf(N)$ when $N \geq n_0$.

Some more definitions

Theta

$T(N) = \Theta(f(N))$ if and only if $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.

Some more definitions

Theta

$T(N) = \Theta(f(N))$ if and only if $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.

Little oh

$T(N) = o(f(N))$ if for all constants c there exists an n_0 such that $T(N) < cf(N)$ when $N > n_0$. This means: $T(N) = O(f(N))$ and $T(N) \neq \Theta(f(N))$.

Examples

- $1000 = O(1)$

Examples

- $1000 = O(1)$
- $N = \Omega(1000N)$

Examples

- $1000 = O(1)$
- $N = \Omega(1000N)$
- $N = O(N^2)$

Rule 1

If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

- $T_1(N) + T_2(N) = O(f(N) + g(N))$
- $T_1(N) \times T_2(N) = O(f(N) \times g(N))$

Rule 2

If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$.

Rule 3

$\log^k N = O(N)$ for any constant k.

Comparing the Growth of Functions

f grows slower than g

$f(N) = o(g(N))$

$$\lim_{N \to \infty} f(N)/g(N) = 0$$

f grows at the same rate as g

$f(N) = \Theta(g(N))$

$$\lim_{N \to \infty} f(N)/g(N) = c \neq 0$$

f grows faster than g

$g(N) = o(f(N))$

$$\lim_{N \to \infty} f(N)/g(N) = \infty$$

Example

Download a file

After setting up the connection, which takes 3 seconds, the download proceeds at a speed of 1.5Kbytes/second.

Large file sizes

We are interested in the download time $T(N)$ where the file size N grows larger and larger.

Big-Oh

As the file size grows, the initial time of 3 seconds becomes negligible. Thus, $T(N) = O(N)$.

Sequential Computer

- The computers we consider can do only one thing at a time.
- Contrast this with:
  - A computer cluster in SoC
  - The graphics card of your laptop
  - The internet
- Since PCs have a very small number of CPUs, the assumption of sequentiality is still “reasonable”.

Everything costs the same

Simple operations all take constant time

Addition, multiplication, comparison, assignment etc

Integers have fixed-size

The size of integers does not grow as the problem size grows!

A Simple Example: Diagonal Sum

```
// assumption: given a square matrix ''array''
public static int diagonalSum(int[][] array) {
    int len = array.length;
    int sum = 0;
    for (int i=0; i < len; i++) {
        sum += array[i][i];
    }
    return sum;
}
```

Observations

- The initialization of len and sum and return take one unit each.
- The line "for (int i=0; i $<$ len; i++)" takes 1 unit for "int i=0", $N + 1$ units for the tests and N units for the increments.
- To execute the line "sum += array[i][i]" takes four time units: one for each array access, one for the addition, and one for the assignment.
- As the size of the input matrix grows, the time to execute the line "sum += array[i][i]" once, remains 4 units.
- The line is executed N times for a matrix of size N, thus it takes $4N$ time units.
- Overall:
  $T(N) = 3 + 1 + (N + 1) + N + 4 \times N = 6N + 5 = O(N)$

Rule 1

for Loops

The running time of a `for` loop is at most the running time of the statements inside the `for` loop times the number of iterations.

Example

```
for(int j = 0; j < n; j++)
  k++;
```

The runtime is $2 \times N = O(N)$, considering that k++; contains one addition and one assignment.

Rule 2 (from Rule 1)

Nested loops

The running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

Example

```
for(int i = 0; i < n; i++)
  for(int j = 0; j < n; j++)
    k++;
```

The runtime is $2 \times N \times N = O(N^2)$.

Rule 3

Consecutive Statements

The running time of two consecutive statements is the sum of the running times of each component statement.

Example

```
for(i = 0; i < n; i++)
  a[i] = 0;
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    a[i] += a[j] + i + j;
```

The runtime is $2N + 6 \times N \times N = O(N^2)$.

Rule 4

if/else

The running time of `if (condition) S1 else S2` is never more than the running time of the condition plus the larger of the running times of S1 and S2.

Example: Naive Fibonacci

```
public static int fib(int n) {
  if (n <= 1) {
    return n;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

Task

Find runtime $T(N)$ where N is the the given integer.

Critical part: Else

fib(n−1) takes $T(N-1)$ units, and fib(n−2) takes $T(N-2)$ units.

Analysis

Overall

$$T(N) = T(N-1) + T(N-2) + 2$$

Analysis

Overall

$$T(N) = T(N-1) + T(N-2) + 2$$

Compare with *fib* itself

$$fib(N) = fib(N-1) + fib(N-2)$$

Analysis

Overall

$$T(N) = T(N-1) + T(N-2) + 2$$

Compare with *fib* itself

$$fib(N) = fib(N-1) + fib(N-2)$$

Assessment

We can show that $T(N) \geq fib(N)$, and know that $fib(N) < (5/3)^N$. Thus

$$T(N) = O(2^N)$$

Search

Problem

Given an integer X and a sorted collection of integers $A_0, A_1, \ldots, A_{N-1}$, find i such that $A_i = X$, or return $i = -1$ if X is not in the collection.

Search

Problem

Given an integer X and a sorted collection of integers $A_0, A_1, \ldots, A_{N-1}$, find i such that $A_i = X$, or return $i = -1$ if X is not in the collection.

Naive Solution

Scan the collection from $i = 0$ to $N - 1$ and stop when X is found.

Binary Search

```
public static int binarySearch(int [] a, int x) {
  int low = 0, high = a.length - 1;
  while (low <= high) {
    int mid = ( low + high ) / 2;
    if ( a[mid] < x )
      low = mid + 1;
    else if ( a[mid] > x )
      high = mid - 1;
    else return mid;
  }
  return -1;
}
```

Analysis

- The body of the while-loop is $O(1)$.

Analysis

- The body of the while-loop is $O(1)$.
- How often do we go through the loop?

Analysis

- The body of the while-loop is $O(1)$.
- How often do we go through the loop?
- What happens to `high - low` in each iteration?

Analysis

- The body of the while-loop is $O(1)$.
- How often do we go through the loop?
- What happens to `high - low` in each iteration?
- Answer: `high - low` is halved each time.

Analysis

- The body of the while-loop is $O(1)$.
- How often do we go through the loop?
- What happens to `high - low` in each iteration?
- Answer: `high - low` is halved each time.
- Example: Initially, `high - low` $= 128$.

Analysis

- The body of the while-loop is $O(1)$.
- How often do we go through the loop?
- What happens to `high - low` in each iteration?
- Answer: `high - low` is halved each time.
- Example: Initially, `high - low` $= 128$. After each iteration, `high - low` is at most $64, 32, 16, 8, 4, 2, 1, -1$.

Analysis

- The body of the while-loop is $O(1)$.
- How often do we go through the loop?
- What happens to `high - low` in each iteration?
- Answer: `high - low` is halved each time.
- Example: Initially, `high - low` $= 128$. After each iteration, `high - low` is at most $64, 32, 16, 8, 4, 2, 1, -1$. Overall, $T(N) = O(\log N)$

This Evening

- Crash course 2: Loops and Arrays
- Proceed straight to PL1 at 6:30

Next Week

- Crash course 3: Objects, Inheritance
- Crash course 4: Generic Types
- Wednesday lecture: Lists, Stacks, Queues (I)
- Friday lecture: Java Collection Framework