Lecture 08

Running Time Analysis

Lecture 08

Today:

- Counting operations
- Time measurements
- Growth rates
- Introduction to the Big-*O*
- Sequence operations

Analysis of Algorithms

Running time model: Count *all* operations

- elementary operations cost 1 time unit
- focus on the most frequent operation — The barometer instruction
  - count loops and recursion

Running time depends on:

- Size of input
- Currebt size of data structure
- nature of input

Growth Rates

Empirical timings

- run your code on a real machine with various input sizes
- plot a graph to determine the relationship
- but actual performance can depend on much more than just your algorithm!
  - cpu speed
  - amount of main memory
  - os
  - programming language / algorithm implementation
  - . . .

Instead shift focus to the growth rate as a function of N

- usually independent of the above factors

Comparing Algorithm Performance

There can be many ways to solve a problem, i.e., different algorithms that produce the same result

- E.g., There are numerous sorting algorithms.

Compare algorithms by their behaviour as *N* gets large

- on today's hardware, **most** algorithms perform quickly for small N

Analyze behaviour in the worst case

- select the most pessimistic input of size *N*
- yields a saleable upper bound

Can also analyze behaviour in the average case

- use statistics, but need a sensible distribution of inputs

Order Notation (the Big-O)

Suppose we express the number of operations used in our algorithm as a function of N, the size of the problem.

Intuitively …

- take the dominant term
- remove the leading constant; and
- put O(…) around it

E.g., $T(N) = 22\,N^2 + 604\,N + 4519$

Big-O

Given a function $T(N)$, we say $T(N) = O(f(N))$ if $T(N)$ is at most a constant times $f(N)$, except perhaps for some small values of N.

Properties:

$T(N) = O(f(N))$ if and only if

there exists constants c, n > 0 such that for every N > n, T(N) <= c*f(N)

- constant factors dont matter
- low-order terms dont matter

<u>Rule:</u> For any k and any function $f(N)$, $k \cdot f(N) = O(f(N))$

- E.g., $5N = O(N)$
- E.g., $\log_a N = O(\log_b N)$. Why?
- Q. Do leading constants really not matter?

Text

Polylogarithmic Functions

Some generalizations:

1. The powers of N are ordered according to their exponents, i.e.,
   - E.g., $N^2 = O(N^3)$, but N^3 is not O(N^2)
2. A logarithm grows more slowly than any other positive power of N
   - E.g., $\log_2 N = O(N^{1/2})$.

For most functions, can compare them using L'Hôpital's Rule:

- Theorem: If lim f(N)/g(N) n->inf exists then $f(N) = O(g(N))$.

3. A polynomial grows more slowly than any exponential

Typical Growth Rates

- $O(1)$ – constant time
  - The time is independent of N, e.g.,
- $O(\log N)$ – logarithmic time
  - Usually the log is to the base 2, e.g.,
- $O(N)$ – linear time, e.g.,
- $O(N \log N)$ – e.g., qsort, mergesort, heapsort
- $O(N^2)$ – quadratic time, e.g.,
- $O(N^k)$ – polynomial (where k is a constant)
- $O(2^N)$ – exponential time, very slow!

Comparing Implementations

ADTs are a collection of data and operations

- operations are tacitly as efficient as possible
- different data structures may yield different running times
  - *N* is either the current size of the data collection OR,,,
  - … the number of operations thus far

Sequence of `int`	Dynamic Array	Singly Linked List
`Sequence()`	O(1)	
`.get(k)`	O(1)	
`.set(k,x)`	O(1)	
`.getSize()`	O(1)	
`.append()`	O(N)	
`.remove(k)` Text	O(N-k)	
`.trunc(len)`	O(1)	
`~Sequence()`	O(1)	