Applied Algorithmics COMP526

- Lecturer: Leszek Gąsieniec, 321 (Ashton Bldg), L.A.Gasieniec@liverpool.ac.uk
- Lectures: Mondays 4pm (BROD-107), and Tuesdays 3+4pm (BROD-305a)
- Office hours: TBA, 321 (Ashton)
- Assessments (25%) + final exam (75%)
- http://www.csc.liv.ac.uk/~leszek/COMP526/

Algorithm Analysis

- We are interested in the design of "good" *algorithms* and *data structures*
- *Algorithm* is a step-by-step procedure that performs tasks in a finite amount of time
- *Data structure* is a system of fixed rules of how to organize and access stored data

Algorithm Analysis

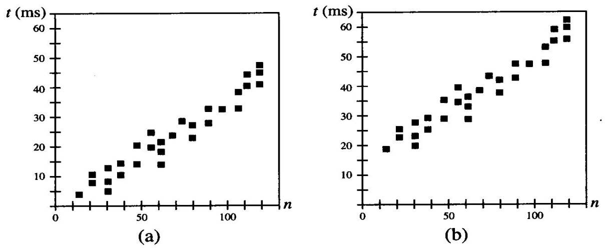
- **Primary interest**: the running time (*time complexity*) of algorithms and operations defined on data structures
- **Secondary interest**: space usage (*space complexity*)
- In more complex models (e.g., distributed systems or networks) we also use other measures, e.g., the number of exchanged messages (*communication complexity*)
- We need some mathematics to describe running times and compare efficiency of algorithms

Algorithm Analysis via experiments

- The main emphasis is on finding *dependency* of the *running time* on the *size of the input*
- In order to determine this, we can perform several well designed *experiments*
- This type of analysis requires a good choice of sample inputs and appropriate number of tests (*statistical certainty*)

Experimental Analysis

- The running time depends on the *size* and the *instance* of the *input* but also the *hardware environment* (lack of universality!)



Experimental Analysis

- Experiments can be performed only on a *limited set of test inputs*
- All experiments should be performed in the *same hardware and software environment*
- The actual *implementation and execution* of the algorithm(s) *is required*

Theoretical Analysis

- Takes into account *all possible inputs*
- Evaluation of relative efficiency of any two algorithms is *independent from hard/software environment*
- Performed by studying *high-level description* of the algorithm

Theoretical Analysis

- This abstract methodology aims at associating with each algorithm a function $f(n)$ that characterizes (provides as accurate as possible bounds on) the running time of the algorithm in terms of the input size n
- Typical functions include n, n^2, $n \cdot \log n$, …

Theoretical Analysis requires

- A formal *language* for describing algorithms
- A *computational model* in which considered algorithms are analysed and compared.
- An accurate *metric* for measuring algorithm running time, space usage, communication, …
- An *approach* for characterizing running times

Pseudo-Code

- Description of algorithms that is formal but *for human eyes only*
- Description of algorithms that is *more structured* than regular prose
- Description that *facilitates* the *high-level analysis* of a data structures and algorithms

Pseudo-Code example

- Finding the maximum element

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.

$currentMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currentMax < A[i]$ **then**
    $currentMax \leftarrow A[i]$
return $currentMax$

Pseudo-Code

- Pseudo-code is a mixture of natural languages and high-level programming (*Ada*, *Pascal, C++, Java* like) constructs
- Pseudo-code describes the main ideas behind generic implementation of data structures and algorithms
- Pseudo-code constructions include: *expressions, declarations, decision structures, loops, arrays, methods of calls, …*

Computational Model

- Set of high-level *primitive operations* that can be found in the pseudo-code includes: *assigning a value, calling method, performing an arithmetic operation, comparing two numbers, array indexing, following object reference, returning from a method*
- *Time complexity* refers to *counting* the number of primitive operations that are executed

Random Access Machine (RAM)

- CPU connected to a bank of *memory cells*
- Each memory cell can store a *number*, a *character string*, or an *address*, i.e., the value of a base type
- We assume that any primitive operation can be performed in *constant time*

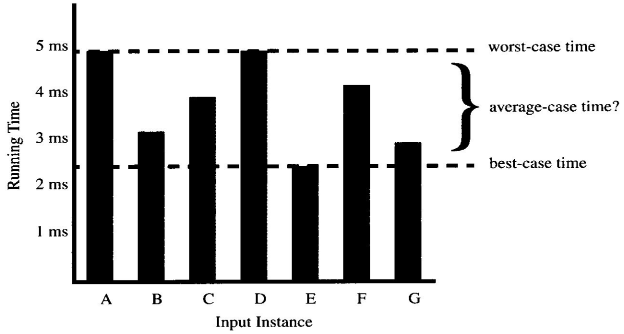
Counting Primitive Operations

	COST
Algorithm arrayMax(A,n):	
Input: An array A storing $n \geq 1$ integers.	
Output: The maximum element in A.	
$currentMax \leftarrow A[0]$	1
for $i \leftarrow 1$ **to** $n-1$ **do**	n
if $currentMax < A[i]$ **then**	n-1
$currentMax \leftarrow A[i]$	≤n-1
return $currentMax$	1
TOTAL	~3n

Average vs. Worst Case

- Algorithm may run faster on some inputs and slower on the others
- *Average case* refers to the running time of an algorithm as an *average taken over all inputs* of the same size
- *Worst case* refers to the running time of an algorithm as the *maximum taken over all inputs* of the same size

Worst vs. Average Case

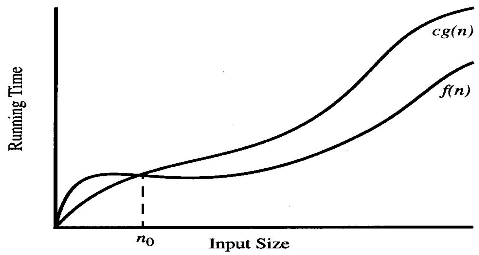


Asymptotic Notation

- *Asymptotic notation* allows to characterize the *main factors* (computational components) affecting an algorithm's running time
- It also allows a *simplified analysis* that estimates the number of primitive operations executed *up to a constant factor*

"Big-Oh" Notation

- $f(n)$, $g(n)$ positive integer functions
- We say $f(n)$ is $O(g(n))$ if there is real constant c, s.t., $f(n) < c\ g(n)$, for $n > n_0$.



Example: $7n - 2 = O(n)$

- We have to find constants c and n_0, s.t., $7n - 2 < c\ n$, for all $n > n_0$
- A possible choice is $c = 7$ and $n_0 = 1$
- In fact this is one of infinitely many possible choices, because any real number $c > 7$ and any integer $n_0 > 1$ would serve as well

Further Examples

- $20\, n^3 + 10\, n \log n + 5$ is $O(n^3)$, and any polynomial $a_k n^k + \ldots + a_1 n^1 + a_0$ is $O(n^k)$
- $3 \log n + \log\log n$ is $O(\log n)$
- 2^{100} is $O(1)$
- $5/n$ is $O(1/n)$

Frequent Functions

- *Logarithmic* $O(\log n)$
- *Linear* $O(n)$
- *Quadratic* $O(n^2)$
- *Polynomial* $O(n^k)$, $k > 2$
- *Exponential* $O(a^n)$, $a > 1$

Relatives of "Big-Oh"

- $f(n)$, $g(n)$ positive integer functions
- We say that $f(n)$ is $\Omega(g(n))$ (big-Omega) if there is a real const. c , s.t., $f(n) > c\, g(n)$, for $n > n_0$.
- We say that $f(n)$ is $\Theta(g(n))$ (big-Theta) if $f(n)$ is $\Omega(g(n))$ and $f(n)$ is $O(g(n))$

Two Examples

- $\frac{3}{4} \cdot \log n + \log\log n$ is $\Omega(\log n)$
- $3 \log n + \log\log n$ is $\Theta(\log n)$
- More examples will be studied at practicals

Importance of Asymptotics

- The maximum size allowed for an input instance for various running times to be solved in 1*sec*, 1*min* and 1*h*

Running Time	Maximum Problem Size (n) 1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$20n\lceil\log n\rceil$	4,096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

Growth Rate (running time)

- Functions ordered by growth rate: $\log n$, $\log^2 n$, $n^{1/2}$, n, $n\cdot\log n$, n^2, n^3, 2^n

n	$\log n$	$\sqrt{n}$	n	$n\log n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2.8	8	24	64	512	256
16	4	4	16	64	256	4,096	65,536
32	5	5.7	32	160	1,024	32,768	4,294,967,296
64	6	8	64	384	4,096	262,144	1.84×10^{19}
128	7	11	128	896	16,384	2,097,152	3.40×10^{38}
256	8	16	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	23	512	4,608	262,144	134,217,728	1.34×10^{154}
1,024	10	32	1,024	10,240	1,048,576	1,073,741,824	1.79×10^{308}

Typical Justification Techniques

- Proof by *counter-example*
  - negative, frequently used in testing
- Proof by *contra-positive* argument
  - uses observation: (A => B) ≡ (~B => ~A)
- Proof by *contradiction*
  - uses observation: (A => B) ≡ ((A and ~B) => ~A)
- Proof by *mathematical induction*
  - direct, used to justify iterative and recursive solutions

Mathematical Induction

- Used to prove some property *P(i)* of analysed program for *all* (or *arbitrarily large*) integers
- The proof is performed as follows:
  - Prove *P(·)* for some small integer, e.g., *P(1)*
  - Prove that *P(1)*,.., *P(i-1)* plus all other knowledge we have imply *P(i)*
  - Then it follows that *P(i)* holds for all integers *i*

Loop Invariant Method

- Used to prove correctness of loops
- The proof is performed as follows:
  - Find an invariant *I(i)* for considered loop, where *i* is the loop index iterated from *1* to *F(i)*
  - Prove that *I(1)* holds just before the 1st loop test
  - Prove that from *I(1)*,.., *I(i-1)* and the content of the loop it follows that *I(i)* holds too
  - Show that from "stop condition" *F(i)* and invariant *I(i)* we get the desired solution *S(i)*

Example of Invariant Method

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

currentMax $\leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if *currentMax* $< A[i]$ **then**
    currentMax $\leftarrow A[i]$
return *currentMax*

Stop condition
F(i) ≡ (i=n)
and
Loop invariant
I(i) ≡ currentMax ≥ A[0],..,A[i-1]
imply
Desired solution
S(n) ≡ currentMax ≥ A[0],..,A[n-1]

I.e., *currentMax* contains the maximum element in *A*

Decreasing Function Method

- Mainly used to prove that a loop stops
- The proof is preformed as follows
  - Find a positive (potential) function *f(i)*, where *i* is the loop index iterated from *1* to *F(i)* with bounded original value
  - Show that each iteration of the loop decreases the value of *f(i)* but it always remains positive
  - This shows that the number of loop iterations is bounded and that the loop stops eventually

Example of Decreasing Function

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

currentMax $\leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if *currentMax* $< A[i]$ **then**
    currentMax $\leftarrow A[i]$
return *currentMax*

Stop condition
F(i) ≡ (i=n)
and
Decreasing function
f(i) = (n+1-i)

During each iteration *f(i)* is decreased by *1*

But it always remains positive with *f(i=n)=1*