Algorithm Design & Analysis: Efficiency

Dr. Dave Parker

d.a.parker@cs.bham.ac.uk

2016/17

Module overview

- Part 1 (Iain Styles): numbers & computer organisation
  - number representations
  - computer architecture, instruction sets
- Part 2 (Ata Kaban): the Java Virtual Machine
  - compilation, interpretation, virtual machines
  - subroutines, stacks and expression evaluation
  - JVM: frames, variables, bytecode, methods, objects
- Part 3 (Dave Parker): algorithm design & analysis
  - efficiency: time and space complexity
  - correctness: errors, invariants, recursion

Module overview

- Continuous assessment remaining

- Assessed quiz 2 (on parts 2, 3)
  - week 11 (due 9 Dec)

- Reflection exercise (MSc only)
  - due Fri 2 Dec (week 10)

Overview: Efficiency

- Algorithm design and analysis
- Efficiency
- Time complexity
- Big-O notation
- Examples

- (Books)

Algorithm design and analysis

- Multiple algorithms often exist for the same task
  - how do we select the best one?
- Many possible (and often conflicting) criteria
  - efficiency
  - simplicity, clarity
  - elegance, proofs of correctness
- We also need to ask
  - is my algorithm correct?
  - does my algorithm always terminate?
  - does an algorithm even exist?

Efficiency

- Resource usage of an algorithm
  - typically: time (runtime) and space (computer memory)
  - also: network usage, hardware requirements, …
  - often, we consider trade-offs between resources

- How do we measure the runtime of an algorithm?
  - benchmarking on representative set of inputs
  - analyse the (time) complexity

Time complexity

- Time complexity:
  - the number of operations that an algorithm requires to execute, in terms of the size of the input or problem
- Note:
  - "algorithm", not implementation (so: pseudocode; no fixed programming language, computer architecture)
  - "in terms of" – complexity defined as a function T(n)
- Questions:
  - what do mean by "operations"?
  - what do we mean by "size"?
- We focus on worst-case, not average-case, analysis

Example

- Example: Look up a value v in an array x of integers

1	4	17	3	90	79	4	6	81

- Algorithm: linear search
  - inputs: array x of size n, integer v
  - return: index of first occurrence of v in x, or -1 if none

- T(n) = n

```
for i=0...n-1:
    if x[i] == v:
        return i
return -1
```

Example

- Example: matrix-vector multiplication: $x = A\,b$
  - n x n matrix A, vector of b size n
- Algorithm:
  - inputs: matrix A, vector b
  - result stored in vector x (initially all 0)

```
for i=0...n-1:
    for j=0...n-1:
        x[i] = x[i] + A[i][j] * b[j]
```

- $T(n) = 2n^2$

Big-O notation

- Usually don't need exact complexity T(n)
  - it suffices to know the complexity class
  - we ignore constant factors/overheads, lower orders
  - focus on performance for large n ("asymptotic")
- Big-O notation
  - for example: $O(n^2)$ "O of n squared" "on the order of n^2"
- Examples
  - $T(n) = n \implies$ complexity class = $O(n)$
  - $T(n) = n+2 \implies$ complexity class = $O(n)$
  - $T(n) = 2n^2 \implies$ complexity class = $O(n^2)$

Big-O notation

- More examples
  - $T(n) = 10n^3 + 1 \implies$ complexity class $= O(n^3)$
  - $T(n) = 5(n+2) \implies$ complexity class $= O(n)$
  - $T(n) = 1000 \implies$ complexity class $= O(1)$
  - $T(n) = n^2+n+1 \implies$ complexity class $= O(n^2)$

- Determining the complexity class
  - intuitively, it suffices to count the number of loops and the number of times they are executed

Big-O notation: Common classes

- Some common complexity classes:
  - $O(1)$ = "constant"
  - $O(\log_2 n)$ = "logarithmic"
  - $O(n)$ = "linear"
  - $O(n^2)$ = "quadratic"
  - $O(n^3)$ = "cubic"
  - $O(2^n)$ = "exponential"

- Polynomial: $O(n)$, $O(n^2)$, $O(n^3)$, … – "tractable"
- Exponential: $O(2^n)$, $O(c^n)$ – "intractable"

Some concrete numbers…

- How many operations needed for an algorithm?
  - with complexity T(n) = f(n) and input size n

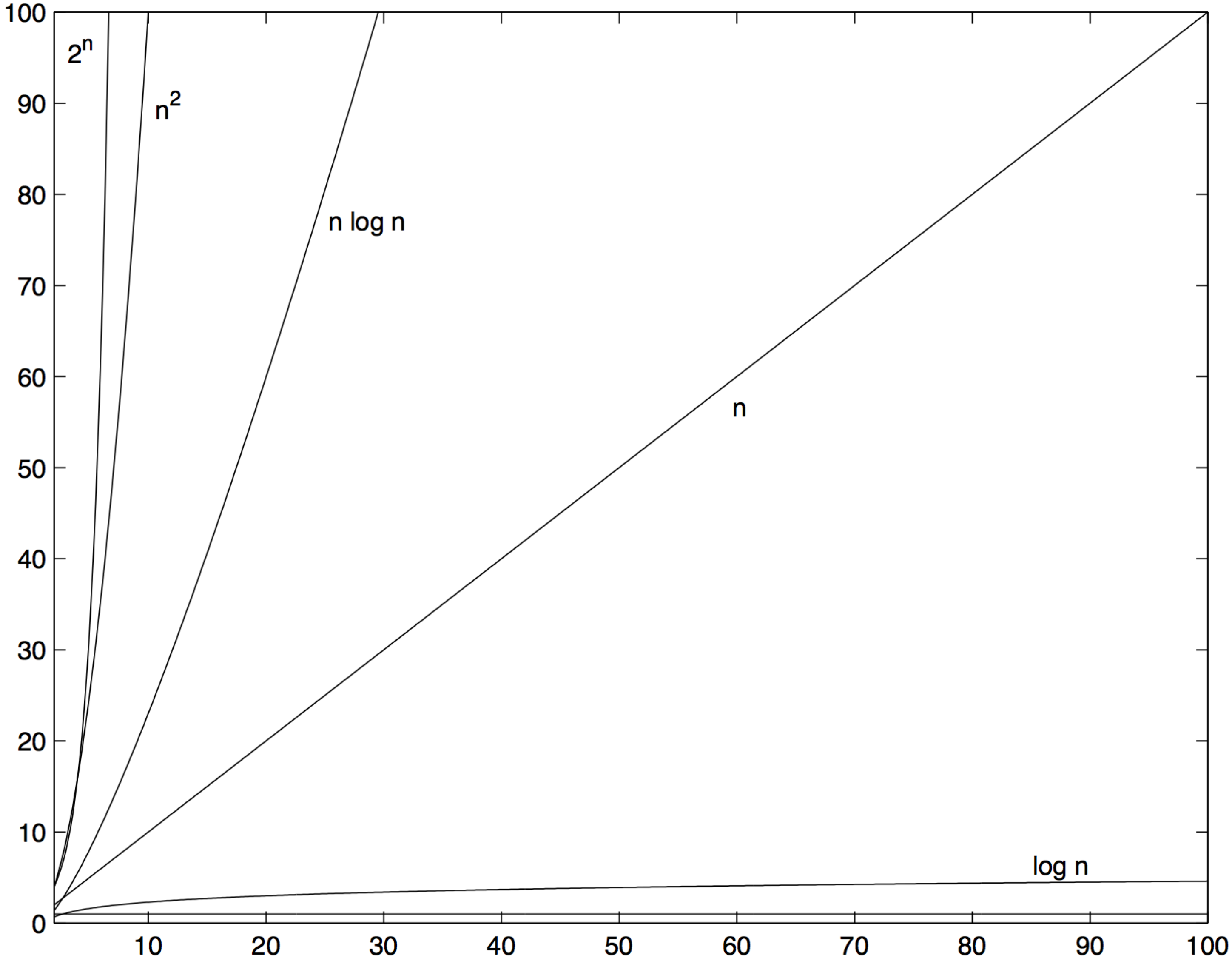
$f(n)$	$n = 4$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1	1	1	1.00×10^0	1.00×10^0
$\log_2 \log_2 n$	1	2	3	3.32×10^0	4.32×10^0
$\log_2 n$	2	4	8	1.00×10^1	2.00×10^1
n	4	16	2.56×10^2	1.02×10^3	1.05×10^6
$n \log_2 n$	8	64	2.05×10^3	1.02×10^4	2.10×10^7
n^2	16	256	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	64	410	1.68×10^7	1.07×10^9	1.15×10^{18}
2^n	256	65536	1.16×10^{77}	1.80×10^{308}	6.74×10^{315652}

Some concrete numbers…

- How much time needed for an algorithm?
  - assuming 1 million operations per second

$f(n)$	$n = 4$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1 μsec	1 μsec	1 μsec	1 μsec	1 μsec
$\log_2 \log_2 n$	1 μsec	2 μsec	3 μsec	3.32 μsec	4.32 μsec
$\log_2 n$	2 μsec	4 μsec	8 μsec	10 μsec	20 μsec
n	4 μsec	16 μsec	256 μsec	1.02 msec	1.05 sec
$n \log_2 n$	8 μsec	64 μsec	2.05 msec	1.02 msec	21 sec
n^2	16 μsec	256 μsec	65.5 msec	1.05 sec	1.8 wk
n^3	64 μsec	4.1 msec	16.8 sec	17.9 min	36,559 yr
2^n	256 μsec	65.5 msec	3.7×10^{63} yr	5.7×10^{294} yr	2.1×10^{315639} yr

Plots



100
90
80
70
60
50
40
30
20
10
0
10
20
30
40
50
60
70
80
90
100
2^n
n^2
n log n
n
log n

Example

- Example: Look up a value v in an array x

1	4	17	3	90	79	4	6	81

Example

- Example: Look up a value v in a sorted array x

1	3	4	4	6	17	79	81	90

Example

- Example: Look up a value v in a sorted array x

1	3	4	4	6	17	79	81	90

- Algorithm: binary search
  - inputs: sorted array x of size n, integer v
  - return: index of first occurrence of v in x, or -1 if none
- Complexity = $O(\log_2 n)$

```
left = 0; right = n-1
while left < right:
    mid = (left+right)/2
    if x[mid] < v:
        left = mid+1
    else:
        right = mid
if x[left] == v:
    return left
else:
    return -1
```

Computing complexity classes

- Determine total algorithm complexity
  - from the complexities of its components
- 1. Sequential algorithm phases
- 2. Function/method calls

Sequential phases

- Example: matrix-vector multiplication: $x = A\,b$
  - matrix-vector multiplication $x = A\,b$
  - initialise vector x (all 0), then add values to x

```
for i=0...n-1:
    x[i] = 0
for i=0...n-1:
    for j=0...n-1:
        x[i] = x[i] + A[i][j] * b[j]
```

- Complexity: $O(n) + O(n^2) = O(n^2)$
- In general: "maximum" of complexities

Functions/methods

- Example: n array look-ups

```
for i=0...n-1:
    binary_search(x,vᵢ)
```

- Complexity $= O(n) \times O(\log n) = O(n \log n)$
- In general: "multiply" complexities

Computing complexity classes

- Determine total algorithm complexity
  - from the complexities of its components
- Sequential algorithm phases: "maximum"
  - e.g. $O(n) + O(n^2) = O(n^2)$
  - e.g. $O(n) + O(\log n) = O(n)$
- Function/method calls: "multiply"
  - e.g. $O(n) \times O(\log n) = O(n \log n)$
  - e.g. $O(n^2) \times O(1) = O(n^2)$

Some harder problems

- Travelling salesman problem (TSP)
  - given n cities and the distances between them, what is the shortest possible route that visits each city exactly once and then returns to the first city?
- Boolean satisfiability problem (SAT)
  - given a formula f in propositional logic over n variables, is there a valuation of the variables that makes f true?
- In both cases:
  - lots of practical applications
  - also important problems in theoretical computer science

Algorithms vs. problems

- Algorithm complexity
  - worst-case run-time of algorithm – "efficiency"
  - actually an upper bound: algorithm A is in O(f(n)) if the worst-case run-time is at most f(n)
  - usually use tightest (most informative) complexity
- Problem complexity
  - complexity class = set of problems
  - problem X is in complexity class O(f(n)) if there exists an algorithm to solve it in O(f(n)) – "difficulty"
  - again: this is an upper bound (and use tightest possible)
  - sometimes consider lower bounds: e.g. sorting: O(n log n)

P, NP, …

- Complexity classes
  - $O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(2^n)$ …
  - polynomial time (PTIME or P) – assumed to be "tractable"
- Another famous class: NP
  - if we can "guess" a solution, it can be checked efficiently (efficiently = in polynomial time)
- NP-hard problems
  - e.g. travelling salesman problem, SAT, …
  - only exponential time algorithms are known
  - but some efficient heuristics exist in practice
- P=NP? Nobody knows…

Summary

- Algorithm design and analysis:
  - efficiency
- Time complexity
  - (worst-case) number of operations an algorithm needs to execute, in terms of the size of the input or problem: T(n)
- Big-O notation
  - complexity classes: O(n), (n^2), $O(2^n)$, …
  - focus on large values of n (i.e. asymptotic behaviour)
  - ignore constants, lower factors
  - count loops/iterations, decompose algorithm
  - complexities for algorithms/problems

Books

- Goldschlager and Lister: *Computer Science: a Modern Introduction* (Prentice Hall, 2nd edition 1988)
  – chapter 3
  – copy in School library
- Aho & Ullman: *Foundations of Computer Science* (Freeman, 1992)
  – chapter 3
  – copy in School library & freely available online: http://infolab.stanford.edu/~ullman/focs.html

- Credits: Some material here taken from "Foundations of Computer Science" notes, by John Bullinaria, Manfred Kerber & Martin Escardo