COMP2054-ADE
Algorithms Data Structures & Efficiency

ADE Lec01
Analysis of Algorithms

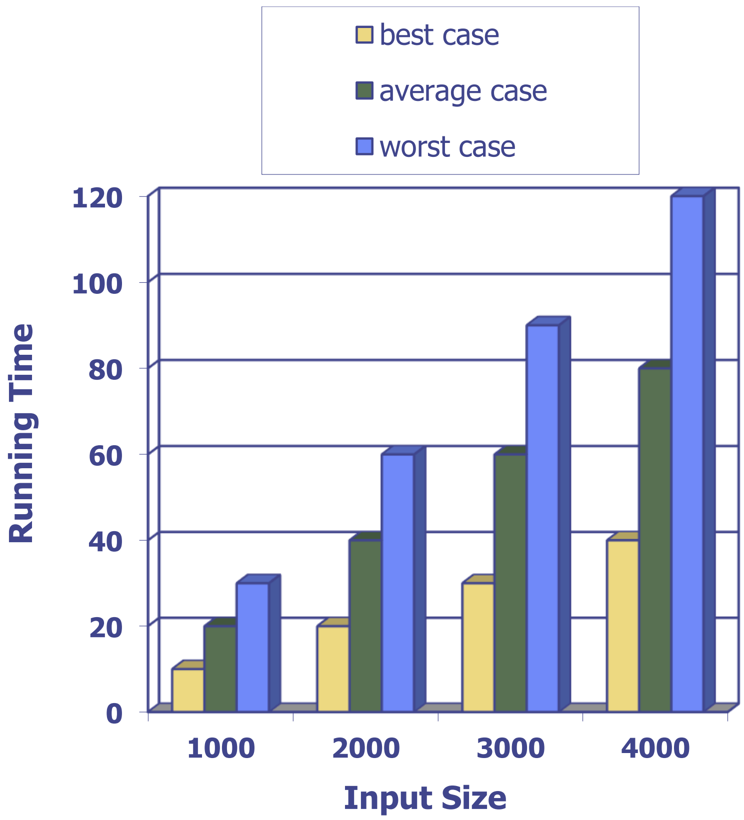
Lecturer: Andrew Parkes

Email: andrew.parkes@nottingham.ac.uk

http://www.cs.nott.ac.uk/~pszajp/

Running Time: "finite" but how big?

- Consider "batch algorithms": transform input data into output data – as opposed to "interactive"
- The running time of an algorithm typically grows with the input size.
- Even at given size the runtime is usually not fixed.
  - So have "best", "average" and "worst" cases.
  - A typical example →

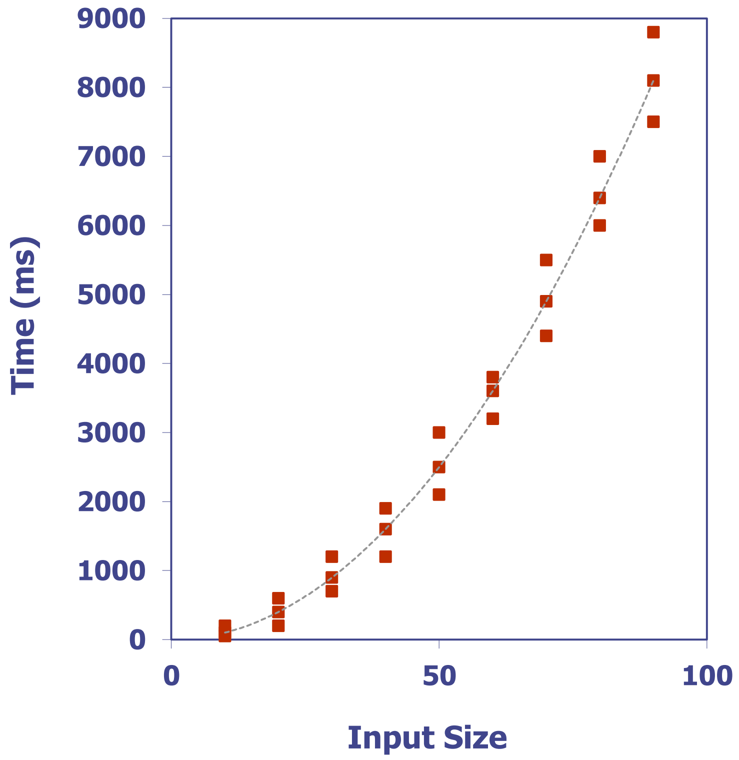


- We (usually) focus on the worst case running time at given size
  - Useful, and easier to analyse
  - Average case time is often difficult to determine.

Experimental Studies

General Pattern:

- Write a program implementing the algorithm
- Run the program with inputs of "varying size and composition"
- Use a system method to get an (in)accurate measure of the actual running time
- Plot the results
  - Example is shown →
- Interpret & analyse. E.g. is it
  - A "power law", n^k, for some k
  - An "exponential", b^n, for some b



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult or time-consuming
- **Results may not be indicative of the running time on other inputs not included in the experiment.**
  - **Maybe we miss the “real worst case”**
- In order to compare two algorithms directly, the same hardware and software environments must be used

Limitations of Theory

- It is necessary to implement the theory, which may be difficult or time-consuming
- Results may not be indicative of the typical running time on inputs encountered in real world.

So can be useful to be able to use both experiment and theory.

Aside: Theory vs. Experiment

Standard science:
"Never believe a theory until it has been `confirmed' by an experiment"

Partially joking:
"Never believe an experiment until it has been confirmed by a theory"

- **Attributed to Sir Arthur Eddington https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3597502/**

Theoretical Analysis

- **AIM: Characterise running time as a function of the input size, *n*.**
- Uses a "high-level" description of the algorithm instead of an implementation
  - Takes into account all possible inputs
  - Allows us to evaluate the speed of an algorithm independently of the hardware/software/language environment

Pseudocode (recap)

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm ***arrayMax***(***A***, ***n***)
  Input array ***A*** of ***n*** integers
  Output maximum element of ***A***

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

Pseudocode Details (recap)

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm *method* (*arg* [, *arg*…])

  Input …

  Output …

- Method call

  var.method (*arg* [, *arg*…])

- Return value

  return *expression*

- Expressions

  $\leftarrow$ Assignment (like $=$ in Java)

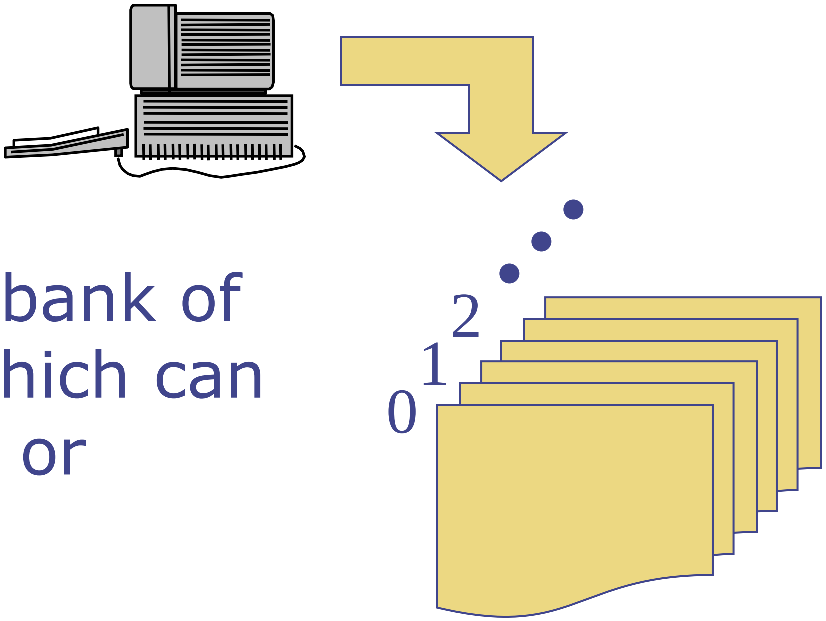
  $=$ Equality testing (like $==$ in Java)

  n^2 Superscripts and other mathematical formatting allowed

Primitive Operations

- Basic computations performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the "RAM model" (next slide)
  - Tend to be close to "Assembly language"
  - No "hidden expenses"
- Examples:
  - Assigning a value to a variable
  - Indexing into an array
  - Comparing two numbers
  - Adding/subtracting/multiplying/dividing two numbers
  - Calling a method
  - Returning from a method

The Random Access Machine (RAM) Model



- A **CPU**
- A potentially-unbounded bank of **memory** cells, each of which can hold an arbitrary number or character

- Memory cells are numbered and accessing any cell in memory takes unit time (some fixed time).
  - (Note that RAM can stand for both "Random Access Machine" and "Random Access Memory," Which is an unfortunate, but standard, over-loading of terminology.)

Limitations of RAM model

- "… can hold an arbitrary number …" ?
  - Can we really expect to store "93856635928615180035166617773577777771771773747177174715717777761365661618161616" in one cell on a real computer?
- Here, we ignore such "bignum" issues. Instead:
- "all numbers are of equal size, as they all fit in a single register of the CPU"
- 64bits (signed int) allows up to 9,223,372,036,854,775,807
  - Exercise (offline): compare to: nanoseconds since big bang; national debt.
- Note: on real machines (usually) computing

  1 + 1 takes as long as 381513 + 243542

  Hence, we typically ignore the sizes of numbers in the arithmetic operations.

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	?
for $i \leftarrow 1$ **to** $n - 1$ **do**	?
if $A[i] > currentMax$ **then**	?
$currentMax \leftarrow A[i]$	?
return $currentMax$	?
Total	**EXERCISE "Try it!"**

Counting Primitive Operations (partial) – "Offline -> Pause"

- **Worst case** number of primitive operations executed as a function of the input size, n

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	?
for $i \leftarrow 1$ **to** $n - 1$ **do**	?
if $A[i] > currentMax$ **then**	?
$currentMax \leftarrow A[i]$	$2\,(n - 1)$
	?
return $currentMax$	?
Total	??

Counting Primitive Operations (partial)

- Worst case number of primitive operations executed as a function of the input size, n

Algorithm *arrayMax*(A, n)	# operations
$\quad currentMax \leftarrow A[0]$	?
$\quad$ **for** $i \leftarrow 1$ **to** $n - 1$ **do**	1 // for $i \leftarrow 1$
$\qquad$ **if** $A[i] > currentMax$ **then**	$2\,(n - 1)$
$\qquad\qquad currentMax \leftarrow A[i]$	??
$\qquad$ **return** *currentMax*	?
Total	??

Counting Primitive Operations (all)

- Worst case number of primitive operations executed as a function of the input size, n

	# operations
Algorithm *arrayMax*(A, n)	
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	1
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$ (worst case)
{ increment counter: i++ }	$2(n - 1)$ ("hidden")
{ test counter: $i \le (n-1)$ }	$2(n - 1)$ ("hidden")
return $currentMax$	1
Total	$8n - 4$

Counting is "underspecified"

- Consider "$c \leftarrow A[i]$" then 'full' process can be
  - get A = pointer to start of array A, and store into a register
  - get i, and store into a register
  - compute A+i = pointer to location of A[i], and store back into a register
  - get value of "*(A+i)" (from RAM) and store value it into a register
  - copy the value into the location of c in the RAM
- might not want to count all this, e.g. just count
  - 'plus' of "A+i"
  - the assignment

Counting is "underspecified"

- There can be multiple right answers – if you get '2' and I count '4' then it does not mean you are wrong!
- Note: If I think an answer is '4' then '2' is probably also acceptable – but "2 n" probably will not be.
- It is most important to be able to
  - know what is happening in the underlying process
    - be able to link to C and assembly level notions
  - be able to use this to give a reasonably consistent justification of your answers

Note: Correctness vs. Efficiency

- Primitive operation counting is relevant to Efficiency – but not (directly) for Correctness
- For correctness, do not about runtime of the alg.
  - Do care about the time to find a proof if doing an automated search for proofs.
  - The verification seems to be quick in lean: e.g.

    #eval 2035713999 + 350135299
  - Very quick (<1sec) – not using succ internally (?)
- Note that we did not prove the algorithm correct
  - Would need to do arrays/lists in Lean first
    - Just for thought: And then what?
  - Can do efficiency of incorrect algorithms ☺

Estimating Running Time

- Algorithm ***arrayMax*** executes $8n - 4$ primitive operations in the worst case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

- Let $T(n)$ be worst-case time of ***arrayMax*.** Then

$$a\,(8n - 4) \le T(n) \le b(8n - 4)$$

- Hence, $T(n)$ is bounded 'above and below' by two linear functions
- Usually said as "*arrayMax* runs in linear time"

Remarks

- Do not get too obsessed with the fine details of counting of primitive operations
- The details of the counting and timing would probably depend
  - the compiler, and require inspection of the assembly code
  - the CPU architecture, pipelining, cache misses, etc, etc

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- **The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax***

Exercise: (exam-style question)

Given the following code fragment:

```
m ← 0
while (n ≥ 2)
    n ← n/2
    m++
return m
```

Give an analysis of its runtime

Exercise: what is T(n) of alg-lec1?

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2.

I.e. there exists k such that $2^k = n$

m ← 0

while (n ≥ 2)

n ← n/2

m++

return m

Exercise: what is T(n) of alg-lec1? (cont)

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2

Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	?
n ← n/2	?
m++	?
return m	1

Exercise

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2

Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	? per pass
n ← n/2	? per pass
m++	? per pass
return m	1

(Pause and try/think)

Internal Steps:

$\boldsymbol{n \leftarrow n/2}$?

1. read ***n*** from memory (RAM) and store in a register r1 (very fast piece of memory on the CPU)
2. read ***2*** from memory and store in a register r2
3. send registers r1 r2 through arithmetic division and store result in a register r3
4. write r3 back to ***n***

CPU steps needed is 4, does not depend on ***n***

Internal Steps: different compiler

$\boldsymbol{n \leftarrow n/2}$?

1. read ***n*** from memory (RAM) and store in a register r1 (very fast piece of memory on the CPU)
2. send registers r1 through a right shift of the bits and store result in a register r3
   e.g. compute 13/2=6 by 1101 → 110
3. write r3 back to ***n***

CPU steps needed is 3, different but still does not depend on ***n***

Exercise (cont)

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2

Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	3 per pass
n ← n/2	3 per pass
m++	3 per pass
return m	1

Thought Exercise (offline)

- Based on your knowledge of assembly, and machine architectures try to estimate the number of CPU cycles that might actually be used.
- "Divide" or "shift"? Which is faster?
- **Point: try to eventually build a mental model that is an "internal interpreter" so as to know how a program will run**
- **Such "internal interpreters" are vital for understanding programming (IMHO)**

How many passes through the loop of alg-lec1?

Hint: If ever stuck:

- Try simple concrete examples
- Start from “ridiculously simple” and work up to harder examples

(Real mathematicians often work this way; but then hide it when writing up ☹)

- Do a “trace of the program” by hand:

How many passes through loop?

- Focus on the relevant portions:

$$\textbf{while } (n \geq 2)$$
$$n \leftarrow n/2$$

- Simplest example?
- Exercise: What is smallest positive integer that is a power of two?
- Answer: 1 as $2^0 = 1$
- (If confused, or if you answered "2", then consider revising your maths about exponents and logarithms)

How many passes through loop? (cont)

while (n ≥ 2) { n ← n/2 ; }

- Case: n = 1 = 2^0 passes = 0
- Case: n = 2 = 2^1
  - n=2, then n=1 ; passes=1
- Case: n = 4 = 2^2
  - n=4, then n=2, then n=1 ; passes = 2
- Case: n = 8 = 2^3
  - n=8, 4 , 2, then n=1 ; passes = 3

How many passes through loop? (cont)

- Case: $n = 2^m$
  - $n=2^m, 2^{m-1}, 2^{m-2}, \ldots, 2$
  - m passes through loop
- but note, n is the input not m, so want to write answer in terms of n. Use
  - $m = \log_2(n)$
- Result: passes through loop = $\log_2(n)$

Exercise (cont)

Algorithm: *alg-lec1*

Input: positive integer n, which is a power of 2

Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	$3\ (\log_2(n)+1)$
n ← n/2	$3\ \log_2(n)$
m++	$3\ \log_2(n)$
return m	1

all together: $9\ \log_2(n) + 5$

(the "+1" on line 2, is because the test is done even if it fails)

Remarks

- Each pass through the loop the size of n is halved
  - the “$\log_2(n)$” is typical of such “halving on each iteration”
- **This concept also appears in sorting and searching; hence you MUST make sure you fully understand this example**
  - **The ADE half of the module will probably be incomprehensible otherwise**

Summary

Goal:

- Build foundations for time-analysis of programs

Skills needed:

- Count primitive operations
- Counting of operations with
  - Loops
  - (Recursion)

Removing details

- According to precisely how we count steps we might get many different answers, e.g. something like
  - $5 \log_2(n) + 2$
  - $9 \log_2(n) + 5$, etc
- Also this counts “steps”
  - the translation to runtime depends on the compiler, hardware, etc
- **Need a way to suppress such details**

Next Lecture

“Suppressing the details”

A motivation and introduction to big-Oh

Oh… ohh!!

Exercise: (Advanced, Offline/Take-home)

Given the following code fragment:

```
m ← 0
while (n ≥ 2)
    n ← sqrt(n)
    m++
return m
```

Give an analysis of its runtime.

For the counting, assume that the square root "sqrt" is a primitive operation.