UNIVERSITY OF BIRMINGHAM

Computer Systems
Algorithm Design & Analysis: Efficiency & Complexity

Motivation

We want to build computer systems that work

- And that work well!

To do this we need to think about several factors:

- They must give the correct answer
    - Is this ALWAYS true?
- They must be reliable, maintainable, quick to produce, cheap …
    - Again, is this always true?
- They must be usable – by the user of the system
- They must be efficient

Overview

Algorithm Design & Analysis

- Motivation
- Efficiency and Space complexity
- Efficiency: Time complexity
  - Of algorithms
  - Of problems
- Conclusions and lessons

Efficient?

We want our software to:

- Run as quickly as possible
- Respond as quickly as possible
- Use as little memory as possible
- Use as little network bandwidth as possible
- Use as little power as possible
- …….

Efficient?

So, we are actually trying to optimise many things:

- The balance will depend on our problem

One serious issue we need to consider is:

- Our solution may work very well for ‘small’ problems
- But, if the problem gets more complex how does its performance change?
  - Does it get worse?
  - Does it get MUCH worse?
  - Does it get MUCH, MUCH worse?

What are we interested in?

Actually, there are several things to consider:

- Is our implementation of our algorithm or data representation efficient:
  - Could we improve our code to make it run faster?
    - This might speed things up a bit
      - or a lot … maybe 10 or 100 times
  - Could we improve our data representation so it uses less memory?
    - Again we might improve things dramatically
- But what if our *algorithm* is inherently going to perform worse as the amount of data increases?
- What if our *problem* is inherently hard and gets much harder as the size of the problem grows?

So?

We need to understand how our problem and/or algorithm changes in complexity as it gets larger.

Usually, we focus more on time complexity – how much computation is involved in solving a problem:

- For instance, if we *double* the size of our problem (e.g. twice as much data) then what happens to the computation time:
  - Does it stay the same?
  - Does it get a bit harder? Maybe +30%
  - Does it get a lot harder? Maybe +100%
  - Does it get even worse? Maybe +400% or +800% or worse?
- Usually, we are interested in an almost *qualitative* measure

Space complexity

$\boxed{L_1 \mid X \mid L_2}$

Usually, space (or bandwidth) complexity is not considered in the same way as computational complexity:

- Memory and bandwidth are limited by practical constraints
- We can add memory (and bandwidth or processing) to cope with a bigger problem
- We can constrain our problem to the practical constraints
  - We restrict the resolution of videos to what is practical
  - We restrict the resolution of 3D models to what is practical
  - We can use other techniques to reduce the memory demands (e.g. compression or streaming)

An Example: video

Typical HD video:

- 1920x1080 pixels (4 bytes per pixel):
  - ~2MPixels, ~8Mbytes
- 25 frames per second
  - 200 Mbytes/sec (~1.6 Gbps connection)!

Solution:

- Compression (up to 200:1)
  - Using empirical knowledge of human vision & video streams
  - Lossy compression (cf. lossless compression)
- Reduce frame rate or resolution
- Constrained by practical constraints

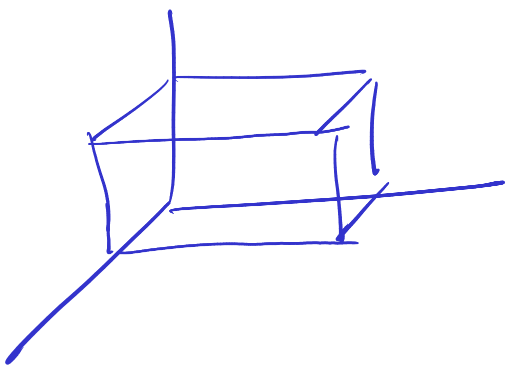
How does the problem change with resolution?

If we double the resolution of a 2D image

- We x4 the number of pixels (n^2)
- If we have a 3D image (e.g. from an MRI scanner)
  - If we double the resolution then: x8 (n^3)

The same problem will occur with 2D models and will be even worse as the number of dimensions increases:

- n^2, n^3 ….



Time complexity

To develop a *basic* understanding
of algorithm design and efficiency in terms of
worst case time complexity.

Outline / Topics

- Algorithm Design and Analysis
- Efficiency
- Time Complexity
- Big-O Notation
- Examples
- Lessons

What is an Algorithm?

In mathematics, computer science, and related subjects, an algorithm is an effective method for solving a problem expressed as a finite sequence of instructions.

- Abu Abdullah Muhammad bin Musa Al-Khwarizmi
  - a Muslim mathematician
  - invented *algorithms* to solve quadratic equations
  - the word *algorithm* derived from his Latin name *Algorithmi*
- An even earlier *algorithm* is the *sieve of Eratosthenes*

Algorithm: question?

Which one is an algorithm?

- A recipe for making tomato soup?
- A procedure to sort 1000 numbers into ascending numeric order?
- A procedure to recognize a particular face in a crowd?
- A method to order objects according to beauty?

Algorithm Design and Analysis

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

Algorithm Efficiency

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

- How do we measure the run-time of an algorithm?
  - benchmarking on *representative* set of inputs: empirical analysis
  - analyse the (time) complexity

Algorithm Efficiency – Empirical Analysis

- Idea: Implement the algorithm (a program) and time it

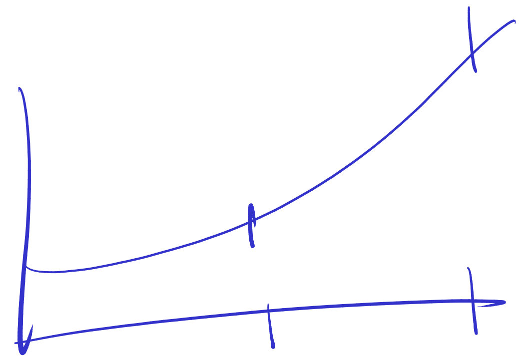
Question: How can we time a program?

- Manual: Using stopwatch
- Automatic: Using some timer function

Run the program for various input sizes and measure run time.

data size	time (ms)
250	5
500	8
1000	10
2000	15
4000	30
8000	50
16000	75

100,000



Algorithm Efficiency – 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)

size of input

- **Questions:**
  - what do we mean by *operations*?
  - what do we mean by *size*?
  - we usually focus on *worst-case*, not *average-case*, analysis

90

100

Example # 1

- 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 # 2

- 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]
```

n n 2 1

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

n x n x 2

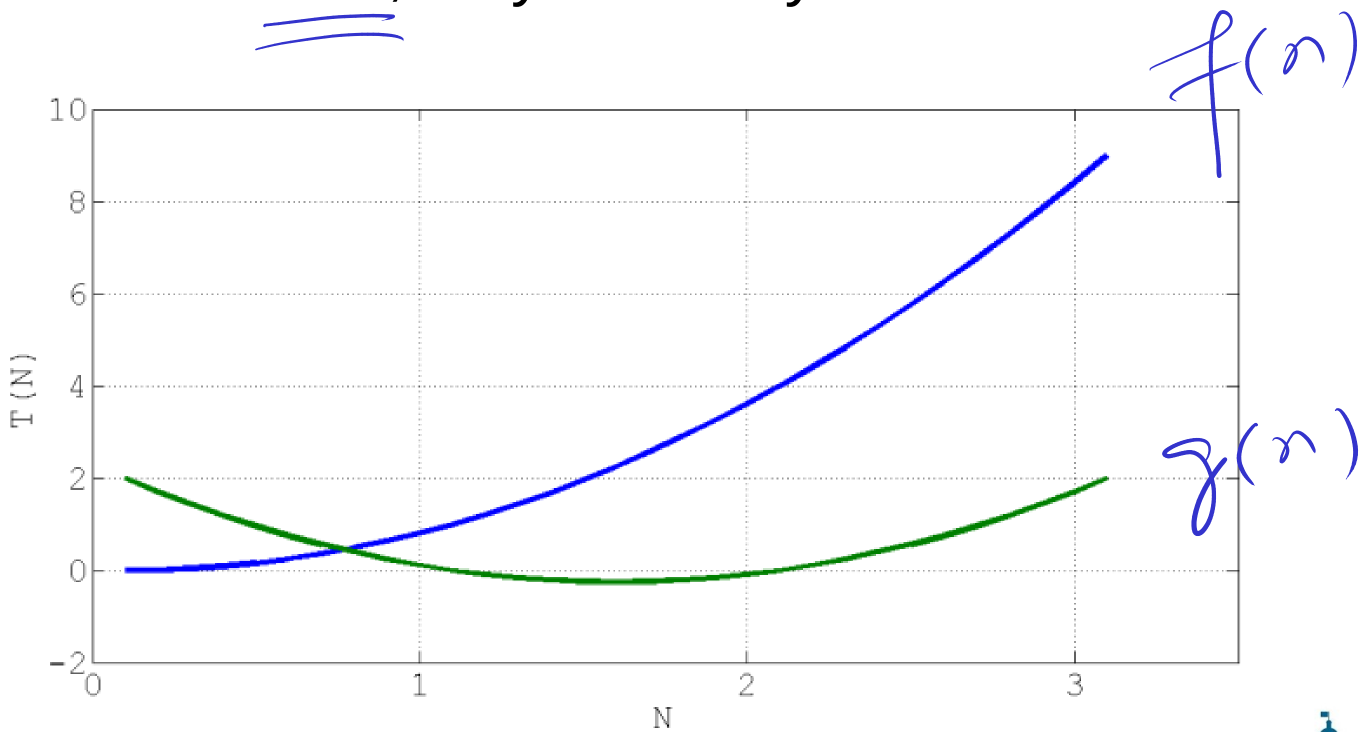
Quadratic Growth

Consider the two functions

$$f(n) = n^2$$

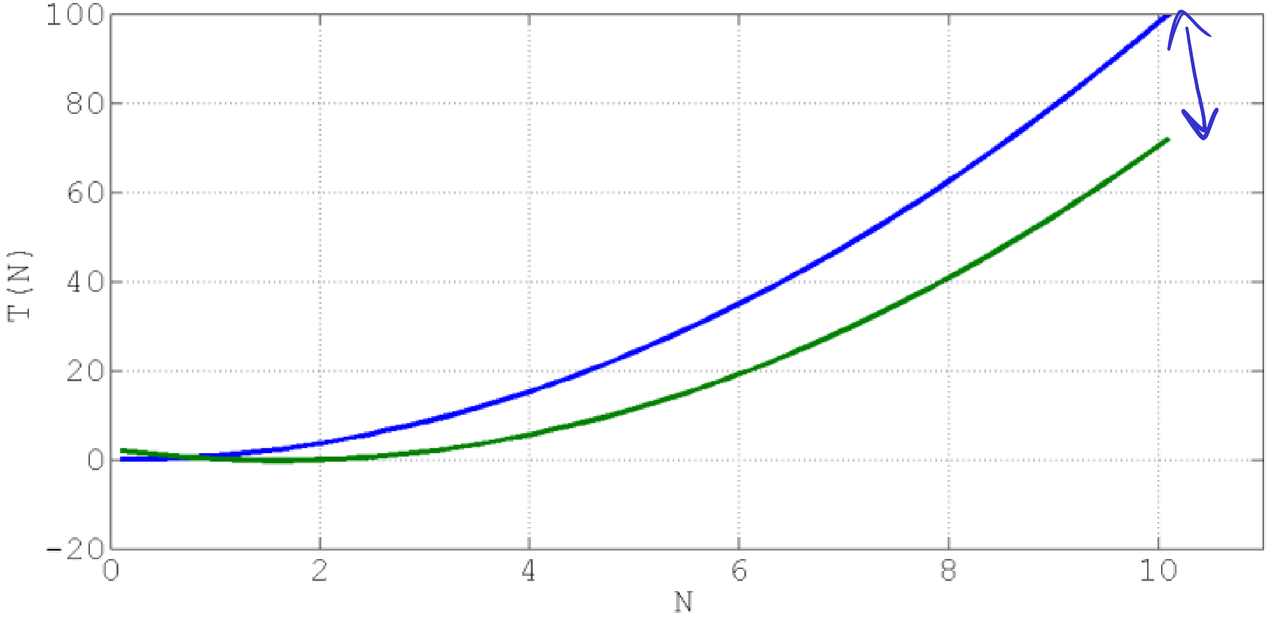
$$g(n) = n^2 - 3n + 2$$

Around n = 3, they look very different



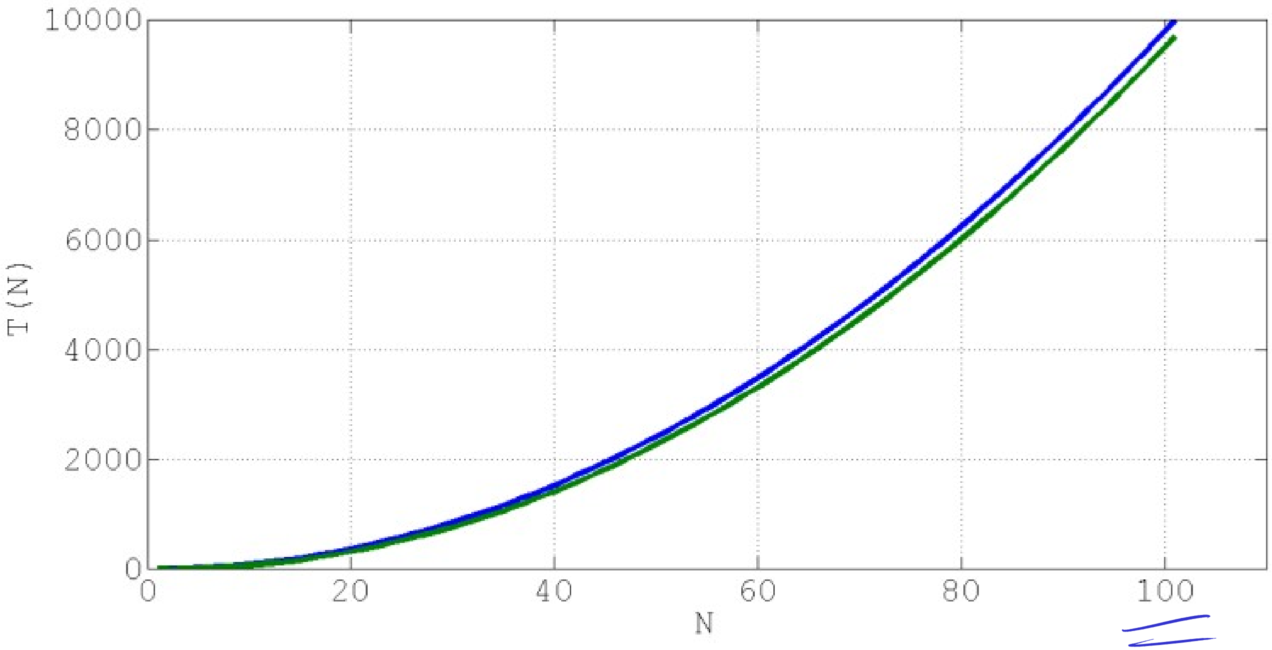
Quadratic Growth

Around n = 10, they look similar



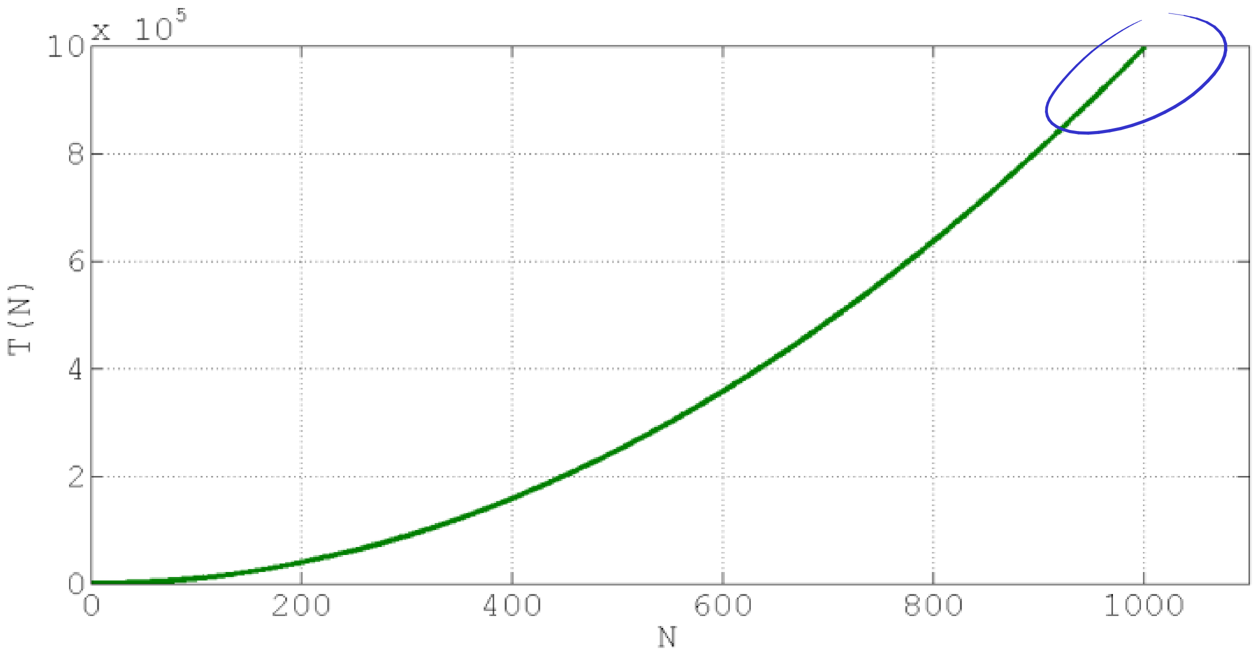
Quadratic Growth

Around n = 100, they are almost the same



Quadratic Growth

Around n = 1000, the difference is indistinguishable!



Quadratic Growth

Asymptotic

The absolute difference is large, such that,

$$f(1000) = 1000000$$
$$g(1000) = 997002$$

but the relative difference is very small,

$$\left| \frac{f(1000) - g(1000)}{f(1000)} \right| = 0.002998 < 0.3\%$$

and this difference goes to zero as n → ∞

Big-O Notation

- Usually we 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
  - $T(n) = n \Longrightarrow$ complexity class = $O(n)$
  - $T(n) = n+2 \Longrightarrow$ complexity class = $O(n)$
  - $T(n) = 2n^2 \Longrightarrow$ complexity class = $O(n^2)$

Big-O Notation

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

constant

- 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 Complexity Classes

$n = 64$

- Some common complexity classes:
  - $O(1)$ = "Constant"
  - $O(\log_2 n)$ = "Logarithmic"
  - $O(n)$ = "Linear"
  - $O(n \log_2 n)$ = "Log Linear"
  - $O(n^2)$ = "Quadratic"
  - $O(n^3)$ = "Cubic"
  - $O(2^n)$ = "Exponential"

2^{64}

2^{128}

$n = 128$

- 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

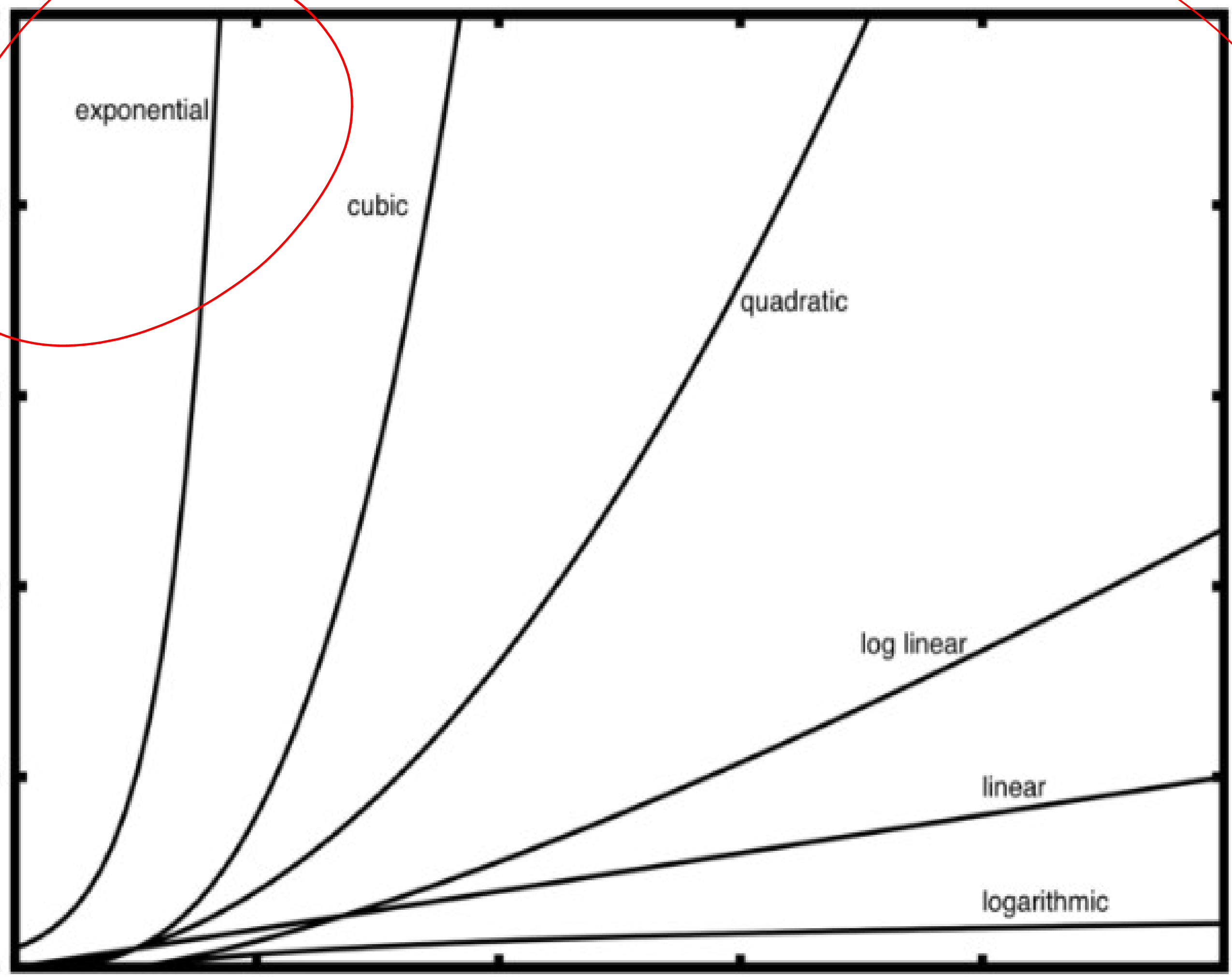
f(n)	n = 4	n = 16	n = 256	n = 1024	n = 1048576
1	1	1	1	1.00×10^{0}	1.00×10^{0}
$\log_2 \log_2 n$	1	2	3	3.32×10^{0}	4.32×10^{0}
$\log_2 n$	2	4	8	1.00×10^{1}	2.00×10^{1}
n	4	16	256	1.02×10^{3}	1.05×10^{6}
$n \log_2 n$	8	64	2.05×10^{3}	1.02×10^{4}	2.10×10^{7}
n^2	16	256	6.55×10^{4}	1.05×10^{6}	1.10×10^{12}
n^3	64	4.10×10^{3}	1.68×10^{7}	1.07×10^{9}	1.15×10^{18}
2^n	16	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 usec	1 usec	1 usec	1 usec	1 usec
$\log_2 \log_2 n$	1 usec	2 usec	3 usec	3.32 usec	4.32 usec
$\log_2 n$	2 usec	4 usec	8 usec	10 usec	20 usec
n	4 usec	16 usec	256 usec	1.02 msec	1.05 sec
$n \log_2 n$	8 usec	64 usec	2.05 msec	10.2 msec	21 sec
n^2	16 usec	256 usec	65.5 msec	1.05 sec	12.72 days
n^3	64 usec	4.1 msec	16.8 sec	17.83 min	36559 yrs
2^n	16 usec	65.5 msec	$\boxed{3.7 \times 10^{63} \text{ yrs}}$	5.7×10^{294} yrs	2.14×10^{315639} yrs

Graphically



Example # 1 (again)

$v = 90$

- Look up a value v in an array x

1	4	17	3	90	79	4	6	81

Example # 1 (again)

- Look up a value v in a <u>sorted</u> array x

1	3	4	4	6	17	79	81	90

v = 81

Example # 1 (again)

- 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

max(O(n) + O(n²))

Sequential Algorithm 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]
```

O(n)

n n

O(n²)

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

$O(n \log_2 n)$

Function/Method calls

- Example: n array look-ups

$O(n)$

$O(\log_2 n)$

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

- Complexity: O(n) x O(log n) = O(n log n)
- In general: "multiplication" of complexities

Computing Complexity Classes

- Determine total algorithm complexity
  - from the complexities of its components

1) Sequential algorithm phases: "maximum"
   - e.g. $O(n) + O(n^2) = O(n^2)$
   - e.g. $O(n) + O(\log n) = O(n)$

2) Function / Method calls: "multiplication"
   - e.g. $O(n) \times O(\log n) = O(n \log n)$
   - e.g. $O(n^2) \times O(1) = O(n^2)$

Question # 1

Given the following code fragment, what is its Big-O running time?

```
test = 0
for i in range(n):
    for j in range(n):
        test = test + i * j
```

a) O(n)
b) O(n^2)
c) O(log n)
d) O(n^3)

Question # 2

Given the following code fragment, what is its Big-O running time?

```
test = 0
for i in range(n):
    test = test + 1

for j in range(n):
    test = test - 1
```

a) O(n)
b) O(n^2)
c) O(log n)
d) O(n^3)

Question # 3

Given the following code fragment, what is its Big-O running time?

1000

```
i = n
while i > 0:
    k = 2 + 2
    i = i / 2
}
```

1000

500

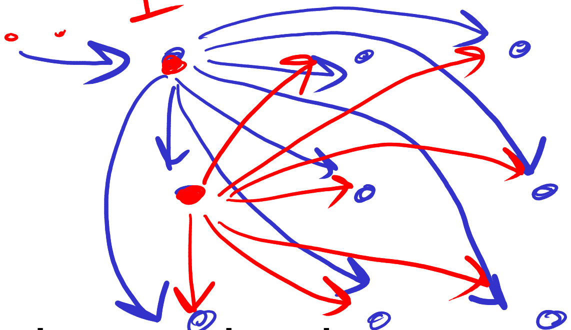
250

125

⋮

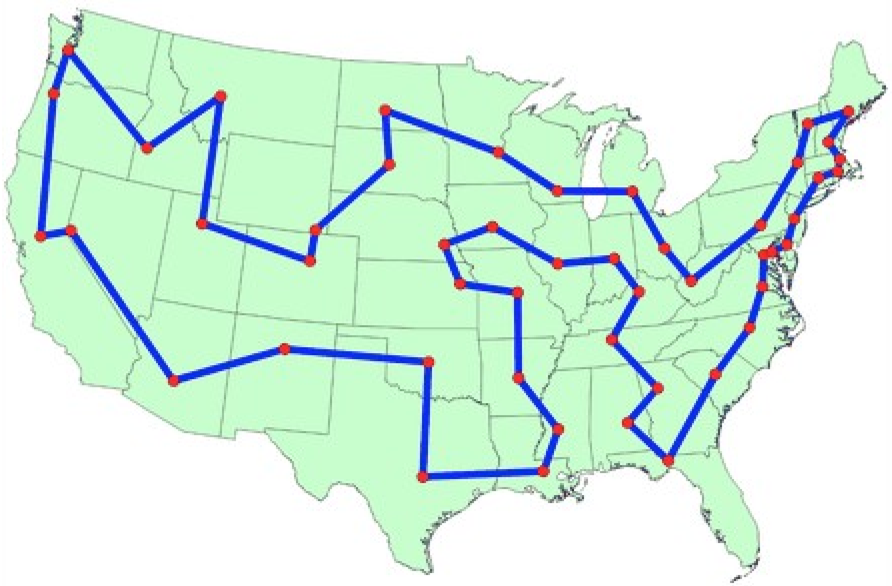
a) O(n)
b) O(n^2)
c) O(log n)
d) O(n^3)

$9! = 9 \times 8 \times 7 \times \dots 1$



Some Harder Problems

- Traveling 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?



“A naive approach to solving TSP would be the brute-force solution, which means finding all possible routes given a certain number of nodes. This is a very expensive way to solve it, with a time complexity of O(n!).”

https://tonicanada.medium.com/introduction-to-the-travelling-salesman-problem-5ace44932cb5

https://www.youtube.com/watch?v=SC5CX8drAtU

Some Harder Problems

- **Boolean Satisfiability Problem (SAT)**
  - Given a Boolean formula $F(x_1, x_2, x_3, \ldots, x_n)$
  - Can F evaluate to 1 (true)?
  - If yes, return values of x_i's (satisfying the assignment) that make F true.
- **In both cases (TSP & SAT):**
  - lots of practical applications (Operations Research, Optimization, Logistics, Model Checking, Software Verification, … etc.)
  - 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 uses the tightest possible)
  - sometimes consider lower bounds:
  - e.g. sorting: O(n log n)

P and 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 (Non-deterministic polynomial time)
  - if we can "guess" a solution, it can be checked efficiently (efficiently = in polynomial time). e.g. Sudoku, Chess (more [at this link])
  - In contrast, polynomial time (or P) problems are those where finding the answer is easy.
- 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 …
  - [US$1,000,000 prize if you solve it!]

				3	7	6		
			6				9	
		8						4
	9							1
6								9
3							4	
7						8		
	1				9			
		2	5	4				

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), O(n log n), 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

Lessons

- Data Representation/Algorithms are tightly linked
- Some algorithms are just better than others:
  - But sometimes the ‘best’ depends on the problem
- Some problems are inherently hard (in the complexity sense)
  - This doesn’t mean they cannot be solved
  - We may need to reformulate the problem
    - Constrain it
    - Relax conditions
    - E.g. instead of “find the best ….” we “find a very good ….”

Books / References

- Goldschlager and Lister: Computer Science: a Modern Introduction (Prentice Hall, 2nd edition 1988)
  - Chapter 3
- Aho & Ullman: Foundations of Computer Science (Freeman, 1992)
  - Chapter 3
  - http://infolab.stanford.edu/~ullman/focs.html