COMP202
Complexity of Algorithms

Introductory Lectures

Reading materials: Chapters 1, 2.1, 2.2, 3 in CLRS

The Era of Algorithms

Real-world applications of algorithms:

- Analysis of Big Data (Machine Learning Algorithms)
- Social Media (Algorithmic Marketing)
- Online Shopping (Recommendation Systems, Online Auctions)
- Stock Market (High-Frequency Trading Algorithms)
- Quantum computing (Next-generation algorithms)

The Era of Algorithms

Real-world applications of algorithms:

- Analysis of Big Data (Machine Learning Algorithms)
- Social Media (Algorithmic Marketing)
- Online Shopping (Recommendation Systems, Online Auctions)
- Stock Market (High-Frequency Trading Algorithms)
- Quantum computing (Next-generation algorithms)
- At Liverpool: Largest Algorithms Research Group in the UK, doing research in algorithmic optimisation, algorithms for networks and distributed computing, algorithmic game theory and algorithmic economics, and complexity of problems.

COMP202: Complexity of Algorithms

- Lecturer: Dr. Nikhil Mande, Room 3.14 Ashton Building, email: `mande (at) liverpool.ac.uk`
- In-person lectures: Mondays (1100-1200), Tuesdays (1700-1800), Thursdays (1400-1500). 3h/week for 11-12 weeks. Slides available on Canvas before (and after) lectures.
- Tutorials: each student assigned to 1h / week tutorial for 11-12 weeks. Face-to-face on campus.
- Assessment: one half-term test (15%), one programming assignment (15%), and final exam (70%)
- Course materials will be posted on Canvas.

Module Outcomes

Goal of module:

To train you to **think algorithmically**, to be able to analyze the correctness of algorithms and their performance in terms of the use of time and space, and to be able to think critically about algorithmics in general.

From Wikipedia: Algorithmics is the systematic study of the design and analysis of algorithms.

Module Outcomes

Goal of module:

To train you to **think algorithmically**, to be able to analyze the correctness of algorithms and their performance in terms of the use of time and space, and to be able to think critically about algorithmics in general.

From Wikipedia: Algorithmics is the systematic study of the design and analysis of algorithms. This course is among the most useful courses to give you knowledge that helps developing your superb **programming skills**!

Module Outcomes (cont.)

More specifically, by the end of this module a student should

1. have an appreciation of the diversity of computational fields to which algorithmics has made significant contributions,

Module Outcomes (cont.)

More specifically, by the end of this module a student should

1. have an appreciation of the diversity of computational fields to which algorithmics has made significant contributions,
2. have fluency with the structure and utilization of abstract data types such as stacks, queues, lists, and trees in conjunction with classical algorithmic problems such as searching, sorting, graph algorithms, etc.,

Module Outcomes (cont.)

More specifically, by the end of this module a student should

1. have an appreciation of the diversity of computational fields to which algorithmics has made significant contributions,
2. have fluency with the structure and utilization of abstract data types such as stacks, queues, lists, and trees in conjunction with classical algorithmic problems such as searching, sorting, graph algorithms, etc.,
3. have a comfortable knowledge of graphs, their role in representing various relationships, and be familiar with several of the algorithms for solving common problems such as finding maximum flows, shortest paths, and matchings,

Module Outcomes (cont.)

4. understand the basic concepts from number theory that are used in "public-key" cryptography, and be able to explain the concepts that underly the RSA encryption/decryption scheme,

Module Outcomes (cont.)

4. understand the basic concepts from number theory that are used in "public-key" cryptography, and be able to explain the concepts that underly the RSA encryption/decryption scheme,
5. be familiar with formal theories providing evidence that many important computational problems are inherently intractable, e.g., NP-completeness,

Module Outcomes (cont.)

4. understand the basic concepts from number theory that are used in "public-key" cryptography, and be able to explain the concepts that underly the RSA encryption/decryption scheme,
5. be familiar with formal theories providing evidence that many important computational problems are inherently intractable, e.g., NP-completeness,
6. understand and appreciate models of computation where one can prove concrete lower bounds, such as query complexity and communication complexity.

Skills this module teaches

- Self-management, e.g., in planning implementation of algorithmic solutions to problems
- Positive attitude: innovative approach to problem solving, creative thinking
- Communications skills: communicating in writing formal problem specification/correctness of algorithms
- Literacy: written specifications of problems and their algorithmic solutions
- Applications of numeracy: algorithmic manipulations of numbers and discrete objects
- Problem solving: formulating problems as algorithmic models, designing algorithms to solve them
- Applications of IT: implementation of algorithms
- Computer Science principles: assessing computational complexity of problems, verifying correctness of algorithms

Course Text Book

CLRS

Introduction to Algorithms, Third Edition

by T. H. Cormen, C. E. Leiserson and R. L. Rivest and C. Stein

For last week of lectures,

Computational Complexity, A Modern Approach

by Sanjeev Arora and Boaz Barak

AB QUERY & COMM. COMP.

Available (also online) in our library.

Some fun algorithms conferences

- *FUN with Algorithms* – bi-annual conference on real-world / fun / unusual algorithmics
- *SIAM Symposium on Simplicity in Algorithms (SOSA)* – new annual conference on elegant / simple / unusual algorithmics
- *Innovations in Theoretical Computer Science (ITCS)* – annual conference on novel / unusual concepts in theoretical computer science / algorithms

Path to Success in this Course: Material

- This course is among the most useful courses to give you knowledge that helps developing your superb programming skills!
- This course is **quite technical** and thus requires **systematic** and **dedicated** work from your side.
- It will **gradually** build up your knowledge in the design and analysis of algorithms, i.e., the following subjects will crucially and systematically build on the previous ones.

Path to Success in this Course: Practical Sessions (Tutorials)

Practicals will begin from week 2.

There are 11 scheduled 1h tutorials during each week.
They are scheduled face-to-face on campus.
Please attend the one (1h/week) to which you have been assigned!

Tutorials, led by Demonstrators, are designed to be opportunities for you to ask questions, clarify things that are not yet clear to you, seek help with the assigned problems you will be given, etc.

Path to Success in this Course: Exercises

You will be given exercises to try out (one exercise list per week). You should try all of them!

Most importantly, you should try them all out ***before*** looking at the solutions.

While these exercises will not be assessed, they are important to help understand the course material, and some of the exercises might turn up (perhaps in a slightly altered format) in the half-term test and/or examination.

Exercises and solutions will be posted on Canvas.

Path to Success in this Course: Summary

- Try solving exercises before tutorials and before looking at solutions.
- Work systematically and keep track on how the material builds on and connects to the previous material.
- Attend all lectures and tutorials!
- If you have doubts and struggle at any point, do let me know, keep asking questions during Q and A sessions, ask your demonstrators during tutorials, etc.

What is an algorithm?

- An *algorithm* is a sequence of steps for performing a task in a finite amount of time.

What is an algorithm?

- An *algorithm* is a sequence of steps for performing a task in a finite amount of time.
- What algorithms do you already know?

DIJKSTRA: SHORTEST PATH

BUBBLESORT

EXTENDED EUCLID: GCD

What is an algorithm? (cont.)

- For computational algorithms, we use *data structures* to store, access, and manipulate data.

What is an algorithm? (cont.)

- For computational algorithms, we use *data structures* to store, access, and manipulate data.
- What data structures have you encountered in the past?

TUPLE

QUEUE

LINKED LIST →

STACK

What is an algorithm? (cont.)

- For computational algorithms, we use *data structures* to store, access, and manipulate data.
- What data structures have you encountered in the past?

Our fundamental interest is to design "good" *algorithms* which utilize efficient *data structures*. In this context "good" (typically) means *fast* in a way to be clarified during this course.

A first simple example

Suppose you have a list of some type (integers, names, etc.), and you want to search through the list to see if some element is already in the list.

How can you do this search? Moreover, how can you do it *efficiently*?

A first simple example

Suppose you have a list of some type (integers, names, etc.), and you want to search through the list to see if some element is already in the list.

How can you do this search? Moreover, how can you do it *efficiently*?

Relevant questions

- How is the list stored?

A first simple example

Suppose you have a list of some type (integers, names, etc.), and you want to search through the list to see if some element is already in the list.

How can you do this search? Moreover, how can you do it *efficiently*?

Relevant questions

- How is the list stored?
- Is it sorted or not?
  If the list is sorted, it's (often) easier to perform the search.

A first simple example

Suppose you have a list of some type (integers, names, etc.), and you want to search through the list to see if some element is already in the list.

How can you do this search? Moreover, how can you do it *efficiently*?

Relevant questions

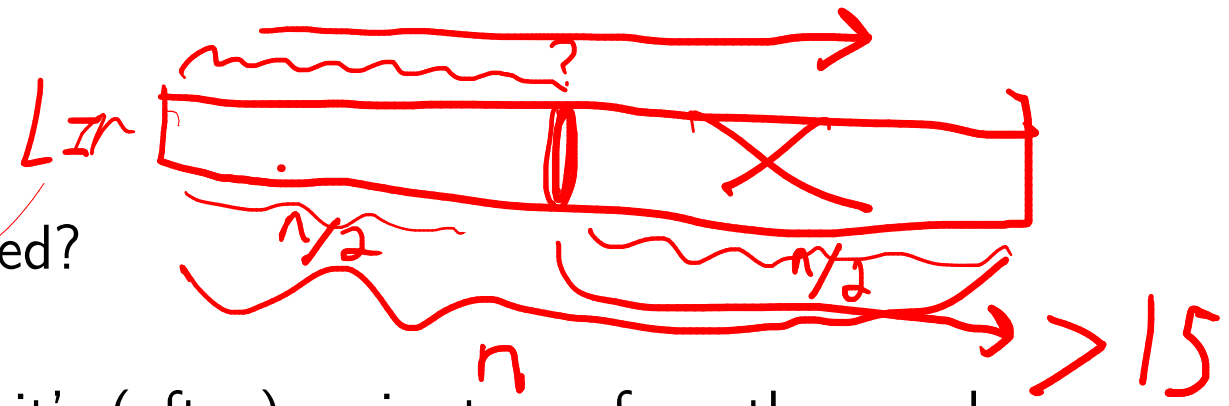
- How is the list stored?
- Is it sorted or not?
  If the list is sorted, it's (often) easier to perform the search.
- If it's not sorted, should we sort it first?

A first simple example

Suppose you have a list of some type (integers, names, etc.), and you want to search through the list to see if some element is already in the list.

$15 \in L$?

How can you do this search? Moreover, how can you do it *efficiently*?



Relevant questions

- How is the list stored?
- Is it sorted or not?
  If the list is sorted, it's (often) easier to perform the search.
- If it's not sorted, should we sort it first?
- Are we going to be doing this search operation many times?
  (Then we might want to sort the list first, if necessary.)

LOG = LOG$_2$.

$n \xmapsto{1} n/2 \xmapsto{1} n/4 \cdots \lceil \log_2 n \rceil$

A first simple example

Suppose you have a list of some type (integers, names, etc.), and you want to search through the list to see if some element is already in the list.

How can you do this search? Moreover, how can you do it *efficiently*?

Relevant questions

- How is the list stored?
- Is it sorted or not?
  If the list is sorted, it's (often) easier to perform the search.
- If it's not sorted, should we sort it first?
- Are we going to be doing this search operation many times? (Then we might want to sort the list first, if necessary.)
- Are we going to be updating the list (i.e. adding and/or deleting items)?

Algorithm Analysis

- **Primary interest**: Running time (*time-complexity*) of the algorithm and the operations on data structures.

Algorithm Analysis

- **Primary interest**: Running time (*time-complexity*) of the algorithm and the operations on data structures.
- **Secondary interest**: Space (or "memory") usage (*space-complexity*).

Algorithm Analysis

- **Primary interest**: Running time (*time-complexity*) of the algorithm and the operations on data structures.
- **Secondary interest**: Space (or "memory") usage (*space-complexity*).
- **Also of interest**: (Near) Optimality of the output solution (for *optimisation* problems).

Algorithm Analysis

- **Primary interest**: Running time (*time-complexity*) of the algorithm and the operations on data structures.
- **Secondary interest**: Space (or "memory") usage (*space-complexity*).
- **Also of interest**: (Near) Optimality of the output solution (for *optimisation* problems).

This module focuses on the *mathematical* design and analysis of algorithms and their associated data structures, utilizing relevant mathematical tools to do so.

Algorithm Analysis

- **Primary interest**: Running time (*time-complexity*) of the algorithm and the operations on data structures.
- **Secondary interest**: Space (or "memory") usage (*space-complexity*).
- **Also of interest**: (Near) Optimality of the output solution (for *optimisation* problems).

This module focuses on the *mathematical* design and analysis of algorithms and their associated data structures, utilizing relevant mathematical tools to do so.

Let's first discuss the difference between *experimental analysis* and *theoretical analysis* of algorithms.

- We are interested in *dependency* of the running time on *size* of the input.

The *size* could mean the number of vertices and edges if we are operating on a graph, the length of a message we're encoding/decoding, and/or the actual length of numbers we're processing in terms of the bits needed to store them in memory.

Experimental Analysis of Algorithms

- We are interested in *dependency* of the running time on *size* of the input.

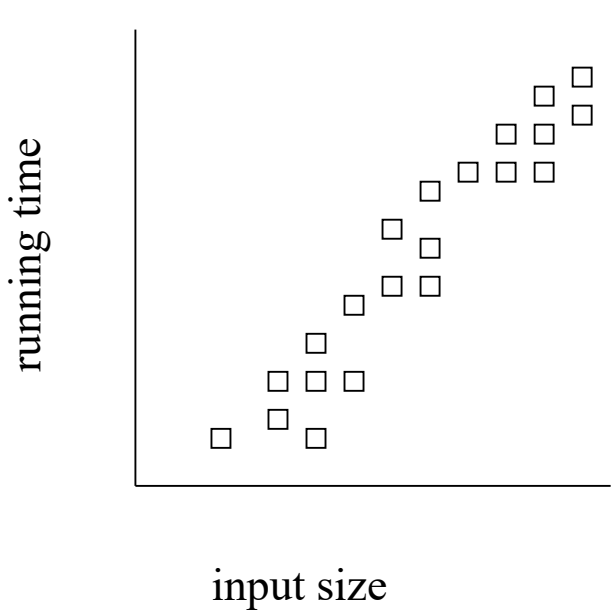
The *size* could mean the number of vertices and edges if we are operating on a graph, the length of a message we're encoding/decoding, and/or the actual length of numbers we're processing in terms of the bits needed to store them in memory.

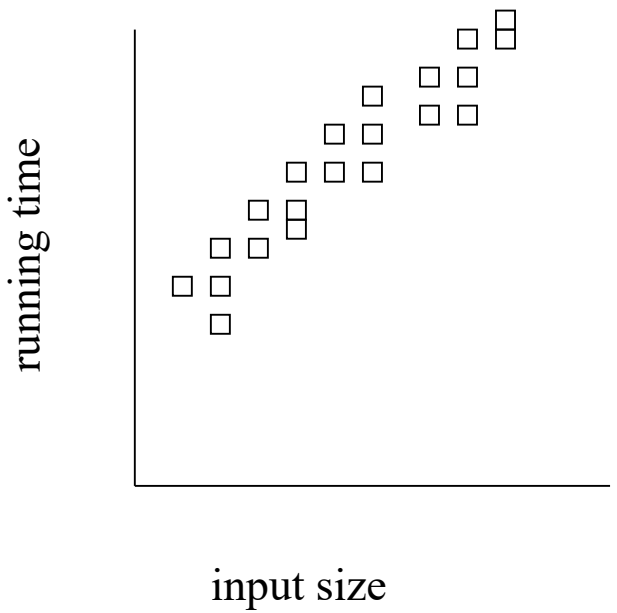
To analyze algorithms we might perform *experiments* to empirically observe the running time.

Doing so requires a good choice of sample inputs, and an appropriate number of tests (so that we can have *statistical certainty* about our analysis).

Experimental Analysis (cont.)

Also, the running time depends on both the *size* and *instance* of input and the algorithm used, as well as the software and hardware environment on which it is run.





Drawbacks of Experimental Analysis

1. Experiments are performed on a *limited set* of test inputs.
2. Requires all tests to be performed using *same hardware and software*.
3. Requires implementation and execution of algorithm.

Theoretical Analysis

Benefits over experimental analysis:

1. Can take all possible inputs into account.
2. Can compare efficiency of two (or more) algorithms, independent of hardware/software environment.
3. Involves studying high-level descriptions of algorithms (e.g. *pseudo-code*).

Theoretical Analysis (cont.)

- When we analyze an algorithm in this fashion we aim to associate a function $f(n)$ to each algorithm, where $f(n)$ characterizes the running-time in terms of some measure of the *input-size*, n.

  (For example, n could be the number of items to be sorted in a list.)

Typical functions include: $n, \log n, n^2, n \log n, n^5, 2^n, \ldots$

Theoretical Analysis (requirements)

For formal theoretical analysis, we need:

1. A *language* to describe algorithms.
2. *Computational model* in which algorithms are executed.
3. *Metric* for measuring running-time.
4. A way of characterizing running-time.

Pseudo-code

- Pseudo-code is a high-level description language for algorithms.

Psuedo-code provides more structured description of algorithms.

Allows high-level analysis of algorithms to determine their running time (and space requirements).

Pseudo-code example

Finding the *minimum element* of a set of numbers, stored in an array A.

Pseudo-code example

Finding the *minimum element* of a set of numbers, stored in an array A.

Minimum-Element(A)

```
    /** Find the minimum of elements in the array A. **/
1   min ← A[1]
2   for j ← 2 to length[A]
3       do
4          ▷ Compare element A[j] to current min
5          ▷ and store it if it's smaller.
6          if A[j] < min
7             then min ← A[j]
8   return min
```

Pseudo-code (cont.)

- Pseudo-code is a mixture of natural language and high-level programming language (e.g., Java, Python, C, etc.).

Describes a generic implementation of data structures or algorithms.

- Pseudo-code includes: *expressions, declarations, variable initialization and assignments, conditionals, loops, arrays, method calls*, etc.

Pseudo-code (cont.)

- Pseudo-code is a mixture of natural language and high-level programming language (e.g., Java, Python, C, etc.).

Describes a generic implementation of data structures or algorithms.

- Pseudo-code includes: *expressions, declarations, variable initialization and assignments, conditionals, loops, arrays, method calls*, etc.

I won't formally define a strict method for giving pseudo-code, but when using it we aim to describe an algorithm in a manner that would allow a competent programmer to translate the pseudo-code into program code *without misinterpretation* of the algorithm.

Correctness

Loop invariants are useful to show correctness of an algorithm given as pseudo-code.

Correctness

Loop invariants are useful to show correctness of an algorithm given as pseudo-code.
In this case, a useful loop invariant is "at the start of the j'th iteration of the **for** loop of Lines 2-7, *min* equals the minimum value in $A[1..j-1]$."

TO SHOW FOR $j = 2, \ldots,$ LEN(A)

– $j = 2$ CLEAR

Loop invariants

Three essential things to show for loop invariants.

- **Initialisation**: It is true prior to the first iteration of the loop.

Loop invariants

Three essential things to show for loop invariants.

- **Initialisation**: It is true prior to the first iteration of the loop.
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.

Loop invariants

Three essential things to show for loop invariants.

- **Initialisation**: It is true prior to the first iteration of the loop.
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination**: When the loop terminates, the invariant gives us a useful property that helps show the algorithm correct.

Loop invariants

Three essential things to show for loop invariants.

- **Initialisation**: It is true prior to the first iteration of the loop.
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination**: When the loop terminates, the invariant gives us a useful property that helps show the algorithm correct.

Additional reading: Chapter 2.1, 2.2 of CLRS for pseudo-code and analysis of the slightly more complicated algorithm of Insertion Sort.

Computational Model

Our computational model is like that typically found on modern-day computers.

- CPU connected to a bank of *memory cells*.
- Each memory cell can store a number, character, or address.

We use a set of high-level *primitive operations*: *assignment, method invocation, arithmetic operations, comparison of numbers, array indexing, returning from a method*, etc.

Computational Model (cont.)

Assumption: primitive operations (like a single addition, multiplication, or comparison) require *constant time* to perform.

(Not necessarily true, e.g. on modern processors, multiplication generally takes about three to four times as long to perform on a computer than addition. Division typically takes at least twice as long as multiplication.)

Computational Model (cont.)

Assumption: primitive operations (like a single addition, multiplication, or comparison) require *constant time* to perform.

(Not necessarily true, e.g. on modern processors, multiplication generally takes about three to four times as long to perform on a computer than addition. Division typically takes at least twice as long as multiplication.)

- When we analyze the *running time* of an algorithm we *count* the number of operations executed during the course of the algorithm.

Counting Primitive Operations

Minimum-Element(A)

```
1  min ← A[1]
2  for j ← 2 to length[A]
3        do
4           ▷ Compare element A[j] to current min
5           ▷ and store it if it's smaller.
6           if A[j] < min
7              then min ← A[j]
8  return min
```

How many primitive operations?

Counting Primitive Operations

Minimum-Element(A)

```
1  min ← A[1]
2  for j ← 2 to length[A]
3        do
4           ▷ Compare element A[j] to current min
5           ▷ and store it if it's smaller.
6           if A[j] < min
7              then min ← A[j]
8  return min
```

How many primitive operations?

1. One assignment in line 1;

Counting Primitive Operations

Minimum-Element(A)

```
1  min ← A[1]
2  for j ← 2 to length[A]
3        do
4           ▷ Compare element A[j] to current min
5           ▷ and store it if it's smaller.
6           if A[j] < min
7              then min ← A[j]
8  return min
```

How many primitive operations?

1. One assignment in line 1;
2. $length[A] - 1$ more assignments (to j) in line 2;

Counting Primitive Operations

Minimum-Element(A)

```
1  min ← A[1]
2  for j ← 2 to length[A]
3        do
4           ▷ Compare element A[j] to current min
5           ▷ and store it if it's smaller.
6           if A[j] < min
7              then min ← A[j]
8  return min
```

How many primitive operations?

1. One assignment in line 1;
2. $length[A] - 1$ more assignments (to j) in line 2;
3. Line 6 gives $length[A] - 1$ comparisons;

Counting Primitive Operations

Minimum-Element(A)

```
1  min ← A[1]
2  for j ← 2 to length[A]
3        do
4           ▷ Compare element A[j] to current min
5           ▷ and store it if it's smaller.
6           if A[j] < min
7              then min ← A[j]
8  return min
```

How many primitive operations?

1. One assignment in line 1;
2. $length[A] - 1$ more assignments (to j) in line 2;
3. Line 6 gives $length[A] - 1$ comparisons;
4. A possible $length[A] - 1$ more assignments from line 7;

Counting Primitive Operations

```
Minimum-Element(A)
1  min ← A[1]
2  for j ← 2 to length[A]
3       do
4            ▷ Compare element A[j] to current min
5            ▷ and store it if it's smaller.
6            if A[j] < min
7               then min ← A[j]
8  return min
```

How many primitive operations?

1. One assignment in line 1;
2. $length[A] - 1$ more assignments (to j) in line 2;
3. Line 6 gives $length[A] - 1$ comparisons;
4. A possible $length[A] - 1$ more assignments from line 7;
5. Finally there's the last statement to execute once;

Counting Primitive Operations

Minimum-Element(A)

```
1  min ← A[1]
2  for j ← 2 to length[A]
3        do
4           ▷ Compare element A[j] to current min
5           ▷ and store it if it's smaller.
6           if A[j] < min
7              then min ← A[j]
8  return min
```

How many primitive operations?

1. One assignment in line 1;
2. $length[A] - 1$ more assignments (to j) in line 2;
3. Line 6 gives $length[A] - 1$ comparisons;
4. A possible $length[A] - 1$ more assignments from line 7;
5. Finally there's the last statement to execute once;

$\Rightarrow$ A maximum of $3(length[A] - 1) + 2$ primitive operations.

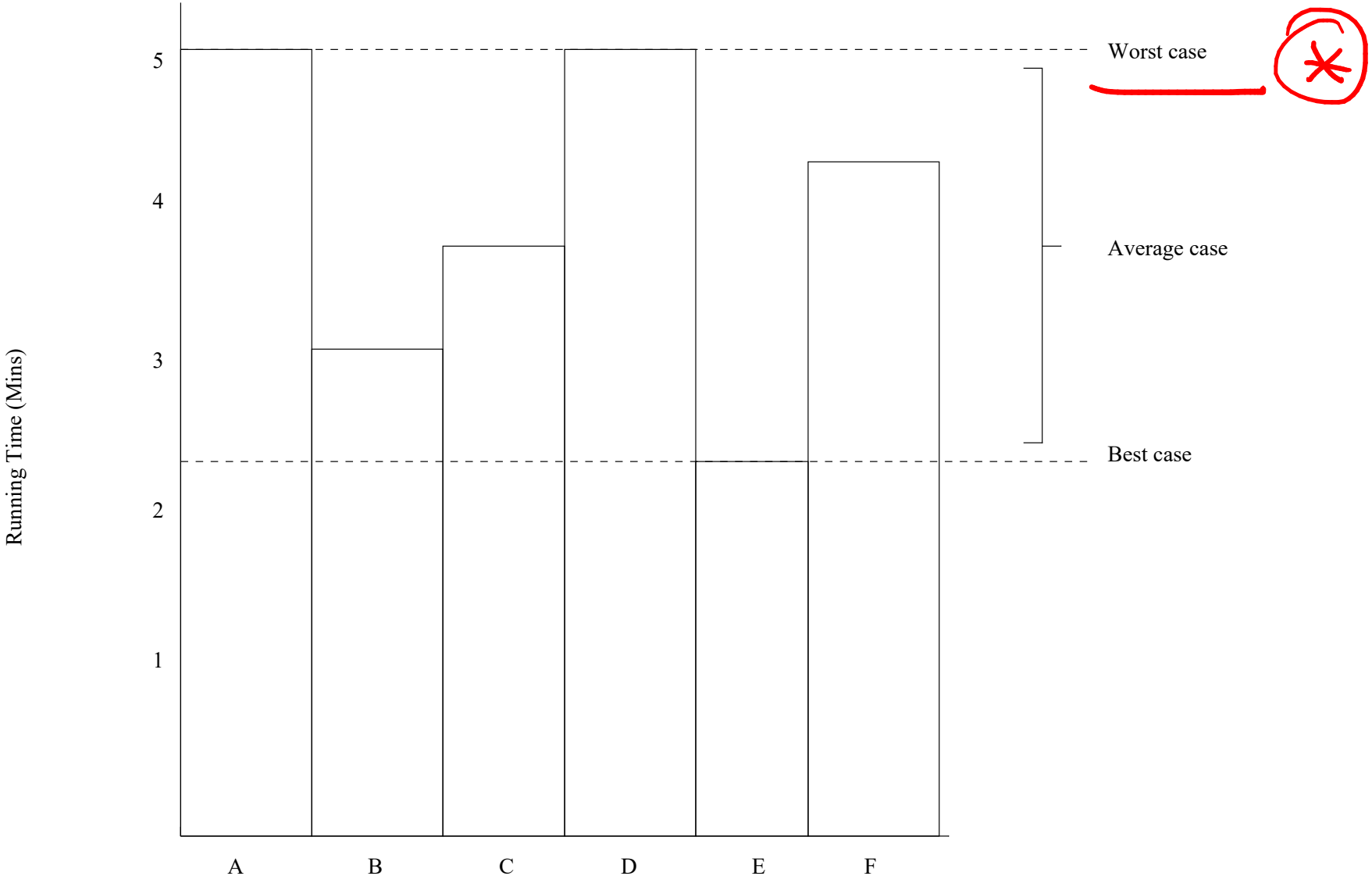
Average- vs. Worst-Case Complexity

An algorithm may run faster on some inputs compared to others.

- *Average-case* complexity refers to running time as an *average taken over all inputs* of the same size.
- *Worst-case* complexity refers to running time as the *maximum taken over all inputs* of the same size.

Usually, we're most interested in worst-case complexity.

Average- vs. Worst-Case Complexity (cont.)



Asymptotic notation

- *Asymptotic notation* allows characterization of the *main factors* affecting running time.

Used in a *simplified analysis* that estimates the number of primitive operations executed *up to a constant factor*.

Such notation lets us compare the running times of two algorithms.

You first encountered this type of notation in COMP108.

"Big-Oh" Notation

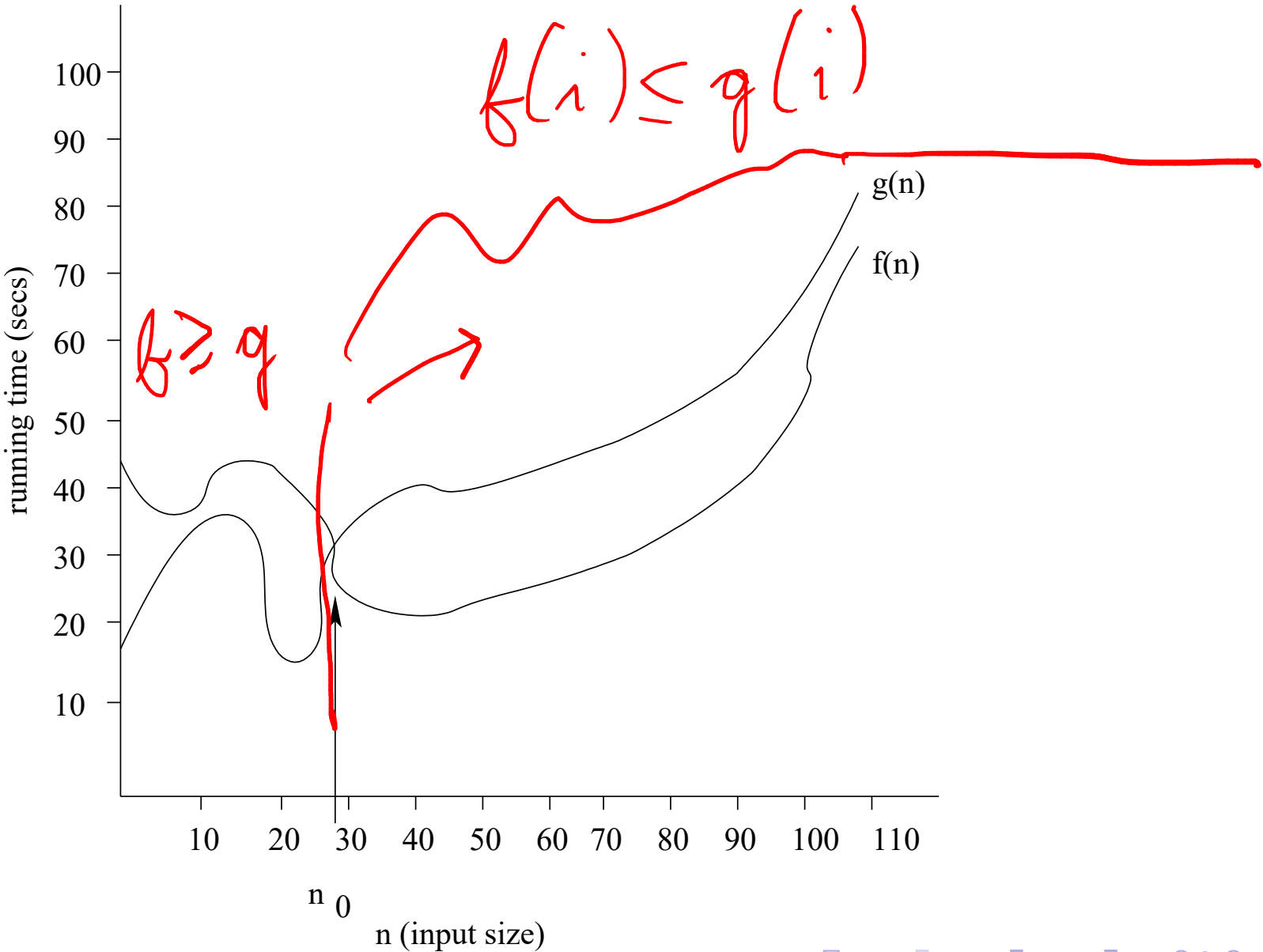
"Big-Oh" notation is probably the most commonly used form of asymptotic notation.

- Given two positive functions $f(n)$ and $g(n)$ (defined on the nonnegative integers), we say $f(n)$ is $O(g(n))$, written $f(n) \in O(g(n))$, if there are constants c and n_0 such that:

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

$f(n) = O(g(n))$

"Big-Oh" Notation (cont.)



Example: $7n - 4 \in O(n)$

$f(n) \in O(g(n)) \equiv f(n) = O(g(n))$

- Need to find constants c and n_0 such that: $7n - 4 \leq cn$, for all $n \geq n_0$.

$c = 7, \; n_0 = 1$

$\forall n \geq 1: \quad 7n - 4 \overset{?}{\leq} 7n$

$\Updownarrow$

$4 \geq 0$

Example: $7n - 4 \in O(n)$

- Need to find constants c and n_0 such that: $7n - 4 \leq cn$, for all $n \geq n_0$.
- One possible choice is $c = 7$ and $n_0 = 1$.
- In fact, this is just one of infinitely many choices, because any real number $c \geq 7$ and any integer $n_0 \geq 1$ would be OK.

Further examples of Big-Oh

1. $3n^2 + n + 16$ is $O(n^2)$. Why?

CLAIM $f(n) = O(g(n))$

IF $\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$

o ω

Further examples of Big-Oh

1. $3n^2 + n + 16$ is $O(n^2)$. Why?

   We know that $n \leq n^2$ for all $n \geq 1$. Also, $16 \leq n^2$ for $n \geq 4$. So

   $$3n^2 + n + 16 \leq 3n^2 + n^2 + n^2 = 5n^2 \quad \text{for all } n \geq 4.$$

$f(n) = O(g(n))$ IF $\exists\, c, n_0$ S.T. $f(n) \leq c\, g(n)$ $\forall\, n \geq n_0$

$c = 5, n_0 = 4$

Further examples of Big-Oh

1. $3n^2 + n + 16$ is $O(n^2)$. Why?

   We know that $n \leq n^2$ for all $n \geq 1$. Also, $16 \leq n^2$ for $n \geq 4$. So

   $$3n^2 + n + 16 \leq 3n^2 + n^2 + n^2 = 5n^2 \quad \text{for all } n \geq 4.$$

2. $13n^3 + 7n \log n + 3$ is $O(n^3)$. Why?

   Because $\log n \leq n \leq n^2$ for all $n \geq 1$, and for similar reasons as above we may conclude that

   $$13n^3 + 7n \log n + 3 \leq 21n^3 \quad \text{for all "large enough" } n.$$

Further examples of Big-Oh

1. $3n^2 + n + 16$ is $O(n^2)$. Why?

   We know that $n \leq n^2$ for all $n \geq 1$. Also, $16 \leq n^2$ for $n \geq 4$. So

   $$3n^2 + n + 16 \leq 3n^2 + n^2 + n^2 = 5n^2 \quad \text{for all } n \geq 4.$$

2. $13n^3 + 7n \log n + 3$ is $O(n^3)$. Why?

   Because $\log n \leq n \leq n^2$ for all $n \geq 1$, and for similar reasons as above we may conclude that

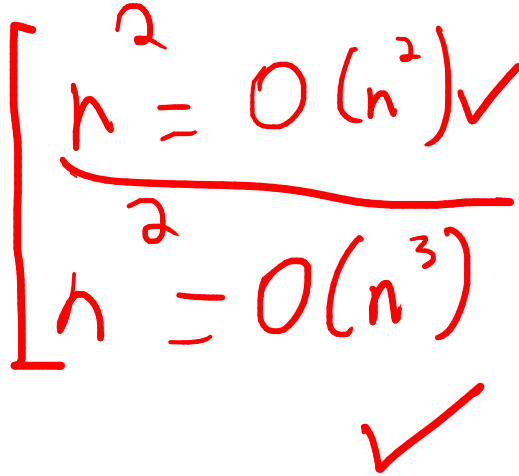
   $$13n^3 + 7n \log n + 3 \leq 21n^3 \quad \text{for all "large enough" } n.$$

3. Any polynomial $a_k n^k + \ldots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$, is $O(n^k)$.

Further examples of Big-Oh (cont.)

4. Also, any polynomial $a_k n^k + \ldots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$, is also $O(n^j)$ for all $j \geq k$.

5. Therefore, $45n^5 - 10n^2 + 6n - 12$ is $O(n^5)$ (and is also $O(n^8)$, or $O(n^9)$, or, indeed, $O(n^k)$ for any $n \geq 5$).



Further examples of Big-Oh (cont.)

4. Also, any polynomial $a_k n^k + \ldots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$, is also $O(n^j)$ for all $j \geq k$.

5. Therefore, $45n^5 - 10n^2 + 6n - 12$ is $O(n^5)$ (and is also $O(n^8)$, or $O(n^9)$, or, indeed, $O(n^k)$ for any $n \geq 5$).

6. $\sqrt{n}$ is $O(n)$.

7. $3 \log n + \log \log n$ is $O(\log n)$.

8. $\log n$ is $O(n)$, and $\log n$ is also $O(\sqrt{n})$.

$n^j = O(n^k)$

$\forall j \leq k$

$\log n = O(n^c)$

$\forall c > 0.$

Further examples of Big-Oh (cont.)

4. Also, any polynomial $a_k n^k + \ldots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$, is also $O(n^j)$ for all $j \geq k$.

5. Therefore, $45n^5 - 10n^2 + 6n - 12$ is $O(n^5)$ (and is also $O(n^8)$, or $O(n^9)$, or, indeed, $O(n^k)$ for any $n \geq 5$).

6. $\sqrt{n}$ is $O(n)$.

7. $3 \log n + \log \log n$ is $O(\log n)$.

8. $\log n$ is $O(n)$, and $\log n$ is also $O(\sqrt{n})$.

9. 2^{70} is $O(1)$.

10. $5/n$ is $O(1/n)$, and $5/n$ is also $O(1/\sqrt{n})$. (Why?)

$\frac{5}{n} = O\left(\frac{1}{n}\right)$

Common functions

- *Logarithmic* $O(\log n)$ → BIN SEARCH
- *Linear* $O(n)$ → SEARCH / MIN FINDING
- *Quadratic* $O(n^2)$ → BUBBLESORT
- *Polynomial* $O(n^k)$ for a positive integer k
- *Exponential* $O(a^n)$, $a > 1$ → $n^c = O(2^n)$ $\forall c > 0$

$\Omega(n)$ and $\Theta(n)$ notation

- We say that $f(n)$ is $\Omega(g(n))$ (*big-Omega*) if there are real constants c and n_0 such that:

$$f(n) \geq cg(n) \text{ for all } n \geq n_0.$$

$f(n) = \Omega(g(n))$

$\Updownarrow$

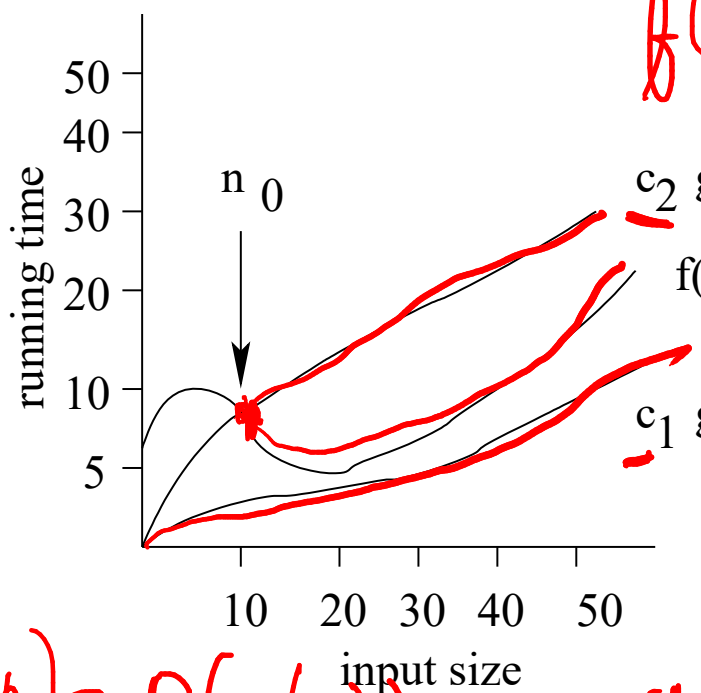
$g(n) = O(f(n))$

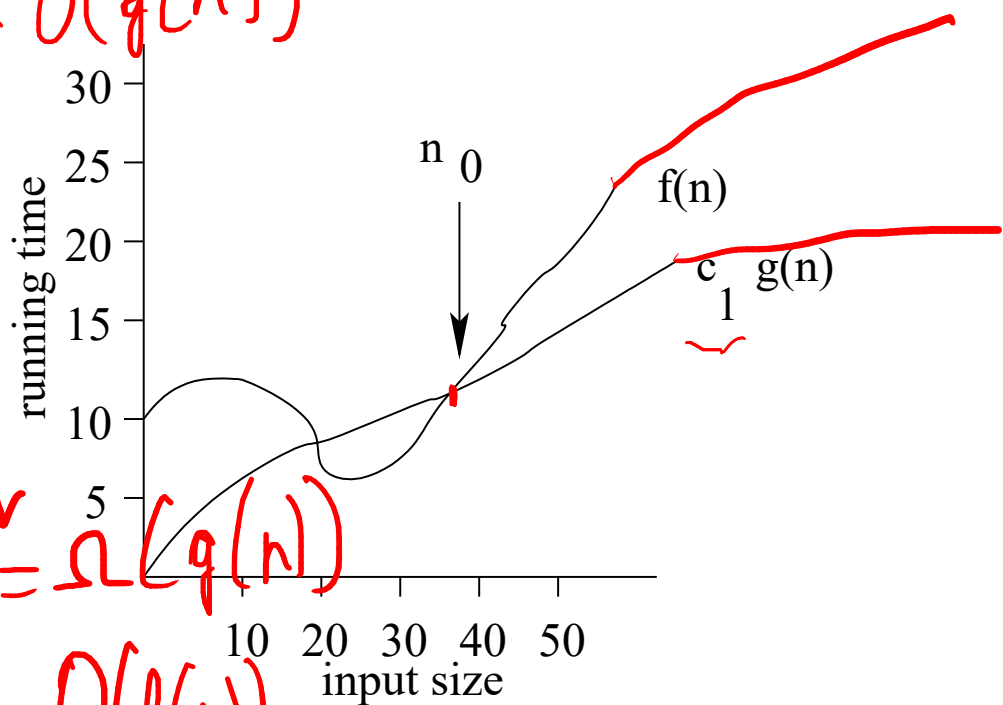
$\Omega(n)$ and $\Theta(n)$ notation

- We say that $f(n)$ is $\Omega(g(n))$ (*big-Omega*) if there are real constants c and n_0 such that:

$$f(n) \geq cg(n) \text{ for all } n \geq n_0.$$

- We say that $f(n)$ is $\Theta(g(n))$ (*Theta*) if $f(n)$ is $\Omega(g(n))$ and $f(n)$ is also $O(g(n))$. (Equivalently, $g(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$.)





Examples

ALSO $O(\log n)$

$0 \leq \log\log n \leq \log n$ $\forall n \geq 2$

1. $3\log n + \log\log n$ is $\Omega(\log n)$.
2. $\frac{2}{3}n^2 - n \in \Omega(n^2)$.
3. $3\log n + \log\log n \in \Theta(\log n)$.
4. $2n^3 - n + 7 \in \Theta(n^3)$.
5. A polynomial $a_k n^k + \cdots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$ is $\Theta(n^k)$.

Examples

CLAIM: $n^3 = O(n)$.

PF(?):

$c = n^2$, $n_0 = 1$

1. $3 \log n + \log \log n$ is $\Omega(\log n)$.
2. $\frac{2}{3}n^2 - n \in \Omega(n^2)$.
3. $3 \log n + \log \log n \in \Theta(\log n)$.
4. $2n^3 - n + 7 \in \Theta(n^3)$.
5. A polynomial $a_k n^k + \cdots + a_2 n^2 + a_1 n + a_0$, with $a_k > 0$ is $\Theta(n^k)$.

Additional reading material: Chapter 3 in [CLRS].

Back to "finding the minimum"

We previously said that if an array A has n items, then the running time of our "minimum finding algorithm" takes $3n - 1$ primitive operations.

Thus, this is an algorithm that is $O(n)$.

Back to "finding the minimum"

We previously said that if an array A has n items, then the running time of our "minimum finding algorithm" takes $3n - 1$ primitive operations.

Thus, this is an algorithm that is $O(n)$.

However, this is best possible, since we have to examine each item in the list to really determine the minimum. (So it's a $\Omega(n)$ algorithm.)

Back to "finding the minimum"

We previously said that if an array A has n items, then the running time of our "minimum finding algorithm" takes $3n - 1$ primitive operations.

Thus, this is an algorithm that is $O(n)$.

However, this is best possible, since we have to examine each item in the list to really determine the minimum. (So it's a $\Omega(n)$ algorithm.)

Putting these observations together, our minimum finding algorithm on a list of n elements is a $\Theta(n)$ algorithm.

Importance of asymptotics

Maximum size allowed for an input instance for various running times to be solved in 1 second, 1 minute and 1 hour, assuming a 1MHz machine:

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$20n \log n$	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 rates (running time)

Functions ordered by growth rate:

$\log_2 n$	$\sqrt{n}$	n	$n \log_2 n$	n^2	n^3	2^n
1	1.4	2	2	4	8	4
2	2	4	8	16	64	16
3	2.8	8	24	64	512	256
4	4	16	64	256	4096	65536
5	5.7	32	160	1024	32768	4294967296
6	8	64	384	4096	262144	1.84^{19}
7	11	128	896	16384	2097152	2.40×10^{38}
8	16	256	2048	65536	16777216	1.15^{77}
9	23	512	4608	262144	134217728	1.34×10^{154}
10	32	1024	10240	1048576	1073741824	1.79^{308}

$\mathcal{NP}$-completeness (a brief mention for now)

- There are certain problems whose exact solutions seem hard to find, i.e. solving large instances of them could take decades or centuries of computing time since it appears we have to try (nearly) all of the **exponentially** many possible solutions.
- Many of these are, however, important problems that business and industry (airlines, manufacturers, telecommunications companies, etc.) are interested in solving, such as some scheduling and optimization problems. This "hardness" can also be described in terms of the game Minesweeper, the very one included with the Windows OS.

In the absence of *exact* solutions, good *approximations* are used, we will see examples of these towards the end of this module.

$\mathcal{NP}$-completeness (cont.)

Certain problems, the so-called class of $\mathcal{NP}$-complete problems, are the "hardest" ones.

Somewhat surprisingly, if *any one* of these special problems can be solved efficiently, then *all* of them can be solved efficiently. So these problems are equivalent in a formal computational sense.

We will examine $\mathcal{NP}$-completeness in closer detail later on in the course.

$\mathcal{NP}$-completeness (cont.)

The class $\mathcal{P}$: problems that are efficiently *solvable*.
The class $\mathcal{NP}$: problems that are efficiently *verifiable*.
Is $\mathcal{P} = \mathcal{NP}$?

$\mathcal{NP}$-completeness (cont.)

The class $\mathcal{P}$: problems that are efficiently *solvable*.
The class $\mathcal{NP}$: problems that are efficiently *verifiable*.
Is $\mathcal{P} = \mathcal{NP}$?

This is such an important and fundamental problem that it's been named as one of the seven "Millennium Problems" by the Clay Institute and can earn you \$1 million dollars for its solution (and a place in mathematical and computer science history).

Go to the website

`https://www.claymath.org/millennium/p-vs-np/`

to read more.

Simple Justification Techniques

Here's some proof techniques that we will likely use in this course.

- Proof by *mathematical induction*.
- Proof by *contradiction*.
- Proof by *contrapositive* argument.
- (Dis)Proof by *counterexample*.

Examples of results provable by induction

1. $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ for all $n \geq 1$.
2. $1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$ for all $n \geq 1$.
3. $2^n \geq n^2$ for $n \geq 5$.

Outline of a proof by induction:

- Base case: Show hypothesis true for smallest value of n.
- Inductive step: assume hypothesis to be true for $n = k$ and show it true for $n = k + 1$.
- Example: first bullet above:

$$1 + 2 + \cdots + k + (k + 1) = \frac{k(k+1)}{2} + k + 1 = \frac{(k+1)(k+2)}{2}.$$

Proof by *contradiction*

Proposition

If ab is odd and a, b are natural numbers, then a is odd or b is even.

Proof: Suppose the proposition is not true. Then, a is even AND b is odd (DeMorgan's Law).
Thus $a = 2i$ for some integer i.
Hence $ab = (2i)b = (2i)b$, which is even.
But ab cannot be simultaneously odd and even (this is a contradiction).

Proof by *contrapositive*

Statement 1: If it is raining, I am not playing football.
Statement 2: If I am playing football, it is not raining.

Proof by *contrapositive*

Statement 1: If it is raining, I am not playing football.
Statement 2: If I am playing football, it is not raining.

Proposition

If ab is odd and a, b are natural numbers, then a is odd or b is even.

Note that $p \implies q$ is equivalent to $\neg q \implies \neg p$. **Proof**: The contrapositive is 'If a is even and b is odd, then ab is even.'
We will prove the contrapositive. So, suppose $a = 2i$ for some integer i. Then $ab = (2i)b = 2(ib)$, which is even.

(Dis)Proof by *counterexample*

Claim

If $i > 1$ is an integer, then $2^i - 1$ is a prime.

(Dis)Proof: Consider $i = 4$. We have $2^4 - 1 = 15 = 3 \cdot 5$.