LECTURER: TAI LE QUY

ALGORITHMS, DATA STRUCTURES, AND PROGRAMMING LANGUAGES

TOPIC OUTLINE

TOPIC OUTLINE

UNITS 2 & 3

DATA STRUCTURES

ALGORITHM DESIGN

STUDY GOALS

- Understand Data Structures: Stack, Queue, Heap, Graph.
- Understand concepts of ADTs, objects, and classes.
- Apply different types of polymorphism.
- Understand the use of iteration and recursion.
- Understand basic algorithm design paradigms.
- Understand program verification methodologies.
- Understand analysis of algorithms and notations.

EXPLAIN SIMPLY

1. Describe how stacks are used in function invocation.
2. Explain what is polymorphism in object-oriented programming.
3. Suppose we have to choose between two algorithms whose running times are given by $f(n) = n^3$ and $g(n) = 2^n$, where n is the size of the input. Notice that $g(n) < f(n)$ for $1 < n < 10$. Explain why we may choose the first algorithm in this case.

STACKS

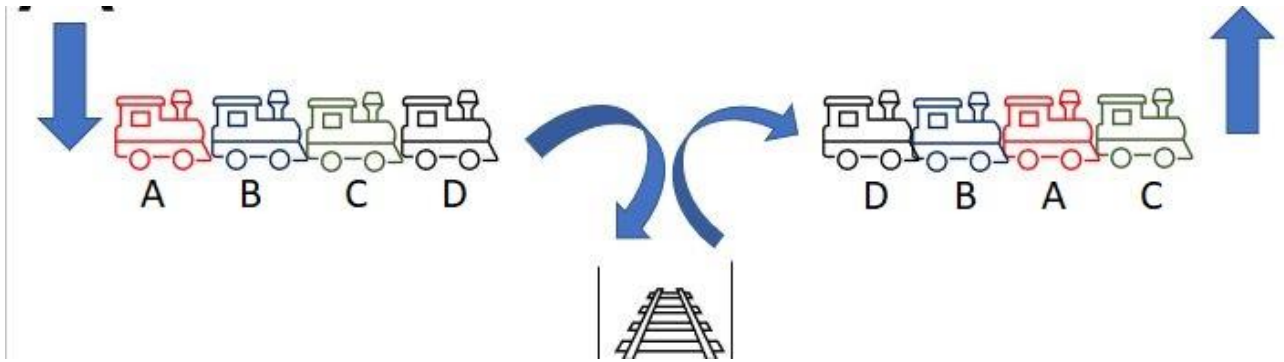
Stack is a collection of data items following a **LIFO** (Last-In-First-Out) paradigm. Basic stack operations include:

- **push (element):** adds element to the top of the stack.
- **pop():** removes element from the top of stack.
- **topOfStack():** reads element from the top of the stack.
- **isEmpty():** checks if stack is empty.
- **size():** returns the number of elements.

Source of the image: Pexels.

STACK APPLICATIONS

Stack frames are used in function invocation. Stack frames are created when a function is invoked and get pushed onto the call stack. Later, as the called function returns control to the calling function, the stack frame is popped from the call stack.
Stacks are also used to switch railroad cars of a train in a switching yard. In the figure, cars arrive at the switching yard in the order D, C, B and A and get switched using the stack-like structure and they leave the yard in the order C, A, B and D.

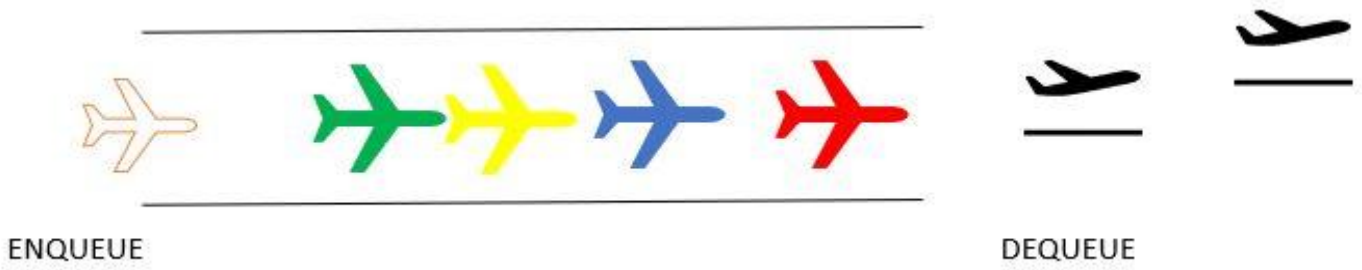


Queue is a data structure following a **FIFO** (First In, First Out) paradigm .

The basic supported operations of a queue are as follows:

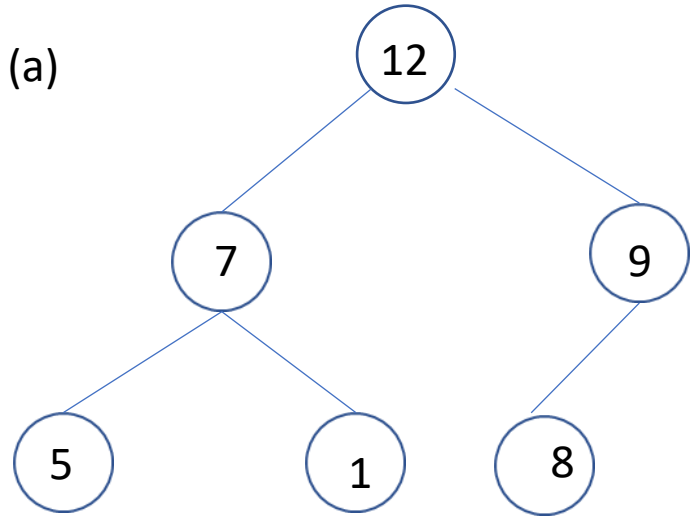
- enQueue (element): adds element to the rear of the queue.
- deQueue(): removes element from the front of the queue.
- isEmpty(): checks if stack is empty and returns True/False.
- size(): returns the number of elements in the queue.

An example of a queue is the departure queue of flights taking off from a runway.



Heap is a data structure used as building block in **Heapsort** and Priority Queue operations**.**

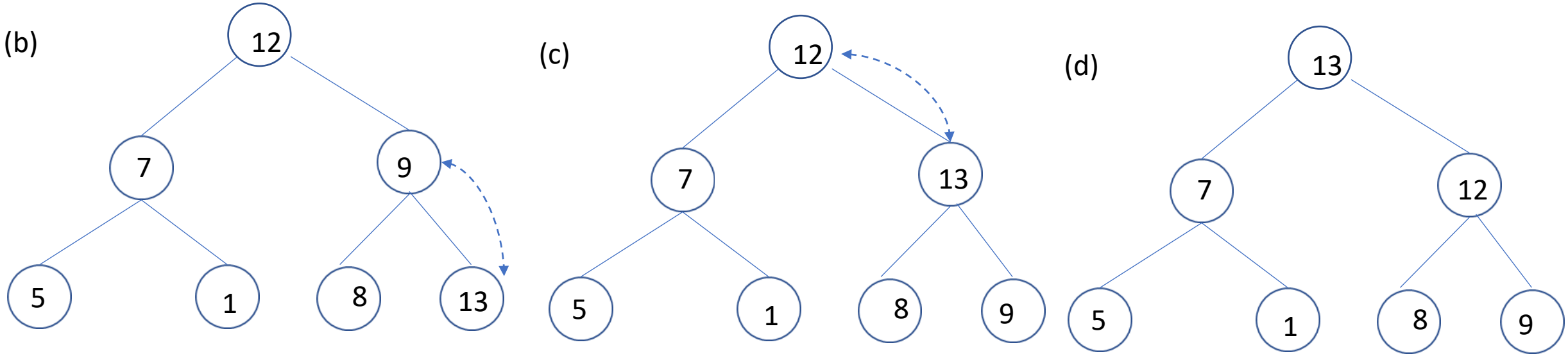
- **T**, a complete binary tree with root **r** is a MAX Heap if
- **T** is NULL OR
  (a) Key(r) ≥ max (key(Left(r)),key(Right(r))
  (b) The subtrees rooted at Left(r) and Right(r) are heaps.
- Similarly a MIN Heap may be defined.



A MAX Heap

The Heap supports the following basic operations:

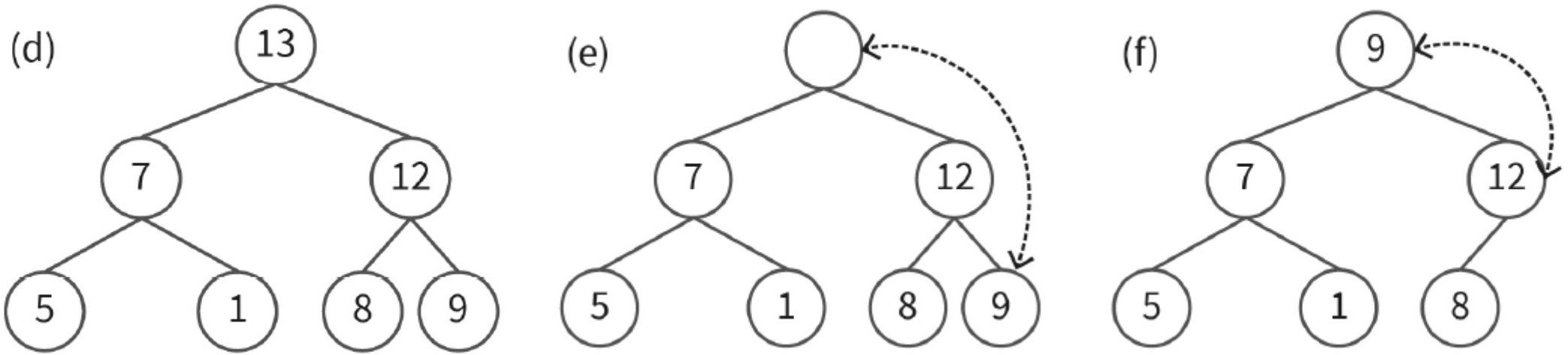
- insert(element) which adds an element to the Heap.
- extractMax(): which extracts (remove) an element from the Heap.
- isEmpty(): returns True if Heap is empty.
- size(): returns the number of elements in the Heap.



insert(13) to the Heap

HEAP OPERATIONS

– extractMax(): which extracts (remove) an element from the Heap.



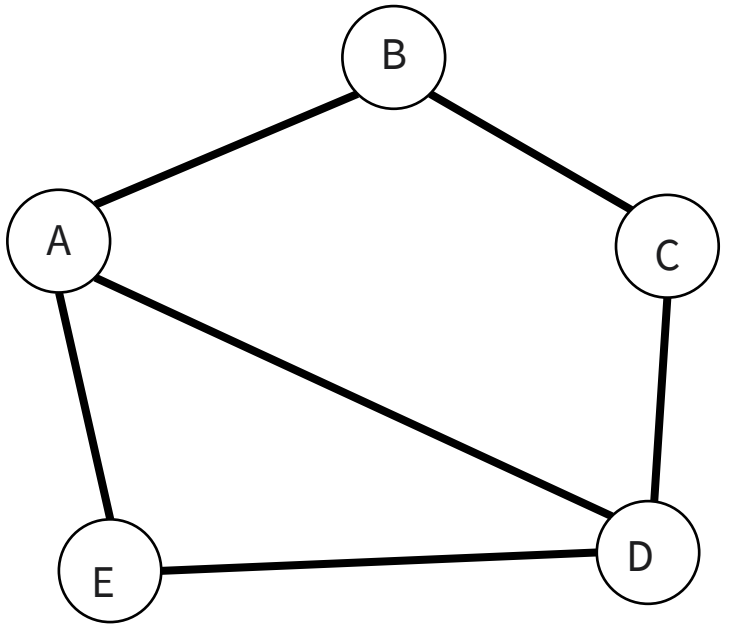
extractMax(13) from the Heap

Source of figures: course book

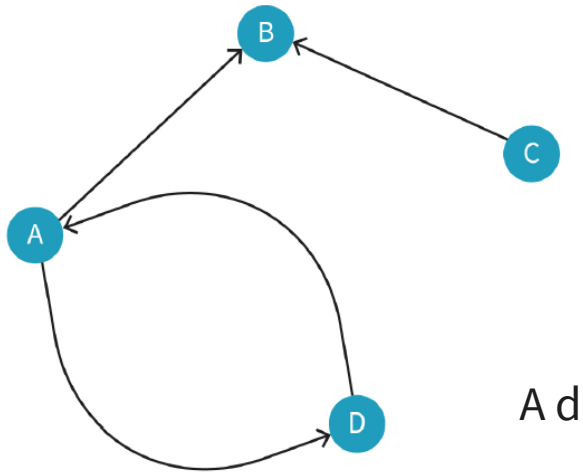
Graphs are used in several applications

- Social networks
- Road networks
- Web pages and their hyperlinks
- Communication networks
- Collaboration networks
- Airline routes

A directed graph has only directed edges between pairs of vertices. An undirected graph has no directed edges.

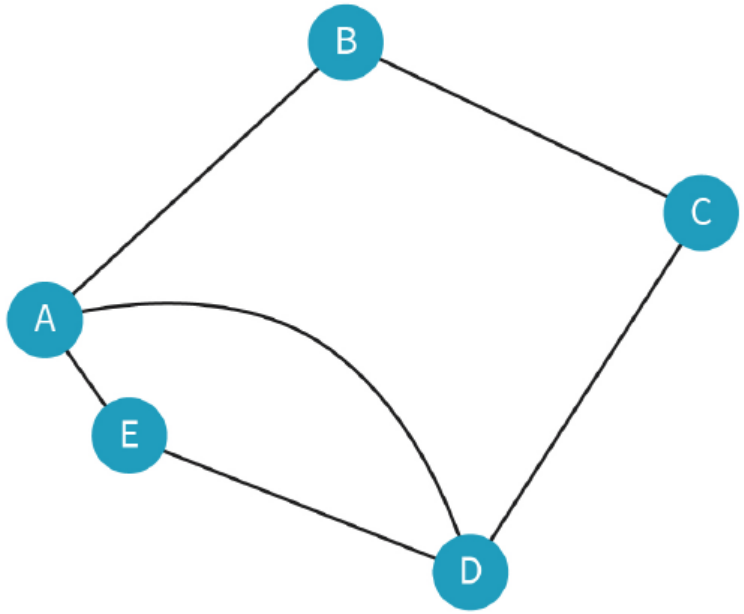


A **graph** $G = (V, E)$ consists of a set of vertices V and a set of edges $E \subseteq V \times V$

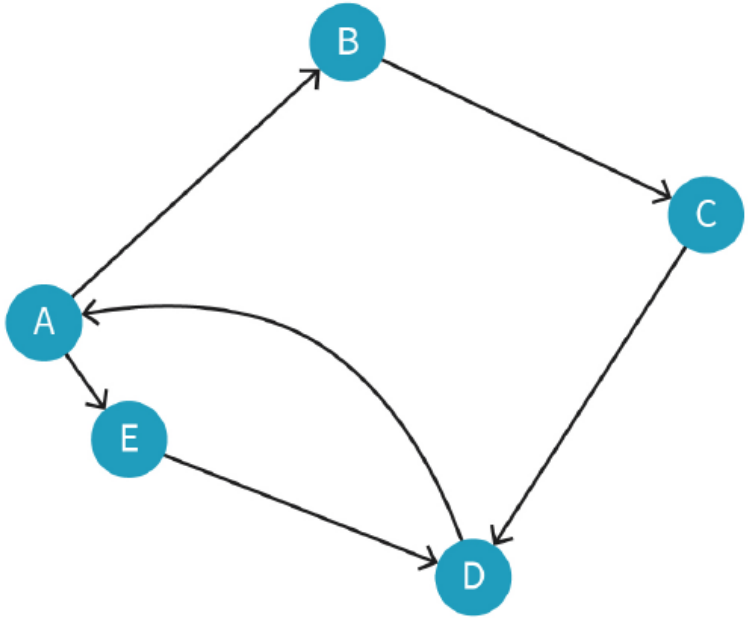


A directed graph

– A cycle is a sequence of vertices such that every consecutive pair in the sequence is connected by an edge, and the last vertex in the sequence is connected to the first



An Undirected Graph with Cycles



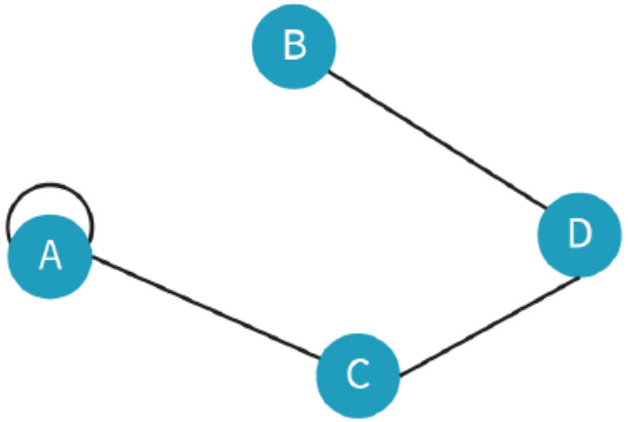
A Graph with Directed Cycles

Source of figures: course book

Adjacency list

Undirected graph G = (V, E)

Adj(v) = {u| (v, u) ϵ E}, (u, v) ϵ E if (v, u) ϵ E

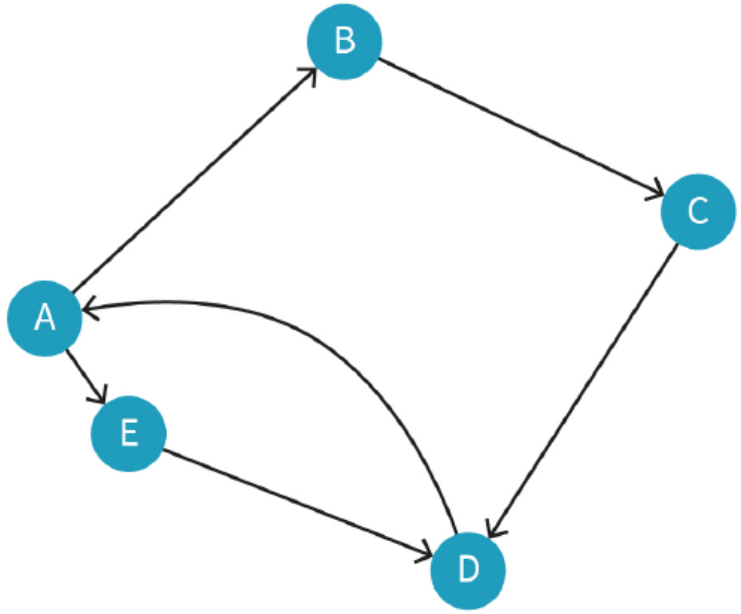


Adj (A) = {A,C}
Adj (B) = {D}
Adj (C) = {A,D}
Adj (D) = {B,C}

The graph is **sparse** (|E| = O(|V|)), the graph is **dense** |E|= O($|V|^2$)

Directed graph G = (V, E)

Adj(v) = {u| (v, u) ϵ E}



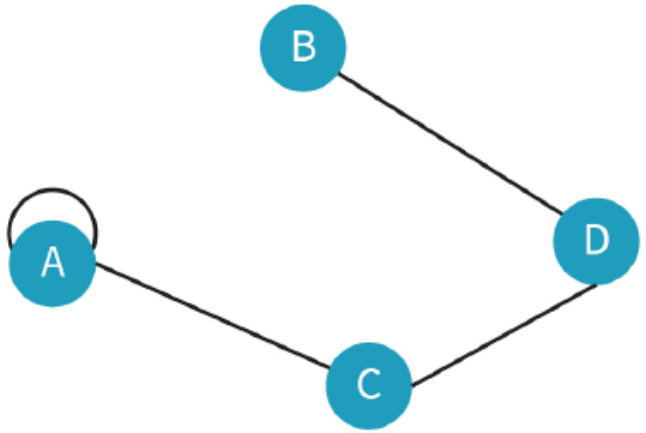
v	Adj(v)
A	B,E
B	C
C	D
D	A
E	D

In some applications, Adj(v) may store incoming edges at v.

Source of figures: course book

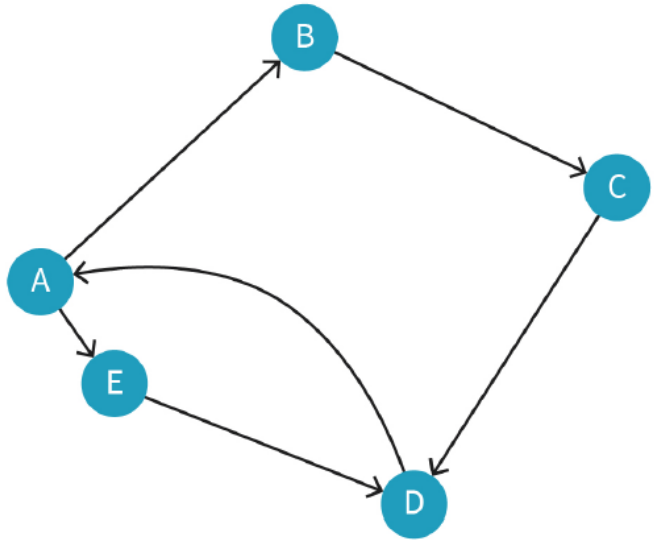
Adjacency Matrix

- The adjacency matrix A is a two-dimensional Boolean matrix where A(i, j) = 1 if, and only if, v(i), v(j) is an edge in the graph
- Preferred representation if the graph is **dense**



1	A	B	C	D
A	1	0	1	0
B	0	0	0	1
C	1	0	0	1
D	0	1	1	0

Adjacency Matrix of an Undirected Graph



0	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	0	0
C	0	0	0	1	0
D	1	0	0	0	0
E	0	0	0	1	0

- Space requirement: Adjacency matrices $O(|V|^2)$, adjacency lists $O(|V|^2 |E|)$

Source of figures: course book

ADT

The ADT (abstract data type) for a data structure specifies what is stored in the data structure and what operations are supported on them but does not detail how the operations are implemented.

Class

A fundamental means of abstraction in object-oriented programming.

Determines how information is represented.

Has fields or members to represent information.

Defines behavior in the form of methods or member functions.

An object is an instantiation of a class.

Inheritance	Powerful feature in Object-oriented programming.
	Facilitates modular and hierarchical organization.
	Enables us to define new classes based on the existing class.
	Subclass: The new class which is derived.
	Base class: The existing class from which the subclass is derived.
SubClass	Inherits some methods from the base class.
	Extends the base class with new methods.
	Overrides some methods from the base class.

Polymorphism is the ability to reuse the same code for different kind of data.

Ad-hoc polymorphism

- Also known as function overloading.
- Function has different implementations based on types of arguments.
- Support in Python for built-in types and user-defined classes.

Polymorphism with inheritance

- If a method is implemented in some classes in the inheritance hierarchy, the implementation in the nearest superclass is invoked.

UNITS 3

ALGORITHM DESIGN

- “Iterations” are repetitive computations of a group of statements.
- Programming languages support various mechanisms for “loops”
- Languages also provide different loop control mechanisms, such as “break” and “continue”, etc.

The while loop

The statements are executed for as long as the condition is true

Python

```
while(condition):
      statements
```

C++

```
do {
statements;
} while (condition);
```

The for loop

The statements are executed for as long as the condition is true

Python

```
for variable in sequence:
        statements
else: #optional and executed when the for loop exits normally
        statements
```

C++

```
for(expression; expression; expression) statements
```

- User-controlled mechanisms
  - In Python: break and continue
  - break: allows the control to exit the loop
  - continue: allows the control to skip the rest of the statements in the loop body and return to the start of the loop
- Iterator: user-defined functions that traverse a data structure in some sequence
- Generator: an alternative to a traditional function and are suitable when we need the results one by one
- Recursion: a function makes one or more calls to itself, trying to express the solution to a problem in terms of solutions to smaller subproblems

– “Mathematical induction” is a fundamental tool in the design and analysis of algorithms.

Weak induction

Suppose we need to prove that a property $P(n)$ is true for all non-negative integers $n \geq 0$.
The steps are as follows:

- basis. Show that $P(0)$ is true.
- induction step. Show that for all $n \geq 1$, if $P(n-1)$ is true, then $P(n)$ is true.

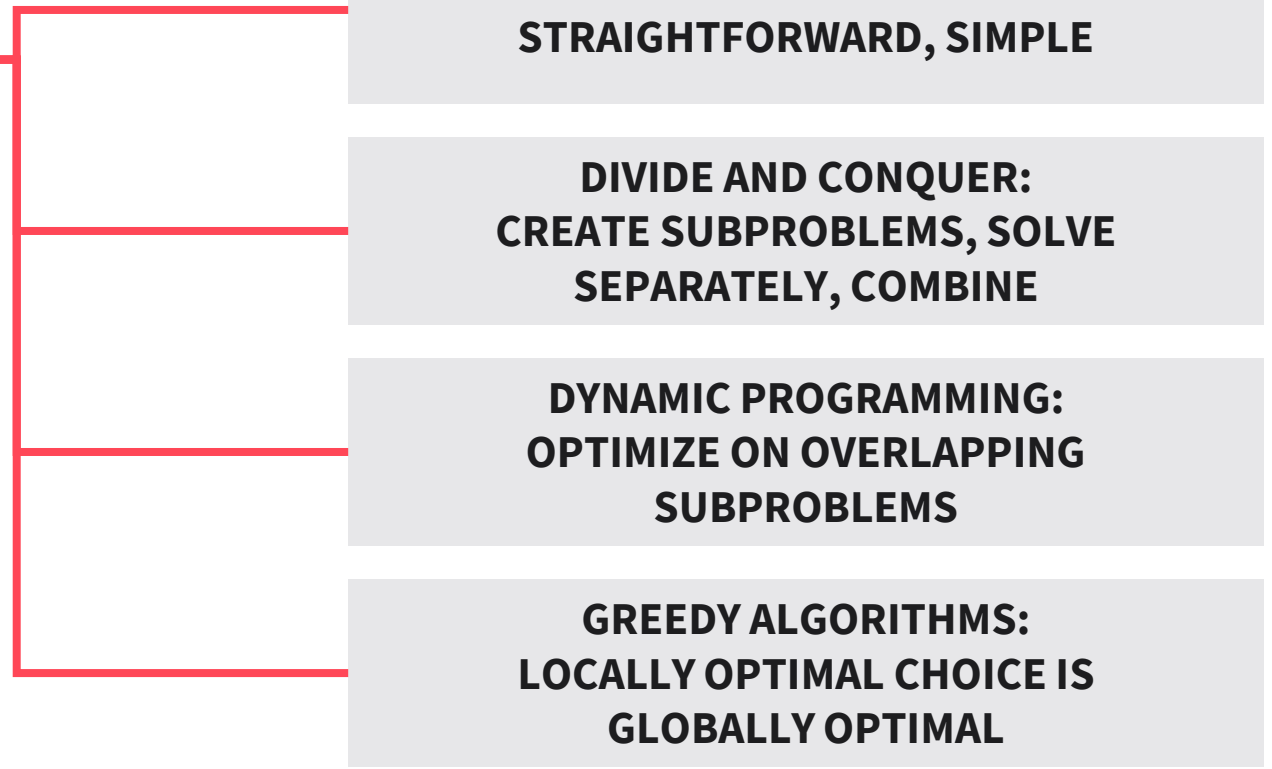
Strong induction

In this case, to prove that a property $P(n)$ is true for all non-negative integers $n \geq 0$, the steps are as follows:

- basis. Show that $P(0)$ is true.
- induction step. Show that for all $n \geq 1$, if $P(k)$ is true for all $k < n$, then $P(n)$ is true.

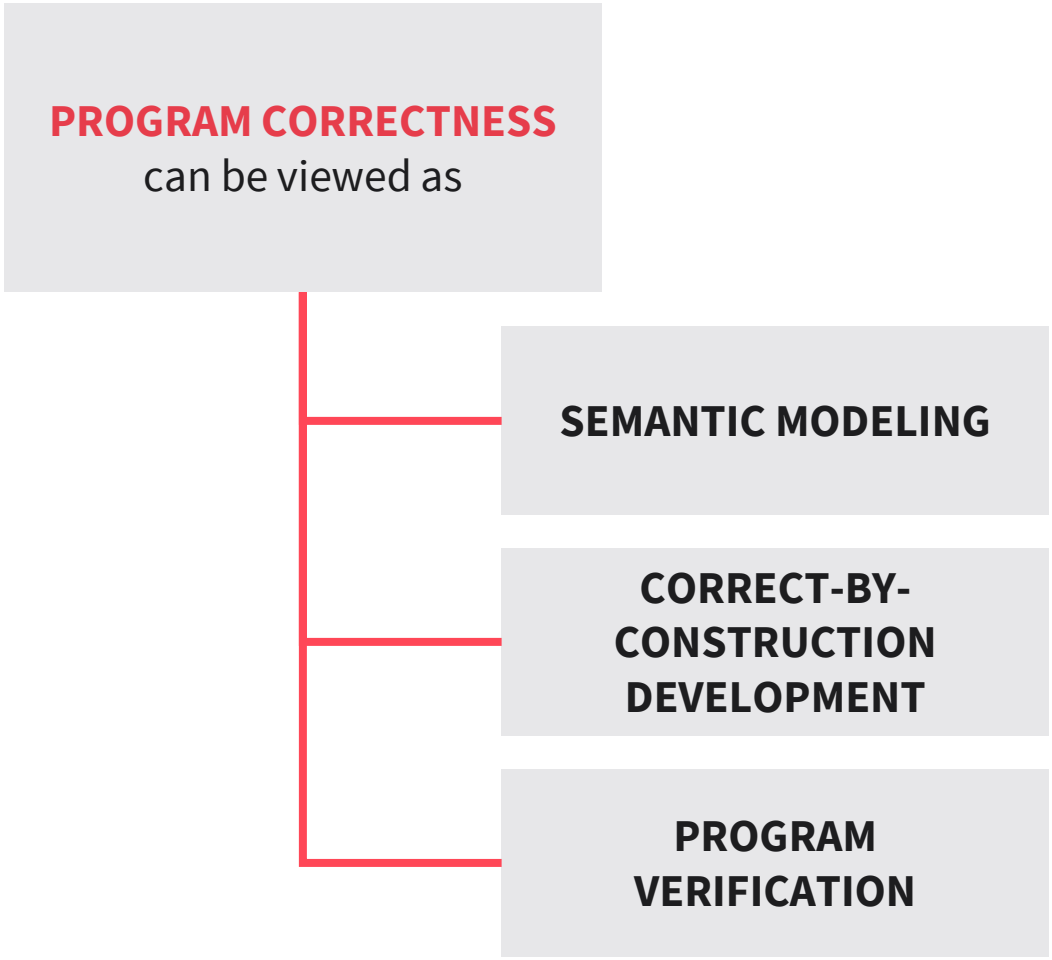
ALGORITHM DESIGN STRATEGIES
Basic techniques

Each problem is different, but some general strategies have been found to be useful for a large class of problems.

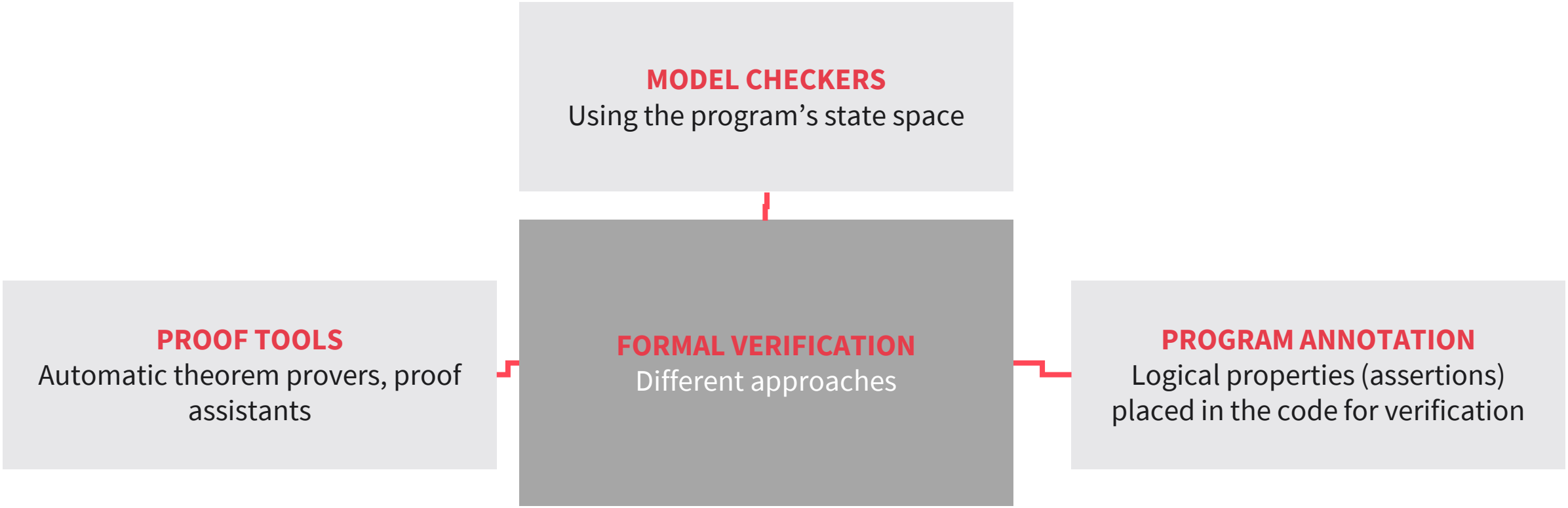


PROGRAM VERIFICATION

- Research has focused on development of rigorous techniques for specification, development and verification of software systems.
- These can help in errors being detected early or being eliminated.
- But formal verification methods only verify whether the system is correct with respect to the specification.
- Specifications themselves may be faulty.



FORMAL VERIFICATION APPROACHES



EFFICIENCY OF ALGORITHMS

Asymptotic upper bound

We define $O(g(n)) = \{f(n)$: There exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n) \, \forall \, n \geq n_0\}$. The expression $f(n) = O(g(n))$ denotes the membership of $f(n)$ in the set $O(g(n))$ (Cormen et al., 2009).

Asymptotic lower bound

We define $\Omega(g(n)) = \{f(n)$: There exist positive constants c and n_0 such that

$0 \leq c \cdot g(n) \leq f(n) \, \forall \, n \geq n$. The expression $f(n) = \ \Omega \ (g(n))$ denotes the membership of $f(n)$ in the set $\Omega \ (g(n))$ (Cormen et al., 2009).

Asymptotic tight bound

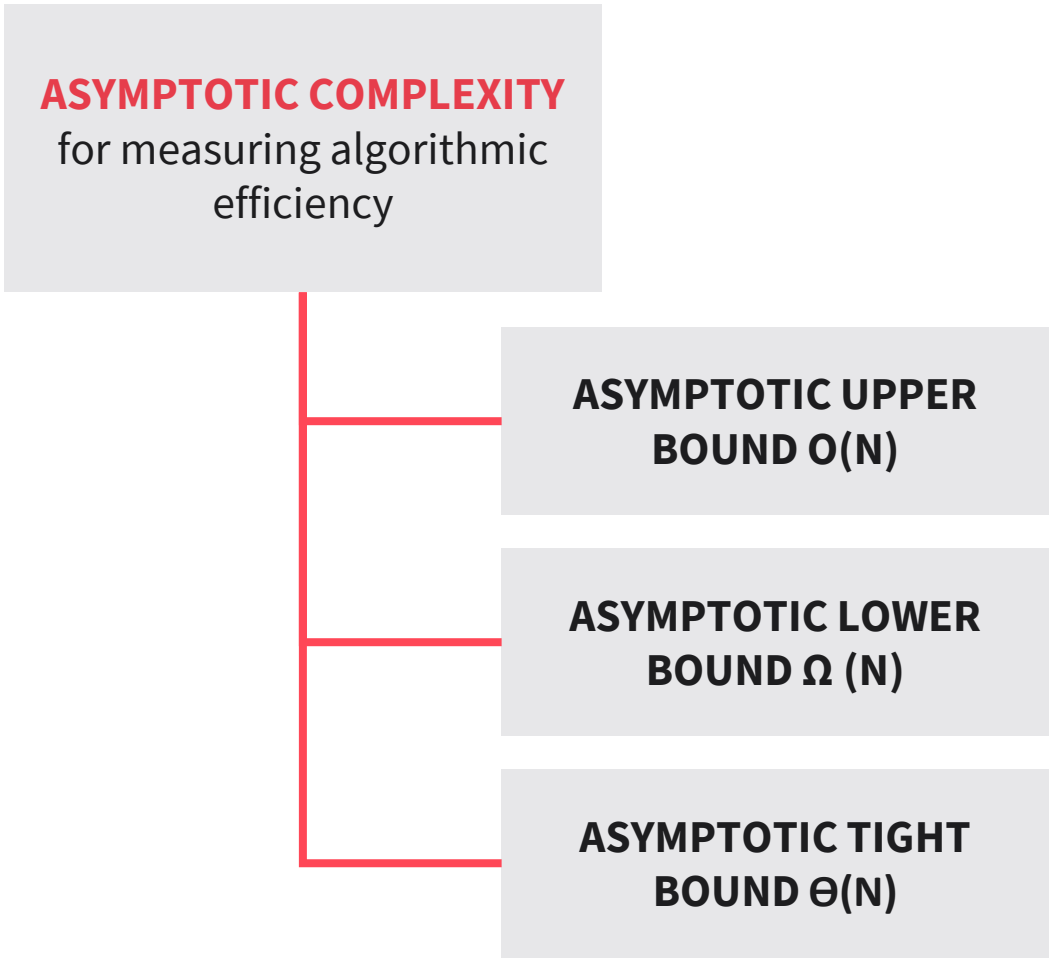
We define $\Theta(g(n)) = \{f(n)$: There exist positive constants c_1, c_2, and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \, \forall \, n \geq n_0\}$. The expression $f(n) = \Theta(g(n))$ denotes the membership of $f(n)$ in the set $\Theta(g(n))$ (Cormen et al., 2009).

Note that $f(n) = \Theta(g(n))$ if, and only if, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ (Cormen et al., 2009).

Source of the text: Course book

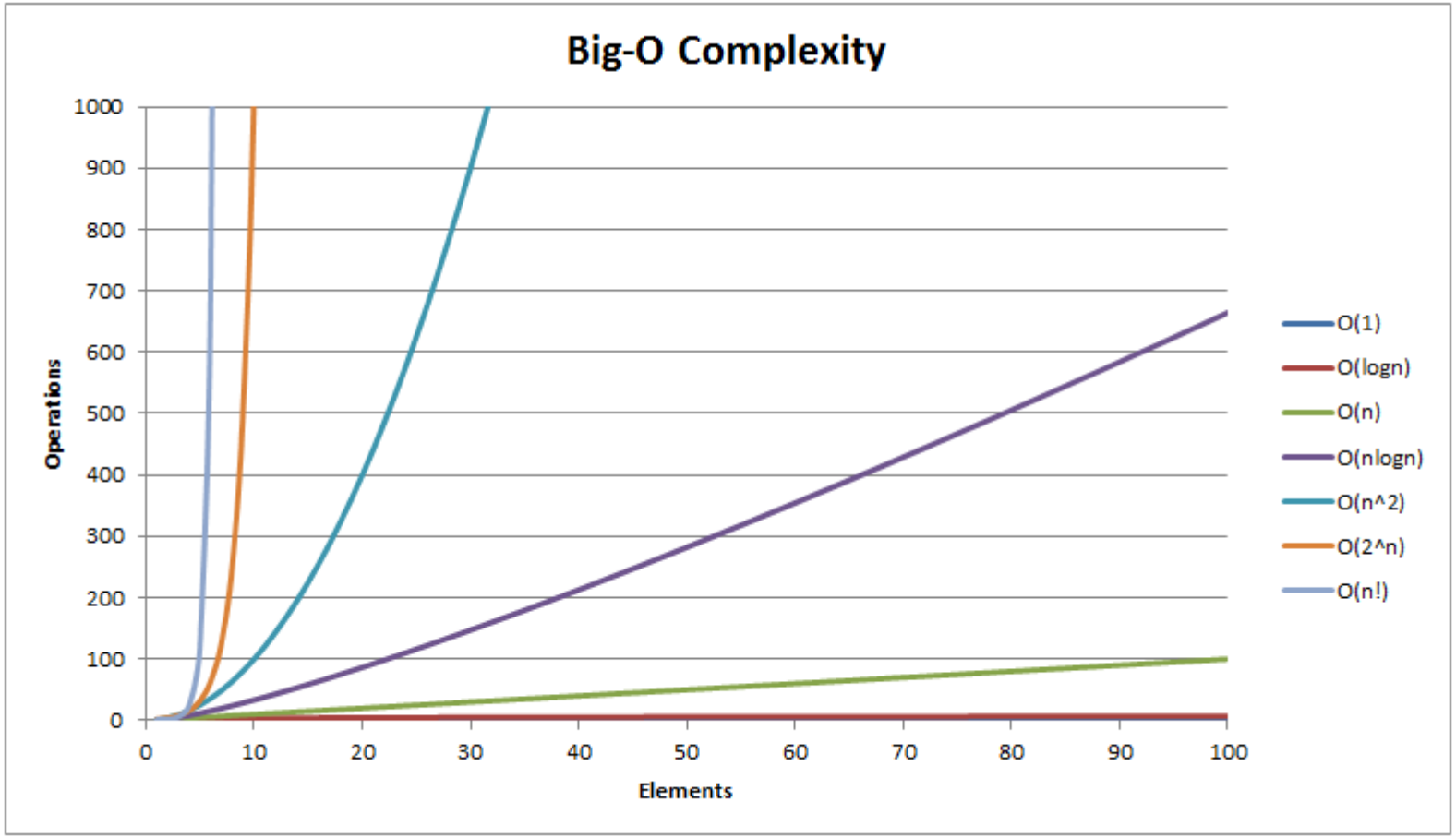
Asymptotic complexity

- The asymptotic complexity captures the practical notion that we are interested in comparisons for sufficiently large n.
- If we have to choose between two algorithms whose running times are given by $f(n) = n^3$ and $g(n) = 2^n$, where n is the size of the input, we choose the former even though $g(n) < f(n)$ for $1 < n < 10$.
- This is because asymptotic complexity matters more.
- P ≠ NP question



COMMON COMPLEXITY MEASURES

- constant: O(1)
- logarithmic: O(logn)
- log: O(log logn)
- linear: O(n)
- n-logn: O(n · logn)
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$
- exponential: $O(2^n)$
- factorial: O(n!)



Source of image: https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/

REVIEW STUDY GOALS

- Understand Data Structures: Stack, Queue, Heap, Graph.
- Understand concepts of ADTs, objects, and classes.
- Apply different types of polymorphism.
- Understand the use of iteration and recursion.
- Understand basic algorithm design paradigms.
- Understand program verification methodologies.
- Understand analysis of algorithms and notations.

SESSION 2

TRANSFER TASK

TRANSFER TASKS

1. We have a road network of a city. How do you represent this as a graph? Give examples of some queries which will be useful to support for this graph.
2. Now assume you have infomation about bus stops in the city. You also have information about bus routes in terms of the sequence of bus stops they connect. How do you represent this infomation as a graph? Give examples of some queries which will be useful to support for this graph.

TRANSFER TASK
PRESENTATION OF THE RESULTS

Please present your results.

The results will be discussed in plenary.

1. Which of the following data-structure is most suitable for implementing a priority queue?
   a) Heap
   b) Queue
   c) Tree
   d) Stack

2. Suppose that, in an inheritance hierarchy of classes in Python, A is a subclass of B and B is a subclass of C. A method f() is defined in all three classes. If an object b of class B invokes b.f(), which class's defined method will be invoked?
   a) A only
   b) B only
   c) C only
   d) Both A and B

3. Which algorithm design strategy involves solving overlapping subproblems only once and reusing the results?
   a) Greedy algorithms
   b) Divide and conquer
   c) Decrease and conquer
   d) Dynamic programming

LIST OF SOURCES

Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Goodrich, M.T., Tamassia, R., & Goldwasser, M.H. (2013). *Data Structures and Algorithms in Python.* Wiley.

Levitin, A. (2012). *Introduction to the Design and Analysis of Algorithms* (3rd ed.). Pearson.

Liang, Y.D. (2017). *Introduction to Programming Using Python.* Pearson.