# Lecture 1

# Introduction

## CS220/SE220 Design and Analysis of Algorithms

Slides are based on the textbook and its notes

## Overview

- This lecture gives you the general information of the course, the concept of algorithms as well as representations of dynamic sets by simple data structures.

- We will look at rudimentary data structures: arrays, matrices, stacks, queues, linked lists, rooted trees.

## General information

- **Course Unit:** 3
- **Course Type:** *Compulsory* for both **Computer Science (CS)** and **Software Engineering (SE)** majors
- **Instructor:** **Professor Hon-Cheng WONG (黃漢青 教授)**


Research Interests: Graphics, Vision, and Image Processing

Office: Room A316

E-mail: hcwong@must.edu.mo


Office Hours: Mon: 10:30 ~ 12:30; Tue: 10:30 ~ 12:30; Wed: 13:30 ~ 17:30; Thu: 10:30 ~ 12:30; or by appointment.

## Objectives

- This is a theory course covering algorithm design and analysis principles as well as basic data structures.
- We will study a series of specific algorithms designed to solve common problems.
- This is not a programming course. But you are encouraged to study the Python programs provided by the textbook.

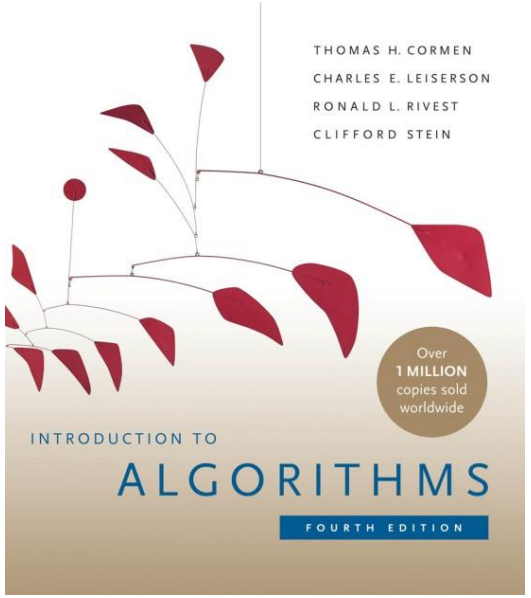
## Grading scheme

- Class participation: 5%
- Midterm: 30% (Closed book and notes)
- Final exam: 65% (Closed book and notes)

- We will have written exercises, **but they are NOT collected for grading**. We will provide you reference solutions.

# Course material, textbook, and references

- **Course material**:
  You can get them from Moodle via WeMust.




- **Required textbook**
  T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, Fourth Edition, MIT Press, 2022.
  https://mitpress.mit.edu/books/introduction-algorithms-fourth-edition

- **References**

1. J. V. Guttag. *Introduction to Computation and Programming Using Python*, Third Edition, MIT Press, 2021.
2. A. Levitin. *Introduction to the Design and Analysis of Algorithms*, Third Edition, Tsinghua University, 2013.

# Lecture schedule

| Index | Topics* | Reading |
|---|---|---|
| 1 | Introduction; Elementary data structures | Ch.1, Ch.10 |
| 2 | Analyzing and designing algorithms | Ch.2 |
| 3 | Characterizing running times | Ch.3 |
| 4 | Divide-and-conquer | Ch.4 |
| 5 | Probabilistic analysis and randomized algorithms | Ch.5 |
| 6 | Quicksort | Ch.7 |
| 7 | Medians and order statistics; Heapsort | Ch.9, Ch.6 |
| 8 | Midterm | ------ |
| 9 | Sorting in linear time | Ch.8 |
| 10 | Binary search trees; Greedy algorithms | Ch.12, Ch.15 |
| 11 | Dynamic programming | Ch.14 |
| 12 | Elementary graph algorithms | Ch.20 |
| 13 | Solve single-source shortest path problems | Ch.22 |
| 14 | Maximum flow | Ch.24 |
| 15 | Revision | Notes |

*Topics may be changed subject to the progress of the course.

# What is an algorithm?

- Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output*** in a finite amount of time. An algorithm is thus a sequence of computational steps that transform the input into the output.
- As an example, suppose that you need to sort a sequence of numbers into monotonically increasing order.
- Here is how we formally define the ***sorting problem***:

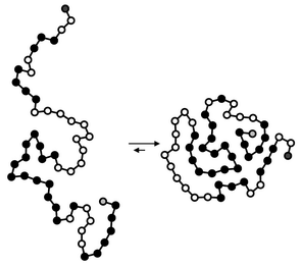
  **Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

  **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- Thus, give the input sequence <31, 41, 59, 26, 41, 58>, a correct sorting algorithm returns as output the sequence <26, 31, 41, 41, 58, 59>. Such an input sequence is called an ***instance*** of the sorting problem. In general, an ***instance of a problem*** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

# Why study algorithms?

- Their impact is broad and far-reaching.
- Internet: Web search, packet routing, distributed file sharing, ...
- Biology: Human genome project, protein folding, ...
- Computers: Circuit layout, file system, compilers, ...
- Computer graphics: Movies, video games, virtual reality, ...
- Security: Cell phones, e-commerce, voting machines, ...
- Multimedia: CD player, DVD, MP3, JPG, DivX, HDTV, ...
- Transportation: Airline crew scheduling, map routing, ...
- Physics: N-body simulation, particle collision simulation, ...

## Why study algorithms?

- Theoretical importance
  - The core of computer science.
- Practical importance
  - A practitioner's toolkit of known algorithms.
  - Framework for designing and analyzing algorithms for new problems.

# Simple array-based data structures: Arrays

- ***Arrays*** store elements contiguously in memory. If
  - the first element of an array has index $s$,
  - the array starts at memory address $a$, and
  - Each element occupies $b$ bytes,

  Then the $i$th element occupies bytes $a + b(i - s)$ through $a + b(i + 1 - s) - 1$. The most common values for $s$ are 0 and 1.
  - $s = 0 \Rightarrow a + bi$ through $a + b(i + 1) - 1$.
  - $s = 1 \Rightarrow a + b(i - 1)$ through $a + bi - 1$.
  - The computer can access any array element in constant time (assuming that the computer can access all memory locations in same amount of time).
  - If elements of an array occupy different numbers of bytes, elements might be accessed incorrectly or not in constant time.

# Simple array-based data structures: Matrices

- Notation: an $m \times n$ matrix has $m$ rows and $n$ columns.
- We represent a matrix with one or more arrays.
- Two common ways to store a matrix:
  - ***Row-major***: matrix is stored row by row.
  - ***Column-major***: matrix is stored column by column.
- ***Example***: Consider the $2 \times 3$ matrix

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

Row-major order: store two rows 1 2 3 and 4 5 6

Column-major order: store three columns 1 4; 2 5; and 3 6.

# Simple array-based data structures: Matrices

- There are four ways to store $M$ as follows:

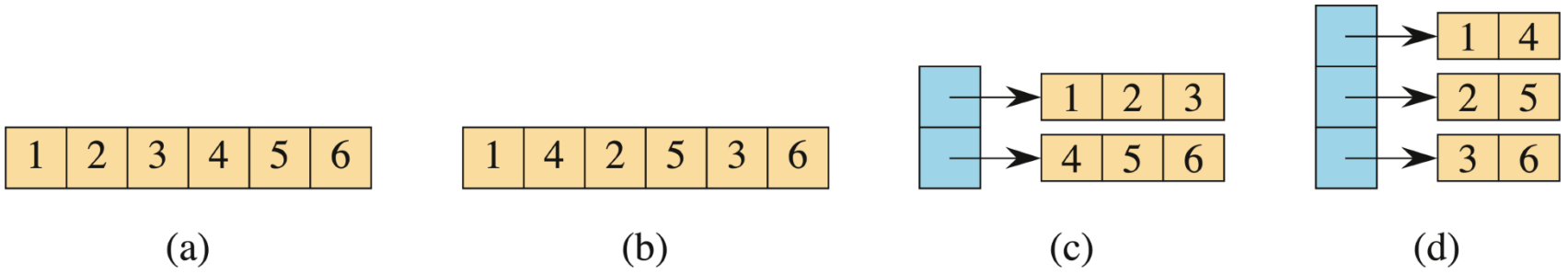
$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

  (a) In row-major order, in a single array.

  (b) In column-major order, in a single array.

  (c) In row-major order, with one array per row and a single array of pointers to the row arrays.

  (d) In column-major order, with one array per column and a single array of pointers to the column arrays.



# Simple array-based data structures: Matrices

- Occasionally, other schemes are used to store matrices.
- In the ***block representation***, the matrix is divided into blocks, and each block is stored contiguously.
- For example, a $4 \times 4$ matrix that is divided into $2 \times 2$ blocks, such as

$$\left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right)$$

might be stored in a single array in the order <1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 13, 14, 11, 12, 15, 16>

# Simple array-based data structures: Stacks

- Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified.

- ***Stack***: the element deleted is the one that was most recently inserted. Stacks use a ***last-in***, ***first-out***, or ***LIFO***, policy.

- Implement a stack of at most $n$ elements with array $S[1 : n]$.
  - Attribute $S.top$ indexes the most recently inserted element.
  - The stack contains elements $S[1 : S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.
  - Attribute $S.size = n$ gives the size of the array.

# Simple array-based data structures: Stacks

- Stack operations:
  - STACK-EMPTY: When $S.top = 0$, the stack contains no elements and is empty.
  - PUSH: The INSERT operation on a stack. Like pushing a plate on top of stack of plates. Pushing onto a full stack causes an overflow.
  - POP: The DELETE operation on a stack. Like popping off the plate o the top of a stack. Order in which plates are popped from the stack is reverse of order in which they were pushed. Popping an empty stack causes underflow.
- All three stack operations take $O(1)$ time.

```
STACK-EMPTY(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE

PUSH(S, x)
1  if S.top == S.size
2      error "overflow"
3  else S.top = S.top + 1
4      S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else S.top = S.top - 1
4      return S[S.top + 1]
```

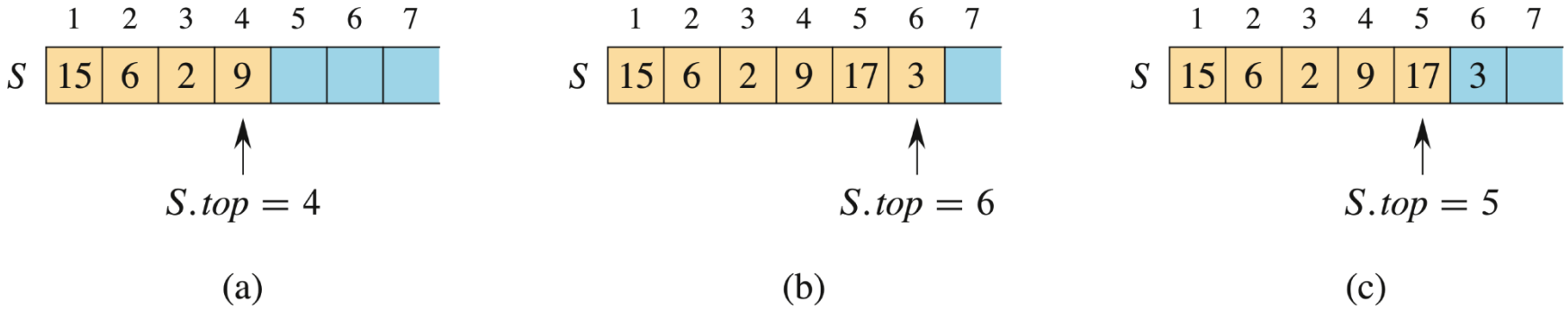
# Simple array-based data structures: Stacks

- An array implementation of a stack $S$.

  (a) Stack $S$ has 4 elements. The top element is 9.

  (b) Stack $S$ after the calls PUSH($S$, 17) and PUSH($S$, 3).

  (c) Stack $S$ after the call POP($S$) has returned the element 3, which is the one most recently pushed. Element 3 is no longer in the stack.



# Simple array-based data structures: Queues

- ***Queue***: the element deleted is the one that has been in the set for the longest time. Queues use a ***first-in***, ***first-out***, or ***FIFO***, policy. Inserting into a queue is ***enqueuing***, and deleting from a queue is ***dequeuing***.
- The FIFO property of a queue causes it to operate like a line of customers waiting for service. A queue has a ***head*** and a ***tail***.
  - When an element is enqueued, it goes to the tail of the queue, just as a newly arriving customer takes a place at the end of the line. 头进尾出
  - The element dequeued is the one at the head of the queue, like the customer at the head of the line who has waited the longest.

# Simple array-based data structures: Queues

- Implement a queue of at most $n - 1$ elements with array $Q[1 : n]$.
  - $Q.head$ indexes the head.
  - $Q.tail$ indexes the next location at which a new element will be inserted into the queue. Initially, 1st element will be enqueued to the "tail"
  - Elements reside in $Q.head$, $Q.head + 1$, …, $Q.tail - 1$, wrapping around so that $Q[1]$ follows $Q[n]$.
  - Initially, $Q.head = Q.tail = 1$.
  - $Q.head = Q.tail$ => queue is empty. Attempting to dequeue causes underflow.
  - in terms of circular queue
  - $Q.head = Q.tail + 1$ or both $Q.head = 1$ and $Q.tail = n$ => the queue is full. Attempting to enqueue causes overflow.
  - Attribute $Q.size$ gives the size $n$ of the array.

# Simple array-based data structures: Queues

- Queue operations:
  - ENQUEUE($Q, x$)
  - DEQUEUE($Q$)

  - These two procedures omit error checking for overflow and underflow. Written exercise 1.2 asks you to add these checks.

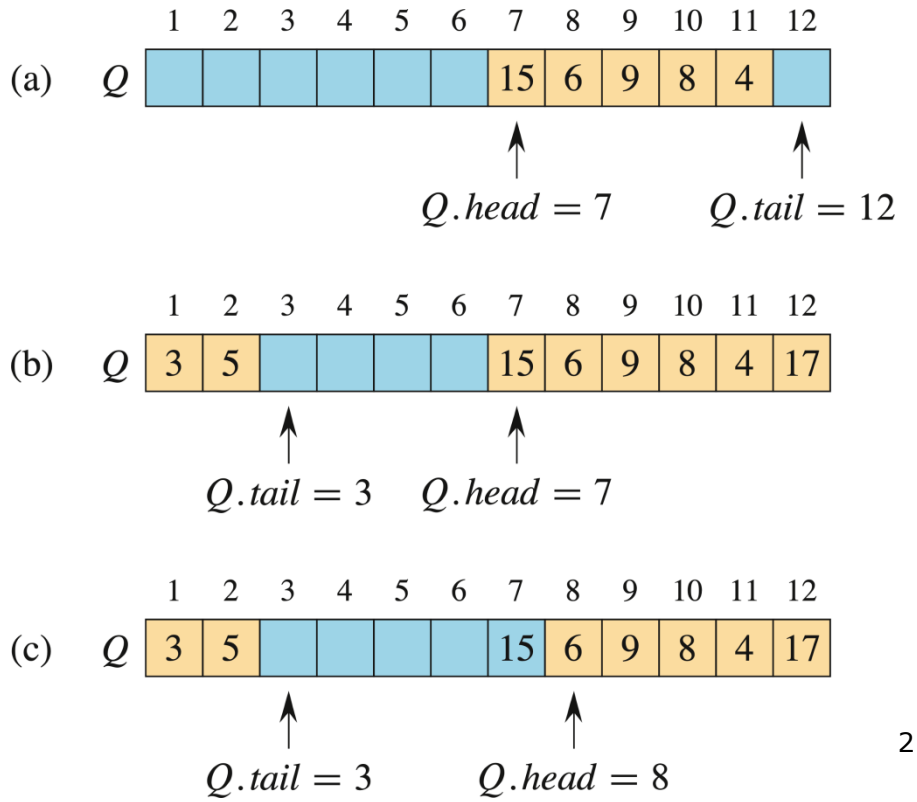
```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.size
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.size
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

- The two queue operations take $O(1)$ time.

# Simple array-based data structures: Queues

- A queue implemented using an array $Q[1 : 12]$.
  (a) Queue $Q$ has 5 elements, in locations $Q[7 : 11]$.
  (b) The configuration of the queue after the calls ENQUEUE($Q$, 17), ENQUEUE($Q$, 3), and ENQUEUE($Q$, 5).
  (c) The configuration of the queue after the call DEQUEUE($Q$) returns the key value 15 formerly at the head of the queue. The new head has key 6.



## Linked lists

- A ***linked list*** has
  - Objects arranged in a linear order.
  - Order is determined by a pointer in each object.
- In a ***doubly linked list***, each element $x$ has the following attributes:
  - $x.key$
  - $x.next$: the successor of $x$, NIL if $x$ has no successor so that it's the tail
  - $x.prev$: the predecessor of $x$, NIL if $x$ has no predecessor so that it is the head

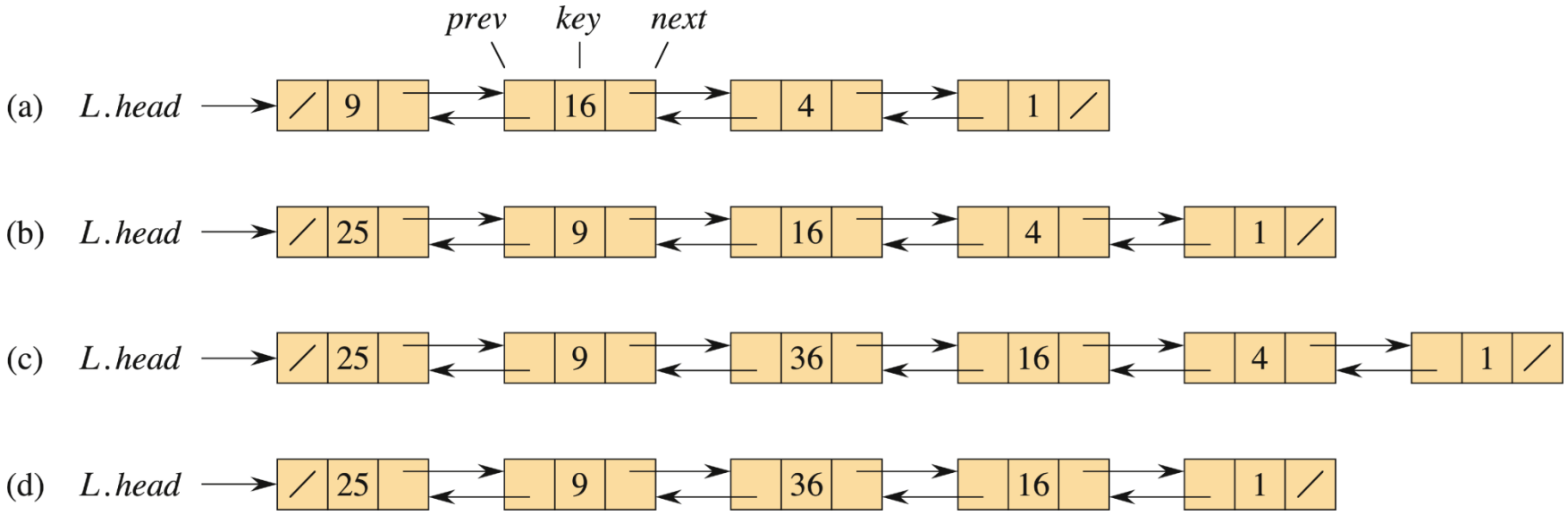
  $L.head$ points to the first element of the list, NIL if the list is empty.

## Linked lists

- Linked lists come in several types:
  - ***Singly linked***: each element has a *next* attribute but not a *prev* attribute.
  - ***Sorted***: the linear order of the list follows the linear order of keys stored in elements of the list.
  - ***Unsorted***: the elements can appear in any order.
  - ***Circular***: the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head.
- Here we assume that the lists we are working with are unsorted and doubly linked.

# Doubly linked lists

- Here is a doubly linked list $L$ whose element have keys 9, 16, 4, 1. Slashes indicate NIL.



## Searching a linked list

- LIST-SEARCH finds the first element with key $k$ in list $L$ by a linear search. It returns either a pointer $x$ to the element, or NIL if no element has key $k$.
- The worst-case time on a list with $n$ elements is $\Theta(n)$.

LIST-SEARCH$(L, k)$

```
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3        x = x.next
4  return x
```

- (a) of the figure shows after searching for key 16 returns a pointer to the second element in the list $L$. Searching for key 49 returns NIL.

# Inserting into a linked list

- There are two scenarios for inserting into a doubly linked list: inserting a new first element and inserting anywhere else.
- Given an element $x$ with the *key* element set, the procedure LIST-PREPEND adds $x$ to the front of the list $L$ in $O(1)$ time.

```
LIST-PREPEND(L, x)
1  x.next = L.head
2  x.prev = NIL
3  if L.head ≠ NIL
4        L.head.prev = x
5  L.head = x
```

- (b) of the figure shows the result of prepending 25.

## Inserting into a linked list

- To insert elsewhere, LIST-INSERT "splices" a new element $x$ into the list, immediately following $y$. Since the list object $L$ is not referenced, it is not supplied as a parameter.
- Like LIST-PREPEND, this procedure takes $O(1)$ time.

```
LIST-INSERT(x, y)
1  x.next = y.next
2  x.prev = y
3  if y.next ≠ NIL
4        y.next.prev = x
5  y.next = x
```

- (c) of the figure shows the result of inserting 36 after 9.

# Deleting from a linked list

- Given a pointer to $x$, LIST-DELETE removes $x$ from $L$ in $O(1)$ time.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

- (d) of the figure shows the result of deleting 4.
- To delete an element just given a key, first call LIST-SEARCH, then call LIST-DELETE. This makes the worst-case running time $\Theta(n)$.

## Linked list and array performance

- Insertion and deletion are faster on doubly linked lists than on arrays.

- In an array, insertion and deletion take $\Theta(n)$ time in the worst case, where all elements need to be shifted. In a doubly linked list, they take $O(1)$ time.

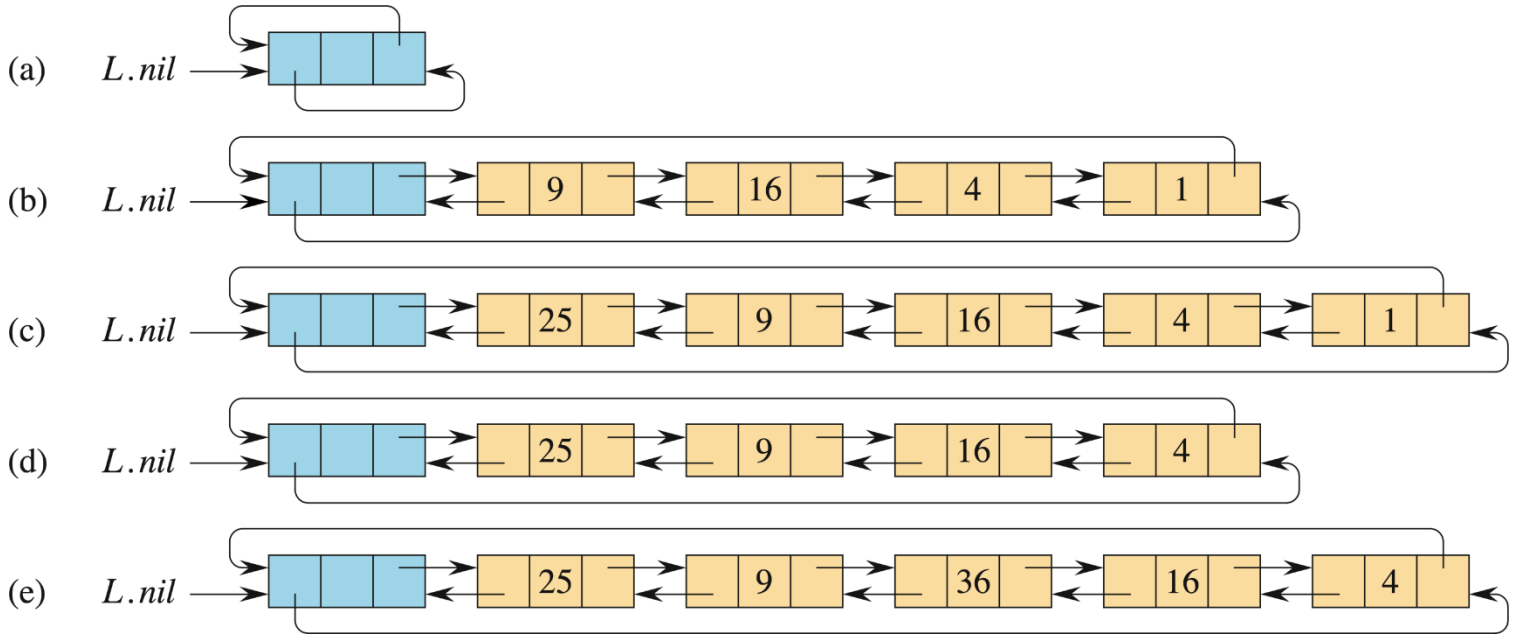
- Access by index, however, is faster in an array. Accessing the $k$th element in the linear order would take $\Theta(k)$ time in a linked list, but only $O(1)$ time in an array.

# Circular doubly linked list with a sentinel

- A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. In a ***circular doubly linked list with a sentinel***, replace NIL with a reference to the sentinel *L.nil*. *L.nil.next* points to the head of the list, and *L.nil.prev* points to the tail. *prev* attribute of the head and *next* attribute of the tail both point to *L.nil*. No attribute *L.head* needed.

  (a) of the figure shows an empty list: $L.nil.next = L.nil.prev = L.nil$.

  (b) of the figure shows a list whose elements have keys 9, 16, 4, 1.



# Searching a circular doubly linked list

- Searching has the same asymptotic running time as without a sentinel, but the constant factor can be better by removing one test per loop iteration.
- The trick is to guarantee that the key will be found, by putting it in the sentinel. Start at the head. If the key is really in the list, it will be found before getting back to the sentinel. If the key is not really in the list, it is found only in the sentinel.

```
LIST-SEARCH′(L, k)
1  L.nil.key = k          // store the key in the sentinel to guarantee it is in list
2  x = L.nil.next         // start at the head of the list
3  while x.key ≠ k
4       x = x.next
5  if x == L.nil          // found k in the sentinel
6       return NIL        // k was not really in the list
7  else return x          // found k in element x
```

# Inserting into a circular doubly linked list

- Now one procedure to insert fits all situations. To insert $x$ at the head of the list, set $y$ to $L.nil$. To insert $x$ at the tail, set $y$ to $L.nil.prev$.

LIST-INSERT$'(x, y)$

1 $x.next = y.next$
2 $x.prev = y$
3 $y.next.prev = x$
4 $y.next = x$

- (c) of the figure shows the result of inserting 25 after $L.nil$.
- (e) of the figure shows the result of inserting 36 after 9.

# Deleting from a circular doubly linked list

- How to delete an element $x$ no longer depends on where in the list $x$ is located. No need to supply $L$ as a parameter. Never delete the sentinel $L.nil$ unless you want to delete the entire list.

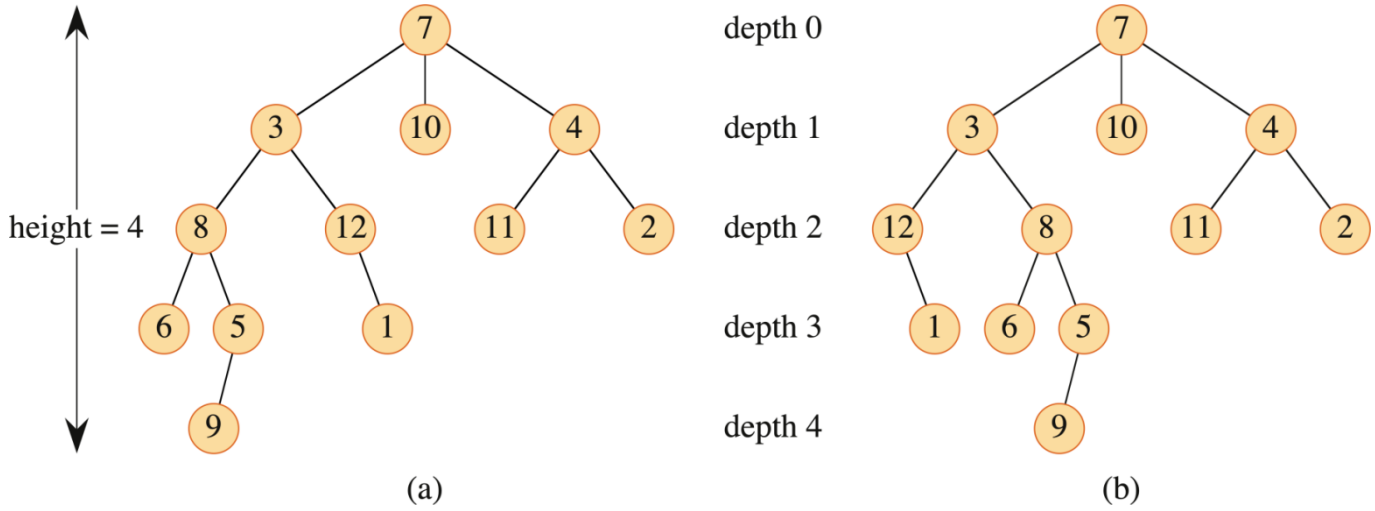
  LIST-DELETE′$(x)$

  1 $x.prev.next = x.next$
  2 $x.next.prev = x.prev$

- (d) of the figure shows the result of deleting 1 by calling LIST-DELETE′($L.nil.prev$).

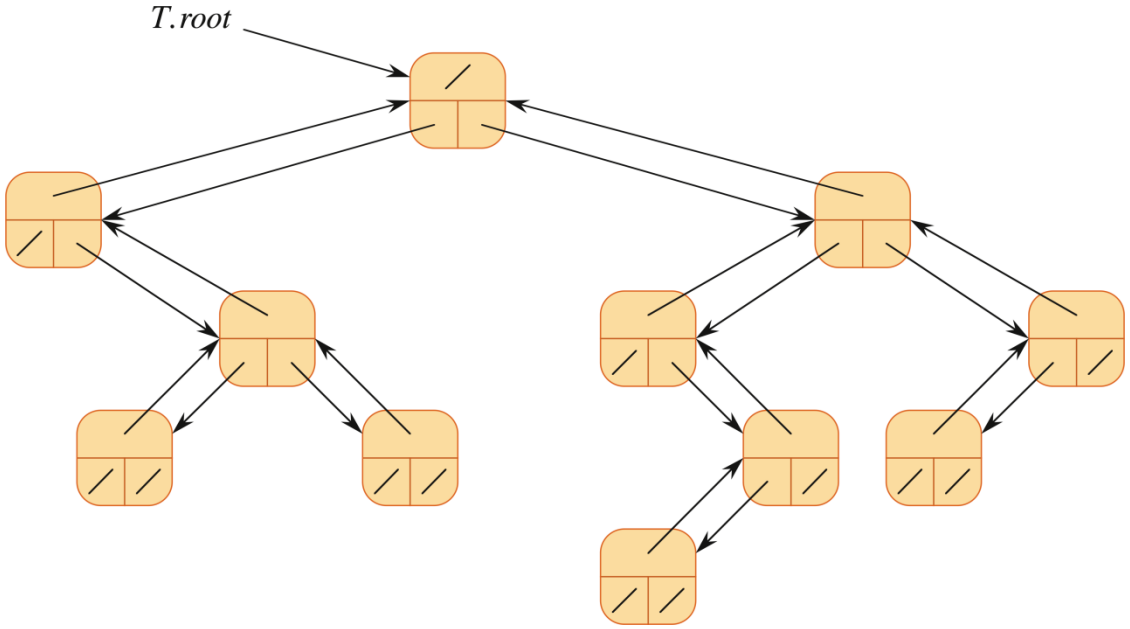
# Rooted trees

- A ***rooted tree*** is a free tree in which one of the vertices is distinguished from the others. We call the distinguished vertex the ***root*** of the tree. We often refer to a rooted tree as a ***node*** of the tree.
- The length of the simple path from the root to a node is the ***depth of a node*** in a tree.
- The ***height of a node*** in a tree is the number of edges on the longest simple downward path from the node to a leaf, and the ***height of a tree*** is the height of its root. The following rooted tree has the height of 4 and 5 levels.



# Representing rooted trees: Binary trees

- How to handle non-linear relationship? Represent a rooted tree by linked data structure. Each node of a tree is an object with a key attribute, like linked lists, and also has attributes that are pointers to other nodes.
- ***Binary trees***: Given each node in a binary tree has the following attributes: *p* (parent), *left*, *right*. If *x.p* = NIL, *x* is the root. The root of tree *T* is *T.root*. If *T.root* = NIL, then *T* is empty. If *x* has no left child, then *x.left* = NIL. Same for right child.
- Each node *x* has the attribute *x.p* (top), *x.left* (lower left), and *x.right* (lower right). *key* attributes are not shown.



## Reading and preparation for programming

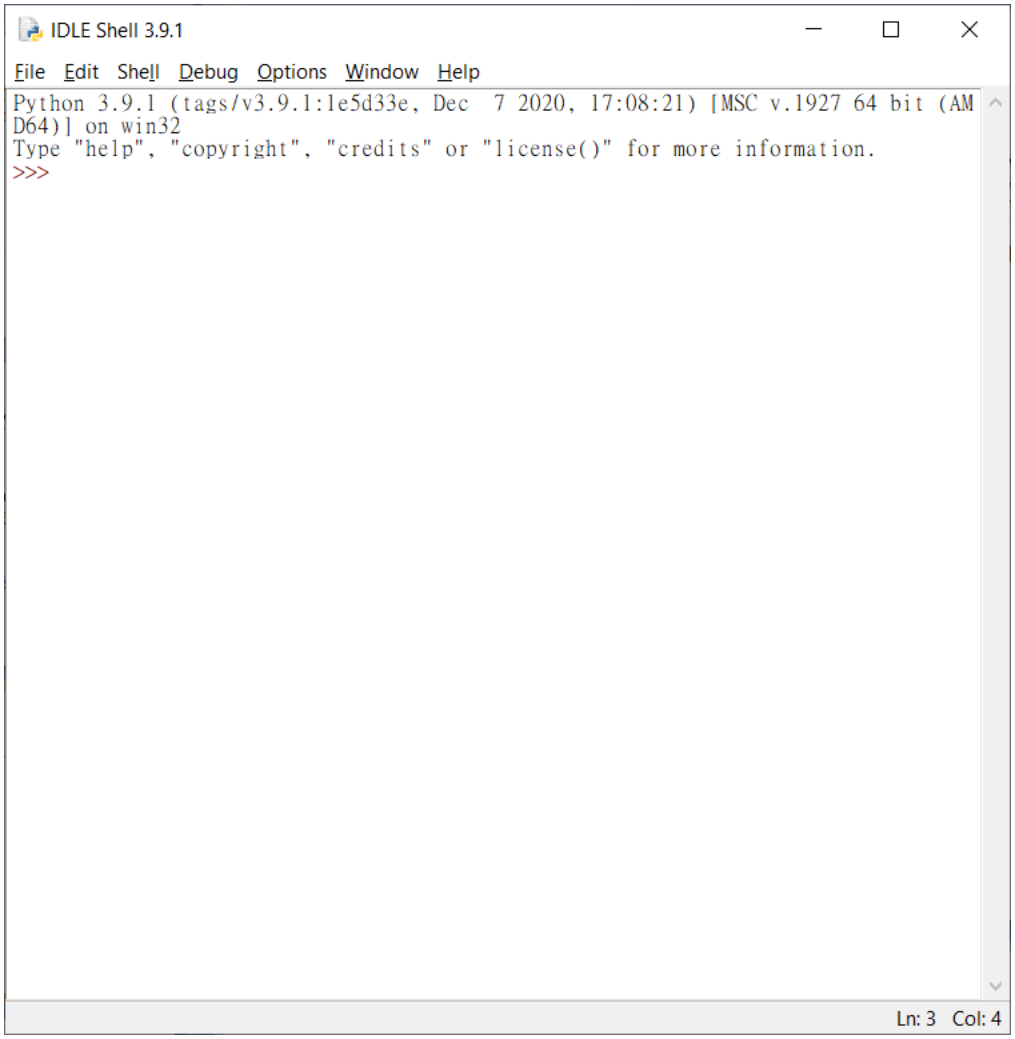
- Reading: Sections 1.1 ~ 1.2, 10.1 ~ 10.3.
- Preparation for Python programming
  - Install Standard Python with IDLE (https://www.python.org/downloads) and SciPy (https://scipy.org/install/).
  - Or install Python distribution Anaconda (https://www.anaconda.com).
  - Run `insertion_sort.py` to test your Python environment. This program can be found in `clrsPython.zip` on Moodle via WeMUST.

## Python notes

- In this lecture, there are four Python programs as follows, you can find them in "Chapter 10" folder after you unzip `clrsPython.zip`.
  - `dll.py` implements a ***doubly linked list***
  - `dll_sentinel.py` implements a ***circular doubly linked list with a sentinel***
  - `fifo_queue.py` implements a ***queue***
  - `lifo_stack.py` implements a ***stack***
- Note that `dll.py` and `dll_sentinel.py` require `key_object.py` which can be found in "Utility functions" folder.

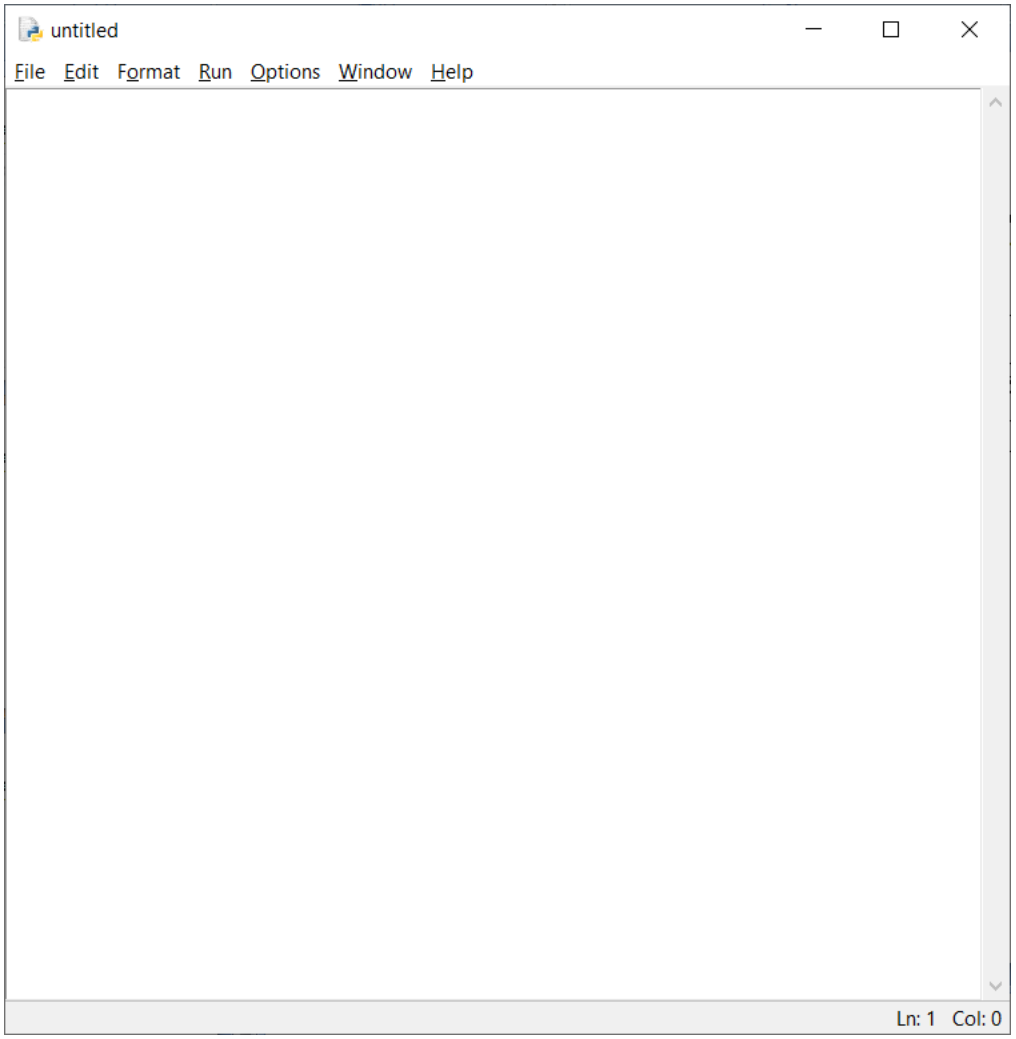
# Using Python 3.9 with IDLE

- Run **IDLE (Python 3.9 64-bit)**
- Then start writing a Python program: **File->New File**



# Using Python 3.9 with IDLE

- Then write the program on the editor.



# Using Python 3.9 with IDLE

- As an example, we implement the ***Euclid's algorithm*** for finding the greatest common divisor of two nonnegative, not-both-zero integers $m$ and $n$: gcd($m$, $n$) = gcd($n$, $m$ mod $n$), $n \neq 0$.

**ALGORITHM** *Euclid*($m$, $n$)

//Computes gcd($m$, $n$) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers $m$ and $n$
//Output: Greatest common divisor of $m$ and $n$
**while** $n \neq 0$ **do**
    $r \leftarrow m$ mod $n$
    $m \leftarrow n$
    $n \leftarrow r$
**return** $m$

**Python implementation:**

```
def Euclid(m, n):
    while n != 0:
        r = m % n
        m = n
        n = r
    return m
```

# Using Python 3.9 with IDLE

- Write the program
- Then select

**File->Save**

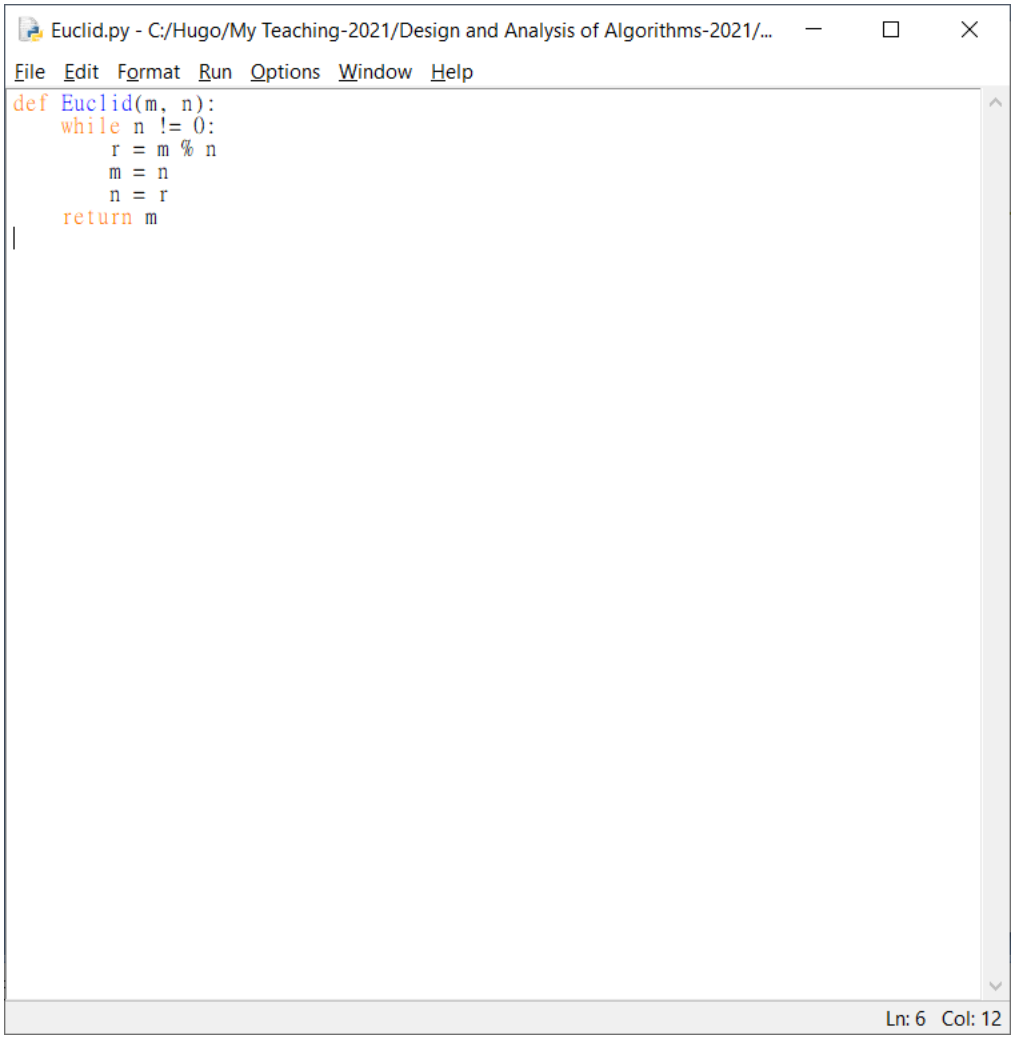
to save it as

a **Euclid.py** file

- Then run the

program:

**Run->Run Module**



# Using Python 3.9 with IDLE

- Input:

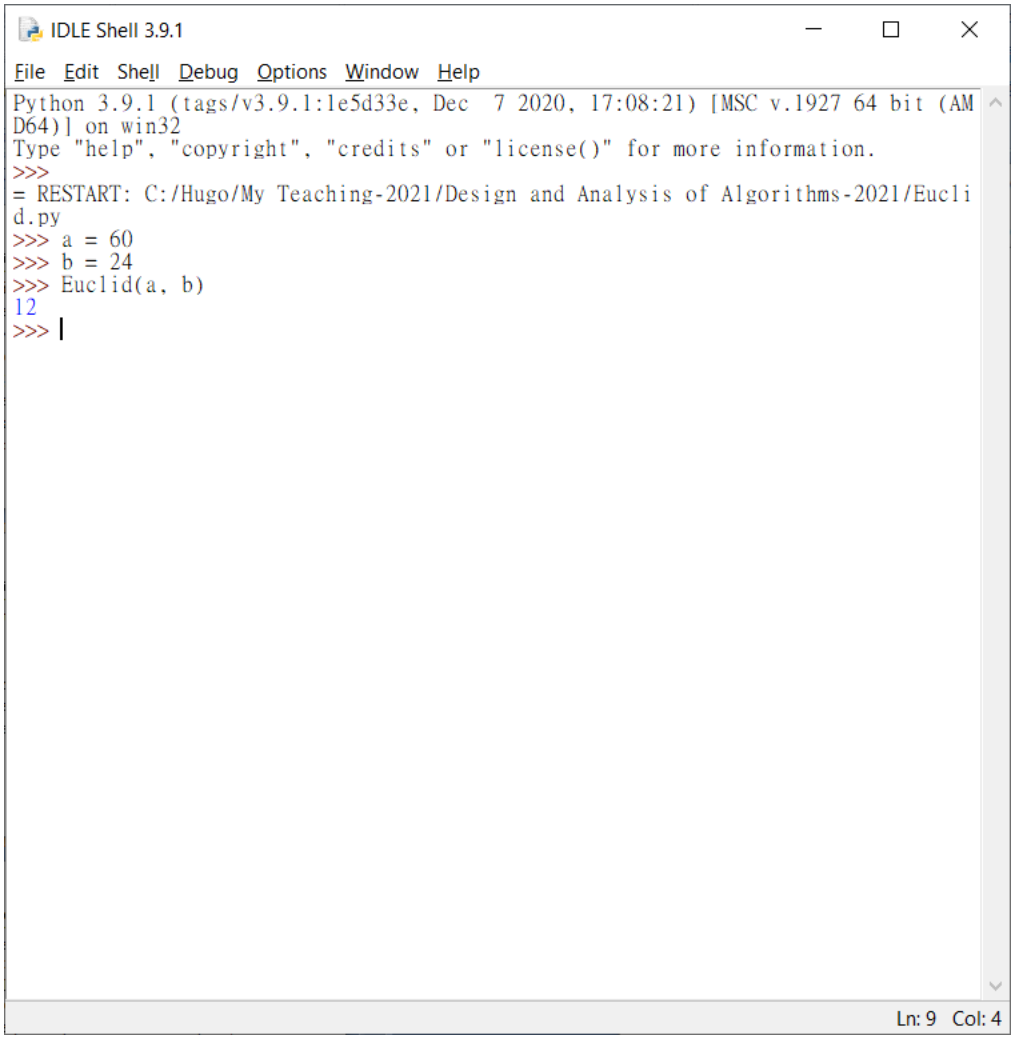
  a = 60

  b = 24

  Euclid(a, b)

- Then the answer is given:

12



# Written exercise 1.1

- Explain how to implement two stacks in one array $A[1 : n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is $n$. The PUSH and POP operations should run in $O(1)$ time.

Reason: Since S1.top has increment and S2.top has decrement when PUSH and POP operation begin and after the 1st operation, S1.top=1, S2.top=n. corresponding with the given array A[1:n], thus S1.top=0, S2.top=n+1 initially.

Solution: Call two stacks S1 and S2

Let S1.top=0, S2.top=n+1

```
Algorithm 1 PUSH(S,x)
if(S==S1){
   if(S1.top+1 == S2.top)
      error "overflow"
   else{S1.top = S1.top+1
   S1[S1.top] = x}
}
if(S==S2){
   if(S2.top-1==S1.top)
      error "overflow"
   else {S2.top = S2.top-1
   S2[S2.top] = x}
}
```


```
Algorithm 2 POP(S)
if(S==S1){
   if(S1.top==0)
      error "underflow"
   else{S1.top = S1.top-1
   return S1[S1.top+1]
if(S==S2){
   if(S2.top==n+1)
      error "underflow"
   else{S2.top = S2.top+1,
      return S2[S2.top-1]
}
```

# Written exercise 1.2

- Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

"empty"

Solution: Denote the queue by "Q"

Algorithm 1: DEQUEUE(Q)

```
if (Q.head == Q.tail)
    error "underflow"
else {
    X = Q[Q.tail]
    Q.tail = Q.tail - 1
    return X
}
```

Algorithm 2: ENQUEUE(Q, X)

```
if (Q.tail == Q.length)
    error "overflow"
else {
    Q.tail = Q.tail + 1
    Q[Q.tail] = X
}
```

# Written exercise 1.3

- Explain why the dynamic-set operation INSERT on a singly linked list can be implemented in $O(1)$ time, but the worst-case time for DELETE is $\Theta(n)$.

Solution:

① In terms of INSERT:

Since an element can be inserted in front of the old head, pointing to it to be the new head, which costs only in constant time, which means that it can be implemented in O(1).

② In terms of DELETE:

As for its worst case, assume that the "target element" to be deleted is the last one in the linked list, and it needs "search" operation first, which costs linear time O(n), and finally finds the one in the last position, because there is no way getting a pointer to the "required index" without starting at the head.

# Written exercise 1.4

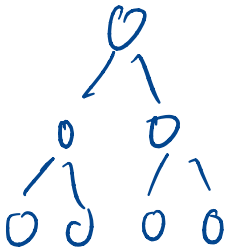
- Write an $O(n)$-time recursive procedure that, given an $n$-node binary tree, prints out the key of each node in the tree.

Solution: Denote the n-node binary tree by "T"

```
PRINT (T.root)
 if (T.root == NIL)
     return
 else {
   print T.root.key
   PRINT (T.root.left)
   PRINT (T.root.right)
}
```

# Written exercise 1.5

- Write an $O(n)$-time nonrecursive procedure that, given an $n$-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.



Solution:

```
PRINT(T.root)
  if(T.root == NIL)
     return
  else{
    print T.root.key
    x = T.root.left_child
    while(x != NIL){
      PRINT(x)
      x = T.root.right_child
}}
```