

**Computer science
Higher level
Paper 2****Mock examination — Set 2**2 hours

Instructions to candidates

- Do not open this examination paper until instructed to do so.
- Answer all of the questions from **one** of the options.
- The maximum mark for this examination paper is **[80 marks]**.

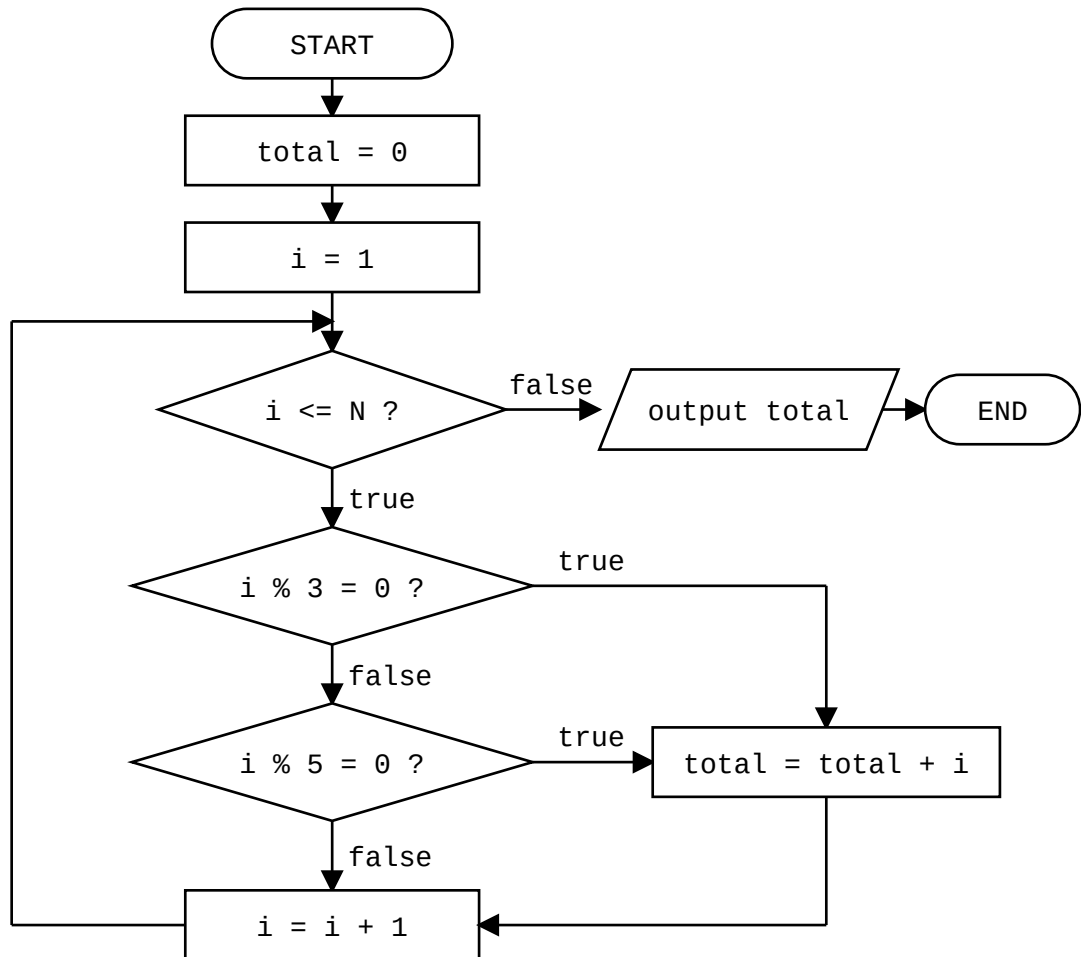
Option	Questions
Option A — Java	1 – 6
Option B — Python	7 – 12

Option A — Java

Answer **all** questions. All code that forms part of your responses must be written in Java.

1. **Figure 1** shows a flowchart for an algorithm. A whole number, N , is input when the algorithm starts. In the flowchart, $\%$ gives the remainder after a division. For example, $7 \% 3$ is 1.

Figure 1: Flowchart for a summing algorithm



- (a) Copy and complete the trace table for this flowchart, where $N = 7$.

[3]

i	i <= N ?	i % 3 = 0 ?	i % 5 = 0 ?	total	output
1				0	

Big O notation is used to describe the efficiency of an algorithm.

- (b) Using Big O notation, describe how the time taken by the algorithm in **Figure 1** grows as N increases.

[2]

(Option A continues on the following page)

(Option A, question 1 continued)

Computational thinking can be used to design an algorithm before any code is written.

A bakery wants a program that calculates how much of each ingredient it needs to order every week. The programmer applies computational thinking in the following two steps:

- Step 1: The programmer records only the mass of each ingredient needed to bake one loaf of each type, and ignores details such as the shape of each loaf and the design of the bags in which the loaves are sold.
- Step 2: The programmer notices that the mass of every ingredient is calculated in the same way: the number of loaves to be baked is multiplied by the mass of that ingredient needed for one loaf.

- (c) Identify the computational thinking concept that has been applied in each of these steps.

[2]

Table 1 shows a list of values.

Table 1: A list of values

Index	0	1	2	3	4
Value	29	14	37	8	21

- (d) Describe the steps that a selection sort would follow to sort this list into ascending order.

[3]

(Option A continues on the following page)

(Option A, question 1 continued)

In a school canteen, clean trays are placed in a dispenser, and the last tray put in is the first tray taken out. A program models the dispenser as a stack, `trays`, in which each tray is represented by its number.

Table 2 shows the operations available on `trays`.

Table 2: Operations available on the stack, `trays`

Operation	Description
<code>push(x)</code>	Adds tray <code>x</code> to the top of the stack.
<code>pop()</code>	Removes the tray at the top of the stack and returns it.
<code>peek()</code>	Returns the tray at the top of the stack without removing it from the stack.
<code>isEmpty()</code>	Returns true if the stack is empty, else it returns false.
<code>size()</code>	Returns the number of trays in the stack.

At the start of lunch, `trays` holds three trays: tray 11 at the bottom, then tray 14, then tray 18 at the top. The following operations are then carried out, in order:

```
pop()
push(25)
push(30)
size()
pop()
peek()
push(41)
isEmpty()
```

- (e) Sketch `trays` after these operations have been carried out.

Your sketch should show which tray is at the top of the stack.

[2]

At the end of lunch, `trays` holds the numbers of the trays that are left. A second, empty stack, with the same operations, is also available.

- (f) Using only the operations shown in **Table 2**, describe the steps that will output the largest tray number in `trays`. At the end, `trays` must contain the same trays, in the same order, as at the start.

Assume that `trays` is not empty.

[3]

(Option A continues on the following page)

(Option A continued)

2. A recycling centre collects four materials: glass, paper, plastic and metal. At the end of each day, the mass of each material collected that day is recorded in kilograms, as a whole number. The centre is open every day.

A program stores the masses for one week in two global variables, `materials` and `mass`. They are declared in the program's main class, outside any method:

```
static String[] materials = {"glass", "paper", "plastic", "metal"};
static int[][] mass = new int[4][7];
```

`mass[m][d]` holds the mass of material `m` collected on day `d`, where material 0 is glass and day 0 is Monday. For example, `mass[1][4] = 25` shows that 25 kg of paper was collected on day 4 (Friday).

Table 3 shows the contents of `mass` for one week.

Table 3: Mass, in kg, of each material collected on each day

Material	m	d = 0	d = 1	d = 2	d = 3	d = 4	d = 5	d = 6
glass	0	20	15	25	10	30	20	20
paper	1	35	40	20	30	25	45	15
plastic	2	10	25	30	15	20	35	40
metal	3	5	10	15	25	10	15	30

The method `showTotals()` is intended to output the total mass of each material collected during the week:

```
public static void showTotals()
{
    int total = 0;
    for (int m = 0; m < 4; m++)
    {
        for (int d = 0; d < 7; d++)
        {
            total = total + mass[m][d];
        }
        System.out.println (materials[m] + ": " + total);
    }
}
```

The variable `total` is a local variable of `showTotals()`.

- (a) Outline why `total` should be a local variable, rather than a global variable. [2]
- (b) State the **first two** lines of output when `showTotals()` runs with the data in **Table 3**. [2]

(Option A continues on the following page)

(Option A, question 2 continued)

The output of `showTotals()` is not correct. To find the error, a programmer adds the following statement as the first statement inside the outer loop, before the inner loop:

```
System.out.println ("m = " + m + ", total = " + total);
```

- (c) Describe how the output of this statement would help the programmer to find **and** correct the error. [3]

At the end of each day, the method `saveDay()` is passed the number of that day. It appends **one** line to the end of the existing text file `week.txt`. The line holds the day number, followed by the masses of glass, paper, plastic and metal collected on that day, all separated by commas, and ends with a new line. For example, using the data in **Table 3**, `saveDay(5)` appends this line:

5, 20, 45, 35, 15

The file is currently closed.

- (d) Construct the method `saveDay()`. [3]

The method `dailyBest()` uses the global variables `materials` and `mass`. For each day, it outputs the day number and the name of the material with the greatest mass collected on that day. For example, using the data in **Table 3**, the first line of output is:

Day 0: paper

- (e) Construct the method `dailyBest()`.

Assume that, on each day, only one material has the greatest mass.

Library methods that sort values or find a maximum, such as `Arrays.sort()`, `Collections.max()` and `Math.max()`, must not be used. [5]

(Option A continues on the following page)

(Option A continued)

3. A parcel delivery company uses object-oriented programming (OOP) to track the parcels it delivers. Each time a parcel is scanned, its status may change, for example from "Collected" to "Out for delivery". Every customer who is tracking the parcel is then informed of its new status. The programmers have used the observer design pattern to do this.

The interface Notifier is shown:

```
public interface Notifier{

    public void update(String code, String status);
}
```

The class EmailNotifier implements Notifier. It is shown:

```
public class EmailNotifier implements Notifier{

    private String email;

    public EmailNotifier(String email)
    {
        this.email = email;
    }

    public void update(String code, String status)
    {
        System.out.println("Email to " + email + ": parcel " + code
                           + " is " + status);
    }
}
```

Each change of status of a parcel is stored as a TrackingEvent object. The TrackingEvent class is shown:

```
public class TrackingEvent{

    private String status;    // e.g. "Out for delivery"
    private String time;      // time of the scan, e.g. "09:40"

    public TrackingEvent(String status, String time)
    {
        this.status = status;
        this.time = time;
    }

    public String getStatus()
    {
        return status;
    }
}
```

(Option A continues on the following page)

(Option A, question 3 continued)

The Parcel class is partially shown:

```
public class Parcel{

    private String code;           // tracking code, e.g. "P-48213"
    private TrackingEvent[] history; // status changes, oldest first
    private int numEvents;         // number of events in history
    private Notifier[] notifiers;   // notifiers tracking this parcel
    private int numNotifiers;       // number of notifiers

    public Parcel(String code)
    {
        this.code = code;
        this.history = new TrackingEvent[50];
        this.numEvents = 0;
        this.notifiers = new Notifier[10];
        this.numNotifiers = 0;
    }

    public void addNotifier(Notifier n)
    {
        notifiers[numNotifiers] = n;
        numNotifiers = numNotifiers + 1;
    }

    public void setStatus(String status, String time)
    {
        // code missing
    }
}
```

The method setStatus() is called at each scan. If the status has changed, it adds a new TrackingEvent object to history and calls update() for every notifier in notifiers.

When a customer chooses to track a parcel, the driver class passes that customer's Notifier object to addNotifier(). A customer's Notifier object is created once, by the driver class, and is used for every parcel that the customer tracks.

- (a) Outline how the use of the interface Notifier allows a new type of notifier, such as a notification sent to the company's app, to be added without changing the Parcel class. [2]

Customers can also choose to be informed by text message. To keep costs low, a text message is sent only when the status changes to "Out for delivery" or to "Delivered".

The class SmsNotifier implements Notifier. Its constructor is passed the customer's mobile phone number, which is stored in a private attribute. Its update() method outputs the text message, as in this example:

SMS to 07700900123: P-48213 Delivered

- (b) Construct the class SmsNotifier. [4]

(Option A continues on the following page)

(Option A, question 3 continued)

(c) Explain how the observer design pattern has been applied in this system. [3]

(d) Construct a UML class diagram that shows the relationships between the classes `Parcel`, `TrackingEvent`, `EmailNotifier` and `SmsNotifier` and the interface `Notifier`.

Attributes and methods do not need to be shown. [3]

Thirty days after a parcel has been delivered, the company's system deletes its `Parcel` object by removing every reference to it.

(e) With reference to the relationships between the classes, describe what happens to the parcel's `TrackingEvent` objects and to its `Notifier` objects when its `Parcel` object is deleted. [3]

When a `Parcel` object is created, `history` is empty. The method `setStatus(String status, String time)` works as follows:

- If `history` is not empty and `status` is the same as the status of the most recent `TrackingEvent` in `history`, the scan is a repeat and nothing is done.
- Otherwise, a new `TrackingEvent` object is created for `status` and `time` and added to the end of `history`. Then `update()` is called for every notifier in `notifiers`, passing it the parcel's code and the new status.

(f) Construct the method `setStatus()`.

Assume that `history` and `notifiers` never become full. [5]

(Option A continues on the following page)

(Option A continued)

4. A hospital pharmacy uses a computer system to manage its stock of medicines. The system stores sets of medicine names. **Table 4** shows the operations of the set abstract data type (ADT) that the programmers use.

Table 4: Operations of the set ADT used by the pharmacy system

Operation	Description
<code>s.add(x)</code>	Adds <code>x</code> to the set <code>s</code> , unless <code>x</code> is already in <code>s</code>
<code>s.remove(x)</code>	Removes <code>x</code> from the set <code>s</code> , if <code>x</code> is in <code>s</code>
<code>s.contains(x)</code>	Returns <code>true</code> if <code>x</code> is in the set <code>s</code> , and <code>false</code> otherwise
<code>s.size()</code>	Returns the number of elements in the set <code>s</code>
<code>for (String x : s)</code>	Visits each element <code>x</code> of the set <code>s</code> in turn, in no particular order

- (a) With reference to **Table 4**, outline **two** properties of an abstract data type (ADT). [2]

The system also stores a three-digit code for each medicine in a hash table with 11 slots, indexed 0 to 10. The hash function is `code % 11` (the remainder when the code is divided by 11). Collisions are resolved by linear probing: the following slots are checked in turn, continuing from slot 0 after slot 10, until an empty slot is found.

Table 5 shows the hash table after five codes have been inserted.

Table 5: The hash table after five codes have been inserted

Index	0	1	2	3	4	5	6	7	8	9	10
Code			189			357	412			603	846

The codes 731 and 384 are now inserted, in this order.

- (b) State the index of the slot in which each of these two codes is stored. [2]

The names of the different medicines on a patient's prescription are stored in the `HashSet<String> prescription`. The names of all the medicines that the pharmacy has in stock are stored in the `HashSet<String> stock`.

The method `canDispense(HashSet<String> prescription, HashSet<String> stock)` returns `true` if every medicine in `prescription` is also in `stock` (that is, if `prescription` is a subset of `stock`), and `false` otherwise. If `prescription` is empty, the method returns `true`. Neither set is changed.

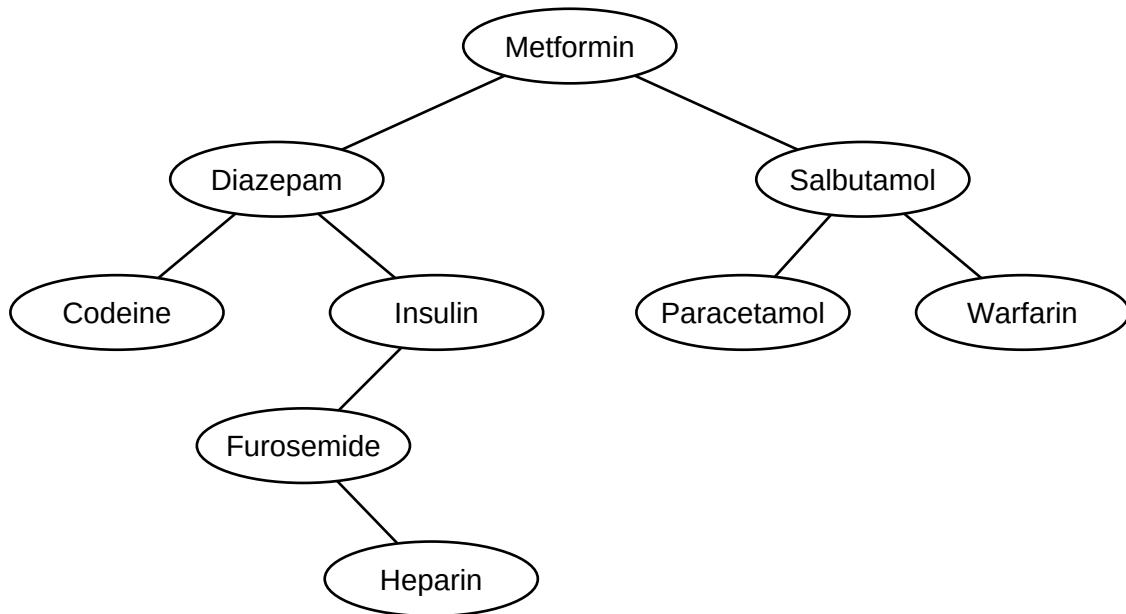
- (c) Using only the operations shown in **Table 4**, construct the method `canDispense()`. [3]

(Option A continues on the following page)

(Option A continued)

5. The pharmacy also stores the names of the medicines it supplies in a binary search tree (BST), in alphabetical order. **Figure 2** shows the BST after the first nine names have been inserted.

Figure 2: The binary search tree after the first nine names have been inserted



- (a) Sketch the binary search tree after the names Omeprazole and Ibuprofen have been inserted, in this order, into the tree in **Figure 2**. [2]

In a pre-order traversal, the node is visited first, then its left subtree is traversed, then its right subtree is traversed.

- (b) State the order in which the names in **Figure 2** are visited in a pre-order traversal. [2]
- (c) Without writing code, describe the steps needed to find and delete Diazepam from the binary search tree in **Figure 2**, so that it remains a binary search tree. [3]

After each half-life, the amount of a medicine in a patient's body is halved. The recursive method `remaining(double dose, int n)` returns the amount (in mg) that is left from a dose of dose mg after n half-lives:

```
public double remaining(double dose, int n)
{
    if (n == 0)
        return dose;
    else
        return remaining(dose / 2, n - 1);
}
```

- (d) Trace the call `remaining(400, 3)` by stating the sequence of calls that are made and the value returned by each call. [2]

(Option A continues on the following page)

(Option A, question 5 continued)

The number of tablets of a medicine dispensed on each day is stored in the array `tablets`, for example:

```
int[] tablets = {120, 95, 140, 80, 110};
```

The recursive method `busiest(int[] tablets, int n)` returns the largest number of tablets dispensed on any one of the first n days, that is, the largest of `tablets[0]` to `tablets[n - 1]`. For example, `busiest(tablets, 2)` returns 120 and `busiest(tablets, 5)` returns 140.

(e) Construct the recursive method `busiest()`.

Assume that $1 \leq n \leq \text{tablets.length}$.

The `Math.max()` method must not be used.

[4]

(Option A continues on the following page)

(Option A continued)

6. A music streaming service stores each playlist as a doubly linked list. Each node holds the title of a track and its length in seconds. The pointer next of each node points to the following node, and the pointer prev points to the previous node.

The linked list makes use of the Node class, which is shown:

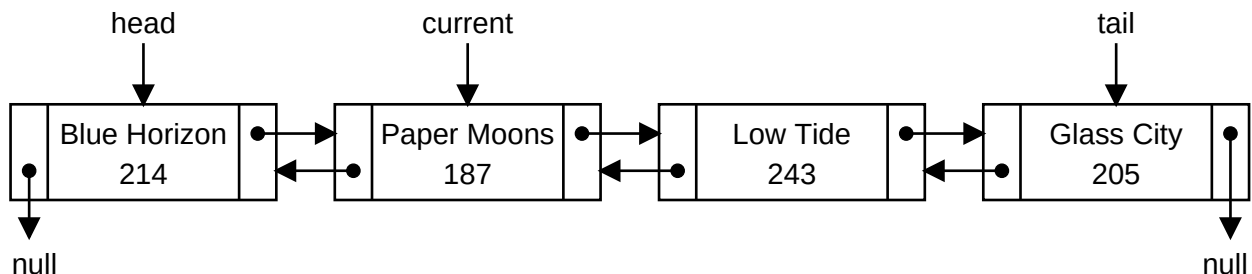
```
class Node{

    public String title;        // e.g. "Paper Moons"
    public int seconds;         // length of the track
    public Node prev;           // previous node in the playlist
    public Node next;           // following node in the playlist

    public Node(String t, int s)
    {
        this.title = t;
        this.seconds = s;
        this.prev = null;
        this.next = null;
    }
}
```

Figure 3 shows a playlist. The Node objects head and tail point to the first and the last nodes in the playlist, and the Node object current points to the node of the track that is playing. head, tail and current are attributes of the class Playlist, which contains the methods playNext() and removeTrack().

Figure 3: A playlist stored in a doubly linked list



The method playNext(Node newTrack) inserts the node newTrack into the playlist immediately after current, so that it is the next track to be played. If current is the last node in the playlist, newTrack becomes the last node.

- (a) Without writing code, describe the steps in the method playNext().

Assume that current is not null.

[5]

(Option A continues on the following page)

(Option A, question 6 continued)

The method `removeTrack(String title)` removes the first node in the playlist whose title is equal to `title`. Both the `next` and the `prev` pointers must be updated so that the playlist remains a correct doubly linked list:

- If the node removed is the first node, `head` must be updated.
- If the node removed is the last node, `tail` must be updated.
- If no track in the playlist has this title, the method outputs a suitable message and the playlist is not changed.

(b) Construct the method `removeTrack()`.

Assume that the playlist contains at least two tracks and that the track to be removed is not the track that is playing.

[5]

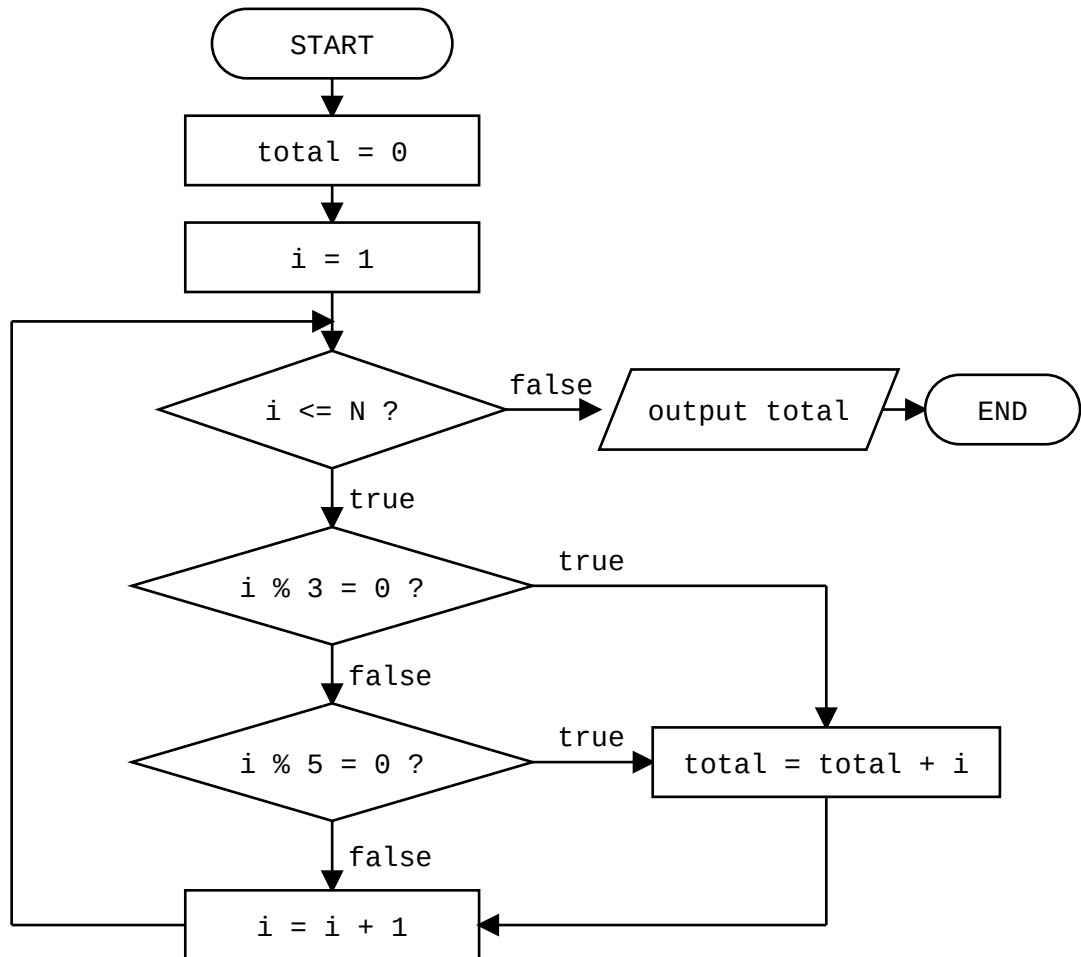
End of Option A

Option B — Python

Answer **all** questions. All code that forms part of your responses must be written in Python.

7. **Figure 4** shows a flowchart for an algorithm. A whole number, N , is input when the algorithm starts. In the flowchart, $\%$ gives the remainder after a division. For example, $7 \% 3$ is 1.

Figure 4: Flowchart for a summing algorithm



- (a) Copy and complete the trace table for this flowchart, where $N = 7$.

[3]

i	$i \leq N$?	$i \% 3 = 0$?	$i \% 5 = 0$?	total	output
1				0	

Big O notation is used to describe the efficiency of an algorithm.

- (b) Using Big O notation, describe how the time taken by the algorithm in **Figure 4** grows as N increases.

[2]

(Option B continues on the following page)

(Option B, question 7 continued)

Computational thinking can be used to design an algorithm before any code is written.

A bakery wants a program that calculates how much of each ingredient it needs to order every week. The programmer applies computational thinking in the following two steps:

- Step 1: The programmer records only the mass of each ingredient needed to bake one loaf of each type, and ignores details such as the shape of each loaf and the design of the bags in which the loaves are sold.
- Step 2: The programmer notices that the mass of every ingredient is calculated in the same way: the number of loaves to be baked is multiplied by the mass of that ingredient needed for one loaf.

- (c) Identify the computational thinking concept that has been applied in each of these steps.

[2]

Table 6 shows a list of values.

Table 6: A list of values

Index	0	1	2	3	4
Value	29	14	37	8	21

- (d) Describe the steps that a selection sort would follow to sort this list into ascending order.

[3]

(Option B continues on the following page)

(Option B, question 7 continued)

In a school canteen, clean trays are placed in a dispenser, and the last tray put in is the first tray taken out. A program models the dispenser as a stack, `trays`, in which each tray is represented by its number.

Table 7 shows the operations available on `trays`.

Table 7: Operations available on the stack, `trays`

Operation	Description
<code>push(x)</code>	Adds tray <code>x</code> to the top of the stack.
<code>pop()</code>	Removes the tray at the top of the stack and returns it.
<code>peek()</code>	Returns the tray at the top of the stack without removing it from the stack.
<code>isEmpty()</code>	Returns true if the stack is empty, else it returns false.
<code>size()</code>	Returns the number of trays in the stack.

At the start of lunch, `trays` holds three trays: tray 11 at the bottom, then tray 14, then tray 18 at the top. The following operations are then carried out, in order:

```
pop()
push(25)
push(30)
size()
pop()
peek()
push(41)
isEmpty()
```

- (e) Sketch `trays` after these operations have been carried out.

Your sketch should show which tray is at the top of the stack.

[2]

At the end of lunch, `trays` holds the numbers of the trays that are left. A second, empty stack, with the same operations, is also available.

- (f) Using only the operations shown in **Table 7**, describe the steps that will output the largest tray number in `trays`. At the end, `trays` must contain the same trays, in the same order, as at the start.

Assume that `trays` is not empty.

[3]

(Option B continues on the following page)

(Option B continued)

8. A recycling centre collects four materials: glass, paper, plastic and metal. At the end of each day, the mass of each material collected that day is recorded in kilograms, as a whole number. The centre is open every day.

A program stores the masses for one week in two global variables, `materials` and `mass`. They are declared outside any function:

```
materials = ["glass", "paper", "plastic", "metal"]
mass = [[0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0]]
```

`mass[m][d]` holds the mass of material `m` collected on day `d`, where material 0 is glass and day 0 is Monday. For example, `mass[1][4] = 25` shows that 25 kg of paper was collected on day 4 (Friday).

Table 8 shows the contents of `mass` for one week.

Table 8: Mass, in kg, of each material collected on each day

Material	m	d = 0	d = 1	d = 2	d = 3	d = 4	d = 5	d = 6
glass	0	20	15	25	10	30	20	20
paper	1	35	40	20	30	25	45	15
plastic	2	10	25	30	15	20	35	40
metal	3	5	10	15	25	10	15	30

The function `show_totals()` is intended to output the total mass of each material collected during the week:

```
def show_totals():
    total = 0
    for m in range(0, 4):
        for d in range(0, 7):
            total = total + mass[m][d]
            print(materials[m] + ":", total)
```

The variable `total` is a local variable of `show_totals()`.

- (a) Outline why `total` should be a local variable, rather than a global variable. [2]
- (b) State the **first two** lines of output when `show_totals()` runs with the data in **Table 8**. [2]

The output of `show_totals()` is not correct. To find the error, a programmer adds the following statement as the first statement inside the outer loop, before the inner loop:

```
print("m =", m, "total =", total)
```

- (c) Describe how the output of this statement would help the programmer to find **and** correct the error. [3]

(Option B continues on the following page)

(Option B, question 8 continued)

At the end of each day, the function `save_day()` is passed the number of that day. It appends **one** line to the end of the existing text file `week.txt`. The line holds the day number, followed by the masses of glass, paper, plastic and metal collected on that day, all separated by commas, and ends with a new line. For example, using the data in **Table 8**, `save_day(5)` appends this line:

5, 20, 45, 35, 15

The file is currently closed.

(d) Construct the function `save_day()`. [3]

The function `daily_best()` uses the global variables `materials` and `mass`. For each day, it outputs the day number and the name of the material with the greatest mass collected on that day. For example, using the data in **Table 8**, the first line of output is:

Day 0: paper

(e) Construct the function `daily_best()`.

Assume that, on each day, only one material has the greatest mass.

Functions and methods that sort values or find a maximum, such as `max()`, `sorted()` and `sort()`, must not be used. [5]

(Option B continues on the following page)

(Option B continued)

9. A parcel delivery company uses object-oriented programming (OOP) to track the parcels it delivers. Each time a parcel is scanned, its status may change, for example from "Collected" to "Out for delivery". Every customer who is tracking the parcel is then informed of its new status. The programmers have used the observer design pattern to do this.

The abstract class `Notifier` is shown. The `ABC` class and the `@abstractmethod` decorator from the `abc` module are used to declare `Notifier` and its method `update()` as abstract.

```
from abc import ABC, abstractmethod
```

```
class Notifier(ABC):

    @abstractmethod
    def update(self, code, status):
        pass
```

The class `EmailNotifier` inherits from `Notifier`. It is shown:

```
class EmailNotifier(Notifier):

    def __init__(self, email):
        self._email = email

    def update(self, code, status):
        print ("Email to " + self._email + ": parcel " + code
              + " is " + status)
```

The underscore character in `self._email` identifies the attribute as private.

Each change of status of a parcel is stored as a `TrackingEvent` object. The `TrackingEvent` class is shown:

```
class TrackingEvent():

    def __init__(self, status, time):
        self._status = status      # e.g. "Out for delivery"
        self._time = time          # time of the scan, e.g. "09:40"

    def get_status(self):
        return self._status
```

(Option B continues on the following page)

(Option B, question 9 continued)

The Parcel class is partially shown:

```
class Parcel():

    def __init__(self, code):
        self._code = code           # tracking code, e.g. "P-48213"
        self._history = []         # status changes, oldest first
        self._notifiers = []       # notifiers tracking this parcel

    def add_notifier(self, n):
        self._notifiers.append(n)

    def set_status(self, status, time):
        # code missing
```

The method `set_status()` is called at each scan. If the status has changed, it adds a new `TrackingEvent` object to `self._history` and calls `update()` for every notifier in `self._notifiers`.

When a customer chooses to track a parcel, the driver code passes that customer's `Notifier` object to `add_notifier()`. A customer's `Notifier` object is created once, by the driver code, and is used for every parcel that the customer tracks.

- (a) Outline how the use of the abstract class `Notifier` allows a new type of notifier, such as a notification sent to the company's app, to be added without changing the `Parcel` class. [2]

Customers can also choose to be informed by text message. To keep costs low, a text message is sent only when the status changes to "Out for delivery" or to "Delivered".

The class `SmsNotifier` inherits from `Notifier`. Its constructor, `__init__()`, is passed the customer's mobile phone number, which is stored in a private attribute. Its `update()` method outputs the text message, as in this example:

SMS to 07700900123: P-48213 Delivered

- (b) Construct the class `SmsNotifier`. [4]
- (c) Explain how the observer design pattern has been applied in this system. [3]
- (d) Construct a UML class diagram that shows the relationships between the classes `Parcel`, `TrackingEvent`, `Notifier`, `EmailNotifier` and `SmsNotifier`. [3]

Attributes and methods do not need to be shown. [3]

Thirty days after a parcel has been delivered, the company's system deletes its `Parcel` object by removing every reference to it.

- (e) With reference to the relationships between the classes, describe what happens to the parcel's `TrackingEvent` objects and to its `Notifier` objects when its `Parcel` object is deleted. [3]

(Option B continues on the following page)

Turn over

(Option B, question 9 continued)

When a `Parcel` object is created, `self._history` is empty. The method `set_status(self, status, time)` works as follows:

- If `self._history` is not empty and `status` is the same as the status of the most recent `TrackingEvent` in `self._history`, the scan is a repeat and nothing is done.
- Otherwise, a new `TrackingEvent` object is created for `status` and `time` and added to the end of `self._history`. Then `update()` is called for every notifier in `self._notifiers`, passing it the parcel's code and the new status.

(f) Construct the method `set_status(self, status, time)`. [5]

(Option B continues on the following page)

(Option B continued)

10. A hospital pharmacy uses a computer system to manage its stock of medicines. The system stores sets of medicine names. **Table 9** shows the operations of the set abstract data type (ADT) that the programmers use.

Table 9: Operations of the set ADT used by the pharmacy system

Operation	Description
<code>s.add(x)</code>	Adds <code>x</code> to the set <code>s</code> , unless <code>x</code> is already in <code>s</code>
<code>s.discard(x)</code>	Removes <code>x</code> from the set <code>s</code> , if <code>x</code> is in <code>s</code>
<code>x in s</code>	Returns <code>True</code> if <code>x</code> is in the set <code>s</code> , and <code>False</code> otherwise
<code>len(s)</code>	Returns the number of elements in the set <code>s</code>
<code>for x in s:</code>	Visits each element <code>x</code> of the set <code>s</code> in turn, in no particular order

- (a) With reference to **Table 9**, outline **two** properties of an abstract data type (ADT). [2]

The system also stores a three-digit code for each medicine in a hash table with 11 slots, indexed 0 to 10. The hash function is `code % 11` (the remainder when the code is divided by 11). Collisions are resolved by linear probing: the following slots are checked in turn, continuing from slot 0 after slot 10, until an empty slot is found.

Table 10 shows the hash table after five codes have been inserted.

Table 10: The hash table after five codes have been inserted

Index	0	1	2	3	4	5	6	7	8	9	10
Code			189			357	412			603	846

The codes 731 and 384 are now inserted, in this order.

- (b) State the index of the slot in which each of these two codes is stored. [2]

The names of the different medicines on a patient's prescription are stored in the set `prescription`. The names of all the medicines that the pharmacy has in stock are stored in the set `stock`.

The function `can_dispense(prescription, stock)` returns `True` if every medicine in `prescription` is also in `stock` (that is, if `prescription` is a subset of `stock`), and `False` otherwise. If `prescription` is empty, the function returns `True`. Neither set is changed.

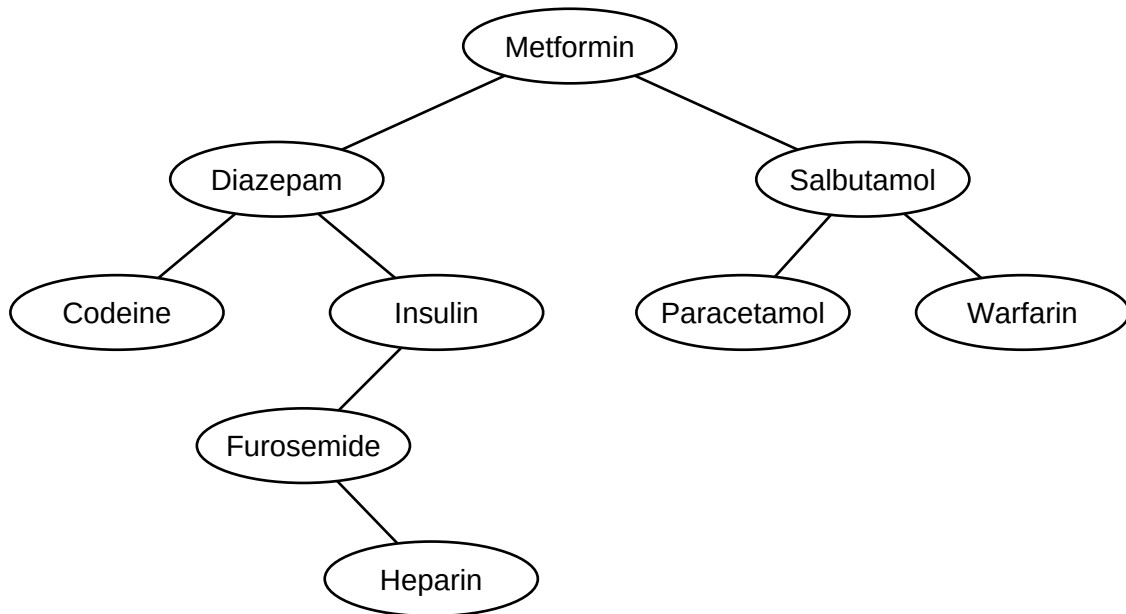
- (c) Using only the operations shown in **Table 9**, construct the function `can_dispense(prescription, stock)`. [3]

(Option B continues on the following page)

(Option B continued)

11. The pharmacy also stores the names of the medicines it supplies in a binary search tree (BST), in alphabetical order. **Figure 5** shows the BST after the first nine names have been inserted.

Figure 5: The binary search tree after the first nine names have been inserted



- (a) Sketch the binary search tree after the names Omeprazole and Ibuprofen have been inserted, in this order, into the tree in **Figure 5**. [2]

In a pre-order traversal, the node is visited first, then its left subtree is traversed, then its right subtree is traversed.

- (b) State the order in which the names in **Figure 5** are visited in a pre-order traversal. [2]
- (c) Without writing code, describe the steps needed to find and delete Diazepam from the binary search tree in **Figure 5**, so that it remains a binary search tree. [3]

After each half-life, the amount of a medicine in a patient's body is halved. The recursive function `remaining(dose, n)` returns the amount (in mg) that is left from a dose of dose mg after n half-lives:

```
def remaining(dose, n):  
    if n == 0:  
        return dose  
    else:  
        return remaining(dose / 2, n - 1)
```

- (d) Trace the call `remaining(400, 3)` by stating the sequence of calls that are made and the value returned by each call. [2]

(Option B continues on the following page)

(Option B, question 11 continued)

The number of tablets of a medicine dispensed on each day is stored in the list `tablets`, for example:

```
tablets = [120, 95, 140, 80, 110]
```

The recursive function `busiest(tablets, n)` returns the largest number of tablets dispensed on any one of the first n days, that is, the largest of `tablets[0]` to `tablets[n - 1]`. For example, `busiest(tablets, 2)` returns 120 and `busiest(tablets, 5)` returns 140.

(e) Construct the recursive function `busiest(tablets, n)`.

Assume that $1 \leq n \leq \text{len}(\text{tablets})$.

The `max()` function must not be used.

[4]

(Option B continues on the following page)

(Option B continued)

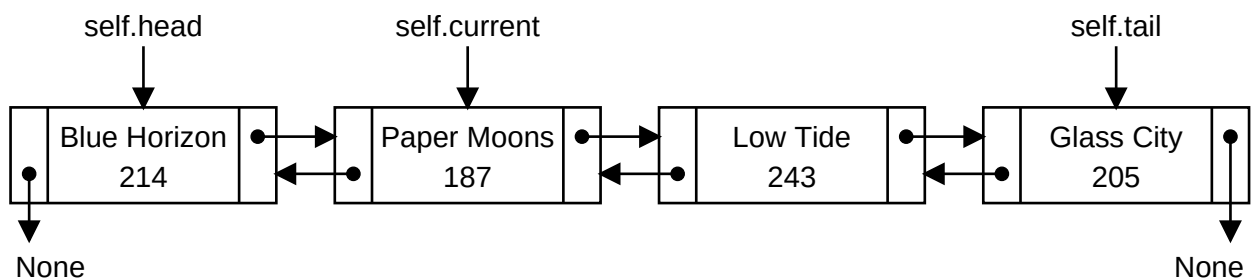
12. A music streaming service stores each playlist as a doubly linked list. Each node holds the title of a track and its length in seconds. The pointer next of each node points to the following node, and the pointer prev points to the previous node.

The linked list makes use of the Node class, which is shown:

```
class Node():  
    def __init__(self, title, seconds):  
        self.title = title          # e.g. "Paper Moons"  
        self.seconds = seconds     # length of the track  
        self.prev = None           # previous node in the playlist  
        self.next = None           # following node in the playlist
```

Figure 6 shows a playlist. The Node objects `self.head` and `self.tail` point to the first and the last nodes in the playlist, and the Node object `self.current` points to the node of the track that is playing. `self.head`, `self.tail` and `self.current` are attributes of the class `Playlist`, which contains the methods `play_next()` and `remove_track()`.

Figure 6: A playlist stored in a doubly linked list



The method `def play_next(self, new_track)` inserts the node `new_track` into the playlist immediately after `self.current`, so that it is the next track to be played. If `self.current` is the last node in the playlist, `new_track` becomes the last node.

- (a) Without writing code, describe the steps in the method `def play_next(self, new_track)`.

Assume that `self.current` is not `None`.

[5]

(Option B continues on the following page)

(Option B, question 12 continued)

The method `def remove_track(self, title)` removes the first node in the playlist whose title is equal to `title`. Both the `next` and the `prev` pointers must be updated so that the playlist remains a correct doubly linked list:

- If the node removed is the first node, `self.head` must be updated.
- If the node removed is the last node, `self.tail` must be updated.
- If no track in the playlist has this title, the method outputs a suitable message and the playlist is not changed.

(b) Construct the method `def remove_track(self, title)`.

Assume that the playlist contains at least two tracks and that the track to be removed is not the track that is playing.

[5]

End of Option B
