

Markscheme

Mock examination — Set 1

Computer science

Higher level

Paper 2

This markscheme is **confidential** and for the exclusive use of the teachers marking this mock examination.

It must **not** be shared with candidates before the mock examination has been sat.

General marking instructions

1. Follow the markscheme provided, award only whole marks and mark only in **RED**.
2. Where a mark is awarded, a tick/check (✓) **must** be placed in the text at the **precise point** where it becomes clear that the candidate deserves the mark. **One tick to be shown for each mark awarded.**
3. Sometimes, careful consideration is required to decide whether or not to award a mark. In these cases write a brief comment in the margin to support your decision, especially for moderation purposes. It should be remembered that the script may be returned to the candidate.
4. Personal codes/notations are unacceptable.
5. Where an answer to a part question is worth no marks but the candidate has attempted the part question, write a zero next to the answer. Where the candidate has not attempted the part question, write “NR”.
6. Ensure that you have viewed **every** page including any additional sheets.
7. A mark should not be awarded where there is contradiction within an answer. Write “CON” next to the contradiction.
8. **If a candidate has attempted more than one Option** within a paper, mark all the candidate’s work and award the marks for the higher scoring Option.

Subject details: Computer Science HL Paper 2 markscheme (Mock examination — Set 1)

Mark allocation

Candidates are required to answer **all** questions in **one** Option. Total 80 marks.

General

A markscheme often has more specific points worthy of a mark than the total allows. This is intentional. Do not award more than the maximum marks allowed for that part of a question.

When deciding upon alternative answers by candidates to those given in the markscheme, consider the following points:

- Each statement worth one point has a separate line and the end is signified by means of a semi-colon (;).
- An alternative answer or wording is indicated in the markscheme by a “/”; either wording can be accepted.
- Words in (...) in the markscheme are not necessary to gain the mark.
- If the candidate’s answer has the same meaning or can be clearly interpreted as being the same as that in the markscheme then award the mark.
- Mark positively. Give candidates credit for what they have achieved and for what they have got correct, rather than penalizing them for what they have not achieved or what they have got wrong.
- Remember that many candidates are writing in a second language; be forgiving of minor linguistic slips. In this subject effective communication is more important than grammatical accuracy.
- Occasionally, a part of a question may require a calculation whose answer is required for subsequent parts. If an error is made in the first part then it should be penalized. However, if the incorrect answer is used correctly in subsequent parts then **follow through** marks should be awarded. Indicate this with “FT”.

General guidance

Issue	Guidance
Answering more than the quantity of responses prescribed in the questions	• In the case of an “identify” question, read all answers and mark positively up to the maximum marks. Disregard incorrect answers. • In the case of a “describe” question, which asks for a certain number of facts eg “describe two kinds”, mark the first two correct answers. This could include two descriptions, one description and one identification, or two identifications. • In the case of an “explain” question, which asks for a specified number of explanations eg “explain two reasons ...”, mark the first two correct answers. This could include two full explanations, one explanation, one partial explanation etc.

Option A — Java

1. (a) Award [1 max]

N must be a whole number / an integer;

N must not be negative / N must be greater than or equal to 0 / N must be positive;

Note: Do not accept answers that only say that N must be a number.

(b) Award [3] as follows:

correct values of N;

correct TRUE/FALSE values in both decision columns;

correct values of count and correct output;

N	N > 0 ?	N is odd ?	count	output
13			0	
	TRUE	TRUE	1	
6	TRUE	FALSE		
3	TRUE	TRUE	2	
1	TRUE	TRUE	3	
0	FALSE			3

Note: Accept any layout in which the values appear in the correct order. Award FT for later values that follow correctly from an earlier error.

(c) Award [2 max]

$O(\log N)$ / logarithmic time;

...because the value of N is (about) halved on each pass through the loop;

...so the loop is repeated once for each binary digit of N / about $\log_2 N$ times;

...so doubling N adds only one more pass through the loop / the time grows very slowly as N increases;

Note: Award [1] for $O(\log N)$ and [1] for a valid description linked to the halving of N. Do not accept $O(N)$ or $O(N^2)$ for the first mark; a correct description of the halving can still gain [1].

(d) Award [4 max]

the search starts with the whole list / low = 0 and high = 14;

the middle item, 47 (index 7), is compared with 29; 29 is smaller so indices 7 to 14 are discarded / high = 6;

the middle of indices 0 to 6, 23 (index 3), is compared with 29; 29 is larger so indices 0 to 3 are discarded / low = 4;

the middle of indices 4 to 6, 34 (index 5), is compared with 29; 29 is smaller so only index 4 remains / high = 4;

the item at index 4 is 29, which matches, so the search stops and the value is found (at index 4);

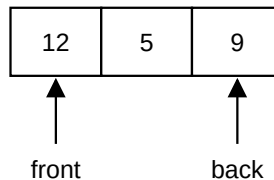
Comparison	low	high	mid	Value at mid	Result
1	0	14	7	47	$29 < 47$, so high = 6
2	0	6	3	23	$29 > 23$, so low = 4
3	4	6	5	34	$29 < 34$, so high = 4
4	4	4	4	29	found at index 4

*Note: Accept positions numbered from 1 (middle positions 8, 4, 6 and 5). Allow FT if a different but consistent choice of middle item is made. A general description of a binary search that is not applied to this list gains **[2]** max. Award **[3 max]** if the search is not shown to end by finding 29 at index 4.*

(e) Award **[2]**.

correct elements in the correct order (12, 5, 9) [1];

correct indication of the front (12) and the back (9) [1];



Note: Accept the queue drawn horizontally in either direction or vertically, provided that the front and the back are clearly labelled.

(f) Award **[3]** as follows:

Award **[1]** for each correct line (or equivalent).

repeat until isEmpty() is true / while isEmpty() is false;

 dequeue an element and enqueue it onto a second (empty) queue only if it is not negative / is 0 or more;

then dequeue each element of the second queue and enqueue it back onto the original queue (until the second queue is empty) / use the second queue in place of the original queue;

Alternative answer

enqueue a marker value (that cannot be in the queue) at the back of the queue;

repeatedly dequeue an element until the marker is dequeued;

 if the element is not negative, enqueue it again at the back of the queue (otherwise do not enqueue it);

Note: Answer must demonstrate an algorithmic approach.

2. (a) **Award [2 max]**

the memory is allocated once, when the structure is created / no time is spent resizing the structure;

...as the maximum number of passengers (150) is known in advance, a fixed size is sufficient;

the amount of memory used is fixed/predictable;

...so the system will not need to request more memory while tickets are being sold / there is no overhead for pointers (unlike a linked list);

the elements are stored in consecutive memory locations / can be accessed directly by their index (unlike a linked list);

...so a passenger name can be found/accessed quickly;

Note: Award [1] for an advantage and [1] for an expansion. The expansion does not need to refer to the ferry for the mark.

(b) **Award [2] as follows:**

Award [1] for each correct pass.

first pass: 48, 35, 90, 61, 27, 112;

second pass: 35, 48, 61, 27, 90, 112;

Note: Award FT for the second pass if it follows correctly from an incorrect first pass.

(c) **Award [2 max]**

Integer.parseInt("3x") cannot convert the text to an integer, so a NumberFormatException is thrown / an exception occurs (and tickets is not changed);

the catch block handles the exception and outputs "Enter a whole number" (instead of the program crashing);

the finally block is always executed, so "Input checked" is also output;

the method returns 0 / the initial value of tickets;

(d) **Award [2 max]**

lowest is initialized to 0 / no fare is less than 0, so lowest is never updated (and "Lowest fare: 0.0" is output);

initialize lowest to the first fare, fares[0] / to a value larger than any possible fare (eg 1000);

Note: Award [1] for identifying the error and [1] for the correction. Changing < to > is not a correct correction (it would output the highest fare). Replacing the loop with a library call (eg Arrays.stream(fares).min()) does not gain the correction mark, because the given code must be corrected.

(e) **Award [3 max] as follows:**

Award [1] for a method header with a String parameter that returns a String;

Award [1] for extracting the hours (characters at index 8 and 9);

Award [1] for extracting the minutes (characters at index 10 and 11);

Award [1] for joining the hours, ":" and the minutes and returning the result;

Example answer

```
public static String departureTime(String code)
{
    String hours = code.substring(8, 10);
    String minutes = code.substring(10, 12);    // allow substring(10)
    return hours + ":" + minutes;
}
```

*Note: Accept any method that correctly isolates the time, eg `split("-")`, `lastIndexOf("-")`, `code.substring(code.length() - 4)`, or a character-by-character answer such as `"" + code.charAt(8) + code.charAt(9) + ":" + code.charAt(10) + code.charAt(11)`. Award **[2 max]** if the result is not in the form `hh:mm`. Ignore minor syntax errors; penalize a repeated error once only.*

(f) Award **[4 max]** as follows:

Award [1] for opening `tickets.txt` for reading;
Award [1] for a loop that reads the file line by line until the end of the file;
Award [1] for comparing the first three characters of each line with the parameter (using `equals()`);
Award [1] for initializing and incrementing a counter and outputting it after the loop;
Award [1] for closing the file;

Example answer

```
public static void countFrom(String port) throws IOException
{
    BufferedReader reader = new BufferedReader(new FileReader("tickets.txt"));
    int count = 0;
    String line = reader.readLine();
    while (line != null)
    {
        if (line.substring(0, 3).equals(port)) // allow line.startsWith(port)
            count = count + 1;
        line = reader.readLine();
    }
    reader.close();
    System.out.println ("Tickets sold from " + port + ": " + count);
}
```

Alternative answer

```
public static void countFrom(String port) throws IOException
{
    Scanner reader = new Scanner(new File("tickets.txt"));
    int count = 0;
    while (reader.hasNextLine())
    {
        String line = reader.nextLine();
        if (line.substring(0, 3).equals(port))
            count = count + 1;
    }
    reader.close();
    System.out.println ("Tickets sold from " + port + ": " + count);
}
```

Note: Exception handling (throws IOException or try/catch) is not required for any mark. Comparing the Strings with == does not gain the comparison mark. contains(port) / indexOf(port) >= 0 does not gain the comparison mark (it would also count arrivals, eg CAL-ARD-1030). Accept line.split("-")[0].equals(port), line.startsWith(port) or line.indexOf(port) == 0. Ignore minor syntax errors; penalize a repeated error once only.

3. (a) *Award [2 max]*

Code reuse: the attributes code, title and value and the methods getCode()/getTitle() are written once in Exhibit;
...and are inherited by both Painting and Fossil, so they do not have to be written again in each class;

Easier maintenance: a change to the shared code (eg to getTitle()) is made only once, in Exhibit;
...and it automatically applies to both Painting and Fossil / fewer places where errors can occur;

Extensibility: a new type of exhibit (eg Sculpture) can be added by extending Exhibit;
...so only its extra attributes and its own insuranceCost() method need to be written;

Award [1] for an advantage and [1] for an expansion that refers to these classes.

(b) *Award [3 max]*

Award [1] for three class boxes, each with the class name, the attributes and the methods in separate compartments;

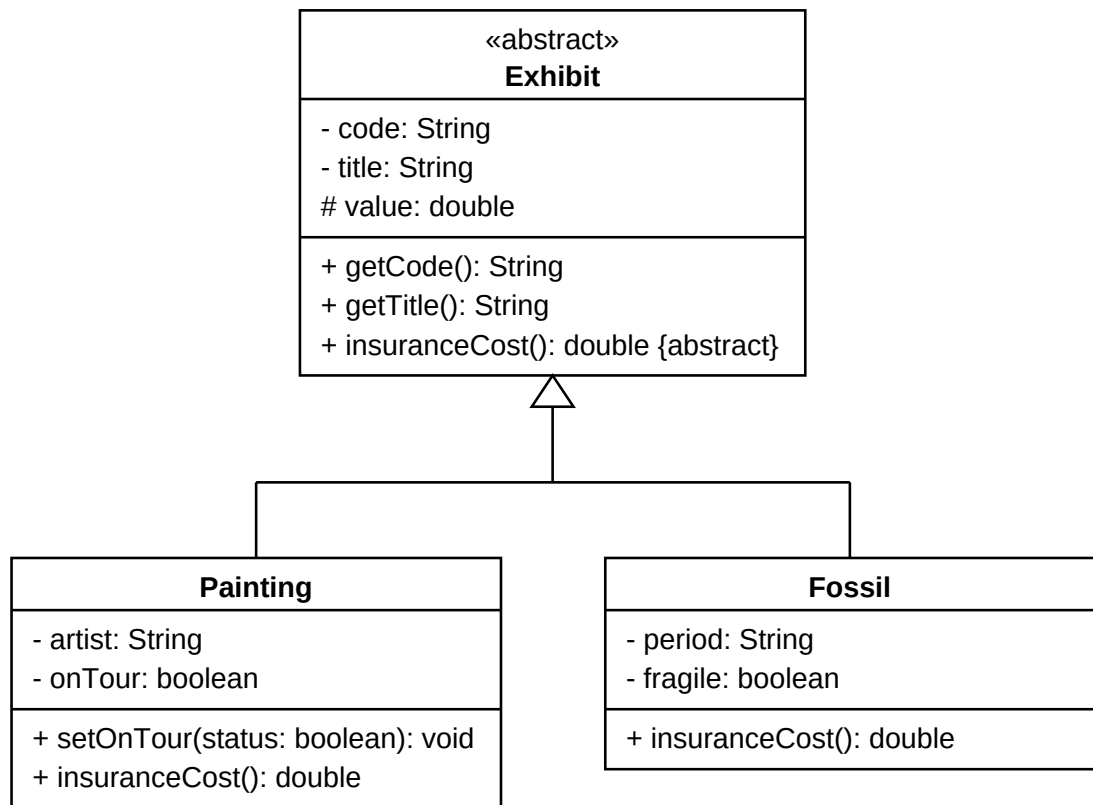
Award [1] for correct data types and visibility symbols (– private, # protected, + public);

Award [1] for inheritance shown by lines/arrows with a hollow (unfilled) triangular head from Painting and from Fossil to Exhibit;

Award [1] for showing Exhibit and insuranceCost() as abstract (italics / «abstract» / {abstract});

Constructors are not required. Accept minor variations in UML notation, eg the type written before the name, or two separate inheritance arrows.

Example answer



(c) **Award [3 max]**

every exhibit is either a painting or a fossil / Exhibit is a general category, not a real exhibit;
...so declaring Exhibit abstract prevents an Exhibit object from being instantiated/created;
there is no general way to calculate the insurance cost of an exhibit, so `insuranceCost()` has no body in Exhibit;
declaring `insuranceCost()` abstract forces every subclass (Painting, Fossil) to override/implement it;
...which guarantees that every exhibit has an `insuranceCost()` method / provides a common interface for the subclasses;
...so `insuranceCost()` can be called on any Exhibit reference / supports polymorphism;

(d) **Award [5] as follows:**

Constructor

Award [1] for `super(code, title, value)` as the first statement;
Award [1] for assigning artist and setting `onTour` to false (accept relying on the default value false);

insuranceCost()

Award [1] for calculating 1.5 % of value (`value * 0.015`);
Award [1] for doubling the cost when `onTour` is true;
Award [1] for applying the minimum cost of 250 (after any doubling);

No return statement in `insuranceCost()`: award at most [2] for the method ([4 max] in total).

Example answer

```
public Painting(String code, String title, double value,
                String artist)
{
    super(code, title, value);
    this.artist = artist;
    this.onTour = false;
}

public double insuranceCost()
{
    double cost = value * 0.015;
    if (onTour)                // allow onTour == true
        cost = cost * 2;
    if (cost < 250)
        cost = 250;           // allow Math.max(cost, 250)
    return cost;
}
```

Exhibit has no default constructor and code/title are private, so super(...) must be used to initialize the inherited attributes; a constructor without it does not compile.

(e) **Award [3 max]**

a separate factory class/method (eg createExhibit()) is responsible for creating all of the Exhibit objects;
the factory method is passed the type letter and the data from each record;
it uses selection on the type letter to decide whether to instantiate a Painting or a Fossil;
it returns the new object as an Exhibit (superclass) reference;
the rest of the program never calls the Painting/Fossil constructors directly / does not need to know which subclass is created;
a new type of exhibit can be added by changing only the factory;

(f) **Award [4] as follows:**

Award [1] for initializing a total to 0 and adding each exhibit's cost to it;
Award [1] for a loop that calls insuranceCost() on every element of collection (without checking its subclass/casting);
Award [1] for comparing the cost with limit and outputting the code and title using getCode()/getTitle();
Award [1] for outputting the total after the loop has finished;

```
public static void insuranceReport(Exhibit[] collection, double limit)
{
    double total = 0;
    for (int i = 0; i < collection.length; i++)
    {
        double cost = collection[i].insuranceCost();
        total = total + cost;
        if (cost > limit)
            System.out.println(collection[i].getCode() + " "
                               + collection[i].getTitle());
    }
    System.out.println("Total insurance cost: " + total);
}
```

*Award **[3 max]** if the total is not output.*

Calling `insuranceCost()` through an `Exhibit` reference is polymorphism: the version in `Painting` or in `Fossil` is selected at run time. Accept a non-static method.

4. (a) **Award [2 max]**

the load factor is the number of items stored divided by the number of slots;
a high load factor means that collisions are more likely / more words are hashed to the same slot;
...so adding a word or checking whether a word is in the set takes longer / needs more comparisons or probes (tends towards $O(n)$);
resizing (and rehashing the words into more slots) reduces the load factor;
...so add()/contains() remain fast / close to $O(1)$;

(b) **Award [2 max]**

[playa, rojo];
7;

Award [1] for each line. Accept the two words in the first line in either order, with or without brackets/commas (a set is unordered). The second line is 7 because wordsA gains rojo and playa ($5 + 2$).

(c) **Award [3 max]**

Award [1] for a loop that visits every element of typed;
Award [1] for checking whether each word is already in mastered (contains());
Award [1] for adding the word to mastered;
Award [1] for counting only the words that were not already in mastered and returning the count;

```
public int countNew(String[] typed, HashSet<String> mastered)
{
    int count = 0;
    for (int i = 0; i < typed.length; i++)
    {
        if (!mastered.contains(typed[i]))    // allow == false
        {
            mastered.add(typed[i]);
            count = count + 1;
        }
    }
    return count;
}
// allow: if (mastered.add(typed[i])) count++;
// allow: store mastered.size() before the loop, add every
//         word, then return the increase in size
```

Alternative answer

```
public int countNew(String[] typed, HashSet<String> mastered)
{
    HashSet<String> newWords =
        new HashSet<String>(Arrays.asList(typed));
    newWords.removeAll(mastered);
    mastered.addAll(newWords);
    return newWords.size();
}
```

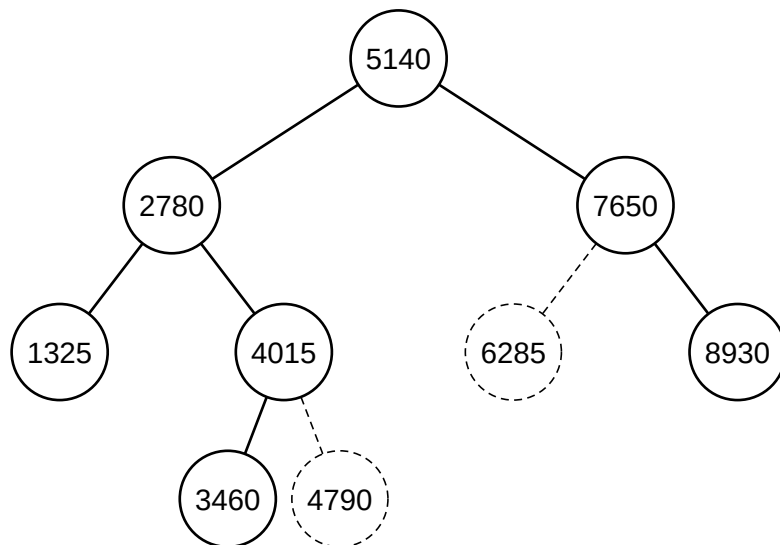
*Award **[2 max]** if words are not added to mastered or a repeated word is counted twice.*

Accept set operations: `removeAll()/addAll()` on a copy of the words in `typed` for the loop and checking marks; `mastered.addAll(Arrays.asList(typed))` for the loop and adding marks.

A repeated word in `typed` is counted only once, because it is already in `mastered` when it is met again.

5. (a) Award [2 max]

Award [1] for 6285 as the left child of 7650;
Award [1] for 4790 as the right child of 4015;



The new nodes are shown with a thinner, dashed outline.

(b) Award [2 max]

1325, 3460, 4015, 2780, 8930, 7650, 5140;

Award [1] for 1325, 3460, 4015, 2780 in this order (the left subtree) and [1] for 8930, 7650, 5140 in this order after them.

(c) Award [2 max]

if the codes are inserted in (ascending/descending) sorted order, each new code is added on the same side of the tree;

...so the tree becomes unbalanced / a long chain, like a linked list;

...and finding a code may need a comparison with every node / $O(n)$;

if the codes are inserted in a mixed order, the tree is (more) balanced;

...so each comparison eliminates about half of the remaining codes / $O(\log n)$;

Award [1] for the effect of the insertion order on the shape of the tree and [1] for linking the shape to the number of comparisons needed.

(d) (i) Award [1 max]

10;

(ii) *Award [2 max]*

each recursive call is placed on the (call) stack / creates a new stack frame;
...which holds its own copy of code/the parameter and the return address;
these frames stay in memory until the base case is reached and the calls return;
check(4015) has five calls on the stack at the same time / the memory used grows with the number of digits;
an iterative method reuses the same variables / uses a fixed amount of memory;

(e) *Award [4 max]*

Award [1] for a base case that returns 0 for an empty string;
Award [1] for comparing the first character of desc with c;
Award [1] for a recursive call on the rest of the string (desc.substring(1)), which leads to the base case;
Award [1] for adding 1 only when the first character matches;
Award [1] for a completely correct method that returns an int;

```
public int countChar(String desc, char c)
{
    if (desc.length() == 0)                // allow desc.equals("")
        return 0;
    int rest = countChar(desc.substring(1), c);
    if (desc.charAt(0) == c)
        return rest + 1;
    else
        return rest;
}
```

Award [2 max] if the method is not recursive.

Accept a recursive call in each branch of an if ... else (only one call is made each time), and a solution that works from the last character (desc.substring(0, desc.length() - 1)).

6. (a) **Award [5 max]**

Award [1] for checking whether the head node matches name and, if so, moving head to the next node;

Award [1] for traversing the list using a temporary node (and a previous node) until the end of the list;

Award [1] for correctly comparing fileName with name;

Award [1] for linking the previous node to the node after the matching node;

Award [1] for stopping after the first match;

Award [1] for outputting a suitable message only when no node matches;

Award [3 max] if a match at the head is not handled. Award [4 max] if no message is output when no node matches.

Example answer

```
public void removePhoto(String name)
{
    boolean removed = false;
    if (head.fileName.equals(name))    // allow compareTo(name) == 0
    {
        head = head.next;
        removed = true;
    }
    else
    {
        Node previous = head;
        Node current = head.next;
        while (current != null && removed == false)
        {
            if (current.fileName.equals(name))
            {
                previous.next = current.next;
                removed = true;
            }
            else
            {
                previous = current;
                current = current.next;
            }
        }
    }
    if (removed == false)
        System.out.println(name + " not found");
}
```

(b) Award **[5 max]**

The algorithm will:

- check whether head is null / the list is empty;
- if it is, set head to newPhoto and set the next pointer of newPhoto to newPhoto (itself);
- otherwise, make a temporary node equal to head;
- repeatedly move the temporary node to the next node;
- until the next pointer of the temporary node is equal to head / the last node is reached (not until null);
- set the next pointer of the last node to newPhoto;
- set the next pointer of newPhoto to head;

*Award **[3 max]** if newPhoto's next pointer is not set to head (the list is no longer circular). Award **[4 max]** if the empty list is not handled. Award **[3 max]** if the traversal stops when a null pointer is reached (a non-empty circular list contains no null pointer). Allow the use of diagrams to illustrate any of the above points.*

Algorithm is:

1. If head is null, set head to newPhoto and set the next pointer of newPhoto to newPhoto.
2. Otherwise, set a temporary node, temp, equal to head.
3. While the next pointer of temp is not equal to head, set temp to the next pointer of temp.
4. Set the next pointer of temp to newPhoto.
5. Set the next pointer of newPhoto to head.

Option B — Python

7. (a) Award [1 max]

N must be a whole number / an integer;

N must not be negative / N must be greater than or equal to 0 / N must be positive;

Note: Do not accept answers that only say that N must be a number.

(b) Award [3] as follows:

correct values of N;

correct TRUE/FALSE values in both decision columns;

correct values of count and correct output;

N	N > 0 ?	N is odd ?	count	output
13			0	
	TRUE	TRUE	1	
6	TRUE	FALSE		
3	TRUE	TRUE	2	
1	TRUE	TRUE	3	
0	FALSE			3

Note: Accept any layout in which the values appear in the correct order. Award FT for later values that follow correctly from an earlier error.

(c) Award [2 max]

$O(\log N)$ / logarithmic time;

...because the value of N is (about) halved on each pass through the loop;

...so the loop is repeated once for each binary digit of N / about $\log_2 N$ times;

...so doubling N adds only one more pass through the loop / the time grows very slowly as N increases;

Note: Award [1] for $O(\log N)$ and [1] for a valid description linked to the halving of N. Do not accept $O(N)$ or $O(N^2)$ for the first mark; a correct description of the halving can still gain [1].

(d) Award [4 max]

the search starts with the whole list / low = 0 and high = 14;

the middle item, 47 (index 7), is compared with 29; 29 is smaller so indices 7 to 14 are discarded / high = 6;

the middle of indices 0 to 6, 23 (index 3), is compared with 29; 29 is larger so indices 0 to 3 are discarded / low = 4;

the middle of indices 4 to 6, 34 (index 5), is compared with 29; 29 is smaller so only index 4 remains / high = 4;

the item at index 4 is 29, which matches, so the search stops and the value is found (at index 4);

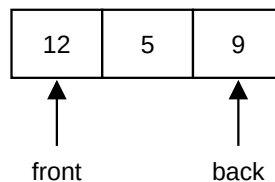
Comparison	low	high	mid	Value at mid	Result
1	0	14	7	47	$29 < 47$, so high = 6
2	0	6	3	23	$29 > 23$, so low = 4
3	4	6	5	34	$29 < 34$, so high = 4
4	4	4	4	29	found at index 4

*Note: Accept positions numbered from 1 (middle positions 8, 4, 6 and 5). Allow FT if a different but consistent choice of middle item is made. A general description of a binary search that is not applied to this list gains **[2]** max. Award **[3 max]** if the search is not shown to end by finding 29 at index 4.*

(e) Award **[2]**.

correct elements in the correct order (12, 5, 9) [1];

correct indication of the front (12) and the back (9) [1];



Note: Accept the queue drawn horizontally in either direction or vertically, provided that the front and the back are clearly labelled.

(f) Award **[3]** as follows:

Award **[1]** for each correct line (or equivalent).

repeat until isEmpty() is true / while isEmpty() is false;

 dequeue an element and enqueue it onto a second (empty) queue only if it is not negative / is 0 or more;

then dequeue each element of the second queue and enqueue it back onto the original queue (until the second queue is empty) / use the second queue in place of the original queue;

Alternative answer

enqueue a marker value (that cannot be in the queue) at the back of the queue;

repeatedly dequeue an element until the marker is dequeued;

 if the element is not negative, enqueue it again at the back of the queue (otherwise do not enqueue it);

Note: Answer must demonstrate an algorithmic approach.

8. (a) **Award [2 max]**

the memory is allocated once, when the structure is created / no time is spent resizing the structure;

...as the maximum number of passengers (150) is known in advance, a fixed size is sufficient;

the amount of memory used is fixed/predictable;

...so the system will not need to request more memory while tickets are being sold / there is no overhead for pointers (unlike a linked list);

the elements are stored in consecutive memory locations / can be accessed directly by their index (unlike a linked list);

...so a passenger name can be found/accessed quickly;

Note: Award [1] for an advantage and [1] for an expansion. The expansion does not need to refer to the ferry for the mark.

(b) **Award [2] as follows:**

Award [1] for each correct pass.

first pass: [48, 35, 90, 61, 27, 112];

second pass: [35, 48, 61, 27, 90, 112];

Note: Award FT for the second pass if it follows correctly from an incorrect first pass.

(c) **Award [2 max]**

int("3x") cannot convert the text to an integer, so a ValueError is raised / an exception occurs (and tickets is not changed);

the except block handles the exception and outputs "Enter a whole number" (instead of the program crashing);

the finally block is always executed, so "Input checked" is also output;

the function returns 0 / the initial value of tickets;

(d) **Award [2 max]**

lowest is initialized to 0 / no fare is less than 0, so lowest is never updated (and "Lowest fare: 0" is output);

initialize lowest to the first fare, fares[0] / to a value larger than any possible fare (eg 1000);

Note: Award [1] for identifying the error and [1] for the correction. Changing < to > is not a correct correction (it would output the highest fare). Replacing the loop with lowest = min(fares) does not gain the correction mark, because the given code must be corrected.

(e) **Award [3 max] as follows:**

Award [1] for a function definition with one parameter that returns a string;

Award [1] for extracting the hours (characters at index 8 and 9);

Award [1] for extracting the minutes (characters at index 10 and 11);

Award [1] for joining the hours, ":" and the minutes and returning the result;

Example answer

```
def departure_time(code):
    hours = code[8:10]
    minutes = code[10:12]          # allow code[10:]
    return hours + ":" + minutes
```

*Note: Accept any method that correctly isolates the time, eg `split("-")`, `rfind("-")`, `code[-4:]`, or a character-by-character answer such as `code[8] + code[9] + ":" + code[10] + code[11]`. Award **[2 max]** if the result is not in the form `hh:mm`. Ignore minor syntax errors; penalize a repeated error once only.*

(f) Award **[4 max]** as follows:

Award [1] for opening `tickets.txt` for reading;
Award [1] for a loop that reads the file line by line until the end of the file;
Award [1] for comparing the first three characters of each line with the parameter;
Award [1] for initializing and incrementing a counter and outputting it after the loop;
Award [1] for closing the file;

Example answer

```
def count_from(port):
    file = open("tickets.txt", "r")
    count = 0
    line = file.readline()
    while line != "":
        if line[0:3] == port:          # allow line.startswith(port)
            count = count + 1
        line = file.readline()
    file.close()
    print("Tickets sold from", port, ":", count)
```

Alternative answer

```
def count_from(port):
    file = open("tickets.txt", "r")
    count = 0
    for line in file:
        if line[0:3] == port:
            count = count + 1
    file.close()
    print("Tickets sold from", port, ":", count)
```

Note: Accept with `open("tickets.txt", "r")` as `file`: for both the opening and the closing marks. `port in line / line.find(port) >= 0` does not gain the comparison mark (it would also count arrivals, eg `CAL-ARD-1030`). Accept `line.split("-")[0] == port`, `line.startswith(port)`, and a loop over `file.readlines()`. Exception handling is not required for any mark. Ignore minor syntax errors; penalize a repeated error once only.

9. (a) *Award [2 max]*

Code reuse: the attributes `_code`, `_title` and `_value` and the methods `get_code()/get_title()` are written once in `Exhibit`;
...and are inherited by both `Painting` and `Fossil`, so they do not have to be written again in each class;

Easier maintenance: a change to the shared code (eg to `get_title()`) is made only once, in `Exhibit`;
...and it automatically applies to both `Painting` and `Fossil` / fewer places where errors can occur;

Extensibility: a new type of exhibit (eg `Sculpture`) can be added by inheriting from `Exhibit`;
...so only its extra attributes and its own `insurance_cost()` method need to be written;

Award [1] for an advantage and [1] for an expansion that refers to these classes.

(b) *Award [3 max]*

Award [1] for three class boxes, each with the class name, the attributes and the methods in separate compartments;

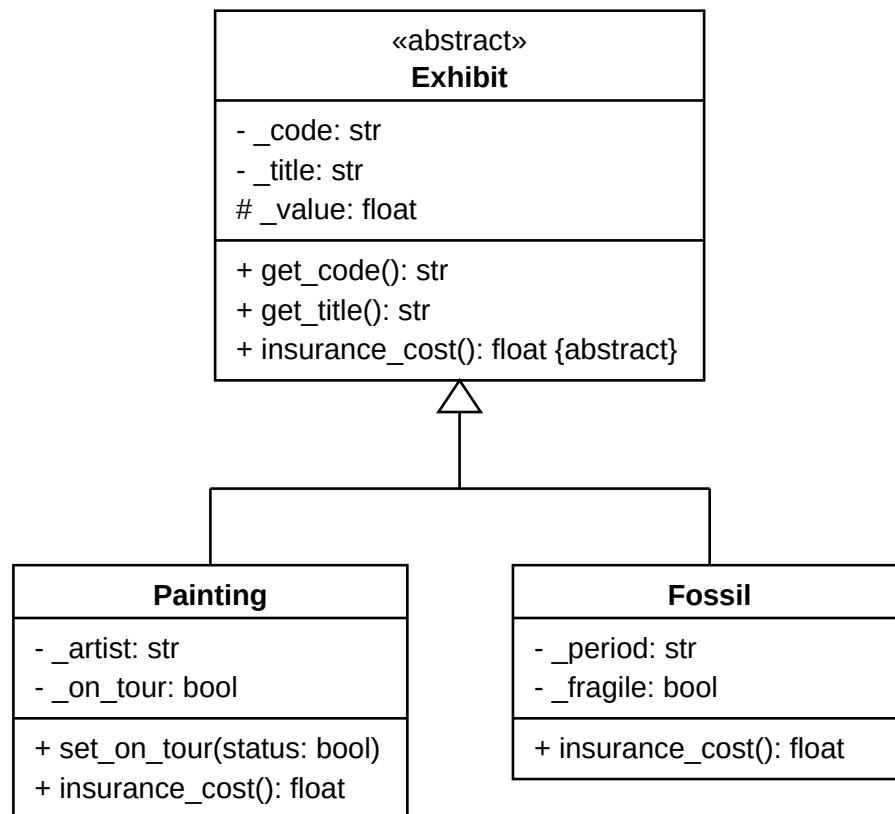
Award [1] for data types and visibility symbols (– private, # protected, + public);

Award [1] for inheritance shown by lines/arrows with a hollow (unfilled) triangular head from `Painting` and from `Fossil` to `Exhibit`;

Award [1] for showing `Exhibit` and `insurance_cost()` as abstract (italics / «abstract» / {abstract});

Constructors are not required. Accept – or # for any underscore attribute. Accept attribute names with or without the underscore, any reasonable type names (str/String, float/double, bool/boolean) and minor variations in UML notation.

Example answer



(c) **Award [3 max]**

every exhibit is either a painting or a fossil / Exhibit is a general category, not a real exhibit;
...so declaring Exhibit abstract prevents an Exhibit object from being instantiated/created (a `TypeError` is raised);
there is no general way to calculate the insurance cost of an exhibit, so `insurance_cost()` has no body in Exhibit;
declaring `insurance_cost()` abstract forces every subclass (Painting, Fossil) to override/implement it;
...which guarantees that every exhibit has an `insurance_cost()` method / provides a common interface for the subclasses;
...so `insurance_cost()` can be called on any Exhibit object / supports polymorphism;

(d) **Award [5] as follows:**

Constructor

Award [1] for `super().__init__(code, title, value);`
Award [1] for `self._artist = artist` **and** `self._on_tour = False;`

insurance_cost()

Award [1] for calculating 1.5 % of the value (`self._value * 0.015`);
Award [1] for doubling the cost when `self._on_tour` is `True`;
Award [1] for applying the minimum cost of 250 (after any doubling);

No return statement in `insurance_cost()`: award at most **[2]** for the method (**[4 max]** in total).
Allow `Exhibit.__init__(self, code, title, value)` for the first mark.

Example answer

```
def __init__(self, code, title, value, artist):
    super().__init__(code, title, value)
    self._artist = artist
    self._on_tour = False

def insurance_cost(self):
    cost = self._value * 0.015
    if self._on_tour == True:
        cost = cost * 2
    if cost < 250:
        cost = 250          # allow cost = max(cost, 250)
    return cost
```

`self._on_tour` must be created in `__init__()`: without it, `insurance_cost()` raises an `AttributeError` for a painting that has never been sent on tour.

(e) **Award [3 max]**

a separate factory class/function (eg `create_exhibit()`) is responsible for creating all of the Exhibit objects;
the factory function is passed the type letter and the data from each record;
it uses selection on the type letter to decide whether to instantiate a `Painting` or a `Fossil`;
it returns the new object (which can be used as an `Exhibit`);
the rest of the program never calls `Painting()`/`Fossil()` directly / does not need to know which subclass is created;
a new type of exhibit can be added by changing only the factory;

(f) **Award [4] as follows:**

Award [1] for initializing a total to 0 and adding each exhibit's cost to it;
Award [1] for a loop that calls `insurance_cost()` on every element of collection (without checking its subclass);
Award [1] for comparing the cost with limit and outputting the code and title using `get_code()/get_title()`;
Award [1] for outputting the total after the loop has finished;

```
def insurance_report(collection, limit):
    total = 0
    for exhibit in collection:
        cost = exhibit.insurance_cost()
        total = total + cost
        if cost > limit:
            print (exhibit.get_code(), exhibit.get_title())
    print ("Total insurance cost:", total)
# allow for i in range(len(collection)): ... collection[i] ...
```

Award [3 max] if the total is not output.

Calling `insurance_cost()` on each element without checking its class is polymorphism: the version in `Painting` or in `Fossil` is selected at run time.

10. (a) **Award [2 max]**

the load factor is the number of items stored divided by the number of slots;
a high load factor means that collisions are more likely / more words are hashed to the same slot;
...so adding a word or checking whether a word is in the set takes longer / needs more comparisons or probes (tends towards $O(n)$);
resizing (and rehashing the words into more slots) reduces the load factor;
...so add()/in remain fast / close to $O(1)$;

(b) **Award [2 max]**

```
{'rojo', 'playa'};  
7;
```

Award [1] for each line. Accept the two words in the first line in either order, with or without quotes/braces (a set is unordered). The second line is 7 because the union holds 5 + 2 different words.

(c) **Award [3 max]**

Award [1] for a loop that visits every element of typed;
Award [1] for checking whether each word is already in mastered (in / not in);
Award [1] for adding the word to mastered;
Award [1] for counting only the words that were not already in mastered and returning the count;

```
def count_new(typed, mastered):  
    count = 0  
    for word in typed:  
        if word not in mastered:  
            mastered.add(word)  
            count = count + 1  
    return count
```

Alternative answers

```
def count_new(typed, mastered):  
    before = len(mastered)  
    for word in typed:  
        mastered.add(word)  
    return len(mastered) - before  
  
def count_new(typed, mastered):  
    new_words = set(typed) - mastered  
    mastered.update(new_words)      # allow mastered |= new_words  
    return len(new_words)
```

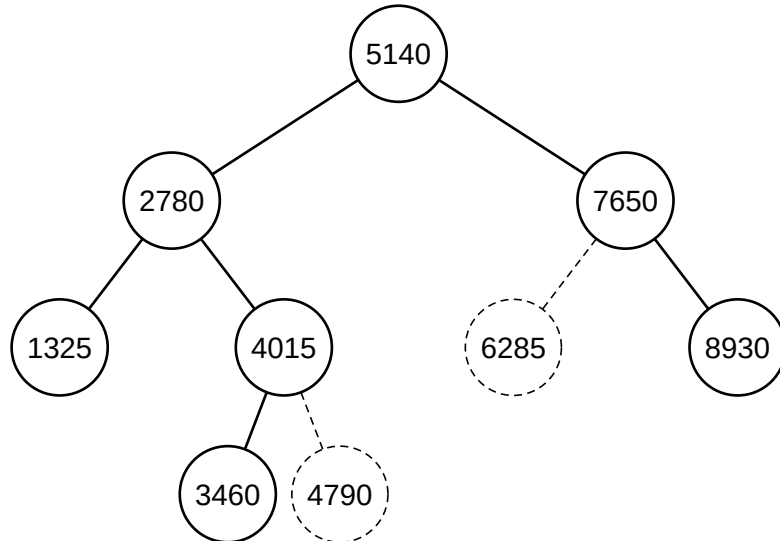
Award [2 max] if words are not added to mastered or a repeated word is counted twice.

*Accept set operations: `set(typed) - mastered` for the loop and checking marks;
`mastered.update(typed) / mastered |= set(typed)` for the loop and adding marks.*

A repeated word in typed is counted only once, because it is already in mastered when it is met again.

11. (a) Award [2 max]

Award [1] for 6285 as the left child of 7650;
Award [1] for 4790 as the right child of 4015;



The new nodes are shown with a thinner, dashed outline.

(b) Award [2 max]

1325, 3460, 4015, 2780, 8930, 7650, 5140;

Award [1] for 1325, 3460, 4015, 2780 in this order (the left subtree) and [1] for 8930, 7650, 5140 in this order after them.

(c) Award [2 max]

if the codes are inserted in (ascending/descending) sorted order, each new code is added on the same side of the tree;

...so the tree becomes unbalanced / a long chain, like a linked list;

...and finding a code may need a comparison with every node / $O(n)$;

if the codes are inserted in a mixed order, the tree is (more) balanced;

...so each comparison eliminates about half of the remaining codes / $O(\log n)$;

Award [1] for the effect of the insertion order on the shape of the tree and [1] for linking the shape to the number of comparisons needed.

(d) (i) Award [1 max]

10;

(ii) *Award [2 max]*

each recursive call is placed on the (call) stack / creates a new stack frame;
...which holds its own copy of code/the parameter and the return address;
these frames stay in memory until the base case is reached and the calls return;
check(4015) has five calls on the stack at the same time / the memory used grows with the number of digits;
an iterative function reuses the same variables / uses a fixed amount of memory;

(e) *Award [4 max]*

Award [1] for a base case that returns 0 for an empty string;
Award [1] for comparing the first character of desc with ch;
Award [1] for a recursive call on the rest of the string (desc[1:]), which leads to the base case;
Award [1] for adding 1 only when the first character matches;
Award [1] for a completely correct function that returns the count;

```
def count_char(desc, ch):  
    if desc == "":  
        return 0  
    rest = count_char(desc[1:], ch)  
    if desc[0] == ch:  
        return rest + 1  
    else:  
        return rest
```

Award [2 max] if the function is not recursive.

Accept a recursive call in each branch of an if ... else (only one call is made each time), and a solution that works from the last character (desc[:-1] and desc[-1]).

12. (a) Award [5 max]

Award [1] for checking whether the head node matches name and, if so, moving self.head to the next node;
Award [1] for traversing the list using a temporary node (and a previous node) until the end of the list;
Award [1] for correctly comparing file_name with name;
Award [1] for linking the previous node to the node after the matching node;
Award [1] for stopping after the first match;
Award [1] for outputting a suitable message only when no node matches;

Award [3 max] if a match at the head is not handled. Award [4 max] if no message is output when no node matches.

Example answer

```
def remove_photo(self, name):
    removed = False
    if self.head.file_name == name:
        self.head = self.head.next
        removed = True
    else:
        previous = self.head
        current = self.head.next
        while current != None and removed == False:
            if current.file_name == name:
                previous.next = current.next
                removed = True
            else:
                previous = current
                current = current.next
        if removed == False:
            print (name, "not found")
```

(b) Award [5 max]

The algorithm will:

- check whether self.head is None / the list is empty;
- if it is, set self.head to new_photo and set the next pointer of new_photo to new_photo (itself);
- otherwise, make a temporary node equal to self.head;
- repeatedly move the temporary node to the next node;
- until the next pointer of the temporary node is equal to self.head / the last node is reached (not until None);
- set the next pointer of the last node to new_photo;
- set the next pointer of new_photo to self.head;

Award [3 max] if new_photo's next pointer is not set to self.head (the list is no longer circular). Award [4 max] if the empty list is not handled. Award [3 max] if the traversal stops when None is reached (a non-empty circular list contains no None pointer). Allow the use of diagrams to illustrate any of the above points.

Algorithm is:

1. If self.head is None, set self.head to new_photo and set the next pointer of new_photo to new_photo.
 2. Otherwise, set a temporary node, temp, equal to self.head.
 3. While the next pointer of temp is not equal to self.head, set temp to the next pointer of temp.
 4. Set the next pointer of temp to new_photo.
 5. Set the next pointer of new_photo to self.head.
-