

Markscheme

Specimen paper

Computer science

Higher level

Paper 2

This markscheme is **confidential** and for the exclusive use of examiners in this examination session.

It is the property of the International Baccalaureate and must **not** be reproduced or distributed to any other person without the authorization of the IB Global Centre, Cardiff.

General marking instructions

1. Follow the markscheme provided, award only whole marks and mark only in **RED**.
2. Make sure that the question you are about to mark is highlighted in the mark panel on the right-hand side of the screen.
3. 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.**
4. Sometimes, careful consideration is required to decide whether or not to award a mark. In these cases use RM™ Assessor annotations to support your decision. You are encouraged to write comments where it helps clarity, especially for re-marking purposes. Use a text box for these additional comments. It should be remembered that the script may be returned to the candidate.
5. Personal codes/notations are unacceptable.
6. Where an answer to a part question is worth no marks but the candidate has attempted the part question, enter a zero in the mark panel on the right-hand side of the screen. Where an answer to a part question is worth no marks because the candidate has not attempted the part question, enter an “NR” in the mark panel on the right-hand side of the screen.
7. Please ensure you check all scanned pages. The candidate may have answered more than one option.
8. **If a candidate has attempted more than one Option** within a paper, mark all the candidate’s work. RM™ Assessor will only award the marks for the higher scoring Option. Once all the work the candidate has attempted has been marked, please click “COMPLETE”; all the other questions from the other Options will auto complete to “NR” for “no response”.
9. Ensure that you have viewed **every** page including any additional sheets. Please ensure that you stamp “SEEN” on any page that contains no other annotation.
10. A mark should not be awarded where there is contradiction within an answer. Make a comment to this effect using a text box or the “CON” stamp.

Subject details: Computer Science HL Paper 2 markscheme

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 <i>etc.</i>

Option A — Java

1. (a) Award [1 max]

because it is in diagrammatic form;
it is program agnostic;
no complicated code / syntax to follow;

(b) Award [3]

correct values of i;
correct values of j;
correct output;

i	i < = N ?	j	j < = N ?	output
1	TRUE			1
		2	TRUE	2
		3	TRUE	3
		4	FALSE	
2	TRUE			2
		3	TRUE	3
		4	FALSE	
3	TRUE	4	FALSE	3
4	FALSE			

(c) Award [1]. Note: the algorithm shows a nested loop.

$O(n^2)$;

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

opening *employees* file for reading / default mode;
open *surnames* file for writing;
looping through *employees* file;
use of a counter or equivalent;
writing surnames only to *surnames* file;
closing both files;

Note: Award [1] for first 2 marking points if files are opened without indicating access modes.

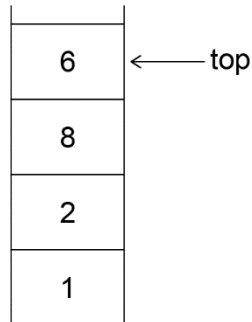
Algorithm is:

```
open employees file for reading / in default mode
open surnames file for writing
set counter to 0
repeat until end of employees file
    read next line from employees file
    if counter%3 = 0 *
        write line to surnames file
    increment counter
close both files
```

* there are alternatives ways of using counter

(e) Award [2].

correct elements in stack [1];
correct indication of top [1];



(f) Award [3] as follows:

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

repeat until isEmpty is true / while isEmpty is false;

pop an element and save its value/save in a variable/push it onto a new stack (stack2)

print the last value saved/print the variable/pop stack2 and print value

Note: Answer must demonstrate an algorithmic approach.

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

It is easier to work with one single structure than several individual ones;
Individual elements that have the same value can be quickly accessed via their indices;
Iteration through arrays is more efficient / Methods associated with arrays can make operations more efficient;

(b) *Award [4 max]*

repeat for each entry/index in the `Name` array
 identify the position of the space in each name
 save the substring to the left as `s1` and the substring to the right as `s2`
 save the new entry as `s2+space+s1`

(c) *Award [1]*

```
MAT[1][2];
```

(d) *Award [1] each;*

The value of `a` is: 6
The value of `b` is: 17

(e) **Award [5 max]** as follows:

Award [1] for initial variables correctly set;

Award [1] for both loops;

Award [1] for checking seat for False;

Award [1] for updating row value;

Award [1] for resetting tempTotal;

Award [1] for printing the correct row;

```
public void least(boolean[][]seats)
{
    int falseTotal = 0;
    int leastRow = 0;

    for (int x = 0; x < 40; x++)
    {
        int tempTotal = 0;
        for (int y = 0; y < 30; y++)
        {
            if (seats[x][y] == false)           // if(!seats[x][y] )
                tempTotal = tempTotal + 1;
        }
        if (tempTotal > falseTotal)
        {
            leastRow = x;
            falseTotal = tempTotal;
        }
    }
    System.out.println ("Row with least seats sold is " + leastRow);
}
```

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

The `(Passenger)` constructor is called;

A new object is created / memory is assigned for a new object ...;

... identified by the variable `p1`;

The `passengerName`, `email` and `ticketPrice` variables / all of the variables are initialized;

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

provides greater security for the values stored in these variables/for the code;

restricts access to such variables from outside of the class /only allows access through public / getter, setter methods;

(helps to) prevents accidental/unintended changes to the variables (from outside of the class);

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

composition;

The variable `ticketPrice` in the `Passenger` class depends upon the `Flight` class;

(c) **Award [4 max]**

Award [1] for calculating the % seats sold;

Award [1] for all 3 conditions correct;

Award [1] for if ... else structure correct;

Award [1] for return;

```
public double updatedTicketPrice()
{
    double price;
    double percentSold = (seatsBooked*100)/maxPassengers;
    if (percentSold >= 10 && percentSold <= 50)
        price = initialPrice*1.5;
    else
        if (percentSold > 50)
            price = initialPrice*3;
        else
            price = initialPrice;
    return price;
}
```

(d) **Award [4 max]**

Award [1] for calling `updatedTicketPrice()`;

Award [1] for creating new `Passenger` **object**;

Award [1] for updating `passengers` **array**;

Award [1] for incrementing `seatsBooked`;

```
public void addPassenger(String name, String email)
{
    double s = updatedTicketPrice();
    Passenger p = new Passenger(name, email, s);
    passengers[seatsBooked] = p;
    seatsBooked = seatsBooked + 1;
}
```

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

the name field of each `Passenger` object is used in the sorting process;

the array is divided into two parts: a sorted subset and an unsorted subset;

the smallest element is found in the unsorted part of the array and compared to the first (leftmost) element in the unsorted set;

if smaller, the two objects are swapped (and the first element in the unsorted set joins the sorted set);

the two steps above are repeated until the whole array is sorted / there are no more elements in the unsorted set;

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

The `HashMap` is set up using pairs of string values;

representing the airport code and airport name / diagram to show this;

when the airport code is used as a parameter for the `HashMap`, the matching/corresponding value of the airport name will be returned;

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

the arrays are combined to form a single set;

the set will automatically eliminate any duplicates / will only hold unique values;

(c) (i) **Award [1 max]**

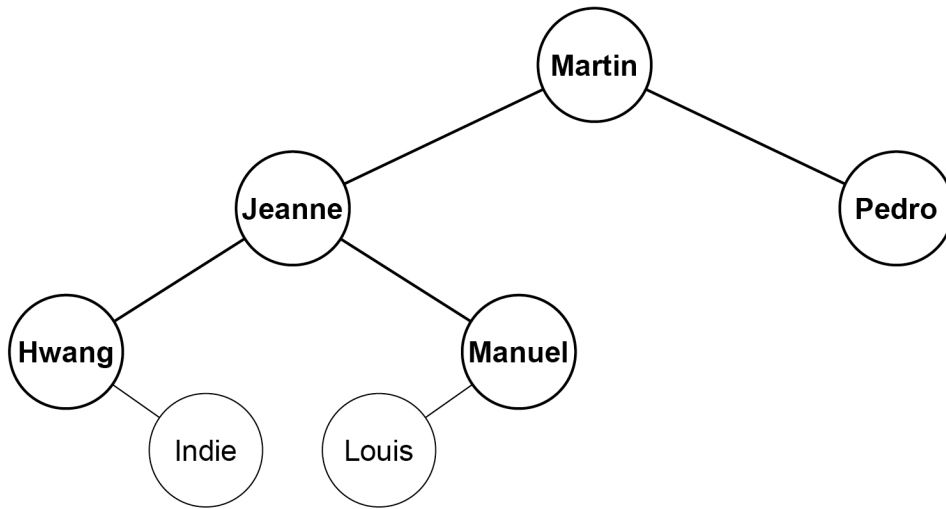
`false`;

(ii) **Award [1 max]**

`[2,6]`;

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

award [1] for each correct entry;



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

The insert method is initially passed the root of the tree as a parameter (where a comparison is made);
Then either the left or right subtree is selected (depending upon the result of the comparison);
The method is called again, using the root of this subtree as the parameter;
This is repeated until the insertion point is found (base case) / subtree is null;

(c) (i) **Award [1 max]**

21;

(ii) **Award [4 max]**

Award [1] for including a recursive call that will lead to the base case;
Award [1] for having a base case;
Award [1] for return + int;
Award [1] for a completely correct algorithm;

```
public int recur(int x)
{
    if (x != 0)
        return x + recur(x-1);
    else
        return x;
}
```

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

data is stored in the memory / on a stack every time a recursive call is made;
if the value of x is very large ...;
... the stack may overflow / system may run out of memory;

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

Award [1] for making use of a temporary node;
Award [1] for Boolean;
Award [1] for list traversal;
Award [1] for ending traversal if found;
Award [1] for correct comparison;
Award [1] for returning the correct values (in both cases);

```
public void findPlayer(String name)
{
    Node temp = head;
    boolean found = false;
    while(temp != null)
    {
        if (temp.name.compareTo(name) == 0) // allow = name
        {
            System.out.println(temp.score);
            found = true;
            break;
        }
        else
            temp = temp.next;
    }
    if (found == false)
        System.out.println("Not Found");
}
```

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

The algorithm will:

- make a temporary node equal to `head`;
- make use of a “previous” node;
- traverse the linked list (using the temporary node) comparing the score field of each node to the score field of the new node;
- until the insertion point is found;
- if insertion is at the beginning, link the new node to the head and make the head equal to the new node;
- if insertion is at the end, link the previous node to the new node;
- copy the *next* pointer of the node previous to the insertion point to the *next* pointer of the new node;
- link the previous node to the new node;

Note: Allow the use of diagrams to illustrate any of the above points.

Option B — Python

7. (a) Award [1 max]

because it is in diagrammatic form;
it is program agnostic;
no complicated code / syntax to follow;

(b) Award [3]

correct values of i;
correct values j;
correct output;

i	i <= N ?	j	j <= N ?	output
1	TRUE			1
		2	TRUE	2
		3	TRUE	3
		4	FALSE	
2	TRUE			2
		3	TRUE	3
		4	FALSE	
3	TRUE	4	FALSE	3
4	FALSE			

(c) Award [1]. Note: the algorithm shows a nested loop.

$O(n^2)$;

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

opening *employees* file for reading / default mode;
open *surnames* file for writing;
looping through *employees* file;
use of a counter or equivalent;
writing surnames only to *surnames* file;
closing both files;

Note: Award [1] for first 2 marking points if files are opened without indicating access modes

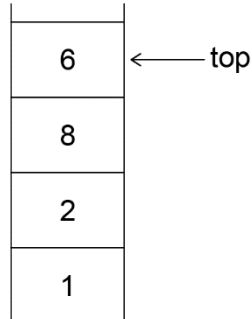
Algorithm is:

```
open employees file for reading / in default mode
open surnames file for writing
set counter to 0
repeat until end of employees file
    read next line from employees file
    if counter%3 = 0 *
        write line to surnames file
    increment counter
close both files
```

* there are alternatives ways of using counter

(e) *Award [2].*

correct elements in stack **[1]**;
correct indication of top **[1]**;



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

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

repeat until isEmpty is true / while isEmpty is false;

 pop an element and save its value/save in a variable/push it onto a new stack (stack2);

print the last value saved/print the variable/pop stack2 and print value;

Note: Answer must demonstrate an algorithmic approach.

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

It is easier to work with one single structure than several individual ones;
Individual elements that have the same value can be quickly accessed via their indices;
Iteration through arrays is more efficient / Methods associated with arrays can make operations more efficient;

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

repeat for each entry/index in the `Name` list
 identify the position of the space in each name
 save the substring to the left as `s1` and the substring to the right as `s2`
 save the new entry as `s2+space+s1`

repeat for each entry/index in the `Name` list
 use the `split()` function
 to separate the 2 strings into 2 list values (`s1` and `s2`)
 save the new entry as `s2+space+s1`

(c) **Award [1]**

```
MAT[1][2];
```

(d) **Award [1] for each:**

The value of `a` is: 6
The value of `b` is: 17

(e) **Award [5 max] as follows:**

Award [1] for initial variables correctly set;
Award [1] for both loops;
Award [1] for checking seat for `False`;
Award [1] for updating row value;
Award [1] for resetting `temp_total`;
Award [1] for printing the correct row;

```
def least(seats):  
  
    false_total = 0  
    least_row = 0  
  
    for x in range(40):  
        temp_total = 0  
        for y in range(30):  
            if seats[x][y] == False:  
                temp_total = temp_total + 1  
        if temp_total > false_total:  
            least_row = x  
            false_total = temp_total  
  
    print ("Row with least seats sold is ",least_row)
```

9. (a) (i) **Award [3 max]**

The `(Passenger)` constructor is called;

A new object is created / Memory is assigned for a new object ...;

... identified by the variable `p1`;

The `passenger_name`, `email` and `ticket_price` variables / all of the variables are initialized;

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

provides greater security for the values stored in these variables/for the code;

restricts access to such variables from outside of the class /only allows access through public / getter, setter methods;

(helps to) prevents accidental/unintended changes to the variables (from outside of the class);

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

composition;

The variable `ticket_price` in the `Passenger` class depends upon the `Flight` class;

(c) **Award [4 max]**

Award [1] for calculating the % seats sold;

(allow `len(self.passengers)` for `self.seats_booked`)

Award [1] for all 3 conditions correct;

Award [1] for if ... else structure correct;

Award [1] for return;

```
def updated_ticket_price(self):
    percent_sold = self.seats_booked * 100 / self.max_passengers
    if percent_sold >= 10 and percent_sold <= 50.0:
        price = self.initial_price * 1.5
    elif percent_sold > 50.0:
        price = self.initial_price * 3
    else:
        price = self.initial_price
    return price
```

(d) **Award [4 max]**

Award [1] for calling calling `updated_ticket_price()`;

Award [1] for creating new `Passenger` object;

Award [1] for updating `passengers` list;

Award [1] for incrementing `seats_booked`;

```
def add_passenger(self, name, email):  
    s = self.updated_ticket_price()  
    p = Passenger(name, email, s)  
    self.passengers.append(p)  
    self.seats_booked = self.seats_booked + 1
```

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

the name field of each `Passenger` object is used in the sorting process;

the list is divided into two parts: a sorted subset and an unsorted subset;

the smallest element is found in the unsorted part of the list and compared to the first (leftmost) element in the unsorted set;

if smaller, the 2 objects are swapped (and the first element in the unsorted set joins the sorted set);

the two steps above are repeated until the whole list is sorted / there are no more elements in the unsorted set;

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

The dictionary is set up using pairs of string values;
representing the airport code and airport name / diagram to show this;
when the airport code is used as a parameter for the dictionary, the matching/corresponding
value of the airport name will be returned;

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

the lists are combined to form a single set/HashSet;
the set/HashSet will automatically eliminate any duplicates / will only hold unique values;

(c) (i) **Award [1 max]**

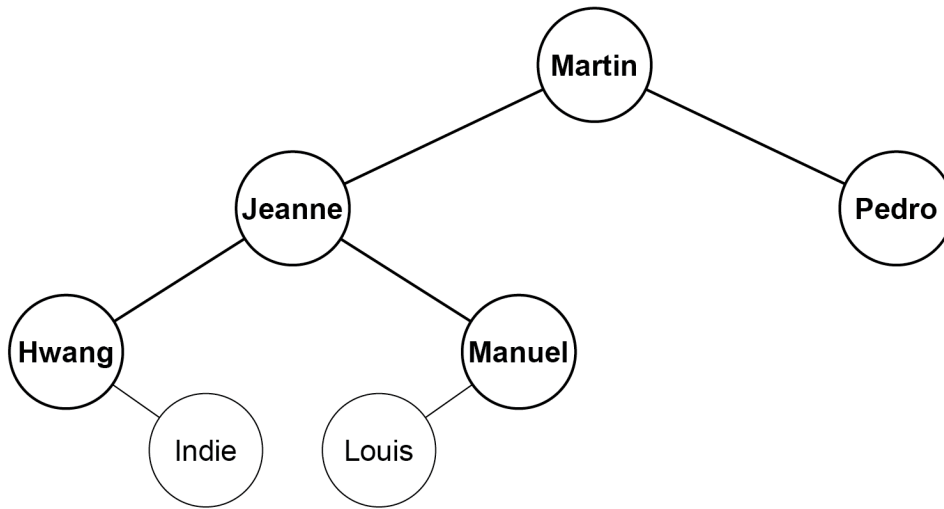
false;

(ii) **Award [1 max]**

{2, 6};

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

award [1] for each correct entry;



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

The insert method is initially passed the root of the tree as a parameter (where a comparison is made);
Then either the left or right subtree is selected (depending upon the result of the comparison);
The method is called again, using the root of this subtree as the parameter;
This is repeated until the insertion point is found (base case) / subtree is null;

(c) (i) **Award [1 max]**

21;

(ii) **Award [4 max]**

Award [1] for including a recursive call that will lead to the base case;
Award [1] for having a base case;
Award [1] for return;
Award [1] for a completely correct algorithm;

```
def recur(x):  
    if x != 0:  
        return x + recur(x-1);  
    else:  
        return x
```

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

data is stored each in the memory / on a stack every time a recursive call is made;
if the value of x is very large ...;
... the stack might overflow / system might run out of memory;

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

Award [1] for making use of a temporary node;
Award [1] for Boolean;
Award [1] for list traversal;
Award [1] for ending traversal if found;
Award [1] for correct comparison;
Award [1] for returning the correct values (in both cases);

```
def find_player(self, name):
    temp = self.head
    found = False
    while temp != None:
        if temp.name == name:
            found = True
            print (temp.score)
            break
        else:
            temp = temp.next
    if found == False:
        print ("Not found")
```

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

The algorithm will:

- make a temporary node equal to head;
- make use of a “previous” node;
- traverse the linked list (using the temporary node) comparing the score field of each node to the score field of the new node;
- until the insertion point is found;
- if insertion is at the beginning, link the new node to the head and make the head equal to the new node;
- if insertion is at the end, link the previous node to the new node;
- copy the *next* pointer of the node previous to the insertion point to the *next* pointer of the new node;
- link the previous node to the new node;

Note: Allow the use of diagrams to illustrate any of the above points
