Week 53

A-Level Computer Science

Homework bundle for this week

本周作业合集

exercises.lol

Contents 目录

19. Computational thinking and Problem-solving Starter Quiz · 课前小测

A-Level Computer Science Name: _____ Score: _____

1. A linear search:

   ☐ checks each element in turn and works on any list

   ☐ requires the data to be sorted

   ☐ always finds the target in one step

   ☐ only works on numbers

2. Bubble sort works by:

   ☐ repeatedly swapping adjacent out-of-order pairs

   ☐ inserting each element into a sorted prefix

   ☐ halving the list each time

   ☐ building a binary tree

3. Which Big-O describes binary search?

   ☐ O(log n)

   ☐ O(n)

   ☐ $O(n^2)$

   ☐ O(n log n)

4. Every recursive algorithm must have a base case because:

   ☐ without it the recursion never stops

   ☐ it makes the code shorter

   ☐ it speeds up the program

   ☐ the compiler requires a comment

5. Linear search is the better choice when the data is:

   ☐ unsorted or only a small list

   ☐ very large and sorted

   ☐ in a database index

   ☐ already a binary tree

6. Recursion is a natural fit for self-similar problems (trees, divide-and-conquer), but each call adds a stack frame — so without a base case it overflows the stack.

   ☐ True ☐ False

7. An O(n squared) algorithm takes 4 seconds on 1,000 items. Roughly how many seconds will it take on 2,000?

   _____

8. What must a recursive routine have to terminate? Select **all** that apply.

   ☐ a base case that is solved directly without another call

   ☐ an input that gets smaller on each recursive call

   ☐ a path that actually reaches the base case

   ☐ a loop inside the recursive case

   Tick every correct option.

9. About how many comparisons does a binary search need for one million sorted items?

   _____

10. What does Factorial(4) return?

    _____

19. Computational thinking and Problem-solving

Answers · 答案

A-Level Computer Science

1. **checks each element in turn and works on any list**

   Linear search needs no preparation and works on any list, at worst O(n).

2. **repeatedly swapping adjacent out-of-order pairs**

   Each pass swaps adjacent pairs, ”bubbling” the largest element to the end.

3. **O(log n)**

   Halving the range each step is logarithmic — O(log n).

4. **without it the recursion never stops**

   The base case is the condition that ends the chain of calls; without it the recursion runs forever.

5. **unsorted or only a small list**

   With no order to exploit (or a tiny list), linear search avoids the cost of sorting first.

6. **True**

   A simple counting loop is cleaner for plain iteration; recursion shines when the problem contains smaller copies of itself.

7. **16**

   Doubling n quadruples an O(n squared) time. The same doubling would take an O(n) algorithm from 4 seconds to 8.

8. **a base case that is solved directly without another call; an input that gets smaller on each recursive call; a path that actually reaches the base case**

   A base case alone is not enough: if the input never shrinks, the base case is never reached and frames pile up until the stack overflows.

9. **20**

   $\log_2(1\,000\,000) \approx 20$ — about 20 comparisons.

10. **24**

    $4 \times 3 \times 2 \times 1 = 24$.

A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

19.1 Algorithms

Name: _____ Class: _____ Date: _____ **Total: 31 marks**

KEY WORDS bubble sort 冒泡排序 • binary search 二分查找 • insertion sort 插入排序 • linear search 线性查找 • array 数组

Objective

Build the skills to answer exam questions on **algorithms** — linear and binary search, bubble and insertion sort, and comparing algorithms.

You must be able to:

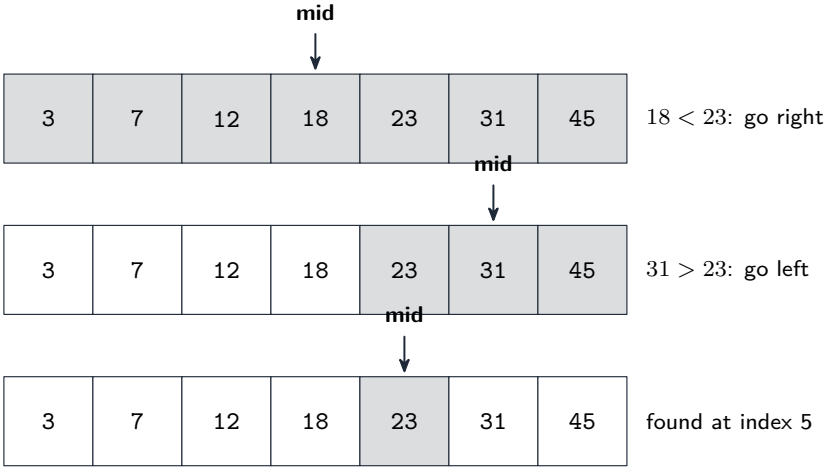
- trace a **linear** and a **binary** search
- trace a **bubble sort** and an **insertion sort**
- compare algorithms by **time** and **memory** (Big-O)

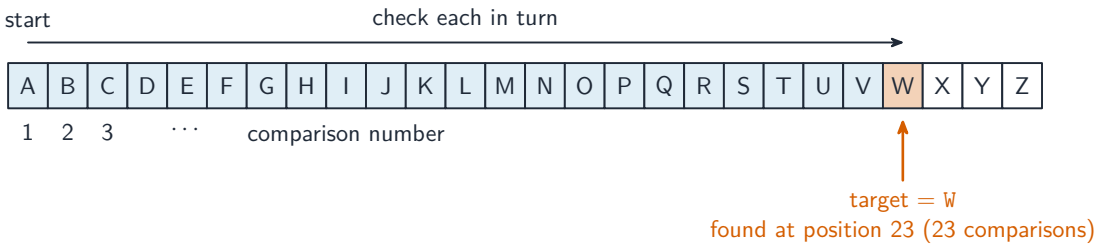
1 Worked examples

Study these first. Each one shows the method for a question type used later — follow the steps and you can do the Practice and Exam-style questions yourself.

■ Searching

A **linear search** checks each element in turn — works on **any** list, worst case **O(n)**. A **binary search** needs the data **sorted**: check the middle, then keep the half that could contain the target, halving the range each step — worst case **O(log n)**:





Linear search compared with binary search

Bubble sort

Repeatedly walk the array, **swapping** adjacent out-of-order pairs, so the largest "bubbles" to the end each pass:

```
FOR pass ← 1 TO n - 1
    FOR i ← 1 TO n - pass
        IF A[i] > A[i+1] THEN swap A[i], A[i+1]
    NEXT i
NEXT pass
```

Insertion sort

Build a sorted prefix; insert each new element by **shifting larger ones right**:

```
FOR i ← 2 TO n
    key ← A[i]
    j ← i - 1
    WHILE j >= 1 AND A[j] > key DO
        A[j+1] ← A[j]; j ← j - 1
    ENDWHILE
    A[j+1] ← key
NEXT i
```

Both are $\mathbf{O(n^2)}$ worst case; both are $\mathbf{O(n)}$ on already-sorted data (with an early exit / no shifts). Compare algorithms by **time taken** and **memory used**.

2 Practice

Now apply the methods above.

2.1 State the **precondition** that a binary search needs. [1]

2.2 State the **worst-case** time complexity of a **linear** search and a **binary** search. [2]

2.3 In a **bubble sort**, describe what happens during **one pass**. [1]

2.4 State **one** criterion used to compare two algorithms. [1]

3 Exam-style questions

3.1 Trace a **binary search** for `23` in `[3, 7, 12, 18, 23, 31, 45]` (indices 1–7). State the **index** examined at each step. [3]

3.2 The array `[5, 2, 8, 1]` is sorted with a **bubble sort**. Show the array **after the first complete pass**. [2]

3.3 State **two** advantages of a binary search over a linear search, and **one** disadvantage. [3]

3.4 State why an **insertion sort** is efficient on a list that is **already nearly sorted**. [1]

3.5 The algorithm below is intended to find the position of the first negative value in an array `Data` of 10 integers, or to output a message if there is none.

```
Found ← FALSE
Index ← 1
FOR Index ← 1 TO 10
    IF Data[Index] < 0
      THEN
        Found ← TRUE
        Position ← Index
    ENDIF
NEXT Index
IF Found = TRUE
  THEN OUTPUT Position
  ELSE OUTPUT "None found"
ENDIF
```

The array holds: `8, 3, -6, 11, -2, 7, 4, 9, 1, 5`.

(a) **Complete** the trace table by dry running the algorithm for the first **five** iterations. [4]

Index	`Data[Index]`	Found	Position

(b) **State** the value the algorithm outputs, and **explain** why it is **not** the position of the *first* negative value. [3]

(c) The outer loop structure is **not** the most appropriate one for this task. **Explain** why, and **name** the loop you would use instead. [3]

(d) **Write** corrected pseudocode using that loop, so the algorithm stops as soon as it finds the first negative value. [4]

(e) **Calculate** how many comparisons your corrected version makes on this data, and how many the original makes. **Describe** what this shows about the two loop choices when the target is near the start. [3]

Solutions

Each answer shows the steps, then the **result**.

2.1

- the data must be **sorted** (in order)

2.2

- linear search: **O(n)**
- binary search: **O(log n)**

2.3

- adjacent pairs are compared along the array and **swapped if out of order**, moving the largest remaining value to the end

2.4

- any one: **time taken** (number of operations); **memory used**

3.1

- mid = index **4** (value 18, $< 23 \rightarrow$ go right)
- mid = index **6** (value 31, $> 23 \rightarrow$ go left)
- mid = index **5** (value 23 $\rightarrow$ found)

3.2

- first pass: `5`↔`2` → `[2,5,8,1]`; `5,8` ok; `8`↔`1` → `[2,5,1,8]`
- **`[2, 5, 1, 8]`**

3.3

- advantages (any two): **far fewer comparisons** on large lists (O(log n) vs O(n)); much faster when searching repeatedly
- disadvantage: the data must be **sorted first** (a one-off cost)

3.4

- most elements are already in place, so the inner `WHILE` does **few or no shifts** — close to O(n)

3.5

(a) rows for `Index` 1 to 5: `Data[Index]` $= 8, 3, -6, 11, -2$

- `Found` stays `FALSE` until `Index` = 3, then `TRUE`
- `Position` is unset until `Index` = 3, when it becomes 3, then becomes 5 at `Index` = 5

(b) it outputs **5**

- the `FOR` loop always runs all ten times

- so every later negative value **overwrites** `Position`, leaving the position of the **last** one rather than the first

(c) a `FOR` loop is a **count-controlled** loop and cannot stop early, but here the number of iterations is not known in advance — it depends on the data

- a **condition-controlled** loop is appropriate: a `WHILE` loop

(d)

```
Found ← FALSE
Index ← 1
WHILE Index <= 10 AND Found = FALSE
    IF Data[Index] < 0
      THEN
        Found ← TRUE
        Position ← Index
      ELSE
        Index ← Index + 1
    ENDIF
ENDWHILE
IF Found = TRUE
  THEN OUTPUT Position
  ELSE OUTPUT "None found"
ENDIF
```

- `WHILE` with both conditions; the flag set and the index advanced correctly; the loop cannot run past element 10; output unchanged and correct

(e) the corrected version stops at the third element, so it makes **3** comparisons

- the original always makes **10**
- a condition-controlled loop can leave as soon as the answer is known, so on data where the target appears early it does a fraction of the work; a count-controlled loop pays the full cost every time

4 Study the algorithm:

```
DECLARE Chars : ARRAY[1:4] OF CHAR
DECLARE I, J : INTEGER
DECLARE Key : CHAR
I ← 2
Chars[1] ← 'D'
Chars[2] ← 'T'
Chars[3] ← 'H'
Chars[4] ← 'R'
WHILE I <= 4                              //Outer loop
   Key ← Chars[I]
   J ← I - 1
   WHILE J >= 0 AND Chars[J] > Key        //Inner loop
      Chars[J + 1] ← Chars[J]
      J ← J - 1
   ENDWHILE
   Chars[J + 1] ← Key
   I ← I + 1
ENDWHILE
```

(a) The outer loop structure used in the algorithm is **not** the most appropriate one to use.

State the type of loop structure that would be the most appropriate to use and justify why it is the most appropriate.

Loop structure .....

Justification .....

.....

[2]

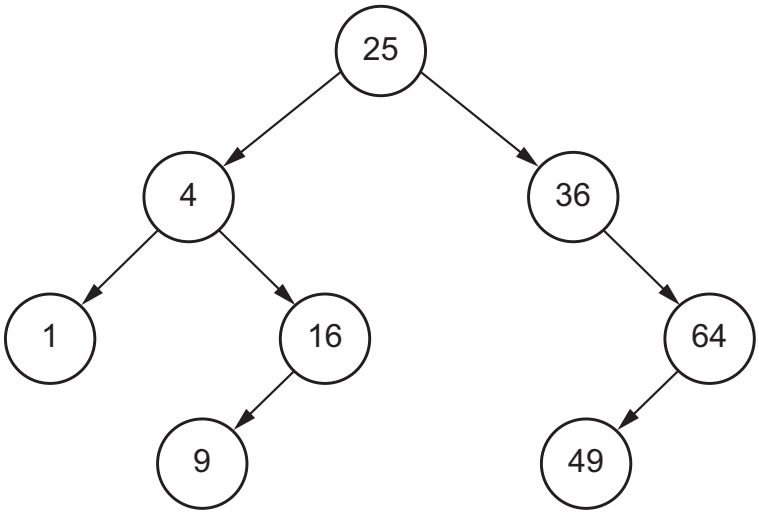
(b) Complete the trace table by dry running the algorithm.

The first row has been completed.

I	Key	J	Chars[J]	Chars			
				[1]	[2]	[3]	[4]
2				'D'	'T'	'H'	'R'

[6]

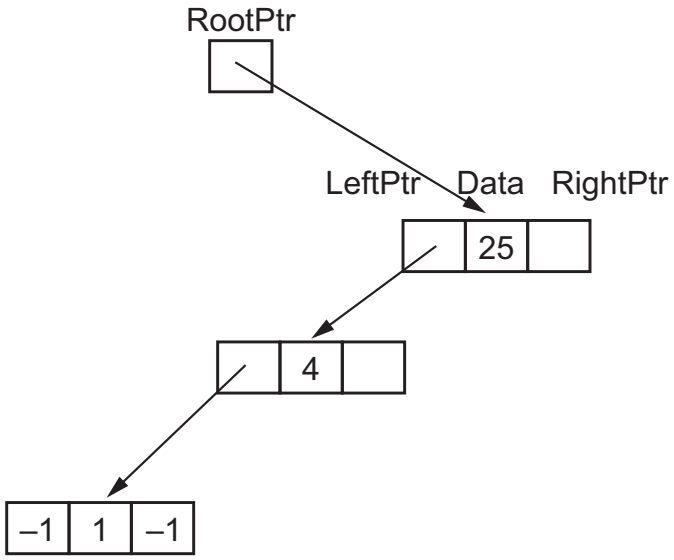
11 This binary tree shows an ordered list of integers.



(a) A linked list of nodes is used to store the data. Each node consists of a left pointer, the data and a right pointer.

−1 is used to represent a null pointer.

Complete this linked list to represent the given binary tree organisation.



[4]

(b) A 2D array is used to store the nodes of the linked list in part **(a)**.

Complete the diagram using your answer for part **(a)**.

RootPtr

0

FreePtr

Index	LeftPtr	Data	RightPtr
0		25	
1		4	
2		36	
3		1	
4		16	
5		64	
6		9	
7		49	
8			

[4]

(c) The linked list in part **(a)** is implemented using a 1D array of records. Each record contains a left pointer, data and a right pointer.

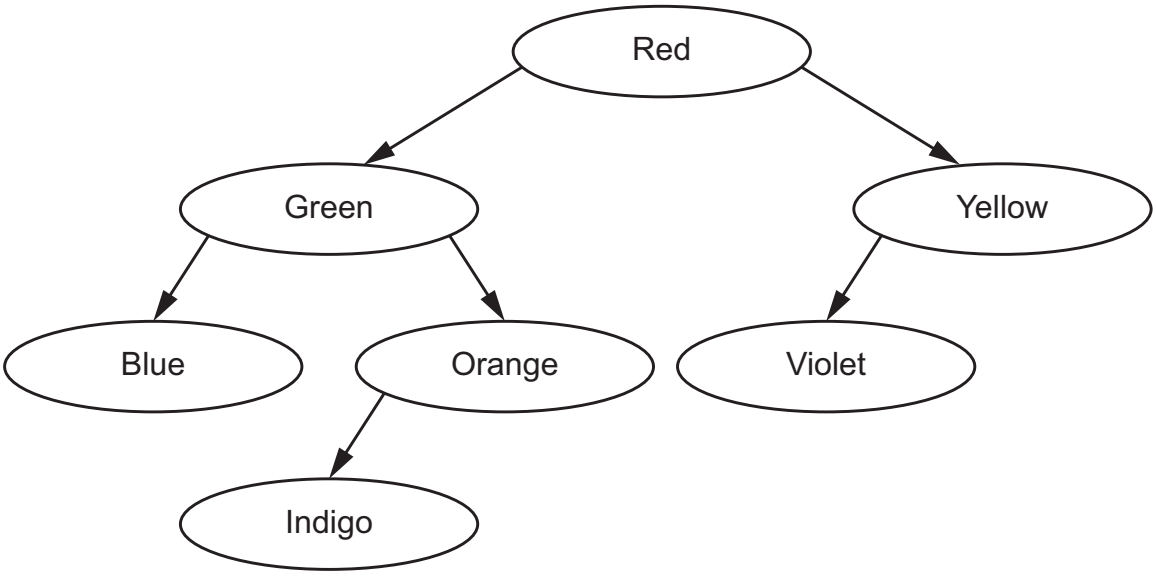
The following pseudocode represents a function that searches for an element in the array of records `LinkList`. It returns the index of the record if the element is found, or it returns a null pointer if the element is not found.

Complete the pseudocode for the function.

```
FUNCTION SearchList(Item : INTEGER).....
   NullPtr ← -1

   ..... ← RootPtr
   WHILE NowPtr <> NullPtr
      IF LinkList[NowPtr].Data < Item THEN
         NowPtr ← LinkList[NowPtr].RightPtr
      ELSE

         IF ..... THEN

            NowPtr ← .....
         ELSE
            RETURN NowPtr
         ENDIF
      ENDIF
   ENDWHILE
   RETURN NullPtr
ENDFUNCTION
```


[4]

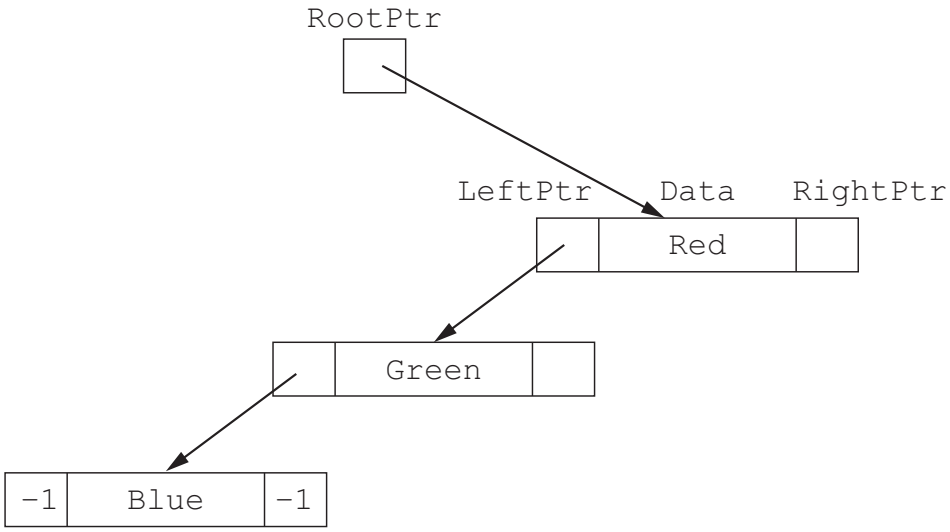
11 The following diagram shows an ordered binary tree.



(a) A linked list of nodes is used to store the data. Each node consists of a left pointer, the data and a right pointer.

–1 is used to represent a null pointer.

Complete this linked list to represent the given binary tree.



[4]

(b) A user-defined record structure is used to store the nodes of the linked list in part **(a)**.

Complete the diagram, using your answer for part **(a)**.

RootPtr

0

Index	LeftPtr	Data	RightPtr
0		Red	
1		Green	
2		Yellow	
3		Blue	
4		Orange	
5		Indigo	
6		Violet	
7			

FreePtr

[4]

(c) The linked list in part **(a)** is implemented using a 1D array of records. Each record contains a left pointer, data and a right pointer.

The following pseudocode represents a function that searches for an element in the array of records `BinTree`. It returns the index of the record if the element is found, or it returns a null pointer if the element is **not** found.

Complete the pseudocode for the function.

```
FUNCTION SearchTree(Item : STRING) .....

   NowPtr ← .....
   WHILE NowPtr <> -1

      IF ..... THEN
         NowPtr ← BinTree[NowPtr].LeftPtr
      ELSE
         IF BinTree[NowPtr].Data < Item THEN

            .....
         ELSE
            RETURN NowPtr
         ENDIF
      ENDIF
   ENDWHILE
   RETURN NowPtr
ENDFUNCTION
```


[4]

1 A program sorts string data using different sorting methods.

(a) The text file `Data.txt` stores string data items. Each data item is on a new line in the text file.

The function `ReadData()`:

- has a local array of strings that can store 45 items
- reads each line of data and stores it in the array
- returns the array.

Write program code for the function `ReadData()`.

Save your program as **Question1_N24**.

Copy and paste the program code into part **1(a)** in the evidence document.

[6]

(b) The function `FormatArray()` takes an array of strings as a parameter. It concatenates the contents of the array into one string with a space between each array element. The function returns the concatenated string.

(i) Write program code for `FormatArray()`.

Save your program.

Copy and paste the program code into part **1(b)(i)** in the evidence document.

[2]

(ii) The main program:

- calls `ReadData()` and stores the returned array
- calls `FormatArray()` with the returned array and outputs the returned string.

Write program code for the main program.

Save your program.

Copy and paste the program code into part **1(b)(ii)** in the evidence document.

[3]

(iii) Test your program.

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **1(b)(iii)** in the evidence document.

[1]

(c) The function `CompareStrings()`:

- takes two strings as parameters
- compares each string, one character at a time, to identify which string comes first alphabetically. If the first two characters are the same, the second character of each string is compared. This continues until the two characters are different.

The function:

- returns `1` if the first parameter comes before the second alphabetically
- returns `2` if the second parameter comes before the first alphabetically.

Write program code for `CompareStrings()`.

Assume that all strings are in lower case.

Assume that a difference between two strings will always be identified before the end of one string is reached.

Do **not** use an in-built string comparison function.

The strings must be compared one character at a time.

Save your program.

Copy and paste the program code into part **1(c)** in the evidence document.

[4]

(d) The function `Bubble()` takes an array of strings as a parameter and sorts the data into ascending alphabetical order, using a bubble sort. The bubble sort uses `CompareStrings()` to compare each string.

The function returns the sorted list.

(i) Write program code for `Bubble()`.

Save your program.

Copy and paste the program code into part **1(d)(i)** in the evidence document.

[3]

(ii) Write program code to amend the main program to:

- call `Bubble()` with the unsorted array as a parameter
- call `FormatArray()` with the sorted array and output the returned string.

Save your program.

Copy and paste the program code into part **1(d)(ii)** in the evidence document.

[2]

(iii) Test your program.

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **1(d)(iii)** in the evidence document.

[1]

1 A program sorts string data using different sorting methods.

(a) The text file `Data.txt` stores string data items. Each data item is on a new line in the text file.

The function `ReadData()`:

- has a local array of strings that can store 45 items
- reads each line of data and stores it in the array
- returns the array.

Write program code for the function `ReadData()`.

Save your program as **Question1_N24**.

Copy and paste the program code into part **1(a)** in the evidence document.

[6]

(b) The function `FormatArray()` takes an array of strings as a parameter. It concatenates the contents of the array into one string with a space between each array element. The function returns the concatenated string.

(i) Write program code for `FormatArray()`.

Save your program.

Copy and paste the program code into part **1(b)(i)** in the evidence document.

[2]

(ii) The main program:

- calls `ReadData()` and stores the returned array
- calls `FormatArray()` with the returned array and outputs the returned string.

Write program code for the main program.

Save your program.

Copy and paste the program code into part **1(b)(ii)** in the evidence document.

[3]

(iii) Test your program.

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **1(b)(iii)** in the evidence document.

[1]

(c) The function `CompareStrings()`:

- takes two strings as parameters
- compares each string, one character at a time, to identify which string comes first alphabetically. If the first two characters are the same, the second character of each string is compared. This continues until the two characters are different.

The function:

- returns `1` if the first parameter comes before the second alphabetically
- returns `2` if the second parameter comes before the first alphabetically.

Write program code for `CompareStrings()`.

Assume that all strings are in lower case.

Assume that a difference between two strings will always be identified before the end of one string is reached.

Do **not** use an in-built string comparison function.

The strings must be compared one character at a time.

Save your program.

Copy and paste the program code into part **1(c)** in the evidence document.

[4]

(d) The function `Bubble()` takes an array of strings as a parameter and sorts the data into ascending alphabetical order, using a bubble sort. The bubble sort uses `CompareStrings()` to compare each string.

The function returns the sorted list.

(i) Write program code for `Bubble()`.

Save your program.

Copy and paste the program code into part **1(d)(i)** in the evidence document.

[3]

(ii) Write program code to amend the main program to:

- call `Bubble()` with the unsorted array as a parameter
- call `FormatArray()` with the sorted array and output the returned string.

Save your program.

Copy and paste the program code into part **1(d)(ii)** in the evidence document.

[2]

(iii) Test your program.

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **1(d)(iii)** in the evidence document.

[1]

A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

19.2 Recursion

Name: _____ Class: _____ Date: _____ **Total: 17 marks**

KEY WORDS recursion 递归 • call stack 调用栈 • stack frame 栈帧 • recursive case 递归情形 • stack overflow 栈溢出

Objective

Build the skills to answer exam questions on **recursion** 递归 — base/recursive cases, tracing, and how the compiler uses the call stack.

You must be able to:

- explain **recursion** (base case + recursive case)
- **trace** a recursive function
- explain what the **compiler / call stack** does for each call

1 Worked examples

Study these first. Each one shows the method for a question type used later — follow the steps and you can do the Practice and Exam-style questions yourself.

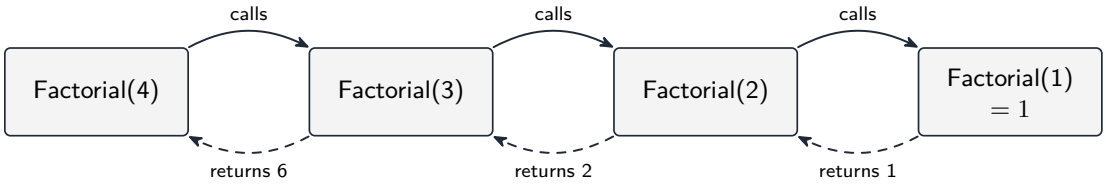
■ What recursion is

A **recursive** function calls **itself** with a smaller input, until a **base case** stops it:

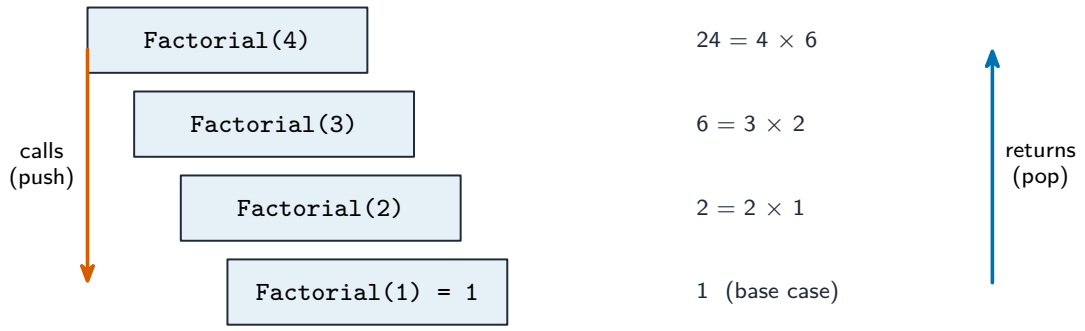
```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    IF n = 0 OR n = 1 THEN
        RETURN 1               // base case
    ELSE
        RETURN n * Factorial(n - 1)   // recursive case
    ENDIF
ENDFUNCTION
```

■ Tracing

`Factorial(4)` calls down to the base case, then the returns **unwind** back up:



result: 24



Recursion builds up a call stack

The call stack

The compiler gives every call its own **stack frame** holding its **parameters**, **local variables** and **return address**. On return, the value is passed back, the frame is **popped**, and execution resumes at the return address — so calls don't overwrite each other's variables. Missing the base case → **infinite recursion** → **stack overflow**.

Recursion vs iteration

Any recursive routine can be rewritten as an **iterative** one (a loop, plus your own stack if needed). Iteration usually uses **less memory** (no stack frames) and runs faster; recursion is often **shorter and clearer** for self-similar problems (trees, divide-and-conquer).

2 Practice

Now apply the methods above.

2.1 State what a **base case** is. [1]

2.2 State what the **recursive case** does. [1]

2.3 State what happens if a recursive function has **no base case**. [1]

2.4 State **two** things each **stack frame** holds. [2]

3 Exam-style questions

3.1 Describe what is meant by **recursion**. [2]

3.2 Trace `Factorial(4)`. State the value **returned** at each level of the recursion. [3]

3.3 Explain what happens on the **call stack** each time a recursive function is called. [3]

3.4 Give **one** feature that makes a problem suitable for recursion, and **one** example application. [2]

3.5 State **one** advantage and **one** disadvantage of writing a routine **recursively** instead of **iteratively**. [2]

Solutions

Each answer shows the steps, then the **result**.

2.1

- the **simplest case**, solved **directly without** another recursive call — it **stops** the recursion

2.2

- **reduces** the input and **calls the function again** (on the smaller problem)

2.3

- the recursion **never stops** — it runs until the **call stack overflows** and the program crashes

2.4

- any two: the **parameters**; the **local variables**; the **return address**

3.1

- a technique where a function **calls itself**
- with a **smaller version** of the problem until a **base case** is reached

3.2

- `Factorial(1)` returns **1**; `Factorial(2)` returns **2**; `Factorial(3)` returns **6**; `Factorial(4)` returns **24**

3.3

- a new **stack frame** is **pushed**, holding that call's parameters, local variables and return address
- the call runs, and on return its **value is passed back** and the frame is **popped**
- execution resumes at the **return address** in the caller

3.4

- feature: the problem is **self-similar** / can be broken into smaller versions of itself
- example: traversing a **tree**, binary search, merge sort, factorial/Fibonacci

3.5

- advantage: **shorter, clearer** code for self-similar problems
- disadvantage: it uses **more memory** (a stack frame per call) and can cause a **stack overflow**, and is often slower

11 Describe when the use of recursion would be beneficial **and** give an example.

Description .....

.....

.....

.....

Example .....

.....

[3]

12 An exception can occur when running a program.

(a) Explain what is meant by an exception.

.....

.....

.....

.....

.....

..... [3]

(b) Identify **one** example of an exception and give **one** reason why the exception may cause a problem.

Example .....

.....

Reason .....

.....

[2]

3 A program is written to perform different individual processes using arrays.

(a) A 1D array stores 10 integers.

(i) A recursive function `RecursiveCount()` takes three parameters:

- `ArrayCopy`, an array of integers
- `NumberElements`, the number of elements in the array of integers
- `DataToFind`, an integer data to find in the array.

The function counts and returns the number of times `DataToFind` is in `ArrayCopy`

The recursive algorithm:

- returns `0` if there are no elements in `ArrayCopy`
- compares the first element in `ArrayCopy` with `DataToFind`
  - if the element matches, the function returns `1` added to the return value from a recursive call (passing the array without the first element)
  - if the element does **not** match, the function returns the return value from a recursive call (passing the array without the first element).

Write program code for `RecursiveCount()`

Save your program as **Question3_N25**.

Copy and paste the program code into part **3(a)(i)** in the evidence document.

[6]

(ii) The main program stores the following data in a 1D array of integers in the order given:

```
0  5  1  2  5  9  9  6  5  0
```

The main program calls `RecursiveCount()` with the parameters:

- `0` as the data to find
- `10` as the number of elements
- the array of 10 integers.

The return value is output.

Write program code for the main program.

Save your program.

Copy and paste the program code into part **3(a)(ii)** in the evidence document.

[3]

(iii) Test your program.

Take a screenshot of the output(s).

Save your program.

Copy and paste the screenshot(s) into part **3(a)(iii)** in the evidence document.

[1]

(b) The program stores the following string. The string has four statements that are each terminated by a semi-colon `';'`

`"x=0;y=1;x=x+y;y++;"`

(i) Write program code to amend the main program to store the string `"x=0;y=1;x=x+y;y++;"` in a local variable.

Save your program.

Copy and paste the program code into part **3(b)(i)** in the evidence document.

[1]

(ii) The function `SplitData()` splits a string parameter into **four** individual lines of code.

The function creates an array of the strings where each line is stored in a new array element without the semi-colon.

The function returns the array of strings.

Do **not** use an inbuilt function to split the string.

Write program code for `SplitData()`

Save your program.

Copy and paste the program code into part **3(b)(ii)** in the evidence document.

[6]

(iii) The main program needs to call `SplitData()` with the string from part **3(b)(i)**. The main program then needs to output each element from the returned array on a new line.

Write program code to amend the main program.

Save your program.

Copy and paste the program code into part **3(b)(iii)** in the evidence document.

[2]

(iv) Test your program.

Take a screenshot of the output(s).

Save your program.

Copy and paste the screenshot(s) into part **3(b)(iv)** in the evidence document.

[1]

13 The recursive procedure `Delete()` is defined as follows:

```
PROCEDURE Delete(Index, Target)
   IF Numbers[Index] > 0 THEN
      IF Numbers[Index] >= Target THEN
         Numbers[Index] ⟵  Numbers[Index + 1]
      ENDIF
      Index ⟵ Index + 1
      CALL Delete(Index, Target)
   ENDIF
ENDPROCEDURE
```

An array `Numbers` is used to store a sorted data set of non-zero positive integers. Unused cells contain zero.

The contents of the array at the start of the algorithm are:

Numbers									
[1]	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**	**[10]**
2	3	7	11	15	17	19	23	0	0

Complete the trace table for the algorithm for the procedure call:

```
CALL Delete(1, 15)
```

		Numbers									
Index	**Target**	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**	**[10]**
		2	3	7	11	15	17	19	23	0	0

[4]