

Computational thinking and problem-solving

A2

Read online at pastopic.com/cambridge/computer-science-9618/notes/computational-thinking-and-problem-solving

Last checked 30 September 2026

Read first: Data types and structures

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

This is an A Level topic (Paper 3). By the end of it you should be able to:

1. **Linear and binary search:** write both in pseudocode, say what a binary search needs (the data must be sorted), and explain how its speed changes as the number of items grows.
2. **Insertion sort and bubble sort:** write both in pseudocode, and explain how a sort's speed depends on how many items there are and on how sorted they already are.
3. **Abstract Data Types (ADTs):** write algorithms to
 - **find** an item in a linked list and in a binary tree;
 - **insert** an item into a stack, a queue, a linked list and a binary tree;
 - **delete** an item from a stack, a queue and a linked list.
4. **Graphs:** know that a graph is an ADT, describe its key features and justify using one for a situation. You will not write code for a graph.
5. **Building one ADT from another:** describe a stack, a queue, a linked list, a dictionary and a binary tree, and show how each can be built from arrays, records or another ADT.
6. **Comparing algorithms:** compare algorithms that do the same job by the time they take and the memory they use, with **Big O notation** for time and space complexity.
7. **Recursion:** its essential features, how it is written in a programming language, writing and tracing recursive algorithms, and when recursion is a good choice.
8. **What a compiler does with recursion:** how it uses a stack, including winding and unwinding.

Understand it

Linear search

A **linear search** checks the items one by one from the start until it finds the target or reaches the end. It works on any list, sorted or not.

```

Found ← FALSE
Index ← 1
WHILE Found = FALSE AND Index ≤ MaxIndex
    IF List[Index] = Target THEN
        Found ← TRUE
    ELSE
        Index ← Index + 1
    ENDIF
ENDWHILE
IF Found = TRUE THEN
    OUTPUT "Found at position ", Index
ELSE
    OUTPUT "Not in the list"
ENDIF

```

With n items, the worst case (target last, or missing) takes n comparisons.

Binary search

A **binary search** only works on sorted data: the list must be sorted first. It looks at the middle item. If that is the target, it stops. If the target is bigger, the whole lower half (and the middle) can be thrown away; if smaller, the upper half. Each comparison **halves** what is left.

```

Lower ← 1
Upper ← MaxIndex
Found ← FALSE
WHILE Found = FALSE AND Lower ≤ Upper
    Mid ← (Lower + Upper) DIV 2
    IF List[Mid] = Target THEN
        Found ← TRUE
    ELSE
        IF List[Mid] < Target THEN
            Lower ← Mid + 1
        ELSE
            Upper ← Mid - 1
        ENDIF
    ENDIF
ENDWHILE

```

When Lower passes Upper there is nothing left to search, so the target is not there. Searching the sorted list 3, 8, 12, 17, 21, 26, 30, 35, 41, 47, 52, 58, 63, 70, 77 for 52:

Lower	Upper	Mid	List[Mid]	Next step
1	15	8	35	$35 < 52$, so $\text{Lower} \leftarrow 9$

Lower	Upper	Mid	List[Mid]	Next step
9	15	12	58	$58 > 52$, so Upper $\leftarrow$ 11
9	11	10	47	$47 < 52$, so Lower $\leftarrow$ 11
11	11	11	52	found

Four comparisons; a linear search would need eleven. In the worst case a binary search needs $\lfloor \log_2 n \rfloor + 1$ comparisons:

Items n	Linear search (worst)	Binary search (worst)
15	15	4
1000	1000	10
1 000 000	1 000 000	20

Doubling the number of items adds only **one** comparison to a binary search, but doubles the work of a linear search.

Big O notation

Big O notation describes how an algorithm's time (**time complexity**) or memory (**space complexity**) grows as the number of items n grows, usually in the worst case. Constants are ignored: only the shape of the growth matters.

Big O	Name	Example
$O(1)$	constant	push or pop on a stack; reading <code>List[5]</code>
$O(\log_2 n)$	logarithmic	binary search
$O(n)$	linear	linear search
$O(n^2)$	quadratic (polynomial)	bubble sort and insertion sort, worst case

$O(\log_2 n)$ means the time grows **logarithmically**: each doubling of n adds one step, so a list a thousand times longer needs only about ten more comparisons. $O(n)$ means the time grows in proportion to n . Bubble and insertion sort sort the array in place, so their extra memory (space complexity) is $O(1)$.

Insertion sort

An **insertion sort** builds a sorted section at the start of the array. It takes the next item (`Value`), shuffles every bigger item in the sorted section one place to the right, then drops `Value` into the gap.

```
FOR Index ← 2 TO MaxIndex
    Value ← List[Index]
    Position ← Index - 1
    WHILE Position >= 1 AND List[Position] > Value
```

```

    List[Position + 1] ← List[Position]
    Position ← Position - 1
ENDWHILE
List[Position + 1] ← Value
NEXT Index

```

Sorting 5, 2, 9, 1, 6:

After Index =	Array
2	2, 5, 9, 1, 6
3	2, 5, 9, 1, 6
4	1, 2, 5, 9, 6
5	1, 2, 5, 6, 9

For **descending** order, change `List[Position] > Value` to `List[Position] < Value`. If a second array holds matching data (names for scores), store and move its element alongside, at the same indexes.

Bubble sort

A **bubble sort** compares each pair of neighbours and swaps them if they are in the wrong order. After one pass the largest item has "bubbled" to the end. A flag records whether any swap happened; a pass with no swaps means the list is sorted, so the sort stops early.

```

Last ← MaxIndex - 1
REPEAT
    Swapped ← FALSE
    FOR Index ← 1 TO Last
        IF List[Index] > List[Index + 1] THEN
            Temp ← List[Index]
            List[Index] ← List[Index + 1]
            List[Index + 1] ← Temp
            Swapped ← TRUE
        ENDIF
    NEXT Index
    Last ← Last - 1
UNTIL Swapped = FALSE OR Last = 0

```

Sorting 5, 2, 9, 1, 6:

After pass	Array
1	2, 5, 1, 6, 9
2	2, 1, 5, 6, 9
3	1, 2, 5, 6, 9

After pass**Array**

4

1, 2, 5, 6, 9

Pass 4 makes no swaps, so it stops.

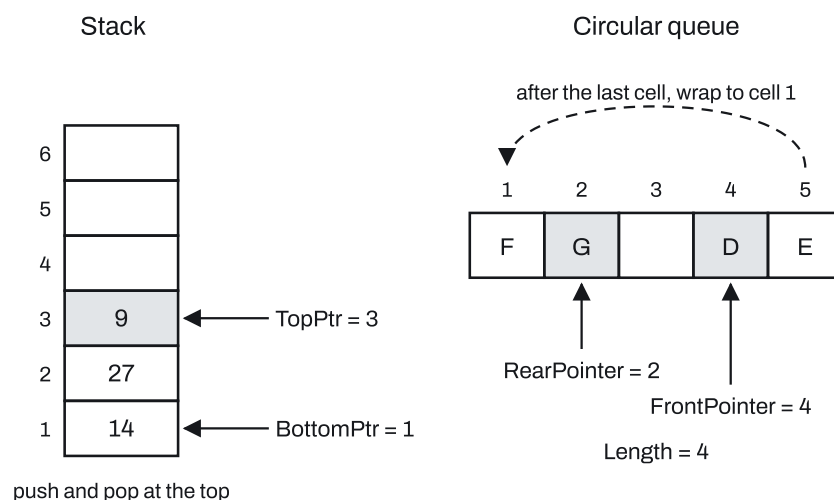
What affects a sort's performance: the **number of items** (more items, more comparisons) and the **initial order**. Data that is already partly sorted is sorted faster: with a sorted list both sorts make one pass of $n - 1$ comparisons, $O(n)$. Data in reverse order is the worst case, $O(n^2)$.

Abstract Data Types

An **Abstract Data Type (ADT)** is a collection of data together with the operations allowed on it, described without saying how it is stored. The ADTs you need are the **stack**, **queue**, **linked list**, **binary tree**, **dictionary** and **graph**. Each is built ("implemented") from built-in types such as arrays and records, or from another ADT.

Stack

A **stack** is **last in, first out (LIFO)**: items are added (**pushed**) and removed (**popped**) at the same end, the top. In an array it needs a base pointer, a top pointer and a maximum size.



Here 14, 27 and 9 were pushed in that order, so BottomPtr = 1 and TopPtr = 3; the next pop returns 9.

```

CONSTANT StackSize = 6
DECLARE Stack : ARRAY[1:StackSize] OF INTEGER
DECLARE BottomPtr, TopPtr : INTEGER

PROCEDURE Initialise()
    BottomPtr ← 1
    TopPtr ← 0           // one below the bottom: the stack is empty
ENDPROCEDURE

PROCEDURE Push(Value : INTEGER)

```

```

IF TopPtr = StackSize THEN
    OUTPUT "Stack full, ", Value, " not added"
ELSE
    TopPtr ← TopPtr + 1    // move up first...
    Stack[TopPtr] ← Value  // ...then store
ENDIF
ENDPROCEDURE

FUNCTION Pop() RETURNS INTEGER
    DECLARE TopValue : INTEGER
    IF TopPtr < BottomPtr THEN
        OUTPUT "Stack empty"
        TopValue ← -1
    ELSE
        TopValue ← Stack[TopPtr]    // read first...
        TopPtr ← TopPtr - 1         // ...then move down
    ENDIF
    RETURN TopValue
ENDFUNCTION

```

If the array starts at 0, the bottom is 0 and an empty stack has top -1 ; the "full" test becomes $\text{TopPtr} = \text{StackSize} - 1$. Match whatever the question sets up.

Queue

A **queue** is **first in, first out (FIFO)**: items join (**enqueue**) at the rear and leave (**dequeue**) from the front, so it needs two pointers and usually a Length count. In a **circular queue** a pointer that passes the last cell wraps back to the first, so freed cells are reused. In the diagram above, A, B, C, D and E were enqueued into cells 1 to 5, A, B and C were dequeued, then F and G were enqueued into cells 1 and 2: $\text{FrontPointer} = 4$, $\text{RearPointer} = 2$ and $\text{Length} = 4$.

```

PROCEDURE Initialise()
    FrontPointer ← 1
    RearPointer ← 0
    Length ← 0
ENDPROCEDURE

PROCEDURE Enqueue(NewItem : STRING)
    IF Length < MaxSize THEN
        RearPointer ← RearPointer + 1
        IF RearPointer > MaxSize THEN
            RearPointer ← 1          // wrap round
        ENDIF
        Queue[RearPointer] ← NewItem
    ENDIF

```

```

    Length ← Length + 1
ELSE
    OUTPUT "Queue full"
ENDIF
ENDPROCEDURE

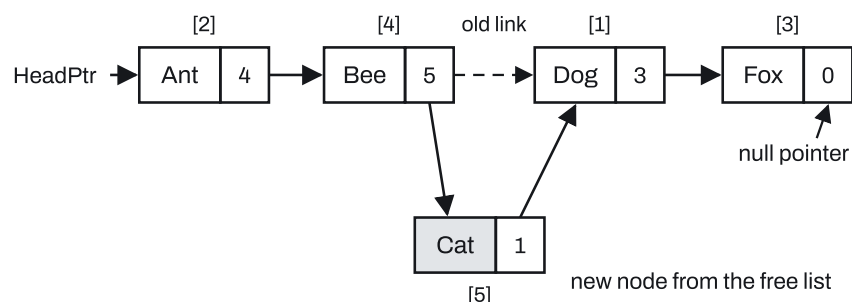
FUNCTION Dequeue() RETURNS STRING
    DECLARE Item : STRING
    Item ← ""
    IF Length > 0 THEN
        Item ← Queue[FrontPointer]
        FrontPointer ← FrontPointer + 1
        IF FrontPointer > MaxSize THEN
            FrontPointer ← 1          // wrap round
        ENDIF
        Length ← Length - 1
    ELSE
        OUTPUT "Queue empty"
    ENDIF
    RETURN Item
ENDFUNCTION

```

Stack or queue? Both vary in size. A stack has one accessible end and one moving pointer; a queue has two ends and two moving pointers. Use a queue when things must be dealt with in the order they arrive (print jobs, keyboard buffer); a stack when the most recent must come first (undo, recursion, evaluating Reverse Polish).

Linked list

A **linked list** stores items in any free cells and keeps them in order with **pointers**: each node holds its data and the index of the next node. A **head pointer** gives the first node, and a **null pointer** (0 or -1) marks the end. Unused nodes are chained into a **free list**, with a **free pointer** to its first node, so deleted space is reused.



Here the list is stored in two arrays, `Data` and `Next`, with 0 as the null pointer. At the start `HeadPtr = 2` (Ant) and `FreePtr = 5`.

Index	Data	Next at the start	Next after adding Cat	Next after deleting Dog
1	Dog	3	3	6
2	Ant	4	4	4
3	Fox	0	0	0
4	Bee	1	5	5
5	Cat	6	1	3
6		0	0	0

- **Find:** start at the head and follow the pointers until the item is found or the pointer is null.
- **Insert Cat:** take node 5 from the free list (FreePtr becomes 6), store Cat, walk the list to find where it goes (after Bee, before Dog), set Cat's pointer to Dog (1), then Bee's pointer to Cat (5). If it goes first, the head pointer changes instead.
- **Delete Dog:** find Dog and the node before it (Cat). Set Cat's pointer to Dog's pointer (3), so the list skips Dog. Then give Dog's node back to the free list: its pointer becomes the old free pointer (6) and FreePtr = 1.

```
// find: returns the index of the item, or 0 if it is not there
```

```
Pointer ← HeadPtr
```

```
WHILE Pointer <> 0 AND Data[Pointer] <> Item
```

```
    Pointer ← Next[Pointer]
```

```
ENDWHILE
```

```
// insert NewItem in order
```

```
IF FreePtr <> 0 THEN
```

```
    NewNode ← FreePtr
```

```
    FreePtr ← Next[FreePtr]
```

```
    Data[NewNode] ← NewItem
```

```
    Previous ← 0
```

```
    Current ← HeadPtr
```

```
    WHILE Current <> 0 AND Data[Current] < NewItem
```

```
        Previous ← Current
```

```
        Current ← Next[Current]
```

```
    ENDWHILE
```

```
    Next[NewNode] ← Current
```

```
    IF Previous = 0 THEN
```

```
        HeadPtr ← NewNode
```

```
    ELSE
```

```
        Next[Previous] ← NewNode
```

```
    ENDIF
```

```
ENDIF
```

```

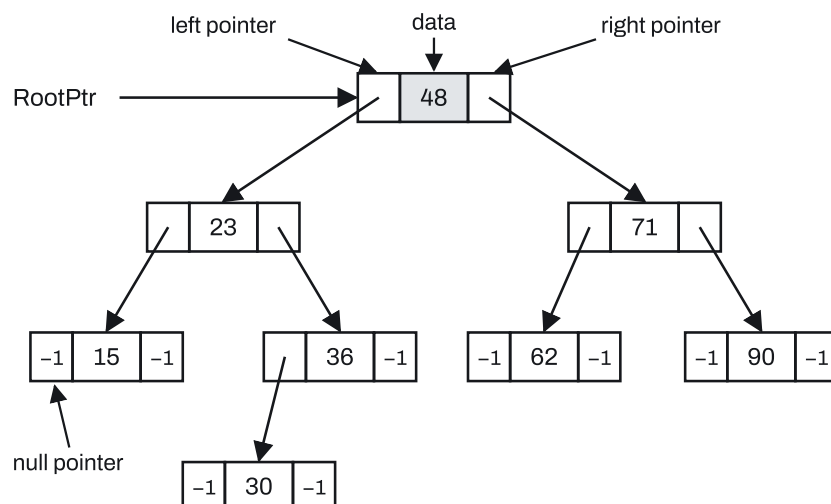
// delete Item
Previous ← 0
Current ← HeadPtr
WHILE Current <> 0 AND Data[Current] <> Item
    Previous ← Current
    Current ← Next[Current]
ENDWHILE
IF Current <> 0 THEN
    IF Previous = 0 THEN
        HeadPtr ← Next[Current]
    ELSE
        Next[Previous] ← Next[Current]
    ENDIF
    Next[Current] ← FreePtr
    FreePtr ← Current
ENDIF

```

A linked list is **dynamic**: it can grow and shrink, and inserting or deleting only changes pointers instead of moving data. An array is **static**: fixed in size, so a stack or queue built on it cannot grow once full.

Binary tree

An ordered **binary tree** is made of nodes, each with a **left pointer**, **data** and a **right pointer**. The first item is the **root**. Each new item starts at the root and goes **left if it is smaller, right if it is larger** (alphabetical order for strings), until it reaches an empty pointer, where it is attached. Every pointer that leads nowhere holds the null value.



Stored as an array of records, each node takes the next free row, in the order the items were added:

Index	LeftPtr	Data	RightPtr
0	1	48	2
1	3	23	4
2	5	71	6
3	-1	15	-1
4	7	36	-1
5	-1	62	-1
6	-1	90	-1
7	-1	30	-1
8			
9			

RootPtr is 0, and the free pointer is 8, the next empty row. To **find** an item, start at the root and go left or right in the same way: 30 is found by $48 \rightarrow 23 \rightarrow 36 \rightarrow 30$; searching for 65 visits $48 \rightarrow 71 \rightarrow 62$, then meets a null pointer, so 65 is not there.

```

FUNCTION FindNode(Target : INTEGER) RETURNS INTEGER
    DECLARE Here : INTEGER
    DECLARE Found : BOOLEAN
    Here ← RootPtr
    Found ← FALSE
    WHILE Found = FALSE AND Here <> -1
        IF Tree[Here].Data = Target THEN
            Found ← TRUE                                // stop: Here is the node
        ELSE
            IF Target < Tree[Here].Data THEN
                Here ← Tree[Here].LeftPtr              // smaller: go left
            ELSE
                Here ← Tree[Here].RightPtr              // larger: go right
            ENDIF
        ENDIF
    ENDWHILE
    RETURN Here                                         // its index, or -1 if not there
ENDFUNCTION

```

```

PROCEDURE AddNode(NewItem : INTEGER)
    DECLARE NewPtr, Here, Parent : INTEGER
    DECLARE GoLeft : BOOLEAN
    IF FreePtr <= MaxIndex THEN
        NewPtr ← FreePtr
    ENDIF

```

```

FreePtr ← FreePtr + 1
Tree[NewPtr].Data ← NewItem
Tree[NewPtr].LeftPtr ← -1
Tree[NewPtr].RightPtr ← -1
IF RootPtr = -1 THEN
    RootPtr ← NewPtr
ELSE
    Here ← RootPtr
    WHILE Here <> -1
        Parent ← Here
        IF NewItem < Tree[Here].Data THEN
            GoLeft ← TRUE
            Here ← Tree[Here].LeftPtr
        ELSE
            GoLeft ← FALSE
            Here ← Tree[Here].RightPtr
        ENDIF
    ENDWHILE
    IF GoLeft = TRUE THEN
        Tree[Parent].LeftPtr ← NewPtr
    ELSE
        Tree[Parent].RightPtr ← NewPtr
    ENDIF
ENDIF
ENDIF
ENDPROCEDURE

```

Dictionary and graph

A **dictionary** stores **key–value pairs** (a word and its meaning, a student ID and a name): you look a value up by its key, and each key appears once. Build it from an array of records (a key field and a value field), searched by key, or from two parallel arrays; a hash table makes look-up fast.

A **graph** is a set of **nodes (vertices)** joined by **edges**. Edges can be **directed** (one way) or **undirected**, and **weighted** (a distance, time or cost) or not. Unlike a tree, a graph can have cycles and any node can connect to any other. Use a graph for anything that is a **network of relationships**: roads between towns, links between web pages, friends on social media, cables between routers, so that routes and connections can be found.

Building one ADT from another

- **Stack or queue** from an array plus pointer variables (above), or from a linked list, so it can grow.
- **Linked list** and **binary tree** from an array of records (or parallel arrays) with integer pointers.

- **Queue from two stacks:** stack 1 holds the queue with the front item on top. To add an item: pop everything from stack 1 and push it onto stack 2 (this reverses the order), push the new item onto stack 2, then pop everything from stack 2 back onto stack 1. The new item is now at the bottom, the back of the queue, and popping stack 1 always removes the oldest item.

Recursion

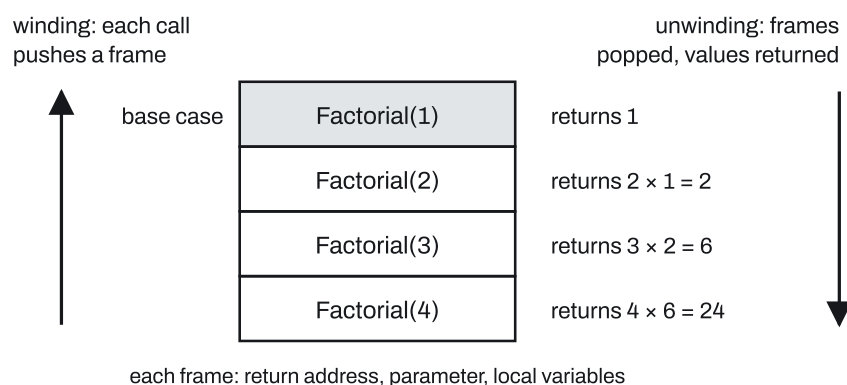
A **recursive** subroutine is one defined in terms of itself: it **calls itself**. Every recursive algorithm has three **essential features**:

1. a **general case**, where it calls itself with a smaller or simpler version of the problem;
2. a **base case** (stopping condition), which returns an answer without another recursive call;
3. each call changes the state so that it **moves towards the base case**.

In a programming language it is written as a function or procedure that contains a call to itself, usually inside an IF that checks for the base case first:

```
FUNCTION Factorial(N : INTEGER) RETURNS INTEGER
  IF N <= 1 THEN
    RETURN 1                                // base case
  ELSE
    RETURN N * Factorial(N - 1)              // general case
  ENDIF
ENDFUNCTION
```

`Factorial(4)` calls `Factorial(3)`, which calls `Factorial(2)`, which calls `Factorial(1)`, the base case. Nothing is multiplied until the base case returns 1. Then the waiting calls finish in reverse order: $2 \times 1 = 2$, then $3 \times 2 = 6$, then $4 \times 6 = 24$.



When recursion helps: when a problem breaks down naturally into smaller versions of the same problem, especially one with many branches: factorials, number series such as Fibonacci, traversing a tree, evaluating a Reverse Polish expression, divide-and-conquer sorts. The code is shorter and closer to the definition. The

cost is memory: every call adds a stack frame, so very deep recursion is slower than a loop and can overflow the stack.

How the compiler runs recursion: a stack

A stack suits recursion because it is **last in, first out**, and the last call made is the first one to finish. The compiler produces object code that:

- on **each recursive call**, pushes a **stack frame** holding the **return address**, the parameter values and the local variables (this is **winding**);
- when the **base case** is reached, starts to **unwind**: each call finishes, its frame is **popped**, and its values and return address are restored, in the **reverse order** to which they were pushed, until the first call returns.

Key facts and formulas

There is no formula list for 9618: learn everything here.

Formula	Status
Linear search: worst case n comparisons, $O(n)$	Learn it
Binary search: worst case $\lfloor \log_2 n \rfloor + 1$ comparisons, $O(\log_2 n)$; data must be sorted	Learn it
Bubble sort and insertion sort: $O(n^2)$ worst case, $O(n)$ if already sorted, space $O(1)$	Learn it
Stack push and pop, queue enqueue and dequeue: $O(1)$	Learn it
Middle index: $\text{Mid} \leftarrow (\text{Lower} + \text{Upper}) \text{ DIV } 2$	Learn it
Circular wrap: IF $\text{Pointer} > \text{MaxSize}$ THEN $\text{Pointer} \leftarrow 1$ (or 0 for a 0-based array)	Learn it
Stack: LIFO, one end, one moving pointer (Top)	Learn it
Queue: FIFO, two ends, two moving pointers (Front, Rear)	Learn it
Recursion: general case, base case, moves towards the base case	Learn it

How to do it

In the Paper 3 questions we have tagged for this topic (31 questions, 2021–2025), the types came up this often. 27 of the 31 questions mix types, so the counts add up to more than 31.

Question type	Questions
Write or complete stack pseudocode	8
Describe recursion and when it helps	8
Search performance and Big O	6
Add items to a binary tree diagram	5

Question type	Questions
Why a stack is used for recursion	4
Complete a binary search	4
Write an insertion sort	4
What affects a sort's performance	4
Compare or justify ADTs	4
Identify ADTs	4
Write or complete queue pseudocode	3
Trace a recursive algorithm	3
Binary search: condition and steps	3
Binary tree as an array of records	3
Search a binary tree	3
Complete a linear search	2
Implement one ADT from another	2
Search and update a linked list in arrays	1

1. Write or complete stack pseudocode (8 questions)

1. **Read the set-up first:** the array's bounds, the starting values of the pointers and the constant's name. Use exactly those identifiers.
2. **Declarations** (a mark each): `CONSTANT MaxSize = 25`, the two pointers as `INTEGER`, and the array with the right bounds **and** type, such as `DECLARE Stack : ARRAY[1:25] OF REAL`.
3. **Initialise:** base at the first index; top one below it (0 for a 1-based array, -1 for 0-based).
4. **Push:** check not full, add 1 to the top pointer, **then** store the item; otherwise an error message.
5. **Pop:** check not empty (top pointer at or above the bottom), read the top item, **then** take 1 off the top pointer, and `RETURN` it. A function header needs `RETURNS <type>`.

Variations you will meet:

- Using the result: `INPUT MyItem` then `CALL Push(MyItem)`, or `OUTPUT "The item removed is "`, `Pop()`.
- A given helper, such as `StackFull()`: call it (`IF StackFull() = FALSE THEN ...`) instead of rewriting its test, and `RETURN` a message if the function is meant to return one, rather than outputting it.
- Filling in a table of identifiers with their data types and descriptions (pointers are `INTEGER`; the array's type is the data's type).

2. Describe recursion and when it helps (8 questions)

1. "Describe recursion": a subroutine that **calls itself** (the general case) **until a base case** is reached, where no further call is made.
2. "State the three essential features": general case (calls itself), base case (stopping condition), and moving towards the base case each call.
3. "When is it beneficial": the problem **breaks down into smaller versions of itself** (often with many branches), plus an **example**: factorial, Fibonacci, tree traversal, evaluating Reverse Polish.

If asked for another ADT that suits recursive algorithms (besides a binary tree), the answer is a **stack**.

3. Search performance and Big O (6 questions)

- **Binary search alone**: Big O is $O(\log_2 n)$; Big O describes time (or space) complexity; the time grows **logarithmically**: each doubling of n adds one step; because each step halves the search area, it needs few comparisons even for huge lists.
- **Compare with linear**: linear checks each item **in turn from the start** until found or the end is reached, $O(n)$, and works on unsorted data; binary starts **in the middle**, discards the half without the item, needs **sorted** data and usually makes far fewer comparisons, $O(\log_2 n)$.
- **"How does performance vary with the number of items"**: more items, more time; for binary search only a little more (one extra comparison per doubling).

If the question asks only about $O(\log n)$, describe that; don't drift into a comparison.

4. Add items to a binary tree diagram (5 questions)

1. For each new item **in the order given**, start at the root: smaller goes left, larger goes right, repeat until an empty pointer, then attach it there.
2. Draw the node as **left pointer | data | right pointer**, with the arrow coming **from the pointer box** (not the middle) to the child.
3. Put the null value the question uses (0 or -1) in **every** empty pointer box, including the new leaves; put nothing extra anywhere.

5. Why a stack is used for recursion (4 questions)

Four points earn the marks: a stack is **LIFO**; **each recursive call pushes** the return address and local variables onto the stack; when the **base case** is reached it **unwinds**; the values are **popped in reverse order** to how they were pushed. For "what a compiler does", start with "the compiler produces object code to push ... with each call ... and pop ... after the base case". Saying only "a stack is LIFO like recursion" gets one mark at most.

6. Complete a binary search (4 questions)

Gap-fill, a mark per line. Check each of these:

1. Set `Lower` to the first index, `Upper` to the last, and the found flag to `FALSE`.
2. The loop runs while the item is not found **and** there is still something to search (`Lower <= Upper`). Some versions use a second "not in list" flag or a `REPEAT ... UNTIL` loop instead; complete whichever the question sets up.
3. `Mid ← (Lower + Upper) DIV 2`, then compare the item at `Mid` with the target; equal means found.
4. Not found once the bounds cross (`Lower > Upper`); a version that compares before moving the bounds stops when they are equal and the item at `Mid` still doesn't match.
5. `Lower ← Mid + 1` when the middle is too small; `Upper ← Mid - 1` when too big.

7. Write an insertion sort (4 questions)

Write it from the pattern in "Understand it", using the question's identifiers. The marks go to: storing `Value` before the inner loop; an inner loop that runs while `Position >= 1` and the item is bigger than `Value`; moving the item up one place; updating `Position`; and putting `Value` back **after** the inner loop, at `Position + 1`. For descending order swap `>` for `<`; if a second array holds matching data, save its element too and move it in step.

8. What affects a sort's performance (4 questions)

Two points: the **number of items** (more items take longer), and the **initial order** (data that is already partly sorted is sorted faster; data in reverse order is slowest).

9. Compare or justify ADTs (4 questions)

- **Stack vs queue:** LIFO vs FIFO; stack has one accessible end and one moving pointer, queue has two of each; both can vary in size.
- **Linked list vs array for a stack:** a linked list is dynamic, so it can grow and shrink and edits only change pointers; an array is static and fixed in size, so a full stack cannot grow.
- **Queue for print jobs:** jobs should print in the order they are sent, which a queue does; a stack would print the first job sent last.

10. Identify ADTs (4 questions)

Name ones **other than** the ADT in the question: stack, queue, linked list, binary tree, graph, dictionary.

11. Write or complete queue pseudocode (3 questions)

1. Declarations: the constant, the array (right bounds and type), and three integers `FrontPointer`, `RearPointer`, `Length`.
2. **Enqueue:** IF `Length < MaxSize`, add 1 to the rear, wrap it if it passes the end, store, add 1 to `Length`.
3. **Dequeue:** IF `Length > 0`, take `Queue[FrontPointer]`, add 1 to the front, wrap it if it passes the end, take 1 off `Length`; else an error message; RETURN the item.

4. Initialise to match the array: for $[0:\text{MaxSize} - 1]$, front and rear -1 (or front 0, rear -1) and Length $\leftarrow 0$. Wrap to 0, not 1.

12. Trace a recursive algorithm (3 questions)

1. Write one row per call **on the way down (winding)**, with the parameter values and the expression still waiting, such as $4 + \text{Fib}(3)$.
2. At the base case, write its return value.
3. Then fill rows **on the way back up (unwinding)**, in reverse order, each using the value just returned.
4. If the procedure changes an array, record only the cells that change, in the order they change, and leave the others blank.

13. Binary search: condition and steps (3 questions)

Condition: the data must be **sorted**. Steps: find the middle item; compare it with the target; if equal, found; if not, discard the half that cannot contain the target; repeat on the remaining half until found, or until the lower bound passes the upper bound (not found).

14. Binary tree as an array of records (3 questions)

Each node's row gets the **indexes** of its left and right children, or -1 . Nodes take rows in the order they were added. The free pointer is the **next empty row**, which is left blank.

15. Search a binary tree (3 questions)

Follow FindNode above: a function header with RETURNS INTEGER; start at the root (Here $\leftarrow \text{RootPtr}$); keep going while the item isn't found and the pointer isn't null (Here $<> -1$); if the node holds the item, stop; if the item is smaller, follow LeftPtr, otherwise RightPtr; finally return the index, which is -1 if the search fell off the tree. Use the question's own identifiers.

16. Complete a linear search (2 questions)

Set Found $\leftarrow \text{FALSE}$ before the loop; loop while not found and not past the end; compare the current item with the target; add 1 to the index; set "not in list" once the index passes the end. Give the target variable the same data type as the array.

17. Implement one ADT from another (2 questions)

The queue from two stacks, as in "Understand it": the second stack exists to **reverse the order** of the first. To add: pop all of stack 1 onto stack 2, push the new item, pop all of stack 2 back onto stack 1.

18. Search and update a linked list in arrays (1 question)

- **After deletions:** point the head at the first remaining item and set each remaining node's pointer to the next remaining one in order; the last gets the null value.
- **Search:** `Pointer ← HeadPointer`, loop `WHILE Pointer <> 0`, compare, else `Pointer ← NextPointer[Pointer]`; output "not found" without using the array (the item isn't in it).
- **Improve the structure:** add a **free list** and free pointer so deleted nodes are reused.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.
-

Example 1

The array `Code[ 1 : 12 ]` holds 4, 9, 13, 18, 22, 27, 31, 36, 40, 45, 51, 57. It is searched with the binary search in "Understand it".

- (a) State the condition that makes a binary search possible on this array. **[1]**
- (b) Trace the search for 45, and then for 20. **[4]**
- (c) A sorted array holds 4000 items. State the largest number of comparisons a binary search and a linear search would need, and describe the performance of the binary search using Big O. **[3]**
- (a) The data must be sorted (it is in ascending order). **B1**

(b) Searching for 45:

Lower	Upper	Mid	Code[Mid]	Next step
1	12	6	27	$27 < 45$, so <code>Lower ← 7</code>
7	12	9	40	$40 < 45$, so <code>Lower ← 10</code>
10	12	11	51	$51 > 45$, so <code>Upper ← 10</code>
10	10	10	45	found

B1 for the Mid values, **B1** for found at position 10.

Searching for 20:

Lower	Upper	Mid	Code[Mid]	Next step
1	12	6	27	$27 > 20$, so $\text{Upper} \leftarrow 5$
1	5	3	13	$13 < 20$, so $\text{Lower} \leftarrow 4$
4	5	4	18	$18 < 20$, so $\text{Lower} \leftarrow 5$
5	5	5	22	$22 > 20$, so $\text{Upper} \leftarrow 4$
5	4			$\text{Lower} > \text{Upper}$: not found

B1 for the bounds, **B1** for stopping when $\text{Lower} > \text{Upper}$ and reporting not found.

(c) Binary: $\lfloor \log_2 4000 \rfloor + 1 = 11 + 1 = 12$ comparisons; linear: 4000. **B1**

A binary search is $O(\log_2 n)$: **B1**. Each comparison halves the items left, so the time grows logarithmically: doubling the items adds only one comparison. **B1**

Example 2

A circular queue of strings is held in `Queue[0:5]` (`MaxSize = 6`) with `FrontPointer`, `RearPointer` and `Length`. `Initialise` sets them to 0, -1 and 0.

(a) Write pseudocode for `Enqueue`. **[4]**

(b) P, Q, R, S and T are enqueued, three items are dequeued, then U, V and W are enqueued. Give the contents and the pointers after each step. **[3]**

(c) How many more items can be enqueued now, and what does the next `Dequeue` return? **[2]**

(a)

```

PROCEDURE Enqueue(NewItem : STRING)
    IF Length < MaxSize THEN
        RearPointer ← RearPointer + 1
        IF RearPointer > MaxSize - 1 THEN
            RearPointer ← 0
        ENDIF
        Queue[RearPointer] ← NewItem
        Length ← Length + 1
    ELSE
        OUTPUT "Queue full"
    ENDIF
ENDPROCEDURE

```

B1 for the test on `Length`, **B1** for incrementing the rear, **B1** for wrapping to 0, **B1** for storing and incrementing `Length`.

(b)

After	Queue[0] ... Queue[5]	FrontPointer	RearPointer	Length
Initialise	–, –, –, –, –, –	0	-1	0
Enqueue P, Q, R, S, T	P, Q, R, S, T, –	0	4	5
Dequeue three times (P, Q, R)	–, –, –, S, T, –	3	4	2
Enqueue U	–, –, –, S, T, U	3	5	3
Enqueue V	V, –, –, S, T, U	3	0	4
Enqueue W	V, W, –, S, T, U	3	1	5

(A dequeued value may still sit in the array; it is shown as – because it is no longer in the queue.) **B1** for the front pointer, **B1** for the rear wrapping to 0, **B1** for Length.

(c) Length is 5 and MaxSize is 6, so one more item fits (it would go in cell 2). **B1** The front pointer is 3, so the next Dequeue returns **S**, the oldest item still waiting. **B1**

Example 3

The names Mia, Eli, Tom, Ava, Jon, Sam, Zoe and Kai are added, in that order, to an empty ordered binary tree stored in an array of records `Tree[0 : 9]`. Each record has `LeftPtr`, `Data` and `RightPtr`; –1 is the null pointer.

(a) Complete the array, `RootPtr` and `FreePtr`. **[4]**

(b) List the names visited when `FindNode` looks for Kai, and when it looks for Leo. **[2]**

(a) Mia is the root. Eli < Mia goes left; Tom > Mia goes right; Ava goes left of Eli; Jon goes right of Eli; Sam goes left of Tom; Zoe right of Tom; Kai: left of Mia, right of Eli, right of Jon.

Index	LeftPtr	Data	RightPtr
0	1	Mia	2
1	3	Eli	4
2	5	Tom	6
3	-1	Ava	-1
4	-1	Jon	7
5	-1	Sam	-1
6	-1	Zoe	-1
7	-1	Kai	-1
8			
9			

RootPtr = 0 and FreePtr = 8, with rows 8 and 9 blank. **B1** for Mia, Eli and Tom's rows, **B1** for Ava, Jon and Sam, **B1** for Zoe and Kai with every other pointer –1, **B1** for FreePtr.

(b) Kai: Mia → Eli → Jon → Kai, found at index 7. **B1** Leo: the same path, Mia → Eli → Jon → Kai, then Kai's right pointer is –1, so Leo is not in the tree and –1 is returned. **B1**

Example 4

```
FUNCTION DigitSum(N : INTEGER) RETURNS INTEGER
  IF N < 10 THEN
    RETURN N
  ELSE
    RETURN (N MOD 10) + DigitSum(N DIV 10)
  ENDIF
ENDFUNCTION
```

(a) Identify the base case and the general case. **[2]**

(b) Complete a trace table for DigitSum(4725). **[4]**

(c) Explain how the stack is used when this function runs. **[3]**

(a) Base case: $N < 10$, which returns N with no further call. **B1** General case: $(N \text{ MOD } 10) + \text{DigitSum}(N \text{ DIV } 10)$, which calls itself with a number one digit shorter, so it moves towards the base case. **B1**

(b)

Call	Function call	N	Result	Return value
1	DigitSum(4725)	4725	$5 + \text{DigitSum}(472)$	
2	DigitSum(472)	472	$2 + \text{DigitSum}(47)$	
3	DigitSum(47)	47	$7 + \text{DigitSum}(4)$	
4	DigitSum(4)	4	base case	4
(3)	DigitSum(47)	47	$7 + 4$	11
(2)	DigitSum(472)	472	$2 + 11$	13
(1)	DigitSum(4725)	4725	$5 + 13$	18

B1 for the calls and N, **B1** for the winding rows, **B1** for the unwinding rows, **B1** for the return values. There are 4 calls, and the first one returns 18.

(c) Each time DigitSum calls itself, the return address and the current value of N are pushed onto the stack.

B1 When DigitSum(4) reaches the base case, the stack unwinds. **B1** The frames are popped in reverse order (LIFO), so each waiting call gets its own N back and adds its digit to the value returned. **B1**

Common mistakes

- **Describing the wrong search.** Some candidates describe a linear search when asked for a binary search. And when asked what $O(\log n)$ means for a binary search, answer only that; a comparison with linear search earns nothing there.
- **Stopping at "a stack is LIFO".** For why a stack runs recursion, most answers say LIFO and stop. Go on: each call is **pushed**, the base case starts the **unwinding**, and calls are **popped** in reverse order.
- **Binary search gap-fills:** the usual errors are a wrong test at the top of the loop, `Lower ← Mid` instead of `Mid + 1` (it can loop for ever), and a wrong stopping condition.
- **Insertion sort:** losing the item being inserted. Store it in `Value` **before** the inner loop and write it back **after** the loop, at `Position + 1`.
- **Pushing or popping in the wrong order.** Push: move the top pointer, **then** store. Pop: read the item, **then** move the pointer. Pop also needs `RETURNS <type>` and `RETURN`.
- **Declarations half done.** Queue and stack questions give a mark for the constant and the array as well as the integers; many only declare the integers. Give the array its bounds **and** its type.
- **Printing instead of returning.** If a function is meant to return a message, `RETURN` it; don't `OUTPUT` it.
- **Rewriting what's given.** When a helper such as `StackFull()` is provided, call it; use the identifiers the question gives.
- **Recursion traces:** filling the result column top to bottom only. Values come back on the way **up**, after the base case.
- **Binary tree diagrams:** arrows from the middle of the node, missing null pointers in new leaves, or nodes attached to the wrong side.
- **Linked list:** in a "not found" message, don't output `Flower[Found]`: the item isn't there. And when asked to improve the **structure**, the answer is a free list to reuse deleted space, not a faster search.
- **Queue from two stacks:** remember the second stack exists to **reverse** the order; say which stack you pop from and push to at each step.

How to get the marks

- **Gap-fill pseudocode** earns one mark per correct line. Use the identifiers given, keep `THEN`, `ENDIF`, `ENDWHILE` and `RETURNS`, and use `←` for assignment.
- **"Write the pseudocode"** for declarations: `CONSTANT`, `DECLARE ... : INTEGER`, `DECLARE ... : ARRAY[1:u] OF <type>`. Bounds must match the size given.
- **"State"** needs a short answer ($O(n)$, "the data must be sorted", three features). **"Describe"** needs the points in order. **"Explain"** needs the reasons, for example why each push and pop happens. **"Compare"** needs both sides of each point: "linear starts at the beginning, binary starts in the middle".
- **Big O answers** get separate marks for the notation, for saying what Big O measures (time or space complexity), and for how time grows with n .
- **Trace tables:** one row per call; show the winding rows and the unwinding rows; only fill the cells that change.

- **Binary tree diagrams** are marked on the new nodes' data, arrows from the correct pointer boxes, a null value in every empty pointer, and nothing extra.
- **"Justify"** a choice (a queue for print jobs, a graph for a road map): say what the situation needs, then why the ADT provides it.
- **Examples** are asked for: always give one when describing when recursion is useful.

Quick check

1. What is the largest number of comparisons a binary search needs to find an item in a sorted array of 500 items?
2. Show the array after each pass of the outer loop when the insertion sort sorts 7, 3, 8, 2 into ascending order.
3. An empty stack in `Stack[1:6]` has `BottomPtr` 1 and `TopPtr` 0. A, B and C are pushed, one item is popped, D is pushed, and two items are popped. Which items are popped, in order, and what is `TopPtr` now?
4. $F(N)$ returns 0 if $N = 0$, otherwise $N * N + F(N - 1)$. What does $F(4)$ return, and how many calls are made?
5. A queue is built from two stacks, S1 and S2, using the method in "Understand it" (S1 holds the queue with the oldest item on top). Both start empty. X, Y and Z are added in that order, then one item is removed. Give the contents of both stacks, bottom to top, after each addition and after the removal, and state which item is removed.

Answers

1. $2^8 = 256 \leq 500 < 512 = 2^9$, so $\lfloor \log_2 500 \rfloor + 1 = 8 + 1 = 9$ comparisons.
2. After Index = 2: 3, 7, 8, 2. After Index = 3: 3, 7, 8, 2 (8 is already in place). After Index = 4: 2, 3, 7, 8.
3. The items popped are C, D, B, in that order. Only A is left, so TopPtr = 1.
4. $F(4) = 16 + F(3) = 16 + 9 + F(2) = \dots$, so $16 + 9 + 4 + 1 + 0 = 30$. $F(4)$, $F(3)$, $F(2)$, $F(1)$ and $F(0)$ make 5 calls.
5. Add X: S1 is empty, so X is pushed onto S2 and popped back: S1 = X, S2 empty. Add Y: X moves to S2, Y is pushed on top, then Y and X are popped back: S1 = Y, X (bottom to top), S2 empty. Add Z: S1 = Z, Y, X, S2 empty. Remove: pop S1, which returns X, the first item added; S1 = Z, Y, S2 empty.

Past questions to try

- **9618/33/M/J/23 Q11**: declaring a queue, completing Dequeue, and a queue made from two stacks.
- **9618/33/M/J/23 Q12**: what recursion is, and tracing `Fib(5)` through winding and unwinding.
- **9618/32/O/N/22 Q11**: updating and searching a linked list held in two arrays.
- **9618/32/M/J/24 Q8**: completing a linear search and comparing it with a binary search using Big O.