

Computational thinking and problem-solving

A-Level Computer Science 9618 · Topic 19 revision notes

Everything from the 5 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

- | | |
|--|---|
| 1. <u>Binary search</u> | 3. <u>Stacks, queues, linked lists and binary trees</u> |
| 2. <u>Insertion sort and comparing sorts</u> | 4. <u>Comparing algorithms: Big O notation</u> |
| | 5. <u>Recursion and the call stack</u> |

1 Binary search

Syllabus 19.1

A **binary search** only works on data that is **sorted** and held in a structure with **direct access** by index, such as an array. It compares the search item with the **middle** element, $\text{Mid} \leftarrow (\text{Low} + \text{High}) \text{ DIV } 2$. If they match, the item is found; if the middle item is smaller, the search continues in the **upper half** ($\text{Low} \leftarrow \text{Mid} + 1$); otherwise in the **lower half** ($\text{High} \leftarrow \text{Mid} - 1$). It stops when the item is found or **Low > High**. Each comparison **halves** the items left, so the worst case for n items is about $\log_2 n$ comparisons: 1000 items need at most 10.

Binary search for 47: each comparison halves the part still to search

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	8	12	17	21	26	30	34	39	42	47	51	58	63	70

Mid = $(1 + 15) \text{ DIV } 2 = 8$: $34 < 47$, so $\text{Low} \leftarrow 9$

3	8	12	17	21	26	30	34	39	42	47	51	58	63	70
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Mid = $(9 + 15) \text{ DIV } 2 = 12$: $51 > 47$, so $\text{High} \leftarrow 11$

3	8	12	17	21	26	30	34	39	42	47	51	58	63	70
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Mid = $(9 + 11) \text{ DIV } 2 = 10$: $42 < 47$, so $\text{Low} \leftarrow 11$

3	8	12	17	21	26	30	34	39	42	47	51	58	63	70
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Mid = $(11 + 11) \text{ DIV } 2 = 11$: $47 = 47$, found after 4 comparisons

Fig. 1: a binary search for 47 in a sorted array of 15 integers. Blue cells are still being searched, grey cells have been discarded.

Worked example: Trace a search that fails

The array in Fig. 1 holds 3, 8, 12, 17, 21, 26, 30, 34, 39, 42, 47, 51, 58, 63, 70 in elements 1 to 15. Trace a binary search for 20.

1. Low = 1, High = 15, Mid = 8: List[8] = 34. $34 > 20$, so $\text{High} \leftarrow 7$.
2. Low = 1, High = 7, Mid = 4: List[4] = 17. $17 < 20$, so $\text{Low} \leftarrow 5$.
3. Low = 5, High = 7, Mid = 6: List[6] = 26. $26 > 20$, so $\text{High} \leftarrow 5$.
4. Low = 5, High = 5, Mid = 5: List[5] = 21. $21 > 20$, so $\text{High} \leftarrow 4$.
5. Now Low = 5 and High = 4, so $\text{Low} > \text{High}$ and the loop ends.

Answer:

Low High Mid List[Mid] Action

1 15 8 34 High $\leftarrow$ 7

1 7 4 17 Low $\leftarrow$ 5

5 7 6 26 $\text{High} \leftarrow 5$

5 5 5 21 $\text{High} \leftarrow 4$

20 is **not found** after 4 comparisons (a linear search would need 15).

Low, Mid, High.

- ✓ **Conditions:** sorted data in a structure with direct access by index.
- ✓ **Mid:** $(\text{Low} + \text{High}) \text{ DIV } 2$; compare $\text{List}[\text{Mid}]$ with the search item.
- ✓ **Move:** item larger means $\text{Low} \leftarrow \text{Mid} + 1$; smaller means $\text{High} \leftarrow \text{Mid} - 1$.
- ✓ **Stop:** found, or $\text{Low} > \text{High}$. Worst case $\text{INT}(\log_2 n) + 1$ comparisons.

Watch out!

"A binary search splits the list in half" is not enough for the marks. Say what is compared (the middle item with the search item) and what is discarded (the half that cannot contain it, because the data is sorted).

Do not use the loop condition $\text{Low} < \text{High}$. When the item is in the last element left ($\text{Low} = \text{High}$) the loop would stop without checking it.

Do not say a binary search is "always faster". It needs sorted data, and on a tiny list, or when the item is the first element, a linear search can need fewer comparisons.

2 Insertion sort and comparing sorts

Syllabus 19.1

An **insertion sort** treats the first element as a **sorted part** of length one. Each pass takes the next element as the **key**, compares it with the sorted part from right to left, **shifts** every larger element one place to the right, and drops the key into the gap. After pass i the first i elements are in order. On data that is **already sorted** each pass makes one comparison, so it needs only $n - 1$ comparisons; on data in **reverse order** it needs $n(n - 1)/2$. The **bubble sort** has the same best and worst cases, but usually makes more comparisons and moves data with three-step swaps.

Insertion sort of 27, 12, 35, 8, 19: the sorted part grows by one each pass

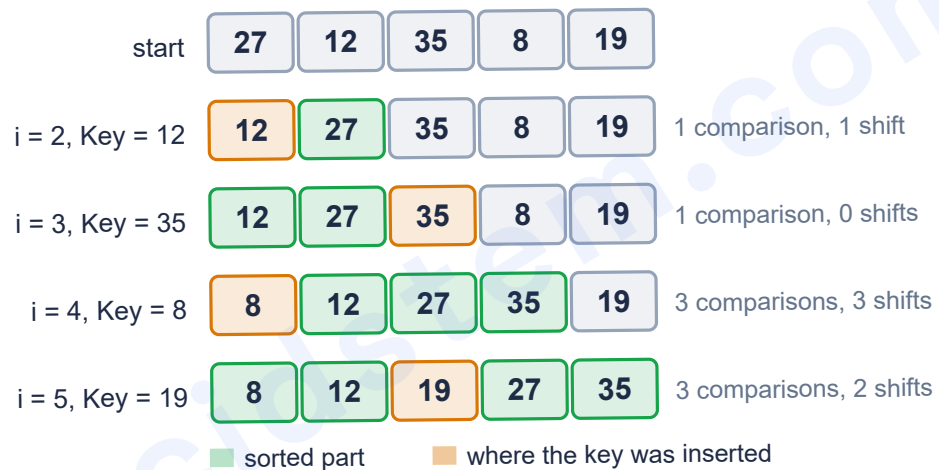


Fig. 1: an insertion sort of five integers, shown after each pass; 8 comparisons in total

Worked example: One pass in detail

After two passes, Nums holds 12, 27, 35, 8, 19 (Fig. 1). Describe pass 3, which inserts the key 8.

1. Key $\leftarrow$ 8, the element in position 4. The sorted part is 12, 27, 35.
2. $35 > 8$: shift 35 into position 4. $27 > 8$: shift 27 into position 3. $12 > 8$: shift 12 into position 2.
3. The start of the list has been reached, so the loop stops: 3 comparisons and 3 shifts.
4. Write the key into position 1.

Answer: Nums holds 8, 12, 27, 35, 19; the first four elements are now sorted

Key, shift, insert.

- ✓ **Outer loop:** FOR $i \leftarrow 2$ TO n , $\text{Key} \leftarrow \text{List}[i]$, $j \leftarrow i - 1$.
- ✓ **Inner loop:** while $j \geq 1$ and $\text{List}[j] > \text{Key}$, shift right and decrease j .
- ✓ **Insert:** $\text{List}[j + 1] \leftarrow \text{Key}$ after the inner loop.
- ✓ **Performance:** $n - 1$ comparisons if sorted, $n(n - 1)/2$ if reversed; $O(n^2)$ worst case.

Watch out!

Do not describe an insertion sort as "comparing adjacent pairs and swapping them": that is a bubble sort. Say that each item is inserted into its correct position in the sorted part.

Writing $\text{List}[j] \leftarrow \text{Key}$ after the loop puts the key one place too far left. When the loop stops, j points at the element that is *not* larger, so the gap is at $j + 1$.

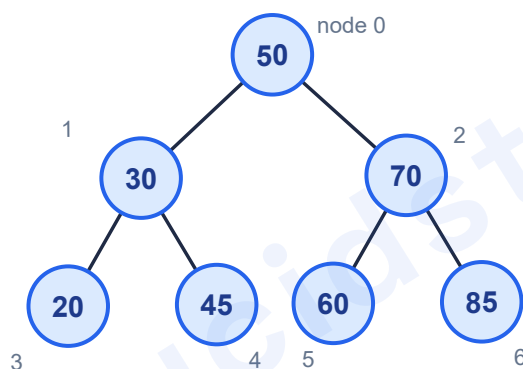
"Insertion sort is faster than bubble sort" earns little on its own. Link the answer to the data: for nearly sorted data both are quick, and in the worst case both need $n(n - 1)/2$ comparisons.

3 Stacks, queues, linked lists and binary trees

Syllabus 19.1

An **abstract data type (ADT)** is defined by its data and its **operations**, not by how it is stored. A **stack** (push, pop) needs an array and a **top pointer**; a **circular queue** (enqueue, dequeue) needs **front** and **rear pointers** that wrap round with **MOD**; a **linked list** needs an array of records with a **start pointer**, a **free list pointer** and a **null pointer** (-1) at the end; a **binary search tree** adds a **left** and a **right pointer** to each node, smaller values to the left. A **dictionary** stores **key-value pairs**, and a **graph** is a set of **vertices** joined by **edges**. ADTs can be **built from other ADTs**, for example a queue from two stacks.

A binary search tree and the array of records that stores it



left subtree smaller, right subtree larger
RootPointer = 0, FreePointer = 7

Index	Left	Data	Right
0	1	50	2
1	3	30	4
2	5	70	6
3	-1	20	-1
4	-1	45	-1
5	-1	60	-1
6	-1	85	-1
7	8	(free)	-1

Fig. 1: a binary search tree of seven integers and the array of records that stores it; unused nodes form a free list through LeftPointer

Worked example: Trace the circular queue

The queue above starts empty. Trace Enqueue 10, 20, 30, 40, 50; Dequeue three times; then Enqueue 60, 70, 80. Give the final pointers and the queue in order.

- Five enqueues: RearPointer goes 0, 1, 2, 3, 4 and NumberInQueue = 5.
- Three dequeues return 10, 20, 30; FrontPointer = 3 and NumberInQueue = 2.
- Enqueue 60: RearPointer = 5. Enqueue 70: RearPointer = $(5 + 1) \text{ MOD } 6 = 0$, into the space 10 left.
- Enqueue 80: RearPointer = 1. NumberInQueue = 5.

Answer: FrontPointer = 3, RearPointer = 1, NumberInQueue = 5; the queue from front to rear is **40, 50, 60, 70, 80** (indices 3, 4, 5, 0, 1)

Check, then move.

- ✓ **Insert:** check for full (free pointer null / count = size), then take a node and set the pointers.
- ✓ **Delete:** check for empty, then reconnect around the item and return the node to the free list.
- ✓ **Circular queue:** move pointers with $(\text{Pointer} + 1) \text{ MOD Size}$ and keep a count.
- ✓ **Binary tree:** from the root, smaller goes left, otherwise right, until a null pointer.

Watch out!

In a circular queue, write

$(\text{RearPointer} + 1) \text{ MOD } 6$,

not $\text{RearPointer} + 1$. Without MOD the pointer runs past index 5 even though spaces at the front are free.

Deleting the first node is a special case: there is no previous node, so the start pointer must change.

Code that only updates

$\text{List}[\text{Previous}].\text{Pointer}$

fails for it.

When inserting into a tree or a linked list, take the node from the free list and update the free pointer before setting the new node's own pointers, or the rest of the free list is lost.

4 Comparing algorithms: Big O notation

Syllabus 19.1

Algorithms that perform the same task are compared by the **time taken** (the number of basic steps, such as comparisons) and the **memory used**, as the number of data items n grows. **Big O notation** gives the **order of growth** in the **worst case**: keep only the **dominant term** and drop constants, so $3n^2 + 5n + 2$ steps is $O(n^2)$. **Time complexity** ranks $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$. **Space complexity** uses the same notation for extra memory: an in-place sort is $O(1)$, a copy of the data is $O(n)$.

How the number of steps grows as the number of items n grows

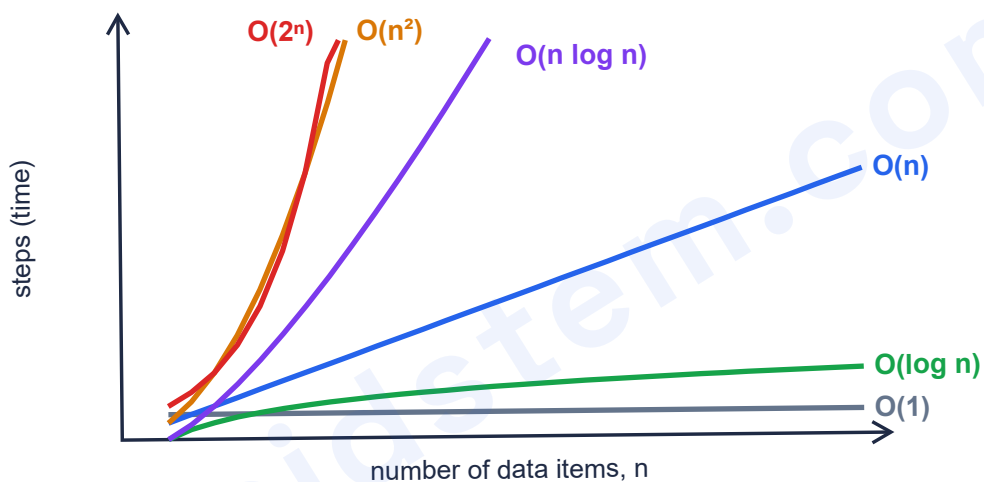


Fig. 1: the common orders of growth. For small n they are close; as n grows, $O(n^2)$ and $O(2^n)$ quickly become unusable.

Worked example: Count the steps

How many times is `Count ← Count + 1` executed by the code above when $n = 10$? What if the inner loop is changed to `FOR j ← 1 TO n`?

1. Original: the inner loop runs n times for each of the n values of i , so $n \times n = 10 \times 10 = 100$.
2. Changed: when $i = 1$ the inner loop runs 10 times, when $i = 2$ it runs 9 times, ..., when $i = 10$ it runs once.
3. $10 + 9 + \dots + 1 = 10 \times 11 / 2 = 55$, which is $n(n+1)/2 = 0.5n^2 + 0.5n$.

Answer: 100 and 55. Both grow with n^2 , so both are $O(n^2)$: halving the work does not change the order of growth

Dominant term only.

- ✓ **Criteria:** time taken (operations as n grows) and memory used (extra storage).
- ✓ **Simplify:** keep the dominant term, drop constants: $4n^2 + 9n$ is $O(n^2)$.
- ✓ **Order:** $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$.
- ✓ **Predict:** scale n by k and the time scales by k for $O(n)$, k^2 for $O(n^2)$.

Watch out!

"Algorithm A is faster because it took 0.3 seconds" earns nothing. Compare the number of operations as n increases, or the Big O order.

$O(\log n)$ does not mean "fewer than n " for every n , and $O(n^2)$ does not give an exact count. Big O describes how the work grows; for very small n a "worse" algorithm can still be quicker.

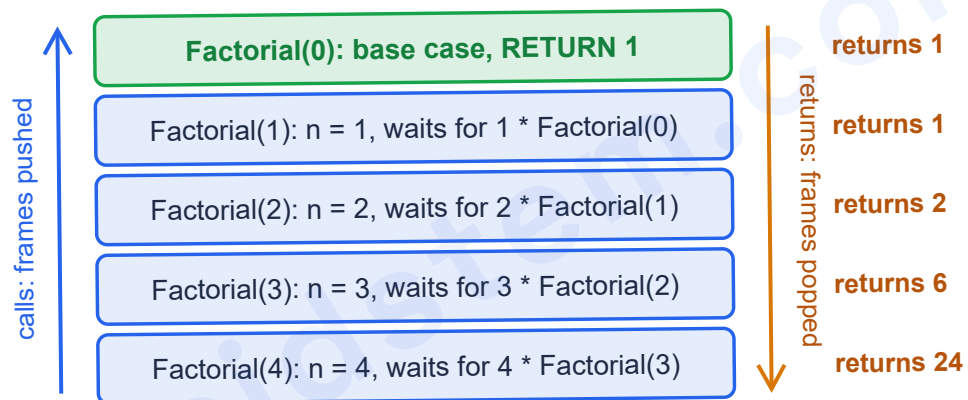
Do not confuse the two criteria. "It uses less memory" is a space argument; "it makes fewer comparisons" is a time argument. Say which one you mean and link it to n .

5 Recursion and the call stack

Syllabus 19.2

A **recursive** subroutine is one that is defined in terms of itself; it **calls itself**. It must have a **base case**, a condition where it returns a result **without** calling itself, and a **general (recursive) case** that calls itself with a value that moves **closer to the base case**. Each call is placed on the **call stack** as a **stack frame** holding its **return address**, **parameters** and **local variables**. When the base case is reached the calls return one by one in reverse order; the stack **unwinds**. With no reachable base case the stack fills up; **stack overflow**.

Factorial(4): the call stack winds up, then unwinds



Each frame holds the return address, the parameter n and any local variables

Fig. 1: the call stack for $\text{Factorial}(4)$. Five frames are pushed; the base case returns 1 and the frames are popped as the results are multiplied back down.

Worked example: Trace $\text{Factorial}(4)$

Trace the calls and returns for $\text{Factorial}(4)$.

1. $\text{Factorial}(4)$ cannot finish: it needs $4 * \text{Factorial}(3)$, so it calls $\text{Factorial}(3)$.
2. In the same way $\text{Factorial}(3)$ calls $\text{Factorial}(2)$, which calls $\text{Factorial}(1)$, which calls $\text{Factorial}(0)$.
3. $\text{Factorial}(0)$ is the base case and returns 1.
4. Unwinding: $\text{Factorial}(1)$ returns $1 * 1 = 1$; $\text{Factorial}(2)$ returns $2 * 1 = 2$; $\text{Factorial}(3)$ returns $3 * 2 = 6$; $\text{Factorial}(4)$ returns $4 * 6 = 24$.

Answer:

Call n Waits for Returns

1 4 4 * $\text{Factorial}(3)$ 24

```
2 33 * Factorial(2) 6
3 22 * Factorial(1) 2
4 11 * Factorial(0) 1
5 0 (base case) 1
```

Five calls in total; the value **24** is returned to the original caller.

Down, then up.

- ✓ **Features:** calls itself; a base case that stops; a general case that moves towards it.
- ✓ **Trace:** one row per call; fill in its result only when the call it made returns.
- ✓ **Compiler:** a stack frame per call (return address, parameters, locals); popped on return.
- ✓ **Judge:** shorter and clearer for recursive problems; more memory and stack overflow risk.

Watch out!

A base case that is never reached is as bad as none. `Factorial(-1)` calls `Factorial(-2)`, then `-3`, ... and never meets `n = 0`, so the stack overflows. State that each call must move towards the base case.

In a trace, do not output a value when a call is made if the **OUTPUT** is after the recursive call. Write each call on a new row, and fill in its output only when control returns to it.

"Recursion is faster" is usually wrong. A recursive solution is often shorter and clearer, but it normally uses more memory and more time than an equivalent loop because of the stack frames.