

Topic 9: Algorithm Design and Problem-Solving

Proving that an algorithm works, predicting how it scales, and a neat trick for turning slow into fast.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 9.1 Computational thinking skills, 9.2 Algorithms. Try the challenge questions on the last pages before you read the answers.

Loop invariants: proving an algorithm correct

Testing shows that an algorithm works for the inputs you tried; a **loop invariant** helps you show it works for *all* inputs. An invariant is a statement that is true before the loop, stays true after every iteration, and, when the loop ends, tells you the answer.

```
Max <- Data[1]
FOR i <- 2 TO N
  IF Data[i] > Max THEN Max <- Data[i]
NEXT i
// Invariant: after processing index i, Max = largest of Data[1..i]
```

- **Initialisation:** before the loop, $\text{Max} = \text{Data}[1]$, the largest of the first one item.
- **Maintenance:** if it holds for $i-1$, the IF keeps it true for i (Max is either unchanged or replaced by a bigger value).
- **Termination:** when $i = N$ the invariant says Max is the largest of all N items.

Growth rates: how does it scale?

Running time depends on the machine, but how it *grows with input size n* does not. This is what Big O notation (A Level) captures. Counting steps for different growth rates:

Growth	$n = 10$	$n = 1000$	$n = 1\,000\,000$	Typical algorithm
$\log n$	3	10	20	Binary search
n	10	1000	10^6	Linear search
$n \log n$	33	10^4	2×10^7	Merge sort
n^2	100	10^6	10^{12}	Bubble sort
2^n	1024	$\approx 10^{301}$	impossible	Trying every subset

A useful rule: if the input doubles, a linear algorithm takes $2\times$ as long, a quadratic one $4\times$. Binary search needs about $\log_2 n$ comparisons: just 20 for a million sorted items.

The two-pointer technique

Problem: given a *sorted* list, are there two values that add to a target T ? The obvious way checks every pair (about $n^2/2$ pairs). Keeping a pointer at each end does it in one pass:

```

Left <- 1
Right <- N
WHILE Left < Right
  Sum <- Data[Left] + Data[Right]
  IF Sum = T THEN RETURN TRUE
  ELSE IF Sum < T THEN Left <- Left + 1    // need a bigger sum
  ELSE Right <- Right - 1                // need a smaller sum
ENDWHILE
RETURN FALSE

```

Each step discards one value that can never be part of an answer, so at most n steps are needed: **linear** instead of quadratic. The key is exploiting the sorted order, a reminder to ask what extra structure the problem gives you.

Challenge questions

1. A loop adds the values of $\text{Data}[1..N]$ into Total . State a loop invariant for the loop and say what it proves when the loop ends.
2. What is the maximum number of comparisons binary search needs for 1 000 000 sorted items?
3. An algorithm with quadratic growth takes 2 seconds for $n = 1000$. Estimate the time for $n = 4000$.

Worked answers

Only look once you have had a proper go at each question.

1. Invariant: after processing index i , $\text{Total} = \text{Data}[1] + \text{Data}[2] + \dots + \text{Data}[i]$ (Total = 0 before the loop).
When $i = N$, Total equals the sum of all N values.
2. Each comparison halves the remaining items; $2^{20} = 1\,048\,576 \geq 1\,000\,000$, so **at most 20** comparisons.
3. Input is $4\times$ bigger, so a quadratic algorithm takes $4^2 = 16\times$ longer: about **32 seconds**.