Algorithms I

Computer Science Tripos – Part IA

Comprehensive Tripos Revision Guide

University of Cambridge – Paper 1

Section	Topics	Key Skills
1. Foundations	Correctness, pseudocode, asymptotic notation	Prove correctness with invariants; manipulate $O, \Omega, \Theta, o, \omega$
2. Recurrences	Substitution, recursion trees, Master Theorem	Solve $T(n)$ recurrences tightly, including non-standard cases
3. Sorting	Insertion, merge, quick, heap, counting/radix sort	Derive complexities; prove the $\Omega(n \lg n)$ comparison lower bound
4. Design Strategies	Divide-and-conquer, dynamic programming, greedy	Identify optimal substructure; construct DP recurrences; prove greedy correctness
5. Data Structures	Stacks, queues, lists, heaps, BSTs, B-trees	Implement and analyse operations; choose the right structure
6. Hashing	Chaining, open addressing, hash functions	Analyse load factor, probing, and amortised performance

Compiled from the 2025/26 Algorithms I lecture course (Dr J. Fawcett), example sheets, and Tripos past papers 2006–2026.

Contents

1 Foundations

1.1 Algorithms, Problems and Correctness

An **algorithm** is a well-defined computational procedure that takes some value(s) as *input* and produces some value(s) as *output*, and which is guaranteed to terminate.

A **problem** specifies, in general terms, what inputs are permitted and what output is required for each input; a **problem instance** is one particular input. For example:

The Sorting Problem

Input: a sequence of numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: a permutation $\langle b_1, b_2, \ldots, b_n \rangle$ of the input such that $b_1 \leq b_2 \leq \cdots \leq b_n$.

The items being sorted are **keys**; they may carry attached **payloads** (values) which must be moved along with their keys.

An algorithm is **correct** if, for *every* input instance, it terminates with the correct output. An incorrect algorithm either fails to terminate on some instance, or terminates with the wrong answer. This course only studies correct algorithms (randomised/incorrect algorithms are Part II material).

How to prove correctness: loop invariants

A **loop invariant** is a property that holds:

1. **Initialisation** – before the first iteration;
2. **Maintenance** – if it holds before an iteration, it holds before the next;
3. **Termination** – when the loop ends, the invariant (plus the reason the loop ended) implies correctness.

This mirrors mathematical induction and is the standard Tripos technique for proving an algorithm correct.

1.2 Pseudocode Conventions (this course)

- Arrays are **1-indexed**: $A[1..n]$ has length n; $A[1]$ is the first element. `A.length` gives n.
- Parameters are passed **by value**; objects/arrays are passed as **pointers**.
- `A|B` and `A&B` always evaluate both operands; `A||B` and `A&&B` short-circuit (Java convention).
- A loop induction variable retains its final value after the loop terminates.
- Indentation (whitespace) delimits blocks, as in Python.

Trap: 0- vs 1-indexing

Some topics (hashing, heaps-as-arrays) are more natural 0-indexed and past papers sometimes switch convention (e.g. Python slice notation). **Always state which convention you are using** at the start of your answer, and be consistent – off-by-one errors are the single most common way to lose easy marks.

1.3 Asymptotic Notation

We describe running time as a function of input size n, ignoring machine-dependent constants and focusing on growth rate for large n.

Definitions ($f, g : \mathbb{N} \to \mathbb{R}^+$)

$$f(n) \in O(g(n)) \iff \exists\, c > 0, n_0 : \forall n \geq n_0,\ f(n) \leq c\, g(n) \quad \text{(asymptotic upper bound)}$$
$$f(n) \in \Omega(g(n)) \iff \exists\, c > 0, n_0 : \forall n \geq n_0,\ f(n) \geq c\, g(n) \quad \text{(asymptotic lower bound)}$$
$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n)) \quad \text{(tight bound)}$$
$$f(n) \in o(g(n)) \iff \forall\, c > 0,\ \exists n_0 : \forall n \geq n_0,\ f(n) < c\, g(n) \quad \left(\lim \tfrac{f}{g} = 0\right)$$
$$f(n) \in \omega(g(n)) \iff \forall\, c > 0,\ \exists n_0 : \forall n \geq n_0,\ f(n) > c\, g(n) \quad \left(\lim \tfrac{f}{g} = \infty\right)$$

Useful facts

- Θ is an equivalence relation; O is like $\leq$, Ω is like $\geq$, o is like $<$, ω is like $>$.
- Polynomials: $an^k + \cdots \in \Theta(n^k)$ for constant $a > 0$.
- Logs: $\log_a n = \Theta(\log_b n)$ for any constants $a, b > 1$ – the base doesn't matter inside $\Theta/O/\Omega$.
- $n^\epsilon \in \omega(\lg n)$ for any $\epsilon > 0$: polynomials beat any power of a logarithm.
- $c^n \in \omega(n^k)$ for any constant $c > 1$, any k: exponentials beat polynomials.
- $n! \in \omega(c^n)$ for constant c; $\lg(n!) = \Theta(n \lg n)$ (Stirling).
- To compare f, g, a reliable method is $\lim_{n\to\infty} \frac{f(n)}{g(n)}$: if it is 0, $f \in o(g)$; if ∞, $f \in \omega(g)$; if a positive constant, $f \in \Theta(g)$.

Common exam traps

- $O(g(n))$ is an *upper bound on the algorithm's actual behaviour*, not a promise of tightness – "worst case is $O(n^2)$" does *not* mean it's not also $O(n^3)$. When asked for the *tight* bound, give Θ.
- Best/worst/average case describe *which inputs* you are quantifying over; O, Ω, Θ describe the *growth rate*. You can (and often must) say things like "worst-case running time is $\Theta(n^2)$".
- Don't confuse "not $O(f)$" with "$\Omega(f)$" – a function can fail to be $O(f)$ without being $\Omega(f)$ if it oscillates. (Rare in this course, but examiners occasionally test the distinction.)

2 Recurrence Relations

Recursive algorithms' running times satisfy recurrence relations. Solving these tightly is a heavily-examined skill (every recent paper has a multi-part recurrence question) – practise until it's mechanical.

2.1 Three methods

Method 1: The Substitution (guess-and-verify) Method

Guess a bound, then prove it by induction.

1. Guess the form of the solution, e.g. $T(n) \leq cn \lg n$.
2. Assume it holds for all smaller values.
3. Substitute into the recurrence and show the guess holds for n, choosing constants c, n_0 appropriately.

Trick for tightening a guess: if $T(n) \leq cn - b$ fails by an additive constant, try subtracting a lower-order term (e.g. guess $T(n) \leq cn - b$ rather than cn) to absorb the slack – a classic manoeuvre for recurrences like $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1$.

Method 2: Recursion Trees

Draw the tree of recursive calls: each node is labelled with the non-recursive cost at that call. Sum the cost **level by level**, then sum over all levels (a geometric-type series). This is the best way to find the right guess to feed into substitution, and is often accepted as a complete proof if argued carefully (state the per-level cost, the number of levels, and sum the series explicitly).

Method 3: The Master Theorem

For $T(n) = aT(n/b) + f(n)$ with $a \geq 1$, $b > 1$ constants, and f asymptotically positive:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. *(leaves dominate)*
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$. *(balanced)*
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, *and* the regularity condition $af(n/b) \leq cf(n)$ holds for some $c < 1$ and all large n, then $T(n) = \Theta(f(n))$. *(root dominates)*

Compare $f(n)$ against $n^{\log_b a}$: case 1 needs f *polynomially smaller*, case 3 needs f *polynomially larger* – a gap of only $\lg n$ factors (e.g. $f(n) = n^{\log_b a} \lg n$) falls into **none** of the three cases and the Master Theorem *cannot* be used.

When the Master Theorem does *not* apply – fall back to substitution/recursion trees

- $T(n) = T(n-c) + f(n)$ – subtractive, not n/b: not of Master Theorem form at all (this is the classic "$T(n) = T(n-1) + \lg n$" style question). Recursion-tree/direct-summation gives $T(n) = T(1) + \sum_k f(n-k)$, e.g. $\sum_{i=1}^{n} \lg i = \lg(n!) = \Theta(n \lg n)$.
- a or b not constant, e.g. $T(n) = nT(\sqrt{n}) + n$: substitute $m = \lg n$ to linearise.
- The polynomial-gap case (between cases 1 and 2, or 2 and 3): use the tree method directly.
- Floors/ceilings and additive shifts inside the recursive term, e.g. $T(n) = 4T(n/4 - 2) + kn$: usually change of variable, or argue the shift is asymptotically negligible and the Master Theorem's conclusion is unaffected – but you must justify this (show the recursion still bottoms out in $\Theta(\log n)$ levels and the per-level cost sums the same way).

2.2 Worked example (recursion tree, non-standard recurrence)

Consider $T(n) = 2T(n/4) + \dfrac{n}{\sqrt{n}} = 2T(n/4) + \sqrt{n}$.

Here $a = 2, b = 4$, so $n^{\log_b a} = n^{\log_4 2} = n^{1/2} = \sqrt{n}$. Since $f(n) = \sqrt{n} = \Theta(n^{\log_b a})$, Master Theorem case 2 applies directly:

$$T(n) = \Theta(\sqrt{n} \lg n).$$

From this we can immediately read off looser bounds requested in past papers, e.g. $T(n) \in \omega(\sqrt{n})$ (since $\lg n \to \infty$) and $T(n) \in O(n^2)$ trivially, and check whether $T(n) \in o(n \lg n)$: since $\sqrt{n} \lg n = o(n \lg n)$ (as $\sqrt{n}/n \to 0$), the answer is **yes**.

2.3 Worked example (Master Theorem fails – polynomial gap)

$T(n) = 8T(n/2) + n^2$. Here $a = 8, b = 2$, $n^{\log_2 8} = n^3$. Since $f(n) = n^2 = O(n^{3-\epsilon})$ for $\epsilon = 1$, case 1 applies: $T(n) = \Theta(n^3)$.

Contrast with $T(n) = T(n/5) + T(3n/5) + kn$ (*unequal* split – not of Master Theorem form since it isn't $aT(n/b)$ with a single b). Use a recursion tree: at each level the total work done across all nodes at that level is exactly kn (since the pieces at each level always sum to n), for as long as the recursion continues. The tree has depth $\Theta(\lg n)$ (the shortest root-to-leaf path shrinks by a factor $5/3$ each time along the "fast" branch $3n/5$, the longest by 5 along the "slow" branch $n/5$ – both are $\Theta(\lg n)$). Hence total work is $\Theta(n \lg n)$, so in particular $T(n) \notin O(n)$ (it is $\Omega(n \lg n) \subsetneq \omega(n)$, so $T(n) \in \omega(n)$, hence not $O(n)$).

Exam technique for recurrence questions

1. State which method you're using and why (e.g. "in Master Theorem form with $a = \ldots, b = \ldots$").
2. Compute $n^{\log_b a}$ explicitly and compare against $f(n)$.
3. If claiming $O/\Omega/o/\omega$ without full Θ, justify with a limit or explicit constants – don't just assert it.
4. When floors/ceilings/shifts appear, briefly argue why they don't change the asymptotics (or show the change of variable that removes them).

Practice: y2025 Paper 1 Q7 (Recurrences)

(a) Find asymptotically tight lower and upper bounds for $T(1) = 1$, $T(n) = 5T(n/5) + n^3$, and explain your answer. *[7 marks]*
(b) Find an asymptotically tight upper bound for $T(1) = 1$, $T(n) = T(n-1) + \lg n$. *[3 marks]*
(c) Show that $T(1) = 1$, $T(n) = T(n/a) + \lg n$ (constant $a > 1$) is in $\omega(\lg n)$ and $O(n)$. Is $T(n) \in o(\lg^2 n)$? *[7 marks]*
(d) Let $T(1) = 1$, $T(n) = T(n/5) + T(3n/5) + kn$ ($k > 0$). Is $T(n) \in O(n)$? Justify. *[3 marks]*

Practice: y2026 Paper 1 Q7 (Recurrences)

(a) $T(n) = 1$ if $n \leq 10$, else $4T(n/4 - 2) + kn$. Is $T(n) \in O(n)$? Justify. *[4 marks]*
(b) Find asymptotically tight lower and upper bounds for $T(1) = 1$, $T(n) = 8T(n/2) + n^2$. *[6 marks]*
(c) Find an asymptotically tight upper bound for $T(n) = 1$ if $n \leq 5$, else $T(n-5) + n^2$, and justify tightness. *[3 marks]*
(d) Show $T(1) = 1$, $T(n) = 2T(n/4) + n/\sqrt{n}$ is in $\omega(\sqrt{n})$ and $O(n^2)$. Is $T(n) \in o(n \lg n)$? *[7 marks]*

3 Sorting

3.1 Insertion Sort

INSERTION-SORT(A)

```
for j = 2 to A.length
    Key = A[j]
    i = j - 1
    while i > 0 && A[i] > Key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = Key
```

Loop invariant P: at the start of each iteration of the `for` loop, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, in sorted order.

- **Initialisation:** $j = 2$, so $A[1..1]$ is trivially sorted.
- **Maintenance:** the `while` loop shifts elements greater than `Key` one place right, then inserts `Key` in the gap – preserving sorted order over $A[1..j]$.
- **Termination:** loop ends when $j = n + 1$, so P gives $A[1..n]$ sorted – this is *partial correctness*. **Total correctness** additionally requires termination: both loops count towards a bound (the `for` loop counts up to a finite length; the `while` loop's i counts down and is bounded below by 0), so the algorithm always halts.

Complexity: $\Theta(n^2)$ worst case (reverse-sorted input: inner loop always shifts all preceding elements) and best case $\Theta(n)$ (already sorted: inner loop never executes). **In-place** ($O(1)$ extra space) and **stable** (equal keys keep their relative order, since the `while` condition is strict $>$).

3.2 Divide-and-Conquer Sorting

The Divide-and-Conquer paradigm

1. **Divide** the problem into smaller subproblems of the same form.
2. **Conquer** the subproblems recursively (base case: trivially small).
3. **Combine** the subproblem solutions into a solution for the original problem.

Merge Sort

MERGE-SORT(A, p, r)

```
MERGE-SORT(A, p, r)
    if p < r
        q = floor((p + r) / 2)
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q+1, r)
        MERGE(A, p, q, r)

MERGE(A, p, q, r)
    // Merge sorted A[p..q] and A[q+1..r] using an auxiliary
    // array of size (r - p + 1); copy the result back into A.
    // Use a sentinel or explicit bounds check to detect
    // when one side is exhausted.
```

Complexity: $T(n) = 2T(n/2) + \Theta(n)$. By the Master Theorem ($a = 2, b = 2$, $n^{\log_2 2} = n = \Theta(f(n))$, case 2): $T(n) = \Theta(n \lg n)$ – worst, best, *and* average case are all $\Theta(n \lg n)$, since MERGE always costs $\Theta(n)$ regardless of input order. **Space:** $\Theta(n)$ auxiliary (for the standard array version) – *not* in-place. **Stable** if MERGE breaks ties by preferring the left subarray.

Merge sort on linked lists

On a doubly linked list, MERGE can splice nodes in $O(1)$ per element without extra array space, and finding the midpoint can be done with a slow/fast pointer scan in $\Theta(n)$ – giving $\Theta(n \lg n)$ time, $O(1)$ *extra* space (excluding recursion stack), which is a classic Tripos comparison against the array version's $\Theta(n)$ auxiliary space.

Quicksort

QUICKSORT(A, p, r)

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)

PARTITION(A, p, r)
    x = A[r]                    // pivot
    i = p - 1
    for j = p to r - 1
        if A[j] <= x
            i = i + 1
            exchange A[i] and A[j]
    exchange A[i+1] and A[r]
    return i + 1
```

PARTITION maintains the invariant that $A[p..i] \leq x < A[i+1..j-1]$, with $A[j..r-1]$ unprocessed and $A[r] = x$; it runs in $\Theta(n)$ for a subarray of size n.

Complexity analysis

Worst case occurs when the partition is maximally unbalanced (e.g. already-sorted input with last-element pivot): $T(n) = T(n-1) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$.
Best case (balanced split): $T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(n \lg n)$.
Average case (random input, or randomised pivot choice) is also $\Theta(n \lg n)$: even a split of $\frac{1}{10} : \frac{9}{10}$ at every level still gives $\Theta(n \lg n)$, since what matters is that the split ratio is bounded away from 0 and 1; a formal expectation argument (summing over all $O(n)$ possible pivot ranks, each equally likely) confirms $E[T(n)] = O(n \lg n)$.

Space: $O(\lg n)$ expected stack depth (worst case $O(n)$) – in-place with respect to the array (partition swaps within A). **Not stable** in general (partition can reorder equal keys).

Avoiding the worst case in practice

- **Randomised pivot:** swap $A[r]$ with a uniformly random element of $A[p..r]$ before partitioning. This makes the $\Theta(n^2)$ worst case astronomically unlikely for *any* fixed input – the guarantee becomes about the algorithm's own coin flips, not about input distribution.
- **Median-of-three:** pivot on the median of $A[p], A[\lfloor (p+r)/2 \rfloor], A[r]$ – cheap heuristic that defeats already-sorted/reverse-sorted inputs but can still be attacked adversarially.
- **Hybrid with insertion sort:** stop recursing once a subarray is small (e.g. ≤ 10 elements) and finish with insertion sort. A single pass of insertion sort over an array that is a concatenation of already-sorted runs of length $\leq k$ still costs only $O(nk)$ (each element moves at most k places), so this is a $\Theta(n)$ (for constant k) clean-up pass – *not* $\Theta(n^2)$, because no element is ever more than k positions from its sorted position.

These points map directly onto a recurring past-paper question type: "why is the worst case unlikely/avoidable in practice?"

3.3 Heapsort

Heapsort uses the **binary heap** (see §??) to sort in-place in $\Theta(n \lg n)$ worst-case time – combining the good worst-case guarantee of merge sort with the $O(1)$ extra space of insertion/quicksort.

HEAPSORT(A)

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)              // Theta(n)
    for i = A.length downto 2
        exchange A[1] and A[i]
        A.heap-size = A.heap-size - 1
        MAX-HEAPIFY(A, 1)          // O(lg n)
```

Complexity: $\Theta(n)$ to build the heap $+\Theta(n) \cdot O(\lg n) = \Theta(n \lg n)$ total, for *every* input (worst = best = average, up to constants) since the heap operations don't depend on input order beyond the (amortised) $\Theta(n)$ build. **Space:** $\Theta(1)$ extra – fully in-place. **Not stable.**

Exam favourite: Heapsort on sorted input

Heapsort's running time is $\Theta(n \lg n)$ *regardless* of whether the input is already sorted (ascending or descending) – unlike insertion sort. Building the heap from an ascending array still costs $\Theta(n)$ (most MAX-HEAPIFY calls are near the leaves and do $O(1)$ work), and each of the $n-1$ extractions still costs up to $\Theta(\lg n)$ because the element swapped to the root can be arbitrarily far from where it belongs. Don't be tempted to answer "$O(n)$" – justify with the shape of the heap after each swap.

3.4 Linear-Time Sorting (Counting Sort, Radix Sort)

Counting Sort

Assumes keys are integers in $\{0, 1, \ldots, k\}$. **Idea:** for each key, count how many elements are $\leq$ it, which gives its final position directly.

```
COUNTING-SORT(A, B, k)          // A: input, B: output, keys in 0..k
    let C[0..k] be new array, initialised to 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1              // C[i] = #elements equal to i
    for i = 1 to k
        C[i] = C[i] + C[i-1]               // C[i] = #elements <= i
    for j = A.length downto 1              // iterate backwards for stability
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Complexity: $\Theta(n+k)$ time, $\Theta(n+k)$ space. **Stable** (crucially depended on by radix sort). Not comparison-based, so is not bound by the $\Omega(n \lg n)$ lower bound below.

Radix Sort

Sorts n d-digit numbers by sorting on each digit from **least significant to most significant**, using a stable sort (typically counting sort) as the subroutine at each digit position. **Correctness** relies on stability: after sorting on digit i, any two elements agreeing on digits $0..i$ retain their relative order from the previous pass, which is exactly their correct relative order w.r.t. all digits examined so far.

Complexity: d passes of counting sort on digits with radix (base) r, each costing $\Theta(n+r)$, giving $\Theta(d(n+r))$. For n keys with b-bit values split into $d = b/\lg r$ digits, choosing $r = \Theta(n)$ gives $\Theta\left(\frac{b}{\lg n}\, n\right)$ overall – linear when $b = O(\lg n)$.

3.5 The $\Omega(n \lg n)$ Lower Bound for Comparison Sorts

Decision-tree argument

Any comparison sort can be modelled as a binary **decision tree**: internal nodes are comparisons $A[i] : A[j]$, leaves are output permutations. For n distinct elements there are $n!$ possible orderings, so the tree needs at least $n!$ leaves. A binary tree of height h has at most 2^h leaves, so:

$$2^h \geq n! \implies h \geq \lg(n!) = \Theta(n \lg n) \quad \text{(by Stirling's approximation)}.$$

The height of the decision tree is exactly the worst-case number of comparisons, so **every** comparison-based sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case. Merge sort and heapsort are therefore asymptotically optimal comparison sorts.

Ternary / k-ary search and search lower bounds (same idea, different tree)

The identical counting argument bounds *comparison-based search*: a search tree distinguishing n outcomes with branching factor k has height $\geq \log_k n$, so k-ary search needs $\Omega(\log_k n)$ probes. Binary search ($k = 2$) needs $\lceil \lg n \rceil$ comparisons in the worst case; ternary search needs $\lceil \log_3 n \rceil \approx \lg n / 1.585$ – *fewer* comparisons per probe count, but *each* ternary step does up to 2 comparisons against 1 for binary search, so ternary search does *more* total comparisons in the worst case ($\approx 2 \log_3 n \approx 1.26 \lg n$, i.e. about 26% more) – a classic "more probes look cheaper but aren't" trap.

3.6 Summary Table

Algorithm	Best	Worst	Space	Stable?	In-place?
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$O(1)$	Yes	Yes
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n)$	Yes	No
Quicksort	$\Theta(n \lg n)$	$\Theta(n^2)$	$O(\lg n)$ exp.	No	Yes
Heapsort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(1)$	No	Yes
Counting Sort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	Yes	No
Radix Sort	$\Theta(d(n+r))$	$\Theta(d(n+r))$	$\Theta(n+r)$	Yes	No

Practice: y2010 Paper 1 Q5 (Merge Sort)

(a) Describe the basic principle of mergesort. Illustrate by sorting $\{9, 3, 6, 2, 4, 1, 5\}$. *[6 marks]*
(b) Insertion sort can be seen as a mergesort splitting an array of size n into pieces of size 1 and $n-1$. By solving the recurrence, show this recursive insertion sort is $O(n^2)$ (merging costs $O(n)$). *[5 marks]*
(c)(i) Show linked-list mergesort is $O(n \lg n)$ time and $O(1)$ space. (ii) Comment on the strategy of converting arrays to linked lists first to save space. *[9 marks]*

Practice: y2006 Paper 1 Q4 (Quicksort)

(a) State the average time and space complexity of quicksort. *[2 marks]*
(b) State the worst-case time and space complexity, and briefly explain how it arises. *[3 marks]*

(c) A two-phase sort does a partial quicksort (no recursion below 10 items) then insertion sort. What is the worst-case time complexity of the second phase? Explain with a diagram. *[5 marks]*

Practice: y2006 Paper 1 Q11 (Comparison lower bound)

(a) Time complexity of binary search on N items? *[1 marks]*
(b) What is the best possible worst-case time complexity of a comparison-based sort? Explain clearly (no formal proof required). *[7 marks]*
(c) A ternary search trisects the search space at each step. (i) Derive asymptotic expressions for the number of items queried by binary vs. ternary search in the worst case. (ii) Approximately what percentage *more* items does ternary search query than binary search, in the worst case? ($\lg 3 \approx 1.6$.) *[12 marks]*

4 Strategies for Algorithm Design

4.1 Divide and Conquer – Recap

Already covered via merge sort (§3.2.1) and analysed via recurrences (§2). The general recipe: divide into a subproblems of size n/b, conquer recursively, combine in $f(n)$ – giving $T(n) = aT(n/b) + f(n)$, solved by the Master Theorem.

Classic D&C example: exponentiation

Naive x^n takes $\Theta(n)$ multiplications. D&C: $x^n = (x^{n/2})^2$ if n even, $x \cdot (x^{\lfloor n/2 \rfloor})^2$ if odd. $T(n) = T(n/2) + \Theta(1) \Rightarrow \Theta(\lg n)$ by Master Theorem case 2 ($a = 1, b = 2$, $n^{\log_2 1} = 1 = \Theta(f(n))$).

4.2 Dynamic Programming

DP solves problems by combining solutions to **overlapping** subproblems, storing (**memoising**) results to avoid recomputation. It applies when a problem has both:

Requirements for Dynamic Programming

1. **Optimal substructure:** an optimal solution to the problem contains within it optimal solutions to subproblems.
2. **Overlapping subproblems:** a recursive algorithm revisits the *same* subproblems repeatedly (as opposed to D&C, where subproblems are disjoint).

Trap: optimal substructure is easy to state *incorrectly*

A recurring Tripos question type gives a plausible-sounding "optimal substructure" claim and asks you to **disprove it** with a counterexample. E.g. for "longest descending subsequence", the claim *"splitting s in two, the optimum is the concatenation of the optima of each half"* is false: the two local optima need not be compatible (the last element of the first optimum might not exceed the first element of the second). **Always sanity-check a substructure claim against a small concrete example before using it.**

Two Implementation Styles

Top-down (memoization) vs. Bottom-up (tabulation)

Top-down: write the natural recursive algorithm; before computing, check a memo table (initialised to "unknown"); after computing, store the result. Only computes subproblems actually needed.

Bottom-up: identify a natural order of subproblems (e.g. increasing size) and fill a table iteratively, so each entry's dependencies are already computed. Usually has lower constant-factor overhead (no recursion) and makes space optimisation easier (e.g. keeping only the last few rows of a table).

Both give the same asymptotic time: Θ(number of subproblems $\times$ time per subproblem excluding recursive calls).

Worked example: Longest Common Subsequence (LCS)

Input: sequences $X = \langle x_1, \ldots, x_m \rangle$, $Y = \langle y_1, \ldots, y_n \rangle$. **Output:** length of the longest subsequence common to both.

Recurrence

Let $c[i, j]$ = length of the LCS of $X[1..i]$ and $Y[1..j]$.

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & x_i = y_j \\ \max(c[i-1, j],\, c[i, j-1]) & x_i \neq y_j \end{cases}$$

Optimal substructure: if $x_i = y_j$, this pair must be the end of *some* LCS, and the rest is an LCS of the prefixes; if $x_i \neq y_j$, the LCS omits x_i or omits y_j (or both), so it's the best of the two sub-problems.

LCS-LENGTH(X, Y) – bottom-up

```
for i = 0 to m: c[i,0] = 0
for j = 0 to n: c[0,j] = 0
for i = 1 to m
    for j = 1 to n
        if X[i] == Y[j]
            c[i,j] = c[i-1,j-1] + 1
        else
            c[i,j] = max(c[i-1,j], c[i,j-1])
return c[m,n]
```

Complexity: $\Theta(mn)$ time and space for the table (can reduce to $\Theta(\min(m, n))$ space if only the length, not the sequence itself, is needed – since row i only depends on row $i - 1$).

Other classic DP problems worth knowing cold

- **Matrix-chain multiplication:** $m[i, j] = \min_{i \le k < j} \big(m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\big)$; $\Theta(n^3)$ time, $\Theta(n^2)$ subproblems.
- **0/1 Knapsack:** $K[i, w] = \max(K[i - 1, w],\ K[i - 1, w - w_i] + v_i$ if $w_i \le w)$; $\Theta(nW)$ time – *pseudo*-polynomial (depends on the *value* W, not $\lg W$).
- **Rod cutting / coin change / recharging-point problems (past-paper favourite):** let $f(i)$ = optimal cost to reach point i; $f(i) = \min_{j \text{ reachable from } i} \big(f(j) + \text{cost}(j \to i)\big)$, base case $f(0) = 0$. This exact template underlies the y2026 haulage-truck recharging question below.

Reduction vs. direct formulation

Tripos questions increasingly ask for *both*: (a) a recurrence that solves the stated problem *directly*, and (b) a reduction to a "well-known" DP problem covered in lectures (e.g. reducing "maximum descending subsequence" to *Longest Increasing Subsequence* by reversing the sequence and negating the order). Practise recognising when your problem is a relabelled version of LCS, LIS, or the shortest-path-in-a-DAG template.

4.3 Greedy Algorithms

A greedy algorithm builds a solution by repeatedly making the choice that looks best *right now*, never reconsidering it. Greedy is only correct when the problem has:

Requirements for a correct Greedy algorithm

1. **Greedy-choice property:** a globally optimal solution can be reached by making a locally optimal (greedy) first choice, then solving the remaining subproblem optimally.
2. **Optimal substructure:** as with DP – but greedy makes just *one* choice per step and recurses on *one* resulting subproblem (never branches/reconsiders), which is what makes it

faster than DP when it applies.

Standard correctness proof technique: the exchange argument

To prove a greedy algorithm optimal:

1. Take an arbitrary optimal solution O.
2. Show that if O doesn't start with the greedy choice, it can be modified ("exchanged") to do so **without making it worse** – i.e. construct another optimal solution O' that agrees with the greedy choice.
3. Conclude by induction on the remaining subproblem: the greedy choice is safe, and the rest of the problem has optimal substructure.

This is the standard structure examiners expect for "prove your greedy algorithm is correct" – a vague appeal to "it obviously works" earns few marks.

Worked example: Activity Selection

Input: activities with start/finish times (s_i, f_i). **Output:** maximum-size set of mutually compatible (non-overlapping) activities.

Greedy rule: always pick the remaining activity with the **earliest finish time**. **Exchange argument sketch:** let a_k be the activity with earliest finish time and O any optimal solution; if $a_k \notin O$, let $a_j \in O$ be the activity in O with the earliest finish time – since $f_k \leq f_j$, replacing a_j by a_k in O cannot break compatibility with the rest of O (everything else starts after $f_j \geq f_k$), and gives a solution of the same size. Hence some optimal solution contains a_k; recurse on activities compatible with a_k. Runs in $\Theta(n \lg n)$ (dominated by sorting by finish time).

Worked example: Huffman Coding

Problem: given symbol frequencies, build a binary prefix code minimising expected code length (equivalently, minimising total encoded length).

Greedy rule: repeatedly take the two **least frequent** nodes/trees and merge them into a new tree (frequency = sum), until one tree remains. Implemented efficiently with a min-priority queue: $\Theta(n \lg n)$ using a binary heap.

Trap: when Huffman gives no compression

If *all* symbols in a string are equally frequent (or more generally, if the frequency distribution is such that the resulting optimal code assigns every symbol the *same* code length, e.g. frequencies that are all equal and a power-of-two alphabet size), Huffman coding degenerates to a fixed-length code and **gives no compression** over the trivial encoding. This is a common "explain why Huffman fails to help here" past-paper part – the key insight is that Huffman's saving comes entirely from *skew* in the frequency distribution; a uniform distribution has none to exploit, and by the entropy bound $H = -\sum p_i \lg p_i$ is already maximised (equal to lg(alphabet size)) at uniform frequencies, so no code can do better than fixed-length.

DP vs. Greedy: how to tell which applies

- If making the locally-best choice can be shown (via exchange argument) never to preclude a global optimum ⇒ **greedy** suffices, and is faster.
- If the locally-best choice *can* rule out the global optimum (must compare multiple choices and their downstream consequences) ⇒ need **DP** (or exhaustive search).
- Classic Tripos test: exhibit a small counterexample where the greedy choice leads to a sub-

optimal result, to justify why DP (not greedy) is needed – e.g. 0/1 knapsack has no valid greedy rule (greedy-by-value-density can be arbitrarily bad), but *fractional* knapsack does.

Practice: y2026 Paper 1 Q8 (DP vs. Greedy – recharging points)

A haulage company has recharging points $A = r_0, r_1, \ldots, r_n = B$ at increasing distances $d_0 = 0, \ldots, d_n$ from A; trucks travel at most distance c per charge (consecutive gaps are $< c$). Minimise the number of recharges needed to get from A to B, and report *which* points are used.
(a)(i) State and explain a DP recurrence for an optimal solution in terms of subproblems. *[5 marks]*
(a)(ii) Devise a DP algorithm; state whether top-down or bottom-up. *[6 marks]*
(a)(iii) Derive the Θ time and space complexity. *[4 marks]*
(b) A friend believes a greedy algorithm solves this. Are they right? If so, describe it; if not, explain why no greedy solution exists. *[5 marks]*

Practice: y2024 Paper 1 Q7 (DP – optimal substructure pitfalls)

For a sequence s (0-indexed, Python slice notation), a *descending subsequence* is one where every item after the first is strictly smaller than its predecessor. Find a maximal-length descending subsequence.
(a) Discuss and *disprove*: "splitting s into two parts, the optimum is the concatenation of the optimal solutions to each part." *[2 marks]*
(b) Disprove: "the optimal solution for $s[:k]$ must start with the optimal solution for $s[:k-1]$, possibly with $s[k-1]$ appended." *[3 marks]*
(c) Give a correct recursive formula for the optimal solution (solving the problem directly), with justification. *[5 marks]*
(d) Explain how to solve it instead by reducing to a well-known DP problem from lectures, with a correct recursive formula for that problem. *[5 marks]*
(e) Give pseudocode for the approach in (d), using top-down memoized DP. *[5 marks]*

Practice: y2019 Paper 1 Q7 (DP: coin change & refuelling)

(a) Maldonia issues stamps of denominations $D \subset \mathbb{N} \setminus \{0\}$ with $1 \in D$. Given n, find a minimum-cardinality multiset of stamps summing to exactly n, via bottom-up DP.
(i) Give and explain a formula expressing the optimal solution in terms of subproblems (also explain how to recover the actual stamps used, not just the count). *[4 marks]*
(ii) Draw and explain the data structure used to accumulate intermediate solutions. *[2 marks]*
(iii) Derive the Θ space and time complexity. *[1 marks]*
(b) Repeat (a)(i)–(iii) for: a car races from A to B, full tank lets it travel distance l; refuelling stations $A = s_0, \ldots, s_n = B$ at distances $d_0 = 0, \ldots, d_n$ (adjacent gaps $< l$). Find a minimum-cardinality set of stations to stop at. *[7 marks]*
(c) Which of the two problems could be solved *more efficiently* with a greedy algorithm instead of DP – and why does that greedy approach fail for the other? *(part (c) continues on the original paper)*

5 Data Structures

5.1 Pointers and Basic Structures

A **pointer** is a reference to a memory location; `NIL` denotes "points to nothing". Objects with pointer fields (e.g. `x.next`, `x.parent`) let us build structures whose size and shape change at runtime, unlike arrays.

5.2 Stacks and Queues

Both are restricted forms of the general **dynamic set**, differing only in *which* element DELETE removes.

Stack – LIFO (Last-In-First-Out)

Supported operations: PUSH(x) (insert), POP() (delete *and return* the most recently inserted element), STACK-EMPTY().

Array implementation: maintain `top` (index of the topmost element).

```
PUSH(S, x)              POP(S)                    STACK-EMPTY(S)
    S.top = S.top + 1       if STACK-EMPTY(S)         return S.top == 0
    S[S.top] = x                error "underflow"
                            else
                                S.top = S.top - 1
                                return S[S.top + 1]
```

All operations $\Theta(1)$. **Overflow** if `S.top` exceeds array capacity – handled by resizing (see amortised analysis, §5.3).

Queue – FIFO (First-In-First-Out)

Supported operations: ENQUEUE(x) (insert at the back), DEQUEUE() (remove/return from the front).

Array implementation: circular buffer with `head` and `tail` pointers (wrap around using mod arithmetic) – avoids the $\Theta(n)$ cost of shifting all elements on dequeue that a naive array implementation would incur. All operations $\Theta(1)$; must track whether the buffer is full vs. empty explicitly (both look like `head == tail`), e.g. via a separate count or by always leaving one slot empty.

Priority Queue

Generalises both: INSERT(x) plus EXTRACT-MIN/EXTRACT-MAX. The standard efficient implementation is a **binary heap** (§5.4): $O(\lg n)$ insert and extract, vs. $O(n)$ for a naive unsorted-array implementation of extract, or $O(n)$ insert for a sorted array.

5.3 Linked Lists

Variants

- **Singly linked:** each node has `key` and `next`; list has a `head` pointer. $O(1)$ insert/delete *at the front* (or given a pointer to the predecessor); $O(n)$ search, and $O(n)$ delete-by-key (must find the predecessor first).
- **Doubly linked:** adds `prev`. $O(1)$ delete *given a pointer to the node itself* (no need to find the predecessor) – the key advantage over singly linked lists.
- **Circular:** the last node's `next` points back to the first (and, if doubly linked, the first's `prev` points to the last) – useful for round-robin structures and avoids special-casing the head/tail.
- **Sentinel** (dummy node): a dummy node `L.nil` simplifies boundary conditions (empty list, insert-at-head) by removing `NIL` checks – a common way to tidy up pseudocode, at the cost of one extra memory cell.

Doubly linked list: INSERT and DELETE

```
LIST-INSERT(L, x)                    LIST-DELETE(L, x)
    x.next = L.head                      if x.prev != NIL
    if L.head != NIL                         x.prev.next = x.next
        L.head.prev = x                  else
    L.head = x                               L.head = x.next
    x.prev = NIL                         if x.next != NIL
                                             x.next.prev = x.prev
```

Both $\Theta(1)$ given a pointer to x (for delete) - this $O(1)$ deletion is the reason doubly linked lists underlie many advanced structures (e.g. LRU caches, deques).

Trap: singly linked list "delete given a pointer"

With only a `next` pointer, deleting node `x` *itself* in $O(1)$ is impossible in general *unless* you use the trick of copying the *next* node's data into `x` and then deleting the next node instead (only works if `x` isn't the last node, and breaks if other pointers reference `x` directly). Examiners like to test whether you spot this doesn't work for the tail.

5.4 Binary Heaps

A (binary) **max-heap** is a nearly-complete binary tree (all levels full except possibly the last, which fills left-to-right) satisfying the **heap property**: $A[\text{Parent}(i)] \geq A[i]$ for every node i (min-heap: reverse inequality). Stored implicitly in an array (1-indexed):

$$\text{Parent}(i) = \lfloor i/2 \rfloor, \qquad \text{Left}(i) = 2i, \qquad \text{Right}(i) = 2i + 1.$$

No explicit pointers needed – the array index encodes tree structure, giving excellent cache behaviour and $O(1)$ navigation.

MAX-HEAPIFY(A, i) – restore the heap property downward from i

```
MAX-HEAPIFY(A, i)
    l = Left(i); r = Right(i)
    largest = i
    if l <= A.heap-size && A[l] > A[largest]: largest = l
    if r <= A.heap-size && A[r] > A[largest]: largest = r
    if largest != i
        exchange A[i] and A[largest]
        MAX-HEAPIFY(A, largest)
```

Complexity of Max-Heapify: $O(\lg n)$ – proportional to the height of the subtree rooted at i, since each recursive call moves one level down and a heap of n nodes has height $\lfloor \lg n \rfloor$.

BUILD-MAX-HEAP(A) – $\Theta(n)$, not $\Theta(n \lg n)$!

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = floor(A.length/2) downto 1
        MAX-HEAPIFY(A, i)
```

Why BUILD-MAX-HEAP is $\Theta(n)$, not $O(n \lg n)$

A loose bound gives $n/2$ calls to Max-Heapify, each $O(\lg n)$, i.e. $O(n \lg n)$ – **but this is not tight**. The key fact: in a heap of n nodes, at most $\lceil n/2^{h+1} \rceil$ nodes have height h, so the total work is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O(n)$$

since $\sum_{h=0}^{\infty} h/2^h = 2$ converges (most nodes are near the bottom, where Max-Heapify does almost no work). **This is one of the most commonly mis-stated complexities in the whole course – know the proof, not just the answer.**

Insert and Extract-Max

Insert: append the new key at the end (position `heap-size+1`), then "bubble up" – repeatedly compare with parent and swap while the heap property is violated. $O(\lg n)$.
Extract-Max: save `A[1]` (the max), move the last element to the root, decrement `heap-size`, then Max-Heapify(A, 1) to restore the property. $O(\lg n)$.
Increase-Key / Decrease-Key: change the key then bubble up or Max-Heapify down as appropriate – $O(\lg n)$; this underlies the priority-queue operations used in Dijkstra's algorithm and Prim's algorithm (Algorithms II).

Sorted array is (usually) *not* a valid heap – a favourite trick question

An array sorted in **ascending** order is generally *not* a max-heap: e.g. $[1, 2, 3]$ has $A[\text{Parent}(3)] = A[1] = 1 < A[3] = 3$, violating the heap property. (It *would* happen to satisfy a *min*-heap property, though — always check which flavour of heap and which order the question specifies.) Conversely, a **descending**-sorted array *is* always a valid max-heap, since every parent (earlier index, hence $\geq$) dominates its children.

5.5 Amortised Analysis (brief)

When a data structure supports a sequence of operations of varying individual cost but bounded *total* cost, **amortised analysis** assigns each operation an average cost over the worst-case sequence, even though no probability is involved.

The Accounting (banker's) method – example: dynamic array doubling

When an array of capacity k is full, allocate a new array of capacity $2k$ and copy all k elements across (cost $\Theta(k)$), then insert. Charging each INSERT \$3 (\$1 for its own insertion, \$2 saved as credit): when a resize of capacity $k \to 2k$ happens, exactly k elements each have \$2 of credit banked from their own insertion, enough to pay for copying all k elements (k credits needed, $2k$ available, covering this and future resizes). Hence *amortised* cost per INSERT is $O(1)$, even though the worst-case cost of a single INSERT (the one that triggers a resize) is $\Theta(n)$.

5.6 Binary Search Trees

A BST is a binary tree where every node x satisfies: all keys in the left subtree $\leq$ `x.key` $\leq$ all keys in the right subtree. **In-order traversal** of a BST visits keys in sorted order.

Core operations

- SEARCH(x, k): compare k with `x.key`, recurse left or right – $O(h)$ where h is the tree height.
- MINIMUM/MAXIMUM: follow `left`/`right` pointers to the end – $O(h)$.
- SUCCESSOR(x): if x has a right subtree, its minimum; else, walk up via `parent` until moving up from a left child – $O(h)$.
- INSERT(T, z): walk down from the root as in SEARCH, following the path a search for `z.key` would take, and attach z as a leaf at the point the search would fail – $O(h)$.
- DELETE(T, z): three cases – (i) z has no children: just remove it; (ii) z has one child: splice it out, linking its parent directly to its child; (iii) z has two children: find z's successor y (the minimum of z's right subtree, which has *at most one* child), replace z's key with y's key, then delete y (which falls into case (i) or (ii)). $O(h)$.

The height problem

All BST operations are $O(h)$, but an adversarial insertion order (e.g. inserting already-sorted data) degenerates a BST into a linked list, with $h = \Theta(n)$ – giving $\Theta(n)$ operations, no better than an unsorted list! **Self-balancing** search trees (red-black trees, splay trees, skip lists, guaranteeing $h = O(\lg n)$) are covered in **Algorithms II, Advanced Data Structures** – know that this is the motivation, even though this course (Algorithms I) covers only plain BSTs and B-trees.

5.7 B-Trees

A B-tree generalises a BST to have many keys and children per node, minimising disk/cache accesses (each node = one disk block). A B-tree of **minimum degree** $t \geq 2$ satisfies:

B-tree properties

- Every node has at most $2t-1$ keys (and $2t$ children); every non-root node has at least $t-1$ keys (at least t children); the root has at least 1 key (if not a leaf).
- A node with k keys has exactly $k+1$ children, which partition the key space: children between consecutive keys hold values strictly between them.
- All leaves appear at the **same depth** – the tree is perfectly height-balanced.
- Height $h = O(\log_t n)$ – for large t (matched to disk block size), this is a very shallow tree even for huge n, minimising the number of (slow) disk accesses.

Search, Insert, Delete

Search: at each node, binary-search the (sorted) keys to find the correct child to recurse into (or find the key itself); $O(t \lg t)$ per node (or $O(t)$ with linear scan) $\times$ $O(\log_t n)$ levels.
Insert: descend as in search; if a full node ($2t-1$ keys) is encountered along the way, **split it** preemptively (median key moves up to the parent, the two halves become separate children) – ensures the parent always has room for the split before you recurse into it. Insert is thus a single downward pass, $O(t \log_t n)$.
Delete: more delicate – ensure any node you recurse into has *at least* t keys before descending (by merging with a sibling, or "borrowing" a key from a sibling via the parent, if it has only $t-1$). $O(t \log_t n)$.

Why B-trees, not plain BSTs, for disk-backed data

The point of a B-tree is amortising the *fixed* cost of a disk seek across many keys per node: a BST with n keys has height $\Theta(\lg n)$, i.e. $\Theta(\lg n)$ disk accesses per operation; a B-tree with branching factor t has height $\Theta(\log_t n) = \Theta(\lg n / \lg t)$ – for t in the hundreds or thousands (chosen to fill a disk block), this can be 3–4 accesses even for billions of keys, versus 30+ for a BST. **Always frame B-trees as an I/O-cost optimisation, not (only) a comparison-count optimisation**, when asked to justify their use.

Practice: y2009 Paper 1 Q6 (Heaps – heapify, extract, complexity)

(a) State the defining properties of a min-heap; show the tree $\leftrightarrow$ zero-based array conversion. *[3 marks]*
(b) "An ascending-sorted array is always a min-heap." True or false – prove or give a counterexample. *[3 marks]*
(c) Given $A = [A, I, E, R, P, M, S, N, L]$ (not a min-heap), redraw as a tree and turn it into a heap using $O(n)$ HEAPIFY, showing intermediate stages with explanation. *[4 marks]*
(d) Perform EXTRACT-MIN on the resulting heap, with intermediate stages. *[3 marks]*
(e) Asymptotic running time of heapsort on an already-ascending-sorted array of length n? Justify. *[3 marks]*
(f) Asymptotic running time of heapsort on an already-descending-sorted array? Justify. *[4 marks]*

Practice: y2008 Paper 1 Q4 (Priority queues & heaps)

(a) Briefly describe the operations supported by a priority queue. *[2 marks]*
(b) Explain the heap data structure and how it can be implemented as a simple linear block of memory. *[2 marks]*
(c) Describe, and estimate the cost of: (i) finding the parent/offspring of a given node; (ii) inserting a new item; (iii) deleting the topmost item from a non-empty heap. *[6 marks]*

Practice: y2010 Paper 1 Q6 (Divide-and-conquer, matrix multiplication)

$Z = AB$ for $n \times n$ matrices, $Z_{ij} = \sum_k A_{ik} B_{kj}$.
(a) Find the time complexity of the direct implementation of this formula, with justification. *[3 marks]*
(b) An alternative strategy divides A, B into four $\frac{n}{2} \times \frac{n}{2}$ blocks; computing Z this way needs eight block multiplications and four additions – derive and solve the resulting recurrence for the time complexity, and compare with (a). *(continues in original paper, including Strassen's algorithm)*

6 Hashing

A hash table supports INSERT, SEARCH, DELETE for a dynamic set, in **expected** $O(1)$ time (under reasonable assumptions), by mapping keys to array slots via a **hash function** $h : U \to \{0, 1, \ldots, m-1\}$ (m = table size). Hash tables are conventionally **0-indexed**.

Collisions are inevitable

By the pigeonhole principle, if $|U| > m$, some two keys must hash to the same slot – a **collision**. All hash table designs are fundamentally about collision resolution.

6.1 Hash Functions

Desirable properties & common constructions

Simple uniform hashing assumption (SUHA): each key is equally likely to hash to any of the m slots, independently – the standard assumption for analysis (rarely provable for a specific h, but a reasonable design goal).

- **Division method:** $h(k) = k \bmod m$. Simple and fast, but choose m carefully: m a power of 2 makes $h(k)$ depend only on the low-order bits of k (bad if keys share low bits); m **prime, not close to a power of 2**, is usually preferred.
- **Multiplication method:** $h(k) = \lfloor m\,(kA \bmod 1) \rfloor$ for constant $0 < A < 1$ – value of m less critical; a good choice of A (Knuth suggests $A \approx (\sqrt{5}-1)/2$) spreads keys well regardless of m.

6.2 Chaining

Each slot $T[i]$ is the head of a linked list of all keys hashing to i.

Chained hash table operations

```
CHAINED-HASH-INSERT(T, x)      insert x at the head of list T[h(x.key)]     -- O(1)
CHAINED-HASH-SEARCH(T, k)      search list T[h(k)] for an element with key k -- O(1 + alpha)
CHAINED-HASH-DELETE(T, x)      delete x from its list                       -- O(1) if doubly linked
```

Load factor $\alpha = n/m$ (average chain length under SUHA). Expected search time (successful or unsuccessful) is $\Theta(1+\alpha)$; keeping $\alpha = O(1)$ (e.g. by resizing when α exceeds a threshold, amortised as in §5.3) gives expected $O(1)$ operations.

Note: a chained table has stack-like behaviour per key

Inserting at the *head* of the chain means that if a key is deleted and then re-inserted (or if a key is overwritten by a later insert of the same key before its old binding is deleted), the *most recent* binding is found first – worth noting if a question asks about repeated insertions of the same key without intervening deletes.

6.3 Open Addressing

All elements are stored directly in the table itself (no chains); on collision, **probe** a sequence of slots $h(k,0), h(k,1), h(k,2), \ldots$ until an empty one is found. Requires $\alpha \leq 1$ always ($n \leq m$).

Probing strategies

- **Linear probing:** $h(k,i) = (h'(k) + i) \bmod m$. Simple, cache-friendly, but suffers **primary clustering** (long runs of occupied slots grow, and are increasingly likely to grow further).
- **Quadratic probing:** $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$. Reduces primary clustering, but two keys with the same initial probe suffer **secondary clustering** (identical subsequent probe sequences).

- **Double hashing:** $h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$, with $h_2(k)$ never 0 and coprime to m (e.g. m prime, $h_2(k) \in \{1,\ldots,m-1\}$). Gives $\Theta(m^2)$ possible probe sequences (vs. $\Theta(m)$ for linear/quadratic), closely approximating true random (uniform hashing) behaviour.

Power-of-two table size and full coverage of quadratic probing

For quadratic probing $h(k,i) = (h(k) + i + i^2) \bmod m$ (a common simplified form) to visit *every* slot before repeating (i.e. to guarantee it can always find an empty slot if one exists), a standard sufficient condition is m a **power of 2**. This is useful because it guarantees the probe sequence doesn't get "stuck" cycling through only some slots while others remain empty – a frequent past-paper part (d)-type question.

Deletion in open addressing – the tombstone problem

Naively marking a slot `EMPTY` on delete **breaks search**: search for k walks the probe sequence until it hits an `EMPTY` slot, but if a later key was inserted *past* a slot that has since been emptied, search would wrongly stop early and report "not found".
Fix: mark deleted slots as `DELETED` (a "tombstone") rather than `EMPTY`. SEARCH treats `DELETED` as occupied (keeps probing past it); INSERT treats `DELETED` as available (can reuse it).

Open addressing with tombstones – pseudocode sketch

```
HASH-DELETE(T, k)
    i = 0
    repeat
        j = h(k, i)
        if T[j].key == k
            T[j] = DELETED
            return
        i = i + 1
    until T[j] == EMPTY || i == m
    // not found

HASH-SEARCH(T, k)
    i = 0
    repeat
        j = h(k, i)
        if T[j] != DELETED && T[j].key == k
            return j
        i = i + 1
    until T[j] == EMPTY || i == m
    return NOT-FOUND
```

6.4 Comparing Chaining vs. Open Addressing

	Chaining	Open Addressing
Load factor	$\alpha = n/m$, can exceed 1	$\alpha = n/m \leq 1$ always
Extra memory	Pointers per node	None (in-place)
Cache behaviour	Poor (pointer chasing)	Good (probes are array-local)
Deletion	Simple (unlink node)	Needs tombstones
Degrades under load	Gracefully (longer chains)	Sharply as $\alpha \to 1$

Practice: y2025 Paper 1 Q8 (Hashing – chaining and open addressing)

(a) Insert hash keys $3, 37, 16, 75, 27, 4, 8, 84$ (in order) into an initially empty hash table of size 10, chaining, $h(x) = x \bmod 10$, using a chaining discipline that ensures present and absent searches take equal average time. Draw the final structure. *[5 marks]*

(b) Insert the same keys into a table of size 16, open addressing, $h(x) = x \bmod 16$, quadratic probe $h(x, i) = \left(h(x) + \frac{i+i^2}{2}\right) \bmod 16$. Draw the final structure. *[6 marks]*

(c) Support efficient deletion from an open-addressed table of size `T.length` with the same quadratic probe. (i) Pseudocode for DELETE. (ii) Pseudocode for SEARCH. *[7 marks]*

(d) The probe sequence in (b) visits every table position before repeating, provided the table size is a power of 2. Explain why this is useful. *[2 marks]*

Practice: y2017 Paper 1 Q8 (Hash functions & quadratic probing)

(a) Briefly explain hash functions, hash tables and collisions. *[3 marks]*

(b) Explain the open-addressing collision resolution strategy and the term "probing sequence". *[3 marks]*

(c) Explain quadratic probing and its advantages/disadvantages (refer to primary and secondary clustering). *[3 marks]*

(d) Give a general mathematical expression for the probing function $p(k, i)$ used in quadratic probing, yielding a 0-based index, in terms of k, i, h, table size m, and constants c_1, c_2. *[3 marks]*

(e) Given a piece of pseudocode implementing a `get(k)` function, determine whether it implements a form of quadratic probing; if so derive c_1, c_2; if not, prove it doesn't. *[8 marks]*

7 Further Practice & Exam Strategy

7.1 Past Paper Index by Topic (2014–2026 era, current 2-question format)

Since 2014, Paper 1 sets exactly one Algorithms I question from a choice of two each year. The table below indexes recent papers by primary topic so you can drill a topic exhaustively; question text is in your source archive (`Algorithms_I_Complete.md`) under the heading `Past Paper: yYYYYp1qN`.

Topic	Papers
Recurrences / asymptotics	y2014p1q7, y2016p1q10, y2018p1q10 (also embedded within most q7's above), y2025p1q7, y2026p1q7
Sorting (comparison-based)	y2014p1q8, y2017p1q7, y2020p1q10
Heaps / priority queues	y2016p1q9, y2017p1q10, y2018p1q10
Hashing	y2017p1q8, y2018p1q9, y2025p1q8
Dynamic programming	y2015p1q8, y2019p1q7, y2023p1q7, y2024p1q7, y2026p1q8
Greedy algorithms	y2015p1q8, y2026p1q8
Stacks/queues/lists (design & complexity)	y2011p1q5, y2016p1q7, y2019p1q9, y2024p1q8
Mixed short-answer (true/false style)	y2007p1q4, y2022p1q7

Pre-2014 papers: check scope carefully

Papers from 2006–2013 predate the current Algorithms I/II split and the current 12+12 lecture structure. Several of their topics (**red-black trees, splay trees, skip lists, 2-3-4 trees**) are now taught under **Algorithms II – Advanced Data Structures**, not Algorithms I. Use pre-2014 sorting/heap/complexity questions freely, but treat pre-2014 "advanced tree" questions as Algorithms II practice instead (see your Algorithms II guide).

7.2 General Exam Technique

Checklist for every Algorithms I answer

- **State your indexing convention** (1- or 0-based) explicitly if it isn't fixed by the question.
- **Number your pseudocode lines** – makes it trivial for the examiner (and you) to reference specific steps in a correctness/complexity argument.
- For *any* "is it correct?" or "prove optimal substructure" question, look for a **small counterexample** first – many marks are earned or lost entirely on whether you spot that a plausible-sounding claim is false.
- For complexity questions, always distinguish **worst** / **best** / **average** case, and state whether you're giving O, Ω, or Θ – read the question carefully for which is being asked.
- For recurrences: identify the method (substitution / recursion tree / Master Theorem) explicitly, show the comparison of $f(n)$ against $n^{\log_b a}$ if using the Master Theorem, and justify any claim that floors/ceilings/additive shifts don't change the asymptotics.
- For DP: state the subproblem definition, the recurrence, the base case(s), and the complexity (in terms of #subproblems $\times$ cost per subproblem) – all four are usually separately credited.
- For greedy: state the greedy rule, then sketch the **exchange argument** (not just "it's obviously optimal").
- Sanity-check pseudocode with tricky inputs before submitting: single-element arrays, already-sorted/reverse-sorted, duplicate keys, empty structures.