Algorithms II

Computer Science Tripos – Part IA

Comprehensive Tripos Revision Guide

University of Cambridge – Paper 1

Section	Topics	Key Skills
1. Graphs & Path-Finding	Representations, BFS, DFS, edge classes, SCCs, SSSP, APSP	Prove Convergence Lemma / SSSP correctness; pick the right SSSP/APSP algorithm and cost it
2. Subgraphs & Flow	Flow networks, residual graphs, Ford-Fulkerson, matchings, MSTs	Apply Max-Flow Min-Cut; construct augmenting-path arguments; prove Safe Edge Theorem
3. Advanced Data Structures	Amortised analysis, binomial/Fibonacci heaps, disjoint sets	Choose aggregate/accounting/potential method; derive $O(1)$ amortised bounds; use $\Phi = r + 2m$
4. Geometric Algorithms	Point-in-polygon, cross products, line intersection, convex hull	Avoid trigonometry with cross products; run Graham's Scan / Jarvis's March by hand

Compiled from the 2025/26 Algorithms II lecture course (Dr J. Fawcett), the 2023/24 course (Dr D. Wischik), Example Sheets 4–6, and cross-referenced against the Algorithms I revision guide.

Contents

1 Graphs and Path-Finding Algorithms

1.1 Graphs: Definitions and Representations

A **graph** $G = (V, E)$ is a set of vertices V and edges $E \subseteq V \times V$. In an **undirected** graph edges are unordered pairs (equivalently, E is symmetric); in a **directed** graph edges are ordered pairs. A **weighted** graph has a weight function $w : E \to \mathbb{R}$. A graph is **fully connected** (complete) if $E = V \times V$.

Adjacency Matrices vs. Adjacency Lists

- **Adjacency matrix** M: a $|V| \times |V|$ matrix, $\Theta(|V|^2)$ space. $M_{u,v} = 1$ (or the edge weight) if $(u, v) \in E$, else 0. Undirected graphs need only the upper (or lower) triangle.
- **Adjacency list**: array of length $|V|$; $A[u]$ points to a linked list of v such that $(u, v) \in E$ (with weights as tuples (v, w) if weighted).

	Adjacency Matrix	**Adjacency List**
Space	$\Theta(\|V\|^2)$ (compact if *dense*)	compact if *sparse*
Is $(u, v) \in E$?	$O(1)$	$O(\|\text{neighbours}\|)$ worst case
List neighbours of u	$O(\|V\|)$	$O(\text{num. neighbours})$
Iterate all edges	$O(\|V\|^2)$	$O(\|E\|)$
Undirected storage	can (roughly) halve	cannot halve without hurting time

Terminology checklist – know these cold

- **Transpose** $G^T = (V, E^T)$: all directed edges reversed.
- **In-/out-degree**: number of incoming/outgoing edges at a vertex (directed graphs); **degree** = number of incident edges (undirected).
- **Square** $G^2 = (V, E^2)$: edge (u, v) present iff there's a $\leq$ 2-edge path $u \to v$ in G.
- Two **edges** are **adjacent** if they share a vertex.
- **Induced subgraph** $G' = (V', E')$: $V' \subseteq V$, E' = all edges of E with both endpoints in V'.
- **Clique**: an induced subgraph that is complete.
- **Complement** $\bar{G} = (V, \bar{E})$ where $\bar{E} = \{(u, v) \mid u, v \in V, (u, v) \notin E\}$.
- A graph is **acyclic** if no vertex can be reached by a path from itself.
- **Vertex/edge/face colouring**: assign colours so that no two adjacent vertices/edges/faces (of a planar graph) share a colour. A **planar** graph can be drawn with no edge crossings; a **face** is a region bounded by edges (including the unbounded outer region).

1.2 Breadth-First Search

BFS explores a graph level-by-level from a source s, using a queue. On graphs (unlike trees) we must guard against enqueuing a vertex more than once.

BFS(G, s) – correct version for graphs

```
1  for v in G.V
2      v.marked = false
3  Q = new Queue
4  ENQUEUE(Q, s); s.marked = true
5  while !QUEUE-EMPTY(Q)
6      u = DEQUEUE(Q)
7      for v in G.E.adj[u]
8          if !v.marked
9              v.marked = true
10             v.d = u.d + 1;  v.pi = u
11             ENQUEUE(Q, v)
```

Marking a vertex *when it is enqueued* (not when dequeued) is what avoids the inefficiency of enqueuing the same vertex multiple times. Running time is $\Theta(|V| + |E|)$: every vertex is enqueued/dequeued once and every edge is examined at most once (twice, for undirected graphs).

Correctness of BFS as an SSSP algorithm for unweighted graphs

Let $\delta(s, v)$ be the true shortest (fewest-edges) distance from s to v (∞ if unreachable).

- **Lemma 1.** If $(u, v) \in E$ then $\delta(s, v) \leq \delta(s, u) + 1$. *(If u unreachable, trivial; otherwise the direct edge from u gives a path no longer than $\delta(s, u) + 1$.)*
- **Lemma 2.** On termination $v.d \geq \delta(s, v)$ for all v – proved by induction on the number of ENQUEUE operations, using Lemma 1 in the inductive step ($v.d = u.d + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$).
- **Lemma 3 (queue shape invariant ϕ).** If $Q = v_1, \ldots, v_x$ (head to tail) then $v_x.d \leq v_1.d+1$ and $v_i.d \leq v_{i+1}.d$. This is preserved by both ENQUEUE and DEQUEUE (check each case). **Corollary:** if v_a is enqueued before v_b, then $v_a.d \leq v_b.d$.
- **Main proof (by contradiction).** Suppose some vertex gets an incorrect d; let v be such a vertex with *minimum* $\delta(s, v)$. Let u be v's predecessor on a true shortest path, so $\delta(s, u) = u.d$ (as u has strictly smaller δ, it cannot be the first mistake). When u is dequeued, v is in one of three states (not yet enqueued / enqueued not dequeued / already dequeued); in *all three* cases the queue-shape corollary forces $v.d \leq u.d + 1$, contradicting $v.d > u.d + 1$ (which followed from $v.d > \delta(s, v) = \delta(s, u) + 1$). No first mistake can exist, so BFS is correct.

1.3 Depth-First Search

DFS explores as deep as possible before backtracking (stack, or the call stack via recursion). Unlike BFS, DFS is often run with *no* fixed source: repeat from any unvisited vertex until all are visited, producing a **forest**.

DFS(G) / DFS-Helper(G, u)

```
DFS(G)                          DFS-HELPER(G, u)
1  for v in G.V                 1  time = time + 1
2      v.marked = false         2  u.discover_time = time
3      v.pi = NIL               3  u.marked = true
4  time = 0                     4  for v in G.E.adj[u]
5  for s in G.V                 5      if !v.marked
6      if !s.marked              6         v.pi = u
7          DFS-HELPER(G, s)     7         DFS-HELPER(G, v)
                                 8  time = time + 1
                                 9  u.finish_time = time
```

Running time $\Theta(|V| + |E|)$ – every vertex and edge is examined a constant number of times. v is a **descendant** of u in the DF-forest iff u.discover $<$ v.discover $<$ v.finish $<$ u.finish (the *parenthesis structure* of discover/finish intervals).

Edge classification (know the discover/finish characterisations!)

- **Tree edge**: (u, v) where v was first discovered via this edge.
- **Back edge** (directed graphs): connects u to an *ancestor* v – indicates a cycle. Characterised by v.disc $\leq$ u.disc $<$ u.fin $\leq$ v.fin.
- **Forward edge**: connects u to a proper *descendant* v, not a tree edge.
- **Cross edge**: everything else – runs between subtrees of the same DF-tree (neither ancestor of the other) or between different DF-trees. Characterised by v.disc $<$ v.fin $<$ u.disc $<$ u.fin.

Tree/forward edges are characterised by u.disc $<$ v.disc $<$ v.fin $<$ u.fin. **Every edge in an undirected graph is a tree edge or a back edge** (no forward/cross edges are possible).

Classifications are not mutually exclusive labels of a fixed partition in general graphs, but for a given DFS run each edge gets exactly one of these four labels.

Two famous corollaries of DFS

- On an **undirected** graph, the number of calls to DFS-Helper from the main loop = the number of **connected components**.
- On a **directed acyclic graph** (DAG), sorting vertices by *descending finish time* gives a **topological sort**.

1.4 Strongly Connected Components

Input: directed graph $G = (V, E)$. **Output:** a partition of V into maximal sets C such that every pair $u, v \in C$ has paths $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

SCC(G) – Kosaraju's algorithm

```
1  Run DFS(G), recording finish times for each vertex
2  Compute G^T (transpose: reverse every edge)
3  Run DFS(G^T), but in the main loop consider vertices
   in ORDER OF DECREASING finish time from step 1
4  Each depth-first tree produced in step 3 is one SCC
```

Running time $\Theta(|V| + |E|)$ (two DFS passes plus computing G^T, each linear). **Idea**: the "component graph" (SCCs contracted to single vertices) is always a DAG; running DFS on G^T in decreasing finish-time order from G visits SCCs in a topological order of that component DAG, so each new DFS tree can only re-discover vertices in its own SCC.

Practice: Example Sheet 4, Q1 – DAGs and trees

Read the definitions of directed acyclic graph and tree. Draw an example of each of the following, or explain why no example exists: (i) a DAG with 8 vertices and 10 edges; (ii) an undirected tree with 8 vertices and 10 edges; (iii) a graph without cycles that is *not* a tree.

1.5 Shortest Path Problems

Given a weighted, directed graph $G = (V, E, w)$, a path p has weight $w(p) = \sum$ of its edge weights; $\delta(u, v)$ is the minimum weight of any path $u \to v$ (∞ if none). Variants: **single-source** (one s, all destinations), **single-destination** (all sources, one t – solve SSSP on G^T), **single-pair** ($u \to v$ – no asymptotically better algorithm than SSSP is known), **all-pairs**.

What makes shortest paths hard

- **Negative-weight edges** complicate relaxation-based algorithms; **negative-weight cycles** mean no shortest path exists at all (you can loop forever, decreasing cost).
- A shortest path can never contain a positive-weight cycle. A path containing a zero-weight cycle is never the *unique* shortest path (removing the cycle doesn't increase weight), so w.l.o.g. we can always find a shortest path that is simple (acyclic).

Optimal substructure: sub-paths of shortest paths are themselves shortest paths (if $u \rightsquigarrow v_i \rightsquigarrow v_j \rightsquigarrow v$ is a shortest $u \to v$ path, then $v_i \rightsquigarrow v_j$ must be a shortest $v_i \to v_j$ path, else splicing in a shorter one would shorten the whole path – contradiction). This licenses both DP (Floyd-Warshall) and greedy (Dijkstra) approaches.

Bellman-Ford

Handles **negative edge weights** and detects negative cycles; $O(|V||E|)$.

Bellman-Ford(G, w, s)

```
RELAX(u, v, w)
1  if v.d > u.d + w(u,v)
2      v.d = u.d + w(u,v)
3      v.pi = u
```

```
BELLMAN-FORD(G, w, s)
1  for v in G.V:  v.d=inf; v.pi=NIL
2  s.d = 0
3  for i = 1 to |G.V|-1
4      for (u,v) in G.E:  RELAX(u,v,w)
5  for (u,v) in G.E
6      if v.d > u.d + w(u,v):  return false
7  return true
```

Returns `false` (negative cycle reachable from s) or `true` with correct `d`/`pi` attributes. Cost: initialisation $\Theta(|V|)$, the double loop $\Theta(|V||E|)$, final check $\Theta(|E|)$: overall $O(|V||E|)$.

Dijkstra's Algorithm

Greedy SSSP algorithm exploiting optimal substructure; requires **non-negative** weights $w(u, v) \geq 0$; cheaper than Bellman-Ford at the cost of that restriction.

Dijkstra(G, w, s)

```
1  for v in G.V:  v.d = inf;  v.pi = NIL
2  s.d = 0
3  S = EMPTY-SET
4  Q = new PriorityQueue(G.V)          // keyed on 'd'
5  while !PQ-EMPTY(Q)
6      u = PQ-EXTRACT-MIN(Q)
7      S = S union {u}
8      for v in G.E.adj[u]
9          RELAX(u, v, w)              // implicitly calls DECREASE-KEY
```

Correctness of Dijkstra – the Convergence Lemma

Convergence property of RELAX(i, j): if $s \rightsquigarrow i \to j$ is a shortest path and $i.d = \delta(s, i)$ *before* (i, j) is relaxed, then $j.d = \delta(s, j)$ *afterwards.*
Proof: after relaxing, $j.d \leq i.d + w(i, j) = \delta(s, i) + w(i, j) = \delta(s, j)$; and since d never underestimates δ, $j.d = \delta(s, j)$.
Main proof (invariant ϕ: "$u.d = \delta(s, u)$ whenever u is added to S", by contradiction): let u be the *first* vertex added to S with $u.d \neq \delta(s, u)$. There is a shortest path $s \rightsquigarrow u$; split it at the first vertex $y \notin S$ on that path as $s \rightsquigarrow x \to y \rightsquigarrow u$ ($x \in S$). Since u is the first mistake, $x.d = \delta(s, x)$ before x was added to S, and edge (x, y) was relaxed at that time, so by the Convergence Property $y.d = \delta(s, y)$ (and stays so, since d only decreases and can't go below δ). Because weights are non-negative and y precedes u on the shortest path, $\delta(s, y) \leq \delta(s, u)$. Combined with $u.d \leq y.d$ (both were still in the queue when u was extracted as the minimum), we get $y.d = \delta(s, y) = \delta(s, u) = u.d$ – contradicting $u.d \neq \delta(s, u)$. So no first mistake exists, and Dijkstra is correct. On termination $Q = \varnothing$ so $S = V$ and $v.d = \delta(s, v)$ for all v.

Cost of Dijkstra by priority-queue implementation

PQ implementation	Extract-Min $\times\|V\|$ / Decrease-Key $\times\|E\|$	Total
Array / hash table	$O(\|V\|)$ / $O(1)$	$O(\|V\|^2 + \|E\|)$
Binary min-heap	$O(\lg \|V\|)$ / $O(\lg \|V\|)$	$O((\|V\| + \|E\|) \lg \|V\|)$
Fibonacci heap	$O(\lg \|V\|)$ / $O(1)$ amortised	$O(\|V\| \lg \|V\| + \|E\|)$

Array implementation wins for *dense* graphs ($|E| = \Theta(|V|^2)$); Fibonacci heap wins asymptotically

for *sparse* graphs – this is exactly why Fibonacci heaps matter for Dijkstra/Prim (§3.3).

Common trap: why does Dijkstra need non-negative weights?

The proof relies on $\delta(s,y) \le \delta(s,u)$ because y is "closer" on the path – this needs weights to be non-negative so that extending a path cannot *decrease* its length. With negative edges, a vertex extracted early (locally cheapest) might later be beaten by a path through a heavier-looking prefix followed by a very negative edge – a classic past-paper counterexample to memorise or reconstruct on the fly.

All-Pairs Shortest Paths

Input: weighted digraph G. **Output:** $|V| \times |V|$ matrix $D = (d_{ij})$, $d_{ij} = \delta(i,j)$.

Comparing APSP strategies

- **Repeat Bellman-Ford** from every source: $O(|V|^2|E|)$ – handles negative edges, $O(|V|^4)$ if dense.
- **Repeat Dijkstra** from every source (needs $w \ge 0$): $O(|V|(|V|+|E|)\lg|V|)$ with a binary heap.
- **Matrix / repeated squaring**: interpret matrix "multiplication" with scalar $+ \to$MIN, scalar $\times \to +$; $(G.E.M)^{\times x}$ (min-plus power) gives all shortest paths using $\le x$ hops once $x \ge |V|-1$. Repeated squaring: $O(|V|^3 \lg|V|)$.
- **Floyd-Warshall** (DP): $O(|V|^3)$ – best for *dense* graphs, simplest to implement, handles negative edges (not negative cycles).
- **Johnson's algorithm** (reweighting + Dijkstra): $O(|V|^2 \lg|V| + |V||E|)$ – best for *sparse* graphs; handles negative edges and detects negative cycles.

Floyd-Warshall(G,w)

```
1  D(0) = w                              // d_ij^(0) = w(i,j), or inf/0 as appropriate
2  for k = 1 to |G.V|
3      let D(k) = new matrix
4      for i = 1 to |G.V|
5          for j = 1 to |G.V|
6              d_ij^(k) = min( d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1) )
7  return D(|G.V|)
```

Optimal substructure (DP argument): let $d_{ij}^{(k)}$ = weight of a shortest $i \to j$ path using only intermediate vertices from $\{1,\dots,k\}$. Either the optimal such path avoids k as an intermediate (so $d_{ij}^{(k)} = d_{ij}^{(k-1)}$), or it passes through k exactly once (so $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$, since the two halves can't revisit k without creating a cycle on a shortest path). Taking the MIN of both options gives the recurrence above. **Extensions:** keep a predecessor matrix $\Pi^{(k)}$ in parallel to reconstruct paths; to compute the **transitive closure** E^*, run the same DP with $w(i,j) = 1$ for edges (Boolean OR/AND in place of MIN/+).

Johnson's Algorithm: reweighting

Goal: find new weights $\hat{w}(u,v) = w(u,v) + h(u) - h(v)$ that are (1) non-negative on every edge and (2) preserve which paths are shortest.

1. Add a new vertex q with zero-weight edges to every vertex; run BELLMAN-FORD(G', w, q) to get $h(v) = \delta(q,v)$ for every v (detects negative cycles here).
2. Reweight: $\hat{w}(u,v) = w(u,v) + h(u) - h(v) \ge 0$ for every edge, because $h(v) \le h(u) + w(u,v)$

by the triangle-inequality property of shortest-path distances.

3. Any path $u \rightsquigarrow v$ has $\hat{w}(p) = w(p) + h(u) - h(v)$ – a *constant shift* depending only on endpoints, so the *same* path minimises both w and $\hat{w}$.
4. Run Dijkstra with weights $\hat{w}$ from every vertex, then recover $\delta(u, v) = \hat{\delta}(u, v) - h(u) + h(v)$.

Total cost: $O(|V||E|)$ for Bellman-Ford + $O(|V|)$ runs of Dijkstra at $O((|V| + |E|) \lg |V|)$ each $= O(|V|^2 \lg |V| + |V||E|)$.

Practice: Example Sheet 4 – SSSP/APSP style questions

(a) Trace Dijkstra's algorithm by hand on a small weighted digraph, showing the priority queue contents after each extraction.
(b) Give a graph with a negative-weight edge on which Dijkstra produces an incorrect result, and explain precisely which step of the correctness proof fails.
(c) Explain why Johnson's algorithm adds a new vertex q connected to every other vertex by zero-weight edges, rather than simply running Bellman-Ford from an existing vertex.

1.6 Section 1 Summary Checklist

Before moving on, make sure you can. . .

- State and use the discover/finish-time characterisation of all four DFS edge classes.
- Reconstruct Kosaraju's SCC algorithm and explain *why* the second DFS must run in decreasing finish-time order.
- State the Convergence Lemma precisely and use it as the crux step of a Dijkstra correctness proof.
- Cost Dijkstra/Bellman-Ford/Floyd-Warshall/Johnson's under different PQ implementations and graph densities, and pick the cheapest for given $|V|, |E|$.
- Explain reweighting and why it preserves shortest paths (constant additive shift per path).

2 Graphs and Subgraphs: Flow Networks, Matchings, and MSTs

2.1 Flow Networks

A **flow network** $G = (V, E)$ is weighted and directed, with distinguished **source** s and **sink** t. We require: no self-loops; no **antiparallel edges** ($\forall u, v.\ (u, v) \in E \Rightarrow (v, u) \notin E$); non-negative **capacities** $c(u, v) \geq 0$ (define $c(u, v) = 0$ if $(u, v) \notin E$).

Handling antiparallel edges and multiple sources/sinks

Antiparallel edges (u, v) and (v, u): cannot simply net them off (might need only the smaller-magnitude direction's capacity, or all of the larger). Fix: split one edge with an extra vertex, $u \to v' \to v$, both new edges keeping the original capacity.
Multiple sources $s_1..s_m$ / **sinks** $t_1..t_n$: add a **supersource** s and **supersink** t, with ∞-capacity edges (s, s_i) and (t_j, t). This reduces to the single-source single-sink problem without loss of generality.

A **flow** $f : V \times V \to \mathbb{R}$ satisfies: **capacity constraint** $0 \leq f(u, v) \leq c(u, v)$; **flow conservation** at every $u \in V \setminus \{s, t\}$: $\sum_v f(v, u) = \sum_v f(u, v)$. The **value** $|f| = \sum_v f(s, v) - \sum_v f(v, s)$ (*not* absolute value or cardinality!).

Residual networks, augmenting paths, and cuts

Residual capacity: $c_f(u, v) = c(u, v) - f(u, v)$ if $(u, v) \in E$; $= f(v, u)$ if $(v, u) \in E$; else 0. (Exactly one case applies, by the no-antiparallel-edges rule.) The **residual network** G_f has an edge (u, v) wherever $c_f(u, v) > 0$ – this can include edges not in E and can be antiparallel.
Augmenting path: a simple $s \to t$ path p in G_f. Its **bottleneck** is $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$; augmenting f by $c_f(p)$ along p strictly increases $|f|$.
Cut (S, T): partition of V with $s \in S, t \in T = V \setminus S$. **Net flow** across the cut $f(S, T) = \sum_{u \in S, v \in T} f(u, v) - \sum_{u \in S, v \in T} f(v, u)$ always equals $|f|$ (from conservation). **Capacity** of the cut $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$. Any flow's value is $\leq$ any cut's capacity.

Ford-Fulkerson(G, s, t)

```
1  for (u,v) in G.E:  (u,v).f = 0
2  while there exists a path p from s to t in G_f
3      c_f(p) = min{ c_f(u,v) : (u,v) in p }
4      for (u,v) in p
5          if (u,v) in G.E:   (u,v).f += c_f(p)
6          else:              (v,u).f -= c_f(p)
7  return f
```

The Max-Flow Min-Cut Theorem

For a flow f in G, the following are equivalent: **(1)** f is a maximum flow; **(2)** G_f has no augmenting path; **(3)** $|f| = c(S, T)$ for some cut (S, T).

- **(1)⇒(2):** if G_f had an augmenting path p, augmenting would give $|f| + |f_p| > |f|$ (as $|f_p| > 0$), contradicting maximality.
- **(3)⇒(1):** $|f| \leq c(S, T)$ always; if equality holds, f cannot be beaten, so it's maximum.
- **(2)⇒(3):** if G_f has no $s \to t$ path, let $S = \{v : \exists \text{ path } s \to v \text{ in } G_f\}$, $T = V \setminus S$. For $u \in S, v \in T$: if $(u, v) \in E$ then $f(u, v) = c(u, v)$ (else it'd be a residual edge, pulling v into S); if $(v, u) \in E$ then $f(v, u) = 0$ (else the reverse residual edge would pull v into S). Hence $f(S, T) = \sum c(u, v) - 0 = c(S, T)$, and $|f| = f(S, T) = c(S, T)$.

This 3-way cycle of implications proves all three equivalent.

Termination and complexity

With **irrational** capacities, Ford-Fulkerson can fail to terminate (an infinite non-convergent series of ever-smaller augmentations) – a classic short-answer trap. With **integer** capacities, each augmentation increases $|f|$ by ≥ 1, so it terminates in $O(|E|\,|f^*|)$ time (finding each path costs $O(|E|)$ via BFS/DFS). **Edmonds-Karp** fixes this by always choosing the *shortest* (fewest-edges) augmenting path via BFS, giving $O(|V||E|^2)$ regardless of capacities.

2.2 Maximum Bipartite Matching

A **matching** $M \subseteq E$ has ≤ 1 edge incident at each vertex. **Bipartite**: $V = V_1 \cup V_2$, $E \subseteq V_1 \times V_2$. **Maximum** matching has max cardinality (contrast **maximal**: cannot be *extended*, but may not be maximum!).

Reduction to Max-Flow

Add s, t; edges (s, u) for $u \in V_1$ and (v, t) for $v \in V_2$, all with capacity 1, and set every original edge's capacity to 1. An integer-valued max flow corresponds to a maximum matching (the **Integrality Theorem** guarantees an integral optimal flow exists when capacities are integers). Cost: $O(|V||E|)$ via Ford-Fulkerson (each augmentation adds exactly 1 unit of flow, and there are $\leq |V|/2$ of them).

An **augmenting path** w.r.t. M is an **alternating path** (edges alternate in/out of M) that starts and ends at *unmatched* vertices. $M \oplus a$ (symmetric difference with an augmenting path a) strictly grows the matching.

Why augmenting paths characterise maximum matchings

Let M' be a maximum matching, M the current matching. Consider the symmetric difference $M \oplus M'$: since M, M' are matchings, every vertex has degree ≤ 2 in this symmetric difference, so it decomposes into isolated vertices, even-length cycles (equal edges from each), and paths (even or odd length). If $|M'| > |M|$, some component must contain strictly more M'-edges than M-edges; only an *odd-length* path can do this, and such a path – alternating, both endpoints unmatched in M – is exactly an augmenting path for M. So while M is not maximum, an augmenting path must exist; conversely once none exists, M is maximum.

Finding augmenting paths – simple vs. Hopcroft-Karp

Simple method: BFS/DFS from each unmatched vertex, alternately taking a non-matching edge then a matching edge. $\leq |V|/2$ augmenting paths found, each search $O(|E|)$: total $O(|V||E|)$.
Hopcroft-Karp: in each phase, find a *maximal set of vertex-disjoint shortest* augmenting paths simultaneously (combined BFS+DFS), and augment along all of them at once. Only $O(\sqrt{|V|})$ phases are needed (an argument bounding how many disjoint shortest augmenting paths can exist once the current matching is close to optimal), each phase costing $O(|E|)$: total $O(|E|\sqrt{|V|})$.

Practice: Example Sheet 5, Q1 – Ford-Fulkerson by hand

Use Ford-Fulkerson by hand to find the maximum flow from s to t in a network shaped like $s \to a$, $s \to b$, $a \to b$, $a \to t$, $b \to t$ with capacities $1000, 1000, 1, 1000, 1000$ respectively. How many iterations did you take? What is the *largest* number of iterations Ford-Fulkerson might take here with an unfortunate choice of augmenting path? *(This is the standard example showing why Edmonds-Karp's BFS rule matters – an adversarial choice can alternate through the capacity-1 edge* 1000 *times.)*

2.3 Minimum Spanning Trees

Input: connected undirected $G = (V, E, w)$. **Output:** acyclic $T \subseteq E$ connecting all vertices with minimum $w(T) = \sum_{(u,v)\in T} w(u, v)$. (An MST need not be unique.)

Cut, cross, respect, light – vocabulary for the Safe Edge Theorem

A **cut** $(S, V \setminus S)$ partitions V. Edge (u, v) **crosses** the cut if $u \in S, v \notin S$. A cut **respects** a set A of edges if no edge of A crosses it. A crossing edge is **light** if its weight is minimum among crossing edges (the minimum crossing weight is unique; light edges achieving it may not be).
Both MST algorithms grow a set $A \subseteq$ some MST T by repeatedly adding **safe edges** – edges that can be added while preserving $A \subseteq T'$ for *some* MST T'.

The Safe Edge Theorem

Statement: let $A \subseteq$ some MST T of G; let $(S, V \setminus S)$ be any cut respecting A; let (u, v) be a light edge crossing this cut. Then (u, v) is safe for A.
Proof: let $T \supseteq A$ be an MST. If $(u, v) \in T$, done. Otherwise adding (u, v) to T creates a cycle containing the unique path $p : u \rightsquigarrow v$ in T; since (u, v) crosses the cut, p must contain some other edge (x, y) that also crosses it. Since the cut respects A, $(x, y) \notin A$. Let $T' = T - (x, y) + (u, v)$: still a spanning tree (connected, acyclic, same vertex set). Then

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$$

since (u, v) is light, so $w(u, v) \leq w(x, y)$. As T was minimum, T' is also an MST, and $A \subseteq T'$ (since $(x, y) \notin A$, removing it can't remove any edge of A). Hence (u, v) is safe for A.
Corollary (used directly by both algorithms): if C is a connected component (tree) of the forest $G_A = (V, A)$, and (u, v) is a *light* edge connecting C to any other component, then (u, v) is safe for A – take the cut $(V_C, V \setminus V_C)$, which respects A since A's edges never leave C.

Kruskal's Algorithm – grows a forest

```
MST-KRUSKAL(G, w)
1  A = {}
2  S = new DisjointSet;  for v in G.V:  MAKE-SET(S, v)
3  MERGE-SORT(G.E)                       // by non-decreasing weight
4  for (u, v) in G.E                     // stop early if |A| = |V|-1
5      if !IN-SAME-SET(S, u, v)
6          A = A union {(u,v)}
7          UNION(S, u, v)
8  return A
```

At each step, the cheapest edge crossing the cut separating its two endpoints' components is safe by the corollary. Cost: sort $O(|E| \lg |E|) = O(|E| \lg |V|)$; $|E|$ In-Same-Set + $|V| - 1$ Union calls at $\approx O(1)$ amortised each (disjoint sets, §3.4) $= O(|E| + |V|)$. **Total:** $O(|E| \lg |V|)$, dominated by the sort.

Prim's Algorithm – grows a single tree (resembles Dijkstra)

```
MST-PRIM(G, w, r)
1  Q = new PriorityQueue
2  for v in G.V:  v.key = inf;  v.pi = NIL;  PQ-ENQUEUE(Q, v)
3  PQ-DECREASE-KEY(Q, r, 0)
4  while !PQ-IS-EMPTY(Q)
5      u = PQ-EXTRACT-MIN(Q)
6      for v in G.E.adj[u]
```

```
7           if v in Q  &&  w(u,v) < v.key
8               v.pi = u;  v.key = w(u,v);  PQ-DECREASE-KEY(Q, v, v.key)
```

Cost of Prim by priority-queue implementation

- **Binary heap**: $|V|$ Extract-Min at $O(\lg|V|)$ + $|E|$ Decrease-Key at $O(\lg|V|) = O(|E|\lg|V|)$ (since G connected, $|E| \geq |V| - 1$).
- **Fibonacci heap**: $|V|$ Extract-Min at $O(\lg|V|)$ + $|E|$ Decrease-Key at $O(1)$ amortised $= O(|E| + |V|\lg|V|)$ – **asymptotically better** for sparse graphs, which is precisely why Fibonacci heaps were invented (Prim and Dijkstra share this structure).

Membership testing "$v \in Q$" is implemented with an auxiliary bitmap, $O(1)$ per test, no effect on the asymptotic bound.

Practice: Example Sheet 5 – MST/cut reasoning

Prove or disprove: if all edge weights in a connected graph are distinct, the MST is unique. *(True – if two different MSTs existed, consider the lightest edge in one but not the other; the Safe Edge Theorem / exchange argument forces a contradiction with distinct weights.)* Then: does the *same* conclusion hold if edge weights may repeat? Give a small counterexample.

2.4 Section 2 Summary Checklist

Before moving on, make sure you can. . .

- Build a residual network from a flow and identify all augmenting paths in it.
- Reproduce all three legs of the Max-Flow Min-Cut proof from memory, not just quote the theorem.
- Explain why Ford-Fulkerson can fail to terminate on irrational capacities, and how Edmonds-Karp/integer capacities avoid this.
- State the Safe Edge Theorem and its corollary, and use the corollary directly to justify why Kruskal's and Prim's next-chosen edge is safe.
- Compare Kruskal vs Prim's costs and know when each wins (sparse vs. dense, and PQ choice).

3 Advanced Data Structures

3.1 Amortised Analysis

Worst-case-per-operation analysis is sometimes too pessimistic (e.g. a resizing vector: most INSERTs are $O(1)$, occasional ones are $O(n)$). **Amortised analysis** bounds the *average* cost per operation over any sequence, without assuming randomness.

Three methods

1. **Aggregate analysis:** bound the total cost $T(n)$ of n operations directly; amortised cost is $T(n)/n$.
2. **Accounting method:** assign each operation an amortised *charge*; overcharges accumulate as **credit** on the structure/items, undercharges are paid from credit. Valid iff credit never goes negative and total amortised cost $\geq$ total actual cost for every prefix of every sequence.
3. **Potential method:** define $\Phi(D_i) \geq 0$ for every reachable state D_i, with $\Phi(D_0) = 0$. Amortised cost of operation i is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ (actual cost + potential change). Summing telescopes: $\sum \hat{c}_i = \sum c_i + \Phi(D_n) - \Phi(D_0) \geq \sum c_i$ since $\Phi(D_n) \geq 0$ – so a valid potential function always gives a genuine upper bound on total actual cost.

Worked example: resizing vector INSERT (aggregate method)

Start with capacity 16. The first 16 inserts are $O(1)$ each. The 17th insert: allocate size-32 array, copy 16 elements ($\Theta(16)$), insert 1 more. In general doubling from n to $2n$ costs $\Theta(n)$ to copy. For $N = 2^k$ inserts, total cost $= 16 + (16+1) + 15 + (32+1) + 31 + \cdots \in O(N)$ (a geometric-type sum dominated by its largest term, $\Theta(N)$). Amortised cost per insert $= O(N)/N = O(1)$. *This is the proof behind the "amortised $O(1)$" claim you assumed for stacks/vectors in Algorithms I.*

Worked example: binary counter INCREMENT (potential method)

INCREMENT adds 1 to a counter stored as a bit array. Potential $\Phi = b_i$ = number of 1-bits after the ith increment. $\Phi(D_0) = 0$ (starts at zero, no 1-bits) ✓.
If INCREMENT resets r_i bits (rippling carries) and sets exactly one bit from 0 to 1, actual cost $\leq r_i + 1$. Potential change $= (b_{i-1} - r_i + 1) - b_{i-1} = 1 - r_i$. Amortised cost $= (r_i + 1) + (1 - r_i) = 2 \in O(1)$. So n increments cost $O(n)$ total: $O(1)$ amortised each, even though a single increment can cost $\Theta(\lg n)$ in the worst case (all bits set).

Exam technique for amortised-analysis questions

State *which* method you're using; for the potential method always **check** $\Phi(D_0) = 0$ **and** $\Phi \geq 0$ **everywhere** before using it (a common part-mark); and always finish by concluding what the total/amortised cost is in $O(\cdot)$ notation for n operations.

3.2 Abstract Data Types and Mergeable Priority Queues

An **ADT** specifies operations (names, signatures, semantics) with no fixed implementation – like an interface. The (binary/d-ary) heaps of Algorithms I implement every Priority Queue operation in $O(\lg n)$ or better, *except* DESTRUCTIVE-UNION (merging two heaps), which costs $\Theta(n_1 + n_2)$ (copy + re-heapify). **Binomial Heaps** and **Fibonacci Heaps** both implement the **Mergeable Priority Queue ADT** with fast merging, at the cost of no SEARCH (callers must hold node pointers for DECREASE-KEY/DELETE).

Binomial Trees B_k – building blocks

B_k is formed by linking two B_{k-1} trees, making one root the leftmost child of the other; B_0 is a single node. Properties (all provable by induction):

- B_k has 2^k nodes and height k; exactly $\binom{k}{i}$ nodes at depth i.
- The root has degree k (largest in the tree); its children, in order, root subtrees $B_{k-1}, B_{k-2}, \ldots, B_0$.
- Max degree of any node is $O(\lg n)$.

A **Binomial Heap** is a collection of min-heap-ordered binomial trees, at most one of each degree – so an n-node heap's tree sizes mirror the **binary representation of** n (e.g. $n = 11 = 1011_2$ needs trees B_3, B_1, B_0), giving at most $\lfloor \lg n \rfloor + 1$ trees in the root list.

Binomial Heap operation costs

Operation	Cost
CREATE	$O(1)$
PEEK-MIN (scan root list)	$O(\lg n)$
DESTRUCTIVE-UNION (merge sorted root lists like binary addition)	$O(\lg n)$
INSERT (= union with a 1-node heap)	$O(\lg n)$
EXTRACT-MIN (cut min's tree out, reverse its children, union back in)	$O(\lg n)$
DECREASE-KEY (bubble up within its tree, like a min-heap)	$O(\lg n)$

Merging two same-degree trees (BH-MERGE) is $O(1)$: attach one root as the new first child of the other.

3.3 Fibonacci Heaps

Fibonacci heaps push INSERT, DESTRUCTIVE-UNION, and DECREASE-KEY down to $O(1)$ *amortised*, at the cost of a more relaxed structure than binomial heaps: a collection of min-heap-ordered trees (not one-per-degree), with lazy consolidation deferred to EXTRACT-MIN.

Structure

Nodes carry: key/payload, parent, one child pointer, a circular doubly-linked **sibling** list (of that node and its siblings), a degree, and a **mark** bit. A node becomes marked when it loses a child *while remaining* in the tree (i.e. after DECREASE-KEY cuts one child out); if it loses a *second* child while marked, it is itself cut and moved to the root list (**cascading cut**), and unmarked (root-list nodes are never marked).

Potential function and $O(1)$-amortised operations

Potential $\Phi = r + 2m$ where r = size of root list, m = number of marked nodes. Check: empty heap has $r = m = 0 \Rightarrow \Phi = 0$ ✓.

- **Insert**: splices a 1-node tree into the root list. $r \to r + 1$, m unchanged. $\hat{c} = k + \Delta\Phi = k + 1 \in O(1)$.
- **Destructive-Union**: splices two root lists. $\Delta\Phi = 0$ relative to the sum of the two input potentials (that "spent" potential pays for repairing the merged structure later). $\hat{c} = k \in O(1)$.
- **Decrease-Key**: if the new key still $\geq$ parent, no structural change: $\hat{c} = O(1)$ directly. Otherwise the node (and, via cascading cuts, up to a marked ancestors) move to the root list: actual cost $k_1 + a \cdot k_2$; $\Delta\Phi = (1 + a) - 2a = 1 - a$ (root list grows by $1 + a$, marked count drops by a as cut nodes unmark). $\hat{c} = k_1 + ak_2 + (1 - a) \in O(1)$ for $k_2 = 1$ – the a terms *cancel*, which is the whole trick of cascading cuts being "free" in amortised terms.
- **Extract-Min**: promotes $\leq D(n)$ children of the old min into the root list, removes the min,

then *consolidates* (repeatedly merges same-degree root-list trees using an array indexed by degree) until all root degrees are distinct, leaving $\leq D(n)+1$ roots. Actual cost $O(r+D(n))$; $\Delta\Phi \leq (D(n)+1)-r$ (worst case, ignoring any unmarking that would only help). $\hat{c} \in O(D(n))$.

So all core operations are $O(1)$ amortised except EXTRACT-MIN, which is $O(D(n))$ where $D(n)$ is the max degree of any node in an n-node Fibonacci heap.

Bounding $D(n) = O(\lg n)$ – the golden-ratio argument

Lemma 1. If x has degree k, with children $c_1, \ldots, c_k$ in the order they were linked, then c_1.degree ≥ 0 and c_i.degree $\geq i-2$ for $i = 2..k$. *(When c_i was linked to x, both had equal degree $i-1$; since then c_i can have lost at most one child – a second loss would have cascade-cut it out from under x – so its degree dropped by at most 1.)*

Define size(x) = number of nodes in the subtree rooted at x. Using Lemma 1 and induction, size(x) $\geq F_{k+2}$ (the $(k+2)$th Fibonacci number) where x has degree k. Using the identity $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$ and the standard bound $F_{k+2} \geq \varphi^k$ (proved by strong induction, $\varphi = (1+\sqrt{5})/2$ the golden ratio, since $\varphi^2 = \varphi + 1$), we get $n \geq \text{size}(x) \geq \varphi^k$, i.e. $k \leq \log_\varphi n$. Hence $D(n) \leq \lfloor \log_\varphi n \rfloor = O(\lg n)$, so EXTRACT-MIN is $O(\lg n)$ amortised. *(This golden-ratio bound is exactly where "Fibonacci" heaps get their name.)*

Why this matters: Dijkstra / Prim with a Fibonacci heap

Both algorithms do $|V|$ EXTRACT-MIN ($O(\lg |V|)$ each) and $O(|E|)$ DECREASE-KEY calls ($O(1)$ *amortised* each). Total: $O(|V| \lg |V| + |E|)$ – strictly better than a binary heap's $O((|V|+|E|) \lg |V|)$ whenever the graph is sparse-ish, since the $|E|$ term drops the $\lg |V|$ factor entirely.

3.4 Disjoint Sets

ADT: initialise n singleton sets; support UNION(s_1, s_2) and IN-SAME-SET(k_1, k_2).

Representations and their costs

Representation	CREATE	UNION	IN-SAME-SET
Doubly linked lists	$O(n)$	$O(n)$	$O(n)$
Cyclic doubly linked lists	$O(n)$	$O(1)$	$O(n)$
Hash table (key→set-id)	$O(n)$	$O(n)$	$O(1)$
Trees + Path Compression & Union-by-Rank	$O(n)$	$\sim O(1)$ amortised	$\sim O(1)$ amortised

Every representation trades off which operation is fast – *except* the tree-based one, which gets *both* operations nearly free amortised.

Trees with Path Compression & Union-by-Rank

```
CHASE(k):  follow parent pointers from k's node to the root r.
           Re-point every visited node directly to r (PATH COMPRESSION).
UNION(a, b):  ra = CHASE(a);  rb = CHASE(b)
              attach the shallower-rank root under the deeper-rank root;
              if ranks are equal, pick either and increment its rank
              (UNION-BY-RANK).
IN-SAME-SET(k1, k2):  return CHASE(k1) == CHASE(k2)
```

Each node stores a parent pointer (`NIL` for roots) and a **rank** (an upper bound on subtree depth, *not* an exact size). Combined, path compression and union-by-rank give an amortised cost of $O(\alpha(n))$ per operation, where α is the inverse Ackermann function – for all n that could ever be represented in a real computer, $\alpha(n) \leq 4$, hence the informal "$\sim O(1)$ amortised". *(The full inverse-Ackermann proof is beyond this course's expected depth – know the result and the two techniques that achieve it, not the*

Ackermann-function proof itself.)

Application: Kruskal's Algorithm cost, revisited

Kruskal calls CREATE once, UNION $|V|-1$ times, IN-SAME-SET $|E|$ times: with the tree representation, this is $O(|V|+|E|)$ amortised, dominated by the $O(|E|\lg|V|)$ sort – confirming the Kruskal cost claimed in §2.3.

Practice: Example Sheet 6 – amortised analysis and heaps

(a) Find a sequence of N INSERTs into a binary heap that takes $\Omega(N \lg N)$ total elementary operations (i.e. show the "amortised $O(\lg n)$ per insert" bound for ordinary heaps is tight, unlike a resizing vector).
(b) A k-bit binary counter supports only `inc()`. Using the potential method, derive the amortised cost of `inc()` and hence the total cost of n calls starting from zero.
(c) Explain carefully how to implement DECREASE-KEY on a Fibonacci Heap, including the cascading-cut rule, and justify why its amortised cost is $O(1)$.

3.5 Section 3 Summary Checklist

Before moving on, make sure you can. . .

- Choose the right amortised-analysis method for a given data structure and correctly verify $\Phi(D_0)=0,\ \Phi\geq 0$ before using the potential method.
- State the Fibonacci heap potential function $\Phi = r + 2m$ and derive the $O(1)$-amortised cost of INSERT, UNION, and DECREASE-KEY (including the cascading-cut cancellation trick).
- Sketch why $D(n) = O(\lg n)$ via the Fibonacci-number/golden-ratio argument, and state why this makes EXTRACT-MIN $O(\lg n)$ amortised.
- Explain path compression and union-by-rank, and why combining them (not just one alone) is what gives near-constant amortised cost.
- Compare all disjoint-set representations' costs and explain why Kruskal specifically benefits from the tree-based one.

4 Geometric Algorithms

4.1 Polygons, Insidedness, and Winding Numbers

A **polygon** is an ordered list of vertices (points in a 2D vector space). We care about **planar** (flat, so "inside"/"outside" are well-defined – unlike, e.g., regions on a sphere), **closed** (edge from last vertex back to first), **simple** (non-self-intersecting) polygons.

Winding number method (and why we avoid it)

Imagine a piece of string from the test point to a point walking around the polygon boundary; count net rotations. If *odd*, the point is inside; if *even* (incl. zero), outside. Implementable by summing subtended angles between consecutive edge endpoints and dividing by 2π – but this needs **trigonometry** (slow) and suffers **floating-point error**. We prefer a **ray-casting** method using only additions/comparisons.

Ray-casting (crossing number) method

Cast a semi-infinite ray from the test point P in any fixed direction (conventionally horizontal, to simplify the "is this point on the ray" test using total ordering of floats). Since P is finite and any point at infinity is "outside", and each edge of a simple closed polygon separates inside from outside, **count edge crossings**: odd $\Rightarrow$ inside, even $\Rightarrow$ outside.

Awkward case: the ray passes exactly through a vertex. Fix: look at that vertex's two neighbours – if they're on the *same* side of the ray, the edge pair doesn't actually cross (case 2, tangent); if on *opposite* sides, it does cross (case 1). If a neighbour is *also* on the ray, keep walking to the next vertex in that direction until you find one that isn't.

4.2 Cross Products for Orientation (Avoiding Trigonometry)

The 2D **cross product** of $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is the scalar $p_1 \times p_2 = x_1 y_2 - x_2 y_1$ (the signed area of the parallelogram they span; equivalently $|p_1||p_2| \sin\theta$).

Reading orientation off the sign

- $p_1 \times p_2 > 0$: p_1 is clockwise from p_2 (equivalently p_2 is anticlockwise/left of p_1).
- $p_1 \times p_2 < 0$: p_1 is anticlockwise from p_2.
- $p_1 \times p_2 = 0$: p_1, p_2 collinear (parallel or antiparallel).
- $p_1 \times p_2 = -(p_2 \times p_1)$ (anticommutative).

Why this beats trigonometry: no division, no $\sin$ / $\cos$ calls, exact for integer/rational coordinates, and numerically well-conditioned. Cross products are the workhorse of every algorithm below.

4.3 Line Segment Intersection

Input: segments p_1p_2 and p_3p_4. **Output:** do they intersect?

Why not just solve $y = mx + c$ simultaneously?

That approach needs **division** (slow, and loses "infinite precision" – exactness under $+, \times$ that is not preserved under $\div$), and the intersection point of the two *infinite lines* can fall just outside the finite *segments*, so tiny floating-point error can flip the answer. This is an **ill-conditioned** numerical problem (small input perturbation $\Rightarrow$ large output change). The cross-product / straddle test below avoids division entirely.

Segment p_1p_2 **straddles** the (infinite) line through p_3p_4 if p_1, p_2 lie on opposite sides of that line. Two segments intersect $\iff$ each straddles the line containing the other, *or* an endpoint of one lies exactly on the other segment.

Segments-Intersect(p_1, p_2, p_3, p_4)

```
DIRECTION(pi, pj, pk) = (pk - pi) x (pj - pi)
ON-SEGMENT(pi, pj, pk) = (min(xi,xj) <= xk <= max(xi,xj)) and
                         (min(yi,yj) <= yk <= max(yi,yj))

SEGMENTS-INTERSECT(p1, p2, p3, p4)
1  d1 = DIRECTION(p3, p4, p1)     // orientation of p1 w.r.t. line p3p4
2  d2 = DIRECTION(p3, p4, p2)     // orientation of p2 w.r.t. line p3p4
3  d3 = DIRECTION(p1, p2, p3)     // orientation of p3 w.r.t. line p1p2
4  d4 = DIRECTION(p1, p2, p4)     // orientation of p4 w.r.t. line p1p2
5  if ((d1>0 && d2<0) || (d1<0 && d2>0)) and
     ((d3>0 && d4<0) || (d3<0 && d4>0))
6      return true                // both segments straddle each other
7  else if d1==0 && ON-SEGMENT(p3,p4,p1):  return true
8  else if d2==0 && ON-SEGMENT(p3,p4,p2):  return true
9  else if d3==0 && ON-SEGMENT(p1,p2,p3):  return true
10 else if d4==0 && ON-SEGMENT(p1,p2,p4):  return true
11 return false
```

Each call is $O(1)$. For n **segments, any pair intersecting?**: the naive all-pairs check is $O(n^2)$; a **sweep-line** algorithm (sweeping a vertical line left-to-right, maintaining segments crossing it in a balanced BST ordered by y-coordinate, checking only newly-adjacent segments) achieves $O(n \lg n)$.

4.4 Convex Hull

Input: $n > 2$ points, not all collinear. **Output:** the ordered vertices of the smallest convex polygon containing all points (the **convex hull**).

Five families of solution (know the complexities!)

Approach	Algorithm	Cost
Rotational sweep	Graham's Scan	$O(n \lg n)$
Rotational sweep	Jarvis's March ("gift wrapping")	$O(nh)$
Incremental	(add points one at a time, repair hull)	$O(n \lg n)$
Divide and conquer	(split, solve, merge two hulls)	$O(n \lg n)$
Prune and search		$O(n \lg h)$

n = number of input points, h = number of points *on* the hull. Since $h \leq n$, Prune-and-Search is asymptotically fastest; Jarvis's March is best when h is known to be small (**output-sensitive**).

Graham's Scan

```
1  Let p0 = the lowest point (leftmost among the bottom-most, to break ties)
2  Sort the other points by increasing polar angle about p0
     (ties: keep only the point farthest from p0)
3  Push p0, then the first two sorted points, onto a stack
4  for each remaining point p (in sorted order)
5      while the turn (2nd-from-top -> top -> p) is NOT a strict left turn
6          pop the stack
7      push p
8  The stack now holds the convex hull, in order
```

Sorting by polar angle without trigonometry: for unit vectors, the cross product $a \times b = \sin\theta$ is monotonic in θ for $-\pi/2 \leq \theta < \pi/2$, so sorting by cross-product sign/magnitude (after normalising, which is cheaper than computing angles) sorts by angle directly. **Left-turn test**: for points a, b, c, the turn at b is a left turn iff $(b - a) \times (c - b) > 0$. Cost: sort dominates at $O(n \lg n)$; each point is pushed once and popped at most once, so the scan itself is $O(n)$ amortised (aggregate analysis!) –

total $O(n \lg n)$.

Jarvis's March (Gift Wrapping)

```
1  p1 = the lowest (leftmost-of-bottom-most) point -- on the hull
2  p2 = the point with least polar angle relative to a horizontal
       line through p1 -- on the hull
3  Repeatedly find p_{i+1} = the point with least polar angle relative
   to the line through p_{i-1}, p_i, until a topmost point is reached
   -- this is the "right chain"
4  Repeat symmetrically (greatest polar angles) from the topmost point
   back down to p1 -- the "left chain"
5  Join the two chains to form the hull
```

Analysis: one angle-finding step scans all n points in $O(n)$; there are h hull points to find; total $O(nh)$. The same cross-product trick applies since angles are always restricted to a half-plane at each step.

Common exam trap: Graham's Scan vs. Jarvis's March

Don't confuse *which* points get sorted by angle relative to *what*: Graham sorts *all n points* once, relative to a single fixed pivot p_0; Jarvis repeatedly finds *one new point at a time*, relative to the *previous hull edge* (not a fixed pivot). This is exactly why their costs differ ($n \lg n$ vs. nh).

Practice: Geometry-style Tripos question (author-composed, following the course's conventions

(a) Given points $p_1 = (0, 0)$, $p_2 = (4, 0)$, $p_3 = (4, 3)$, $p_4 = (0, 3)$, and a test point $p_5 = (2, 1.5)$, use the cross-product straddle test to determine whether the diagonal p_1p_3 intersects the diagonal p_2p_4. *[6 marks]*

(b) Explain why the "high-school" method of solving two line equations $y = mx + c$ simultaneously is considered numerically inferior to the cross-product method, using the concept of an *ill-conditioned* problem. *[4 marks]*

(c) Trace Graham's Scan on a small set of 6 points (sketch a diagram), showing the stack contents after each step. *[10 marks]*

4.5 Section 4 Summary Checklist

Before moving on, make sure you can. . .

- Explain why planarity, closedness, and simplicity all matter for defining "inside" a polygon.
- State the ray-casting rule and correctly resolve the vertex/tangent edge cases.
- Read orientation and collinearity directly off the sign of a 2D cross product, with no trigonometry.
- Reproduce SEGMENTS-INTERSECT and justify each of its five `return true` branches.
- Trace Graham's Scan and Jarvis's March by hand and state their complexities in terms of n (and h for Jarvis).

5 Further Practice & Exam Strategy

5.1 Why Algos II Questions Feel "Nasty"

The examiners combine mechanics with proof

Algorithms II questions typically demand *both* the mechanical trace of an algorithm *and* a formal justification (a named theorem, a proof sketch, or a correctness/complexity argument), often in the same part-question. To get full marks:

1. **Identify the bottleneck**: always state the overall complexity *and* which line/step dominates it.
2. **Formalise the problem**: don't just say "it's a shortest-path problem" – say "this is an SSSP problem with non-negative weights, so Dijkstra applies (not Bellman-Ford)", or "this is a max-flow problem, and by Max-Flow Min-Cut the answer equals the min cut's capacity."
3. **Master the four set-piece proofs**: the **Convergence Lemma** (Dijkstra), the **Safe Edge Theorem** (MSTs), the **Max-Flow Min-Cut Theorem**, and the **Fibonacci heap potential argument** ($\Phi = r + 2m$) are exam favourites – know each well enough to reproduce from a blank page, not just recognise.

5.2 Algorithm Selection Cheat-Sheet

Situation	Algorithm	Cost
Unweighted SSSP	BFS	$O(\|V\| + \|E\|)$
SSSP, possibly negative weights	Bellman-Ford	$O(\|V\|\|E\|)$
SSSP, non-negative weights	Dijkstra (Fib. heap)	$O(\|V\| \lg \|V\| + \|E\|)$
APSP, dense graph	Floyd-Warshall	$O(\|V\|^3)$
APSP, sparse graph, poss. negative	Johnson's (reweight + Dijkstra)	$O(\|V\|^2 \lg \|V\| + \|V\|\|E\|)$
Topological order	DFS, sort by desc. finish time	$O(\|V\| + \|E\|)$
Strongly connected components	Kosaraju (2 × DFS + transpose)	$O(\|V\| + \|E\|)$
Max flow, general	Ford-Fulkerson	$O(\|E\|\|f^*\|)$
Max flow, capacity-independent bound	Edmonds-Karp	$O(\|V\|\|E\|^2)$
Max bipartite matching	Ford-Fulkerson on unit-capacity network	$O(\|V\|\|E\|)$
Max bipartite matching, faster	Hopcroft-Karp	$O(\|E\|\sqrt{\|V\|})$
MST, sparse graph	Kruskal (+ disjoint sets)	$O(\|E\| \lg \|V\|)$
MST, dense graph / Fib. heap available	Prim (Fib. heap)	$O(\|E\| + \|V\| \lg \|V\|)$
Mergeable PQ, simplest	Binomial Heap	$O(\lg n)$ per op
Mergeable PQ, fast Decrease-Key	Fibonacci Heap	$O(1)$ amort. (not Extract-Min)
Dynamic connectivity queries	Disjoint sets, PC + union-by-rank	$\sim O(1)$ amort.
Convex hull, general	Graham's Scan	$O(n \lg n)$
Convex hull, few hull points expected	Jarvis's March	$O(nh)$

5.3 Past-Paper-Style Question Index by Topic

How to use this table

Cambridge's Example Sheets 4–6 (Graphs & Path-Finding; Flows & Subgraphs; Advanced Data Structures) are the closest available proxy for Tripos-style Algorithms II questions in the source material used to build this guide. **If you send over actual Tripos past-paper questions (by year), this guide can be extended with a dedicated "Past Paper Index" table and matching `examqbox` practice blocks in the same style as the Algorithms I guide.**

Topic	Where to drill
BFS/DFS, edge classification, SCCs	Example Sheet 4, Q1–Q7
Bellman-Ford, Dijkstra, correctness proofs	Example Sheet 4, Q8–Q13
Floyd-Warshall, Johnson's, matrix methods	Example Sheet 4, Q14–Q19
Flow networks, Ford-Fulkerson, Max-Flow Min-Cut	Example Sheet 5, Q1–Q9
Bipartite matching, Hopcroft-Karp	Example Sheet 5, Q9–Q13
MSTs, Safe Edge Theorem, Kruskal/Prim	Example Sheet 5 (later questions) / Example Sheet 6 (cross-topic)
Amortised analysis (aggregate/accounting/potential)	Example Sheet 6, Q1–Q5
Binomial & Fibonacci Heaps	Example Sheet 6, Q6–Q9
Disjoint sets	Example Sheet 6, Q10–Q13
Geometric algorithms	No dedicated example sheet found in source material – practise wit CLRS Ch. 33 and this guide's §4 exam-style question

5.4 General Exam Technique

Checklist for every Algorithms II answer

- **Name the algorithm explicitly** before diving into pseudocode or a trace – markers award marks for correct *identification*, separately from correct *execution*.
- **State preconditions**: "non-negative weights" for Dijkstra, "no negative cycles" for a meaningful Bellman-Ford/Floyd-Warshall answer, "integer capacities" if you assert Ford-Fulkerson terminates.
- For any **correctness proof**, identify whether it's induction (BFS/Dijkstra/DFS-property style), proof by contradiction with a minimal counterexample (Dijkstra/BFS main proofs), or an exchange argument (Safe Edge Theorem, MST uniqueness).
- For **complexity questions**, always state which priority-queue / data-structure implementation you're assuming – the same algorithm's cost can change by a $\lg n$ factor (Dijkstra/Prim with array vs. heap vs. Fibonacci heap).
- For **amortised-analysis questions**, explicitly name the method (aggregate/accounting/potential), and for the potential method, always verify $\Phi(D_0) = 0$ and $\Phi \geq 0$ before using it.
- For **geometry questions**, default to the cross-product formulation over trigonometry unless the question explicitly asks for an angle-based method – and justify why (numerical stability, avoiding division).
- Sanity-check graph algorithms against small tricky inputs: a single vertex, a disconnected graph, a graph with a self-loop-free but otherwise degenerate structure, an already-sorted edge list (Kruskal), a star graph (Prim/Dijkstra worst case for array PQ).

Highest-yield theorems to have completely memorised

1. **Convergence Lemma / Dijkstra correctness** (§1.5.2)
2. **Safe Edge Theorem + Corollary** (§2.3)
3. **Max-Flow Min-Cut Theorem**, all three legs of the proof (§2.1)
4. **Fibonacci Heap potential function** $\Phi = r + 2m$ and the amortised cost of each operation (§3.3)
5. **Augmenting-path characterisation of maximum matchings** via symmetric difference (§2.2)

These five appear, in some form, in almost every recent Algorithms II-style question set available in the source material for this guide.