CS 180 - Algorithms and Complexity

GRAPHS

1 Basic Definitions and Properties

Graph. A graph G is an ordered pair (V, E) where V is a set of vertices and E is a set of edges. An edge is either an unordered pair $\{u, v\}$ (undirected) or an ordered pair (u, v) (directed).

Directed and Undirected Graphs. In an undirected graph, edges have no orientation. In a directed graph (digraph), edges have orientation and are written $u \to v$.

Adjacency. Vertices u and v are adjacent if they share an edge. In a digraph, u is adjacent to v if there is an edge $u \to v$.

Degree. In an undirected graph, the degree $\deg(v)$ of a vertex v is the number of incident edges. In a digraph, $\deg^-(v)$ is the in-degree and $\deg^+(v)$ is the out-degree.

Paths and Cycles. A path is a sequence of vertices connected by edges. A cycle is a path whose first and last vertices coincide and which contains at least one edge. A graph with no cycles is acyclic.

Connectivity. An undirected graph is connected if any two vertices can be joined by a path. A digraph is strongly connected if every vertex can reach every other by a directed path, and weakly connected if this holds after ignoring edge directions.

Trees. A tree is a connected acyclic undirected graph. A forest is a disjoint union of trees. A rooted tree has a designated root and edges oriented away from it.

Subgraphs. A subgraph of $G = (V, E)$ is a graph $H = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. An induced subgraph on $V' \subseteq V$ contains all edges of G with both endpoints in V'.

Representation. Common representations:

- **Adjacency list**: For each vertex, store its neighbors. Uses $O(V + E)$ space.
- **Adjacency matrix**: A $V \times V$ matrix indicating edges. Uses $O(V^2)$ space.

Weighted Graphs. A weighted graph assigns weights to edges. The weight of a path is the sum of its edge weights.

DAGs. A Directed Acyclic Graph (DAG) is a digraph with no directed cycles and always admits a topological ordering.

2 Algebraic and Counting Facts

Maximum number of edges. For a simple undirected graph on n vertices (no loops, no multiple edges),

$$|E| \le \binom{n}{2}.$$

For a simple directed graph (no self-loops, no parallel edges),

$$|E| \le n(n-1).$$

Handshaking Lemma (undirected). In any undirected graph,

$$\sum_{v \in V} \deg(v) = 2|E|.$$

The sum of degrees is even, and the average degree is

$$\frac{1}{n} \sum_{v \in V} \deg(v) = \frac{2|E|}{n}.$$

Directed Handshaking. For any digraph,

$$\sum_{v \in V} \deg^-(v) = |E| = \sum_{v \in V} \deg^+(v).$$

Trees. A tree on n vertices has exactly $n - 1$ edges. Equivalently, an undirected graph is a tree if and only if any two of the following hold:

- It is connected.
- It is acyclic.
- It has $n - 1$ edges.

Connected components. For any undirected graph with n vertices and c connected components,

$$|E| \ge n - c.$$

Equality holds if and only if each component is a tree.

Complete graphs. The complete graph K_n has

$$|E(K_n)| = \binom{n}{2}.$$

Bipartite graphs. If a graph is bipartite with partition sizes a and b, the maximum number of edges is

$$|E| \le ab,$$

achieved by the complete bipartite graph $K_{a,b}$.

Average degree bounds. For any simple undirected graph,

$$0 \le \deg(v) \le n - 1$$

and

$$0 \le \frac{2|E|}{n} \le n - 1.$$

Acyclic directed graphs. Every DAG has at least one vertex of in-degree zero and at least one vertex of out-degree zero.

3 Topological Sort

Description. Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge $u \to v$, vertex u comes before v in the ordering. There may be several topological orderings for a graph.

Algorithm. One of the idea is to repeatedly identifying and "removing" vertices with an in-degree of zero

1. Initialize:
   (a) Compute the in-degree of every vertex.
   (b) Enqueue all vertices with in-degree zero into an empty queue.
2. While the queue is not empty:
   (a) Dequeue a vertex and output it to the topological order.
   (b) For each neighbor of that vertex:
      i. Decrease the neighbor's in-degree by one.
      ii. If the in-degree of the neighbor becomes zero, enqueue it.

Proof of correctness. We prove correctness by induction and a contradiction argument.

At every iteration, the algorithm removes a vertex of in degree zero. Such a vertex has no incoming edges, so removing it cannot violate edge order constraints. The algorithm also decreases the in degree of outgoing neighbors, preserving correct in degree counts.

Claim: If the graph is a DAG, a vertex with in degree zero always exists until all vertices are removed.

Proof. Suppose instead that at some point the remaining graph has no vertex with in degree zero. Then every remaining vertex has at least one incoming edge. Starting from any vertex and repeatedly following an incoming edge must eventually revisit a vertex due to finiteness of the graph, forming a directed cycle. This contradicts the assumption that the graph is a DAG. Therefore a zero in degree vertex always exists.

By induction on the number of removed vertices, every removed vertex appears only after all of its prerequisites, so the produced ordering is a valid topological ordering.

Time complexity. Computing the in degree of all vertices requires scanning every edge once, so this takes $O(E)$ time.

Each vertex is inserted into and removed from the queue at most once, contributing $O(V)$ time for queue operations.

When a vertex is removed, the algorithm iterates over all of its outgoing edges and decreases the in degree of each neighbor. Across the entire algorithm, every edge is considered exactly once, which adds another $O(E)$.

Thus, the total running time is

$$\boxed{O(V + E)}$$

which is optimal for adjacency list representation.

4 BFS

Description. Breadth-First Search (BFS) is a graph traversal algorithm that explores a graph level by level, visiting all neighbors of a node before moving to the next level of neighbors. It uses a queue (FIFO) to manage nodes, ensuring that nodes closer to the starting node are processed first, which makes it ideal for finding the shortest path in unweighted graphs.

Algorithm.

1. Initialize:
   (a) Mark the start node as visited.
   (b) Enqueue the start node into an empty queue.
2. While the queue is not empty:
   (a) Dequeue the front node.
   (b) For each unvisited neighbor of that node:
      i. Mark the neighbor as visited.
      ii. Enqueue the neighbor.

Proof of correctness. We prove that BFS discovers every reachable vertex and assigns shortest path distances from the start vertex s.

Let $d(v)$ denote the shortest path distance from s to v in the graph. Let $\hat{d}(v)$ denote the distance assigned by BFS.

Initially, $\hat{d}(s) = 0$. When BFS dequeues a vertex u of distance $\hat{d}(u)$, it sets each unvisited neighbor v to distance $\hat{d}(u) + 1$. Thus,

$$\hat{d}(v) = \hat{d}(u) + 1.$$

To prove correctness, we show $\hat{d}(v) = d(v)$ for all reachable vertices. By construction, BFS never assigns a distance larger than $d(v)$, therefore $\hat{d}(v) \leq d(v)$.

Assume toward contradiction that there exists a vertex v such that $\hat{d}(v) > d(v)$. Consider the shortest path

$$s = v_0, v_1, \ldots, v_k = v.$$

Then v_{k-1} has a shorter BFS distance than v, and since BFS processes vertices in nondecreasing order of distance, v_{k-1} must be dequeued before v. When this happens, BFS will visit v and set its distance to $\hat{d}(v_{k-1}) + 1 = k = d(v)$, contradicting $\hat{d}(v) > d(v)$.

Thus $\hat{d}(v) = d(v)$ and BFS correctly computes shortest paths.

Time complexity. Each vertex is enqueued at most once and dequeued at most once, so all queue operations account for $O(V)$ time.

For every vertex, BFS scans all outgoing edges from that vertex. Since each edge is examined at most one time across the whole run, this contributes $O(E)$ time.

Therefore, the total running time of BFS is

$$\boxed{O(V + E)}$$

when using adjacency lists.

5 DFS

Algorithm (recursively).

1. Initialize:
   (a) Mark all vertices as unvisited.
2. For each vertex that is unvisited, call `DFS(vertex)`:
   (a) Mark the current vertex as visited.
   (b) For each neighbor of the current vertex:
      i. If the neighbor is unvisited, recursively call `DFS(neighbor)`.

Algorithm (iteratively).

1. Initialize:
   (a) Mark all vertices as unvisited.
   (b) Push the start vertex onto an empty stack.
2. While the stack is not empty:
   (a) Pop the top vertex from the stack. Call it the current vertex.
   (b) If the current vertex is unvisited:
      i. Mark the current vertex as visited.
      ii. For each neighbor of the current vertex:
         A. If the neighbor is unvisited, push the neighbor onto the stack.

Proof of correctness. We prove that DFS visits every vertex reachable from the start vertex.

When DFS is at a vertex u, it recursively explores every unvisited neighbor v. Thus every vertex reachable from u via a path of length one is visited. By induction on the path length, assume DFS reaches all vertices at distance k from u. Let x be a vertex at distance $k + 1$. Then there exists an edge (y, x) where y is at distance k. By the induction hypothesis, y is visited, hence DFS eventually processes y and recursively visits x.

Therefore every reachable vertex is visited exactly once.

Time complexity. Each vertex is marked visited exactly once. When a vertex is visited, DFS scans through all edges leaving that vertex. Since every edge in the graph is explored a single time, the total work spent scanning adjacency lists is $O(E)$.

The bookkeeping for marking visited vertices and performing recursive or stack operations adds $O(V)$.

Thus, the total running time of DFS is

$$\boxed{O(V + E)}$$

for adjacency list graphs.

6 Dijkstra's Algorithm

Priority queue terminilogy.

- Key: The value that represents the element's priority. This is the value that the priority queue uses to sort and retrieve elements.
- Value (or Data): The actual data or object associated with the element. This is the information that the user wants to store and retrieve based on its priority.

Algorithm.

1. Initialize:
   (a) For every vertex, set its tentative distance (shortest distance from start node) to infinity.
   (b) Set the distance of the start vertex to zero.
   (c) Mark all vertices as unvisited.
   (d) Insert all vertices into a priority queue keyed by tentative distance.
2. While the priority queue is not empty:
   (a) Extract the vertex with the smallest tentative distance. Call it the current vertex.
   (b) If the current vertex is the target (optional optimization), stop.
   (c) For each unvisited neighbor of the current vertex:
      i. Compute the alternative distance: current distance plus the edge weight to the neighbor.
      ii. If this alternative distance is smaller than the neighbor's current tentative distance, update the neighbor's distance.
      iii. Update the neighbor's priority in the priority queue.
   (d) Mark the current vertex as visited. Its tentative distance is now finalized.

Proof of correctness. We prove that when a vertex u is extracted from the priority queue, $dist[u]$ is the length of the shortest path from the start vertex s to u.

Proof by contradiction. Suppose there exists a shorter path to u that BFS has not discovered when u is extracted. Let $P = s \to v \to \ldots \to u$ be such a shortest path and let x be the first vertex on P that has not been extracted yet. Let y be the vertex before x on P, so y has been extracted earlier.

Because edge weights are non negative and $dist[y]$ is already finalized as the shortest distance to y, relaxing the edge (y, x) must update $dist[x]$ to the correct shortest path value. Therefore x should have a smaller key in the priority queue than u, so it should have been extracted before u. This contradicts the choice of u.

Thus, the algorithm finalizes shortest paths in nondecreasing order of path length. Therefore, $dist[u]$ is correct.

Time complexity. Each vertex is inserted into the priority queue once, and each extract-min operation costs $O(\log V)$, contributing $O(V \log V)$.

For every edge, the algorithm may perform a decrease-key operation. Each such update costs $O(\log V)$, and since there are E possible relaxations, edge processing costs $O(E \log V)$.

Summing both contributions yields a total running time of

$$\boxed{O((V+E)\log V)}$$

which simplifies to $O(E \log V)$ for connected graphs where $E \geq V$.

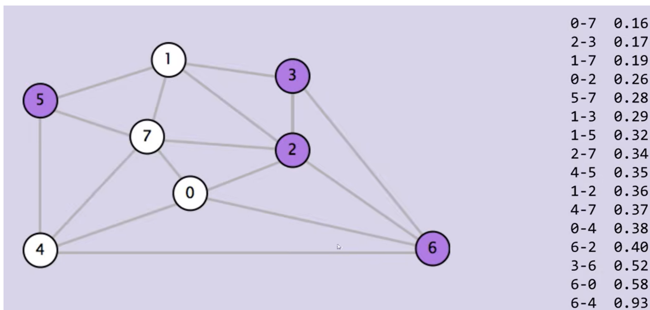
MINIMUM SPANNING TREES

Problem. Given a weighted graph, find a subset of edges that connects the entire graph, has no cycles (forms a tree), and has the smallest total weight possible. When edge weights are distinct, the minimum spanning tree is unique (there only exists one minimum spanning tree).

Number of possible spanning trees. In a given graph $G = (V, E)$, the maxinum possible number of spanning trees produced by this graph is

$$T = \binom{|E|}{|V|-1} - \text{no. of cycles}$$

Crossing Edge Property. Ffor any cut (a partition of a graph's vertices into two sets), the edge with the minimum weight crossing that cut must be part of every MST



In this graph, (0, 2) must be part of every MST, and so is (7, 2), (1, 3), or (1, 5), etc

7 Prim's Algorithm (using binary heap)

Algorithm.

1. Initialize:
   (a) Choose any start vertex.
   (b) Insert all incident edges into a priority queue.
2. While the priority queue is not empty:
   (a) Extract the edge with minimum weight that connects a visited vertex to an unvisited one.
   (b) Add that edge to the MST and mark the vertex as visited.
   (c) Insert all edges from the new vertex into the priority queue.

Proof of correctness. Let T be the tree constructed by Prim's algorithm. At every step, Prim chooses the lightest edge crossing the cut $(S, V \setminus S)$ where S is the set of vertices already in the tree.

By the Cut Property of MSTs: for any cut in the graph, the minimum weight edge crossing that cut belongs to every MST.

Since Prim always chooses such edges, every chosen edge is safe and cannot prevent the existence of an MST containing T. After $n-1$ iterations, T contains $n-1$ edges and spans all vertices, so T is an MST.

Time complexity. Each extract-min operation from the priority queue selects an edge to add to the MST. Since this happens once per vertex, the total cost of these operations is $O(V \log V)$.

Every time a new vertex is added to the tree, the algorithm scans all outgoing edges from that vertex and may update their priorities in the queue. Each edge can cause at most one decrease-key operation, so edge updates cost $O(E \log V)$.

The total running time of Prim's algorithm using a binary heap is therefore

$$\boxed{O(E \log V)}$$

for adjacency list graphs.

8 Kruskal's Algorithm.

Algorithm.

1. Initialize:
   (a) Sort all edges by weight.
   (b) Initialize a union-find structure with each vertex in its own set.
2. For each edge in sorted order:
   (a) If the endpoints are in different sets:
      i. Add the edge to the MST and union the two sets.

Proof of correctness. Kruskal sorts edges in nondecreasing weight and considers them in order. An edge is added only if it does not form a cycle.

By the Cycle Property of MSTs: for any cycle, the edge with the largest weight is not in any MST.

Since edges are processed from smallest to largest, Kruskal never chooses a maximum edge of any cycle. Therefore each selected edge is safe and maintains the possibility of forming an MST. After selecting exactly $n-1$ edges, the resulting graph is connected and contains no cycles, hence it is a spanning tree and an MST.

Time complexity. First, sorting the edges by weight costs $O(E \log E)$.

Then, the algorithm processes each edge once. For each edge, we perform two find operations and possibly one union operation. With union find data structures that use both path compression and union by rank, each of these operations has amortized cost $O(\alpha(V))$, where α is the inverse Ackermann function, which grows extremely slowly and is treated as constant in practice.

Thus, processing edges costs $O(E)$ and dominates only by the sorting step

$$\boxed{O(E \log E) = O(E \log V)}$$

since $E \leq V^2$ and therefore $\log E = O(\log V)$.

BIPARTITE GROUPING

Description. A graph $G = (V, E)$ is **bipartite** if its vertices can be divided into two disjoint sets U and W such that every edge connects a vertex in U to one in W.

Algorithm. (Using BFS or DFS Coloring)

1. Initialize all vertices as uncolored.
2. For each unvisited vertex v:
   (a) Assign v a color (say, red).
   (b) Perform BFS (or DFS) from v:
      i. For each neighbor u of a vertex x:
         A. If u is uncolored, color it with the opposite color of x and continue.
         B. If u has the same color as x, then the graph is not bipartite.
3. If no conflicts are found, the graph is bipartite.

Proof of correctness. We show two implications.

1. If the algorithm finds a conflict (an edge (u, v) with the same color on both ends), then G is not bipartite.
   Proof: when the conflict is detected during BFS/DFS, both u and v have been assigned colors via alternating coloring along root-to-vertex paths in the search tree. Let P_u and P_v be the paths in the search tree from the BFS/DFS root r to u and v respectively, and let w be their lowest common ancestor in the search tree. By construction, the parity of lengths $\ell(P_u)$ and $\ell(P_v)$ determines the colors of u and v. If u and v have the same color, then $\ell(P_u)$ and $\ell(P_v)$ have the same parity. Concatenating the path P_u from u to w, then P_v from w to v, and the edge (v, u) produces a closed walk whose length is odd. This walk contains an odd cycle; hence the graph is not bipartite.
2. If the algorithm finds no conflict, then G is bipartite.
   Proof: suppose the algorithm colors vertices by two colors (call them 0 and 1) such that every time an edge (x, y) is explored, y is assigned color $1 - \text{color}(x)$ (or it was already colored consistently). Then every edge joins vertices with different colors. Let $U = \{v : \text{color}(v) = 0\}$ and $W = \{v : \text{color}(v) = 1\}$. By construction every edge has one endpoint in U and the other in W, so (U, W) is a bipartition and G is bipartite.

Thus the algorithm correctly decides bipartiteness. The BFS/DFS traversal ensures every connected component is examined; repeating from every unvisited vertex covers the entire graph.

Time complexity. Each vertex and edge is visited at most once.

$$\boxed{O(V+E)}$$

9 Basic Idea

Definition. A greedy algorithm builds a solution step by step by repeatedly choosing the locally optimal option, hoping that these locally optimal decisions lead to a globally optimal solution.

Greedy algorithms require two key properties:

- **Greedy-choice property:** A globally optimal solution can be obtained by making a locally optimal choice at each step.
- **Optimal substructure:** Optimal solutions to subproblems combine to form an optimal solution to the entire problem.

Not all problems with optimal substructure satisfy the greedy-choice property. Dynamic programming can often solve such problems when greedy methods fail.

10 Correctness Strategy

To prove a greedy algorithm correct, one typically uses:

- **Greedy-choice lemma:** Show that there exists an optimal solution that begins with the greedy choice.
- **Exchange argument:** For any optimal solution, prove that substituting some part of it with the greedy choice yields another optimal solution. Repeating these exchanges transforms the optimal solution into the greedy one without decreasing its value.
- **Structure induction:** Assume the greedy algorithm is correct for the first k choices, then show it makes a correct choice for the $(k+1)$st.

11 General Template

Many greedy algorithms follow this outline:

1. Identify the locally optimal choice.
2. Sort or prioritize items according to this choice.
3. Repeatedly select the next feasible item.
4. Update the current solution.

12 Characteristics of Greedy Algorithms

- Decisions are final and never reconsidered.
- Running time usually depends on sorting or priority queue operations.
- Greedy solutions are optimal only when the greedy-choice property holds.

13 The Celebrity Problem (Famous Problem)

Problem. There are n people in a room. A celebrity is a person who:

- is known by everyone, and
- knows no one.

You are given access to a function Knows(A, B) that returns whether person A knows person B. The task is to determine the celebrity, using as few calls to Knows as possible.

A class cannot contain two celebrities. If A and B were both celebrities, then A would know no one. In particular A would not know B. But then B is not known by everyone. This contradiction shows at most one celebrity exists.

Algorithm.

1. Initialize a candidate $c = 1$.
2. For $i = 2$ to n:
   (a) If $\text{Knows}(c, i)$ is true, then c cannot be the celebrity, so set $c = i$.
   (b) Else, i cannot be the celebrity, so keep c unchanged.
3. After this pass, c is the only remaining possible celebrity.
4. Verify by checking:
   - for all $j \neq c$, $\text{Knows}(c, j)$ is false,
   - for all $j \neq c$, $\text{Knows}(j, c)$ is true.
5. If both conditions hold, output c. Otherwise, report that no celebrity exists.

Proof of correctness. The algorithm uses a greedy elimination argument.

If $\text{Knows}(c, i)$ is true, then c cannot be the celebrity, since a celebrity knows nobody. Thus i becomes the new candidate. If $\text{Knows}(c, i)$ is false, then i cannot be the celebrity because a celebrity must be known by everyone. Hence c remains a valid candidate.

At each step, exactly one of the two people under consideration is eliminated from celebrity consideration. After $n - 1$ comparisons, only one candidate remains.

To justify correctness, we use an exchange-style argument: for any pair of people (x, y), the rule determines which one could still be part of an optimal solution (in this case, a valid celebrity). Repeating these eliminations reduces any optimal candidate set to exactly the same single person c produced by the greedy algorithm. The final verification step ensures that this candidate truly satisfies the celebrity conditions.

Time complexity. The elimination pass performs $n-1$ calls to Knows. The verification requires at most $2(n-1)$ calls. Total time:

$$\boxed{O(n)}$$

14 Stable Marriage Problem

Description. The Stable Marriage Problem involves pairing N men and N women, each having ranked all members of the opposite sex by preference. The goal is to form marriages where no two individuals would prefer each other over their assigned partners. If no such pair exists, the marriages are considered stable.

Algorithm. (Gale-Shapley)

1. Initialize:
   (a) Every man is free and unmatched.
2. While some man m is free and has not proposed to every woman:
   (a) He proposes to the highest ranked woman he has not yet proposed to.
   (b) If woman w is free, she tentatively accepts m
   (c) Else, if w is tentatively matched to some m' already, she compares m and m'
      i. If she prefers m, she accepts m and m' becomes free
      ii. Else, she rejects m, m stays free and will propose to the next highest ranked woman

Proof of correctness. Let every man propose in decreasing preference order. Since preferences are finite, the algorithm terminates.

Suppose the final matching is not stable. Then there exists a blocking pair (m, w) where both prefer each other over their assigned partners. Consider the moment when m proposed to w. Since m proposes from most preferred to least preferred, he must have proposed to w before proposing to his eventual partner. When m proposed to w, she either was free or rejected him for someone she preferred more. In either case, she ends with a partner she prefers at least as much as m, contradicting that (m, w) is blocking.

Therefore the produced matching is stable.

Time Complexity. Initialization takes $O(n)$ times since we need to go over n mens. The main loop, we iterate over n man, and in each iteration, if a man m is rejected by the woman, he has to propose to the next highest ranked woman. At the worst case, he has to propose to n woman, so the total number of possible proposal can be made is at most n^2. The total runtime is therefore

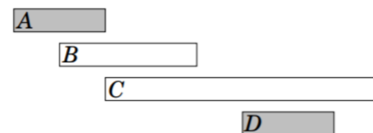
$$\boxed{O(n^2)}$$

15 The Scheduling Problem

Problem. Given N events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially.

Note that this problem is not asking to maximize the area of the intervals, just the amount. The two intervals [0, 1] and [1, 2] are better than the single interval [0, 100].

Idea. We use greedy. The idea is to always select the next possible event that ends as early as possible.



This algorithm always produces an optimal solution. The reason for this is that it is always an optimal choice to first select an event that ends as early

as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until any other event can't be selected. One way the algorithm works is to consider what happens if first select an event that ends later than the event that ends as early as possible.

Algorithm. Given a set of events, each with a start time and finish time:

1. Sort all events in nondecreasing order of their finish times.
2. Initialize an empty set A to store the selected events.
3. Let $t = -\infty$ be the finish time of the last event chosen.
4. For each event in sorted order:
   (a) If the event's start time is at least t:
      i. Add the event to set A.
      ii. Update t to be the event's finish time.

Proof of correctness. Let $A = (a_1, a_2, \ldots, a_k)$ be the set of intervals selected by the greedy algorithm, ordered by finishing time. Let $O = (o_1, o_2, \ldots, o_m)$ be an optimal solution containing the maximum possible number of compatible intervals, also ordered by finish time.

We prove that $k = m$. We use an exchange argument.

The greedy algorithm chooses a_1 to be the interval with the earliest finishing time among all intervals. Since o_1 is a feasible first interval, we have

$$f(a_1) \leq f(o_1)$$

where $f(x)$ is the finishing time of interval x. Replace o_1 with a_1 in optimal solution O. The replacement is valid because a_1 finishes no later than o_1, so a_1 is compatible with all following intervals in O.

Apply the exact same argument inductively to the remaining time interval after a_1. After i steps we have replaced the prefix $(o_1, \ldots, o_i)$ with $(a_1, \ldots, a_i)$ without reducing the size of the solution.

Therefore after k replacements, we obtain a solution of size m that contains all greedy choices $(a_1, \ldots, a_k)$. Hence $k \leq m$. Since O is optimal and the greedy produces k intervals, we conclude $k = m$. Thus, the greedy algorithm is optimal.

Time complexity. Sorting by finish time takes $O(n \log n)$. The greedy scan takes $O(n)$. Total time is

$$\boxed{O(n \log n)}$$

16 Boyer-Moore Majority Voting Problem

Problem. Given a list of n elements, find the majority element.

Requirements:

- With $O(1)$ extra memory so you can't just store a hashmap of frequencies then find the max
- Majority here actually means majority, not plurality, so the element has to take up more than half the list
- The list is assumed to have a majority, so [1, 2, 3, 1] is not a valid input, but [1, 1, 2, 3, 1] is valid.

Algorithm.

1. Initialize a candidate variable $c =$ None and a counter cnt $= 0$.
2. For each element a in A:
   (a) If cnt $= 0$, set $c = a$.
   (b) If $a = c$, increment cnt by 1.
   (c) Else, decrement cnt by 1.
3. After the pass, c is the candidate.
4. Verify in a second pass that c appears more than $n/2$ times.

Proof of correctness. During the scan, the algorithm maintains a candidate c and a counter cnt. Whenever the counter becomes zero, the next element is chosen as the new candidate. Increasing the counter represents pairing the candidate with another occurrence of itself. Decreasing the counter represents canceling a single occurrence of the current candidate against a different element.

Consider pairing every element that is not the majority element with a distinct occurrence of the majority element. Since the majority element M appears more than $n/2$ times, after all such cancellations, at least one copy of M remains unpaired. No other element can remain after all possible cancellations because every non-majority element occurs fewer times than M.

The algorithm performs exactly these cancellations implicitly: each time cnt decreases, a candidate occurrence is canceled with a different element. Any time cnt reaches zero, all previous candidate occurrences have been fully canceled, and the next element begins a new sequence of dominance. Since the majority element can never be fully canceled (it occurs more than all other elements combined), it must be the final surviving candidate at the end of the scan.

Thus, if a majority element exists, it is exactly the element returned at the end of the first pass. The second pass verifies that it appears more than $n/2$ times.

Time complexity. Both passes (voting and verification) take linear time.

$$\boxed{O(n)}$$

Space usage is constant: $O(1)$.

DIVIDE AND CONQUER

17 Basic Idea

Definition. Divide and Conquer is an algorithmic paradigm that solves a problem by:

1. **Dividing** the problem into smaller subproblems of the same type,
2. **Conquering** the subproblems by solving them recursively,
3. **Combining** the subproblem solutions into a solution for the original problem.

This approach is effective when the subproblems are substantially smaller than the original, and when the combination step is efficient.

General Template.

1. If the problem is small enough, solve it directly (base case).
2. Divide the input into pieces (usually equal-sized).
3. Recursively solve each piece.
4. Combine the solutions.

18 Recurrence Relations

Divide and Conquer algorithms typically yield recurrences of the form

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where:

- a is the number of subproblems,
- each subproblem is size n/b,
- $f(n)$ is the cost of dividing and combining the subproblems.

Master Theorem. For $T(n) = aT(n/b) + \Theta(n^d)$ with $a \geq 1$, $b > 1$:

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

19 Merge Sort

Algorithm.

1. Split the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves into a single sorted list.

Proof of correctness. We prove by induction on the input size n.

Base case: when $n = 1$, the list is already sorted.

Induction hypothesis: assume Merge Sort correctly sorts any list of size smaller than n. For a list of size n, Merge Sort divides the list into two parts, recursively sorts each using the induction hypothesis, and merges them. The merge step always selects the smallest remaining element from the front of each list, ensuring the result is sorted.

Thus Merge Sort produces a sorted list for all $n \geq 1$.

Time complexity. Let $T(n)$ denote the running time of Merge Sort on an input of size n.

The algorithm splits the array into two halves, recursively sorts both halves, and then merges them. Splitting requires constant work. The merge step compares and copies each element exactly once, taking $O(n)$ time.

This gives the recurrence

$$T(n) = 2T(n/2) + O(n).$$

To solve it, expand the recurrence:

$$\begin{aligned} T(n) &= 2\,(2T(n/4) + O(n/2)) + O(n) \\ &= 4T(n/4) + O(n) + O(n) \\ &= 4T(n/4) + 2O(n). \end{aligned}$$

After k expansions,

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + k \cdot O(n).$$

The recursion stops when the subproblem size becomes 1, that is

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \log_2 n.$$

Substituting $k = \log n$,

$$T(n) = n \cdot T(1) + (\log n) \cdot O(n) = O(n \log n).$$

Thus, the running time of Merge Sort is

$$\boxed{O(n \log n)}$$

20 Binary Search

Algorithm.

1. Initialize:
   (a) Set low to start index and high to end index.
2. While low $\leq$ high:
   (a) Let mid be the middle index.
   (b) If target equals the middle element return it.
   (c) If target is smaller search left half otherwise search right half.

Proof of correctness. Let $A[1 \ldots n]$ be sorted in nondecreasing order and let x be the target. We prove correctness by the loop invariant method.

Loop invariant: At the start of each iteration of the while loop, if x occurs in A then x lies in the index interval $[low, high]$.

Initialization: before the first iteration we set $low = 1$, $high = n$. If x is in the array then trivially $x \in [low, high]$.

Maintenance: suppose the invariant holds at the beginning of an iteration, and compute $mid = \lfloor (low + high)/2 \rfloor$. Three cases:

- If $A[mid] = x$, the algorithm returns mid and is correct.
- If $A[mid] < x$, then for any index $i \leq mid$ we have $A[i] \leq A[mid] < x$ by sortedness, so x cannot be in indices $\leq mid$. Setting $low \leftarrow mid + 1$ preserves the invariant because any possible occurrence of x remains in $[low, high]$.
- If $A[mid] > x$, then for any index $i \geq mid$ we have $A[i] \geq A[mid] > x$, so x cannot be in indices $\geq mid$. Setting $high \leftarrow mid - 1$ preserves the invariant.

Termination: the loop terminates when $low > high$ or when an index with $A[mid] = x$ is found. If x is present, the invariant guarantees it remains inside $[low, high]$ until some iteration finds $A[mid] = x$. If the loop ends with $low > high$ the invariant implies x is not present, so returning "not found" is correct.

Time complexity. Define the interval length $L = high - low + 1$. At each iteration (unless we immediately return) the new interval length satisfies

$$L' \leq \left\lfloor \frac{L-1}{2} \right\rfloor,$$

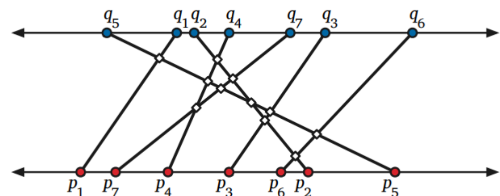
so L decreases by at least a factor of 2 each iteration. Hence the number of iterations is at most $\lfloor \log_2 n \rfloor + 1$, giving time complexity

$$\boxed{O(\log n)}$$

Space complexity: $O(1)$ (iterative version).

21 Crossing Pair Problem

Problem. Suppose we are given two sets of n points, one set $P = \{p_1, p_2, ..., p_n\}$ on the line $y = 0$ and the other set $Q = \{q_1, q_2, ..., q_n\}$ on the line $y = 1$. Consider the n line segments connecting each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time.



Idea. Once the points are sorted along each horizontal line, crossings correspond exactly to *inversions* in a permutation: label the p_i from left to right as $1, \ldots, n$, and record the order of the corresponding q_i. A pair (i, j) with $i < j$ is a crossing pair iff it is an inversion in that permutation. Thus the problem reduces to counting inversions, which admits an $O(n \log n)$ divide and conquer algorithm using a modified Merge Sort.

Algorithm.

1. Sort the points in P by increasing x-coordinate, and relabel them $1, \ldots, n$ in that order.
2. Construct an array $A[1 \ldots n]$ where $A[i]$ = the rank of q_i when the points Q are sorted by increasing x-coordinate.
3. Count the number of inversions in A using the standard divide and conquer inversion-counting algorithm:
   (a) Divide A into two halves.
   (b) Recursively count inversions in each half.
   (c) Count cross-inversions while merging the two sorted halves (each time an element from the right half is placed before an element from the left half).
4. Return the total number of inversions.

Proof of correctness. Sorting P and Q independently produces consistent left-to-right orderings on each line. For two segments (p_i, q_i) and (p_j, q_j), assume without loss of generality that p_i lies left of p_j, so $i < j$. These two segments cross exactly when the order at the top line is reversed, that is, when $x(q_i) > x(q_j)$. This condition is equivalent to saying $A[i] > A[j]$, meaning the pair (i, j) is an inversion of the permutation A. Thus every crossing pair corresponds to exactly one inversion, and vice versa. Since the divide and conquer inversion algorithm counts all and only inversions, it outputs the number of crossing pairs.

Time complexity. Sorting P and Q each requires $O(n \log n)$ time. The inversion counting via Merge Sort takes $O(n \log n)$. Therefore the total running time is

$$\boxed{O(n \log n)}.$$

22 Minimum Distance between Two Points

Problem. Suppose we are given a set of points

$$P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

Find the minimum distance between any two points (x_i, y_i) and (x_j, y_j) in $O(n \log n)$ time.

Idea. The brute-force method for finding the closest pair among n points checks all $\binom{n}{2}$ pairs, which takes $\Theta(n^2)$ time. To improve on this, we exploit geometric structure. If the points are sorted by their x-coordinates, we can split the point set into two equal halves and recursively compute the closest pair in each half. Any closest pair must either lie completely inside one half or cross the dividing line.

A key difficulty is handling pairs that cross the boundary. However, geometry provides a strong restriction: if δ is the minimum distance found in the two recursive halves, then any cross-boundary closest pair must consist of points whose x-coordinates differ by less than δ. Thus only points in a vertical strip of width 2δ centered on the dividing line need to be considered.

Furthermore, once the strip points are sorted by y-coordinate, each point needs to be compared with only a constant number of nearby points in that order. This geometric fact keeps the combination step linear and yields an overall running time of $O(n \log n)$.

Algorithm.

- *Input:* P_x: points sorted by x-coordinate, P_y: same points sorted by y-coordinate
- *Output:* closest pair and their distance

$\textsc{ClosestPair}(P_x, P_y)$

1. If $|P_x| \leq 3$, compute all pairwise distances by brute force and return the minimum.
2. Let mid $= \lfloor |P_x|/2 \rfloor$. Split P_x into Q_x = left half, R_x = right half. Let x_{mid} be the largest x-coordinate in Q_x.
3. Partition P_y into Q_y and R_y by scanning in y-order: for each point $p \in P_y$, if $p.x \leq x_{\text{mid}}$ put p in Q_y, else put p in R_y.
4. Recursively compute:
$$(p_1, q_1, d_1) = \textsc{ClosestPair}(Q_x, Q_y)$$
$$(p_2, q_2, d_2) = \textsc{ClosestPair}(R_x, R_y)$$
5. Let $d = \min(d_1, d_2)$ and keep the better of the two pairs.
6. Build the strip $S = \{p \in P_y : |p.x - x_{\text{mid}}| < d\}$ (points within distance d of the vertical midline).
7. For each index i from 1 to $|S|$: compare $S[i]$ with the next up to 7 points in y-order. For each such comparison:
   (a) Compute the Euclidean distance.
   (b) If it is smaller than d, update d and the best pair.
8. Return the best pair and distance d.

Proof of correctness. By the induction hypothesis, the recursive calls return the correct closest pairs in the left and right halves, so d is at most the true global minimum. Any pair closer than d with one point in each half must lie within distance $\delta = d$ of the dividing line. The strip S therefore contains all candidates for a closer cross-boundary pair.

Because the strip points are sorted by y-coordinate, each point need only be compared with the next at most 7 points: a packing argument shows that in a $d \times 2d$ rectangle one can place at most 8 points at pairwise distance at least d, so after 7 comparisons no closer pair is geometrically possible. Thus the algorithm checks every feasible candidate pair.

The algorithm returns the minimum among: the best left-half pair, the best right-half pair, and the best strip pair, so it returns the correct global closest pair.

Time complexity. The recurrences satisfy

$$T(n) = 2T(n/2) + O(n),$$

since the divide step and strip construction each take linear time, and only a constant amount of extra work is done per point in the strip. By the Master Theorem,

$$T(n) = O(n \log n).$$

Thus the closest pair of points can be found in

$$\boxed{O(n \log n)}.$$

DYNAMIC PROGRAMMING

23 Basic Idea

Definition. Dynamic Programming (DP) is a method for solving optimization and counting problems by breaking them into overlapping subproblems, solving each subproblem once, and storing the results. DP applies when a problem exhibits:

- **Optimal substructure:** an optimal solution can be constructed from optimal solutions of subproblems.
- **Overlapping subproblems:** the same subproblems recur many times.

Two main approaches.

- **Top-down (memoization):** write a recursive solution and store answers the first time each state is computed.
- **Bottom-up (tabulation):** order the subproblems from smallest to largest and fill a DP table iteratively.

General Template. Many DP solutions follow the pattern:

1. Identify the **state**: a parameterization of subproblems (e.g., $dp[i][j]$).
2. Define the **meaning** of the state, usually:
$$dp[\text{state}] = \text{value that solves the subproblem.}$$
3. Derive a **recurrence relation** from optimal substructure.
4. Specify **base cases**.
5. Choose an **order of computation** that respects dependencies.
6. Return the final answer using the table.

24 Correctness Strategy

To prove correctness of a DP algorithm:

- **Base cases:** show they correctly solve the smallest subproblems.
- **Inductive step:** assume all smaller states are correct; the recurrence must combine these states to produce a correct result for the current state.
- **Termination:** bottom-up always terminates; top-down terminates if there are finitely many states and memoization avoids repeated recursion.

25 Time and Space Complexity

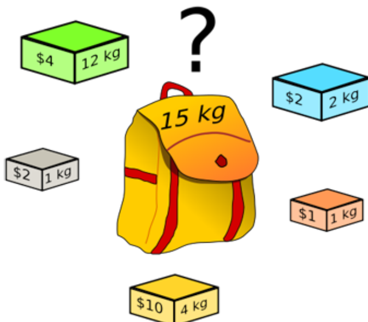
Let S be the number of DP states and let T be the time needed to compute a single state from previously computed ones.

Then the total time is
$$O(S \cdot T).$$
Space is typically $O(S)$, though it can often be reduced using rolling arrays or storing only essential states.

26 Knapsack

Problem. Given n items, where item i has value v_i and weight w_i, and given a knapsack capacity W, choose a subset of the items whose total weight does not exceed W and whose total value is maximized. Each item may be taken at most once.

In summary, we need to maximize
$$\sum_i v_i, \quad \text{such that} \quad \sum_i w_i < W$$



Algorithm.

1. Create a table $\text{DP}[0\dots n][0\dots W]$ where $\text{DP}[i][c]$ represents the maximum value achievable using the first i items with capacity limit c.
2. Initialize the base cases:
$$\text{DP}[0][c] = 0 \quad \text{for all } c, \qquad \text{DP}[i][0] = 0 \quad \text{for all } i.$$
3. For each $i = 1, \dots, n$ and for each capacity $c = 1, \dots, W$, compute
$$\text{DP}[i][c] = \begin{cases} \text{DP}[i-1][c] & \text{if } w_i > c, \\ \max\big(\text{DP}[i-1][c],\ v_i + \text{DP}[i-1][c-w_i]\big) & \text{otherwise.} \end{cases}$$
4. Return $\text{DP}[n][W]$ as the maximum attainable total value.

Proof. We prove by induction on i and c that $\text{DP}[i][c]$ equals the maximum value obtainable using items $1, 2, \dots, i$ subject to capacity c.

Base cases. If $i = 0$, no items are available, so the optimal value for every capacity is zero, which matches the initialization $\text{DP}[0][c] = 0$. If $c = 0$, no positive weight items can be taken, so $\text{DP}[i][0] = 0$ is also correct.

Induction hypothesis. Assume the claim holds for all (i', c') with $i' < i$ or $i' = i$ and $c' < c$.

Inductive step. Consider item i and capacity c.

- If $w_i > c$, item i cannot be included. Any feasible solution must use only the first $i-1$ items, whose optimal value is $\text{DP}[i-1][c]$ by the induction hypothesis. The algorithm assigns this same value, which is correct.
- If $w_i \le c$, every optimal solution falls into one of two categories:
  - The solution excludes item i. The best such value is $\text{DP}[i-1][c]$.
  - The solution includes item i. The remaining capacity is $c - w_i$, and the best achievable value using items $1, \dots, i-1$ and that remaining capacity is $\text{DP}[i-1][c-w_i]$. Including item i yields total value
$$v_i + \text{DP}[i-1][c-w_i].$$

  The optimal choice is the maximum of these two possibilities, exactly as the recurrence defines.

By induction, $\text{DP}[i][c]$ is optimal for all i and c, so $\text{DP}[n][W]$ is the optimal knapsack value.

Time complexity. The table has $(n+1)(W+1)$ entries, each filled in constant time. The total running time is $O(nW)$ and the space usage is also $O(nW)$.

27 The Weighted Intervals Problem

Problem. Given n intervals $I_1, I_2, \dots, I_n$, where each
$$I_i = (s_i, f_i, w_i)$$
has a start time, finish time, and weight, the goal is to choose a subset of pairwise non-overlapping intervals whose total weight is maximized.

Algorithm.

1. Sort the intervals by nondecreasing finish time so that
$$f_1 \le f_2 \le \dots \le f_n$$
2. For each interval i, compute $p(i)$, the largest index $j < i$ such that $f_j \le s_i$. If none exists, set $p(i) = 0$.
3. Initialize the DP table with
$$DP[0] = 0.$$
4. For each interval $i = 1, \dots, n$, compute
$$DP[i] = \max\big(DP[i-1],\ w_i + DP[p(i)]\big).$$
5. Return $DP[n]$ as the maximum achievable total weight.

Proof. We prove that $DP[i]$ equals the maximum weight obtainable from any feasible subset of intervals drawn from $\{1, 2, \dots, i\}$.

Base case. For $i = 0$, the empty set is the only feasible choice and its weight is 0. The algorithm sets $DP[0] = 0$, so the claim holds.

Induction hypothesis. Assume the statement holds for all indices less than i.

Inductive step. Any optimal solution among intervals $\{1, \dots, i\}$ must fall into exactly one of the following cases:

- **The solution excludes interval i.** Then all selected intervals lie in $\{1, \dots, i-1\}$. By the induction hypothesis, the best weight possible is $DP[i-1]$.

- **The solution includes interval i.** Then no selected interval may overlap i, so all others must finish no later than s_i. By definition, these lie in $\{1, \dots, p(i)\}$. The total weight is $w_i + DP[p(i)]$.

The optimal solution must be the better of these two possibilities, which yields

$$DP[i] = \max(DP[i-1],\ w_i + DP[p(i)]).$$

By induction, this holds for all $i \le n$, so $DP[n]$ is the optimal total weight for the entire set of intervals.

Time complexity. The intervals are sorted in $O(n \log n)$ time. Each $p(i)$ can be found in $O(\log n)$ using binary search, and the DP recurrence runs in linear time. Thus the total complexity is $O(n \log n)$.

28 Sequence Alignment (Longest Common Subsequence)

Problem. Given two sequences

$$X = x_1 x_2 \dots x_m \qquad \text{and} \qquad Y = y_1 y_2 \dots y_n,$$

the goal is to find the length of the longest sequence that appears as a subsequence of both.

Algorithm.

1. Create a table $DP[0 \dots m][0 \dots n]$ where $DP[i][j]$ denotes the LCS length of prefixes $X[1..i]$ and $Y[1..j]$.
2. Initialize
$$DP[0][j] = 0 \quad \text{for all } j, \qquad DP[i][0] = 0 \quad \text{for all } i.$$
3. For each $i = 1, \dots, m$ and $j = 1, \dots, n$, compute
$$DP[i][j] = \begin{cases} DP[i-1][j-1] + 1 & \text{if } x_i = y_j, \\ \max(DP[i-1][j],\ DP[i][j-1]) & \text{otherwise.} \end{cases}$$
4. Return $DP[m][n]$ as the LCS length.

Proof. The recurrence follows from considering whether the last characters match. If $x_i = y_j$, any common subsequence must include this matching pair, giving $1 + DP[i-1][j-1]$. If they differ, the optimal subsequence must drop one of the two characters, giving the max of the two smaller subproblems. Base cases correctly give 0 when one sequence is empty. By induction on (i, j), the table entries are correct, so $DP[m][n]$ gives the LCS length.

Time complexity. The table has $(m+1)(n+1)$ entries and each takes constant time, so the running time is $O(mn)$ and the space is $O(mn)$.

29 Shortest Path with Negative Weights (Bellman–Ford)

Problem. Given a directed graph $G = (V, E)$ that may contain negative edge weights but contains no negative cycle reachable from a designated source vertex s, compute the shortest path distance from s to every vertex in the graph.

Idea. Greedy approaches such as Dijkstra's algorithm fail in the presence of negative weights, so instead we examine paths by the number of edges they contain. Any simple shortest path can have at most $|V| - 1$ edges, so we can build a dynamic programming solution where $\text{DP}[k][v]$ represents the shortest distance from s to v using at most k edges. This leads directly to the Bellman–Ford relaxation recurrence.

Algorithm.

1. Initialize a table $\text{DP}[0 \dots |V| - 1][v]$ for all $v \in V$.
2. Set
$$\text{DP}[0][s] = 0, \qquad \text{DP}[0][v] = \infty \text{ for all } v \neq s.$$
3. For each $k = 1, 2, \dots, |V| - 1$ and for each vertex $v \in V$, compute
$$\text{DP}[k][v] = \min\Big(\text{DP}[k-1][v],\ \min_{(u,v) \in E} \big(\text{DP}[k-1][u] + w(u,v)\big)\Big).$$
4. Output $\text{DP}[|V| - 1][v]$ as the shortest path estimate for each vertex v.

Proof. We prove by induction on k that $\text{DP}[k][v]$ equals the minimum possible distance from s to v among all paths using at most k edges.

Base case. For $k = 0$, the only vertex reachable with zero edges is s itself. The algorithm sets $\text{DP}[0][s] = 0$ and $\text{DP}[0][v] = \infty$ for $v \neq s$, which is correct.

Induction hypothesis. Assume the claim holds for all indices smaller than k.

Inductive step. Consider any vertex v. A shortest path from s to v that uses at most k edges must fall into one of the following two cases:

- The path uses at most $k - 1$ edges. By the induction hypothesis, the best such path has length $\text{DP}[k-1][v]$.
- The path uses exactly k edges. Then its final edge is some $(u, v) \in E$, and the preceding portion of the path uses at most $k - 1$ edges to reach u. By the induction hypothesis this portion has minimum length $\text{DP}[k-1][u]$, so the full path has length
$$\text{DP}[k-1][u] + w(u, v).$$

These are the only possibilities. The recurrence in the algorithm takes the minimum over both options, thus correctly computing the shortest distance using at most k edges:

$$\text{DP}[k][v] = \min\Big(\text{DP}[k-1][v],\ \min_{(u,v) \in E} \big(\text{DP}[k-1][u] + w(u,v)\big)\Big).$$

Since any simple shortest path contains at most $|V| - 1$ edges and no negative cycle is reachable, $\text{DP}[|V| - 1][v]$ equals the true shortest path distance for every vertex v.

Time complexity. There are $|V| - 1$ iterations, and in each iteration every edge is relaxed once. The total running time is therefore $O(|V||E|)$.

NETWORK FLOW

30 The Maximum Flow Problem

1. Definitions

- **Flow Network:** A directed graph $G = (V, E)$ with source s and sink t.
- **Capacity $c(u, v)$:** The maximum limit of an edge.
- **Flow $f(u, v)$:** The actual amount passing through.

Rules (Constraints):

1. **Capacity Constraint:** Flow cannot exceed pipe size.
$$0 \leq f(u, v) \leq c(u, v)$$
2. **Conservation:** Flow In = Flow Out (for all nodes except s, t).
$$\sum_u f(u, v) = \sum_w f(v, w)$$

Goal: Maximize $|f|$, the total flow leaving the source.

31 The Algorithm (Ford-Fulkerson)

Core Concept: As long as there is a path with "spare room" (residual capacity), push more flow along it.

Key Terms:

- **Residual Capacity (c_f):** How much space is left?
$$c_f(u, v) = c(u, v) - f(u, v)$$
(Note: Backward edges allow us to "undo" flow, so $c_f(v, u) = f(u, v)$).
- **Residual Graph (G_f):** The graph containing only edges with valid space ($c_f > 0$).

Algorithm Steps:

1. **Initialization:**
   - Set flow $f(u, v) = 0$ for all edges (u, v)
   - Set total flow $F = 0$
2. **Main Loop:** While an augmenting path exists from source s to sink t:
   (a) **Find Augmenting Path:** Find *any* path P in the residual graph G_f where residual capacity
   $$c_f(u, v) = c(u, v) - f(u, v) > 0$$
   (b) **Compute Bottleneck:** Find minimum residual capacity (bottleneck) along path P:
   $$\Delta f = \min\{c_f(u, v) : (u, v) \in P\}$$
   (c) **Update Flow:** For each edge (u, v) in path P:
   - $f(u, v) = f(u, v) + \Delta f$
   - $f(v, u) = f(v, u) - \Delta f$

   (d) **Update Total:** $F = F + \Delta f$
3. **Return:** The maximum flow F from s to t

Specific Implementation: Edmonds-Karp

- Use **BFS** to find the augmenting path.
- This guarantees the *shortest* path (fewest edges).
- Time Complexity: $O(VE^2)$.

32 Proof of Correctness (Max-Flow Min-Cut)

The Theorem. The value of the Max Flow equals the capacity of the Minimum Cut.
$$\max |f| = \min c(S, T)$$

Part 1: Weak Duality (Upper Bound) For *any* flow f and *any* cut (S, T), the flow can never exceed the cut's capacity.
$$|f| \leq c(S, T)$$

Intuition: You cannot push more water than the pipes connecting the two halves allow.

Part 2: The Proof We must prove that when the algorithm stops, the flow is exactly equal to some cut capacity.

Step 1: The Algorithm Stops Suppose the algorithm terminates. This means there is **no path** from s to t in the residual graph G_f.

Step 2: Construct the Cut Let S be the set of all vertices reachable from s in G_f. Let T be all other vertices ($V \setminus S$).

- $s \in S$ (by definition).
- $t \in T$ (because there is no path to t).

This forms a valid s-t cut (S, T).

Step 3: Analyze the Edges Consider any edge (u, v) going from S to T.

- **Forward Edges ($u \in S, v \in T$):** Must be fully saturated: $f(u, v) = c(u, v)$. *Reason:* If it wasn't full, there would be residual capacity ($c_f > 0$). Then we could travel from u to v, putting v inside set S. But v is in T. Contradiction.
- **Backward Edges ($u \in T, v \in S$):** Must be empty: $f(u, v) = 0$. *Reason:* If there was flow, the residual backward edge would have capacity. We could travel $u \to v$ in G_f, putting u in S. Contradiction.

Step 4: Calculate Flow Value The total flow is the net flow crossing the cut:
$$|f| = \sum_{u \in S, v \in T} f(u, v) - \sum_{u \in T, v \in S} f(u, v)$$

Substituting our findings from Step 3:
$$|f| = \sum_{u \in S, v \in T} c(u, v) - 0$$

This is exactly the definition of **Cut Capacity**:
$$|f| = c(S, T)$$

Conclusion: We found a flow $|f|$ equal to a cut capacity $c(S, T)$. Since flow is always $\leq$ capacity, getting them equal means:

1. The flow is Maximum.
2. The cut is Minimum.