Stable matching algorithm (Gale-Shapley)

- **What?** For two groups of elements, where each element has a **preference list** for all members of the other group, create pairs such that there are no two pairs where elements from them would prefer to form a new pair with each other.
- **How?** Each **student** in the queue applies to **companies** in their order of preference until they get **accepted**, either because the company doesn't yet have a student, or they prefer them to their current student. If a student gets **rejected** or dumped, they are put back into the queue.
- **How fast?** O(n^2) in the **worst case** because each student could have to apply to all companies, where n represent the number of students/companies.
- **Why?** Assume the solution is unstable due to S and C not being a pair when they should be for stability. Scenario 1: S applied to another company first -> S prefers it over C -> not unstable. Scenario 2: S applied to C but C rejected S initially or later -> C prefers another student over S -> not unstable. (GS solution is optimal for students)

Algorithm analysis

- **What?** Figuring out how quickly the **runtime** for an algorithm will grow as the **input size** gets larger.
- **How?** O(n^2) is the is a set of functions with a max **running time** (worst case): c * n^2, where n is the size of the input. Ω(n^2) is the equivalent for the **minimum running time** (best case). If a function is in both sets, it is said to be Θ(n^2). An algorithm with **polynomial running time** is regarded as (comparatively) efficient. O(log n): searching in a sorted array O(n + m): visiting all n nodes in a graph with m edges O(n log n): sorting an array O(n^2): two for loops O(2^n): all subsets of n objects O(n!): all permutations of n objects

Hash Tables

- **What?** Storing **key value pairs** in a datastructure with a **constant average lookup time**, provided that a good **hash function** is used and α is low.
- **How?** Hashing can lead to **collisions**, so there are two methods of dealing with them.

1. **Open addressing**, which means that any item can be stored at any position, even if it's not the one calculated by the hash. 1a. **Linear probing** means trying to store the pair at adjacent positions, until an available one is found, starting from the already occupied one from the hash function. 1b. **Quadratic probing** is similar, but the step size will grow quadratically with each failed attempt, instead of being constant at 1. 1c. **Double hashing** means that there is a second hash function that gets used when there is a collision, to determine the number of steps to move along for the next attempt. A pair where a collision has previously occurred can be deleted if the key with the same hash is then moved to the left, where the first one originally was, so that the probe doesn't fail immediately.
2. **Closed addressing** aka **chaining**, means that an index in the hash table will refer to a list and if there is a collision, the new element will simply get added to the list. Will be slow if α gets big.

The likelihood of collisions can also be reduced by making the **hash table bigger** than theoretically necessary. This can be measured with the **load factor** α = n / m, where n is the number of items stored, and m is the size of the underlying array. Lower is generally better, and for **quadratic probing** you can even guarantee there will be less than m / 2 probes if α < 1 / 2 in the event of a collision.

Graph Representations

- **What?** Describing a **graph** with conventional data structures, to work with it programmatically. Represented as n being the number of nodes and m being the number of edges
- **How?**

1. **Adjacency matrix** of size m*n. An edge is stored as m[i][j] and m[j][i] which means storing half the matrix is enough. It's also smaller for very dense graphs. Θ(n^2) space. Θ(1) time to check if there is an edge i − j. Θ(n) time to find all neighbors of a node. Θ(n^2) time to list all edges.
2. **Adjacency list**. Every node has a list of neighbors. **degree(n)** is the number of neighbors. Θ(degree(n)) time to find all neighbors of a node. Θ(m) time to list all edges.

Depth First Search

- **What?** Figuring out if a **target node** is reachable from the **root node**
- **How?** Start with a **stack** that contains the root node. While the stack is not empty, **pop** the current node from the stack, iterate over each neighbor, mark it as **visited**, and add it to the stack. Repeat this process until the target node has been found or the stack is empty, in which case no path between the nodes exists. Store the **predecessor** of each node for **path reconstruction**.
- **How fast?** O(n + m) because the outer while loop has to iterate through every node (n), and the inner for loop through every edge (m).
- **Why?** Every node is visited, so basically a brute-force approach.

Breadth First Search

- **What?** Finding the **shortest path** between two nodes in a graph
- **How?** Start with a **queue** that contains the root node. While the queue is not empty, **pop** the current node from the queue, iterate over each neighbor, mark it as **visited**, and add it to the queue. Repeat this process until the target node has been found or the queue is empty, in which case no path between the nodes exists. Store the **predecessor** of each node for **path reconstruction**.
- **How fast?** O(n + m) because the outer while loop has to iterate through every node (n), and the inner for loop through every edge (m).
- **Why?** Every node is visited, so basically a brute-force approach. Since the exploration happens one layer at a time, we are guaranteed to find the shortest path between two nodes.

Tarjan's algorithm

- **What?** Finding all **strongly connected components** in a directed graph. A strongly connected graph is one where every node is reachable from every other node.
- **How?** We're gonna give every node an **id of unvisited**. Then, we'll use **DFS** and assign every node we visit an id and an equal **low-link value** (id of the node with the lowest id that's reachable from the starting node). We're also gonna add every node we visit to a **stack**. When DFS moves back up, if the previous node is on the stack then min the current nodes low-link value with the previous nodes value. After visiting all neighbors, if a nodes id is equal to its low-link value then pop nodes off the stack until the current node is reached. Repeat this process until the entire graph has been explored. At the end, nodes with the same low-link value will be part of the same component.

Two greedy algorithm proof techniques — main ideas

- **What?** Proving that the solution from a **greedy algorithm** is optimal
- **How?** Approach one: Prove that at every step of the problem the greedy algorithm makes a choice that's at least as good as the greedy algorithm. Approach two: Assume that there is an **optimal solution** and transform it into the greedy solution by only performing swaps that don't make the solution worse.

Greedy scheduling algorithms

- **What?** 1. Maximize the number of **available time blocks** chosen to be included in a schedule. 2. Minimize the maximum **delay** of any task with a deadline and a time length.
- **How?** 1. At every step select the time block that finishes earliest and remove any overlapping ones. 2. Sort the tasks by their deadline such that there is no downtime between them.
- **How fast?** O(n log n) since times are first sorted and then selected in linear time.
- **Why?** Let S and T be the greedy and optimal sequences of intervals. By induction it can be proven that that every finishing time in S at position will be smaller or equal to the finishing time in T. By contradiction it can then also be proven that the size of S is equal to or smaller than T. 2. Let T be the optimal schedule and identify any **inversions** where a later deadline task is scheduled before an earlier one. **Swap** them and prove that it doesn't increase the maximum delay. Once there are no more inversions, the optimal schedule will be equal to the greedy one.

Dijkstra's Algorithm

- **What?** Find the **shortest distance** from the root node to every other node while minimizing total weight, as long as the edge weights are **non-negative**. Otherwise the shortest path to an already explored node might only be found after the fact, which this greedy algorithm can't detect.
- **How?** Start at the root node and keep a **running total** of the minimum weight needed to reach every other node in the graph. Traverse the graph by picking the smallest weight to an **unvisited node** and

update the running total if a shorter path from the root node to any other node opens up.

- **How fast?** Each node must be explored, so O(n) iterations for the while loop, and in every iteration, the smallest edge weight must be found, which takes O(log n), so O(n log n) for the traversal part of the process. Each selected node also needs to check each of its neighbors and potentially update the total distances. Updating (reducing a key) is assumed to be O(log n) and this needs to potentially be done for every edge, so O(m log n) for the path weight updating part of the process. In total, that leads O(n log n + m log n), which can be simplified to O(m log n) if m >= n-1, which is true since all nodes are reachable from the root node.
- **Why?** Proof by **induction** goes like this: In the first step, assume that the set of explored nodes only contains the root node s, so d(s) is 0. Then assume that d(v) is the shortest path for all nodes in a set of explored nodes bigger than one. In the next step, let v be the next node added. Then assume that a shorter path from s to v exists that goes through some node y that has not yet been explored. In that case, d(y) < d(v) which is not possible since y would have been added before v if that were the case. Therefore d(v) must be the shortest path from s to v.

Safe edges when creating a minimum spanning tree

- **What?** An edge that can be added to a **MST** so that it's still a MST afterwards.

- **How?** Start with a partial MST T and make a **cut**. Select the **smallest edge** that crosses the cut. If it's already in T, there's nothing to prove. If not, adding it will form a **cycle**, which we can remove by taking away another edge in the cycle that crosses the cut. This way, the total weight won't increase and we that this edge is a **safe edge**.

Jarnik's Algorithm

- **What?** Construct a **minimum spanning tree**, a graph where all nodes are still connected while the total edge weight is minimized.
- **How?** Start at any node, and select the smallest **safe edge**. Repeat the process until all nodes have been visited. The resulting graph is the MST.
- **How fast?** O(m log n) because inserting each node into the **priority queue** (binary heap) that's used to find the smallest edge weight is O(log n), and reducing the key of a node is also O(log n), where a key needs to be reduced at most once for every edge.
- **Why?** Because it is a **greedy algorithm** that starts at any node and then builds up the MST by traversing the graph while always choosing the smallest edges to nodes that are not yet included in the MST.

Kruskal's Algorithm

- **What?** Construct a **minimum spanning tree**, a graph where all nodes are still connected while the total edge weight is minimized.
- **How?** Sort all edges in the graph by **weight**, and continuously add the smallest edge that does not result in a **cyclic graph**, i.e. doesn't connect two nodes that are both already in the MST, to the MST.
- **How fast?** The most expensive operation is the sorting of edges, which takes O(m log m), which is equivalent to O(m log n), since m = n^2 in a dense graph.
- **Why?** Because it is a **greedy algorithm** that starts at any node and then builds up the MST by traversing the graph while always choosing the smallest edges to nodes that are not yet included in the MST.

Union Find

- **What?** Determine if an edge is safe by checking if adding it will lead to a **cycle**
- **How?** Start with a set of sets where each node in the graph is in its own set. The **union operation** will merge two given sets, and the **find operation** will return the set for a given node. If two nodes connected by an edge are in the same set, we know that they are already connected and we can check another edge. Once we have found a safe edge, we merge the sets of the nodes that are connected by the new safe edge.

Since each node has a **parent attribute** and a subtree is identified by its **root node**, the naive implementation for **find** is to simply search through the chain of parents until you reach the root node, and letting the root of one set point to the root of the second set for merging. This leads to **deep trees** that are slow to traverse.

These operations can be sped up with **path compression** and **union-by-size**. Union-by-size means that the tree with the larger size will always be the parent tree. This means that we will have to traverse fewer nodes because the tree is more shallow. Path compression means that when we first try to find a node, and we pass by other nodes along the way, all of those nodes will have the same root, so we can simply make all of those point to the root directly.

Main ideas of divide and conquer

- **What?** Solve problems that involve n items and two nested for loops as efficiently as possible.
- **How?** Continuously **divide** the items in half until the problem is trivially solvable, then continuously **combine** the solutions of the small subproblems until it encompasses the entire input.
- **How fast?** Commonly O(n log n), but see **master theorem**

Main ideas of master theorem

- **What?** Calculating the **time complexity** of a divide and conquer function
- **How?** The running time can be analyzed with **recurrence relation**, and specifically with an equation of the form T(n) = aT(n/b) + f(n), where a is the number of branches, b is the division factor for the input, and f(n) represents the cost of dividing the problem and combining the subproblem solutions. We can determine the time complexity by figuring out which value this function will converge towards for a large n, with three cases. n^log_b(a) describes the complexity at increasing division levels.

1. If f(n) grows slower than n^(log_b(a)), then the total subproblem size dominates the time complexity, which is Θ(n^(log_b(a))).
2. If f(n) grows at the same rate as n^(log_b(a)), then the work done at each level and the total subproblem size are in balance, and the time complexity includes an extra logarithmic factor: Θ(n^(log_b(a)) * log n).
3. If f(n) grows faster than n^(log_b(a)), then the work done at each level dominates the time complexity, which is Θ(f(n)).

Jarvis march

- **What?** Calculate the smallest **convex shape** that encompasses all given points, known as the **Convex Hull (CH)**.
- **How?** Start with the **leftmost point** (smallest x-coordinate) and create a vector from the current point to any potential next point in the CH. Then iterate over all other possible vectors from the current point to all other points that's not the current next one. In each iteration, compare the current vector to the next point vector, and if it is to the **left** of the next point vector, it will become the new next point. After the loop, set the next point as the current point and pick new potential next point. Then repeat the process until the current point is equal to the initial point. A vector is to the left if the third value of the vector product is greater than zero.
- **How fast?** O(nh) where n is the number of points, and h is the number of points in the CH. This is because the algorithm needs to iterate over every point for every point in the CH to find the next one. If all the points are part of the CH, it's O(n^2).

Graham scan

- **What?** Calculate the smallest **convex shape** that encompasses all given points, known as the **Convex Hull (CH)**.
- **How?** Start with the **bottommost point** and then sort the points by their **polar angle** in counterclockwise order around the starting point. Push the first three points onto a **stack**, with the furthest one at the top. From the first point in the stack, construct two vectors, the first to the second point in the stack, and the second to the next point in the sorted list. If the second vector is to the **right**

of the first one (see Jarvis march explanation), **pop** the top point from the stack, otherwise push the point from the list onto the stack. Repeat the process until every point has been visited.
- **How fast?** Finding the minimum point is O(n), sorting the points is O(n log n), and manipulating the stack is also O(n) since every point can be pushed and popped at most once. So the total time complexity is O(n log n).

Preparata-Hong

- **What?** Calculate the smallest **convex shape** that encompasses all given points, known as the **Convex Hull (CH)**.
- **How?** Sorted the points by their y-coordinate, **divide** them into one upper and one lower group and compute their CHs separately. This can be done trivially if there are 3 points or fewer. To then **merge** the CHs compute the slopes (k-values) of all the line segments in both groups (

α and β) and also all the line segments connecting the two hulls (γ). By comparing α and β with γ the upper and lower tangents connecting the hulls can be determined.

To determine if p is on a **line segment** is between q and r, you can form the vectors qp and qr and compare their cross and dot products.

- **How fast?** O(n log n)

Finding nearest points in a plane

- **How?** Divide the points into left and right regions and compute the minimum distance in those separately. This can be done trivially if there are 3 points or fewer. When merging, select the lower from the two minimums as the new minimum. Use this distance d to select all points whose distance from the middle line is less than d. For every point in this strip, compare it to at most its 8 closest neighbors by y-value. If there were more than eight, a shorter distance than d would have been found, so that's not possible. If one of these comparisons gives a distance less than d, it will be the new minimum d. After the comparisons are done, d will be the global minimum.
- **How fast?** Sorting the points has a time complexity of O(n log n). All the operations in the function are done at most n times, and the recursion has log n levels, because of the division, so O(n log n) for the function itself. Therefore the total time complexity is O(n log n).

Main ideas of dynamic programming

- **What?** Solving a problem by dividing it into smaller **subproblems** that might appear multiple times (overlapping subproblems, different to divide and conquer) and that can be solved by combining solutions to **optimal subproblems** (optimal substructure).
- **How?** Compute the answer to a subproblem as it appears once and then store the result, so it can be reused later (**memoization**), or compute every subproblem ahead of time and store the answers in a **table** (**tabulation**) to reuse them later.

Bellman-Ford algorithm

- **What?** Find the shortest distance from the root node to every other node while minimizing total weight.

- **How?** Keep a running total of the smallest path leading every other node. For every node in the graph, starting with the root node, examine all the outgoing edges and calculate every possible new path that opens up. If it's shorter than the current minimum, it becomes the new minimum. Repeat this process at most (graph size - 1) times. If there are any changes after one one extra iteration, there is a negative cycle.
- **How fast?** In the worst case, we explore every edge in every iteration, so it's O(nm).

Sequence alignment

- **What?** Quantify how similar two strings of characters are in different contexts, using a similarity matrix for the characters and a cost of inserting a new character.
- **How?** Use a **dynamic programming** approach and view the alignment as a graph, with one corner representing two empty strings and the other corner representing the optimally aligned strings A and B. At every step there are three options, 1. keep everything as it is, 2. insert a character into A, 3. insert a character into B. The problem can be solved be choosing the option with the minimum cost at every step, taking into account all previous steps. The solution relies on **path reconstruction** in the graph, and to avoid computing the entire graph at every step, we will iteratively (or recursively) calculate the cost of every possible step ahead of time, and store each optimal step with minimum cost in a **matrix**. It can also be solved with a **shortest path** algorithm.
- **How fast?** We're building up a m * n matrix, where m and n is the length of the first and second sequence respectively. Each element in the matrix needs to be populated, which takes constant time, so the total time complexity is O(mn).

Ford-Fulkerson algorithm

- **What?** Calculate the maximum **flow** from a **source node** to a **sink node** in a graph where the edge weights represent capacity.
- **How?** Find an **augmenting path** (a path where the residual capacity is greater than zero for all edges in the path) and calculate the **bottleneck capacity** (the minimum residual capacity) in that path. Add this value to the running total network flow and for each edge in the path, decrease its capacity by the bottleneck capacity and increase the capacity of the **reverse edge** by the same amount. Repeat this process until no more augmenting paths can be found to calculate the maximum network flow. The order in which the paths are found does not matter.
- **How fast?** Finding a path with **BFS** or **DFS** has a time complexity of O(n + m), which simplifies to O(m) if you assume there are more edges than nodes. In the worst case, each iteration with a new path will increase the maximum flow by one, so the total time complexity is O(Cm), where C is the maximum flow.
- **Why?** At the end of the algorithm, you can make a cut in the residual graph such that A will have the nodes reachable from s and B will have all other nodes. Since there is no more s-t path any edge that crosses over from A into B must have maximum flow, and therefore the respective reverse edges must have a flow of zero. The total flow from the source to the sink will equal the capacity of this cut. By the **max-flow min-cut theorem**, the flow value that the algorithm has found is equal to the capacity of this minimum cut, proving that the flow is maximized.

Biparite graph matching

- **What?** In a graph where the nodes are partitioned into two sets such that are no edges between any nodes in the same set, we want to create as many one-to-one **matchings** as possible.

- **How?** Add a **source node** and edges from it to every node in the first set and add edges from every node in the second set to a **sink node**. Assign a capacity of 1 to every edge, and maximize the network flow with the **Ford-Fulkerson algorithm**. This maximum flow will be number of matchings and the augmenting paths will correspond to the matchings.

Preflow-push algorithm

- **What?** Calculate maximum flow in a flow network.
- **How?** Idea 1: **Preflow**, hypothetical flow value where the what goes into a node does not have to come out of a node. Idea 2: **Node height**, where the source node has height n and the sink node has height 0. All other nodes start at zero, but get raised by one step at a time. We start by putting the maximum possible flow through all outgoing edges from s, which will lead to preflow in those nodes. Now, if there is **excess flow** at a node u and a neighbor v that has a lower height, we push as much flow as possible from u to v. If there is excess flow but there is no neighbor with a lower height, we **relabel** that node so that the height is increased by one. Apply these operations to nodes with excess flow until only t has excess flow. At that point, the preflow will be equal to real flow, and the excess flow of t will be equal to the maximum flow.
- **How fast?** It can be proven that the maximum height of a node is 2n - 1, so it follows that that there will be less than 2n^2 relabel operations. For the push operation, we differentiate between saturating and non-saturating ones, where it's saturating if the capacity of the edge will be completely filled by that push. It can be proven that the maximum number of operations in these respective push categories is 2nm and 4mn^2, so non-saturating pushes are the limiting factor, which gives a time complexity of O(4mn^2).
- **Why?** Apart from the source and sink nodes having the correct compatibility, the preflow and height function are said to be compatible if for every edge (v, w) v is at most one higher than w. If v is higher than w, a push is performed. Otherwise a relabel is performed, and since there is a tolerance of one, compatibility will be maintained even after a relabel. Since compatibility means that there is no augmenting path in the residual graph, we know that the preflow at the end will be a valid maximum flow.

Linear initialization of a binary heap from an array

- **What?** Constructing a **binary heap** directly from an **unsorted array** in linear time.
- **How?** Treat the given array as a bunch of binary heaps of size 1. To select all nodes that aren't **leaves**, start at the middle of the array and iterate towards the start. In every iteration, move the current element **down** by swapping if needed to maintain the heap property, so that every node will be the **root** of a valid heap.
- **How fast?** Each move down is O(log n) and it needs to be done at most n times, so that would be O(n log n). However, by describing how the work increases as the height increases with a **geometric sum**, you can arrive at the conclusion that O(n) is more accurate.

Hollow heap with multiple roots

- **What?** A heap that consists of a set of **trees** and a list of multiple **root nodes**.
- **How?** A node represents a key value pair, where the value is a pointer to

an item that might not exist, in the case of a **hollow node**. To maintain the heap order on **delete-min** and **merge**, a **link operation** is used, where the keys of two nodes are compared and the one with the smaller

keys becomes parent of the other. **Rank**: Each node also has a rank that represents the number of children, this gets incremented for the parent on a link operation. **Insert**: When a new element is inserted, a new node is created and added to the root list. The new node is then checked to see if it should be the new **minimum node**. **Decrease key**: If the element is a root, then the key is simply reduced — and check if this is the new min. If not, a new root is created with the element. The element is then moved from the previous node which becomes hollow. Some of the children are moved to the new node as well. **Delete**: If the deleted element is is not the minimum, the node with it simply becomes hollow and we are done. If it is the minimum element, all hollow root nodes are destroyed by making their children new full root nodes. To reduce the number of root nodes, a number of link operations are performed.

- **How fast?** The difference compared to binary heaps is that hollow heaps have **constant time insert** and **reduce-key** operations.

Complexity classes

- **What?** Categorize problems into how **difficult** they are to solve.
- **How? P** means that a problem can be solved in polynomial time, and **NP** means that a problem can be verified in polynomial time. **NP-complete** means that any problem in NP can be reduced to it and that it itself is in NP. **NP-hard** just means that any problem in NP can be reduced to it. A **reduction** is essentially a transformation from one problem into another.

Independent set

- **What?** Find the **largest possible subset** of nodes S in a graph (V, E) such that none of them are **neighbors**.

Vertex cover

- **What?** Find the **smallest possible subset** of nodes S in a graph (V, E) such that one of every edge is included. If we know that S is an **independent set**, we can also deduct that V - S must be a **vertex cover**. The same thing can be done the other way, which means you can also come up with polynomial time reduction functions both ways, which means both of these problems have the same complexity.

Set cover

- **What?** Find the **smallest possible number** of **subsets** of S such that their **union** is S. The vertex cover problem can be reduced to a set cover problem by letting S = E.

Circuit satisfiability

- **What?** Given a **Boolean circuit** with n inputs and one output, determine if there exists an **assignment** of the input values such that the output is 1.
- **Why? Verifying** the output of a circuit is simple, so it is NP. To prove that it is also **NP-complete**, we can construct a circuit that will simply implement any **verification algorithm** for any problem in NP, where the first n bits represent the input to the problem, and the remaining bits represent the solution candidate. Then the circuit will output 1 iff it's a valid solution. This is possible because any algorithm can be computed with **logic gates**. Since any such problem in NP can be reduced to CSAT, CSAT is NP-complete.

Formula satisfiability (SAT)

- **What?** Any circuit can be described with **variables** and the operators **AND, OR, and NOT**. Do this efficiently, rewrite the expression to **conjunctive normal form** with the laws of **boolean algebra** and make new variables for statements that appear multiple times. This problem is also NP-complete since it itself is NP just like CSAT, and CSAT can be reduced to it.

3-SAT

- **What?** A special case of the SAT problem where each clause has exactly **three terms**. It can be reduced to SAT, so it is also **NP-complete**.

Hamiltonian cycle

- **What?** Starting from an **arbitrary node** in a graph, find a path through the graph that **visits every node exactly once** and comes back to the starting node.
- **Why? Checking** that the output is a cycle and that it contains every node once is easy, so it is in NP. A 3-SAT problem can be reduced to this problem by representing a **boolean algebra clause** as a graph where each variable corresponds to one row of nodes and every clause corresponds to one node where the edges are determined based on the variables they contain. A variable having the value 1 means going right and 0 means going left. The entire expression should output 1 if the inputs are chosen such that it results in a correct cycle. By reducing a 3-SAT problem into a Hamilton cycle problem we have shown that it is **NP-complete**.

Travelling salesperson problem

- **What?** "Given a list of **cities** and the **distances** between each pair of cities, what is the **shortest possible route** that visits each city exactly once and returns to the origin city? The total weight should be minimal. The Hamilton Cycle problem can be reduced to this problem by treating nodes in the graph as cities and assigning a different distance to pairs depending on if an edge connects them in the graph. This is possible because a any complete tour in this problem is the same thing as a Hamilton cycle. Since the Hamilton cycle problem is NP complete, and this problem can't easily be solved, it is also **NP-complete**.

Graph coloring

- **What?** Assign each node in a graph one of **k different colors** such that no connected nodes have the same color. Since the 3-SAT problem is NP-complete and it can be reduced to the 3-coloring problem, as well as being NP, the graph coloring problem is also **NP-complete**.

Linear and integer linear programming main ideas

- **What?** Maximize a given **objective function** within two **constraint functions**
- **How?** The set of all points that satisfy the constraints forms a **feasible region**. This region is **convex** and can be visualized as the intersection of several **half-planes** (constraints). The **optimal solution** to a linear program, if it exists, will be at one of the vertices (corner points) of the feasible region. A commonly used algorithm to solve LP problems is the **Simplex algorithm**, which moves from one vertex to another, along the edges of the feasible region, to find the optimal vertex.

For **integer linear programming** we can start by solving the LP problem without the integer requirement first. Then, we can use **branch and bound** to divide the problem into two new linear programs, where one has the additional constraint of having to be above the rounded initial solution, and the other where it has to below the rounded solution. Once we have found an integer solution, we remember it and stop exploring (bound) branches whose solution is lower than the current optimum. At the end, we will have the optimal integer solution.

Graph coloring example with integer linear programming

- **What?** Assign colors to the nodes of a graph such that no two adjacent nodes share the same color, using the **minimum number of colors**.
- **How?** Define n **binary variables** where each one represents if a certain color is being used or not. Do the same thing for every node. The **objective function** will be minimizing the sum of the overall variables. The first **constraint** will be that any node can only have one color, so the sum of all node variables should be equal to one. The second constraint is that **neighboring nodes** should have different colors, so the sum of the node variables for color i must be smaller than the overall variable for color i.

Instruction scheduling example with integer linear programming

- **What?** Schedule instructions in a **pipeline** of a CPU to **minimize delays**. The instructions have **dependencies** and different **execution latencies**.
- **How?** Let x_ij be a **binary variable** indicating that instruction i is scheduled in cycle j. The first constraint is that the instruction i has to appear exactly once, so the sum of all variables where x_i has to equal 1. The second constraint is that all instructions in a cycle cannot exceed the limit r, so the the sum of variables x_j is less than r. The third constraint is that **dependencies** need to be fulfilled. These dependencies can be modeled as a graph with edges (k, i) where i is dependent on k, so the cycle in which k is scheduled plus its latency must be smaller than the cycle of i.