PotW 2: Deck of Cards

- Algorithmic description

Given an array of cards, calculate the sum between two indices which is the nearest to a given value k: $|k - \sum_{k=i}^{j} v_k|$

- Solution

Sliding window to discover sums between two indices i and j, keep the ones that minimize the function written above.

Complexity O(n)

Week 2: Burning Coins

- Algorithmic description

Given an array of coins, calculate the largest guaranteed amount of coins in a one-to-one setting, where one could pick the leftmost or the rightmost coin.

- Solution

DP with a condition on the turn: if the turn is mine, maximize the sum;

If the turn is the other’s, play to minimize (worst-case)

Week 2: Beach Bars

- Algorithmic description

Given an array of locations and a scope (within 100m), find the maximum number of locations covered by the scope [x-100, x+100], the minimum largest distance in that interval and all the possible locations.

- Solution

Sort the array based on the positions and do a sliding window:

Scan until the distance between x[i] and x[j] is 200. Calculate the number of positions as j-i and the distance as ceil((x[j-1]-x[i])/2.0).

Update best results in a results array. If num and dist are the same as best, just add a new location, otherwise clear the array.

Insert function: if((par[j-1]-par[i])%2 == 0)

```
                    results.push_back((par[j-1]-par[i])/2 + par[i]);
else{
                    results.push_back(floor((par[j-1]-par[i])/2.0) + par[i]);
results.push_back(ceil((par[j-1]-par[i])/2.0) + par[i]);  }
```

Week 2: Defensive Line

- Algorithmic description

Given an array of defense values and an attack value k, find the maximum number of attacked defenders: $\sum_{i=0}^{m-1} b_i - a_i + 1$ by choosing non-overlapping intervals [a, b] such that the sum of defense values over each interval is equal to k.

- Solution

Sliding window to calculate the k-sum starting from every index.

Knapsack DP to select starting points. Don’t select a starting point if no k-sum is found starting from that index.

- Take home

Switching to different variables through precomputation is sometimes useful to simplify the DP relation.

Week 2: The Great Game

- Algorithmic description

Given a set of paths from 1 to n, play a game where you must move one of two meeples depending on the parity of the round number. It could be the one you own or not.

Trick: explore the moves for the two meeples separately, they are not linked.

Output who wins, namely who's the owner of the first meeple to arrive at n.

- Solution

DP, if it's the owner's turn play to minimize the number of moves, otherwise play to maximize.

Edge case: if the number of moves are the same for the two meeples, its parity is a tie-breaker.

PotW 3: From Russia With Love

- Algorithmic description

Given an array of coins and a certain number of passengers, calculate the largest guaranteed amount of coins one could gain. Extension to Burning Coins.

- Solution

Similarly to Burning Coins, play to maximize if it’s my turn, play to minimize others’ if it’s not my turn (worst case) => DP.

DP table: left index, right index, bool turn (mine or not)

DP state can further be reduced by removing the turn variable (it can be shown it’s useless), but it’s not necessary to achieve full points.

Week 3: Hit?

- Algorithmic description

CGAL toy problem, given a ray and a set of segments, check if there exists an intersection between the ray and a segment

- Solution

Since it is a predicate-only problem and the input size can be stored in *double*, we could use the EPIC kernel without any loss.

To make the program faster, check for intersections only if one has not been found yet (considerable speedup).

Week 3: First Hit

- Algorithmic description

Given a ray and a set of segments, check if there exists an intersection between the ray and a segment and find the nearest one.

- Solution

Since this problem requires constructions, we must use the EPEC kernel.

The trick here is to avoid useless intersection computations by clipping the ray always to the nearest intersection found. This way, only nearer intersection points than the current one are computed, for a total of O(logn) in a random sequence.

However, this strategy may not reduce the number of computations in case of an adversarial input, namely where every segment has a nearer intersection than the preceding ones. Since the input order is irrelevant in this problem, a random_shuffle() is advised to avoid this annoying case.

Week 3: Antenna

- Algorithmic description

Given a set of points, find the radius of the minimum enclosing circle.

- Solution

The algorithm and library methods to compute the minimum enclosing circle can be found on the tutorial slides.

This is one of the rare cases where using the EPEC_with_sqrt kernel is unavoidable, as we need to output the radius of the minimum enclosing circle.

Week 3: Hiking Maps

- Algorithmic description

Given a set of segments and a set of triangles (the input is made of pairs of points that define the boundaries of each), find the minimum number of triangles that cover together the whole set of segments

- Solution

The problem can be split into two subproblems:

1. Find triangles that contain a given path (This could be done avoiding constructions: FIRST orient the pairs of points so that the other sides of the triangle always lie on the left, THEN check that the extremes of a segment don't lie at the right side of every pair with CGAL::right_turn(q1, q2, extreme)). Store for every triangle boolean values for every segment (the latter is contained or not).
2. Kind of sliding window (quadratic) to check if the interval of triangles [b, e] contains ALL of the segments.

- Take home

Constructions can be avoided and allow to pass with a naiver solution.

PotW 4: The Fighting Pits of Meereen

- Algorithmic description

Given a queue of k distinct types of elements $(2 \leq k \leq 4)$, an excitement function which asks for the last $2 \leq m \leq 3$ elements per selection (1000 times the number of distinct types of people minus the abs diff of number of elements between two sets), calculate a partition order between two sets so to maximize the excitement function.

- Solution

DP approach, DP state keeps track of the number of elements selected in both sets, as well as the last m types of elements assigned to each set.

To sum up: starting index in the queue, last and second-last type for each set, number of people assigned to each set (or difference to cut one dimension).

A total of 7 dimensions, but small overall => Complexity O(n*relevant-const)

N.B.: keep track of **m** in the excitement function!!!

Week 4: First Steps With BGL

- Algorithmic description

Given an undirected weighted graph, compute the sum of the weights of the minimum spanning tree and the longest shortest path from node 0.

- Solution

Simply use Kruskal's and Dijkstra's algorithms provided by the boost library.

Week 4: Important Bridges

- Algorithmic description

Given an undirected graph, find all bridges (or cut-edges) in sorted order.

- Solution

A biconnected component of a graph is one of its maximal subgraphs such that the removal of one of its vertices does not disconnect that subgraph. It is possible to observe that the only edge of a biconnected component of two vertices is a **bridge**.

Week 4: Ant Challenge

- Algorithmic description

We are given at most 10 weights for each edge of a graph. Find the shortest path from a to b such that the entire path lies on the minimum spanning trees each computed with one set of weights.

- Solution

Since we have at most 10 types of weights, it is possible to iteratively compute the MST for each type.

The next step is to merge all these MSTs in a new graph (we shouldn’t worry about overlapping edges, since Dijkstra works on multigraphs too).

Finally, call Dijkstra’s algorithm.

Week 4: Buddy Selection

- Algorithmic description

Given a list of strings per each node, find a maximum pair matching so the matched pair has more than f common strings.

- Solution

Add an edge between two nodes if and only if they have more than f strings in common.

It is convenient to count the number of common strings between two nodes with a merge-like approach (O(n) intersection between two arrays)).

A solution is found if the matching is maximum (perfect), namely its size is exactly n/2, with n being the number of nodes.

PotW 5: Motorcycles

- Algorithmic description

Given a set of lines starting from the y-axis, tell which of them will make it to infinity, namely which lines don’t have intersections or, if any, are the earliest to reach them (in terms of Euclidean distance), with respect to the others.

- Solution

Greedy approach: sort the lines with respect to their y starting point.

Store for every line its starting point, its slope and its original index (the latter because of the sorting).

Do an iteration from bottom to top and one from top to bottom, marking as *false* the lines that have a bigger slope than the current minimum.

Be careful to compare slopes: from bottom to top, use CGAL::abs(curr) < CGAL::abs(best) || CGAL::abs(curr) == CGAL::abs(best) && curr > 0 (EDGE CASE) to update best and curr < best to set current line as false.

From top to bottom, just use CGAL::abs(curr) <= CGAL::abs(best) to update best and curr > best to set current line as false.

I used CGAL::Gmpq to achieve full points, not a kernel!

Note: 75 points solution is pretty immediate.

Week 5: Boats

- Algorithmic description

Given a set of pairs (segment, point), output the maximum amount of pairs so that they do not overlap in a 1-D context, that is: every point must be inside the respective segment and two adjacent segments don’t overlap.

- Solution

Greedy approach: sort the positions of the points by increasing order and try to place every segment so that its paired point is at the rightmost end.

Use start and last_start pointers to track the current and last rightmost endpoints. If the next point is not covered, place the segment and update count, otherwise, consider removing last segment and placing the new one (without updating the total), if the new starting position is smaller.

- Take home

Greedy solutions may go back to the previous choice.

Week 5: Moving Books

- Algorithmic description

Given a set of people and a set of boxes, calculate the minimum amount of time needed to move all boxes.

- Solution

Greedy solution: everone picks the largest box they can carry.

Multiset on boxes and upper_bound().

Iterate over the boxes until they run out, assigning each box to the person who has minimum, sufficient capacity.

Trick: break the inner loop if I find a person that is not capable of carrying the current weight.

Week 5: Severus Snape

- Algorithmic description

We are given two sets of elements:

➢ The elements of one set are given in pairs (p_i, h_i)

➢ The elements of the other one are described by w_i.

The elements of the first set each reduce by a the amount of w, while the elements of the second one reduce by b the amount of p.

Calculate the minimum number of elements among those two sets such that $p \geq P, \;\; h \geq H, \;\; w \geq W$

- Solution

An intuitive strategy would be to greedily calculate the number of B elements by sorting the B set and to calculate the minimum number of A elements with a knapsack-like DP (capacity1 ≥ H && capacity2 ≥ P).

However, $P \leq 2^{31}$, which is too large to keep track in a DP dimension. On the other hand, $\mathrm{H} \leq 2^{10}$, which is a reasonable dimension for a DP.

The DP relation must hence be changed to maximizing p given a fixed number of elements and capacity ≥ H (see **San Francisco**). This could be done by iterating over this DP while linearly increasing the number of elements that must be picked. The minimum number of elements is found as soon as the result of the DP is ≥ P.

Remark: the h dimension is **capped** at 0 => if(h <=0) h = 0, so to have at most 1024 values. Values where h < 0 are not important.

To deal with decreasing values adopt the following strategy:

calculate w for the first k elements in B, check if A's DP can return a value ≥ P+(long)numB*b. If $w + \mathrm{numA} * \mathrm{a}$ is still greater than W, this solution is suitable. Because w decreases when k does, it is possible to apply a binary search to find the minimum feasible total amount of elements.

Beware to first start with all the elements in B and THEN to bisect in the search.

Week 5: Asterix The Gaul

- Algorithmic description

Given a list of pair elements (d, t), tell if it’s possible to achieve d >= D && t < T.

Addition: bonus vector where the i-th element adds s[i] to d every time.

- Solution

Given the input bounds, the only approach is brute-force.

First try without adding bonus, otherwise binary search to add bonuses (they are given in sorted order).

This is a Split & List problem where the condition sum == k becomes d >= D && t < T.

To reconstruct the result, we must sort L2 with respect to t and find the best d such that t < T (create a new vector with best d within given t of L2).

The new vector is created so that we have some sort of double ordering (without it, I cannot make assumptions on d since I sorted on t, it would be a linear search).

Iterate over L1 until I find t1 + L2[i].t < T && d1 + bestd[i] >= D

PotW 6: Planet Express

- Algorithmic description

Given a graph with k starting points and one destination, calculate the minimum shortest path from one starting points to the fixed destination.

The graph has «teleport» edges that connect special vertices that are in the same connected components. The weight of those edges depend on the number of those special vertices in the same scc.

- Solution

Invert the graph so to calculate Dijkstra once, starting from the destination. Add a dummy vertex per every component and connect it to all the special vertices with two directed edges: weight 0 and weight calculated as above.

- Take home

It is usually useful to modify the given structure of the graph, by adding nodes, inverting edges or even duplicating the whole graph.

Week 6: What Is The Maximum?

- Algorithmic description

LP toy problem where we are directly given all of the variables and the constraints

- Solution

Simply apply CGAL LP methods.

Week 6: Diet

- Algorithmic description

LP problem:

Constraints: $min_i \leq \sum_j C_{j,i} x_j \leq \ max_i$ (~ 80 constraints)

Objective function: $\min \left(\sum_j p_j x_j \right)$ (~100 variables)

- Solution

The most difficult part here is to translate the story into the mathematical formulation written above, carefully choosing which are the variables of the linear program.

Week 6: Inball

- Algorithmic description

Given a set of d-dimensional half-planes, find the maximum radius of a ball that fits inside of the closed region they can possibly generate.

- Solution

The most difficult part here is to use **geometric reasoning** in order to properly define linear constraints.

In order for the ball to fit, the most sensible points are the ones nearest to every half-plane. It is possible to observe that these points have the same direction as the normal vector of that plane, with respect to the center of the ball. We are therefore focused in defining constraints for these extreme points.

The problem statements gives a hint stating that the norm is an **integer**, so I infer it could appear as a coefficient inside the constraints.

Given the center $\vec{x}$, an extreme point with respect to an half-plane is $\vec{x} + r\frac{\vec{a}}{||a||}$, with $\vec{a}$ being the normal vector of that plane.

The trick is therefore to add a new variable that represents an r-distant extreme point.

- Take home

Watch for lower and/or upper **bounds** on single variables.

Carefully look at the problem statement (there might be hints).

Add variables to represent distances.

Week 6: Lannister

- Algorithmic description

Given a 2D plane with two oblique orthogonal lines and a set of points, solve the following linear program:

1. Divide the points into two sets with a line.
2. Maximize the longest vertical segment from a point to the orthogonal line.
3. Make sure that the sum of the horizontal segments does not exceed a LONG value s.

- Solution

Given a line ax+by+c=0, a generic orthogonal line is described by: bx-ay+c2=0.

Since a line of the two is not horizontal, a = 1 without loss of generality.

Thus, its orthogonal line is: bx – y + c2 = 0.

Variable d to compute constraints on distances from the points, such that $\left| y_p - y_l \right| \leq d.$

Input type is LONG !!!

- Take home

Try to make assumptions so to always obtain linear constraints and objective functions (If a != 1, constraints for the orthogonal line would not be linear!).

Output may depend on multiple calls to solve_linear_program (maybe after adding more constraints).

PotW 7: Octopussy

- Algorithmic description

Given a **complete** binary tree where each node has a property, tell if it's possible to visit the tree from the leaves up to the root such that the property of the visited node is always greater than the number of currently visited nodes.

- Solution

Tree, a greedy approach works: sort the properties in increasing order and visit the respective subtrees. If the overall time spent to visit the subtrees is larger than the property of that subtree's root at some point, the answer is «no», otherwise it's «yes».

Use flags to mark already visited subtrees (do not visit them again!!!).

Use a recursive function to calculate the visiting time of each subtree.

- Take home

Greedy solutions might make use of some precomputation, which is usually achieved through recursion in trees.

Week 7: Shopping Trip

- Algorithmic description

Given an undirected graph with edges of capacity 1 (streets that may be traversed only once), tell if it’s possible to reach all the demand nodes => circulation problem

- Solution

Add an edge from every demand node to the sink, calculate max-flow.

If it is equal to the total number of demand nodes, output «yes».

- Take home

Max-Flow is the most suitable approach to calculate edge-disjoint paths.

Week 7: Knights

- Algorithmic description

Given an undirected graph with edges of capacity 1 (streets that may be traversed only once), and vertices of capacity c, calculate the maximum flow

- Solution

To model vertex capacities, split each vertex into two nodes: one will only have incoming edges, whereas the other one will have only outgoing edges. Add an edge of capacity c between those two nodes.

Add an edge from the source to every starting position and from escape nodes to the sink, calculate max-flow.

Week 7: Coin Tossing Tournament

- Algorithmic description

Given a set of matches and a final leaderboard, tell if it is possible to obtain the latter given the former.

- Solution

Bipartite graph: on one side matches, on the other side the final leaderboard.

The leaderboard is obtainable if the points flowing from the matches fulfill the amount of points for each node in the leaderboard (i.e. flow == sum_points) AND all the matches are assigned a winner (i.e. flow == number_of_matches, edge case).

The max-flow approach is used so to maximize matching between those sets.

Week 7: London

- Algorithmic description

Given a string and a set of pair characters, tell if it's possible to construct that string starting from the set.

- Solution

MaxFlow is a suitable approach because of its intrinsic capability of maximizing a matching between two sets.

Trick: instead of sequentially storing every appearing character in the graph, it is enough to store two sets of letters of the alphabet, for a total of 26*2 nodes. Their multiplicity is then accounted in separate vectors, which will represent the supply/demand of flow from/to that letter.

Scan through the input and use count vectors to store the number of repeating letters.

To account for paired letters, use a data structure to map the second letter to the first one and still draw an edge starting from the latter.

If the flow is equal to the length of the given string, the problem is solved.

N.B.: the add_edge function cannot always be considered irrelevant for the runtime. I stored the matchings in a matrix and added the respective edges **after** reading all the input. If I were to do this on-the-fly, the program would **timeout**.

- Take home

Be careful to how many times add_edge function is invoked.

Always try to minimize the number of needed nodes.

PotW 8: Suez

- Algorithmic description

Given a set of rectangles (h*w) and their center points, add scaled(ah*aw) rectangles to available free center points so they don’t overlap. Maximize their sum of perimeters.

- Solution

Given the input size and the problem description, LP is the best way to solve this problem.

At a first glance, contraints have a logic-OR form (on x || on y). Linear constraints are inferred by looking at **which** of the two conditions in the OR is most restrictive, taking into account the h and w dimensions (**scaled comparison**).

Trick: After setting up constraints for new rectangles, we can add only **one** more constraint **each** for the nearest, already present, rectangle. It is useless indeed to add more constraints on old rectangles.

To infer which old rectangle we should put in that constraint (**if present**), we keep **for every new rectangle** the minimum **scaled** distance (either x or y) between the new rectangle and the old ones and store the index (0,…,m-1) for which this parameter is minimized.

- Take home

Be careful with the size of the input (int or long).

Linear constraints must be derived, otherwise LP is not applicable.

Always try to reduce the number of constraints to really useful ones.

Week 8: Bistro

- Algorithmic description

Given a set of current points, find the squared distance of the nearest one to every new point given.

- Solution

Delaunay Triangulation toy problem.

Use nearest_vertex() query (O(logn)).

- Take home

CGAL::squared_distance outputs a *double* in EPIC kernel, with exponential format.

Week 8: Germs

- Algorithmic description

Given a set of growing disks where $\rho(t) = t^2 + \frac{1}{2}$, and a boundary defined by $l \leq x \leq r$ and $b \leq y \leq t$, output three times, namely the time where the first disk touches another one or a boundary, the median time and the largest time.

- Solution

Delaunay Triangulation contains the nearest neighbour graph.

For each vertex of the DT, first calculate the distance to the boundary and then compare it with distances with neighbouring vertices.

Sort an array times and output times[0], times[n/2] and times[n-1].

Use EPEC kernel with sqrt to calculate the sqrt (optional, not needed).

Week 8: H1N1

- Algorithmic description

Given a set of disks of radius $\sqrt{d}$ and a set of points, tell if it's possible for each point to go infinitely far away from the disks without touching them.

- Solution

Delaunay Triangulation on the center of the disks.

Use a Dijkstra-like algorithm starting from the infinite faces of the DT.

Use f->info() to store best escape distance for every face.

Update if min(curr->info(), squared_distance(v1,v2)) > neigh->info().

Priority queue is with maximum on top (because I am relaxing the maximum).

For every point, first check if its starting position is inside one of the disks and then locate() the face and calculate the minimum escape distance (should be $> (2\sqrt{d})^2 = 4d$).

- Take home

Be careful to define distances.

Face with info has O(1) access.

Week 8: Light The Stage

- Algorithmic description

We are given a set of disks of radius r and an additional set of disks of radius h where each disk spawns one at a time. Output the indices of the first disks which never intersect with the second set. If this is not possible, output the indices of the last disks to intersect with the second set, with respect to the spawning pattern.

- Solution

Delaunay Triangulation on the SECOND set. For each disk of the first set, check if the nearest of the second one does not intersect. If this happens, the query is solved. Otherwise, iterate over the disks of the second set in increasing order and store the minimum index of the disk that intersects with it (namely, the round).

Output indices of the first set that have the maximum round.

- Take home

Check out which set is better for DT.

Why does EPIC kernel work?

Why does O(n^2) solution work?

PotW 9: Kingdom Defence

- Algorithmic description

Circulation problem with minimum capacity constraint, just like the one on the tutorial slides. Each node has a current amount and a needed one.

- Solution

To model a minimum edge constraint, modify the capacity of that edge to Cmax – Cmin, increase the demand of the source node of that edge by Cmin while decreasing the demand of the target node of that edge by the same amount Cmin.

Circulation problem: calculate for each node the difference between the needed amount and the current one. If it is positive, we have a demand node, which is to be linked to a common sink. If it is strictly negative, we have a supply node, which is to be linked to a common source.

The circulation problem is solved if and only if the maximum flow is equal to the sum of the demands (positive differences).

Week 9: Algocoön Group

- Algorithmic description

Minimum cut problem without knowing the two endpoints

- Solution

Based on the MinCut-MaxFlow theorem (see slides), the value of a minimum cut between two vertices s and t is equal to the maximum flow flowing from s to t.

The problem here is we don’t know what are the best endpoints to calculate the flow on.

Instead of naively trying all of the combinations (s,t) (O(n^2)), try to think of one node: it is either part of my partition or not.

Hence, if we pick this node, we could choose it as a starting point for a flow without loss of generality and iterating over all of the remaing nodes chosen as targets.

If we don’t pick this node, we could do the same thing iterating over source nodes and leaving the target as fixed (O(n)).

The minimum cut over all pairs (s,t) is the minimum MaxFlow we came across.

- Take home

Partial test sets sometimes give hints!

Week 9: Real Estate Market

- Algorithmic description

Maximum weighted matching with capacity constraints on one side.

- Solution

Given the size of the input, MinCost-MaxFlow is the best choice.

To account for capacity constraints, add filtering node on that side before the sink.

Cost has to be maximized: cycle_canceling() is slow, use successive_shortest_paths() offsetting negative weights.

The offset can be easily derived by looking at the maximum cost given at the input (100).

Final cost is corrected subtracting offset*max_flow to cost and changing its sign.

- Take home

Look at the starting indices of the input (this case one set starts from 1, not 0).

Each source-sink path must go through the same number of offsetted edges!

Week 9: Canteen

- Algorithmic description

Time graph with edges between every two nodes (to store unused flow for each node).

Nodes can use discarded flow by preceding nodes.

Apply MinCost-MaxFlow

- Solution

The problem description literally builds the graph, where we have both negative and positive costs.

Since cost has to be maximized, offset negative costs by the same total amount for s-t paths (here is 20). Apply successive_shortest_path()

- Take home

Add edges between nodes if available flow is discarded and reused by other nodes.

Each source-sink path must go through the same number of offsetted edges!

Week 9: Placing Knights

- Algorithmic description

Given a chessboard, find the maximum number of knights that do not threaten each other, that is a maximum independent set in a bipartite graph (two knights threaten each other only if they are not on the same color).

Addition: some of the cells cannot be occupied (holes).

- Solution

Maximum independent set on a bipartite graph => MaxFlow = MinVC = N – MaxIS.

Holes are part of the MaxIS, must subtract the total.

Graph is **directed** with edges starting from one color to another (there are no undirected edges).

- Take home

Geometric reasoning and examples are very useful to model the problem.

PotW 10: GoldenEye

- Algorithmic description

We are given a set of disks whose centers are given, but the radius is variable. There is a collection of source and target points. These points are connected if they lie in the same component of the union of disks.

Output three values:

1. Which pairs of points can be connected using radius p;
2. The smallest radius a so that all points are connected;
3. The smallest radius b so that only the points in (1.) can be connected.

- Solution

Delaunay Triangulation of the disks.

Extract a vector of all edges of the DT and sort them by squared length.

Use 3 union-find structures: one (ds) with all edges with length <= p and two empty at the moment (dsa, dsb).

For every pair of points:

1. Find closest disk centers for both and ensure they are within a distance of 4*p from the given pair (ds, dt).
2. If they are in the same component, output «y» and update value b = max{0, b, ds, dt} with minimum squared distance until the two points are connected in dsb, otherwise output «n».
3. Do the same for a with dsa, but this time with all points (and not only the ones that answer «y» to the first query).

- Take home

DT significantly reduces the amount of useless edges, but keeps all the useful ones (in this case the EMST).

Week 10: Asterix In Switzerland

- Algorithmic description

Given a directed d-weighted graph, where each vertex has a property b_i, tell if it's possible to find a subset X of vertices that is free-standing, namely:

$$\sum_{i \in X} b_i \; > \; \sum_{i \in X,\, j \notin X} d_{i,j}$$

- Solution

The main idea is to use flow and think of what the constraint above implies on it: if some vertices are free-standing, they will form a bottleneck in flow, since flow leaving this subset will be less than the sum of all the properties on those vertices (b). By connecting positive properties to the source and negative ones to a sink, the problem is solved, i.e. there is a free-standing set of vertices if and only if sum_positive_b > flow.

- Take home

Consider using flows for problems with yes/no answer.

Week 10: World Cup

- Algorithmic description

We are given two sets of points. One set has a certain supply and the other has a demand. Let's call the unit of supply l. Maximize the total revenue among all exchanges between supply and demand nodes, given the constraints below:

1. Supply nodes have a total upper bound of s and a given percentage a.
2. Demand nodes demand **exactly** d and cannot consume more than u in total when considering the percentage $a\%\ l$.
3. Each node can lie on some disks of variable radius. The disks do not intersect. The revenue for each pair of points decreases by $t/100$, where t is the number of circles the segment that connects the two points intersects.

- Solution

We have many constraints (flow not applicable). They are all linear and the input size is small => LP.

In order to have only integer coefficients, multuply the percentage constraint by 100.

To deal with the disks, initialize a matrix for each of the two sets of points to all 0s. Iterate over all disks. If a point is included in a disk (i.e., squared_distance <= r*r), then set the cell to 1.

When iterating over all pairs of points, check with a XOR operation how many disks we have to cross (sum all the 1s after the XOR).

Maximize integer objective function => multiply by 100 and change its sign.

- Take home

When the task description states to round the result to an integer, it suggests using LP.

Naive solution, but still works.

Week 10: Evolution

- Algorithmic description

Given a tree where vertex values are decreasing from root to leaves, find the ancestor vertex closest to root from a starting vertex that has weight at most b.

- Solution

Explicitly storing all paths is not allowed in this problem, since O(n^2) would be too large.

A better solution is to store all the queries first, grouping them per starting vertex and do a **single** DFS visit of the tree. Use a tmp vector to store the indexes visited and then do a binary search on the fly over tmp per every query that has the same starting vertex.

We need to find the root by searching for the index of the largest element.

Remember to backtrack the vector tmp.

Rearrange the results at the end.

- Take home

It is sometimes useful not to follow the natural flow of the problem (in this case we store all the queries first).

Week 10: Asterix And The Chariot Race

- Algorithmic description

Given a rooted tree and a cost for each vertex, find the minimum cost of a subset of vertices such that every vertex in the tree has an edge to at least one of them.

- Solution

Although it may seem like a vertex cover, surprisingly it's not.

DP state: bool covered, bool compulsory, int start.

Precomputations inside DP:

cov += f(target, covered=true, compulsory=false); cov += costs[start]

If(compulsory) return cov; //must be covered, return

not_cov += f(target, covered=false, compulsory=false);

If(covered) return min(cov, not_cov); //father covered, choose minimum

comp = cov;

//father not covered, choose minimum between covering this node or covering one of the children

comp = min(comp, not_cov – f(target, covered=false, compulsory=false) + f(target, covered=false, compulsory=true);

- Take home

EXTREMELY PAINFUL

PotW 11: Phantom Menace

- Algorithmic description

Given a directed graph, a set of starting nodes and a set of end nodes, output the minimum number of nodes that cut every path between the two sets.

- Solution

The problem statement clearly tells to find a minimum vertex cut.

Since we know how to find an edge cut using flow, simply split each vertex into two nodes (in and out) and add an edge in between of capacity 1.

The MaxFlow corresponds to the minimum vertex cut.

Week 11: Return Of The Jedi

- Algorithmic description

Calculate the second-best minimum spanning tree.

- Solution

First calculate the minimum spanning tree.

Discard one of its edges at a time while building a second best minimum spanning tree and find the one which has minimum overall cost.

- Take home

We can be asked to rewrite basic graph algorithms that we have in the library.

Week 11: Idefix

- Algorithmic description

We are given a set of disks of radius r (initally) and a set of points.

Answer two queries:

1. The maximum number of points one can inspect on a path through some disks, such that the trajectory must be contained in the union of those disks. Start and end points are freely chosen.
2. Given a number k, the smallest disk radius such that one can inspect k points, under the same constraints as above.

- Solution

Use a union-find structure for the each of the queries:

For the first query, add to the union find only edges between disks that have length at most 2*r and store in an array how many points are reachable within a set (i.e. connected component).

For the second query, add to the union find ALL edges (including those between points and disks scaled by 4) and mark them with a bool to distinguish the edge type, then sort them. For each edge:

If it connects a point and a disk, just increase by 1 the number of points reachable by the respective set of disk which the latter belongs to (using a new array, as in the first query).

If it connects two disks, do a union and add the count of points to the newly formed set.

As soon as one set has more than k points, break the loop.

Week 11: Asterix And The Roman Legions

- Algorithmic description

We are given a starting point (x_s, y_s) and several straight lines that surround it. Choose another point reachable from the start such that distance from the lines is maximized

Addition: for every line we are given a factor v (which is supposed to be the velocity), calculate the time instead.

- Solution

Similar to **Inball**, however, we need to look at the orientation of the lines, based on the starting point:

if $ax_s + by_s + c \leq 0$, then the configuration is exactly the one of Inball: define variables x, y, r such that $ax + by + \|(a,b)\|r \leq -c$.

Otherwise, the constraint is: $ax + by - \|(a,b)\|r \geq -c$.

Remember to put a lower bound on r.

To calculate the time instead, it is sufficient to multiply the norm by v.

- Take home

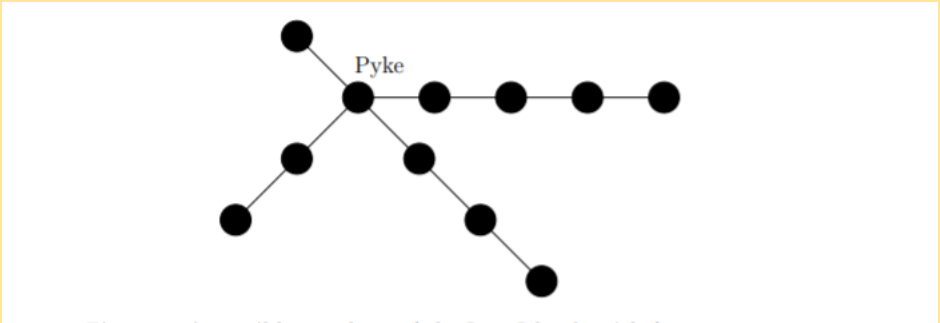
Rounding the output suggests using LP.

Partial test sets give hints! (see the one where $ax_s + by_s + c < 0$).

Problem description gives the hint of using the norm, by telling us we can assume it is always an integer.

Be careful with **int** and **long** !!!

Week 11: The Iron Islands



- Algorithmic description

We are given a tree with only branches of degree 2 (see picture). Each node has a property. We are asked to find a consecutive sequence of nodes of maximum length such that the sum of the properties is exactly equal to k.

- Solution

This sounds like a sliding window problem. However, this technique allows to only inspect sequences on the same branch.

To match sequences on different branches, precomputation is needed.

Calculate for each branch the current sum at every index and store the longest sequence in a map<sum, number_of_nodes>.

If I find a complementary sum in the map (such that the current sum + complementary sum = k – prop[node_0]) then update the maxcount variable: maxcount = max(maxcount, j + complem[diff] + 1).

Do not include in the precomputation the root of the tree (node 0, start from 1 in the precomputation).

At the end of every iteration, update the map if there is a larger number of nodes for the same sum.

PotW 12: San Francisco

- Algorithmic description

Given a weighted directed graph with a fixed starting point, tell if a k-long path starting from there has a total weight greater than a *long* threshold x.

Points that do not have out edges are linked to the starting point with a 0-capacity edge. Flowing through this edge does not count as a move.

- Solution

The most feasible approach is to use DP, since the problem has an optimal substructure: telling if I can reach x with k moves is equivalent to finding the best out edge such that I can reach x-w[edge] with k-1 moves starting from the target node.

Since $x \leq 10^{14}$, it is impossible to build the intuitive DP where I have to minimize the number of moves while keeping track of this big intermediate value in my state. Hence, I change the DP relation to calculating the maximum weight I could have with a FIXED number of moves.

By linearly increasing this fixed number outside the DP relation I keep track of the minimum amount needed and break from the loop.

- Take home

Before deriving a DP relation, look at the possible dimensions of its state!

Week 12: On Her Majesty's Secret Service

- Algorithmic description

Given a set of starting points and a set of destinations in a graph, output the longest shortest path such that every starting point reaches a destination.

Additions: every destination has an additional weight d and up to $1 \leq c \leq 2$ points can match with the same destination, but always one at a time (i.e. the second point must wait $2d$ before reaching it).

- Solution

There is no direct algorithm for this problem.

First thing to do is to reduce the problem to an assignment problem => modify the graph and keep only shortest path edges from every starting point to every destination.

Then do a binary search (guess) and only add paths that have weight $\leq mid - d$. For the case where $c = 2$, create additional destinations and add paths to them that have weight $\leq mid - 2d$.

Compute a maximum matching. If its size is exactly equal to the number of starting points, try to decrease mid.

- Take home

Modifying the graph is usually a good choice to better understand the problem.

Nodes not reachable by Dijkstra have path weight of numeric_limits<int>::max() => watch out for overflows!

Partial points give hints! (the second test set has a binary answer, which is suggesting that guessing with binary search is a good approach.

Week 12: Hong Kong

- Algorithmic description

Problem setting very similar to H1N1, however escape points are also inside the convex hull. One can move by keeping a distance at least s+r from the points and can escape if that distance is at least 2*s+2*r.

- Solution

Same solution as H1N1, but push all Delaunay faces in the priority queue with the farthest distance from their vertices whithin each face (i.e., distance from the associated Voronoi vertex to one of them, it is the circumcenter).

Pay attention to the field type => EXACT constructions needed, K::FT.

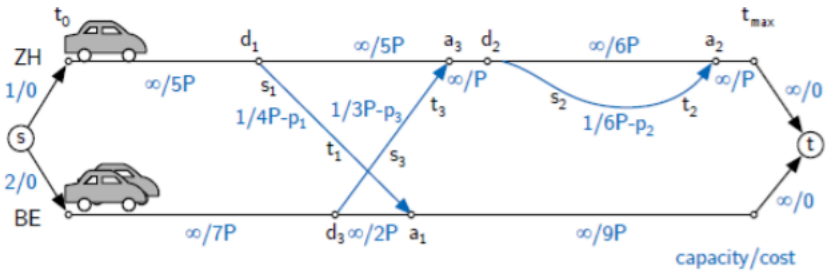
For every point to be queried, check if it can escape immediately or if it won’t ever escape because it’s far too near.

- Take home

Drawing geometric problems might be useful to define constraints.

Pay attention to the input size!

Week 12: Car Sharing



- Algorithmic description

Graph of movements over time. Each movement has a cost, maximize the cost over all movements with a planning schedule that is feasible.

- Solution

MinCost-MaxFlow, see graph.

Use successive_shortest_paths where offset is ruled by time (arrival-departure)*max_cost**. Every source-sink path should have the SAME offet.**

Represent only useful times in the graph => first store the relationships and then create the graph. Add a 0 and MAXTIME node per every location.

Use a **set** to sort time slots for every location.

Play a bit with offsets to identify nodes' index.

Add edges between two moments of the same location to store unused flow (see **Canteen**).

Week 12: Bonus Level

- Algorithmic description

Given a matrix where one could move down or to the right from the left-upper point and up and left from the right-downmost point, collect the maximum value starting from the left-upper point, reaching the most right-down point and back.

- Solution

The main idea here is to refine the DP formulation.

First, it is possible to keep track of both paths at the same time with 4 indices: i1, i2, j1, j2

Pick the i1, j1 cell (set it to 0) and then recur over all the 4 combinations of indices.

Remember to backtrack the i1,j1 cell.

Add at the end the i2, j2 cell.

Trick: next move is always to the next diagonal => space (i1,i2,j1,j2) could be reduced to (i1,i2,diag) (n*n*(2*n-1)).

j1 = diag – i1, j2 = diag – i2

- Take home

Always try to reduce the dimension of the state to the strictly necessary.

Naive DP may not always be sufficient.

PotW 13: Clues

- Algorithmic description

We are given a set of fixed points in the plane and a set of pairs of points. Tell if those pairs are reachable within a given radius r (i.e., either they are closer than r or they can form a chain of points involving the fixed ones such that the maximum distance between every pair of points is less than r.

Addition: the network of fixed points itself can be non-acceptable. This happens whether, connecting points that are at distance less than r, we cannot form a 2-coloring. In this case, output «n» for every query.

- Solution

Without the addition, the problem is very similar to **GoldenEye**: DT on the fixed points, union-find to keep track of <= r connections and see if a given pair of points belongs to the same component.

To account for the addition, do a DFS on the graph constructed by fixed points connected by edges <= r. Mark the visited points as either black or red (visited[circ->info()] = 1 – visited[u->info()], with WHITE = -1, BLACK = 0, RED = 1).

Generate two triangulations, one for only black points and the other for only red ones. Check if all the edges in those triangulations have length at least greater than r. If this does not happen, output all «n» for the testcase.

- Take home

Be careful to LONG input!

Week 13: Hagrid

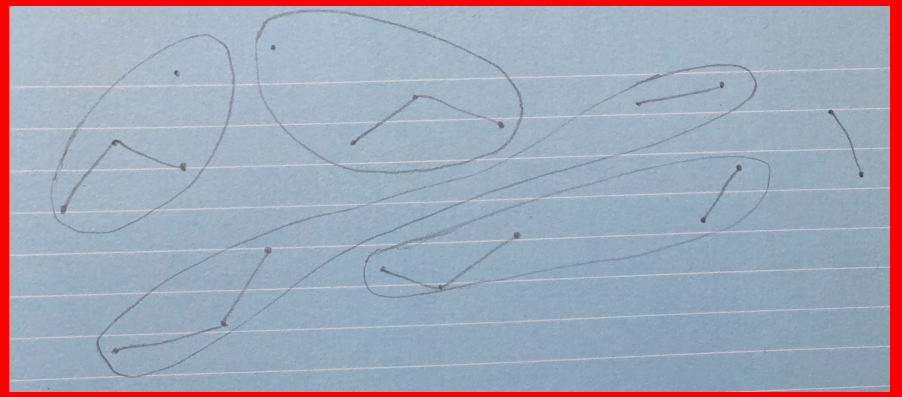
- Algorithmic description

Given a tree, find the best visiting order so to maximize the cost of each vertex. The cost diminishes as visiting time increases.

- Solution

Recursively calculate the time t needed to visit each subtree and its number of nodes n and always prefer the lowest t/n in the DFS.

Week 13: Hand



- Algorithmic description

Geometric assignment problem: given a set of points, divide them into groups of f such that every group as at least k points. Whenever two points are assigned to different groups, they are at squared distance at least s.

Answer two queries:

1. Given f_0, what is the largest s such that there exists an assigment compatible with the constraints above.
2. Given s_0, what is the largest f such that there exists an assignment compatible with the constraints above.

- Solution

Build a Kruskal-like algorithm, with union-find and edge sorting.

Initialize a vector that counts the number of components of size 1, 2, 3, and 4 or larger and a vector which keeps track of the size of every component.

For every edge:

If we can find f0 groups (with an algorithm specified below), then s = curr_dist.

Union sets, update components and sizes.

If curr_dist < s0, then f = curr_groups.

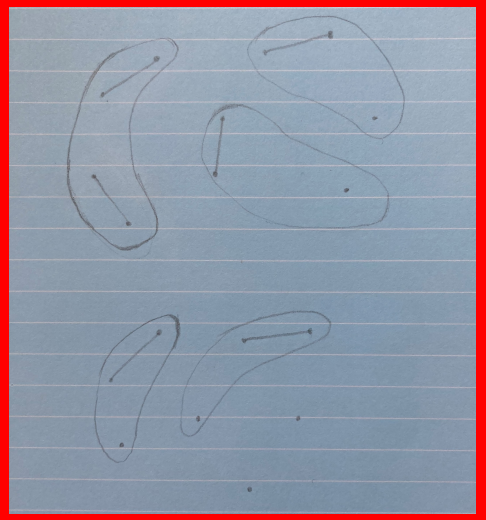
Break the loop when there is only one component.

Group finding algorithm: try to match components of complementary size to build k groups (drawing points. might help)

- Take home

Solution with k = 1 is very straight forward => consider it!

In the group algorithm, divide components in the **return** clause!



Week 13: Ludo Bagman

- Algorithmic description

Assignment problem in a bipartite graph with constraint on minimum participation of all nodes. Tell if it's possible to satisfy this constraint and output the minimum cost for an assignment of cardinality p.

Addition: two types of matching, minimum constraint only on one kind.

- Solution

Add a **dummy** node on each side and split the flow in two: part of it must flow through each of the E vertices (exactly L), while the remaining could flow anywhere (**no more than** P-E*L supply for dummy node).

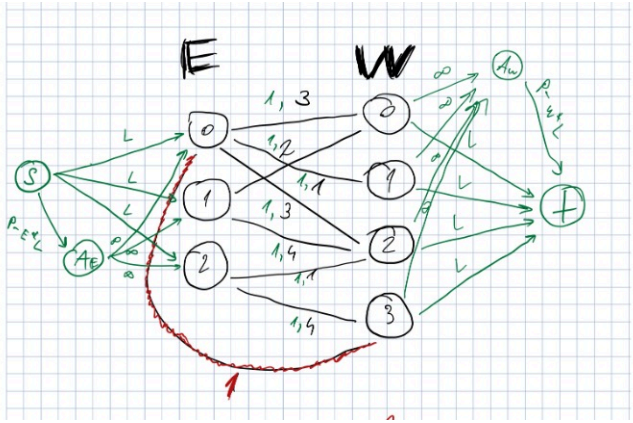
Same for the other side (demand of exactly L for each of the W vertices and P-W*L for another dummy node).

The other type of matching does not require constraints, so we could **duplicate** the graph and use the dummy node's remaining flow to generate more matches in there.

- Take home

The key here is to force some flow to go through preferred paths.

Adding nodes and modifying the graph might be very beneficial.



Week 13: Punch

- Algorithmic description

Given an array, minimize the cost of a collection of capacity at least k.

As a tie-breaker, maximize the number of distinct elements.

- Solution

Unbounded knapsack DP with pair<int,int> as a result. Use a custom compare function to maximize distinct number when cost is the same.

Use a bool flag to keep track of when DP switches to a new element.

- Take home

Start from the smallest index in the memo table to optimize memory and time access.

PotW 14: India

- Algorithmic description

Calculate the maximal admissible flow such that its minimum cost is not greater than a given threshold.

- Solution

Kind of the inverse problem of MinCost-MaxFlow. We don’t have an algorithm that directly solves this problem, we need a maximum flow to start with.

Best approach is guessing the result => binary search the maximal admissible flow between given lower and upper bounds. Add a supply node and try to adjust its value based on the last result of MinCost-MaxFlow for a given supply (with c_map[edge]). If mincost is not greater than the threshold, we are allowed to increase supply.

Binary search is applicable because increasing the amount of flow results in increasing the total cost.

- Take home

If it appears to be no algorithm for a given problem, help yourself with binary search as an aid to guess the solution (if applicable).