JAG 2017+ Contest

March 22, 2018

Mikhail Tikhomirov (Moscow IPT),
Artem Vasilyev (ITMO U)

Hello India x Russia Programming Bootcamp 2018

A. Window

Given N windows are shifted either to the left or to the right.

Calculate the total area not covered by any window.

A. Window

Constraints are small enough to just mark all covered position in a boolean array. It takes $O(W)$ time for one window and $O(NW)$ time total.

After that, just calculate the number of `false` entries in that array.

Total time complexity is $O(NW)$.

B. Tournament Chart

There was a single elimination-style tournament. Results of matches are lost, but we know the number of matches each participant won.

Is this information consistent?

B. Tournament Chart

First, you have to parse the tournament structure. The grammar is simple enough to parse it using just one recursive function:

- If the current symbol is a letter, return it.
- Otherwise, skip left brace, parse first participant, skip dash, parse second participant, skip right brace.

B. Tournament Chart

Now, do a DFS on this parse tree and figure out the winner of each match.

Keep the count of remaining wins for each player. When you encounter a match, exactly one of these counts should be non-zero. If this is not the case, there's no solution. When it's true, subtract 1 from the number of wins for the winner of that match and return it.

Time complexity is linear in the size of the input.

C. Prime-Factor Primes

Calculate the number of integers x between L and R such that the number of prime divisors of x (counting multiplicity) is prime, given that $R - L \leqslant 10^6$.

C. Prime-Factor Primes

We are going to find the number of prime factors for all numbers in range $[L, R]$.

C. Prime-Factor Primes

We are going to find the number of prime factors for all numbers in range $[L, R]$.

First, find all prime numbers up to $\sqrt{R}$ using a sieve of Eratosthenes.

C. Prime-Factor Primes

We are going to find the number of prime factors for all numbers in range $[L, R]$.

First, find all prime numbers up to $\sqrt{R}$ using a sieve of Eratosthenes.

Second, for each prime p iterate over all numbers in range $[L, R]$ that are divisible by p and find how many times p divides that number. This step takes $O(\frac{R-L}{p})$ time for one prime p. If we sum over all primes $\leqslant \sqrt{R}$, we get $O((R-L)\log\log R)$ time.

C. Prime-Factor Primes

We are going to find the number of prime factors for all numbers in range $[L, R]$.

First, find all prime numbers up to $\sqrt{R}$ using a sieve of Eratosthenes.

Second, for each prime p iterate over all numbers in range $[L, R]$ that are divisible by p and find how many times p divides that number. This step takes $O(\frac{R-L}{p})$ time for one prime p. If we sum over all primes $\leqslant \sqrt{R}$, we get $O((R-L)\log\log R)$ time.

All numbers that are still greater than 1 will be prime, so we add 1 to the count for those.

D. Revenge of the Broken Door

You want to get from vertex s to vertex t in a weighted undirected graph. One of the roads is broken, and you won't know the broken road until you get to any of its endpoints. Find the shortest time to get from s to t assuming worst-case scenario.

D. Revenge of the Broken Door

For better understanding we can view the process as a game between us and an adversary. Whenever we get to a new vertex, the adversary decides the state of adjacent roads: he can declare at most one road as broken, and all the others as passable (these cannot be declared broken further on). When the adversary breaks a road, all other roads must be passable.

D. Revenge of the Broken Door

For better understanding we can view the process as a game between us and an adversary. Whenever we get to a new vertex, the adversary decides the state of adjacent roads: he can declare at most one road as broken, and all the others as passable (these cannot be declared broken further on). When the adversary breaks a road, all other roads must be passable.

The goal of the adversary is to maximize our time to get from s to t, while we move so that the time is minimal (assuming best play from the adversary).

D. Revenge of the Broken Door

Let's assume that we've reached a vertex v without witnessing any broken roads anywhere before v. If the adversary commits to breaking a road incident to v now, which one should he break to maximize our travelling distance?

D. Revenge of the Broken Door

Let's assume that we've reached a vertex v without witnessing any broken roads anywhere before v. If the adversary commits to breaking a road incident to v now, which one should he break to maximize our travelling distance?

The length of the remaining path is at least $\rho(v, t)$ — the distance between vertices v and t. Further, let e be the first road on a shortest path from v to t. If the adversary does not break the road e, we will be able to get to t in optimal time safely, that is, his move will be useless.

D. Revenge of the Broken Door

Let's assume that we've reached a vertex v without witnessing any broken roads anywhere before v. If the adversary commits to breaking a road incident to v now, which one should he break to maximize our travelling distance?

The length of the remaining path is at least $\rho(v, t)$ — the distance between vertices v and t. Further, let e be the first road on a shortest path from v to t. If the adversary does not break the road e, we will be able to get to t in optimal time safely, that is, his move will be useless.

Let us construct a shortest path tree T rooted at t so that the parent p_v of a vertex $v \neq t$ is the first vertex on a shortest path from v to t. The above reasoning implies that when the adversary breaks a road while we stand at a vertex v, it is going to be the road from v to p_v.

D. Revenge of the Broken Door

Suppose that the above situation happened: we are at a vertex v, and the adversary has broken the road from v to p_v. We want to find the shortest distance from v to t in the absence of this road. Further, we would like to do this for all vertices v.

D. Revenge of the Broken Door

Suppose that the above situation happened: we are at a vertex v, and the adversary has broken the road from v to p_v. We want to find the shortest distance from v to t in the absence of this road. Further, we would like to do this for all vertices v.

Consider any path from v to t; in particular, this path has to escape T_v — the subtree of v in the tree T. Let u be the first vertex in the path outside of T_v, and w be the vertex preceding u in the path. Since T is a shortest path tree, it is optimal to get between v and w via the path in T, the same holds for getting from u to t.

D. Revenge of the Broken Door

Let d_v be the (weighted) distance between v and t in the tree T (that is, the length of the shortest $v \to t$ path), and $around_v$ be the shortest distance from v to t assuming that the edge (v, p_v) is broken.

D. Revenge of the Broken Door

Let d_v be the (weighted) distance between v and t in the tree T (that is, the length of the shortest $v \to t$ path), and $around_v$ be the shortest distance from v to t assuming that the edge (v, p_v) is broken.

From the above reasoning, $around_v$ can be computed by choosing the optimal edge not in T that allows us to get outside T_v. Formally,

$$around_v = \min_{(w,u)} (d_w - d_v) + cost(w, u) + d_u,$$

where the minimum is taken over all edges (w, u) with w in T_v and u not in T_v.

D. Revenge of the Broken Door

How to compute $around_v$ for all vertices v fast enough? Let $e = (u, w)$ be an arbitrary edge, and z be the LCA of u and w in the tree T. The edge e allows to relax $around_v$ with the value $(d_w - d_v) + cost(w, u) + d_u$ for all vertices v on the path from w to p_z in T.

D. Revenge of the Broken Door

How to compute $around_v$ for all vertices v fast enough? Let $e = (u, w)$ be an arbitrary edge, and z be the LCA of u and w in the tree T. The edge e allows to relax $around_v$ with the value $(d_w - d_v) + cost(w, u) + d_u$ for all vertices v on the path from w to p_z in T.

We will compute $around'_v$ as the smallest value of $d_w + cost(w, u) + d_u$ over all edges (w, u) under $w \in T_v$, $u \notin T_v$, then $around_v = around'_v - d_v$. Now, processing every edge (w, u) amounts to the operation "decrease the value of $around'_v$ to … for all larger values on a vertical path in the tree T".

D. Revenge of the Broken Door

How to compute $around_v$ for all vertices v fast enough? Let $e = (u, w)$ be an arbitrary edge, and z be the LCA of u and w in the tree T. The edge e allows to relax $around_v$ with the value $(d_w - d_v) + cost(w, u) + d_u$ for all vertices v on the path from w to p_z in T.

We will compute $around'_v$ as the smallest value of $d_w + cost(w, u) + d_u$ over all edges (w, u) under $w \in T_v$, $u \notin T_v$, then $around_v = around'_v - d_v$. Now, processing every edge (w, u) amounts to the operation "decrease the value of $around'_v$ to ... for all larger values on a vertical path in the tree T".

This can be done quickly either with an HLD structure on T ($O(\log^2 n)$ per edge), or with offline binary lifting technique ($O(\log n)$ per edge).

D. Revenge of the Broken Door

Now that we know *around*$_v$ for all vertices, how to find the answer?

D. Revenge of the Broken Door

Now that we know $around_v$ for all vertices, how to find the answer?

Basically, we are allowed to move freely in the graph, but the adversary can finish our movement at any vertex v and add $around_v$ to the distance.

D. Revenge of the Broken Door

Now that we know $around_v$ for all vertices, how to find the answer?

Basically, we are allowed to move freely in the graph, but the adversary can finish our movement at any vertex v and add $around_v$ to the distance.

We can either binary search on the answer R and check if the adversary can keep us from getting to t in less than R units, or find a shortest path from t to s while forbidding to relax distances below values of $around_v$.

D. Revenge of the Broken Door

Now that we know $around_v$ for all vertices, how to find the answer?

Basically, we are allowed to move freely in the graph, but the adversary can finish our movement at any vertex v and add $around_v$ to the distance.

We can either binary search on the answer R and check if the adversary can keep us from getting to t in less than R units, or find a shortest path from t to s while forbidding to relax distances below values of $around_v$.

Depending on the implementation, the solution can have $O(m \log n)$ or $O(m \log^2 n)$ complexity (the latter one may need additional optimizations to pass).

E. Tree Separator

Remove a path between two distinct vertices in a tree T so that the number of connected components with at least k vertices is maximized.

E. Tree Separator

Root the tree at an arbitrary vertex. Consider T_v — the subtree of T with root v. Let us count s_v — the largest number of connected components of size $\geqslant k$ that can be obtained from T_v by removing a path starting at v (this path is allowed to consist only of v).

E. Tree Separator

Root the tree at an arbitrary vertex. Consider T_v — the subtree of T with root v. Let us count s_v — the largest number of connected components of size $\geqslant k$ that can be obtained from T_v by removing a path starting at v (this path is allowed to consist only of v).

Let w_v be the number of vertices in the subtree of v. Further, let $I_v = 1$ if $w_v \geqslant k$, and $I_v = 0$ otherwise. Finally, let $c_v = \sum_u I_u$ over all children u of v.

E. Tree Separator

Root the tree at an arbitrary vertex. Consider T_v — the subtree of T with root v. Let us count s_v — the largest number of connected components of size $\geqslant k$ that can be obtained from T_v by removing a path starting at v (this path is allowed to consist only of v).

Let w_v be the number of vertices in the subtree of v. Further, let $I_v = 1$ if $w_v \geqslant k$, and $I_v = 0$ otherwise. Finally, let $c_v = \sum_u I_u$ over all children u of v.

Then, $s_v = c_v + \max(0, \max_u(s_u - I_u))$. Informally, we can either let $s_v = c_v$ by removing only the single vertex v, or continue the path to the subtree of u; in the latter case the optimal number of components is given by s_u, but the subtree T_u itself no longer contributes to the answer, so we have to subtract I_u.

E. Tree Separator

Finally, consider all paths in T with v being the closest vertex to the root. In this case, regardless of the path chosen, the component to the top of v gets separated. Let $I'_v = 1$ is $n - w_v \geqslant k$, and $I'_v = 0$ otherwise.

E. Tree Separator

Finally, consider all paths in T with v being the closest vertex to the root. In this case, regardless of the path chosen, the component to the top of v gets separated. Let $l'_v = 1$ is $n - w_v \geqslant k$, and $l'_v = 0$ otherwise.

The optimal number of components in this case is one of the following:

- If the path has v as one of the endpoints, the answer is $l'_v + c_v + \max_u(s_u - l_u)$, where the maximum is taken over all children of v. Note that in this case we are no longer allowed to remove only the vertex v, hence there is no $\max(0, \ldots)$ in the formula.

E. Tree Separator

Finally, consider all paths in T with v being the closest vertex to the root. In this case, regardless of the path chosen, the component to the top of v gets separated. Let $l'_v = 1$ is $n - w_v \geqslant k$, and $l'_v = 0$ otherwise.

The optimal number of components in this case is one of the following:

- If the path has v as one of the endpoints, the answer is $l'_v + c_v + \max_u(s_u - l_u)$, where the maximum is taken over all children of v. Note that in this case we are no longer allowed to remove only the vertex v, hence there is no $\max(0, \ldots)$ in the formula.
- Otherwise, the path has to proceed into two distinct subtrees, hence the answer is $l'_v + c_v + \max_{u \neq w}(s_u - l_u + s_w - l_w)$, with a similar justification.

E. Tree Separator

To compute the second formula, we store two largest values of $s_u - l_u$ among the children of v.

E. Tree Separator

To compute the second formula, we store two largest values of $s_u - l_u$ among the children of v.

Do the above for all vertices $v \in T$ and choose the best answer. The total complexity is $O(n)$.

F. RPG Maker

Given a grid with `@` (starting point) and $*$ (a city) symbols. Grid size is $(4n-1) \times (4m-1)$ and all cities are located at cells (i,j) where both i and j are odd.

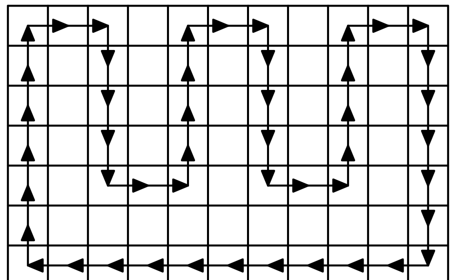
Build a simple path starting at `@`, passing through as many cities as possible, and finishing at any city.

F. RPG Maker

Let's show that it's possible to construct a cyclic path (hamiltonian cycle) that visits all (odd, odd) points. Since the size of the grid is $(4n-1) \times (4m-1)$, the number of (odd, odd) points is $2n \times 2m$.

F. RPG Maker

This picture shows the hamiltonian path for 7×11 grid:



F. RPG Maker

Starting with the hamiltonian path, you can then shift it so it starts at `@` and cut its tail so it ends in a city. This way, you have a simple path that visits all cities and satisfies all other requirements from the statement.

G. Coin Slider

Move maximum number of coins of different radius r_i from their starting position to their target position. Number of coins is $n \leqslant 16$.

G. Coin Slider

$n \leqslant 16$ suggests an $O^*(2^n)$ solution.

G. Coin Slider

$n \leqslant 16$ suggests an $O^*(2^n)$ solution.

DP on subsets: dp[A] is true if you can move coins from set $A \subseteq [1..n]$ to their target positions (in some order).

When we try to add some $i \notin A$ to A, we check if moving coin i to its target interferes with other coins (either at their starting or target position).

G. Coin Slider

How to check if coin j interferes with coin i moving?

Check if distance from center of j-th coin to segment ($start_i$, $target_i$) is less than $r_i + r_j$.

Total time is $O(2^n n^2)$.

H. Separate String

Count the number of ways to cut the given string t into parts so that each part is present in the specified dictionary S.

H. Separate String

Let's start with a simple DP. Let dp_i be the answer for the prefix of t of length i. Then, $dp_0 = 1$, and

$$dp_i = \sum_{1 \leqslant l \leqslant i, t[i-l..i-1] \in S} dp_{i-l}.$$

H. Separate String

Let's start with a simple DP. Let dp_i be the answer for the prefix of t of length i. Then, $dp_0 = 1$, and

$$dp_i = \sum_{1 \leqslant l \leqslant i, t[i-l..i-1] \in S} dp_{i-l}.$$

We can speed up checking that a substring of t is in S either with some text-processing structures (like Aho-Corasick or suffix structures), or with rolling hashes comparison (in which case it is advised to use several independent comparisons with different modulos to avoid collisions).

H. Separate String

Let's start with a simple DP. Let dp_i be the answer for the prefix of t of length i. Then, $dp_0 = 1$, and

$$dp_i = \sum_{1 \leqslant l \leqslant i, t[i-l..i-1] \in S} dp_{i-l}.$$

We can speed up checking that a substring of t is in S either with some text-processing structures (like Aho-Corasick or suffix structures), or with rolling hashes comparison (in which case it is advised to use several independent comparisons with different modulos to avoid collisions).

Still $O(|t|^2)$ (possibly even with some log factors).

H. Separate String

To speed up further, notice that it doesn't make sense to try l in the above formula when there are no strings of length l in S.

H. Separate String

To speed up further, notice that it doesn't make sense to try l in the above formula when there are no strings of length l in S.

Fact

The number of different lengths among elements of S is $O(\sqrt{T})$, where T is the total length of strings in S.

H. Separate String

To speed up further, notice that it doesn't make sense to try l in the above formula when there are no strings of length l in S.

Fact

The number of different lengths among elements of S is $O(\sqrt{T})$, where T is the total length of strings in S.

Indeed, if there are k different lengths among S, then their total length is at least $1 + \ldots + k = \Omega(k^2) \leqslant T$, hence $k \leqslant O(\sqrt{T})$.

H. Separate String

Now, we can rewrite the formula above as

$$dp_i = \sum_{l \in L, t[i-l..i-1] \in S} dp_{i-l},$$

where L is the set of lengths of elements of S.

H. Separate String

Now, we can rewrite the formula above as

$$dp_i = \sum_{l \in L, t[i-l..i-1] \in S} dp_{i-l},$$

where L is the set of lengths of elements of S.

Due to the reasoning above, the complexity of computing this DP is now $O(|t|\sqrt{T})$, which is fine (even with a log factor).

I. Revenge by Endless BFS

We are given a directed graph. A set *found* is initialized to a single vertex $\{s\}$. Each step we perform $found = \cup_{v \in found} N(v)$, where $N(v)$ is the set of vertices reachable from v via an edge. Find the number of steps until $found = V$ — the set of all vertices of the graph, or determine that this never happens.

I. Revenge by Endless BFS

Let us give a rough estimate for the answer. If the set *found* ever repeats values before reaching V, then the process becomes looped, which means that V will never be reached. Otherwise, all values of *found* before V have to be distinct, which means that the answer is at most 2^n (when it exists).

I. Revenge by Endless BFS

Let us give a rough estimate for the answer. If the set *found* ever repeats values before reaching V, then the process becomes looped, which means that V will never be reached. Otherwise, all values of *found* before V have to be distinct, which means that the answer is at most 2^n (when it exists).

Let us represent *found* as a boolean vector u of length n. Then, making a step is equivalent to $u \to Au$, where A is the adjacency matrix of the graph, and boolean matrix multiplication is defined with $(AB)_{ij} = \vee(A_{ik} \wedge B_{kj})$ (here $\vee$ is logical OR, and $\wedge$ is logical AND).

I. Revenge by Endless BFS

Let us give a rough estimate for the answer. If the set *found* ever repeats values before reaching V, then the process becomes looped, which means that V will never be reached. Otherwise, all values of *found* before V have to be distinct, which means that the answer is at most 2^n (when it exists).

Let us represent *found* as a boolean vector u of length n. Then, making a step is equivalent to $u \to Au$, where A is the adjacency matrix of the graph, and boolean matrix multiplication is defined with $(AB)_{ij} = \vee(A_{ik} \wedge B_{kj})$ (here $\vee$ is logical OR, and $\wedge$ is logical AND).

Let u_i represent the state of u after i steps: u_0 has a single 1 at position s, and $u_i = Au_{i-1} = A^i u_0$. The problem boils down to finding the smallest number k such that $A^k u_0 = \mathbf{1}$, where $\mathbf{1}$ is the vector of all 1's.

I. Revenge by Endless BFS

Let us compute $B_i = A^{2^i}$. Naturally, $B_0 = A$, and $B_{i+1} = B_i^2$. If we have $B_n u_0 \neq \mathbf{1}$, then the answer is -1 due to reasoning above.

I. Revenge by Endless BFS

Let us compute $B_i = A^{2^i}$. Naturally, $B_0 = A$, and $B_{i+1} = B_i^2$. If we have $B_n u_0 \neq \mathbf{1}$, then the answer is -1 due to reasoning above.

Otherwise, we search for the smallest k starting from the higher bits. Small detail: let us rather search for the largest k with $A^k u_0 \neq \mathbf{1}$, the actual answer is $k+1$.

I. Revenge by Endless BFS

Let us compute $B_i = A^{2^i}$. Naturally, $B_0 = A$, and $B_{i+1} = B_i^2$. If we have $B_n u_0 \neq \mathbf{1}$, then the answer is -1 due to reasoning above.

Otherwise, we search for the smallest k starting from the higher bits. Small detail: let us rather search for the largest k with $A^k u_0 \neq \mathbf{1}$, the actual answer is $k + 1$.

If we have $B_{n-1} u_0 = \mathbf{1}$, then $k < 2^{n-1}$. Otherwise, we have $k = 2^{n-1} + k'$ with the smallest $k' < 2^{n-1}$ such that $A^{2^{n-1}+k'} u_0 = A^{k'}(B_{n-1} u_0) = \mathbf{1}$.

I. Revenge by Endless BFS

Let us compute $B_i = A^{2^i}$. Naturally, $B_0 = A$, and $B_{i+1} = B_i^2$. If we have $B_n u_0 \neq \mathbf{1}$, then the answer is -1 due to reasoning above.

Otherwise, we search for the smallest k starting from the higher bits. Small detail: let us rather search for the largest k with $A^k u_0 \neq \mathbf{1}$, the actual answer is $k+1$.

If we have $B_{n-1} u_0 = \mathbf{1}$, then $k < 2^{n-1}$. Otherwise, we have $k = 2^{n-1} + k'$ with the smallest $k' < 2^{n-1}$ such that $A^{2^{n-1}+k'} u_0 = A^{k'}(B_{n-1} u_0) = \mathbf{1}$.

In both cases, the problem is the same, but in the latter case we have to update u_0 with $B_{n-1} u_0$, and add 2^{n-1} to the answer.

I. Revenge by Endless BFS

This solution requires computing $O(n)$ boolean matrix multiplications of size n, as well as $O(n)$ boolean matrix-vector multiplications; the latter is negligibly faster than the former.

I. Revenge by Endless BFS

This solution requires computing $O(n)$ boolean matrix multiplications of size n, as well as $O(n)$ boolean matrix-vector multiplications; the latter is negligibly faster than the former.

The boolean matrix multiplication can be computed with bitset optimizations in time $O(n^3/w)$, where w is the length of the machine word, and is equal to 32 or 64 depending on the machine type. The total complexity is $O(n^4/w)$, which seems too slow...

I. Revenge by Endless BFS

This solution requires computing $O(n)$ boolean matrix multiplications of size n, as well as $O(n)$ boolean matrix-vector multiplications; the latter is negligibly faster than the former.

The boolean matrix multiplication can be computed with bitset optimizations in time $O(n^3/w)$, where w is the length of the machine word, and is equal to 32 or 64 depending on the machine type. The total complexity is $O(n^4/w)$, which seems too slow...

...but the constant factor of bitsets is so small that this solution works fast enough.

I. Revenge by Endless BFS

This solution requires computing $O(n)$ boolean matrix multiplications of size n, as well as $O(n)$ boolean matrix-vector multiplications; the latter is negligibly faster than the former.

The boolean matrix multiplication can be computed with bitset optimizations in time $O(n^3/w)$, where w is the length of the machine word, and is equal to 32 or 64 depending on the machine type. The total complexity is $O(n^4/w)$, which seems too slow...

...but the constant factor of bitsets is so small that this solution works fast enough.

Bitsets are *good*.

I. Revenge by Endless BFS

This solution requires computing $O(n)$ boolean matrix multiplications of size n, as well as $O(n)$ boolean matrix-vector multiplications; the latter is negligibly faster than the former.

The boolean matrix multiplication can be computed with bitset optimizations in time $O(n^3/w)$, where w is the length of the machine word, and is equal to 32 or 64 depending on the machine type. The total complexity is $O(n^4/w)$, which seems too slow...

...but the constant factor of bitsets is so small that this solution works fast enough.

Bitsets are *good*.

Like, *really good*.

J. Farm Village

There are n houses along a road. Each house can grow up to two crops and needs to consume one crop. We know the cost p_i of growing a crop at each house, as well as the cost d_i of moving a unit of crop between any two adjacent houses. Find the smallest total cost to satisfy all houses.

J. Farm Village

We can solve the problem by min-cost max-flow. Construct a network as follows:

J. Farm Village

We can solve the problem by min-cost max-flow. Construct a network as follows:

- Create a source s, a sink t, and a vertex v_i for each house.

J. Farm Village

We can solve the problem by min-cost max-flow. Construct a network as follows:

- Create a source s, a sink t, and a vertex v_i for each house.
- Create edges from s to each v_i with capacity 2 and cost p_i (production).

J. Farm Village

We can solve the problem by min-cost max-flow. Construct a network as follows:

- Create a source s, a sink t, and a vertex v_i for each house.
- Create edges from s to each v_i with capacity 2 and cost p_i (production).
- Create edges between v_i and v_{i+1} with capacity ∞ and cost d_i (moving the crops).

J. Farm Village

We can solve the problem by min-cost max-flow. Construct a network as follows:

- Create a source s, a sink t, and a vertex v_i for each house.
- Create edges from s to each v_i with capacity 2 and cost p_i (production).
- Create edges between v_i and v_{i+1} with capacity ∞ and cost d_i (moving the crops).
- Create edges from v_i to t with capacity 1 and cost 0 (consumption).

J. Farm Village

We can solve the problem by min-cost max-flow. Construct a network as follows:

- Create a source s, a sink t, and a vertex v_i for each house.
- Create edges from s to each v_i with capacity 2 and cost p_i (production).
- Create edges between v_i and v_{i+1} with capacity ∞ and cost d_i (moving the crops).
- Create edges from v_i to t with capacity 1 and cost 0 (consumption).

The answer to the problem is the minimal cost of the maximal flow (which quite obviously has magnitude n). Of course, we can not find it explicitly since the network is too large.

J. Farm Village

Let us introduce houses one by one and recalculate the smallest cost. Suppose that the newest house has index i. Augment the flow by pushing one unit of flow along $s \to v_i \to t$ (i-th house produces a crop for itself, pay p_i, leave everything else unchanged).

J. Farm Village

Let us introduce houses one by one and recalculate the smallest cost. Suppose that the newest house has index i. Augment the flow by pushing one unit of flow along $s \to v_i \to t$ (i-th house produces a crop for itself, pay p_i, leave everything else unchanged).

The new flow may not be min-cost. We know that to transform it to a min-cost one we have to find negative-cost cycles in the residual network and push flow along them, until there are no negative cycles left.

J. Farm Village

If we assume that the previous answer was optimal, then there cannot be negative cycles that do not visit v_i. Further, there can be at most one negative cycle through v_i.

J. Farm Village

If we assume that the previous answer was optimal, then there cannot be negative cycles that do not visit v_i. Further, there can be at most one negative cycle through v_i.

Indeed, the edge $s \to v_i$ cannot be traversed in either direction. The edge $v_i \to t$ is present in the residual network in both directions, and at most one unit of flow can be pushed in either direction.

J. Farm Village

If we assume that the previous answer was optimal, then there cannot be negative cycles that do not visit v_i. Further, there can be at most one negative cycle through v_i.

Indeed, the edge $s \to v_i$ cannot be traversed in either direction. The edge $v_i \to t$ is present in the residual network in both directions, and at most one unit of flow can be pushed in either direction.

However, we cannot have that negative cycles traverse the $v_i \to t$ in both direction, since their sum would be a negative cycle with respect to the previous answer.

J. Farm Village

Now, to find a negative cycle. Let us store two multisets A and B — costs of paths that correspond to pushing one unit from s to v_i, and from v_i to s respectively.

J. Farm Village

Now, to find a negative cycle. Let us store two multisets A and B — costs of paths that correspond to pushing one unit from s to v_i, and from v_i to s respectively.

If $\min A - p_i < 0$, then we have a negative cycle consisting of pushing a unit flow from s to v_i, and cancelling the production at i.

J. Farm Village

Now, to find a negative cycle. Let us store two multisets A and B — costs of paths that correspond to pushing one unit from s to v_i, and from v_i to s respectively.

If $\min A - p_i < 0$, then we have a negative cycle consisting of pushing a unit flow from s to v_i, and cancelling the production at i.

Also, if $p_i + \min B < 0$, we can produce one more unit at i and cancel production somewhere prior to i.

J. Farm Village

How do we recalculate A and B? When we use a path of cost x from either A or B, we obtain a reverse path with cost $-x$ that should go to the opposite set.

J. Farm Village

How do we recalculate A and B? When we use a path of cost x from either A or B, we obtain a reverse path with cost $-x$ that should go to the opposite set.

Also, when proceeding from the house i to $i+1$, all costs in A increase by d_i, and all costs in B decrease by d_i.

J. Farm Village

How do we recalculate A and B? When we use a path of cost x from either A or B, we obtain a reverse path with cost $-x$ that should go to the opposite set.

Also, when proceeding from the house i to $i+1$, all costs in A increase by d_i, and all costs in B decrease by d_i.

We can store both A and B in (multi)sets to perform all operations efficiently.

J. Farm Village

This *appears* to work, but there are a few questions.

J. Farm Village

This *appears* to work, but there are a few questions.

When we augment paths from A to B, the “horizontal” edges between v_i change their capacity, which can result in changing costs of all other paths in A and B. Does this affect this solution?

J. Farm Village

This *appears* to work, but there are a few questions.

When we augment paths from A to B, the "horizontal" edges between v_i change their capacity, which can result in changing costs of all other paths in A and B. Does this affect this solution?

Is there another presentation/justification of this solution which doesn't have this problem? Or another, more clear solution even?

K. Conveyor Belt

A conveyor carries plates between positions 1 and n. Plates appear starting from time 1 at position 1 and move forward with uniform speed of one position per tick. A number of deliveries have to be made: we can put a product on a free plate at position a and take it later from position $b > a$ when it arrives. What is the smallest time to finish all deliveries? Delivery batches are introduced successively, answer for each prefix.

K. Conveyor Belt

Let T be the optimal answer. If a delivery is to be made from a to b, then it has to occupy a plate in all positions $a, a+1, \ldots, b$ for a tick. Also, in a position i the plates are unavailable for the first $i-1$ ticks.

K. Conveyor Belt

Let T be the optimal answer. If a delivery is to be made from a to b, then it has to occupy a plate in all positions $a, a+1, \ldots, b$ for a tick. Also, in a position i the plates are unavailable for the first $i-1$ ticks.

This implies that $T \geqslant \max_{i=1}^{k} (i - 1 + c_i)$, where c_i is the number of delivery segments that cover i, and k is the last position that expects a delivery.

K. Conveyor Belt

This estimate turns out to be precise. Indeed, let us show that when $T \geqslant \max_{i=1}^{k} (i - 1 + c_i)$ we can make all deliveries in time T.

K. Conveyor Belt

This estimate turns out to be precise. Indeed, let us show that when $T \geqslant \max_{i=1}^{k} (i - 1 + c_i)$ we can make all deliveries in time T.

Consider position k; it awaits several packages from previous positions. Schedule them to arrive on arbitrary distinct positions between n and T. The inequality above guarantees that we have enough distinct positions for this scheduling, namely, $c_k \leqslant T - k + 1$.

K. Conveyor Belt

This estimate turns out to be precise. Indeed, let us show that when $T \geqslant \max_{i=1}^{k} (i - 1 + c_i)$ we can make all deliveries in time T.

Consider position k; it awaits several packages from previous positions. Schedule them to arrive on arbitrary distinct positions between n and T. The inequality above guarantees that we have enough distinct positions for this scheduling, namely, $c_k \leqslant T - k + 1$.

For position $k - 1$, we have to schedule some arrivals as well. Since $c_{k-1} \leqslant T - (k - 1) + 1$, we can choose the times for arrivals to $k - 1$ even in presence of deliveries making their way to k.

K. Conveyor Belt

This estimate turns out to be precise. Indeed, let us show that when $T \geqslant \max_{i=1}^{k} (i - 1 + c_i)$ we can make all deliveries in time T.

Consider position k; it awaits several packages from previous positions. Schedule them to arrive on arbitrary distinct positions between n and T. The inequality above guarantees that we have enough distinct positions for this scheduling, namely, $c_k \leqslant T - k + 1$.

For position $k - 1$, we have to schedule some arrivals as well. Since $c_{k-1} \leqslant T - (k - 1) + 1$, we can choose the times for arrivals to $k - 1$ even in presence of deliveries making their way to k.

Proceeding further, we are able to schedule all arrivals and departures of packages before time T with no collisions.

K. Conveyor Belt

We maintain a single counter k for the last position expecting a delivery, as well the values of $a_i = i - 1 + c_i$.

K. Conveyor Belt

We maintain a single counter k for the last position expecting a delivery, as well the values of $a_i = i - 1 + c_i$.

Introducing a batch may result in increasing k, and also increases values of a_i in a range. To answer the query, we have to find the largest a_i in the range $[1, k]$.

K. Conveyor Belt

We maintain a single counter k for the last position expecting a delivery, as well the values of $a_i = i - 1 + c_i$.

Introducing a batch may result in increasing k, and also increases values of a_i in a range. To answer the query, we have to find the largest a_i in the range $[1, k]$.

Let us store the values of a_i in a lazy-propagation segment tree, this way both range-increasing and maximum-finding queries can be processed in $O(\log n)$ time. Hence, the total complexity is $O(q \log n)$.

L. Rock Climbing

A ciimber is a figure of five segments that can rotate and stretch arbitrarily (within length limits). The climber starts holding four rocks, and can move either an arm or a leg to another rock while changing positions of all body parts under conditions above. Find the smallest number of grabs needed to reach a target rock with any limb.

L. Rock Climbing

For all rocks a, b, c, d we want to count $dist_{a,b,c,d}$ — the smallest number of grabs needed to reach a position when grabbing rocks a and b with both arms, and rocks c and d with both legs (assuming that such a position is possible).

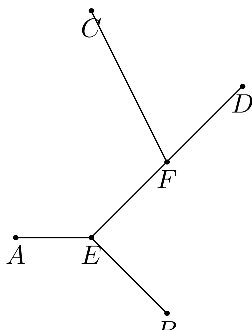
L. Rock Climbing

For all rocks a, b, c, d we want to count $dist_{a,b,c,d}$ — the smallest number of grabs needed to reach a position when grabbing rocks a and b with both arms, and rocks c and d with both legs (assuming that such a position is possible).

The transitions are natural: a transition is possible between any two realizable positions that differ in moving a limb. It now suffices to check which positions are at all realizable.

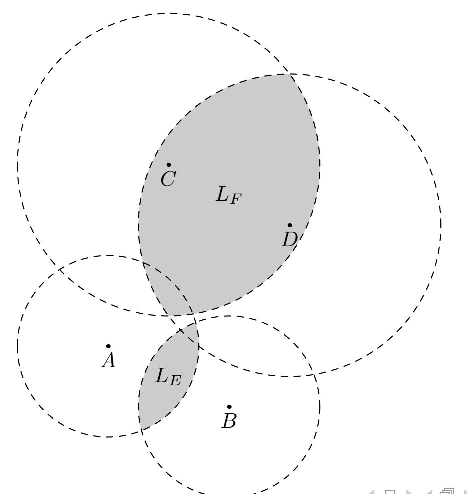
L. Rock Climbing

Let us check if it possible to grab rocks A, B with both arms and C, D with both legs. Points E and F are positions of "shoulders" (common point of the arms) and "waist" (common point of the legs) respectively.



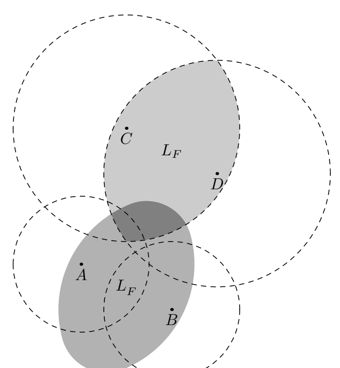
L. Rock Climbing

Clearly, the loci of E and F are bound to L_E and L_F — circle intersections centered at A, B and C, D respectively.



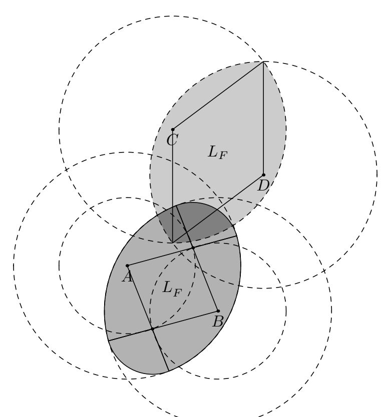
L. Rock Climbing

Moreover, F is bound to be at most the specified distance away from E. This means that F is additionally bound to L'_F — the Minkowski sum of L_E with a circle. The position is realizable as long as L_F and L'_F have non-empty intersection.



L. Rock Climbing

Clearly, borders of both L_F and L'_F consist of circle arcs, thus checking non-emptiness of their intersection boils down to intersecting several pairs of circles with additional checks.



L. Rock Climbing

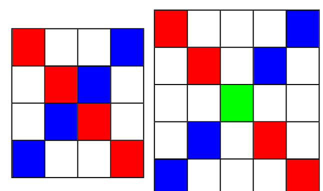
After checking realizability for all quarduplets of points, the problem boils down to finding a shortest path in the graph of realizable positions. The number of edges in this graph is $O(n^5)$, hence the total complexity.

M. Controlled Patterns

Find the number of ways to place n tokens in an $n \times n$ square so that each row, column, and large diagonal has at least one token, while avoiding a specified set of cells.

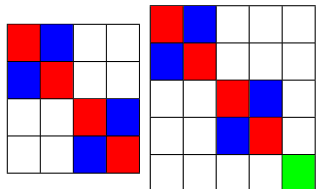
M. Controlled Patterns

Let us mark the cells on the major diagonal red, and the cells on the minor diagonal blue. If n is odd, we mark the intersection of diagonals green.



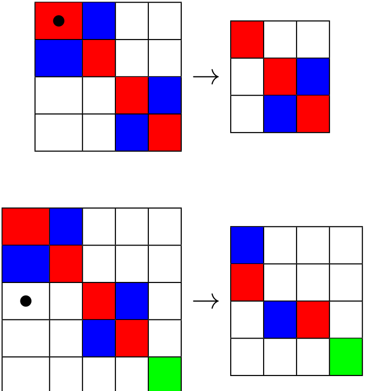
M. Controlled Patterns

First, let's assume that there are no cells we have to avoid. By rearranging rows and columns we can group colored cells in blocks, like this:



M. Controlled Patterns

Consider placing a token in a cell of the first column. This results in probably satisfying one of the diagonals condition, and forbidding to place a token in a row, effectively deleting this row.



M. Controlled Patterns

Observe that further token placements can result in obtaining different types of blocks, namely:

- full block: ;

M. Controlled Patterns

Observe that further token placements can result in obtaining different types of blocks, namely:

- full block: ;
- horizontal block: ;

M. Controlled Patterns

Observe that further token placements can result in obtaining different types of blocks, namely:

- full block: ;
- horizontal block: ;
- vertical block: ;

M. Controlled Patterns

Observe that further token placements can result in obtaining different types of blocks, namely:

- full block: ;
- horizontal block: ;
- vertical block: ;
- singular block: , , .

M. Controlled Patterns

Note that only the number of blocks of each type matters. Thus, we can compute DP that remembers the count of each type of blocks as well as whether the diagonal conditions are satisfied.

M. Controlled Patterns

Note that only the number of blocks of each type matters. Thus, we can compute DP that remembers the count of each type of blocks as well as whether the diagonal conditions are satisfied.

The transitions are straightforward: canonically choose the type of the block for the first column, and the type of block that intersects the chosen row (if we are placing the token outside the block)

M. Controlled Patterns

How do we deal with the cells we have to avoid? Let us apply inclusion-exclusion principle.

M. Controlled Patterns

How do we deal with the cells we have to avoid? Let us apply inclusion-exclusion principle.

Let S be the set of forbidden cells, and $T \subseteq S$. Let us place tokens in all cells in T, and compute the number of ways to extend this placement to a complete one.

M. Controlled Patterns

How do we deal with the cells we have to avoid? Let us apply inclusion-exclusion principle.

Let S be the set of forbidden cells, and $T \subseteq S$. Let us place tokens in all cells in T, and compute the number of ways to extend this placement to a complete one.

In any of cells of T share a row or a column, then there is no way that this placement can be extended to a complete placement.

M. Controlled Patterns

How do we deal with the cells we have to avoid? Let us apply inclusion-exclusion principle.

Let S be the set of forbidden cells, and $T \subseteq S$. Let us place tokens in all cells in T, and compute the number of ways to extend this placement to a complete one.

In any of cells of T share a row or a column, then there is no way that this placement can be extended to a complete placement.

If any of cells in T happen to lie on one of the diagonals, then we do not care about covering this diagonal anymore.

M. Controlled Patterns

How do we deal with the cells we have to avoid? Let us apply inclusion-exclusion principle.

Let S be the set of forbidden cells, and $T \subseteq S$. Let us place tokens in all cells in T, and compute the number of ways to extend this placement to a complete one.

In any of cells of T share a row or a column, then there is no way that this placement can be extended to a complete placement.

If any of cells in T happen to lie on one of the diagonals, then we do not care about covering this diagonal anymore.

Erase the rows and columns covered by cells in T; this gives rise to a configuration of blocks described above. Count the number of complete placements with the same DP.

M. Controlled Patterns

If $f(T)$ is the number of ways to extend the placement of tokens in the cells of T, then the answer is $\sum_{T \subseteq S}(-1)^{|T|} f(T)$ by inclusion-exclusion principle.

M. Controlled Patterns

If $f(T)$ is the number of ways to extend the placement of tokens in the cells of T, then the answer is $\sum_{T \subseteq S} (-1)^{|T|} f(T)$ by inclusion-exclusion principle.

The complexity of this solution is roughly $O(n^8)$ (?), but considering only reachable configurations of blocks and reusing the answers should be fast enough in the end.