SPb SU Contest: LVII SPb SU Championship

August 27, 2023

Mixed Messages

- We are given a string. Our goal is to find the minimal number of swaps of adjacent letters required to obtain a substring "`spbsu`" without changing the relative order of letters of this string.

Mixed Messages

- We are given a string. Our goal is to find the minimal number of swaps of adjacent letters required to obtain a substring "spbsu" without changing the relative order of letters of this string.
- The key observation is that the cost of each non-chosen letter depends on its position inside the chosen subsequence. Letters near either end of the word cost one swap, while letters close to the center cost two swaps.

Mixed Messages

- We are given a string. Our goal is to find the minimal number of swaps of adjacent letters required to obtain a substring "`spbsu`" without changing the relative order of letters of this string.
- The key observation is that the cost of each non-chosen letter depends on its position inside the chosen subsequence. Letters near either end of the word cost one swap, while letters close to the center cost two swaps.
- Now, there are two possible approaches.
  1. A dynamic programming `dp[n][k]`: the minimum number of swaps, if we have considered the first n characters of the input string and typed k of them into the string "`spbsu`".

Mixed Messages

- We are given a string. Our goal is to find the minimal number of swaps of adjacent letters required to obtain a substring "`spbsu`" without changing the relative order of letters of this string.
- The key observation is that the cost of each non-chosen letter depends on its position inside the chosen subsequence. Letters near either end of the word cost one swap, while letters close to the center cost two swaps.
- Now, there are two possible approaches.
  1. A dynamic programming `dp[n][k]`: the minimum number of swaps, if we have considered the first n characters of the input string and typed k of them into the string "`spbsu`".
  2. A greedy algorithm that tries all possibilities for the central letter "`b`" and then selects other letters of the string "`spbsu`" to be as close to the chosen letter as possible.

B (I flipped the Calendar...)

- We are interested in the number of days that are Mondays or the first days of the month.
- If both events happen, the day is still counted exactly once.
- You can use the unbuilt date/time functions. We were kind enough to make the Year 2038 problem impossible.
- Or you can start counting from the 1st of January; you only need to figure out on which day of the week the 1st of January lands on.

Many Many Cycles

- Given a graph, we need to find the GCD of lengths of all cycles in it.

Many Many Cycles

- Given a graph, we need to find the GCD of lengths of all cycles in it.
- This problem has a proven deterministic solution (that we are going to discuss), but may also be solved by many randomizations that we can't neither prove nor break.

Many Many Cycles

- Given a graph, we need to find the GCD of lengths of all cycles in it.
- This problem has a proven deterministic solution (that we are going to discuss), but may also be solved by many randomizations that we can't neither prove nor break.
- The algorithm itself is simple. We need to build a spanning tree and find the GCD of all cycles that have at most two edges outside the tree. It will be the answer.

Many Many Cycles

- Given a graph, we need to find the GCD of lengths of all cycles in it.
- This problem has a proven deterministic solution (that we are going to discuss), but may also be solved by many randomizations that we can't neither prove nor break.
- The algorithm itself is simple. We need to build a spanning tree and find the GCD of all cycles that have at most two edges outside the tree. It will be the answer.
- Let g be the answer found by our algorithm. This number obviously cannot be smaller (in this context 0 is infinity) than the real answer, so we only have to prove that it is not greater or, equivalently, that g divides the length of each cycle.

Many Many Cycles

- Let a cycle be *simple* if it contains at most one edge outside the tree. The condition we check verifies that all simple cycles are divisible by g and also that the doubled length of the intersection of any two simple cycles is divisible by g.

Many Many Cycles

- Let a cycle be *simple* if it contains at most one edge outside the tree. The condition we check verifies that all simple cycles are divisible by g and also that the doubled length of the intersection of any two simple cycles is divisible by g.
- Note that the intersection of any number of simple cycles is actually an intersection of some two of them.

Many Many Cycles

- Let a cycle be *simple* if it contains at most one edge outside the tree. The condition we check verifies that all simple cycles are divisible by g and also that the doubled length of the intersection of any two simple cycles is divisible by g.
- Note that the intersection of any number of simple cycles is actually an intersection of some two of them.
- Any cycle could be represented as a XOR of several simple cycles, therefore, its length could be found as a linear combination with integer coefficients of lengths of those cycles and doubled lengths of their intersections. All summands are divisible by g, therefore, g divides the length of the cycle as well.

Bishops

- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.

Bishops

- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.
- There is a general way of solving the problem and a ton of explicit constructive solutions. We start with the first one.

Bishops

- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.
- There is a general way of solving the problem and a ton of explicit constructive solutions. We start with the first one.
- Consider a bipartite graph with vertices denoting diagonals and edges denoting their intersections (i.e., squares on a chessboard).
- We can consider white and black squares independently, and we need to find a maximal matching in this graph.

Bishops

- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.
- There is a general way of solving the problem and a ton of explicit constructive solutions. We start with the first one.
- Consider a bipartite graph with vertices denoting diagonals and edges denoting their intersections (i.e., squares on a chessboard).
- We can consider white and black squares independently, and we need to find a maximal matching in this graph.
- Note that each vertex in one part is connected to a segment of vertices in another.

Bishops

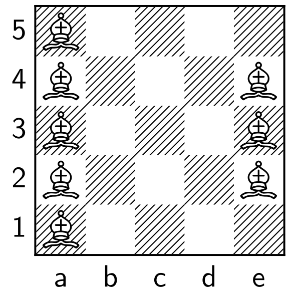
- We have to place the maximum number of bishops on an $n \times m$ chessboard in such a way that none of them attack each other.
- There is a general way of solving the problem and a ton of explicit constructive solutions. We start with the first one.
- Consider a bipartite graph with vertices denoting diagonals and edges denoting their intersections (i.e., squares on a chessboard).
- We can consider white and black squares independently, and we need to find a maximal matching in this graph.
- Note that each vertex in one part is connected to a segment of vertices in another.
- Finding a matching in a graph with this property is easy: we can simply sort all segments by their right ends and greedily choose the leftmost suitable vertex for each of them.

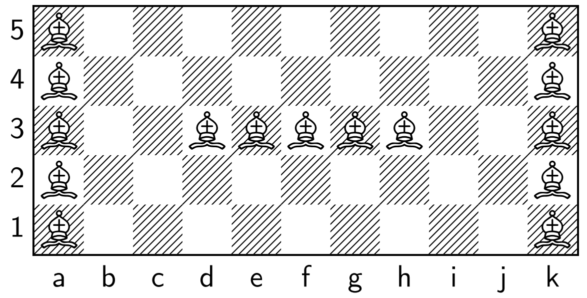
Bishops

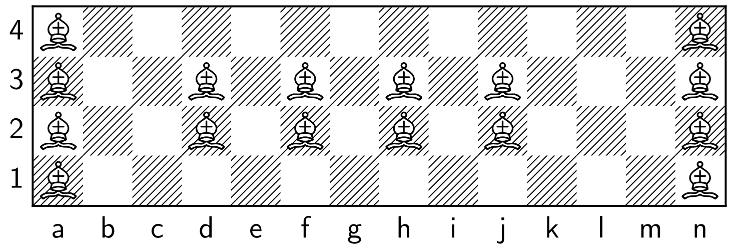
- Now, the explicit construction.
- Rotate the board horizontally and then place bishops in cells with the following priority.
  1. On the left edge of the board.
  2. On the right edge of the board.
  3. On the one or two (depending on the parity of the board dimensions) middle rows going from the left to the right.

Bishops

- Now, the explicit construction.
- Rotate the board horizontally and then place bishops in cells with the following priority.
  1. On the left edge of the board.
  2. On the right edge of the board.
  3. On the one or two (depending on the parity of the board dimensions) middle rows going from the left to the right.
- Some examples:







E (Fischer's Chess's Guessing Game)

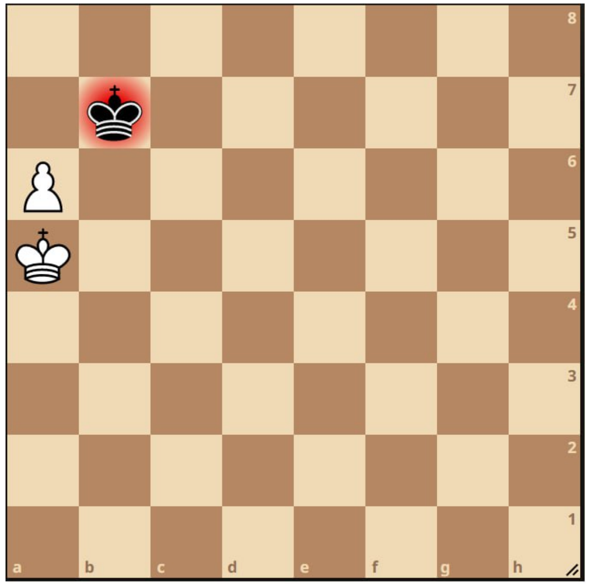
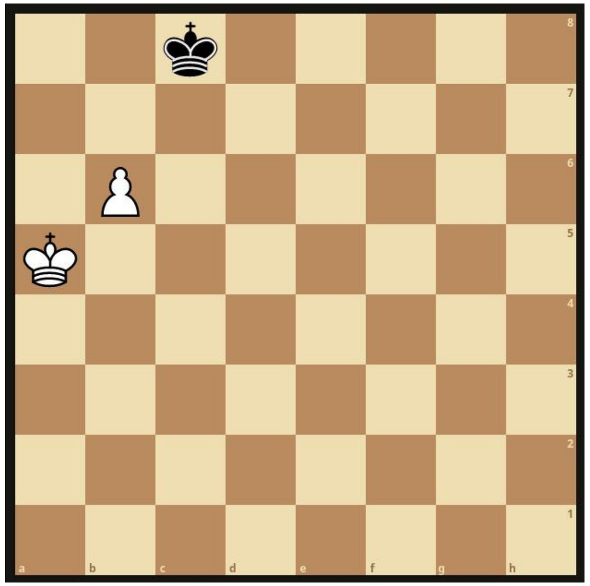
- Use a greedy algorithm. Let the current *state* be the set of positions that still could be the corrent answer.
- Suppose that we guessed a position g.
- Distribute the positions from the current state into 9 bin The position h goes to the bin with a number that would be the Jill's response if the correct answer was h.
- Here the greedy comes: enumerate over all possible g-s and pick g that minimizes the size of the most filled bin.
- An exhaustive search proves that 5 turns isn't enough.

F (Forward-Capturing Pawns)

- A classic game theory problem (the threefold repetition rule doesn't change the outcome and only makes all games finite).
- Retrograde analysis. Remember that stalemates are draws (not losses).
- Implement all chess rules (both old and new) *very* carefully.
- If your implementation does not fit into the time limit, consider the following facts.
- The positions can be easily encoded as a single number.
- A symmetry with respect to a vertical axis doesn't change the outcome.
- You can precompute the answers.

F (Forward-Capturing Pawns), continuation

- You can also try to create a full list of *rules* that determine the outcome of the game.
- It is *very* difficult. For example, it is *not* true that white win in each situation where the pawn is defended and there are no immediate stalemate troubles.

Graph Cuts

- Given a graph, we are asked to maintain a set U and answer queries to find an edge between U and its complement and remove it or to report they do not exist.

Graph Cuts

- Given a graph, we are asked to maintain a set U and answer queries to find an edge between U and its complement and remove it or to report they do not exist.
- There are two approaches: sqrt-decomposition and bitsets. The first one can lead to two different solutions, so we start with it.

Graph Cuts

- Given a graph, we are asked to maintain a set U and answer queries to find an edge between U and its complement and remove it or to report they do not exist.
- There are two approaches: sqrt-decomposition and bitsets. The first one can lead to two different solutions, so we start with it.
- The first solution. Naive.
- Arrange vertices into two types: light and heavy, depending on their degree in the original graph.

Graph Cuts

- Given a graph, we are asked to maintain a set U and answer queries to find an edge between U and its complement and remove it or to report they do not exist.
- There are two approaches: sqrt-decomposition and bitsets. The first one can lead to two different solutions, so we start with it.
- The first solution. Naive.
- Arrange vertices into two types: light and heavy, depending on their degree in the original graph.
- For each heavy vertex, we will explicitly store all its neighbors of its color and the opposite color separately. To maintain this, when changing the color of any vertex, we will need to make at most $\mathcal{O}(\sqrt{m})$ additions and deletions, and at most one swap of the lists (if the recolored vertex is heavy).

Graph Cuts

- We solved our problem with edges that have at least one endpoint in a heavy vertex, but there are still edges passing between light vertices.
- They can all be added to one large lazy queue for light edges since any recoloring of a light vertex will affect no more than $\mathcal{O}(\sqrt{m})$ edges.

Graph Cuts

- We solved our problem with edges that have at least one endpoint in a heavy vertex, but there are still edges passing between light vertices.
- They can all be added to one large lazy queue for light edges since any recoloring of a light vertex will affect no more than $\mathcal{O}(\sqrt{m})$ edges.
- The second solution. Short and without constants.
- Get rid of a queue with light edges! Instead, we can treat all vertices as heavy, but only store edges to vertices with smaller of equal initial degrees.

Graph Cuts

- We solved our problem with edges that have at least one endpoint in a heavy vertex, but there are still edges passing between light vertices.
- They can all be added to one large lazy queue for light edges since any recoloring of a light vertex will affect no more than $\mathcal{O}(\sqrt{m})$ edges.
- The second solution. Short and without constants.
- Get rid of a queue with light edges! Instead, we can treat all vertices as heavy, but only store edges to vertices with smaller of equal initial degrees.
- The only problem is the number of heavy vertices, which we solve by maintaining vertices with non-empty opposite-color neighbour lists in a lazy queue.

Graph Cuts

- We solved our problem with edges that have at least one endpoint in a heavy vertex, but there are still edges passing between light vertices.
- They can all be added to one large lazy queue for light edges since any recoloring of a light vertex will affect no more than $\mathcal{O}(\sqrt{m})$ edges.
- The second solution. Short and without constants.
- Get rid of a queue with light edges! Instead, we can treat all vertices as heavy, but only store edges to vertices with smaller of equal initial degrees.
- The only problem is the number of heavy vertices, which we solve by maintaining vertices with non-empty opposite-color neighbour lists in a lazy queue.
- The running time of both solutions is $\mathcal{O}(m + q\sqrt{m})$ (assuming $n \leq m$ and $q > 0$).

Graph Cuts

- There is an alternative approach with bitsets.
- Let’s store the incidence matrix as a vector of bitsets and maintain the set of all edges in the cut (also as a bitset).

Graph Cuts

- There is an alternative approach with bitsets.
- Let's store the incidence matrix as a vector of bitsets and maintain the set of all edges in the cut (also as a bitset).
- Each add/remove operation is just a XOR operation of some matrix row with our current set. Each "?" query is just finding true bit in a bitset and flipping it.

Graph Cuts

- There is an alternative approach with bitsets.
- Let's store the incidence matrix as a vector of bitsets and maintain the set of all edges in the cut (also as a bitset).
- Each add/remove operation is just a XOR operation of some matrix row with our current set. Each "?" query is just finding true bit in a bitset and flipping it.
- The challenging part is avoiding Memory Limit Exceeded. To do so, we need to compress the incidence matrix.
- One way of doing this is to store it as a list of pairs of an index and a mask value.

Graph Cuts

- There is an alternative approach with bitsets.
- Let's store the incidence matrix as a vector of bitsets and maintain the set of all edges in the cut (also as a bitset).
- Each add/remove operation is just a XOR operation of some matrix row with our current set. Each "?" query is just finding true bit in a bitset and flipping it.
- The challenging part is avoiding Memory Limit Exceeded. To do so, we need to compress the incidence matrix.
- One way of doing this is to store it as a list of pairs of an index and a mask value.
- The complexity is $\mathcal{O}(m/w)$ per query. Some secondary optimizations may be required to get Accepted.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.
- We will solve the more general version of a problem for k allowed operations per query (our case is $k = 2$). The construction is recursive.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.
- We will solve the more general version of a problem for k allowed operations per query (our case is $k = 2$). The construction is recursive.
- Split the array into blocks of size B (will be defined later), compute prefix and suffix sums inside each block and construct the sparse table with $k - 2$ operations on the blocks themselves.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.
- We will solve the more general version of a problem for k allowed operations per query (our case is $k = 2$). The construction is recursive.
- Split the array into blocks of size B (will be defined later), compute prefix and suffix sums inside each block and construct the sparse table with $k - 2$ operations on the blocks themselves.
- Now, we can answer any query that is not contained inside a single block.

Very Sparse Table

- We need to construct a sparse table that will work with an arbitrary monoid and answer queries by making no more than two monoid operations per query and be more efficient than the regular disjoint sparse table.
- We will solve the more general version of a problem for k allowed operations per query (our case is $k = 2$). The construction is recursive.
- Split the array into blocks of size B (will be defined later), compute prefix and suffix sums inside each block and construct the sparse table with $k - 2$ operations on the blocks themselves.
- Now, we can answer any query that is not contained inside a single block.
- To answer any other query we need to construct the same sparse table recursively inside each block.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B?

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B? The optimal B is such that the construction on top of the blocks is linear.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B? The optimal B is such that the construction on top of the blocks is linear.
- For example, in our case $B \approx \sqrt{n}$, and if we had one operation more, the optimal choice would have been $B \approx \log n$.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B? The optimal B is such that the construction on top of the blocks is linear.
- For example, in our case $B \approx \sqrt{n}$, and if we had one operation more, the optimal choice would have been $B \approx \log n$.
- It could be shown that this construction is optimal both in time and memory.

Very Sparse Table

- What are sparse tables with zero and one operations per query?
- With zero operations we have to precompute all possible answers (in $\Theta(n^2)$ time). With one operation we can build a disjoint sparse table.
- How to choose B? The optimal B is such that the construction on top of the blocks is linear.
- For example, in our case $B \approx \sqrt{n}$, and if we had one operation more, the optimal choice would have been $B \approx \log n$.
- It could be shown that this construction is optimal both in time and memory.
- In case we know that the number of queries is $\Theta(n)$, we can achieve complexity $\Theta(n\alpha(n))$ where α is the inverse Ackermann function (and this is also optimal).

Operations per query	Time to construct (Θ omitted)
0	n^2
1	$n \log n$
2	$n \log \log n$
3	$n \log^* n$
4	$n \log^* n$
5	$n \log^{**} n$
6	$n \log^{**} n$
7	$n \log^{***} n$
$\vdots$	$\vdots$
k + 1	$n \log^{\overbrace{* * \cdots *}^{\lfloor k/2 \rfloor}} n$
$\vdots$	$\vdots$

I (Password)

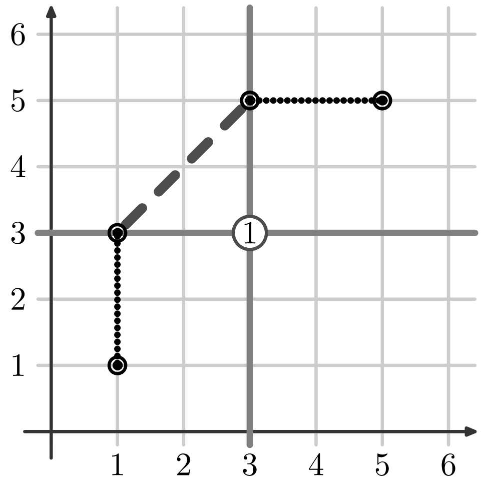
- The simplest problem of the contest. Solved by almost all teams.
- We are allowed to make a password fully consisting from non-letters. Therefore, the maximum number of non-letters is n, where n is the length of the password.
- If we ignore the center requirement, we need at least $\lfloor n/3 \rfloor$ non-letters.
- To deal with the center requirement, fill the center with non-letters; the remaining string splits into two parts of equal length. By itself, each part is equivalent to the case with no center requirement.

Statement

Problem J: Transport Pluses

In this problem, we have to get from point A to point B on a plane.

- You can go from one point to another directly, paying for the Euclidean distance between them.
- Also, you can use $n \leqslant 100$ transport pluses (x_i, y_i).
- Pay t, and get from any point with $x = x_i$ OR $y = y_i$ to any other such point.
- The goal is to get from A to B with minimum possible total cost.

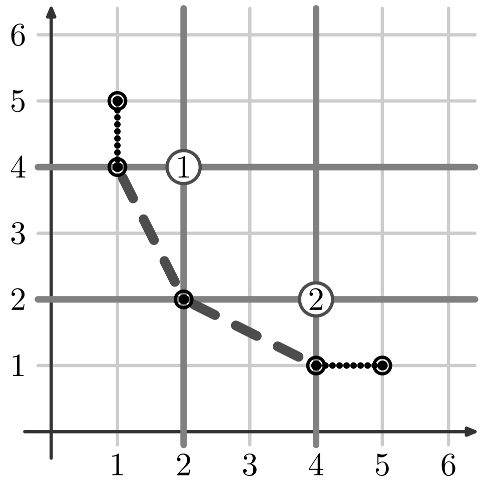


What Can a Path Look Like

Consider a travel path from A to B: some direct travel, using a plus, another direct travel, using another plus, and so on.

- Each pair of pluses is connected.
- So, if we use at least two pluses, our travel can skip every step between the first and the last plus.
- To move from a point to plus i (and the other way too), we move vertically to its x_i or horizontally to its y_i, whichever is closer.

This leaves only a small number of cases to consider as possible optimal solutions.

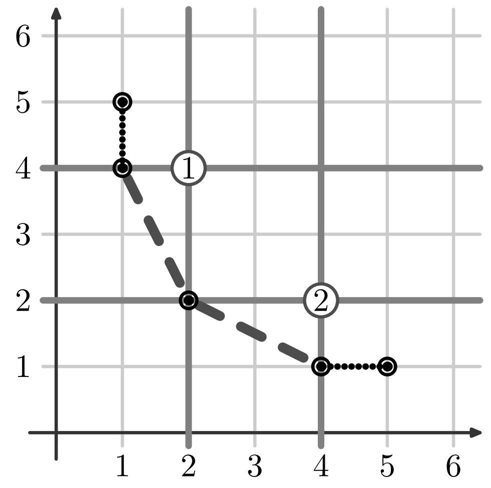


What Can a Path Look Like

Consider a travel path from A to B: some direct travel, using a plus, another direct travel, using another plus, and so on.

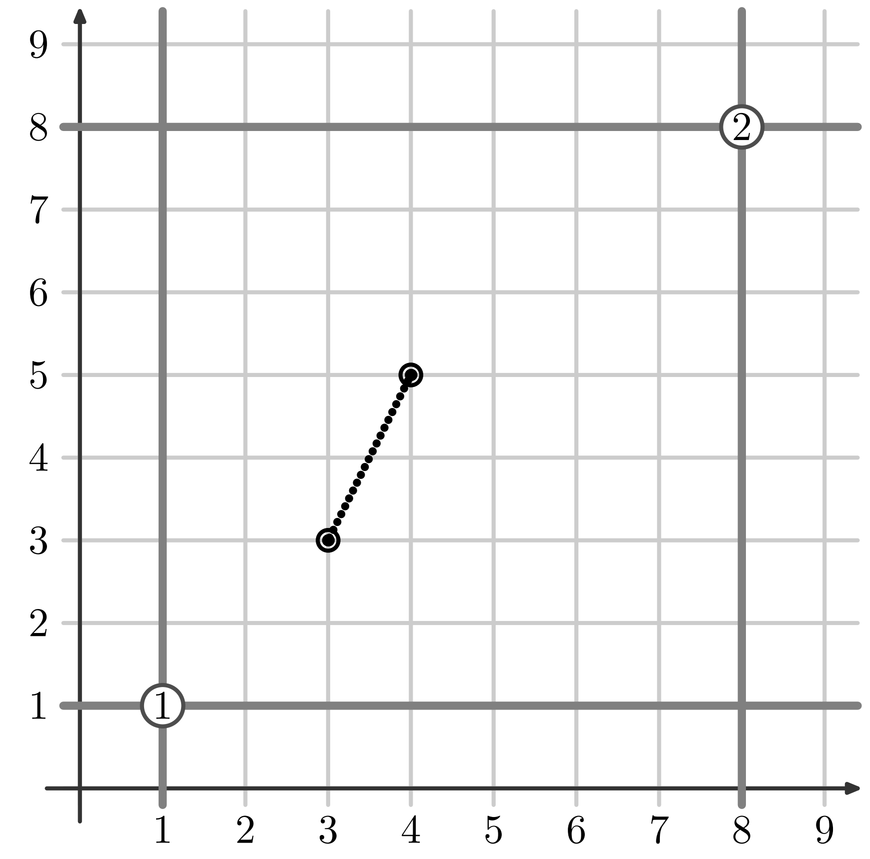
- Each pair of pluses is connected.
- So, if we use at least two pluses, our travel can skip every step between the first and the last plus.
- To move from a point to plus i (and the other way too), we move vertically to its x_i or horizontally to its y_i, whichever is closer.

This leaves only a small number of cases to consider as possible optimal solutions.



Possible Cases

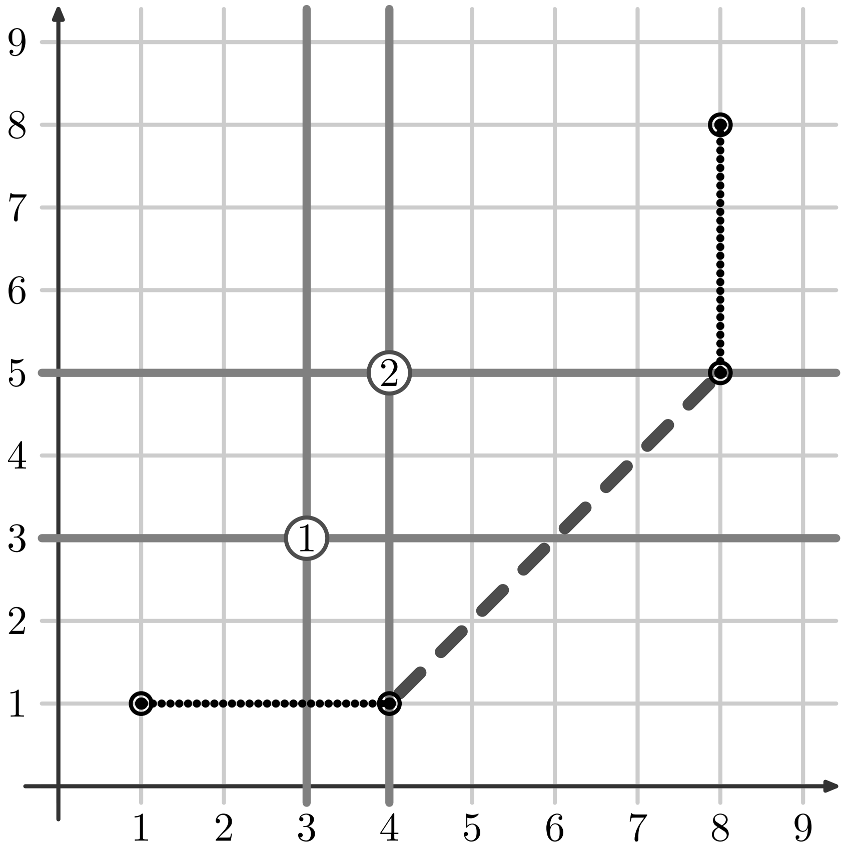
Case 1: direct travel.



Can check in $\mathcal{O}(1)$.

Possible Cases

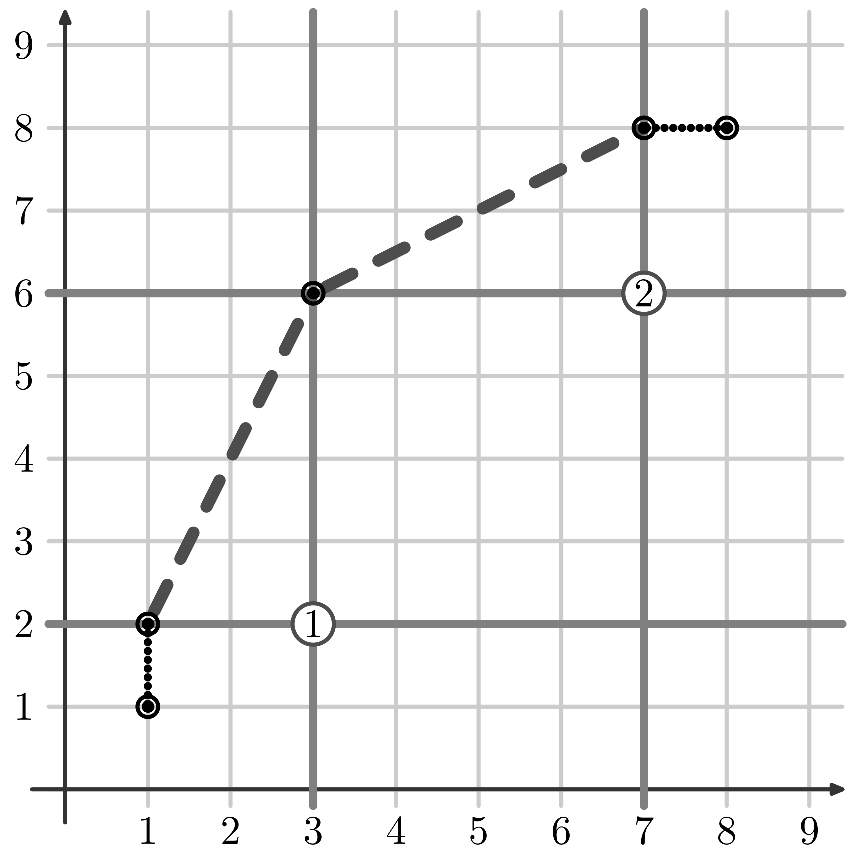
Case 2: travel to a plus, use it, travel to destination.



Can check in $\mathcal{O}(n)$.

Possible Cases

Case 3: travel to a plus, transfer to another plus, get closer to destination, go directly.



Can check in $\mathcal{O}(n^2)$, or in $\mathcal{O}(n)$ if we precalculate distances from A and B to each plus.

K (Poor Students)

- A standard min-cost maxflow problem, but the constraints are too big.
- Indeed, build a flow network with two parts: the left one with the students and the right one with exams. Create an edge from i-th student to j-th exam with capacity 1 and cost $c_{i,j}$. Create $\mathrm{cap} = 1, \mathrm{cost} = 0$ edges from the source to all students. Create a $\mathrm{cap} = a_j, \mathrm{cost} = 0$ edges from the j-th exam to the sink.
- The standard min-cost maxflow algorithm repeatedly finds the shortest path in the residue network. How does it look like? It looks like $s \to \ell_0 \to r_1 \to \ell_1 \to r_2 \to \ldots \to \ell_u \to r_u \to t$. Here, ℓ_i are the students and r_i are the exams.
- Basically, we take a student ℓ_0 and make them pass the exam r_1. To compensate, we take a student ℓ_1, who passed r_1 but not r_2 and transfer them from r_1 to r_2, take ℓ_2 and transfer them from r_2 to r_3, e.t.c.

K (Poor Students), continuation

- Then, for each pair of the courses j and k, we only need to know the best student we can transfer from j to k. The quality of the student i is defined by $c_{i,k} - c_{i,j}$: the less, the better.
- Now, instead of searching for the shortest path in the whole graph, we can construct a smaller graph on the courses and search for a short path within the graph (of course, we still need to take the student ℓ_0 into account; to do so, keep a best new student for each course).
- We have n iterations, each taking $O(k^3)$ time (Ford-Bellman in the course graph).
- To construct the graph quickly, keep $O(k^2)$ sets (described below).
- For each course, order the potential new students by their cost: k sets.
- For each pair of courses, order the potential transfers by their cost (defined above): k^2 sets.
- Each time, at most k students actually transfer. Therefore, we need to do $O(k^2)$ set updates. Total time for set updates: $O(nk^2 \log n)$.
- $O(nk^3 + nk^2 \log n)$ time in total.

Statement

Problem L: “Memo” Game With a Hint

- In this problem, we play the “Memo” game:
  - There are 50 cards lying face down.
  - On the faces, there are 25 pictures, each appears twice.
  - In a turn, open one card, then open another, trying to find the same picture.
  - If the pictures are the same, leave the pair face up.
  - If the pictures are different, turn the two cards face down again.
  - The goal is to open all cards in as little turns as possible.

Statement

Problem L: “Memo” Game With a Hint

- In this problem, we play the “Memo” game:
  - There are 50 cards lying face down.
  - On the faces, there are 25 pictures, each appears twice.
  - In a turn, open one card, then open another, trying to find the same picture.
  - If the pictures are the same, leave the pair face up.
  - If the pictures are different, turn the two cards face down again.
  - The goal is to open all cards in as little turns as possible.
- During the preparation phase, we can cheat: look at all the pictures, and then rotate some of the cards 180 degrees before turning them face down.
- In the main phase, we don’t remember the pictures, but we see the rotations.

Basic Strategy

Here is the strategy without hints.
Before each turn, there are the following types of cards:

- known pairs face up,
- known pairs face down,
- known cards without a known pair,
- unknown cards.

On each turn:

- If there is a known pair face down, open it.
- Otherwise, open an unknown card.
- If we know its pair, open the pair.
- Otherwise, open another unknown card.
- Either we found a new pair by luck, or we now know 2 more cards.

This strategy has ≈ 14.83 misses per game. We want 13.50 misses per game on average. There are several ways to get there.

Basic Strategy

Here is the strategy without hints.
Before each turn, there are the following types of cards:

- known pairs face up,
- known pairs face down,
- known cards without a known pair,
- unknown cards.

On each turn:

- If there is a known pair face down, open it.
- Otherwise, open an unknown card.
- If we know its pair, open the pair.
- Otherwise, open another unknown card.
- Either we found a new pair by luck, or we now know 2 more cards.

This strategy has $\approx$ 14.83 misses per game. We want 13.50 misses per game on average. There are several ways to get there.

Basic Strategy

Here is the strategy without hints.
Before each turn, there are the following types of cards:

- known pairs face up,
- known pairs face down,
- known cards without a known pair,
- unknown cards.

On each turn:

- If there is a known pair face down, open it.
- Otherwise, open an unknown card.
- If we know its pair, open the pair.
- Otherwise, open another unknown card.
- Either we found a new pair by luck, or we now know 2 more cards.

This strategy has ≈ 14.83 misses per game. We want 13.50 misses per game on average. There are several ways to get there.

Using the Hint: Solution 1

Here is a simple idea for the solution.

- Note that, in 13 turns, we can learn about 26 cards.
- In preparation phase, for each of the 25 pairs, rotate one card and don't rotate another.
- In playing phase, spend the first 13 turns (and 13 misses) to see all rotated cards.
- In the next turns, open a non-rotated card. We already know where its pair is.
- With a patch, can do 12 misses instead of 13.

Using the Hint: Solution 2

Here is a mechanical solution.

- Treat the hint as 50 bits of information.
- Write down the information allowing to open pairs as long as we have bits for it.

For example:

- Write down the number of pair for the leftmost card (49 choices).
- Out of the 48 remaining cards, pick the leftmost one and write down the number of its pair (47 choices).
- And so on.
- We can transfer information for 9 turns: $49 \cdot 47 \cdot 45 \cdot 43 \cdot 41 \cdot 39 \cdot 37 \cdot 35 \cdot 33 = 304\,513\,870\,485\,825$, and it is less than $2^{50} = 1\,125\,899\,906\,842\,624$.
- After that, use the basic strategy, but for $16 \cdot 2 = 32$ remaining pairs.
- The number of misses is ≈ 9.31. We can use the gap to skip some technicalities.

M (Hardcore String Counting)

- Denote: g_n — the number of good strings of length n, h_n — the number of strings of length n that don't contain w, $A := 26$ — the alphabet size, $m := |w|$.
- All good strings are obtained in the following way: take a string that doesn't contain w and append w. But not all such strings are good.
- The problem: the last m letters *are not* the first appearance of w.
- Iterate over the first appearance.
- A *border* p of a string s — a string that is simultaneously a prefix and a suffix of s.
- The result: $g_n = h_{n-m} - \sum_{\text{all borders } p \text{ of } w} g_{n-m+|p|}$.

M (Hardcore String Counting), continuation

- Also, $h_n - Ah_{n-1} = g_n$. Therefore,
  $$Ag_{n-1} = Ah_{n-m-1} - \sum_{\text{all borders } p \text{ of } w} Ag_{n-m-1+|p|}$$
- Subtract. All h-s disappear, because $h_{n-m} - Ah_{n-m-1} = g_{n-m}$.
- The result is a linear recurrence for g-s with $m + O(1)$ terms.
- Compute the n-th term by a standard $O(m \log m \log n)$ algorithm.
- Each time, we take the answer modulo the same polynomial, so we don't need to invert a series more than once. Then, we have $O(\log n)$ iterations, each taking $O(m \log m)$ time, but $O(m \log m)$ comes from a normal polynomial mutliplication and not from a costier inversion.
- Also, there is an even faster way to divide by this particular polynomial. It exploits the fact that border lengths split into $O(\log m)$ arithmetic progressions. However, it wasn't necessary to solve the problem.