Problem Tutorial: "Area of Smileys Intersection"

Problem: Find the area of intersection of two smileys — circles with circular and semicircular holes.

It is possible to obtain a precise solution, but it is fairly complex (hint: use inclusion-exclusion principle). We will describe an easier solution that uses numerical integration. The basic idea is to cut the plane into thin strips of width, say, ε, and then approximate parts of the intersection inside each strip with rectangles (or trapezoids).

Here's one way to reduce to one-dimensional problem inside each strip. We can consider each outer circle of a smiley as a "+1 over an area" operation, and each hole as a "-1 over an area" operation. Upon fixing a strip, we will intersect each of the objects with the strip to obtain a number of "+/-1 over a segment" operations.

We now have to count total length of segment parts that have a total result of 2 after these operations. This is a standard scan-line application.

We have to choose small enough ε to obtain high enough precision, however, that increases computation time. In this problem, it may be impossible to achieve both incentives with basic methods. One way of solving this issue is to use adaptive precision method that uses thinner strips in regions where higher precision is crucial.

- First, cut the region into fairly wide strips, say, $\varepsilon = 1$.
- For each strip, compute the answer A. Next, try to cut it into two halves and compute the total answer for the halves $A_1 + A_2$. If the difference between A and $A_1 + A_2$ is large, then we recursively proceed to halves of the strip. Otherwise, we consider A to be good enough approximation and halt.

Problem Tutorial: "Bakery"

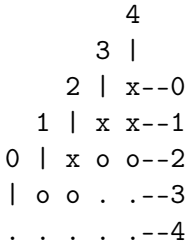
Let's move through the days from the first to last, keeping the multiset of the pairs "cost/quantity". When moving from day i to day $i+1$ we will add e_i to all members of the multiset. While processing the single day we will add c_i and f_i to the multiset, then we will try to answer the request by the greedy way (taking from the multiset). The stock limitations can be implemented while moving between the days by removal of the elements from the multiset until the quantity of the loafs remaining will be no more than g_i.

Time complexity $O(N \log N)$. Memory complexity $O(N)$.

Problem Tutorial: "Coverings" We will use the profile DP approach.

The state (i, j, p):

- first $i - 1$ lines are filled;
- In i-th line first $j - 1$ cells are filled;
- among the next i cells with the coordinates $(i + 1, 0)$, $(i + 1, 1)$, ..., $(i + 1, j - 1)$, (i, j), $(i, j + 1)$, ..., $(i, n - 1)$ are filled only those that correspond to 1's in the profile p.



'x' denote the filled cells. '.' denote the empty cells, 'o' denote the cells that are filled or empty corresponding with p.

The $f(i, j, p)$ is number of the ways to complete the configuration.

$(i, j, p) \to (i, j + 1, q)$ or to the new line, if the current one is over.

If in p j-th bit is set to 1, then move to $q{=}p$ with this bit set to zero. If it is equal to zero, that no more than 2 branches are possible — put the triangle with single circle up or single circle down (if there is a free place for this action).

We will calculate $f(i, j, p)$ from bottom to top. Then we will move from top to bottom, keeping the current profile. If there is single branch, we follow it. If there are two branches, consider one that puts 'A' in (i, j):

- if it is less than the current index, we put 'A' and follow that branch;
- Otherwise subtract the number of ways to put 'A' from the current index, after that we put 'B' and use alternative branch.

Possible optimizations:

- Keep $f(i, j, p)$ only for the reachable states;
- To save memory and optimize the cache usage, use profile DP only for the heights 20 or less, and after it switch to some form of brute force.

Problem Tutorial: "Dense Polygon"

For each convex polygon downmost leftmost point can be determined uniquely.

Let's sort all points by y and then by x and iterate over all points to choose the downmost-leftmost point, let's call it **BASE**.

As soon as we fixed BASE we should try to construct the polygon that are after BASE in our $y - x$ order. Firstly, let's sort these points by polar angle of vector starting in BASE.

Now, every convex polygon's clockwise order is a subsequence.

We can calculate $dp[PRELAST][LAST][TAKEN]$, that is the minimal area of convex polygon where $PRELAST$ and $LAST$ are indices of prelast and last taken points, and $TAKEN$ is number of points we taken, **or infinity** if there is no such polygon.

This dp can be calculated in natural way (go through all states and try to append every possible point).

We can check for convexity using cross-product.

Area is maintained by adding the area of appended triangle (can be find via cross-product). Note, that $2 \cdot AREA$ is allways an integer, so we can use only integers in calculation.

Overall, we need $\mathcal{O}(N^3 \cdot K)$ time, and we should do it N times since we brute force the BASE.

Total time complexity: $\mathcal{O}(N^4 \cdot K)$, but it passes the tests due to low constant.

There exists an alternative solution, with better complexity:

Let's look at all vector $(x_i, y_i) - (x_j, y_j)$. If we consider points as vertices and vectors as directed edges, then convex polygon is a cycle where polar angles of edges increase through cycle if we start from the downmost-leftmost point. Let's sort all vectors by polar angle in $\mathcal{O}(N^2 \cdot \log(N))$ time.

Now we can iterate over all possible downmost-leftmost points and try to find optimal subsequence of k vectors which will form a convex polygon with minimal area. Area can be maintained as sum of signed-areas of triangles on points $(0, 0), (x_i, y_i), (x_j, y_j)$. We will have similar $dp(last, taken)$ and we will relax it with edges in sorted order.

Total time complexity: $\mathcal{O}(N^3 \cdot K)$.

Problem Tutorial: "Effective Painting"

At the beginning, we will make the compression of the coordinates.

Let $f_{i,j}$ — is the maximum number of the visible colors that can be obtained using the segments, that are completely inside $[i, j]$.

To recalculate we will consider the segment which will be drawn first and is visible at the end (all invisible segments do not affect the answer and may be ignored, and at the final output we can print them first to get rid of them). If this segment is visible, we will iterate over all possible points k where it may be visible; then note that all other segments that are containing k are not visible, so we can recalculate answer for that k, using the values of $f_{i,k-1}$ and $f_{k+1,j}$.

Time complexity: $O(N^3)$.

Problem Tutorial: "Fibonacci Palindromes"

To solve this problem we can check manually for small size substrings if they are palindromes, then note that S_n without two last characters is a palindrome. Let call this palindrome P_n.

Then we can decompose the sequence of several S_i into the sequence of zeroes, ones and P_j and then check it directly using the fact that S_i is prefix S_j for any $i < j$.

Problem Tutorial: "Guess The Integer"

Game analysis:

Let's ignore the fact that we're asking about cyclic intervals for now and do a more general analysis. At first, the secret number is x from the set $S = \{0, 1, 2, ..., N-1\}$. Let's assume that we are asking for a set A. If the answer was true, we could replace S with A and be in the same situation going forward. If the answer was false, the number will be in the complement of A to S (let's call it set B, $A \cup B = S$) and all other answers will be false.

Important: the new situation resembles the starting situation!

Let $G(A, B)$ denote the situation where we know that either the Sphinx's number is in set A and subsequent answers follow the rules from the beginning of the game, or the number is in set B and all subsequent answers will be lies. From now on, we consider A and B to be disjoint sets. The initial situation is $G(S, 0)$. Let the next question we ask is the set $C = A_1 \cup B_1$, where $A = A_1 \cup A_2$ is a partition of A into two sets, $B = B_1 \cup B_2$ is a partition of B. Suppose the answer to set C is "yes". If the Sphinx is telling the truth, then the secret number is in the set A_1. If the Sphinx is lying, then the number is in $A_2 \cup B_2$. That is, we enter a situation $G(A_1, A_2 \cup B_2)$. If the answer of the jury is "no", with similar reasoning we arrive at situation $G(A_2, A_1 \cup B_1)$. If we reach a situation $G(A, B)$ in which the number of integers in $A \cup B$ is exactly one, then we know that this integer is the answer to the problem.

If we allow questions with arbitrary sets (and not necessarily with cyclic intervals), then the number of steps to guess the number depends only on the size of the sets A and B, not on the integers in the sets. Let $F(a, b)$ denote the number of questions that are needed in the worst case to guess the number if we start from $G(A, B)$, where a and b are the sizes of the sets A and B. We can find the value of $F()$ with dynamic programming.

The value $F(a, b)$ is equal to $1 + max\{F(a_1, a_2 + b_2), F(a_2, a_1 + b1)\}$ for some a_1, a_2, b_1, b_2 such that $a = a_1 + a_2$, $b = b_1 + b_2$. If we try all possible values to find a_1, a_2, b_1, b_2 that give the smallest value of $F(a, b)$ we can compute $F()$ in $O(N^4)$ steps, that is not enough, so we need another approach. Also, we will know the size of the sets A_1, A_2, B_1, B_2 with which to do the question from step $G(A, B)$. Then let's go back to cyclic intervals — if we can count $F()$, can we generate questions $C = A_1 \cup B_1$, of arbitrary size on A_1, B_1?

Let's see what are the possible transitions between states of G: $G(A, B) \to G(A_1, A_2 \cup B2)$ or $G(A, B) \to G(A_2, A_1 \cup B_1)$. Always the $A-$set is a subset of the A-set from the previous step. Let us maintain the A-set in $G(A, B)$ as an interval of numbers $[L_A, R_A)$. If we want to generate sets A_1,

B_1 of size a_1, b_1, we can start the interval $C = [L_C, R_C)$ from $L_C = R_A - a_1$ and keep stretching the interval "to the right" until we get the b_1 element of the set B. If we are get to N and we still don't have enough elements from B, we can take a cyclic interval, with $R_C < L_C$ (take elements from the beginning of $[0, N)$). With this choice of C, we will be asking for elements that are neither in A nor in B, knowing that we are in a $G(A, B)$ situation. But we know from the answers so far that the secret number is not among these elements, so adding them to the set C will not affect the answer.

To find the approach that can help to solve the task, let's look at the values of $F(i, j)$.

b \ a	0	1	2	3	4	5	6	7	8	9
0	1	0	3	5	5	6	6	6	6	7
1	0	2	4	5	5	6	6	6	6	7
2	1	3	4	5	5	6	6	6	6	7
3	2	3	4	5	5	6	6	6	6	7
4	2	3	4	5	5	6	6	6	6	7
5	3	4	4	5	5	6	6	6	6	7
6	3	4	4	5	5	6	6	6	6	7
7	3	4	5	5	5	6	6	6	6	7
8	3	4	5	5	5	6	6	6	6	7
9	4	4	5	5	6	6	6	6	7	7

The values in each row and column grow slowly. Let's compress the table by generating a function $H(a, k)$: the largest value of b such that $F(a, b) \le k$. If $a = u + v$, $b = b_u + b_v$, $F(a, b) \le 1 + max\{F(u, v + b_v), F(v, u + b_u)\}$, and equality holds for some u, v, b_u, b_v. Let $F(a, b) = k$. $F(u, v + b_v) \le k - 1, F(v, u + b_u) \le k - 1$.

And at another side, if we have $H(u, k - 1) = v + b_v$, $H(v, k - 1) = u + b_u$, we get $F(u + v, b_u + b_v) \le 1 + k - 1 = k$, $H(u + v, k) \le b_u + b_v = H(u, k - 1) + H(v, k - 1) - (u + v)$. That is, we get a recurrence $H(a, k) = max\{H(u, k - 1) + H(v, k - 1)\} - a$, for some $u, v : u + v = a, u \le c(v, k - 1), v \le c(u, k - 1)$.

Let's look at the values of H in a table:

k \ a	0	1	2	3	4	5	6	7	8	9
0	1	0	×	×	×	×	×	×	×	×
1	2	0	×	×	×	×	×	×	×	×
2	4	1	×	×	×	×	×	×	×	×
3	8	4	0	×	×	×	×	×	×	×
4	16	11	6	×	×	×	×	×	×	×
5	32	26	20	14	8	×	×	×	×	×
6	64	57	50	43	36	29	22	15	8	×
7	128	120	112	104	96	88	80	72	64	56
8	256	247	238	229	220	211	202	193	184	175
9	512	502	492	482	472	462	452	442	432	422

The cross means that $F(a, b) > k$ for every b. We can see that each row k forms an arithmetic progression that starts at 2^k and decreases by $k + 1$. When does the arithmetic progression stop? Let $H(m_k, k)$ be the last nonzero element of row k. From the recurrence for H, we see that there are $u \le v$ such that $m_k = u + v$, $v \le H(u, k - 1)$, and $u \le H(v, k - 1)$.

After trials with small numbers, we see that the numbers $H(s_{k-1}, k - 1)$ for which $s_{k-1} \le H(s_{k-1}, k - 1)$ are important: we put $u = v = s_{k-1}$ and see, that $m_k \ge 2s_{k-1}$. Equality holds for $3 \le k \le 9$.

Is equality always achieved?

If $v \leq s_{k-1}$, then $m_k = u + v \leq 2s_{k-1}$. If $v > s_{k-1}$, then $m_k = u + v \leq H(v, k-1) + v \leq H(s_{k-1}+1, k-1) + s_{k-1} + 1$ (since $H(w,k) + w$ is a strictly decreasing function of w — remember that $H(.,k)$ is an arithmetic progression) $\leq 2s_{k-1} + 1$ (since $s_{k-1} + 1 > s_{k-1}$ gives us $H(s_{k-1}+1, k-1) < s_{k-1} + 1$ by definition of s_{k-1}). That is, the only way to have $m_k > 2s_{k-1}$ is if $H(s_{k-1}+1, k-1) = s_{k-1}$, $u = s_{k-1}$, $v = s_{k-1} + 1$, and $m_k = 2s_{k-1} + 1$.

We recall that $H(a,k) = 2^k - a(k+1)$, for $a \leq m_k$. We can prove by induction exact formulas for s and m:

$s_k = 2^k/(k+2)$ (integer division — follows directly from the definition of s_k and the arithmetic progression of $H(a,k)$).

$m_k = 2 \cdot s_{k-1}$ if $s_{k-1} \neq H(s_{k-1}+1, k-1)$

$m_k = 2 \cdot s_{k-1} + 1$ if $s_{k-1} = H(s_{k-1}+1, k-1)$

For the constraints given in the problem, $m_k = 2 \cdot s_{k-1}$ is always true. This gives the formula for F(a, b): the smallest positive integer k such that $a \leq m_k$ and $a(k+1) + b \leq 2^k$.

So how to play?

By computing $k = F(a,b)$, we must compute a partition $a = a_1 + a_2$ such that $H(a_1, k-1) \geq a_2$ and $H(a_2, k-1) \geq a_1$. Then, we can find $b = b_1 + b_2$ such that $H(a_1, k-1) \geq a_2 + b_2$ and $H(a_2, k-1) \geq a_1 + b_1$.

From the analysis above it follows that we can take a_1 and a_2 close to $a/2$.

That is, $a_1 := a/2$ and $a2 := a - a_1$ works. If a is even, then $a_1 = a_2$ and $b_1 = b/2$ works. When a is odd, it is easy to see that $b_1 = (b - k + 1)/2$ works.

If we use the abovementioned way to calculate $F()$, we obtain the solution of the task.

Problem Tutorial: "Hypercycles"

Let's precalc polynomial hashes for all prefixes os S This allows us to find hash of each substring in $\mathcal{O}(1)$ Moreover, this allows us to find has of every cyclic shift of every substring in $\mathcal{O}(1)$, because it's a concatenation of two substrings. Now, let's fix L (we will answer queries separately). Let's find hashes of all substrings of S with length L and put create to maps

- `where[hash]` = any position of substring with that hash;
- `count[hash]` = frequency of substring with that hash.

Note, that every hyperccycle is a substring of S.

Let's run DFS on our substring of length L, from each substring we will go into its 1-cyclic-shift. If 1-cyclic-shift doesn't present as substring of S we will abort our process and mark this substring as invalid. Hypercircle is represented as cycle in our DFS-graph. We can find all cycles and sum up all frequencies whie running DFS. We should mark all visited substrings in order to not visit them twice. This way, we can find best hypercircle in $mathcal N$ time as we don we run dfs in graph with $V = N$, $E = N$. There will be extra $\log(N)$ in the complexity as we are using map to get data by hash.

Time complexity: $\mathcal{O}(N \cdot M \cdot \log(N))$.

Problem Tutorial: "Interactive Smiley Face"

Problem: Given a happy/sad smiley face on a large grid, determine its mood by asking questions "how many black pixels are in a rectangle".

- First, notice that the leftmost column of smiley face is exactly the smallest x such that the rectangle $(1,1) - - - (x, 10^9)$ contains at least one black pixel. We can find this x by binary search in 30 queries. Right, bottom and top edges of the face are determined similarly. Note that we know the size of the face at this point.

- Next, we have to locate the smile and determine its mood. Let us use the binary search to determine, say, left edge of the smile. It is equal to the maximal x such that the number of black pixels inside $(1,1)---(x,10^9)$ is not equal to the predicted value, that is, the number of black pixels in the rectangle if the face didn't have a smile at all.
- How to compute the predicted value for a given rectangle? We know that the face can be split into 41×41 squares that are colored according to the basic pattern. For each of these squares, it is easy to find how many black pixels does it contribute to the sum (this is the area of rectangle/square intersection).
- Find other edges of the smile in the same way. Now that we know the bounding box of the smile, compare the number of white pixels in the top and bottom row of the smile to determine the mood.

Problem Tutorial: "Join The Relay"

Firstly, we know, that in good mood all runner must run at least $D \cdot \sum_i T_i$ seconds, and in bad mood — $D \cdot \sum_i S_i$. Note that if $D \cdot N > L$ or $(D \cdot \sum_i S_i) > W$, then there is no solution. Another important case is when $D \cdot N = L$ then answer is exactly $D \cdot \sum_i T_i$, as we have no choice. Otherwise, let's try to distribute extra $L' := L - D \cdot N$ distance, such that extra time in bad mood would not exceed $W' := W - D \cdot \sum_i S_i$, and we are trying to minimize extra time in good mood. Suppose we have distributed distance in some way. We can divide all distances by L and get distribution of 1. We can easily construct distribution of L' from distribution of 1, due to linearity of both times (in bad/good mood), we don't change minimality then considering distribution of 1 instead of distribution of L'. We will try to distribute 1, in other words — assign $\{z_i \geq 0\}_{i=1}^{n}$, such that $1 = \sum_i z_i$. We should maintain $\sum_i z_i \cdot S_i \leq \frac{L'}{W'}$, and minimize $\sum z_i \cdot T_i$. Due to geometric properties, set of points $\{\sum_i z_i \cdot (S_i, T_i)\}$ is convex hull of vectors $\{(S_i, T_i)\}$. So we need to find point with minimal y-coordinate inside of this convex hull, such that it has x-coordinate less than or equal to $\frac{L'}{W'}$. It's possible that there is no such point, so there will be no solution to the problem. It can be shown, that such point must lie on an edge of convex hull if it exists. We can iterate through all edges of convex hull and find intersection of edge and vertical line ($x = \frac{L'}{W'}$). Total time complexity: $\mathcal{O}(N \cdot \log(N))$, which comes from the construction of convex hull.

Problem Tutorial: "King's Voyage"

Firstly, we can precalc distances between each pair of vertices in $\mathcal{O}(N^2)$ time in order to use them in $\mathcal{O}(1)$. We will solve the problem with the dynamic programming on subtrees. So we should hang tree by arbitrary node, for example 1. Let's define $dp(v)$ as maximal profit we can get if we have solve problem only in $Subtree(v)$. We will also define $tmp(v, pivot)$ as maximal profit we get from $Subtree(v)$ if King visits city $pivot$. For simplicity, we will not consider $-S$ in calculation of tmp. There are 2 cases in calculation of $dp(v)$.

- citizens of city v don't meet the King. In this case we should solve problem independelty for children of v. So we will assign $dp(v) := max_{i \in Children(v)}(dp(u))$.
- Otherwise we should iterate other possible cities such that citizens of city v could visit king there. We will relax dp as $dp(v) := max(dp(v), max_{pivot \in Subtree(v), dist(pivot,v) \leq D}(tmp(v,c) - S)$.

There are also 2 cases in calculation of $tmp(v, pivot)$.

- If $dist(v, pivot) > D$, then $tmp(v) := dp(v)$.
- Otherwise, $tmp(v) := F(v) \cdot T + \sum_{c \in Children(v)}(tmp(c, pivot))$.

We will calculate both dp, tmp recursively, so we can give them recursive definition. We will go with dfs from root through the tree and recalculate $dp(v)$, $tmp(v)$ with previously calculated values of children of v. Total complexity: $\mathcal{O}(N^2)$, because there are $mathcalO(N^2)$ states and transitions in our dp and tmp.

Problem Tutorial: "Lazy Teacher"

First consider the "standard" solution to the task. With it, the query is relatively fast, but it does not handle the update of submatrix. However, we gain fast queries, which can be used in future.

The idea is as follows — we start from the upper right corner of the matrix. If the number in the current cell is greater than the number we are looking for, we go left. If it is more little, let's go down. Thus, in a way, we form a sort of stepped diagonal with which we move to the cell in which the number we are looking for will potentially be.

With this type of search, from each cell we move either down or to the left, which means that at most we can go N cells down and M cells to the left. In terms of complexity, this search is $O(N + M)$. However, the update is $O(NM)$, so we need to find some improvements.

How can we keep $O(1)$ for the cell query while at the same time our update is not slower than $O(N+M)$?

It turns out that the way we traverse the matrix (down and left) allows this. For this we will need two arrays `int rowAdd[1000][1000]`, and `int colSub[1000][1000]`. On each update (say in cell (R, C)) with value val we will add val to cells `rowAdd[R][C]`, `rowAdd[R][C + 1]`, ..., `rowAdd[R][M]`. We will also add val to `colSub[R][C]`, `colSub[R + 1][C]`, ..., `colSub[N-1][C]`.

We will also need an additional variable tmp, in which we will keep track of how previous updates change the cell we are in at any given moment. We will know that when we reach a row we need to add to tmp `rowAdd[row][col]`. Since we can get out of the range of a query that we have already added to tmp by moving to the left, we will also use `colSub[row][col]` to decrease tmp. At any time (if we have changed tmp correctly) we will be able to find the value of the current cell as `a[row][col] + tmp` (which is a constant time operation).

The time complexity of the solution is $O(NM + Q(N + M))$.

Problem Tutorial: "Manipulation With Strings"

Let's create Aho-Corasick trie from strings S_i. For each node in trie we will have not only suffix-link, but terminal-suffix-links, that can be defined as longest suffix (not equal to the whole string) which is represented by terminal node in trie. Note, that note is terminal, if it corresponds to a string from the input. Terminal-suffix-link of node is either its suffix-link or terminal-suffix-link of suffix-link, so we can build them in the similar way we build suffix-links. Now let's define $ANS(s_i) :=$ {maximal sequence with condition from the statement, but largest string is s_i}. Answer to the whole problem is $max_{i=1}^{n} ANS(s_i)$. Let's go with bfs through our trie, each time we enter the node (v) we look at the $val(v) := ANS(terminal - suffix - link)$. Each time we enter terminal node t (correspondent to S_i), we set $ANS(S_i)$ as 1 + (max of $1 + val(v)$) over all v on path from root to t. So, all we need is to maintain $dp(u) :=$ {maximum of $val(v)$ over nodes on path from root to u}. This strategy works as for each string we iterate thourgh all possible optimal ways to take a substring from it. Optimal means that at each position is better to choose the longest substring ending in it. Total complexity: $mathcal O(sum of |S_i|)$.