Problem Tutorial: "The Seven Dwarves"

Sort the six ages: $a_0 \le a_1 \le \cdots \le a_5$. In the sorted order of all seven ages, either a_0 and a_1 are still consecutive, or a_1 and a_2, or both. Thus, one of $a_1 - a_0$ and $a_2 - a_1$ must be the correct difference d. Try both possibilities, and for each of them try all values $a_i \pm d$ as the missing element. Don't forget to discard ages that are not positive.

Problem Tutorial: "Four Steps Back"

We are given a permutation s and our task is to count permutations p such that $p^4 = s$.

If we split p into independent cycles, those have to remain independent in s as well, but the opposite is not true: a single cycle in p can correspond to multiple cycles in s. For example, if we have a cycle of length 5 in p, those five elements will also form a (different) cycle in s, but a cycle of length 4 in p produces four cycles of length 1 in s.

It's easy to verify that in general, a cycle of length ℓ in p will break into $x = gcd(\ell, 4)$ cycles, each of length ℓ/x, in s.

For our problem we now have to go backwards: look at the cycle lengths we see in s and count all ways in which they can be assigned to original cycles in some valid p.

If we see a cycle of an even length a, it's clear that it must have initially been a part of a cycle of length $4a$. And thus, there now have to be three other cycles of length a that were originally parts of that same cycle.

If we see a cycle of an odd length b, there are three possibilities: in the original permutation it could have been a lone cycle of length b, it could have been one half of a cycle of length $2b$, or one quarter of a cycle of length $4b$.

Clearly, different cycle lengths are completely independent: we can find the answer for each length separately, and multiply all those answers to get the final answer we seek. Thus, all that remains is the solution to the following problem: "Suppose the input permutation consists exactly of x cycles, each of length y. How many permutations have this permutation as their fourth power?"

We can use dynamic programming. Let dp[i] denote the number of ways in which a fixed set of i cycles (each of the currently fixed length y) can be obtained in the fourth power of a permutation. The value dp[i] can be calculated as follows:

- We pick any one fixed cycle C.
- We iterate over all possibilities for the number g of cycles in the input permutation that were originally on the same cycle as C. That is, we only try $g = 4$ if y is even, and we try $g \in \{1, 2, 4\}$ if y is odd.
- Given g, we can count the number of ways in which we can pick an ordered sequence of the remaining $g - 1$ cycles. This is just a simple product. E.g., if we need to choose three more cycles, there are $(i - 1)(i - 2)(i - 3)$ ordered ways to choose which three they are.
- We count the number of ways in which these three cycles can be rotated with respect to the original one. (see below for a more detailed explanation of what this means.) As we have y ways to rotate each of them, this gives us y^{g-1} different options to choose from.
- We add (dp[i-g], times the product of the previous two counts) to dp[i].

Here's an illustration of what's going on during the counting. Suppose we have $y = 3$ and we chose $g = 4$. Let the fixed cycle we started with be $1 \to 2 \to 7 \leftarrow$. As we chose $g = 4$, in the original permutation there must have been some cycle of the following form:

$$1 \to a_1 \to b_1 \to c_1 \to 2 \to a_2 \to b_2 \to c_2 \to 7 \to a_3 \to b_3 \to c_3 \leftarrow$$

Here, $a_1 \to a_2 \to a_3 \hookleftarrow$ is one of the other three cycles. We can choose any other 3-element cycle we have, for example, the cycle $5 \to 13 \to 9 \hookleftarrow$. Then, we can also choose which of its elements will be a_1.

Problem Tutorial: "Buzzzz"

Clearly, it is always optimal to position the spray on the room boundary.

Construct a convex hull and consider only mosquitos on its boundary. Then, the minimum angle is achieved in one of two cases:

- The spray is positioned in the corner.
- The spray is positioned such that two consecutive mosquitos on the convex hull are collinear with the spray.

This is non-obvious, so let's prove it.

Consider the line l going through on of the room sides and an arbitrary point P on it. The spray angle is so that all mosquitos are inside the angle, and some are on one of the rays enclosing the angle. In general case, there are only two of these points, one on each arc. Let's call these L_P and R_P.

There is an interval on which the pair (L_P, R_P) is constant – we can move along the line l and the pair of mosquitos that are "the widest"is the same. Let's call this interval $I(P)$. Its endpoints correspond to points Q that are collinear with two consecutive mosquiots on the convex hull.

We are looking for a minimum angle L_PQR_P on the interval $I(P)$. Consider a function $f(Q)$ matching a point Q on line l to the angle L_PQR_P. This function goes to 0 as Q tends to infinity (on both sides of l).

If L_PR_P is not parallel to l, it has two local maxima – at the point of tangency of circle passing through L_P and R_P – and one local minimum – the intersection of L_PR_P and l. The latter is not part of the interval $I(P)$ (unless all mosquitos are collinear). Because of this, the f attains minimum on the interval $I(P)$ in one of its endpoints. If L_PR_P is parallel to l, the two maxima coincide and there is no local minimum.

Hence, we need to find all intervals $I(P)$ along with the corresponding L_P and R_P. This can be done using rotating calipers or ternary search. To make the implementation a bit simpler, the solution below only considers the line $x = 0$ and the right endpoint of each interval – then rotates and/or flips the room 7 times to cover all cases.

Problem Tutorial: "Stack of Paper"

The constraints are small enough that we can try all possibilities for the division point i, and for each of them check whether the conditions are satisfied in linear time, requiring $\mathcal{O}(n^2)$ time in total.

We can also solve this in $\mathcal{O}(n)$ by checking whether the number of indices j such that $a_{(j+1) \bmod n} < a_j$ is at most one.

Problem Tutorial: "Similar Strings"

Regardless of n, it is possible to achieve $f(S, T) = 2n$. The solution is unique (up to switching `a` and `b`):

- `S = aa...aabb...bb`
- `T = aa...abab...bb`

Possible way of finding this pattern is writing a bruteforce solution and observing the results for small n. One simple way that runs fast enough for $n \leq 5$ is as follows:

- Construct all $\binom{2n}{n}$ options for S, similarly for T.

- Generate all substrings of S and find the shortest of them not present in T.

This needs $\mathcal{O}(2^{6n})$ time, which finishes in seconds for $n = 5$.

Problem Tutorial: "Rectangle in Rectangles"

Two possibilities for the largest rectangle in the union are the rectangles themselves. If they are disjoint and are not touching, those are the only two options.

Otherwise, there are two other candidates. In the first we consider the union of the x coordinates and intersection of y coordinates. This rectangle is clearly in the union and it cannot be enlarged in either direction. The second candidate is constructed the same way, just with the coordinates swapped. Time complexity is $\mathcal{O}(1)$.

Problem Tutorial: "Averaging the Average"

We can compute the following DP: Let $D[j][b_1][c_1][b_2][c_2][b_3][c_3] = true$ if it is possible to paritition the first j students in such a way that there are b_i students in the i-th group and the sum of their grades is c_i. Using this, we can calculate the answer easily by finding the state $D[n][b_1][c_1][b_2][c_2][b_3][c_3]$ for which $\frac{c_1+c_2+c_3}{b_1+b_2+b_3}$ is closest to k.
Initially $D[0][0][0][0][0][0][0] = true$, and the transitions are

- $D[j+1][b_1+1][c_1+a_j][b_2][c_2][b_3][c_3] \mid= D[j][b_1][c_1][b_2][c_2][b_3][c_3]$
- $D[j+1][b_1][c_1][b_2+1][c_2+a_j][b_3][c_3] \mid= D[j][b_1][c_1][b_2][c_2][b_3][c_3]$
- $D[j+1][b_1][c_1][b_2][c_2][b_3+1][c_3+a_j] \mid= D[j][b_1][c_1][b_2][c_2][b_3][c_3]$

If we compute the above naively, it will not fit into time and memory limit. However, since $b_1+b_2+b_3 = j$ and $c_1+c_2+c_3 = \sum_{i=1}^{j} a_i$ we can omit the last two dimensions of the DP. This reduces the time complexity to $\mathcal{O}(n^3(\sum a_i)^2)$, which is fast enough.

Problem Tutorial: "Countdown Letters Game"

We will preprocess the wordlist so that we can answer the queries faster.

A useful question to ask is the following one: "If this is the exact collection of letters I have to use, can I build a valid word? And if I can, what is the lexicographically smallest such word?"

We can compute the answers to these questions while reading the wordlist. We will store these answers in a hashmap.

A good representation of a collection of letters is simply a sorted string of those letters. For example, if we read the word `enlist`, the corresponding set of letters is the sorted string `eilnst`. Our hashmap will now get a new record in which the key `eilnst` points to the answer `enlist`.

Another record in the hashmap: `eemrt` → `meter`. Note that duplicate letters are preserved when sorting them.

Also note that sometimes words are anagrams of each other. E.g., the words `listen` and `silent` have the same collection of letters as `enlist`. As the wordlist is already given in lexicographic order, this is easy to handle: whenever we encounter a set of letters we've seen before, we just skip it (as the new word is lexicographically larger than the one we already had).

Given the hashmap we just computed, we can answer each query by trying all $2^9 - 1 = 511$ different subsets of the given letters. For each subset of letters, we find the best word that we can build using those exact letters, and we pick the best of all those options.

Problem Tutorial: "Game of Die"

Let $D[x][i]$ be 1 if the current player wins if the running total is x and the i-th face is the top face and 0 otherwise. Clearly, $D[x][i] = 0$ if $x \geq S$.

For a given x and i, the value $D[x][i]$ can be computed by trying all possible moves. If for any j, such that $j \neq i$ and $j + i \neq 7$, it holds $D[x + a_j][j] = 0$, then $D[x][i] = 1$, otherwise $D[x][i] = 0$. We can compute the DP table row by row, yielding an $\mathcal{O}(S)$ algorithm, which is too slow.

The key to a faster algorithm is realising that D is periodic. To see why this is true, note that $D[x][i]$ depends only on values $D[y][j]$ where $x < y \leq x + 10$ (where 10 is the maximum number written on a face). As soon as the configuration $(D[x + i][j])$ for $i \in \{1, 2, \ldots, 10\}, j \in \{1, 2, \ldots, 6\}$ repeats, the values of $D[x][i]$ start to repeat periodically.

A faster solution maps the tuple $(D[x + i][j])$ onto x. Once $(D[x + i][j]) = (D[y + i][j])$ for $x \neq y$, we can conclude that the period is $|x - y|$ and reduce S accordingly. This solution works in $\mathcal{O}(length\,of\,period)$.

It remains to show that the period cannot be very long. Naively, $(D[x+i][j])$ contains 60 bits of information, so one might think that the period can have length up to 2^{60}. Note, however, that $D[x][i] = D[x][7 - i]$, because the opposite faces have the same set of adjacent faces. Hence there are only 3 bits of information for each value of x, reducing the period length to 2^{30}.

Furthermore, observe the following fact: if $D[x][i] = 1$, then there exists $j \neq i, 7 - i$ such that $D[x][j] = 1$. This is because that in state (x, i) one can flip the die to some state $(y + a_k, k)$ that is losing. The k-th face has 4 adjacent faces, so some other $D[x][j]$ must be also winnable. As a result, $D[x][i]$ can be winnable for 0, 4 or 6 values of i — leaving only 5 possible options for the tuple $(D[x][i])$. As a result, there are at most 5^{10} possible tuples, which is a tighter limit on the period length.

Problem Tutorial: "Seating Pairs"

Consider the graph formed by guests and their friendships. Clearly, two people seated together must belong to the same connected component. Hence, if any connected component has an odd number of people, there is no solution. As we show below, in all other cases a valid solution exists.

We will construct one valid solution using depth-first search. The recursive function `dfs(v)` will do the traditional DFS traversal. Additionally, it will do one of two things:

- If the number of vertices in the DFS subtree rooted at v is even, the call to `dfs(v)` will pair up all these vertices and then it will return true.
- If the number of vertices in the DFS subtree rooted at v is odd, the call to `dfs(v)` will pair up all these vertices except for v and then it will return false.

In order to do this, we will implement `dfs(v)` as follows:

1. Start with an empty list of unmatched children.
2. For each neighbor w of v, if w is unvisited, call `dfs(w)`. Then, if the call returned false, append w to the list of unmatched children.
3. As long as there are at least two unmatched children, take any two and pair them up (with v being their common friend).
4. If there is one unmatched child left, pair it with v and return true.
5. Otherwise, return false.

Clearly, the above solution finds a valid seating whenever one can exist. Its running time is linear in the size of the input graph.

Problem Tutorial: "Cookies"

Assume that (i, j) is a solution and $i > n$. Then $(i - n, j)$ is also a solution with a smaller sum. Similarly with $j > n$. Hence, we can brute force all pairs (i, j) such that $0 \leq i, j \leq n$. The time complexity is $\mathcal{O}(n^2)$.

Problem Tutorial: "Tree Queries"

Let a_i be the number written in the i-th leaf. We can process the queries fast enough if we maintain b_j - the sum of a_is for which the binary representation of i has exactly j ones.

The answer to the query of the second type is then simply $\sum_{j=l}^{r-1} b_{30-j}$. This is computed in $\mathcal{O}(\log N)$ time.

How to process the queries of the first type? First, we split the addition of x on $[l, r)$ into an addition of x on $[0, r)$ and $-x$ on $[0, l)$.

How to process an addition on interval $[0, y)$? Let $y_1 y_2 y_3 \dots y_n$ be the binary representation of y. It can be seen that $[0, y)$ can be partitioned into a set of intervals of form $[y_1 y_2 \dots y_{i-1} 000 \dots 0, y_1 y_2 \dots y_{i-1} 100 \dots 0)$ for each i such that $y_i = 1$. For example, $[0, 11)$ is the union of intervals $[0, 8)$, $[8, 10)$ and $[10, 11)$.

These intervals are easy to work with - they consist of all numbers with a given prefix $y_1 y_2 \dots y_{i-1}$ and then all binary suffices of length $n - i + 1$. There are exactly $c_{ij} = \binom{n-i+1}{j-\sum_{k=1}^{i-1} y_k}$ numbers with exactly j ones in this interval.

The update is thus processed by adding $c_{ij} \cdot x$ to b_j. When the binomial coefficients are precomputed, each update takes $\mathcal{O}(\log^2 N)$ time.