A.Too Rich

Let s be the total value of your wallet. To pay p dollars by as many coins as possible, you are equivalently trying to keep $s-p$ dollars by as less coins as possible. We consider this problem in that way as follows.

If $c_{50}=c_{500}=0$, we can simply the greedy strategy that use larger denomination first. But greedy may not work in general. An example is that you want to keep 60 dollars, and you have $20,20,20,50$ in your wallet.

An observation is that $2\times 50=100$, which are good denominations that can be handled in greedy strategy. So we can enumerate whether to keep a "50" or not, and group the remaining "50" two by two, treat them as "100".

This trick also works for "500". Thus after enumeration of "50" and "500", we can use greedy strategy to solve it.

B.Count $a\times b$

First we look into $f(m)$, for any a we find that if $a\times b=0 \bmod m$ then $(m/gcd(a,m))|b$ and for any $b=k(m/gcd(a,m))$ has $a\times b=0 \bmod m$. So, $a\times b=0 \bmod m$ iff $(m/gcd(a,m))|b$. Now, $f(m)=m^2-\sum_{a=1}^{m} m/(m/gcd(a,m))=m^2-\sum_{a=1}^{m} gcd(a,m)$. More, let $h(m)=\sum_{a=1}^{m} gcd(a,m)$ and $\phi(d)=$ Euler's phi function, we have $h(m)=\sum_{d|m} d\phi(m/d)$. Because the number of (a,m,d) such that $gcd(a,m)==d$ equals to the number of (a,m,d) such that $gcd(a/d,m/d)==1$.

If we enumerate all factors of n as m and then enumerate all factor of m to calculate $h(m)$, because integers between 1 and 10^9 have at most 1344 factors, the complexity will be $O(1344^2)$, and it may be too slow for 20000 querys.

We can find $h(m)$ is a multiplicative function because it is a term like $a(x)=\sum_{d|x} b(d)$ which b is a multiplicative function. So, we only need to solve $h(p^k)$ where p is a prime.

$$h(p^k)=\sum_{d|p^k} d\phi(p^k/d)=\sum_{i=0}^{k} p^i\phi(p^{k-i})=p^k\phi(1)+\sum_{i=0}^{k-1} p^i p^{k-i-1}(p-1)=p^k+kp^{k-1}(p-1)=p^{k-1}(k(p-1)+p).$$

Finally, we enumerate all factors of n as m and calculate all $m^2-h(m)$ in $O(1344)$ operations. You probably find that two parts of $g(n)$ are both multiplicative function too. The bottle neck is to factorize n in the simple $O(\#\text{ of primes below}\sqrt{n})$ way, so the running time won't improve that much if you solve $g(n)$ in $O(log(n))$ time.

C.Play a game

First we introduce one $O(n^2+\sum|a_i|+n\sqrt{\sum|a_i|}+q)$ solution and then improve it to $O(n\lceil n/64\rceil+\sum|a_i|+n\sqrt{\sum|a_i|}+q)$.

Let $T_{ij}=t_it_{i+1}\dots t_j$, $dp(j,i)=\begin{cases}1, \text{if first player wins when } S=T_{i(i+j-1)}\\ 0, \text{otherwise}\end{cases}$ and $isA(j,i)=\begin{cases}1, \text{if } T_{i(i+j-1)}\in A\\ 0, \text{otherwise.}\end{cases}$

We can use AhoCorasick algorithm to find all matching points to obtain isA. Note that since there are at most $O(\sqrt{\sum|a_i|})$ different lengths, there are at most $O(\sqrt{\sum|a_i|})$ matching points for each prefix of T and it leads to $O(n\sqrt{\sum|a_i|})$ matching points in total. This step takes $O(\sum|a_i|+n\sqrt{\sum|a_i|})$ operations.

We can find $dp(j,i)=(isA(i,j)==0)$ and $(dp(j-1,i)==0$ or $dp(j-1,i+1)==0)$. This step takes $O(n^2)$ operations which is too slow.

To improve, we find that it is possible to use bitwise or and bitwise and operation in C++ to speedup. Let $dp^c(j,i)=\sum_{k=0}^{63} 2^k dp(j,i\times 64+k)$, $isA^c(j,i)=\sum_{k=0}^{63} 2^k isA(j,i\times 64+k)$.

The way to get isA^c is almost the same. We use $O(n\lceil n/64\rceil)$ operation to initialize isA^c to all zero and then use $O(n\sqrt{\sum|a_i|})$ operations to update isA^c.

Now, $dp^c(j,i)=(\sim isA^c(j,i))\ \&\ (dp^c(j-1,i)\mid \lfloor dp^c(j-1,i)/2\rfloor \mid (dp^c(j-1,i+1)\&1)\times 2^{63})$. (& means bitwise and, | means bitwise or and $\sim$ means bitwise not.)

For space problem, we can find that dp^c only needs the values in $isA^c(j)$ and $dp^c(j-1)$, so we only need to keep $O(n\sqrt{\sum|a_i|}+n)$ varialbe at a time. We can scan all queries at the beginning and anwser them while doing dynamic programing.

D.Pipes selection

Let $s_i = \sum_{k=1}^{i} a_k$. Discover that the number of possible (l, r) for j equals to the number of ways choose (x, y) such that $x \in \{0, s_1, s_2, \ldots, s_{n-1}, s_n = s\}$, $y \in \{s, s - s_1, s - s_2, \ldots, s - s_{n-1}, s - s_n = 0\}$ and $x + y == s - j$.

We can use FFT or NTT to caculate the number of possible (l, r) for j from 1 to s in $O(slogs)$ time. But this won't tell us the position of (l, r).

One way to find the k-th smallest (l, r) is to divide s into $\lceil s/X \rceil$ same length subsections. We can use FFT or NTT to caculate the number of segments for all length j from 1 to s which start from $[X \times t, X \times t + X)$ and end with $[X \times t + X, s]$ in $O(slogs)$ time. Next, enumerate segments in $[X \times t, X \times t + X)$ and upate the counter. After these two process, we know the number of segments for all length j from 1 to s which start from $[X \times t, X \times t + X)$ for all t from 0 to $\lceil s/X \rceil - 1$. This step takes $O(\lceil s/X \rceil (X^2 + slogs))$.

And then use binary search or $O(\lceil s/X \rceil)$ search to find which subsection k-th (l, r) belongs to for each j and find how many (l, r) in the subsection are smaller than it. This step takes $O(\lceil s/X \rceil s)$.

For each j from 1 to s, enumerate the position (as st) in the subsection it belongs to as start points, we can $O(1)$ find whether st and $st + j$ are both some prefixsum of $\{a_i\}$ or not, and then find which two prefixsum are the k-th smallest (l, r). This step takes $O(sX)$.

Theoretically, we can choose $X = \sqrt{slogs}$ lead to $O(s\sqrt{slogs})$. Because FFT has a large constant, it would be better we choose X greater than $\sqrt{slogs}$. We may pass the testdata by choosing 1024 or 2048 as X.

There is a way to reduce the total number of operations in FFT. You don't need to do this to get accepted (theoretically). We can divide the polynomials into all length X so we can pre-convert these polynomials to the other domain. $3\frac{s}{X}FFT(s)$ reduce to $2\frac{s}{X}FFT(2X) + (\frac{s}{X})^2 FFT(2X)$.

E.Rebuild

Let $d_i = \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2}$ for i from 1 to $n - 1$, and $d_0 = d_n = \sqrt{(x_n - x_0)^2 + (y_n - y_0)^2}$.

If $2 \nmid n$, then $\sum_{i=1}^{n} d_i = \sum_{i=1}^{n-1} (r_i + r_{i+1}) + (r_n + r_1) = 2\sum_{i=1}^{n} r_i$. We get $r_1 = \frac{1}{2}\sum_{i=1}^{n} d_i - \sum_{i=1}^{(n-1)/2} d_{2i}$, and then we can calculate all r_i. Note that we need to check if $0 \le r_i$ for all i. According to description, d_i are integers so we can avoid the round-off error to judge whether $0 \le r_i$ or not.

If $2|n$, then $r_2 = d_1 - r_1, r_3 = d_2 - d_1 + r_1, r_4 = d_3 - d_2 + d_1 - r_1, \ldots$. That is, for i from 1 to n, r_i can be represented by $a_i r_1 + b_i$. So, what we want is minimize $\sum_{i=1}^{n} (a_i r_1 + b_i)^2$ which is a polynomial of degree 2, let's let P be that polynomial. Thus, we can use calculus to obtain the minimum. $\frac{dP}{dr_1} = \sum_{i=1}^{n} 2a_i(a_i r_1 + b_i) = 2(nr_1 + \sum_{i=1}^{n} a_i b_i)$. $P' = 0$ at $r_1 = -\frac{1}{n}\sum_{i=1}^{n} a_i b_i$.

One more thing is that r_1 is bounded by $0 \le a_i r_1 + b_i \le \min(d_{i-1}, d_i)$ for i from 1 to n. Again, according to description, d_i are integers so we can avoid the round-off error to judge whether there exists a r_1 in the range or not. To minimize, we must choose the r_1 which is closest to $-\frac{1}{n}\sum_{i=1}^{n} a_i b_i$ and is in the range. There will not be multiple anwsers because $P'' = n$ and if $x \ne y$ and $P(x) = P(y)$ then $P(x) = \frac{P(x) + P(y)}{2} > P(\frac{x + y}{2})$.

F.Almost Sorted Array

Since non-increasing and non-decreasing cases are symmetric, here we only consider the non-decreasing case. You can reverse the array to check the symmetric case.

If the original array is already non-decreasing, remove any element will make it sorted. Otherwise, there must exist a position p such that $a_p > a_{p+1}$. We can simply try to remove a_p or a_{p+1}, and check whether the remaining array is sorted or not.

G.Dancing Stars on Me

The key observation is that the answer always be "NO" if $n \neq 4$. Thus you can simply enumerate all possible permutations, and check whether it is a regular polygon or not.

H.Partial Tree

We want to make a valid degree sequence for tree with maximum coolness. Let $1 \leq d_1 \leq d_2 \leq \dots d_n$ be integers. There exists a tree with degrees $d_1, d_2, \dots, d_n$ if and only if $d_1 + d_2 + \cdots d_n = 2(n-1)$. This can be proved by mathematical induction easily.

Thus what should we do is distribute $2(n-1)$ degrees to n nodes. Since each node must has non-zero degree, we can let each node has 1 degree first, then distribute the remaining $n-2$ degrees. The distribution process just like a knapsack problem, which can be solved by a classical $O(n^2)$ dynamic programming.

I.Chess Puzzle

We divide the algorithm to severel steps:

1.Assume that we'd got some scores, and lose 1 point for every **same color** adjacent chesses. The goal becomes to minimize the scores that we lose.

2.An observation is that this graph is always bipartite, let us denote the original graph $G = (V, E)$ as a bipartite graph $G = (X, Y, E)$. We can invert the color of chess that belongs to set Y.

3.After inversion, the problem becomes a simple s-t minimum cut problem. Since most of the edges have capacity 1, we can solve maximum flow problem using Dinic's algorithm fastly, although the number of nodes may be as large as 10000.

4.The final step is to find the lexicographical smallest answer. We can enumerate the color of each chess greedily. The complexity of this step would not be larger than traverse the residual network.

J.Chip Factory

The basic idea is to enumerate all the possible $s_i + s_j$ and choose a s_k greedily from higher bit to lower bit to maximize the xor value. We can maintain all the possible s_k on a binary tree from higher to lower bit, which can find a best s_k after enumerate $s_i + s_j$ in a constant time. The algorithm solves in $\mathcal{O}(n^2)$.

K.Maximum Spanning Forest

Since n is small, the first thing we do is to discretize the operations so that all the nodes can be divided into $\mathcal{O}(n^2)$ rectangles. So we can calculate "the maximum spanning forest(MSF) in each rectangle" and "the MSF between rectangles" individually. The hardest part of implementation is to maintain MSF between rectangles, this can be easily solved by Kruskal or Prim's algorithm.

L.House Building

We can calculate the surface area of top, left, front side separated. If a cube has no other cube adjacent to its left side, it should be counted for left side surface area. Moreover, since the cubes are stacked up from bottom to top, we can simply check the height of one square and its left square, and calculate the number of cubes that should be counted. We can also calculated other sides by the same way.

M.Security Corporation

Since the maximum degree in the graph is always smaller than 5, if we arrange all nodes in sweep line order, the

answer would always be smaller or equal to 3.

The algorithm is quite simple: check if the graph is 1-colored or 2-colored(bipartite), otherwise the graph should be 3-colored.