Ping-pong (pingpong)

Author: Zsolt Németh

Developer: Stefan Dascalescu

Solution

The main idea of the solution is to greedily assign the scores to each player in such a way that the first player wins. We can fix the number of sets we want to use and the idea is to give the first $i - 3$ sets to the second player and the last 3 sets to the winner.

Once we distributed the wins, we can check if we can distribute in each set at most 10 points in order to fulfill the constraints asked. For further details, refer to the sample implementation below.

```python
def solve(A, B):
    #3:
    if A==33 and B<=30:
        print(11, B//3)
        B -= B//3
        print(11, B//2)
        B -= B//2
        print(11, B)
        return
    #4:
    if 33<=A<=43 and 11<=B<=41:
        B -= 11
        print(A-33, 11)
        print(11, B//3)
        B -= B//3
        print(11, B//2)
        B -= B//2
        print(11, B)
        return
    #5:
    if 33<=A<=53 and 22<=B<=52:
        A -= 33
        B -= 22
        print(A//2, 11)
        A -= A//2
        print(A,11)
        print(11, B//3)
        B -= B//3
        print(11, B//2)
        B -= B//2
        print(11, B)
        return
    print("-1 -1")
    return
```

Baby Bob's Bracket Sequence (brackets)

Author: Áron Noszály

Developer: Áron Noszály

Solution

Let's revisit the classical algorithm of deciding whether a given string of brackets is a valid bracket sequence or not. In that algorithm, we maintain a counter and loop through the elements of the string from left to right. If the current element of the string is an opening bracket we increase the counter by one, if it's a closing bracket we decrease the counter by one. The string is valid if and only if the counter is 0 at the end, and at no point it was negative.

From this as inspiration, we can deduce an $O(N \sum A_i)$ dynamic programming algorithm to solve the problem. Let $f(i, j)$ be `true` if it's possible to have the counter at j, if we only run the algorithm on C's first $A_1 + A_2 + \ldots + A_i$ characters. The base cases are the following: $f(0, 0) =$ `true` and $f(0, j) =$ `false` for any $j \neq 0$. The transition is:

$$f(i, j) = f(i - 1, j - A_i) \vee f(i - 1, j + A_i)$$

(where $\vee$ is the binary or operator)

We have a solution if $f(N, 0)$ is `true`. You can get a possible construction by the usual means (storing a parent for each state along the f values).

Binary Chess (binarychess)

Author: Áron Noszály

Developer: Bernard Ibrahimcha

Solution

Let's consider a graph where the vertices are the occupied cells and there's an edge between two vertices if the respective cells are in the same row, or the same column or the same diagonal. Now if we look at a connected component in this graph, we can see that there can't be both a rook and a bishop in it. Why? If there's both a rook and a bishop, then there should also be a rook and a bishop that are connected by an edge with no other occupied cells between them (you can prove this by contradiction). But that couldn't happen because in that case either the rook attacks the bishop (if the edge is here because of a row or column), or the bishop attacks the rook (if the edge is here because of a diagonal).

Thus in every connected component we can decide if we place only rooks or only bishops. So the answer is 2^{cc} where cc is the number of connected components.

It's easy to find the connected components naively in $O(N^2)$, but we can do better. Notice there's only $O(N)$ that actually matter. Let's sort the occupied cells by row indices, column indices, sum of row and column indices and difference of row and column indices. Then in each of these orders, we only need to add edges between some adjacent elements. The connected components themselves can be found with a simple depth first search or union-find data structure.

The overall time complexity is dominated by the sorting, so it is $O(N \log N)$.

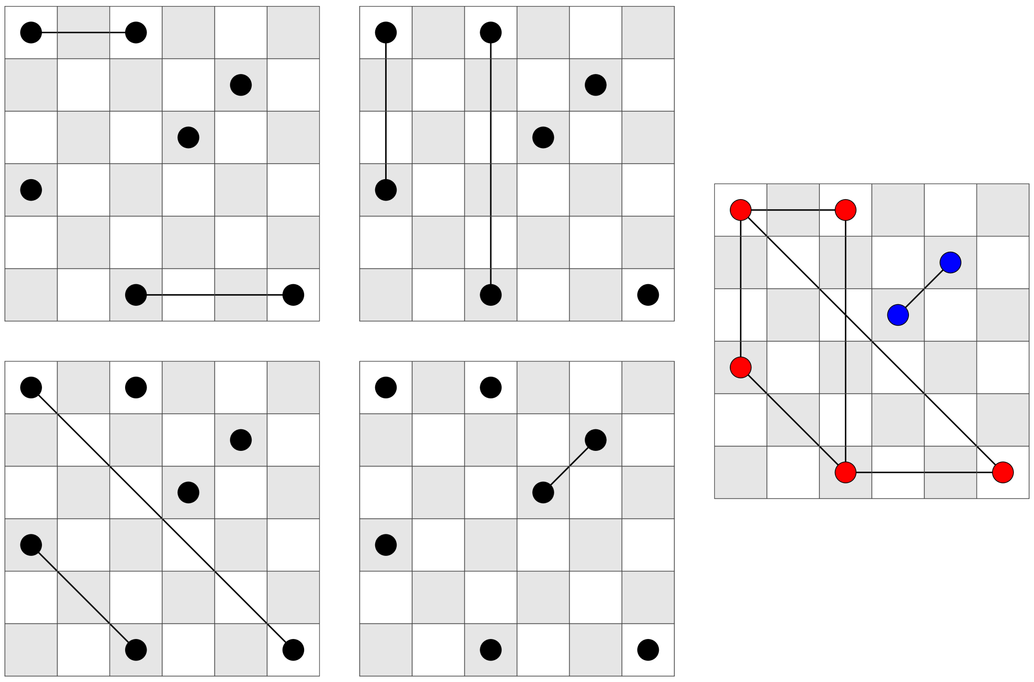

Figure 1: On the left you can see the edges from the 4 different orderings, and on the right the connected components.

Destroy the village (barbarian)

Author: Tommaso Dossi

Developer: Tommaso Dossi

Solution

Subtask 2: Since N is small, it suffices to simulate the process for each starting point. Every time we destroy a house, we mark it as destroyed and look for the closest unmarked house. This takes $O(N^3)$ time.

Subtask 3: We can again simulate the process, but we need to find the closest house in $O(1)$. Notice that the destroyed houses always form an interval, with the one destroyed last being one of the extremes of this interval. Hence the closest undestroyed house is one of the two bordering the interval.

Subtask 4: To simplify the explanation, we will now suppose that the house are arranged in a line going from left to right, with the shore being on the left. Let's define a *direction change* in our simulation as the moment the last destroyed house changes from being the leftmost to being the rightmost, or vice versa.
Let's suppose that the last house we destroyed is house l, which also is the leftmost destroyed house, the rightmost being house r and that the next house to be destroyed is $r+1$. Hence $D_{r+1} - D_l < D_l - D_{l-1}$.
After another *direction change*, the leftmost and rightmost destroyed houses will be $l' < l$ and $r' > r$. We have that:

$$D_{r'} - D_{l'} \geq D_{r+1} + D_{l-1} = D_{r+1} - D_l + D_l - D_{l-1} > 2(D_{r+1} - D_l) > 2(D_r - D_l)$$

Every two *direction changes*, the width of the interval is at least doubled. Hence, there are at most $O(\log(D_{N-1} - D_0))$ *direction changes.*
Now, given our interval $[l, r]$ of destroyed houses, the last being house l, we need to find when the next *direction change* happens, i.e. how many houses to the left of l we destroy before destroying house $r+1$. We define P_i as the smallest j such that $D_j - D_i \geq D_i - D_{i-1}$, for each $0 < i < N$. If no such j exists for some i, $P_i := i$.
We notice that the next destroyed point will be $l-1$ if and only if $P_l > r$, since

$$D_l - D_{l-1} \leq D_{r+1} - D_l \iff P_l \geq r+1$$

We can find the maximum $i < l$ such that $P_i \leq r+1$ in $O(\log N)$ using a binary search and a sparse table. Note that we also have to handle the *direction change*, from right to left, but it is almost identical to what we have shown. This solution handles each *direction change* in $O(\log N)$ and for each starting point there are at most $O(\log(D_{N-1} - D_0))$ direction changes. The complexity is $O(N \log N \log(D_{N-1} - D_0))$.

Subtask 5: Let's call $p(i)$ the first house where we change direction if we start from house i. Let's assume without loss of generality that $p(i) < i$, we know by definition of $p(i)$ that $D_{p(i)} - D_{p(i)-1} > D_{i+1} - D_{p(i)}$, then for every j such that $D_{p(i)} \leq D_j \leq D_i$ we have $D_j - D_{p(i)-1} > D_{i+1} - D_j$. This means that the path starting from $p(i)$ will move to the right at least until it surpasses i, at this point both paths, starting from i and starting from $p(i)$ will have destroyed the interval $[p(i), i+1]$ and the barbarian will be at position D_{i+1}, since the state is the same both paths will continue in the same way, thus the path starting from house $p(i)$ is a suffix of the path starting from house i. If we can compute $p(i)$ we can solve the problem using a recursive function with memorization.

We first compute $p(i)$ only for is such that $p(i) < i$, we iterate from 0 to $n-1$ and we keep a stack s of candidates for $p(i)$. If the barbarian starting at i goes right we do nothing, if he goes left we know he can only change direction on an element of s so he will reach x, the top of s. After reaching x the barbarian will go left if and only if $D_x - D_{x-1} \leq D_{i+1} - D_x$. If the barbarian goes right then $p(i) = x$, if the barbarian goes left we pop x from s, x won't be a candidate for any $p(j)$ with $j > i$ because $D_x - D_{x-1} \leq D_{i+1} - D_x \leq D_{j+1} - D_x$, then we can check the next element on top of s. After processing element i we push i onto s and we proceed to $i+1$. Similarly we can compute $p(i)$ for is such that $p(i) > i$.

The time complexity of the algorithm is linear.

Periodic Words (periodicwords)

Author: Péter Gyimesi

Developer: Alexandru Lorintz

Solution

The key ingredient is polynomial hashing. We assign a polynomial to the string $A = \overline{a_0 a_1 a_2 \ldots a_{n-1}}$:

$$H(A) = a_0 + a_1 b + a_2 b^2 + \ldots + a_{n-1} b^{n-1}$$

Now if we assign different random integers to characters, and the base b this $H(A)$ become an integer for any string A. Since it can get very large, we rather compute it modulo a large prime p. It is well known, that if A and B are different strings then $H(A) = H(B)$ with only approximately probability $\frac{1}{p}$. For making debugging easier we usually assign character ch the integer $ch -$ `'a'` $+1$, assign b the integer 29 or 31, and compute everything modulo $10^9 + 7$ or $10^9 + 9$.

We can precompute the powers of b modulo p and the prefix hash vector

$$h = [0, H(A[0 \ldots 0]), H(A[0 \ldots 1]), H(A[0 \ldots 2]), \ldots, H(A[0 \ldots n-1])]$$

in $O(N)$ time (we store everything modulo p). Now the hash value of a substring $H(A[l \ldots r]) \equiv H(A[0 \ldots r]) - H(A[0 \ldots l-1]) \equiv h[r+1] - h[l] \pmod p$ can be computed in $O(1)$ time.

We have that $A[a \ldots b] = A[c \ldots d]$ $(a \leq c)$ with high probability if

$$H(A[a \ldots b]) \cdot b^{c-a} \equiv H(A[c \ldots d]) \mod p$$

Finally for query l_i, r_i we need the divisors of $r_i - l_i + 1$. It is better to precompute it for every possible length with the Sieve of Eratosthenes in $O(N \log(N))$ time:

```
vector<int> divs[100001];
for (int i = 1; i <= N/2; ++i) {
  for (int j = 2 * i; j <= N; j += i) {
    divs[j].emplace_back(i);
  }
}
```

Now we go through all divisors $d \mid r_i - l_i + 1$. The last trick is not to compare all $\frac{r_i - l_i + 1}{d}$ substrings, just two: $A[r_i \ldots (l_i - d)]$ and $A[(r_i + d) \ldots l_i]$.

Long Chain (`longchain`)

Author: Péter Gyimesi

Developer: Alexandru Lorintz

Solution

The first thing we should notice is that if we can split the edges in such a way that the length of the shortest chain is equal to l, then we can definitely split the edges in such a way so that the length of the shortest chain is equal to $l - 1$. So, what this means is that we can find the biggest value of l such that we can split the edges so that the length of the shortest chain is l using binary search.

Now, for a fixed value of l we want to know if we can split the edges in such a way so that each chain has a length of at least l.

For a given node (call it u) that has k children we just need to know the length of the chain going up from each children of u. Note the length of the chain going up from the i^{th} child of u with v_i. We just need to pair those chains in such a way so that every chain has a length of at least l and the length of the chain going up from u (if any) is as big as possible. We will have two cases, depending on the parity of k. For both of these cases, we will consider v to be sorted in increasing order.

k **is odd:** In this case, we will make $\frac{k-1}{2}$ pairs and propagate one chain up. So, we will have to pair v_0 with v_{k-1}, v_1 with v_{k-2} and so on, until we reach an index i that we will want to skip, in order to propagate the i^{th} chain to the father of u. We will want i to be as big as possible. We can see that if we can choose to exclude the i^{th} chain and still be able to pair all the other values in a way so that all the pairs have a sum of at least l, than we can exclude the $i - 1^{th}$ chain and still be able to pair up the other chains. Knowing this, we can just binary search the rightmost index i such that we can pair up all the other values, so that each of the $\frac{k-1}{2}$ chains have a length of at least l.

k **is even:** In order to maximize the length of the chain we will want to propagate, we will try to make only $\frac{k}{2} - 1$ pairs. We can only do that if we already have a chain whose length is at least l. We can choose not to pair this chain with any other and solve the problem only considering the other $k - 1$ chains. (in this case, we just need to solve the "k is odd" case). If we cannot find such a chain or we cannot pair the other $k - 1$ values, we will just try to make $\frac{k}{2}$ chains, pairing v_0 with v_{k-1}, v_1 with v_{k-2}.

The time complexity for the solution above is $O(N \cdot log^2 N)$.

Area Under Path (areaunderpath)

Author: Péter Csorba

Developer: Bence Deák

Solution

Subtask 2: As we have at most $\binom{20}{10} = 184\,756$ different paths: *brute force* is enough here.

Subtask 3: Let $f(N, M, P, R)$ denote the solution (the number of different paths with area $\equiv R \pmod P$). Considering the last step we get that

$$f(N, M, P, R) = f(N, M-1, P, R) + f(N-1, M, P, (R-M)\%P)$$

if $N, M \geq 1$. Moreover $f(0, M, P, 0) = f(M, 0, P, 0) = 1$, and if $R \neq 0$ $f(0, M, P, R) = f(M, 0, P, R) = 0$. So this subtask can be done by *dynamic programming* in $O(N \cdot M \cdot P)$ time (make sure that you store every number modulo $10^9 + 7$).

Subtask 4: Let $N = n \cdot P$, and $M = m \cdot P$. There is a formula (see proof below) for the solution this case! If $R = 0$:

$$f(N, M, P, 0) = \frac{\binom{N+M}{N} - \binom{n+m}{n}}{P} + \binom{n+m}{n}$$

If $R \neq 0$:

$$f(N, M, P, R) = \frac{\binom{N+M}{N} - \binom{n+m}{n}}{P}$$

As $\binom{a}{b} = \frac{a!}{b! \cdot (a-b)!}$, and we want the answer modulo $10^9 + 7$: we need to pre-compute factorials modulo $10^9 + 7$ and compute modular multiplicative inverses of those we need ($O(N + M)$).

Subtask 5: Let $N = n \cdot P + R_N$, and $M = m \cdot P + R_M$. Using the previous tasks and the proof: We have

$$ALL = \binom{N+M}{N}$$

different path. And the BAD ones are those paths going through $A = (n \cdot P, m \cdot P)$, but getting there with $(0, P)$ or $(P, 0)$ steps. From A to (N, M) we get by an $R_N \times R_M$ rectangle:

$$BAD = \binom{n+m}{n} \cdot \binom{R_N + R_M}{R_N}.$$

Now $ALL - BAD$ contributes evenly to all remainder classes. Any BAD path until point A have area divisible by P (even by P^2). The area under a BAD path modulo P only depends on the $A \to (N, M)$ part. This means:

$$f(N, M, P, R) = \frac{ALL - BAD}{P} + \binom{n+m}{n} \cdot f(R_N, R_M, P, R)$$

which can be computed in $O(N + M + P^3)$ as described in the previous tasks.

Proof: The main idea is to partition many paths to groups of size P such that inside a group they have different area modulo P. After that we can deal with the rest.

A $(0,0) \to (N,M)$ path can be encoded to a string of length $N+M$, for each step to the right we write 'r', for each step up we write 'u'. This string consist the character 'r' N times, and 'u' M times: this is why we have $\binom{N+M}{N}$ different paths.

Let pick a path from $(0,0) \to (N,M)$. We look at the corresponding 'u,r' string, and assume that we have P consecutive letters such that they are not the same, i.e. this substring of length P is not $uu\dots u$ nor $rr\dots r$. By cyclically shifting this block we get P different[1] $(0,0) \to (N,M)$ path.

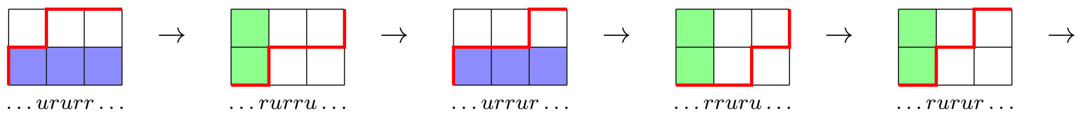

Figure 1: Example for cyclic shifts for $P = 5$.

Now we look at the length P cyclically shifted part. Since cyclic shift does not change the number of characters, this part is a path from (x,y) to $(x+a, y+b)$, where $a+b = P$, $a, b \neq 0$. When u in the front is shifted to the end of this substring, the area is decreased by a (see blue rectangles). When r in the front is shifted to the end of this substring, the area is increased by b (see green rectangles). Since $-a \equiv b \pmod{P}$ each cyclic shift changes the area by the same amount modulo P: the P cyclically shifted paths have different area modulo P (and hence they are different).

Going back to Subtask 4: Now $N = n \cdot P$, and $M = m \cdot P$. We cut the string of length $N+M$ into $n+m$ substring of length P. We pick the first (from left) which contains both u and v. By cyclically shifting this substring we get all area classes modulo P. We can not do this when all substring contains only u or v. By replacing these length P blocks by a single letter of it we get a new path from $(0,0)$ to (n,m). So we have $\binom{n+m}{n}$ BAD paths, but the are under them is divisible by P. We have $ALL = \binom{N+M}{N}$ paths, this gives the answer for $R! = 0$: $\frac{ALL-BAD}{P}$ and for $R = 0$: $\frac{ALL-BAD}{P} + BAD$.

[1] We need here that P is prime!