TRIK

We know the ball is initially under cup 1. At any moment, depending on the current location of the ball and the move used, we can determine its next location (for example with if statements).

NATRIJ

For both given times, we'll calculate the number of seconds elapsed since the current day started. For the starting time that number is H·60·60 + M·60 + S. For the ending time we use the same expression, except if the ending time is earlier than the starting time. In that case the ending time is actually the next day, so we need to add 24·60·60 = 86400 seconds.

Subtracting the two numbers we get the number of seconds between the two times. This needs to be converted into hours, minutes and seconds.

Also note the case when the given times are equal. The task description says that 24 hours should be output, not 0.

TENIS

The rules are seemingly simple, but checking if a result could have been achieved according to the results is less so. If any of the following applies, the result is not legal:

- Players won the same number of games in a set
- A player won a set with less than 6 games
- A player won a set by a difference of only one game (except 7:6 in the first two sets), or more than two games if he won 7
- The player won more than 7 games in the first or second set
- The result is 1:0, 1:1, 3:0 (sets), or more than 3 sets have been played
- Federer lost a set

An alternative solution is to generate all legal results according to the rules and check for each match if it's among the valid results.

LIGA

Let the five numbers be A, B, C, D and E. We know that A=B+C+D and E=3·B+C. Additionally, all numbers are integers, at least 0, and A is at most 100.

The sample data shows that cases where one or two numbers are missing can be solved. It is less obvious that cases where three numbers are missing can be solved, and even some cases where four are missing (for example, `0 ? ? ? ?` or `? ? ? ? 300`).

Note first that A and D can't both be unknown, except in the special case when A=B+C=100 and D=0. If A and D were unknown, it would be possible to increase or decrease both numbers by one, and the solution would not be unique.

With that in mind, notice that if B and C are known, then all other numbers can be calculated. The solution tries every possible pair B-C (each of the numbers is either fixed, or we try all values between 0 and 100), calculates the other numbers and checks if the quintuplet satisfies the conditions.

IVANA

The task description doesn't explicitly state Zvonko's strategy, just that he tries to win when he can i.e. looks for a sequence of moves that guarantees victory. One such strategy is to try to win with the largest score difference possible.

Zvonko does not know how well Ivana plays so part of his strategy is to assume she plays optimally as well, since he wants to be absolutely certain that he won't lose.

After Ivana's first move, the circle falls apart and we're left with a chain of numbers. The players alternate in taking numbers from the two ends of the chain (they get to pick which end to take off).

Suppose Ivana has made her first move and a chain of N–1 numbers is left. Let f(a, b) the largest score difference that the player currently up can achieve (this number can be negative if the player always loses), if what is left of the chain are numbers originally at indices a to b.

The current player either takes number at index a and leaves the opponent with the subproblem f(a+1, b), or he takes the number at index b and the opponent is left with the subproblem f(a, b–1). The function f can be efficiently calculated for all pairs of indices using dynamic programming.

The following identities hold:

- f(i, i) = 1, if the number at index i is odd, or 0 if it is even
- f(a, b) = max{ f(a, a) – f(a+1, b),
  f(b, b) – f(a, b–1) }

The solution has Ivana try each of the N numbers as her first move and the uses the function f to calculate if she has a winning strategy. The function f can be calculated at once for all the chains, because the chains share many subproblems.

DVAPUT

If there is a string of length K that occurs twice, then there is such a string of length K–1 and less. We can use binary search to find the largest such K. The remaining 94.5% of this text focuses on solving this subproblem.

We'll use an algorithm based on the Rabin-Karp string searching algorithm. Each substring will be assigned an integer (finite, relatively small compared to the size of the string), the so called hash value. The hash value of a string contains information about the string it came from.

The function that maps strings to their hash values is called the hash function. The function must always assign the same hash value to the same string, but different strings may map to the same hash value (the hash function is not injective). However, if two hash values are different, then they couldn't have come from the same string.

The algorithm uses a special type of hash function, a rolling hash function, with the additional property that the hash value of the next substring can be efficiently calculated from the hash value of the previous substring. One such function is similar to a positional numeral system, with an arbitrary base (say 26, the size of our alphabet).

The hash value of the string "abc" is hash("abc") = 'a'·26^2 + 'b'·26 + 'c'.

Similarly, hash("bcd") = 'b'·26^2 + 'c'·26 + 'd'.

Notice the relation hash("bcd") = 26·hash("abc") – 'a'·26^3 + 'd'. When generalized for strings of any length, the relation allows us to calculate the hash value of the next substring in O(1) complexity, instead of calculating it anew in O(K).

Here's the algorithm for the subproblem:

```
1  calculate the hash value of the first substring of length K
   and insert the pair (hash value, substring) into a set
2  for every other substring:
3     calculate the new hash value
4     if the hash value is already in the set:
5        check if there is a substring in the set equal to the
         current string (if so, this is the second occurrence)
6     else:
7        insert the pair (hash value, substring) into the set
```

The additional check in step 5 is needed since different strings can map to the same hash value (this is called a hash collision). The speedup over the naive algorithm is reducing the string comparison (slow) to an integer comparison (fast, step 4). Rarely do we have to do an actual string comparison (step 5).

All arithmetic operations are done modulo some integer M, so that all hash values are between 0 and M–1. The larger the M, the fewer collisions occur. In fact, if M is large enough (say 10^{15}), collisions become rare enough that we can omit the safety string comparison and have the algorithm work correctly with high probability (albeit not 100%).

It is important that the number M be relatively prime with the base of our hash function (26), so that the hash values disperse uniformly between 0 and M–1. If M is prime, then it is relatively prime with every base so a prime number is always a good choice.

The above algorithm does O(N) arithmetic operations and O(N–K) set operations. We still need to implement a set structure. It we use the `set` class in C++, the overall complexity is $O(N \log^2 N)$.

We can do better. If we make M smaller (say 200003), we can build what is called a hash table. The table consists of M buckets, with bucket i being a linked list containing all substrings that map to the hash value i.

Reducing M implies having the additional check in step 5, because the number of collisions is considerable (with M as large as L, the collision ration is about 0.5%). Increasing the number of buckets reduces the number of collisions, but the ratio is already small enough that decreasing it doesn't affect the overall efficiency of the solution.

Inserting into a hash table is O(1) since we can always put the new element at the front of the linked list.

The complexity of a find operation depends on the number of elements in the relevant bucket. But if the hash function disperses the elements into buckets mostly uniformly, the complexity will be O(1) on average.

The overall complexity of the solution is O(N log N).