培正喇沙 編程挑戰賽
LA SALLE – PUI CHING PROGRAMMING CHALLENGE

第三屆 · 2018

2018 年 5 月 22 日 (星期二)

香港培正中學

題解

題號		名稱	作者	難易度
A		Ambidexterity	董鑫泉	★☆☆☆☆
B		Bob the Builder	黃梓駿	★★★★☆
C		Consecutive Numbers	黃梓駿	★★☆☆☆
D		Differentiation	黃敏恆	★★★☆☆
E		Error	黃敏恆	★★☆☆☆
F		Final Fixture	董鑫泉	★★★☆☆
G		Go	黃敏恆	★★★★★★
H		Handicap	董鑫泉	★★☆☆☆
I		Infection	周君珽	★★★★★
J		Jakanda Forever	黃亦駿	★★★★☆
K		Kids' Entertainment	董鑫泉	★☆☆☆☆
L		Labyrinth	董鑫泉	★★★☆☆

難易度是指作者團隊認為能在比賽中解決該題的隊伍比例

由 1 星至 6 星為: >75%, 50-75%, 25-50%, 5-25%, 1-5%, <1%

A - Ambidexterity

This task is intended to be solved by all teams in the contest. Here is what the psuedo-code may look like:

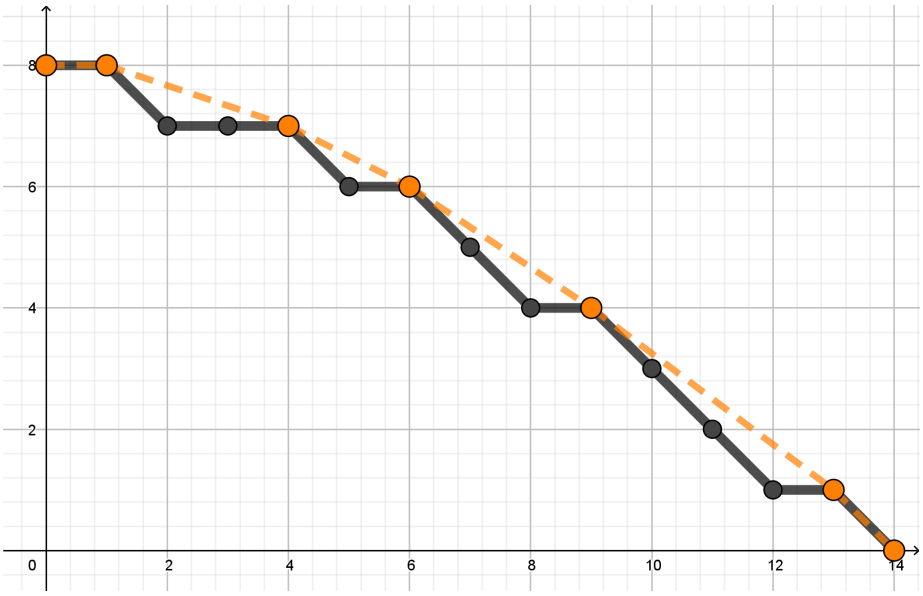
```
input L, R, P
if min(L, R) × 100 ≥ max(L, R) × P
   output "Ambidextrous"
else if L > R
   output "Left-handed"
else
   output "Right-handed"
```

It is better to check $min(L, R) \times 100 \geq max(L, R) \times P$ then to check $min(L, R) \geq max(L, R) \times P/100$, because the former does not involve floating-point operation.

B - Bob the Builder

Solution

First, you should observe that the set of vertices of the convex hull formed by the points of the railroad, excepting $(N, 0)$, is equivalent to the set of exciting points, and it's true for the converse. So, the problem can be modelled as: constructing the railroad to maximize the number of vertices in the convex hull.



Considering the convex hull formed, for each edge, consider the points in the railroad between the two vertices on this edge, they should be first going down (i.e. $H_i = H_{i-1} + 1$) for Y times, and go right (i.e. $H_i = H_{i-1}$) for X times. The slope of this edge can be considered as Y/X (it should actually be $Y/(X+Y)$, while using Y/X does not affect the comparisons). We can observe that for each of the edges, their slopes must be sorted in ascending order. Also, each edge has a width of $X + Y$.

If we first list out all possible pairs (X, Y), we can notice that in order to maximize the number of edges (our goal), we should choose those pairs with minimum $X+Y$ as they occupy less space. However, we chose two pairs (X_1, Y_1) and (X_2, Y_2) that have the same value of Y/X (i.e. $Y_1/X_1 = Y_2/X_2$), these two edges will be placed next to each other in the final placement, and the edges will be merged (since the middle point is lying on the line segment formed by the first and the last point). Therefore, it is not clever to choose two pairs with same value of Y/X. Under this situation, we should only consider comprime pairs (i.e. $\gcd(X, Y) = 1$) as they have smaller value of $X + Y$ (the width cost).

So, we should first list out sufficient comprime pairs (X, Y), and then select them starting from minimum $X + Y$, while their sum does not exceed N. After that, we should sort the selected pairs (X, Y) by the value Y/X to construct our railroad.

Implementation

We may first list all comprime pairs (X, Y) with their sum no more than 170 (170 is just enough for $N = 10^5$). Sort them by $X + Y$ and keep selecting pairs with minimum $X + Y$ until their sum is greater than N. For the selected pairs, sort them by Y/X and construct the railroad. For remaining space (if any), you may either add horizontal railroad at the front, or oblique railroad at the end.

Comments

No non-observer teams solved this problem during the contest:(

C - Consecutive Numbers

Solution

There are many ways to solve the problem, while only faster can pass the time limit.

$\mathcal{O}(X^3)$ per query - one loop exhausting the smallest number in the expression, another loop exhausting the largest number in the expression, the third loop finding the sum by iterating through the numbers.

```
for (int i = 1; i < X; i++)
  for (int j = i+1; j <= X; j++) {
    int sum = 0;
    for (int k = i; k <= j; k++)
      sum += k;
    if (sum == X)
      return true;
  }
```

$\mathcal{O}(X^2)$ per query - one loop exhausting the smallest number in the expression, another loop exhausting the largest number in the expression, then use the formula $\frac{1}{2}(L+R)(R-L+1)$ to calculate the sum.

```
for (int i = 1; i < X; i++)
  for (int j = i+1; j <= X; j++)
    if ((i+j) * (j-i+1) / 2 == X)
      return true;
```

$\mathcal{O}(\sigma_0(2X))$ per query - ($\sigma_0(2X)$ is the number of divisors of $2 \times X$) iterate through the divisors d of $2X$, check if d and $2X/d$ can be expressed as $(L+R)$ and $(R-L+1)$

However, the above solutions are too slow that cannot answer 10^5 queries within a second. The crucial observation is that, the answer is `No` if and only if X can be expressed as 2^k with some non-negative integers k.

$\mathcal{O}(\log_2(X))$ or $\mathcal{O}(1)$ per query - check if X is power of 2 by keep dividing it by 2, or use GCC built-in function `__builtin_popcount` to check if `__builtin_popcount(X)==1`.

Comments

During the contest, a possible approach to this problem is to list the answers for small X. Once you listed, it is easy to observe the pattern, and you may give it a try.

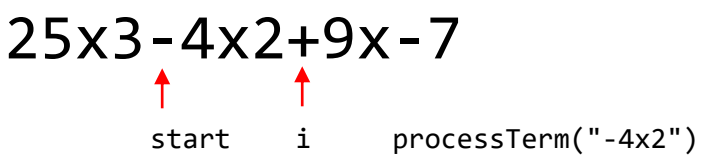
Additional challenge

Prove why is the observation true.

D – Differentiation

We store the entire input string into variable `str`. Initially the output string `out` is empty.

1. Split `str` into terms, and process them individually.




```
int start = 0;
for (int i = 1; i <= str.length(); i++) {
  if (i == str.length() || str[i] == '+' || str[i] == '-') {
    out += processTerm(s.substr(start, i - start));
    start = i;
  }
}
```

2. In `processTerm(termstr)`, determine the coefficient and degree separately. To do so we first check the existence / position of `x` in `termstr`

```
auto xpos = termstr.find('x');
if (xpos == string::npos) { // x is not found
  return; // This is constant term, so its derivative is 0
}
int degree = strToInt(termstr.substr(xpos, 100));
int coeff = strToInt(termstr.substr(0, xpos));

int derivativeDegree = degree - 1;
int derivativeCoeff = degree * coeff;

stringstream termOutput;
if (derivativeCoeff > 0) termOutput << "+";
// Hint: We don't need to check whether derivativeCoeff is +1 or -1
termOutput << derivativeCoeff;
if (derivativeDegree >= 1) termOutput << "x";
if (derivativeDegree >= 2) termOutput << derivativeDegree;

return termOutput.str();
```

```
int strToInt(string s) {
  if (s == "" || s == "+")
    return 1;
  if (s == "-")
    return -1;
  return atoi(s.c_str());
}
```

3. If the output starts with `+`, remove it.
   `cout << out.substr(out[0] == "+", 1000);`

4. Special case: The polynomial contains only the constant term, then the output is `0`.

E - Error

First, we illustrate a greedy algorithm that is not a correct solution:

> Repeat ***N*** times:
>
> - Among the remaining numbers, choose the one that causes the result to be as close to 0 or 99999999 as possible. (without going over)
> - For example, if the current number is 60000000 and there are -20000000, +5000000 and +10000000 remaining, +10000000 would be chosen because the result (70000000 is 29999999 from 99999999 but 60000000 – 20000000 = 40000000 is 40000000 from 0).

A counter example to the above algorithm is:

44444444
55555555
80000000
-99999999

The greedy algorithm would choose 80000000 as the first step, then in the next step the error would be inevitable.

Since ***N*** is small (≤ 10), we can try all ***N***! permutations and see if there is a solution. The permutations can be generated using recursion. C++ users may also use `next_permutation(a, a + n)`.

F - Final Fixture

Best Rank

We focus on team i.

For team i to obtain the highest possible rank, assume that they win by a huge margin, say $10^9 - 0$, and that scorelines in other matches are not that "extreme", so that team i will have the best goal difference among all teams.

Let the score of team i be s after the final match. In other words, $s = S_i + 3$. Classify all teams other than team i and its opponent, according to their points **before** the final fixture.

Category	Points	Meaning
I	$s+1$ or more	will rank better than team i
II	s	will rank better than team i with a draw/win
III	$s-1$ / $s-2$	will rank better than team i with a win
IV	$s-3$ or fewer	will rank worse than team i

For all matches, we only need to care about the categories of the two teams, and think about how to minimize the number of teams that end up above team i. For example:

- For Category I vs Category II, a win for the Category-I team is preferred.
- For Category II vs Category II, anything but a draw is preferred.
- For Category III vs Category III, a draw is preferred.

Worst Rank

For team i to obtain the lowest possible rank, assume that they lose by a huge margin, say $0-10^9$, and that scorelines in other matches are not that "extreme", so that team i will have the worst goal difference among all teams.

Similar to the part of finding the best rank, categorize the other teams as below:

Category	Points	Meaning
I	s or more	will rank better than team i
II	$s-1$	can rank better than team i with a draw/win
III	$s-2$ / $s-3$	can rank better than team i with a win
IV	$s-4$ or fewer	will rank worse than team i

How to maximize the number of teams ending above team i?

- For Category I vs Category III, a win for the Category-III team is preferred.
- For Category II vs Category II, a draw is preferred.
- For Category III vs Category III, anything but a draw is preferred.

et cetera. The overall time complexity is $O(N^2)$.

Why Allow Inconsistent Data?

If consistency is to be taken into account, it would be incredibly difficult to generate good data - one has to generate a list of matches and carefully tune the results of each of them.

Besides, a correct solution to this problem will work for many football leagues in the world. While extreme scorelines are rare in reality (two sides rarely score 10 goals combined, let alone 10^9), teams with similar points usually have similar goal differences, so the program should work just as well.

G – Go

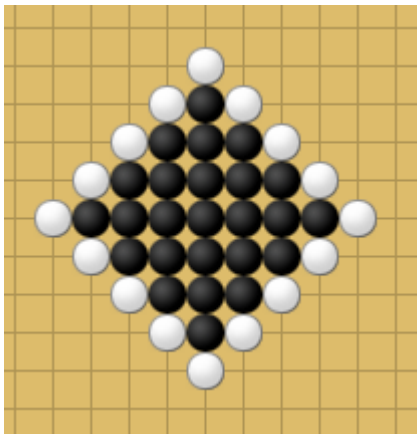
This is the hardest task in the problemset. It was expected that no teams could solve it during contest time.

Recall that we need to use exactly ***N*** white stones to surround ***M*** black stones in a 19x19 grid.

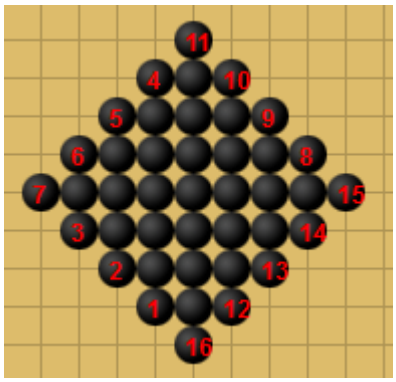
Possible condition

For this kind of constructive algorithm task, it's useful to first determine the possible condition. For a specific ***M***, there is a range of possible ***N***. For small ***M***, it can be done carefully by hand. In general, the upper bound of ***N*** is 4***M*** because you can use 4 white stones to surround 1 black stone. However, the lower bound, i.e. the minimum number of white stones required to surround ***M*** black stones, is much harder.

To use the minimum number of white stones, we try to arrange the black stones in a diamond shape like this:



For other values of ***N***, we place stones around the diamond in some order, so that the number of added white stones is minimized. The following illustrates one of the possible ordering:



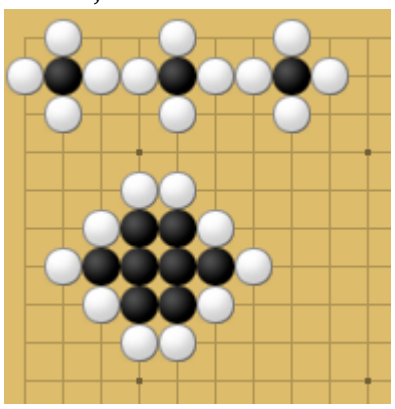
The black stones labelled 1, 4, 8, 12 causes the number of white stones to increase by 1. Other stones will not cause the number of white stones to increase.

Although it's not possible to do so during contest, if you type 4, 6, 7, 8, 8, 9 into OEIS, you can get the formula: a(n) = ceiling(2 + sqrt(8*n-4)) (https://oeis.org/A261491)

Construction

There are many possible algorithms. Here is one:

First, we make groups of 4 white stones and 1 black stone. By doing so we can make ***N*** much smaller, so that we can use the algorithm above to construct the remaining part. For example, ***N*** = 22, ***M*** = 11



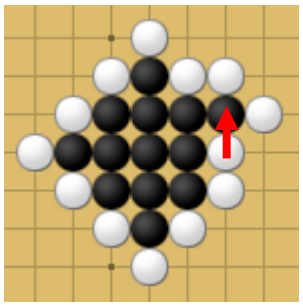
After this step, the only possible values of ***N*** relative to the lower bound of ***N*** (*minN*) are:

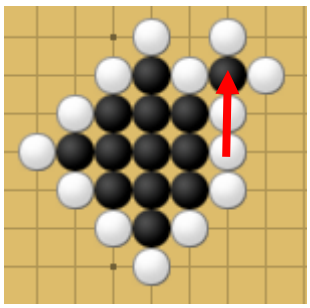
minN (where you can apply the construction directly)
minN+1, *minN*+2, or *minN*+3 (where we adjust the construction slightly)

minN+4 or more is impossible (otherwise you can take 4 white stones and 1 black stone to make another separate group).

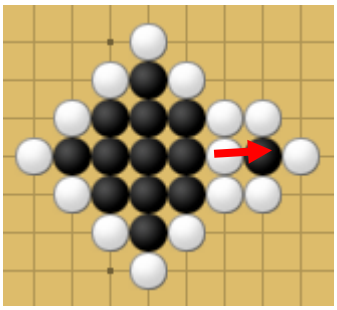
For *minN*+1, we shift a corner stone sideways by one position. This increases the number of white stones by 1.

For *minN*+2, we shift a corner stone sideways by two positions. This increases the number of white stones by 2.





For *minN*+3, we move a corner stone away from the diamond by one position. This increase the number of white stones by 3.



Implementation

After figuring out the algorithm, you may implement its logic by placing black stones only and ignore the white stones. The white stones can be added to the grid during output phase. (Output `o` instead of `.` if there is an adjacent `x`)

H - Handicap

To simplify matter, notice that "no handicap" is not so special; one can treat it as the *zeroth* handicap, with $H[0] = 0$.

The obvious solution, which runs through all handicaps and check which one minimizes $|Y_{new} - X|$, has time complexity $O(NQ)$ and is too slow.

We first handle the requirement that Y_{new} must be positive, by finding the largest index i, $0 \le i \le N$, such that $Y - H[i] > 0$. This can be done using a binary search. Note that such index i must exist, since $Y - H[0] = Y > 0$.

Next, we need to find the largest k, $0 \le k \le i$, such that $Y - H[k] - X$ is nonnegative. Again, such index must exist.

Then, the answer is either k or $k + 1$. Here is why:

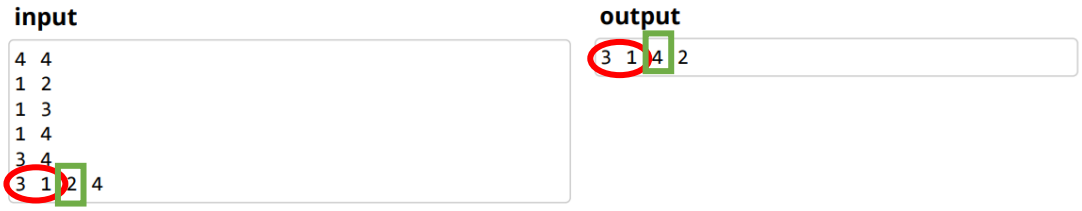
- For indices j smaller than k, $H[j]$ is smaller than $H[k]$, and $Y - H[j] - X > Y - H[k] - X \ge 0$, implying $|Y - H[j] - X| > |Y - H[k] - X|$.
- For indices j larger than $k + 1$, $H[j]$ is larger than $H[k + 1]$, and either $H[j] \ge Y$ or $Y - H[j] - X < Y - H[k+1] - X \le 0$. In any case, choosing $k + 1$ is better.

Choosing the better of the two indices can of course be done in $O(1)$ time. Therefore, the overall time complexity is $O(Q \log N)$.

I – Infection

We can observe that the answer is always in form of something like this, copying first x^{th} element of the given order for some value x and append to some new ordering.

Also, in order to make the answer lexicographically greater than the given one, the first number in the new ordering should be greater than the $(x+1)^{th}$ element in the given order.



For example, in sample 1, the answer – {3, 1, 4, 2} is constructed by copying the first two element of the given order {3, 1} and append to {4, 2}. As 4 is greater than 2, it is lexicographically greater than the given order.

Moreover, we can observe that if value x is greater, our answer will be lexicographically smaller. So, our problem is now transformed into finding the maximum value of x which we can construct some ordering in the way we mentioned.

We can find such value x by binary search or linear search. Let's talk about linear search. We iterate i from 1 to N, in each iteration, we perform BFS on the first i^{th} cities of the given order, if we can reach some cities which its index is greater than the $(i+1)^{th}$ cities of the given order, we can always construct an answer in the way we mentioned, so we can update value x as i.

After all the iteration, we found the maximum value of x and we can construct our answer using it. First, copying the first x^{th} element of the given order. Then, in all the reachable cities by performing BFS on the first x^{th} element of the given order, find the city with smallest index which is greater the the (i+1)th cities in the given order. After that, to make it lexicographically smallest, we continue to perform BFS and greedily visit city with smallest index and finally we get the answer.

You may use some data structure such as *std::set* to help you to construct the answer. Naïve implementation on linear search will have time complexity of $O(N^2 \log N)$. But you can improve it to $O(N \log N)$ or you may use binary search with time complexity of $O(N \log^2 N)$

J - Jakanda Forever

The problem could be solved in multiple ways, heavy light decomposition, lowest common ancestor (LCA) with segment tree and more. The easiest way is using DFS order and segment tree. Denote dist(v, u) be the distance of node v and node u on tree.

First root the tree at node 1. One critical(but quite obvious) observation is that if dist(v, u) is even. then parity of dist(root, v) and dist(root, u) is the same. as dist(v, u) = dist(v, lca) + dist(u, lca) = depth(v) + depth(u) - 2 * depth(lca) where lca is their lowest common ancestor. One could notice that depth(lca) is being subtracted twice. so lca actually don't affect the parity of dist(v, u).

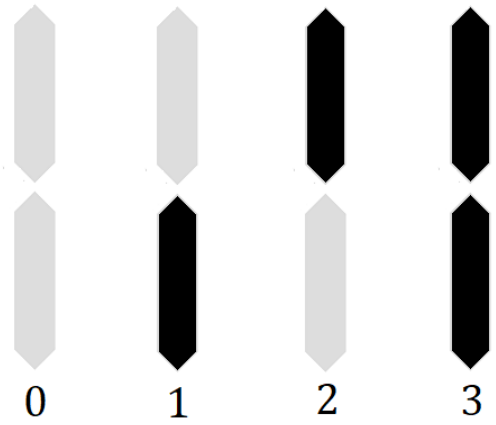
Things become easy now. We just have to maintain parity of dist(root, v) for every node. We could use segment tree. To make subtree as an interval, we use DFS order to maintain the segment tree. Whenever we change the parity of edge (u, v), assumer u is parent of v. we flip the element which belong to subtree of v. We could do range update in segment tree. For query just use point query in segment tree.

It's also possible to use binary indexed tree instead of segment tree.

K - Kids' Entertainment

To solve this task, one only needs to store the necessary information given in the task, and answer queries as required.

For each digit, store its number of segments $num[]$ and its left and right parts $L[]$ and $R[]$. For example, we can encode the parts with numbers $0 - 3$:



Let the number be XY, where X is the tens digit and Y is the unit digit. The psuedo-code may look like:

```
Set ans := num[X] + num[Y]
If X ≠ 1 and R[X] = L[Y]
   Set ans := ans - (number of '1' bits of R[X])
Output ans
```

L - Labyrinth

The first basic observation is that for any two points U and V, the *parity* of $dist(U, V)$ is invariant, as long as $dist(U, V)$ stays finite. Therefore, if $dist(A, X)$ and $dist(B, X)$ have different remainders modulo 2, the answer is automatically `Impossible`.

Have we covered all the `Impossible` cases? No, not yet.

For simplicity, assume that from now on:

- $dist(A, X) < dist(B, X)$
- $dist(A, X) \equiv dist(B, X) \pmod 2$

We ignore the case $dist(A, X) = dist(B, X)$ because it is trivial. As for $dist(A, X) > dist(B, X)$, the scenario is basically the same, but with A and B reversed.

The reader is encouraged to draw diagrams to visualize the solution described below.

Case 1: $dist(A, X) = 1$

This implies $A = (1, 1)$. No matter how many obstacles are added, the path $(1, 1) \to (2, 1)$ cannot be blocked. Therefore, $dist(A, X)$ cannot be changed, and the answer is `Impossible`.

Case 2: $dist(A, X) = 2$

This means $A = (1, 2)$. To increase $dist(A, X)$, obstacles must be added to both $(1, 1)$ and $(2, 2)$, forcing $dist(A, X) = 6$.

If $B = (3, 4)$, one can check that it is impossible to have $4 < dist(B, X) < \infty$. Output `Impossible` in this case. Otherwise, add obstacles to $(2, 3), (2, 4), \ldots$ in this order, until $dist(A, X) = dist(B, X)$. Each added obstacle increases $dist(A, X)$ by 2.

Case 3: $dist(A, X) \geq 3$

Adding obstacles to $(1, a - 1)$ and $(2, a - 1)$ makes $dist(A, X) = 4$. For further increase, add enough obstacles to $(2, a), (2, a + 1), \ldots$, in this order. Each added obstacle increases $dist(A, X)$ by 2.

The time complexity is $O(W)$.

Some challenges for the reader who is interested in case-bashing:

- Can you solve the task if X can be any square on the second row?
- Can you solve the task if the grid is of size $H \times W$, and A, B, X are allowed to be any *boundary* squares?