September 19, 2023

New Zealand Informatics Competition 2023

Round Two

Editorial by Anatol C

Contents

Introduction

The solutions to this round of NZIC problems are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution, e.g. $O(N)$. There is an explanation of Big O complexity at the end of this document.

Resources

Ever wondered what the error messages mean?

https://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:

https://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

Ever wondered why your submission scored zero?

https://www.nzoi.org.nz/nzic/resources/why-did-i-score-zero.pdf

See our list of other useful resources here:

https://www.nzoi.org.nz/nzic/resources

Accounting Woes

Problem authored by Bruce C
https://train.nzoi.org.nz/problems/1322

Subtask 1

In this subtask, $N = 2$ so there are only two branches that we can move expenditure between. We can intuitively balance the expenses by summing up the spendings and dividing it by two. If the total spending is even then the spending can be evenly divided, so no extra spending is needed. However if the total spending is odd, then one branch must end up with a dollar more than another. As such, one additional dollar must be given to the lesser branch to balance the expenses.

There are many other ways to solve this subtask.

Python Solution

```python
N = int(input()) # Always 2 in this subtask
total_spending = sum(input())
# To determine whether a number is odd or even, we can use the modulo operator (%) to
find the remainder when dividing by two.
print(total_spending % 2) # Prints 0 if spending is even or 1 if spending is odd
```

Subtask 2

Because we can move around money all we want, we can actually arrange the expenditure between in any way we want, as long as the total expenditure stays the same. Thus, if we calculate the total expenditure among all branches, call this value T, we can think of the problem as how do we ditribute T units across N branches to minimize the extra spending needed to ensure the expenses across the branches balance.

Note that we can always give every branch at least $\lfloor \frac{T}{N} \rfloor$ expenditure ($\lfloor x \rfloor$ rounds x down). Then, the amount of expenditure we still need to distribute, R, must be less than N. This means we can just add 1 expenditure to R distinct branches and this actually leaves us with an optimal configuration. Now, the result can be calculated as follows:

- If $R = 0$, all branches will end up with the same expenditure so the answer is 0
- Otherwise, there are $N - R$ branches where we need to add one expenditure to get all expenditures equal so the answer is $N - R$

Python Solution

```python
N = int(input())
nums = list(map(int, input().split()))
T = sum(nums)
base_value = T // N # The value we can always add to all branches
R = T - base_value * N

if R == 0: print(0)
else: print(N - R)
```

Extra Challenge

Can you figure out what these alternative solutions are doing and why they work?

Alternative 1

```python
N = int(input())
T = sum(map(int, input().split()))
print((T + N - 1) // N * N - T)
```

Alternative 2

```python
N = int(input())
nums = list(map(int, input().split()))
max_value = max(nums)
extra_needed = 0
for x in nums:
  extra_needed += max_value - x
print(extra_needed % N)
```

Crochet Rounds

Problem authored by Joseph G

https://train.nzoi.org.nz/problems/1321

Subtask 1

Because all the instructions are "increase" instructions we can easily simulate the entire process row by row. When processing a row we can keep track of the amount of stitches in the next row. Then, for each element in the current row we get the next instruction which will be an "increase x" instruction and we simply add x to the counter for the next row. Throughout this process we can easily track the amount of unused stitches and the row of the last created_stitch.

Python Solution

The code for this solution can be found at the end of this document .

Subtask 2

For this subtask we can adapt our subtask one solution to handle decreases. We can still do things row by row because decrease stitches will never use stitches from different rows. Then, instead of iterating over a row stitch by stitch, we need to modify our code to be able to skip over multiples stitches at once when there is a decrease instruction.

Python Solution

The code for this solution can be found at the end of this document .

Subtask 3

For full solution, we can keep track of all the unused stitches and their rows. Then for each instruction:

- If it is an "increase x" instruction, get the oldest stitch (the one that was added earliest), say it is in row r, remove it from the unused stiches and add x stiches of row $r + 1$.
- If it is a "decrease x" instruction, get the x oldest stitches, let the maximum row among them be r (this will also be the row of the youngest of these stitches), remove all of them from the unused stiches and add a single stitch of row $r + 1$.

How do we store the unused stitches to do these operations efficiently? We need to be able to quickly remove the oldest stitches and add new ones. The perfect data structure is a queue! (There are other ways of doing this).

Then, once we have simulated the whole process, the number of unused nodes is simply the size of our queue and we can easily track the row of the last added stitch like before. Altenrativaly, if our queue structure supports querying the youngest element, we can use that instead (This method is used in the example solutions below).

Python Solution

The code for this solution can be found at the end of this document .

C++ Solution

The code for this solution can be found at the end of this document .

Magic Square

Problem authored by Jonathan K
https://train.nzoi.org.nz/problems/1323

Subtask 1

Since there is only one element missing and it can take one of 10000 values, we can simply try all possible values for that missing square until we find a valid one.

Python Solution

The code for this solution can be found at the end of this document .

Subtask 2

In this subtask, since only at most two squares are missing, there will always be a complete row (and column). This means we can immediately know what the target sum for all the lines is by finding the complete row. This means if there is a line with only one unknown element, we can easily calculate the value of that element by subtracting the first two elements from the target sum.

Note that every square is part of at least two lines. Thus, for any missing element, since there is at most one other element missing, one of its two or more lines must have the other two squares filled in. This means for any missing element, since we know the target sum, we can always find it's value with the method given above.

To make implementation easier instead of: "for every missing square, find the lines it is a part of and use one to find the value", we can just iterate over every line once and if it has exactly one missing element, filling it in. This has the same effect.

Python Solution

The code for this solution can be found at the end of this document .

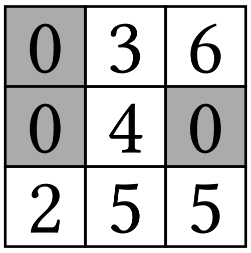
Subtask 3

We can use a similar method than in subtask 2 to solve subtask 3. The first issue in subtask 3 is in some cases, like the one shown below, we will have no complete line so we don't know the target sum.

0	3	6
7	0	1
2	5	0

However, because our procedure for filling in the grid once we know the target sum is very fast, we can simply try every possible target sum from 3 to 30,000 until we find a valid grid. This means:
- All values are between 1 and 10,000
- All lines add up to the target sum

There is still one more change needed. In subtask two, because there were at most two elements missing, we just needed to iterate over every line once to fill the grid. However, this breaks now because can have up to three missing elements and so an edge element might not always have a line with which we can calculate it's value. Take the case below as an example. The value of the leftmost element in the second row cannot be determined immediately.



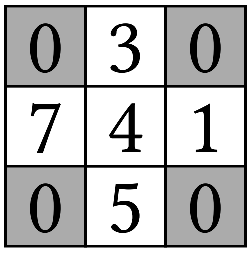
The solution for this is simple. Because every non-edge square is still a part of at least three lines, they will always be able to be determined immediately. Once we have all the non-edge pieces, it is clear that the value of any egdes can now be determined immediately. This means if we just iterate over every line *twice*, we will be able to fill the grid.

Note: even if you wan't figure out that you only need to iterate over every line twice, you can alternatively iterate over the lines until the grid is filled.

Python Solution
The code for this solution can be found at the end of this document .

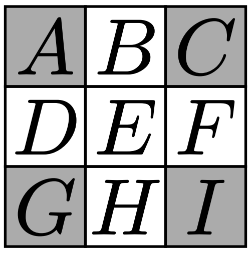
Subtask 4

The subtask 3 solution is actually very close to solving the full problem. The only problem is out grid-filling procedure will not always be able to completely fill the grid. Consider the case below. Even if we know the target value, there is no way to immediately determine the value of the corners.



Luckily, this is actually the only possible configuration of 5 or fewer missing elements where we cannot immediately find the value of any missing element. Hopefully it is easy to convince yourself of this fact. Otherwise, it can be proven by showing that for any such configuration, none of the corners' values can be known otherwise we would be able to find an unknown value.

Since this is the only breaking case, a solution is to solve the system of equations that arises from it by hand and then implement it as a special case. If we label the squares like below, we can solve for A, B, C, and D.



If we call the target sum for every line T, then we can form the equations below. (We can form more equations but these ones are enough to solve the system.)

$$\begin{aligned} A + B + C &= T \\ C + F + I &= T \\ I + H + G &= T \\ A + E + I &= T \end{aligned}$$

If we solve this system we get the following.

$$\begin{aligned} A &= \frac{T - E + F - B}{2} \\ C &= T - B - A \\ I &= T - F - C \\ G &= T - H - I \end{aligned}$$

Once we have filled in the special case, we still need to check if the configuration is valid because T may still be wrong.

One more thing, when filling in the grid in subtask 3, we just needed to iterate over all the lines twice. In this subtask it is trickier to find how many times we actually need to iterate over the lines. To be safe, we can simply do it until the grid is full. Otherwise, we can just do it five times since every iteration must fill at least one element and there are at most five missing elements (the example python solution does **not** do this as python is too slow).

Note: the example solutions integrate checking into their loop so are made to run for one extra iteration to ensure the checking is done.

Python Solution

The code for this solution can be found at the end of this document .

C++ Solution

The code for this solution can be found at the end of this document .

Number Friends

Problem authored by Nicholas G

https://train.nzoi.org.nz/problems/1318

Subtask 1 & 2

Note that the sequence we are looking for will just be an arrangement of all the values of a. Becuase N is so small, we can simply try every possible permutation in lexicographical order and check if it forms the correct friendships. This works in $O(N \times N!)$.

Python Solution

The code for this solution can be found at the end of this document .

Subtask 3

Because the sequence we are looking for is the lexicographically smallest, we can order the friendships in a greedy fashion. That is start with no friendships and at every step, find the friendship with the smallest a value that we can safely create at this point. This works because at every step, the sequence we find by picking the smallest value will be lexicographically smaller than any sequence we could possibly form by picking a larger value.

How do we decide if we can safely form a certain friendship at some point? For us to be able to select a friend pair (a, b), all the numbers $2a, 3a, ..., b - a$ must have all already have friends and b must not have a friend yet. Note the second condition. This means if we have one pair (a_1, b_1) and another (a_2, b_2) where $a_2 = b_1$, we cannot form the second pair before the first because making a_2 have a friend makes it impossible for a_1 to friend b_1.

So, at each step, to check if we can form a certain friend pair (a, b) we first check that all the numbers $2a, 3a, ..., b - a$ have friends in $O(N)$ (in the worst case the list $2a, 3a, ...$ will contain $O(N)$ elements) and then check that the number a does not appear as a b value in any unformed pair in $O(1)$.

However with N steps where we need to check N pairs in $O(N)$, this makes our algorithm $O(N^3)$ which is too slow for $N \leq 1{,}000$. However we can optimise the checking the numbers $2a$ through $b - a$. Instead of checking every time, we form a set of those numbers for every friend pair. Then, when a number x makes friends, we iterate over every pair and remove x from their set. This update operation is $O(N)$. Then, to check if any numbers are left to be friended before we can form a pair, we just check if its set is empty. This makes our algorithm $O(N^2)$ overall which is good enough for this subtask.

C++ Solution

The code for this solution can be found at the end of this document .

Sutask 4

To solve the full problem for $N \leq 100{,}000$, we need to optimize our subtask 3 solution. Firstly, instead of iterating over all N friend pairs to find the smallest valid one at each step, we can keep track all the ones that we can safely form in a priority queue so that at each step we can find

Firstly, storing all the sets of numbers $2a, 3a, ..., b - a$ for all friend pairs is very inefficient. We can start by aggregating them by distinct values of a. That is, for every **unique** value of a we have a set storing $2a, 3a, ..., b_{\max} - a$ where $b_{\max}$ is the largest integer b such that the friend pair (a, b) exists. It is tricky to figure out how many numbers we end up storing this way, but it appears to be around $O(N \log N)$ in the worst case. Then, when a number is friended, we can remove it from the set. If that number was the smallest element in that set, we find the next element in the set b. If the friend pair (a, b) exists we enqueue it as we now know that all the requisite numbers have been friended. (Note that because we need the elements' order we need an ordered set).

However, when a number gets a friend, we cannot iterate over every set and remove it. There are up to N sets and we would need to friend elements $O(N)$ times which would make this process $O(N^2 \log N))$ overall. Instead, for every number that we put into sets we can store what sets it is a part of. This allows us to only remove numbers from sets they are actually a part of. Since the total amount of numbers in sets is $O(N \log(N))$, this makes the process $O(N \log^2 N)$.

What is the time complexity of this solution? The setup involves adding $O(N \log N)$ numbers into ordered sets which has $O(N \log^2 N)$ time complexity. At each of N steps we need to poll from a priority queue once to get the next element. This is $O(N \log N)$ overall. Finally, the removal process is $O(N \log^2 N)$. This makes the solution $O(N \log^2 N)$ overall. For $N = 100000$, $N \log^2 N \approx 30{,}000{,}000$ which is good enough to pass in just under a second. (The time limit is two seconds).

C++ Solution

The code for this solution can be found at the end of this document .

Additional Example Solutions

Crochet Rounds Subtask 1 Python Solution

```python
n = int(input())
increases = []
for i in range(n):
  increase = input()
  increases.append(int(increase[1:]))

last_created = 0
unused = 6

curr_round = 0
curr_round_count = 6
next_increase_idx = 0

while next_increase_idx < n:
  next_round_count = 0
  for i in range(curr_round_count):
    if next_increase_idx >= n:
      break

    x = increases[next_increase_idx]
    next_increase_idx += 1

    next_round_count += x

    last_created = curr_round + 1
    unused -= 1
    unused += x

  curr_round += 1
  curr_round_count = next_round_count

print(unused)
print(last_created)
```

Crochet Rounds Subtask 2 Python Solution

```python
n = int(input())
instructions = []
for i in range(n):
  instruction = input()
  instructions.append((instruction[0], int(instruction[1:])))

last_created = 0
unused = 6

curr_round = 0
curr_round_count = 6
next_instruction_idx = 0

while next_instruction_idx < n:
  next_round_count = 0
  while curr_round_count > 0:
    if next_instruction_idx >= n:
      break
    kind, x = instructions[next_instruction_idx]
    next_instruction_idx += 1

    last_created = curr_round + 1

    if kind == 'I':
      next_round_count += x
      unused += x

      unused -= 1
      curr_round_count -= 1
    else:
      next_round_count += 1
      unused += 1

      unused -= x
      curr_round_count -= x


  curr_round += 1
  curr_round_count = next_round_count

print(unused)
print(last_created)
```

Crochet Rounds Subtask 3 Python Solution

```python
from collections import deque

stitches = deque()
for i in range(6):
  stitches.append(0) # Add the six initial stitches in row 0

n = int(input())

for i in range(n):
  instruction = input()
  kind = instruction[0]
  x = int(instruction[1:])

  if kind == 'I':
    row = stitches.popleft()

    for i in range(x):
      stitches.append(row + 1)
  else:
    max_row = 0

    for i in range(x):
      max_row = max(max_row, stitches.popleft())

    stitches.append(max_row + 1)

print(len(stitches))
print(stitches.pop()) # Get youngest element
```

Crochet Rounds Subtask 3 C++ Solution

```cpp
#include <iostream>
#include <queue>

using namespace std;

int main() {
  int n;
  cin >> n;

  queue<int> stitches;
  for (int i = 0; i < 6; i++) {
    stitches.push(0);
  }

  for (int i = 0; i < n; i++) {
    char kind;
    int x;
    cin >> kind >> x;

    if (kind == 'I') {
      int row = stitches.front();
      stitches.pop();

      for (int i = 0; i < x; i++) {
        stitches.push(row + 1);
      }
    } else {
      int row = 0;
      for (int i = 0; i < x; i++) {
        row = max(row, stitches.front());
        stitches.pop();
      }

      stitches.push(row + 1);
    }
  }

  cout << stitches.size() << endl;
  cout << stitches.back() << endl;
}
```

Magic Square Subtask 1 Python Solution

```python
grid = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
for i in range(3):
  row = input().split()
  for j in range(3):
    grid[i][j] = int(row[j])

def check_grid():
  #Calculate the sums on all lines and check if they are all equal
  row_one = grid[0][0] + grid[0][1] + grid[0][2]
  row_two = grid[1][0] + grid[1][1] + grid[1][2]
  row_three = grid[2][0] + grid[2][1] + grid[2][2]

  col_one = grid[0][0] + grid[1][0] + grid[2][0]
  col_two = grid[0][1] + grid[1][1] + grid[2][1]
  col_three = grid[0][2] + grid[1][2] + grid[2][2]

  diag_one = grid[0][0] + grid[1][1] + grid[2][2]
  diag_two = grid[2][0] + grid[1][1] + grid[0][2]

  return row_one == row_two == row_three == col_one == col_two == col_three == diag_one
== diag_two

#Find the zero
for i in range(3):
  for j in range(3):
    if grid[i][j] == 0:
      for val in range(1, 10001):
        grid[i][j] = val
        if check_grid():
          break

for row in grid:
  print(*row)
```

Magic Square Subtask 2 Python Solution

```
LINES = [
  [(0, 0), (0, 1), (0, 2)],
  [(1, 0), (1, 1), (1, 2)],
  [(2, 0), (2, 1), (2, 2)],
  [(0, 0), (1, 0), (2, 0)],
  [(0, 1), (1, 1), (2, 1)],
  [(0, 2), (1, 2), (2, 2)],
  [(0, 0), (1, 1), (2, 2)],
  [(2, 0), (1, 1), (0, 2)]
]

grid = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
for i in range(3):
  row = input().split()
  for j in range(3):
    grid[i][j] = int(row[j])

for row in grid:
  if 0 not in row:
    target_sum = sum(row)
    break

for line in LINES:
  num_zeros = 0
  total = 0
  for p in line:
    val = grid[p[0]][p[1]]
    total += val
    if val == 0:
      num_zeros += 1
      last_zero = p

  if num_zeros == 1:
    grid[last_zero[0]][last_zero[1]] = target_sum - total
```

```python
for row in grid:
  print(*row)
```

Magic Square Subtask 3 Python Solution

```python
LINES = [
  [(0, 0), (0, 1), (0, 2)],
  [(1, 0), (1, 1), (1, 2)],
  [(2, 0), (2, 1), (2, 2)],

  [(0, 0), (1, 0), (2, 0)],
  [(0, 1), (1, 1), (2, 1)],
  [(0, 2), (1, 2), (2, 2)],

  [(0, 0), (1, 1), (2, 2)],
  [(2, 0), (1, 1), (0, 2)]
]

grid = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
for i in range(3):
  row = input().split()
  for j in range(3):
      grid[i][j] = int(row[j])

for target_sum in range(3, 30000):
  valid = True
  result = [row.copy() for row in grid]

  for i in range(2):
    for line in LINES:
      num_zeros = 0
      total = 0
      for p in line:
        val = result[p[0]][p[1]]
        total += val
        if val == 0:
          num_zeros += 1
          last_zero = p

      if num_zeros == 1:
        new_val = target_sum - total
        if new_val > 10000 or new_val < 1:
          valid = False
        result[last_zero[0]][last_zero[1]] = new_val
      elif num_zeros == 0:
        if total != target_sum:
          valid = False

  if valid:
    for row in result:
      print(*row)
    break
```

Magic Square Subtask 4 Python Solution

```python
LINES = [
  [(0, 0), (0, 1), (0, 2)],
  [(1, 0), (1, 1), (1, 2)],
  [(2, 0), (2, 1), (2, 2)],

  [(0, 0), (1, 0), (2, 0)],
  [(0, 1), (1, 1), (2, 1)],
  [(0, 2), (1, 2), (2, 2)],

  [(0, 0), (1, 1), (2, 2)],
  [(2, 0), (1, 1), (0, 2)]
]

grid = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
for i in range(3):
  row = input().split()
  for j in range(3):
      grid[i][j] = int(row[j])

for target_sum in range(3, 30000):
  valid = True
  result = [row.copy() for row in grid]
  done = False

  while True:
    for line in LINES:
      num_zeros = 0
      total = 0
      for p in line:
        val = result[p[0]][p[1]]
        total += val
        if val == 0:
          num_zeros += 1
          last_zero = p

      if num_zeros == 1:
        new_val = target_sum - total
        if new_val > 10000 or new_val < 1:
          valid = False
        result[last_zero[0]][last_zero[1]] = new_val
      elif num_zeros == 0:
        if total != target_sum:
          valid = False

    if done: break

    if not valid or not any(0 in x for x in result):
      done = True

    if valid and result[0][0] == 0 and result[0][2] == 0 and result[2][0] == 0 and
result[2][2] == 0:
      result[0][0] = (target_sum - result[1][1] + result[1][2] - result[0][1]) // 2
      result[0][2] = target_sum - result[0][1] - result[0][0]
      result[2][2] = target_sum - result[1][2] - result[0][2]
      result[2][0] = target_sum - result[2][1] - result[2][2]


  if valid:
    for row in result:
      print(*row)
    break
```

Magic Square Subtask 4 C++ Solution

```
#include <iostream>
#include <vector>
#include <cstring>

using namespace std;

vector<pair<int, int>> LINES[8] = {
  {{0, 0}, {0, 1}, {0, 2}}, {{1, 0}, {1, 1}, {1, 2}},
  {{2, 0}, {2, 1}, {2, 2}}, {{0, 0}, {1, 0}, {2, 0}},
  {{0, 1}, {1, 1}, {2, 1}}, {{0, 2}, {1, 2}, {2, 2}},
  {{0, 0}, {1, 1}, {2, 2}}, {{2, 0}, {1, 1}, {0, 2}}
};

int grid[3][3], result[3][3];

int main() {
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      cin >> grid[i][j];
    }
  }

  for (int targetSum = 3; targetSum <= 30000; targetSum++) {
    bool valid = true;
    memcpy(result, grid, sizeof(grid));
    for (int i = 0; i < 6; i++) {
      for (auto& line : LINES) {
        int total = 0, numZero = 0;
        pair<int, int> lastZero;
        for (pair<int, int> p : line) {
          int val = result[p.first][p.second];
          if (val == 0) {
            numZero++;
            lastZero = p;
          }
          total += val;
        }

        if (numZero == 0) {
          valid = valid && total == targetSum;
        } else if (numZero == 1) {
          int newVal = targetSum - total;
          result[lastZero.first][lastZero.second] = newVal;
          valid = valid && newVal >= 1 && newVal <= 10000;
        }
      }

      if (valid && !result[0][0] && !result[0][2] && !result[2][2] && !result[2][0]) {
        result[0][0] = (targetSum - result[1][1] + result[1][2] -result[0][1]) / 2;
        result[0][2] = targetSum - result[0][1] - result[0][0];
        result[2][2] = targetSum - result[1][2] - result[0][2];
```

```
          result[2][0] = targetSum - result[2][1] - result[2][2];
        }
      }

      if (valid) {
        for (int i = 0; i < 3; i++) {
          for (int j = 0; j < 3; j++) {
            cout << result[i][j] << " ";
          }
          cout << endl;
        }
      }
    }
  }
```

Number Friends Subtask 1 & 2 Python Solution

```
from itertools import permutations

N = int(input())
a_values = []
friends = set()
for i in range(N):
  a, b = map(int, input().split())
  a_values.append(a)
  friends.add((a, b))

for order in permutations(a_values):
  friended = set()
  friends_made = set()
  for x in order:
    friend = 2 * x
    while friend in friended:
      friend += x
    friended.add(x)
    friended.add(friend)
    friends_made.add((x, friend))

  if friends_made == friends:
    print(*order)
    break
```

Number Friends Subtask 3 C++ Solution

```
#include <iostream>
#include <unordered_set>
#include <vector>

using namespace std;

int n, a[100000];
long long b[100000];
unordered_set<long long> conditions[100000], cantFriend;
bool formed[100000];

int main() {
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> a[i] >> b[i];

    for (long long j = 2 * a[i]; j < b[i]; j += a[i]) {
      conditions[i].insert(j);
    }
    cantFriend.insert(b[i]);
  }

  for (int i = 0; i < n; i++) {
    int minIdx = -1;
    for (int j = 0; j < n; j++) {
      if (formed[j]) continue;
      if (conditions[j].size() > 0) continue;
      if (cantFriend.find(a[j]) != cantFriend.end()) continue;
      if (minIdx == -1 || a[minIdx] > a[j]) {
        minIdx = j;
      }
    }

    cout << a[minIdx] << " ";
    formed[minIdx] = true;
    for (int i = 0; i < n; i++) {
      conditions[i].erase(a[minIdx]);
      conditions[i].erase(b[minIdx]);
    }
    cantFriend.erase(b[minIdx]);
  }
  cout << endl;
}
```

Number Friends Subtask 4 C++ Solution

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <set>
#include <unordered_set>
#include <queue>

using namespace std;

struct AData {
  set<long long> toRemove;
  unordered_set<long long> friends;
  vector<long long> deferredQueue;
  bool locked;
};

int n;
unordered_map<int, AData> aData;
unordered_map<long long, vector<int>> memberSets;
unordered_set<long long> hasFriends;
priority_queue<pair<int, long long>, vector<pair<int, long long>>, greater<pair<int,
long long>>> pq;

void enqueue(AData& data, int a, long long b) {
  if (data.locked) {
    data.deferredQueue.push_back(b);
  } else {
    pq.push({a, b});
  }
}

void release(long long x) {
  if (hasFriends.find(x) != hasFriends.end()) return;
  hasFriends.insert(x);
  for (int factor: memberSets[x]) {
    AData& data = aData[factor];
    if (data.toRemove.size() >= 2) {
      auto it = data.toRemove.find(x);
      if (it == data.toRemove.begin()) {
        auto next = it; next++;
        if (data.friends.find(*next) != data.friends.end()) {
          enqueue(data, factor, *next);
        }
      }
    }
    data.toRemove.erase(x);
  }

  if (aData.find(x) != aData.end()) {
    AData& data = aData[x];
    if (data.locked) {
      data.locked = false;
      for (long long b: data.deferredQueue) pq.push({x, b});
      data.deferredQueue.clear();
    }
  }
}

int main() {
  cin >> n;

  unordered_map<int, vector<long long>> friendsPerA;
  unordered_set<int> lockedNumbers;
  for (int i = 0; i < n; i++) {
    int a;
    long long b;
    cin >> a >> b;
    friendsPerA[a].push_back(b);
    lockedNumbers.insert(b);
  }

  for (auto& e: friendsPerA) {
    long long bmax = 0;
    AData& data = aData[e.first] = AData();
    data.locked = lockedNumbers.find(e.first) != lockedNumbers.end();
    for (long long x: e.second) {
      bmax = max(bmax, x);
      data.friends.insert(x);
      if (x == 2 * e.first) {
        enqueue(data, e.first, x);
      }
    }
    for (long long b = 2 * e.first; b <= bmax; b += e.first) {
      data.toRemove.insert(b);
      memberSets[b].push_back(e.first);
    }
  }

  for (int i = 0; i < n; i++) {
    pair<int, long long> p = pq.top(); pq.pop();
    cout << p.first << " ";

    release(p.first);
    release(p.second);
  }

  cout << endl;
}
```

Big O Complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list N items long. We often call the measured parameter N. For example, a list of length N.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can perform around 100 million (100,000,000) operations per second. For example, if the problem specifies a time limit of 1 second and an input of N as large as 100,000, then you know that an $O(N^2)$ algorithm might be too slow for large N since $100{,}000^2 = 10{,}000{,}000{,}000$, or 10 billion operations.

Time Complexity

The time taken by a program can be estimated by the number of processor operations. For example, an addition $a + b$ or a comparison $a < b$ is one operation.

$O(1)$ time means that the number of operations a computer performs does not increase as N increases (i.e. does not depend on N). For example, say you have a program containing a list of N items and want to access the item at the i-th index. Usually, the computer will simply access the corresponding location in memory. There might be a few calculations to work out which location in memory the entry i corresponds to, but these will take the same amount of computation regardless of N. Note that time complexity does not account for constant factors. For example, if we doubled the number of calculations used to get each item in the list, the time complexity is still $O(1)$ because it is the same for all list lengths. You can't get a better algorithmic complexity than constant time.

$O(\log N)$ time suggests the program takes a constant number of extra operations every time N doubles in size. For example, finding a number in a sorted list using binary search might take 3 operations when $N = 8$, but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very, very large before a computer starts to struggle.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually, if you have one loop that iterates N times your algorithm is $O(N)$.

$O(N^2)$ time means the number of operations is proportional to N^2 . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O(N^2)$. This is generally not great for large values of N, which can take a very long time to compute. As a general rule of thumb in contests, $O(N^2)$ algorithms are only useful for input sizes of $N \lesssim 10{,}000$. Usually, if you have a nested loop in your program (loop inside a loop) then your solution is $O(N^2)$ if both these loops run about N times.