Competitive Programming and Mathematics Society

Programming Workshop #2

IMC Competition Solutions Walkthrough

Tunan Shi et al.

Credits

- **Doss Product**: Isaiah Iliffe, Tunan Shi
- **Jonathan and Tunan’s Sweets**: Jonathan Lam, Joseph Luo
- **All of the Above**: Isaiah Iliffe, Joseph Luo
- **IMC Banner I/II (Div A and B)**: Isaiah Iliffe, Tunan Shi, Angus Ritossa
- **Trading at IMC I/II (Div A and B)**: Isaiah Iliffe, Angus Ritossa
- **CSESocial Distancing**: Isaiah Iliffe, Jonathan Lam
- **Cece’s Honeycomb**: Joseph Luo, Jonathan Lam
- **Pear Pairs**: Angus Ritossa, Tunan Shi, Isaiah Iliffe
- **Laser Cutting**: Tunan Shi, Isaiah Iliffe
- **Housing Restrictions**: Isaiah Iliffe, Tunan Shi, Joseph Luo
- **Infinity War Spoilers**: Tunan Shi, Isaiah Iliffe
- **Mexican Wave**: Tunan Shi, Angus Ritossa

Table of contents

CSESocial Distancing

Problem

What is the minimum number of seats required to seat n people in a circular table, given the kth person must be A_k seats away from any other person?

Sample Input

```
3
1 2 3
```

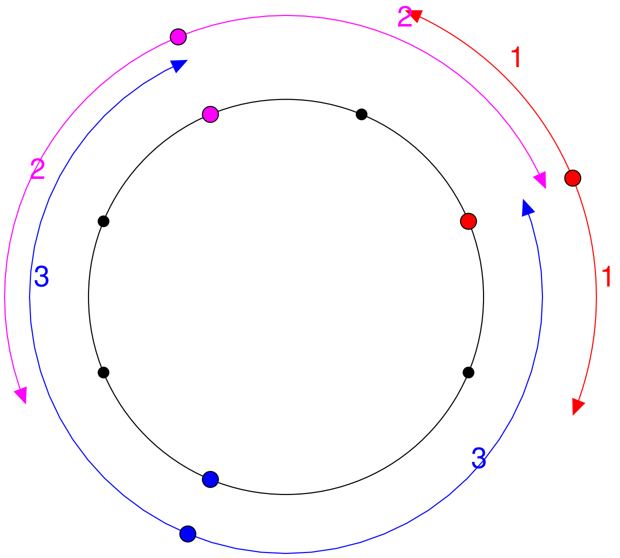
Sample Output

```
8
```

Constraints: $1 \leq N \leq 100000$, $1 \leq A_i \leq 100$

Subtask: $1 \leq N \leq 10$

Distancing



Distancing

- It is worthwhile trying out small cases by hand!
- $N = 1$: Just one seat is needed
- $N = 2$: $2 \times \max(A_1, A_2)$
- $N = 3$: not as obvious?
- Try a greedy approach:
  - Sort the distances and seat the people in that order
  - This ensures no 'extra' seats are reserved unnecessarily... except
  - The gap between the 'most sick' and 'least sick'.
  - Total seats: All the distances, except the smallest, plus the
  - Last gap size: $\max(\mathrm{A})$
- Formula:

$$\text{Min seats} = \begin{cases} 1 & \text{for } N = 1 \\ \operatorname{sum}(A) - \min(A) + \max(A) & \text{for } N > 1 \end{cases}$$

Distancing

```python
# Solution by Jonathan
N = int(input())
A = list(map(int, input().split()))

if (N == 1):
    print("1")
else:
    print(sum(A) + max(A) - min(A))
```

Honeycomb

Problem

A honeycomb is composed of a collection of cells of the same size. The length of a side of the honeycomb is the number of hexagons that makes up that side. Find the number of cells with side lengths A, B, C, D, E, and F in clockwise order.

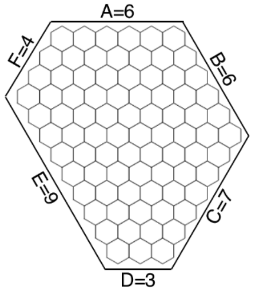
Sample Input

```
6 6 7 3 9 4
```

Sample Output

```
81
```

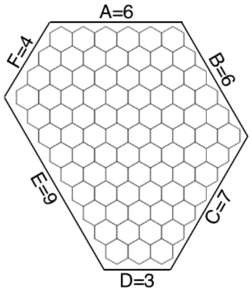
Constraints: $2 \leq A, B, C, D, E, F \leq 5000$



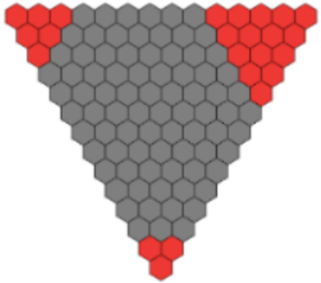
Honeycomb

- Whenever there's an equiangular hexagon, always try to extend sides to form an equilateral triangle! This will provide you with sufficient information to find the missing lengths.

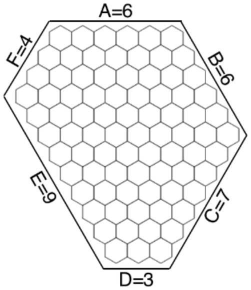
From This

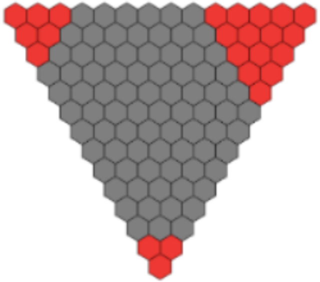


To This



Honeycomb





- The figure gives us information that
  the number of cells in honeycomb = (the number of cells in the big triangle) - (the number of cells in three red triangles).
- The length of big triangle is $F + A + B - 2$, and consecutively $B - 1, D - 1, F - 1$ for the red-sided triangles.

Honeycomb

- The number of cells in a triangle can be obtained from the triangular number.
- The triangular numbers formula:

$$T_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Honeycomb - Python Code

```
#include <cstdio>

int main(){
        int a, b, c, d, e, f;
        scanf("%d%d%d%d%d%d", &a, &b, &c, &d, &e, &f);
        printf("%d\n", (f+a+b-2)*(f+a+b-1)/2
        -(f-1)*f/2
        -(b-1)*b/2
        -(d-1)*d/2
        );
}
```

Pear Pairs

CPMSOC

Problem

Given a sequence of N positive integers, how many distinct ordered pairs of integers can we make? $1 \leq N \leq 100000$

Sample Input

```
5
1 1 2 3 2
```

Sample Output

```
6
```

Pear Pairs

- To count the total number of pairs, we could just test every possible pair of indices in our sequence.
- For each index i, for each index $j > i$, store the pair (a_i, a_j), and at the end, count the number of distinct values.
- Too slow, because we have to test all $\binom{N}{2}$ combinations of indices, which is a total of 4999950000!
- To see if an algorithm is slow, we can roughly estimate the number of times things (array elements, variables) are processed, and if it's significantly bigger than 20 million then it will probably not run in time.

Pear Pairs - A Better Solution?

CPMSOC

Observation

*If we are choosing the position for our left endpoint (our index i) then we only care about the **leftmost occurrence** of each value.*

Example

1 2 3 1 4 5

- The blue 1 can make the pairs $\{(1,4),(1,5)\}$
- The red 1 can make the pairs $\{(1,2),(1,3),(1,4),(1,5)\}$
- Anything the blue 1 can make, the red 1 can also make!
- If we only pick the leftmost occurrence of each value as our i, we only have to check up to KN values, where K is the number of distinct values in the sequence.
- This solves the $K \leq 10$ subtask which is 50% of test cases.

Pear Pairs - Even Better?

Observation

The number of pairs that an index i can create is equal to the number of ***distinct elements*** *after i.*

Example

```
1  2  3  2  2  4  5  4  6
```

- Pairs 1 can make: $\{(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)\}$
- Pre-calculate the number of distinct elements after each index. On the next page this will be the purpose of the array u,
- Like last time, only consider the first occurrence of each value. This allows us to avoid double-counting.
- Allows us to check each i very quickly, even if all N elements are distinct!

Pear Pairs - Pseudocode

CPMSOC

Algorithm 1: Pear Pairs

```
Initialise array $u$ of length $N$ to $0$;
Initialise array $seen$ of length $N$ to $false$;
Initialise integer $answer$ to $0$;
for $j \leftarrow N-1$ to $0$ do
    if $j < N-1$ then
        $u[j] \leftarrow u[j+1]$
    if not $seen[a[j]]$ then
        $seen[a[j]] \leftarrow true$;
        $u[j] \leftarrow u[j]+1$
Reset array $seen$ of length $N$ to $false$;
for $i \leftarrow 0$ to $N-2$ do
    if not $seen[a[i]]$ then
        $seen[a[i]] \leftarrow true$
        $answer \leftarrow answer + u[i+1]$
```

Pear Pairs - Python Code

```
# Solution by Tunan
N, K = [int(x) for x in input().split()]
pears = [int(x) for x in input().split()]

# Store whether we've seen a number
seen = [False for _ in range(K+1)]
# Store the number of distinct values seen so far
seen_so_far = [0 for _ in range(N)]
ans = 0

for j in range(N-1, -1, -1):
    if j < N-1:
        seen_so_far[j] = seen_so_far[j+1]
    if not seen[pears[j]]:
        seen[pears[j]] = True
        seen_so_far[j] = seen_so_far[j] + 1

seen = [False for _ in range(K+1)]
for i in range(N-1):
    if not seen[pears[i]]:
        seen[pears[i]] = True
        ans = ans + seen_so_far[i+1]

print(ans)
```

Banner II

Problem

You are given a string consisting of the letters 'I', 'M' and 'C', in any order. You can erase some of the letters. Determine the maximum number of consecutive strings 'IMC' that can be formed, and the number of ways to achieve this maximum. Two ways are considered different if a letter at a certain position is erased in one way, but not in the other.

Sample input

```
8
IMCIMIMC
```

Sample output

```
2
3
```

Banner II

- If we just want to find the maximum number of IMCs, we can use a greedy algorithm
- Take the first I, the next M, the next C, the next I, etc. This will always create the maximum number of IMCs.

  ```
  IMCIMIMC
  ```
- The hard part is counting the number of ways to do it.
- We will use *dynamic programming* for this. We will describe the solution today, and we will introduce DP more generally in a future workshop!

Banner II

- In dynamic programming, we split a big problem into subproblems
- Let $f(i, c)$ be the maximum number of IMCs we can make if we start at the i-th character in the input, and for the first IMC we start it at the letter c (where $c \in \{I, M, C\}$.
- In `IMCIMIMC`:
  $f(0, I) = 2$, $f(1, M) = 2$, $f(4, I) = 1$, $f(5, I) = 1$, $f(6, I) = 0$, $f(6, M) = 1$.
- How can we compute this?

Banner II

$$f(i, I) = \begin{cases} 0 & \text{if } i = n \\ f(i+1, I) & \text{if } s_i \neq I \\ f(i+1, M) & \text{if } s_i = I \end{cases} \tag{1}$$

$$f(i, M) = \begin{cases} 0 & \text{if } i = n \\ f(i+1, M) & \text{if } s_i \neq M \\ f(i+1, C) & \text{if } s_i = M \end{cases} \tag{2}$$

$$f(i, C) = \begin{cases} 0 & \text{if } i = n \\ f(i+1, C) & \text{if } s_i \neq C \\ f(i+1, I) + 1 & \text{if } s_i = C \end{cases} \tag{3}$$

Banner II

- Let $g(i, x)$ be defined similarly, except it is the number of ways to achieve the maximum.

$$g(i, I) = \begin{cases} 1 & \text{if } i = n \\ g(i+1, I) & \text{if } s_i \neq I \\ g(i+1, M) & \text{if } s_i = I \text{ and } f(i+1, M) > f(i+1, I) \\ g(i+1, M) + g(i+1, I) & \text{if } s_i = I \text{ and } f(i+1, M) = f(i+1, I) \end{cases} \tag{4}$$

  Note in the last case, both options lead to optimal solutions, which is why we add the number of combinations together.
- The cases for M and C are similar, and are omitted.

Banner II

- The answers to the original question are $f(0, I)$ and $g(0, I)$.
- Time complexity? $O(n)$ states, $O(1)$ recurrence, and so $O(n)$ overall (as long as we only compute each function once!)
- Implementing this isn't too difficult - each case is just an if statement.

Restrictions

CPMSOC

Problem

You are building on a street with N spaces. Each can have a house or no house on it, subject to R restrictions, each stating that any X_i consecutive spaces can contain no more than Y_i houses. Print a street with maximum number of houses.

Sample Input

```
8 2
6 3
3 2
```

Sample Output

```
HH___HH_H
```

Constraints: $N \leq 10000, R \leq 100$

Subtask: $N \leq 100$

Restrictions

Conjecture

It is optimal to go from left to right and always place a house unless it violates a restriction.

- Proof
  - Suppose we don't place a house where we could have
  - Then we can move the next placed house to this spot
    - Won't make any houses on the right invalid as we move further away from them
    - Won't make any houses on the left invalid by our supposition
  - We could have been no worse off by placing a house where we could have, hence it is optimal to do so

Restrictions

Conjecture

It is optimal to go from left to right and always place a house unless it violates a restriction.

- We can simulate this naively in $O(N^2R)$
  - For each of the N spaces, we check if building a house breaks any of the R restrictions
  - Breaking a restriction means building $\geq Y_i$ houses in $\leq X_i$ spaces for any i
  - Check each restriction in $O(X_i) = O(N)$
- Use prefix sum to check each restriction in $O(1)$ so total of $O(NR)$
- Exercise: solve the problem in faster complexity than $O(NR)$ or prove impossible or give a good reason why this is probably impossible

Restrictions

```
N, R = map(int, input().split())
X = list(map(int, input().split()))
Y = list(map(int, input().split()))                    # input

pre = [0]                                              # pre[x] = how many houses built before or at space x
for i in range(1,N+1):
    flag = False
    pre.append(pre[-1])
    for j in range(R):
        x, y = X[j], Y[j]
        if pre[i] - pre[max(0, i - x)] >= y:           # if placing will violate a restriction
            flag = True
    if not flag:
        pre[-1] += 1
        print("H", end='')                             # build house and update prefix sum
    else:
        print("_", end='')                             # don't build
print()
```

Mexican Wave

Problem

Given an array of N positive integers, how many subarrays (ranges) of the array have all the elements 1 to K? $1 \leq N, K \leq 100000$

Sample Input

```
5 2
3 2 1 4 2
```

Sample Output

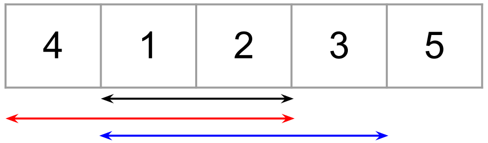
```
7
```

- Play around with some testcases by hand!

Mexican Wave

Observation

If a range of elements $e_l \cdots e_r$ *can form a valid Mexican wave, then so can* $e_{l-1} \cdots e_r$ *and* $e_l \cdots e_{r+1}$



Mexican Wave

Example

```
1 2 3 1 2 2 3 1 2
```

- Suppose we fix our right endpoint. What is the furthest left endpoint such that it is not a Mexican wave?

1 2 3 1 2 2 3 1 2

- Keep track of how many of each element are in our range, and keep track of whether any of these values are 0.

Mexican Wave - Pseudocode

CPMSOC

Algorithm 2: Pear Pairs

```
Initialise array $freq$ of length $K + 1$ to $0$;
Initialise $numzeroes \leftarrow K$ and $answer \leftarrow 0$ and $l \leftarrow 0$;
for $r \leftarrow 0$ to $N - 1$ do
    if $e[r] \leq K$ then
        if $freq[e[r]] = 0$ then
            $numzeroes \leftarrow numzeroes - 1$
        $freq[e[r]] \leftarrow freq[e[r]] + 1$
    while $numzeroes = 0$ do
        if $e[l] \leq K$ then
            $freq[e[l]] \leftarrow freq[e[l]] - 1$;
            if $freq[e[l]] = 0$ then
                $numzeroes \leftarrow numzeroes + 1$
        $l \leftarrow l + 1$
    $answer \leftarrow answer + l$
```

Mexican Wave - Python Code

CPMSOC

```python
# Solution by Tunan
N, K = [int(x) for x in input().split()]
e = [int(x) for x in input().split()]

# Array to keep track of how many times we have seen something in our current range
# We want at least one value to be seen to count number of invalid ranges.
freq = [0 for _ in range(K+1)]
# Number of values that have zero occurrences in our range. Will be updated on our walk.
num_zeroes = K
ans = 0

leftpointer = 0
for rightpointer in range(N):
    # Add the new element
    if e[rightpointer] <= K:
        if freq[e[rightpointer]] == 0:
            num_zeroes -= 1
        freq[e[rightpointer]] += 1
    # Remove elements until num_zeroes positive
    while num_zeroes == 0:
        if e[leftpointer] <= K:
            freq[e[leftpointer]] -= 1
            if freq[e[leftpointer]] == 0:
                num_zeroes += 1
        leftpointer += 1

    # Anywhere before the left pointer must be a valid starting position
    ans += leftpointer

print(ans)
```