●●●● Very Frequent 0/10

Ad Hoc Problems

Authors: Michael Cao, Aarav Sharma
Contributor: Ryan Chou

Problems that do not fall into standard categories with well-studied solutions.

Language: Python Edit This Page

TABLE OF CONTENTS

According to USACO Training section 1.2:

> **Ad hoc** problems are those whose algorithms do not fall into standard categories with well-studied solutions. Each ad hoc problem is different; no specific or general techniques exist to solve them.

In this module, we'll go over some general tips that may be useful in approaching problems that appear to be ad hoc.

- Draw lots of small cases to gain a better understanding of the problem. If you're having trouble debugging, draw more cases. If you don't know how to start with a problem, draw more cases. Whenever you don't know how to further approach a problem, you're probably missing an important observation, so draw more cases and make observations about properties of the problem.
- Whenever you find an observation that seems useful, write it down! Writing down ideas lets you easily come back to them later, and makes sure you don't forget about ideas that could potentially be the solution.
- Don't get stuck on any specific idea, unless you see an entire solution.
- Try to approach the problem from a lot of different perspectives. Try to mess around with formulas or draw a visual depiction of the problem. One of the most helpful things you can do when solving ad hoc problems is to keep trying ideas until you make progress. This is something you get better at as you do more problems.

In the end, the best way to get better at ad hoc problems (or any type of problem) is to do a lot of them.

Example - Milking Order

Milking Order
Bronze - Hard

Focus Problem – try your best to solve this problem before continuing!

Don't be afraid to give up on some approach if you aren't making progress!

Hint 1

Solution

Official Analysis (C++)

What if we tried placing cow 1 at every possible position?

Then, we'll have some hierarchy we have to fit in and some free cows which can go anywhere. Let's just handle the hierarchy, since we can fit in the free cows at the end.

As we sweep through the hierarchy, we'll also store a pointer that indicates our current position. Greedily, we should try to place these cows as early as possible to make sure that we have room to fit in all of them. As we go through the list, we have to make sure that this pointer never outruns some previous cow in our hierarchy.

This check takes $\mathcal{O}(N+M+K)$, which brings our total time complexity to $\mathcal{O}(N(N+M+K))$.

Copy PYTHON

```python
import sys

sys.stdin = open("milkorder.in", "r")
sys.stdout = open("milkorder.out", "w")

n, m, k = map(int, input().split())

hierarchy = [i - 1 for i in list(map(int, input().split()))]
order = [-1] * n

for i in range(k):
    cow, pos = map(int, input().split())

    order[pos - 1] = cow - 1

    if cow == 1:  # already fixed, nothing we can do
        print(pos)
        exit()


def check():
    """
    :return: whether it's possible to construct a
    valid ordering with given fixed elements
    """
    new_order = order.copy()

    cow_to_pos = [-1] * n
    for i in range(n):
        if order[i] != -1:
            cow_to_pos[order[i]] = i

    h_idx = 0
    i = 0
    while i < n and h_idx < m:
        # we know the next cow has to be in front of it
        if cow_to_pos[hierarchy[h_idx]] != -1:
            if i > cow_to_pos[hierarchy[h_idx]]:
                return False

            i = cow_to_pos[hierarchy[h_idx]]
            h_idx += 1
        else:
            while i < n and new_order[i] != -1:
                i += 1

            # run out of places
            if i == n:
                return False
```

```
            new_order[i] = hierarchy[h_idx]
            cow_to_pos[hierarchy[h_idx]] = i
            h_idx += 1

        i += 1

    return True


for i in range(n):
    # if already fixed, skip
    if order[i] == -1:
        # try placing cow 1 @ position i
        order[i] = 0

        if check():
            print(i + 1)
            break

        order[i] = -1
```

Casework

A casework problem is a type of ad hoc problem that needs to be broken down into different cases that each need to be accounted for.

These usually require drawing out a lot of cases and making observations about each. We can try to spot similarities and differences between cases and their solutions as well.

Example - Sleepy Cow Herding

Sleepy Cow Herding

Bronze - Easy

Focus Problem – try your best to solve this problem before continuing! *View Internal Solution*

Explanation

There are 3 cases for the minimum amount of moves:

1. The 3 positions are already consecutive.
2. Two elements are already in-position consecutively (including gaps), but the other is not.
3. Any other case that did not satisfy the two above.

For the first case, the answer would be 0 because the elements are already consecutive.

For the second case, the answer would be 1 because the only swap required is the one that would insert the isolated element into the gap between the two other elements.

The third case would output 2 because for any other test case, the optimal solution would be to take the minimum element to $\max - 2$, and then the new minimum to fit right in between the gap.

The maximum will always be finite because these operations group the cows closer together over time, as mentioned in the problem statement. The best approach to maximize the amount of moves is to place each element as close to a gap as possible (while not remaining an endpoint). Therefore, the maximum is the largest gap between two adjacent elements minus 1. Try it yourself to see that this is indeed the case.

Implementation

Time Complexity: $\mathcal{O}(1)$

Copy PYTHON

```python
with open("herding.in", "r") as file_in:
    a, b, c = map(int, file_in.readline().split())

"""
The minimum number of moves can only be 0, 1, or 2.
0 is if they're already consecutive,
1 is if there's a difference of 2 between any 2 numbers,
and 2 is for all other cases.
"""
if c == a + 2:
    minimum = 0
elif b == a + 2 or c == b + 2:
    minimum = 1
else:
    minimum = 2

# max is equal to largest difference between end and middle, minus one.
maximum = max(b - a, c - b) - 1

print(f"{minimum}\n{maximum}", file=open("herding.out", "w"))
```

Problems

Of course, ad hoc problems can be **quite easy**, but the ones presented below are generally on the harder side.

STATUS	SOURCE	PROBLEM NAME	DIFFICULTY	TAGS
	Bronze	**Sleepy Cow Sorting**	Hard	▸ Show Tags Complete Search
	Bronze	**Taming the Herd**	Hard	
	Bronze	**Modern Art**	Very Hard	
	Bronze	**Hoofball**	Very Hard	
	Silver	**Spaced Out**	Very Hard	▸ Show Tags Greedy

Quiz

What is most useful when solving ad hoc problems?

1. Being able to quickly recall complex algorithms and data structures.
2. Exploring lots of cases and making observations.
3. Relying on previous knowledge on well-studied topics.

← Previous Question 1 of 1 Skip →