Snail

Clarence Chew
clarence99chew@gmail.com

March 3, 2018

1 Introduction

This is a simple problem with quite a few edge cases.

2 Subtasks

As usual, if we do not know the full solution, we can start from easier subtasks first.

2.1 Subtask 1

In Subtask 1, $N = 1$. Just divide the height by the amount travelled per day and round correctly, and we are done.

2.2 Subtask 2

This is similar to Subtask 1 as all the increments are the same. Further divide the answer by the number of phases in a day and deal with the remainder.

2.3 Subtask 3

In Subtask 3, we notice that $H \times N$ is small, so this can be simulated in Python. Simulate H days. If the snail is not out yet, output that it is not possible.

2.4 Subtask 4

This subtask is left for buggy implementations of subtask 5. It is always possible to reach the top in this subtask unless all the numbers are 0.

2.5 Subtask 5

Here is the algorithm required:

Simulate the snail for $2N$ phases. We use $2N$ instead of N phases so we can detect the following case:

Sample Testcase

Input	Output
5 5 1 1 1 -10 2	1 2

If the snail does not reach the top, find out how much change happens per day. If the change is not positive, it is impossible.

Take the simulated values on day 1 and use them to figure out how many more days are required if we end on that phase.

Compare and select the minimum.

Task 2: Knapsack

Author: Gan Wei Liang
Singapore IOI Team 2012-2013
ganweiliang@u.nus.edu

Introduction

The abridged problem:
Given N items where the i-th item costs W_i per item and gives V_i value per item. There are also only K_i copies of the i-th item. Given a total budget of S, what is the maximum value that can be attained?

Subtasks

Subtask 1

Time complexity: $O(1)$ special case

Subtask 1 has the limit $N = 1$ which means that there is only 1 item. Since we only have a budget of S, if this item has weight W, we can only take $\lfloor \frac{S}{W} \rfloor$ items within the budget. However, we can also only take at most K items and each of them have value V. Thus, the maximum value we can get is $V \times \max(\lfloor \frac{S}{W} \rfloor, K)$

Subtask 2

Time complexity: $O(SN)$

Subtask 2 has the same budget but has $N \leq 100$ items and each item appears exactly once. This is in fact a classic dynamic programming problem called the 0-1 Knapsack problem. In short, you maintain a dynamic programming table $DP[i][w]$ which represents the maximum possible value that can be attained by only considering the first i items and with only a budget of w. We notice that we can either take the i-th item or don't take it. This corresponds to the formula

$$DP[i][w] = \max(DP[i-1][w-W_i] + V_i, DP[i-1][w])$$

which represents taking and not taking the i-th item respectively. We can then implement this using a recursive function. For more details of the implementation, please refer to `https://en.wikipedia.org/wiki/Knapsack_problem#0/1_knapsack_problem`

Subtask 3

Time complexity: $O(SNK)$

The only difference in this subtask is that there can be more than one copy of each item, but there will be no more than 10 copies. In this case, we can simply treat each copy of the item a separate item, thus creating $\sum_{i=1}^{N} K_i$ items which is still small enough.

Subtask 4

Time complexity: $O(SN \log K)$

In this subtask, K_i can be up to 10^9 so our previous solution does not work anymore. We need a way to try taking $1, 2, 3, ..., K_i$ copies of each item quickly.

Notice that we can group the K_i copies into $O(\log K_i)$ groups where the sizes are powers of 2 except the last one. For example, if $K_i = 25$, we can group into 5 groups of size $1, 2, 4, 8, 10$ respectively. Notice that the size of the groups add up to 25 and also any number from 1 to 25 can be represented by a sum of some subset of the above groups. Thus, by considering subsets of these groups, we would have considered every possibility of taking 1, 2, 3 ..., K_i items.

We can treat each group of size P as one item of weight PW_i and value PV_i and we would only have a total of $O(\sum_{i=1}^{n} \log K_i)$ items. Then, we can apply the solution in Subtask 2.

Subtask 5

Time complexity: $O(S^2 \log S + N \log N)$

In this subtask, N is increased to 10^5 so all the previous solutions do not run in time and we need another observation.

Notice that of all objects of weight W, we only need to consider the top $\lfloor \frac{S}{W} \rfloor$ valued items since we cannot take more than that. Hence, we group the objects by weight and sort each group by value and take the top $\lfloor \frac{S}{W} \rfloor$ valued items. This takes at most $O(N \log N)$ time.

After that, we run the solution from Subtask 2. The total number of items we have left is

$$\sum_{w=1}^{S} \lfloor \frac{S}{w} \rfloor = O(S \log S)$$

Thus the total time complexity is $O(S^2 \log S + N \log N)$ which is small enough.

Task 3: Island

Author: Bernard Teo
Singapore IOI Team 2012-2013
bernardteo@u.nus.edu

Introduction

The abridged problem:
Given a labelled tree with N leaves and M internal nodes, find the number of of ways to order its leaves when embedded into the plane. Furthermore, output this number as a product of some factors raised to exponents where each factor is at most 10^{18}.

Subtasks

Subtask 1

In Subtask 1, we are given the additional constraint that $M \leq 1$. This means that there are only two possibilities for M: either $M = 0$ or $M = 1$.

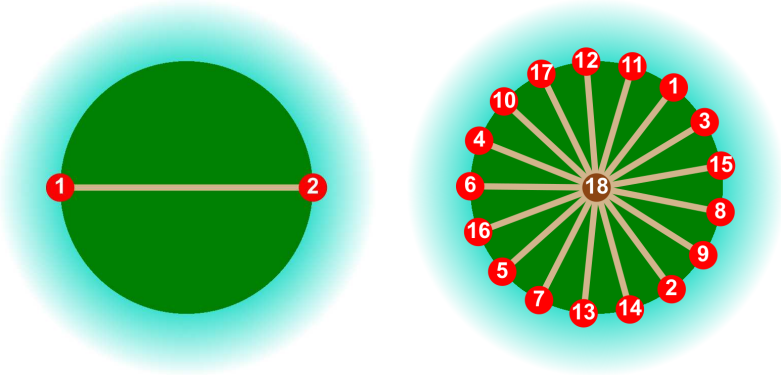

Figure 1: Some islands that satisfy subtask 1

When $M = 0$, there is only one possible island, which is shown on the left side of the figure above. There is no other possibility.

The case when $M = 1$ is more interesting. When $M = 1$, there is only one junction on the island, with a road between that junction and each town, such as the map of the island on the right of the figure above. Given N, the number of possibilities of town orderings is the number of circular permutations with N distinct elements, which is $(N-1)! = (N-1)^1 \times (N-2)^1 \times (N-3)^1 \times \cdots \times 2^1 \times 1^1$.

Subtask 2

In Subtask 2, there are at most 10 towns ($N \leq 10$), which makes for very small trees (since it may be proven that $M + 2 \leq N$). This makes it possible to validate each possible arrangement of the towns.

In addition, the constraints of Subtask 2 have been chosen such that it is unnecessary to maintain any factors. One can simply store the number of permutations as a 32-bit integer throughout the whole computation, and simply output the result without decomposition into factors.

Subtask 3 and 4

Subtask 3 and beyond is likely to require the contestant to develop a general polynomial-time algorithm to compute the number of valid orderings given the structure of the tree.

We can form a recursive formula for the number of valid orderings as follows:

Root the tree at an arbitrary town (i.e. leaf) and notice the following: At every junction, the number of possibilities for the subtree rooted at that junction is the product of the number of possibilities for all its children's subtrees multiplied by the number of ways to order its children.

In mathematical notation,

$$p(V) = \left(\prod_{W \in c(V)} p(W) \right) |c(V)|!$$

where $p(V)$ is the number of possibilities of the subtree rooted at junction V, and $c(V)$ is the set of children of junction V (note that since this tree is rooted, the number of children is usually one less than the degree of V).

This gives rise to a recursive solution.

While not absolutely necessary depending on the implementation, it may also be noted that we can solve the above recursive formula to find that

$$\text{total number of valid orderings of towns} = \prod_{V \in J} (\deg(V) - 1)!$$

where J is the set of junctions, and $\deg(V)$ is the degree of node V, and hence devise a simpler non-recursive algorithm that refrains from the selection of an arbitrary root.

We are now left with finding a way to output the answer in the format required

— as a product of factors raised to some powers. In other words, we want to express

$$\prod_{i=1}^{M} a_i!$$

as a product of factors raised to powers, given positive integers $a_1, a_2, a_3, \ldots, a_M$.

Observe that $n! = n^1 \times (n-1)^1 \times (n-2)^1 \times \cdots \times 2^1 \times 1^1$ for any integer n.

Hence, we may express the required factorization as

$$\prod_{i=1}^{M} a_i! = \prod_{j=1}^{\infty} j^{k_j}$$

where k_j is the number of a_is that are at least j, i.e. $k_j = |\{i \in [1, M] \mid a_i \geq j\}|$.

This is a static prefix sum query, which may be computed in $O(M + N)$ time (with bucket sort).

While this will already obtain a full score, the algorithm can be simplified further. Observe that the total number of factors in the expression above is equal to $\sum_{V \in J} (\deg(V) - 1)$, which is bounded by

$$\sum_{V \in J} (\deg(V) - 1) < \sum_{V \in J} \deg(V) < 2 \times (\text{total number of roads})$$

meaning that there is no need to collect similar factors and raise them to powers, because printing the $(n-1)$ factors of $(n-1)!$ every time a junction with degree n is encountered will output at most $2 \times$ (total number of roads) factors only.

Sample C++ code (with time complexity of $O(N + M)$):

```
#include <iostream>
#include <algorithm>
using namespace std;
unsigned n, m;
unsigned* adjlist;
int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cin >> n >> m;
    adjlist = new unsigned[m];
    fill_n(adjlist, m, 0);
    for(unsigned i=1; i<n+m; ++i){
        unsigned x, y;
```

```
        cin >> x >> y;
        if(x-- > n){
            if(adjlist[x-n] > 1){
                cout << adjlist[x-n] << " 1\n";
            }
            ++adjlist[x-n];
        }
        if(y-- > n){
            if(adjlist[y-n] > 1){
                cout << adjlist[y-n] << " 1\n";
            }
            ++adjlist[y-n];
        }
    }
    delete[] adjlist;
}
```

Subtask 3 allows for a time complexity of $O((N+M)^2)$, perhaps due to a slow sorting algorithm or some other inefficiency that causes each junction to be processed with a time complexity quadratic to its degree. It is unlikely in my opinion that anyone will solve Subtask 3 but not Subtask 4.