# SCT In-House Contest 1 2019-2020

Stephen Huan

December 2, 2019

## 1 A. Richard's Tax Evasion

In my opinion, the original statement is sort of confusing.

Richard has $1 \leq N \leq 10^5$ companies. Each company starts off with one box. Richard can move boxes between companies $1 \leq M \leq 3 \cdot 10^5$ ways, where each way is one-directional. Tax is for each company and on the presence of a box - a company holding 10 boxes is taxed the same amount as if it held 1. To evade tax, Richard attempts to temporarily move boxes and then move them back (i.e. each company starts off with one box and ends with one box). Output the lowest cost.

### 1.1 Example

```
Input
5
2 8 0 6 0
6
1 4
1 3
2 4
3 4
4 5
5 1
Output
8
```

Richard transfers all the boxes to companies 1 and 2. Companies are 1, 2, 3, 4, and 5. $5 \to 1$ directly, $4 \to 5 \to 1$, $3 \to 4 \to 5 \to 1$. Two doesn't change.

The amount of tax he has to pay is cost of 1 + cost of $2 = 8 + 0 = 8$. After the transfer he can move the boxes back by $1 \to 4$, $1 \to 3$, $1 \to 4 \to 5$.

### 1.2 Analysis

For Richard to use some node $A$ as a storage for $B$ there must exist some path $A \to B$ and $B \to A$. Suppose $A$ is also a storage for $C$. Then, there must exist a path $C \to B$ and $B \to C$ because they both can go through $A$. Therefore, it suffices to reduce the graph into connected components where each node in a component can reach every other node. The answer is then sum of the minimum cost node in each component.

The general concept is Strongly Connected Components. Such an algorithm runs in $O(|V| + |E|)$ which in this case is sufficient.

### 1.3 Implementation

In Python, the standard postorder recursive algorithm will run out of memory.

```
sys.setrecursionlimit(10**5)
```

is too big for codeforces and you end up with a catch-22: decrease the limit and you runtime error, increase it and you run out of memory. Instead, you have to write an iterative DFS. The idea is to keep a pointer with each node that keeps track of which child it's on. Once it's done with all its children, you can pop it off the stack and add it to the postorder traversal.

```
def visit(u, seen=set(), l=deque([])):
    if u in seen: return
    stk = [(u, 0)]
    seen.add(u)
    while len(stk) > 0:
        n, p = stk[-1]
        # either leaf or done with children
        if len(reverse[n]) == p:
            l.appendleft(n)
            stk.pop()
        # has children left to process
        else:
            child = reverse[n][p]
            if child not in seen:
                stk.append((child, 0))
                seen.add(child)
                stk[-2] = (n, p + 1)
            else:
                stk[-1] = (n, p + 1)
    return l
```

Given connected components, the solution is then a clean, Pythonic one-liner.

```
print(sum(min(map(lambda x: l[x], comps[i])) for i in range(num)))
```

## 2 B. Mario's Exploration

### 2.1 Example

```
Input
5 5 4
2 3
4 1
5 4
3 5
811 767 320 670 596
1 4
2 5 945
2 5 917
2 5 -851
1 1
Output
3164
811
```

In this case, 4 is the root. "1 4" means the sum of all the nodes in the subtree of that room, which for the root is all the rooms. $811 + 767 + 320 + 670 + 596 = 3164$. "1 1" will query the sum of just room 1. The updates don't effect its value, so it's still going to be 811.

### 2.2 Analysis

The basic idea is to turn the tree into an array. Each node in the tree will have a left and right index, such that its sum will be the sum of the values in the array between those indicies. For the root, it would have (0, N - 1) where N is the number of nodes.

A perfectly balanced binary tree with 7 nodes will have indicies of (0, 6), (0, 2), (3, 5), (0, 0), (1, 1), (3, 3), (4, 4) going from top to bottom, left to right (draw it out!)

First, run a DFS to convert the tree into a Directed Acyclic Graph (DAG). This is just for convenience, so you can tell whether a node is a leaf or not.

Then, run *another* DFS to generate the actual left and right bounds. For a node with children, its left index is going to be its leftmost child's left bound and its right index is going to be its rightmost child's right bound + 1. If the node is a leaf, its left and right will be the current length of the array.

See the iterative post-order DFS implementation from problem A. The formulation is the same, just keep track of used and add logic on removing a node from the stack.

```
# initialization
indexes = {}
used = -1

# either leaf or done with children
if len(graph[n]) == p:
    if p == 0:
        used += 1
        indexes[n] = (used, used)
    else:
        l, r = indexes[graph[n][0]][0], indexes[graph[n][-1]][1] + 1
        indexes[n] = (l, r)
        used = r

return indexes
```

Once you have the array, use your favorite $O(\log n)$ range query $O(\log n)$ range update data structure (I found 2 Binary Indexed Trees worked best, although I may just be bad at implementating Segment Trees). Sum queries for a room $r$ is just the sum of the array between the precomputed indicies, and so are the updates. This runs in $O(N \log N + Q \log Q)$ for BITs and $O(N + Q \log Q)$ for Segment Trees.

### 2.3 Implementation

Believe it or not, the most difficult part of this problem is reading in the input!

This cost me 20 submissions and Justin Choi a whopping 39. In C++, cin is synchronized with the C input styles (scanf) which makes it considerably slower.

Add the following to your main in order to speed it up:

```
int main(void) {
  // fast cin
  std::ios::sync_with_stdio(false);
  std::cin.tie(NULL);

  return 0;
}
```

# 3 C. Problem With Short Statement Part 1

## 3.1 Analysis

Suppose we have $k$ 1's in the prefix. That means the sorted prefix will look like:

```
0...0 1...1
n - k   k
```

The number of swaps necessary is then just the number of 1's in the $n - k$ region which is equivalent to the sum of the numbers from 0 to $n - k$. Both computations - finding $k$ and calculating the sum up to $n - k$ can be done in O(1) with prefix sums.

Given query(i, j) computes the sum between $[i, j]$, the number of swaps is:

```
query(0, i - query(0, i))
```

This runs in $O(N)$.

# 4 D. Problem With Short Statement Part 2

## 4.1 Analysis

The problem is identical to the first with the added complexity of updates. To efficiently update, the simplest solution is to use a BIT (mentioned in problem B). The only catch is that you have to have space both to the left and to the right in order to add onto the beginning and end. I intialize in the middle then keep track of left and right pointers.

This runs in $O(N \log N + Q \log Q)$.

# 5 E. Napsack Queries

## 5.1 Analysis

This is a classic "order matters" DP solution. Read this lecture by the writer himself.

The second part (answering queries) can be answered in $O(1)$ with prefix sums. The overall solution runs in $O(N + Q)$.

## 5.2 Implementation

There are only two points that need to be addreessed.

- The modulo can be negative since the prefix sum does higher minus lower. Output the answer with an if-statement where $y$ is the prefix sum and $m$ is the modulo.

```
print(y if y >= 0 else m + y)
```

- Since you have to output a fluffiness between $F-d$ and $F+d$, $F-d$ can be negative so you should use $\max(F-d, 0)$. Note that you should *not* use $\min(F+d, x)$ since that just means your array isn't big enough.

# 6 F. Word Problems

## 6.1 Analysis

This problem is ad hoc. Note the recurrence in the "dependancy". On each iteration, the length of the string doubles. The character of a given index depends on the characters in the previous iteration (obviously). The characters in the first half of string entirely follow the previous iteration (since they're copied over). However, the characters in the second half are shifted by one as one of the characters rotated.

```
012
BAT

012|120
BAT|ATB

012345|123450
BATATB|ATATBB

0123456789ab|1234567890ab
BATATBATATBB|ATATBATATBBB
```

Each iteration takes $O(1)$ to process and there are $O(\log N)$ iterations, making this algorithm run in $O(\log N)$.

## 6.2 Implementation

If the length of the string is $S$ and the index $N$, then the number of cycles $C$ is

$$S \cdot 2^C \geq N$$

Solving for $C$ yields $C = \lceil \log_2 \frac{N}{S} \rceil$.

The recurrence is simply:

```
def recur(s, i, l):
    if l == len(s):
        return s[i]
    m = l >> 1
    if i < m:
        return recur(s, i, l >> 1)
    else:
        return recur(s, (i - m + 1) % m, l >> 1)
```

The solution is then found with $i = N - 1$ and $l = S \cdot 2^C$.

```
print(recur(s, N - 1, len(s) << cycles))
```

# 7 G. ConnectN

## 7.1 Analysis

It suffices to reduce each number into its prime factors, as if two numbers share a factor they also share a prime factor. For example, 6 and 72 share a factor of 6, but they also share a factor of 2 and 3.

Once you have each prime factor and the numbers which have it as a factor, these are your preliminary connected components. To combine connected components, apply union-find. Each connected component can be represented as a tree and to find the largest connected component union each number with its prime factors. This runs in $O(pN)$ where $p$ is the number of primes and $N$ is the number of numbers plus the cost of factoring each number. Thanks to the asymptotic distribution of prime numbers, $\pi(n) = \frac{n}{\log n}$ where $\pi(n)$ is the number of prime numbers less than or equal to $n$. Thus, the algorithm runs in $O(\frac{n^2}{\log n} + \frac{n\sqrt{n}}{\log n}) = O(\frac{n^2}{\log n})$ if one factors with trial division. This is sufficient for $N = 2 \cdot 10^4$.

## 7.2 Implementation

Given that union joins two connected components, the solution is just

```
for p in ccs:
    for n in ccs[p]:
        union(parents, size, n, p)
```

Generation of a list of primes is efficient with the Sieve of Eratosthenes.

```
def sieve(n: int) -> list:
    """ Generates a look up table of primality up to a given number - O(n log log n)"""
    l = [True]*(n + 3)
    for i in range(2, int(n**0.5)):
        if l[i]:
            j = i*i
            while j < len(l):
                l[j] = False
                j += i
    return l
```