

Union find

- A union-find structure maintains a collection of sets. The sets are disjoint, so no element belongs to more than one set.
- Two nearly-constant time operations are supported:
 - the **union** operation joins two sets
 - the **find** operation finds the representative of the set that contains a given element
- Each member points to another member of the set (root) or itself (set size = 1)
- We keep the size of set for each set
- Implementation:
 - **Find**: just find the root of the set
 - **Is same?**: check if roots match
 - **Union**: point smaller set to the larger set



- `find(8)` is 2
- `same(2, 6)` is true

Union find

```
# we have n elements
link = [i for i in range(n)]
size = [1 for i in range(n)]

def find(x): # just find the root of the set
    if x == link[x]: return x
    return find(link[x])

def same(x, y):
    return find(x) == find(y)

def union(x, y):
    x, y = find(x), find(y)
    if size[x] < size[y]: x, y = y, x
    size[x] += size[y]
    link[y] = x
```

- **Problem:** you might end up with long chains, making `find` linear ($O(n)$)!

Union find: path compression

- Trick: anytime you see a node, point it to the root

```
# we have n elements
link = [i for i in range(n)]
size = [1 for i in range(n)]

def find(x): # just find the root of the set
    if x == link[x]: return x
    link[x] = find(link[x]) # yup, just this line!
    return link[x]
```

Virtual Friends

- <https://open.kattis.com/contests/rva2u2/problems/virtualfriends>
- Just do union find and print size after each union

These days, you can do all sorts of things online. For example, you can use various websites to make virtual friends. For some people, growing their social network (their friends, their friends' friends, their friends' friends' friends, and so on), has become an addictive hobby. Just as some people collect stamps, other people collect virtual friends.

Your task is to observe the interactions on such a website and keep track of the size of each person's network.

Assume that every friendship is mutual. If Fred is Barney's friend, then Barney is also Fred's friend.

Input

The first line of input contains one integer specifying the number of test cases to follow. Each test case begins with a line containing a positive integer F , the number of friendships formed. Each of the following F lines contains the names of two people who have just become friends, separated by a space. A name is a string of 1 to 6 letters (uppercase or lowercase). The sum of F over all test cases in the file is at most 100 000.

Output

Whenever a friendship is formed, print a line containing one integer, the number of people in the social network of the two people who have just become friends.

Sample Input 1

```
1
3
Fred Barney
Barney Betty
Betty Wilma
```

Sample Output 1

```
2
3
4
```

Lone Knight: <https://qoj.ac/problem/5597> (Div1)

- You have lots of rooks and knights
- Can you jump from cell X to cell Y with a knight without being attacked by a rook? You need to answer many queries!
- **Nobody solved this problem!** 🤖
- **Solution:**
 - Rooks cut infinite board to rectangles (sets!)
 - Max. $O(n^2)$ rectangles (n is below 1000)
 - Key trick: transform rectangles to sets!

Dynamic programming

Introduction

- Dynamic programming is a fancy name for complete search (brute force) that “remembers” the already seen states.
- Key ideas:
 - Find how to encode substates of the problem.
 - Make sure that substates / subtrees are independently solvable.
 - Make sure to remember already seen substates (e.g. hashmap).

Introduction

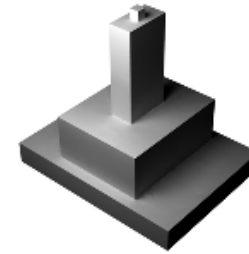
- When to use?
 - Anything that requires counting (count the number of ways to do X or Y).
 - Anything that requires optimization (find the largest / smallest item, best / worst score etc.).
 - Anything that requires optimality (find the optimal cut) and that can be done by enumerating all possibilities.

Two DP modes

- **Top-down DP:**
 - Just do standard recursion, remember what you have seen.
 - Hint: substates are usually the recursion parameters that keep changing (if your recursion is correct).
- **Bottom-up DP:**
 - Tabular method: fill the table from the smallest substate to the largest one.
 - Reversed top-down approach.
- Neither is “better”.
 - Top-down is easier to reason about and might use less memory.
 - Bottom-up might be faster as you do not waste time on recursion calls.

1D optimization: BABTWR

- **BABTWR:** <https://www.spoj.com/problems/BABTWR/>
 - n different types of blocks are available. Each one may be duplicated as many times as you like. Each type has a height y , a width x and a depth z .
 - Stack the blocks so that the resulting tower is as high as possible. Blocks can be rotated before stacking. For stability, a block can only be stacked upon another if both of its baselines are shorter.
- **Input**
 - The number of types of blocks n is on the first line of each test case. The subsequent n lines give the height y_i , width x_i and depth z_i .
 - At most 30 different types. Test cases end with $n = 0$.
 - You can assume $\max(x_i, y_i, z_i) \leq 2500$.
- **Output**
 - For each test case, output the height of the highest possible tower.



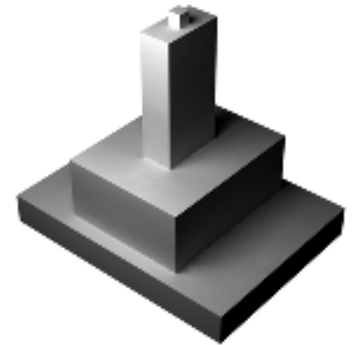
Sample input

```
5
31 41 59
26 53 58
97 93 23
84 62 64
33 83 27
1
1 1 1
0
```

Sample output

```
342
1
```

1D optimization: BABTWR



- Try each of the three choices for height; normalize each base so $x \leq y$.
- Sort orientations by (x, y) , from smaller bases to larger ones.
- Let H_i be the tallest tower whose **bottom** block is orientation i .
- A block above i must be strictly smaller in **both** base dimensions:

$$H_i = h_i + \max(\{0\} \cup \{H_j : x_j < x_i \text{ and } y_j < y_i\})$$

- The answer is $\max_i H_i$. Smaller bases have already been processed.
- Copies are unlimited: different rotations of the same type may appear in a tower.
 - **Strictly shrinking bases prevent cycles.**

BABTWR: Top-down

```
from functools import cache

def tallest_tower(blocks):
    rotations = []
    for a, b, c in blocks:
        for h, x, y in [(a, b, c), (b, a, c), (c, a, b)]:
            rotations.append((min(x, y), max(x, y), h))

    @cache # this "remembers" result for each i
    def tower(i):
        x, y, h = rotations[i]
        above = 0
        for j, (a, b, _) in enumerate(rotations):
            if a < x and b < y:
                above = max(above, tower(j))
        return h + above

    return max((tower(i) for i in range(len(rotations))), default=0)
```

BABTWR: Bottom-up

```
def tallest_tower(blocks):
    rotations = []
    for a, b, c in blocks:
        for h, x, y in [(a, b, c), (b, a, c), (c, a, b)]:
            rotations.append((min(x, y), max(x, y), h))
    rotations.sort() # smaller bases first
    height = [h for x, y, h in rotations]
    for i, (x, y, h) in enumerate(rotations):
        for j in range(i):
            a, b, _ = rotations[j]
            if a < x and b < y:
                height[i] = max(height[i], h + height[j])
    return max(height, default=0)
```

- $3n$ orientations: $O(n^2)$ time and $O(n)$ space.

2D counting: maximum rectangle sums

- UVa 108
- $O(n^6)$

```
def max_submatrix(a):  
    rows, cols = len(a), len(a[0])  
    best = a[0][0] # Handles all-negative matrices.  
  
    for top in range(rows):  
        for bottom in range(top, rows):  
            for left in range(cols):  
                for right in range(left, cols):  
                    # Sum every cell in this rectangle.  
                    total = 0  
                    for r in range(top, bottom + 1):  
                        for c in range(left, right + 1):  
                            total += a[r][c]  
                    best = max(best, total)  
  
    return best
```

Example: maximum sum

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

$$9 + 2 - 4 + 1 - 1 + 8 = \boxed{15}$$

Shaded: rows 2-4, columns 1-2.

Source: UVa 108

2D counting: maximum rectangle sums (faster)

- Use zero-based, **half-open** ranges: rows $[r_1, r_2)$ and columns $[c_1, c_2)$.
- Let $P_{i,j}$ be the sum of the first i rows and j columns; $P_{0,j} = P_{i,0} = 0$.
- Add the two neighboring prefixes, then subtract their overlap:

$$P_{i+1,j+1} = A_{i,j} + P_{i,j+1} + P_{i+1,j} - P_{i,j}$$

- Any rectangle sum takes four table lookups:

$$S(r_1, c_1, r_2, c_2) = P_{r_2, c_2} - P_{r_1, c_2} - P_{r_2, c_1} + P_{r_1, c_1}$$

- $O(n^2)$ preprocessing and space; $O(1)$ per query.

```
def prefix_sums(a):
    rows, cols = len(a), len(a[0])
    # Extra padding to avoid checking +- 1
    p = [[0] * (cols + 1) for _ in range(rows + 1)]
    for i in range(rows):
        for j in range(cols):
            p[i + 1][j + 1] = (a[i][j] + p[i][j + 1]
                               + p[i + 1][j] - p[i][j])
    return p

def rectangle_sum(p, r1, c1, r2, c2):
    return p[r2][c2] - p[r1][c2] - p[r2][c1] + p[r1][c1]

a = [[1, 2, 3], [4, 5, 6]]
p = prefix_sums(a)
# [[0, 0, 0, 0],
```

2D counting: maximum rectangle sums (faster)

- Try every **nonempty** rectangle; use the prefix table for its sum.
- $\text{answer} = \max_{r_1 < r_2, c_1 < c_2} S(r_1, c_1, r_2, c_2).$

```
def max_submatrix(a):  
    p = prefix_sums(a) # defined on the previous added slide  
    rows, cols = len(a), len(a[0])  
    best = a[0][0] # not zero: all entries might be negative!  
    for r1 in range(rows):  
        for r2 in range(r1 + 1, rows + 1):  
            for c1 in range(cols):  
                for c2 in range(c1 + 1, cols + 1):  
                    total = rectangle_sum(p, r1, c1, r2, c2)  
                    best = max(best, total)  
    return best
```

- $O(n^4)$ for an $n \times n$ matrix.

2D counting: maximum rectangle sums (fastest)

- **Kadane's algorithm** in 1 dimension: find the subarray with max. sum
- Let E_i be the best **nonempty** subarray sum ending at position i .
- DP: either start here, or extend the best subarray ending just before here:

$$E_0 = a_0, \quad E_i = \max(a_i, E_{i-1} + a_i), \quad \text{answer} = \max_i E_i$$

```
def kadane_1d(values):
    prev = best = values[0]
    for i in range(1, len(values)):
        prev = max(values[i], prev + values[i])
        best = max(best, prev)
    return best

kadane_1d([4, 11, -10, 1]) # 15: subarray [4, 11]
kadane_1d([-5, -2, -3])   # -2: not an empty subarray!
```

- $O(n)$ time and $O(1)$ space

Kadane in 2D: compress rows — Added

UVa 108 example: fix rows 2–4.

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

Add these rows **column by column**:

```
[ 9,  2, -6,  2]
+ [-4,  1, -4,  1]
+ [-1,  8,  0, -2]
= [ 4, 11, -10, 1]
```

Run Kadane on `[4, 11, -10, 1]` :

x	Best ending here	Best so far
4	4	4
11	15	15
-10	5	15
1	6	15

- The best subarray is columns **1–2**, with sum $4 + 11 = 15$.
- In the matrix, this is exactly the shaded rectangle.
- Try **every pair of top/bottom rows**; Kadane chooses the left/right columns in linear time.

Maximum rectangle: Kadane — Added

```
def kadane_1d(values):
    prev = best = values[0]
    for i in range(1, len(values)):
        prev = max(values[i], prev + values[i])
        best = max(best, prev)
    return best

def max_submatrix_kadane(a):
    rows, cols = len(a), len(a[0])
    best = a[0][0]
    for top in range(rows):
        sums = [0] * cols
        for r in range(top, rows):
            for c in range(cols):
                sums[c] += a[r][c]
            best = max(best, kadane_1d(sums))
    return best

a = [[ 0, -2, -7,  0],
      [ 9,  2, -6,  2],
      [-4,  1, -4,  1],
      [-1,  8,  0, -2]]
max_submatrix_kadane(a) # 15
max_submatrix_kadane([[-5, -2], [-3, -4]]) # -2
```

- `sums[c]` is the sum of column `c` from `top` through `bottom`.
- $O(n^3)$ for a square matrix.

2D counting

- **Variations:**

- Find a submatrix with smallest mean.
- Find a submatrix with all ones if input is a 0/1 matrix.
- Find a largest submatrix by area containing all ones if input is a 0/1 matrix.
- Find a largest square submatrix containing all ones if input is a 0/1 matrix.
- Find the number of submatrices containing all zeros (or ones) if input is a 0/1 matrix (Google Interview).