Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

Student Copy
AIEP Send-off Python 2 Session 9

DISJOINT-SET DATA STRUCTURE

In the previous handout, we explored priority queues, how they can be implemented using binary heaps, and how they can be applied to greedy algorithms where a partial ordering suffices and the list of values is frequently altered. This session is a continuation of our "journey" with efficient data structures.

To preface our discussion, consider this group of shinobi from the anime *Naruto.*[1] Treat the numbers on the top row as identifiers (IDs) of the shinobi, not as array indices.

0	1	2	3	4	5	6	7
Hinata	Gaara	Rasa	Sakura	Naruto	Sasuke	Chiyo	Sasori

After years of having no alliances, they decided to form clans (clans are distinguished by color). A shinobi can belong to only a single clan.

0	4	2	1	3	5	6	7
Hinata	Naruto	Rasa	Gaara	Sakura	Sasuke	Chiyo	Sasori

Clans merged to become villages. Once again, a shinobi can belong to only one village.

0	4	3	5	2	1	6	7
Hinata	Naruto	Sakura	Sasuke	Rasa	Gaara	Chiyo	Sasori

Suppose we would like to know the village to which Sasuke belongs. From a computational standpoint, this underscores two interesting considerations. First, the shinobi (technically, their names or IDs) should be stored in a data structure that enables efficient merging. Second, this data structure should also permit efficient finding.

These motivate the need for a union-find data structure (*union* is the formal term for merging sets). Introduced in a 1964 paper by American computer scientists Michael J. Fisher *(bottom right)* and Bernard A. Galler *(left)*, it is also referred to as a disjoint-set data structure, precisely because it is a collection of disjoint sets: sets that do not have any element in common. In our sample scenario, the disjoint sets are the clans and, later on, the villages.

Its standard implementation is via a collection of trees, known as a disjoint-set forest. Each tree maps to a set, i.e., values that belong to the same tree belong to the same set. At this point in our discussion, our only focus is on representation; we will tackle concerns related to efficiency separately. Therefore, neither the shape nor the structure of the trees should matter for now. Moreover, a node can have any number of children, and the root is arbitrarily selected from the nodes.

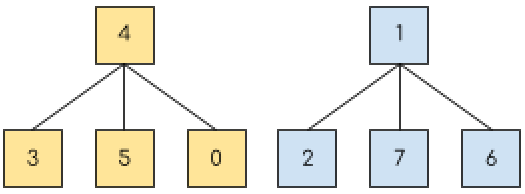
[1] This is not a faithful retelling of *Naruto.*

In our previous session, we learned how a binary heap, a tree-based data structure, can be represented as an array — allowing us to track parent-child relationships using only indices. We can take a bit of inspiration from this idea in trying to understand the general rationale for implementing a disjoint-set forest as an array.

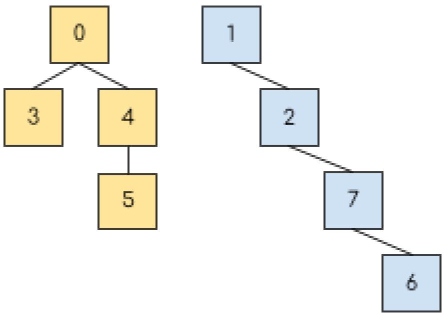
Let `P` be the array representation of a given disjoint-set forest[2]:

- `P[i]` is the parent of `i`
- If `i` is the root node, then `P[i] = i`.

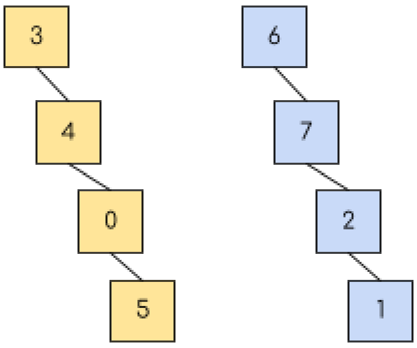
Below are three different (but equally valid) ways to translate the villages from our sample scenario into disjoint-set forests, side by side their corresponding array representation. Note that the array representation restricts the values that can be stored in our data structure to integers (since `i` has to be an integer). Nevertheless, this is not a quandary, as we can identify the shinobi using their numerical IDs.



i	0	1	2	3	4	5	6	7
P[i]	4	1	1	4	4	4	1	1



i	0	1	2	3	4	5	6	7
P[i]	0	1	1	0	0	4	7	2

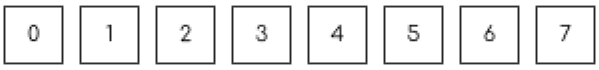


i	0	1	2	3	4	5	6	7
P[i]	4	2	7	3	3	0	6	6

Let us delve into the operations supported by this data structure.

✨ CREATING A DISJOINT-SET DATA STRUCTURE

Initially, all the sets are disjoint singletons (single-element sets), as seen below:



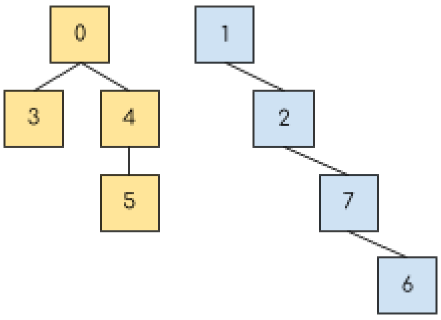
i	0	1	2	3	4	5	6	7
P[i]	0	1	2	3	4	5	6	7

[2] For now, let us dispense with the distinction between a node and its key in order to make our explanations less verbose. Hence, when we say that `i` is the root node, what we really mean is that `i` is the key of the root node.

We create a class[3] for the implementation of our data structure. Sometimes, as we will see later in our first illustrative example, it may also be useful to keep track of the number of elements in each set (i.e., its size).

```
class DisjointSet:
    def __init__(self, num_elements):
        self.parent = [i for i in range(num_elements)]
        self.size = [1 for _ in range(num_elements)]
```

🔍 FINDING THE ROOT



Consider the disjoint-set forest shown in the adjoining figure. Suppose we would like to find the root of the tree to which 6 belongs (i.e., the representative element of the set to which the said value belongs). Running a simple trace evinces a recursive chain:

- The root of 6 is the root of its parent (its parent is 7).
- The root of 7 is the root of its parent (its parent is 2).
- The root of 2 is the root of its parent (its parent is 1).
- The root of 1 is the root of its parent. But, following our array representation, its "parent" is itself — signaling that we have reached the base case of our recursion and that 1 is the root of the tree to which 6 belongs.

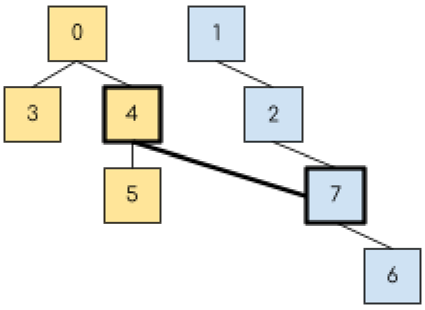
While this task can be accomplished iteratively, the recursive solution is more elegant:

```
def find(self, element):
    if self.parent[element] != element:
        return self.find(self.parent[element])

    return self.parent[element]
```

🤝 MERGING TWO DISJOINT SETS

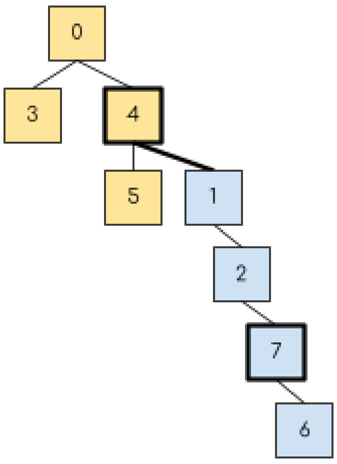
Using the same disjoint-set forest as in the previous section, consider the task of taking the union of the sets to which the elements 4 and 7 belong. An initial attempt might be to directly connect the two of them, as illustrated in the figure below:



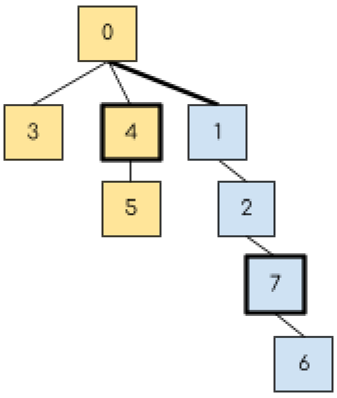
However, this can be problematic. Turning 7 into a child of 4 will result in 7 having two parents: 4 and 2, but each node in a tree data structure is allowed only one parent. Turning 4 into a child of 7 leads to a similar issue. We can force ourselves to resolve this, but the price to pay is a convoluted restructuring of a number of existing parent-child relationships.

[3] A *class* is a concept in *object-oriented programming* (OOP). It can be thought of as a blueprint for bundling the variables and methods needed by an object (in our case, a disjoint-set data structure) in its implementation.
This handout does not require prior knowledge of OOP since our emphasis is more on the logic of the data structure; coding is important, but it is only secondary. Nevertheless, you may refer to these tutorials for a quick introduction to OOP in Python.

A counterproposal would be to connect one of the elements with the root of the tree to which the other element belongs, as shown in the adjoining figure. We arbitrarily choose 1 (the root of the tree to which 7 belongs) to be a child of 4.



We are able to avoid the dilemma from earlier since 1 does not have any parent prior to the two sets' union. This scheme is also advantageous in that we need to change the parent of only a single node. However, our tree has gotten quite tall, which poses a challenge to efficiency. To find the root, we essentially have to "climb up" the tree; consequently, as the height increases, the runtime of the find and union operations increases as well.



Fortunately, we still have some tricks up our sleeves. Instead of turning one of the elements into the parent of the root of the tree to which the other element belongs, we can establish a root-to-root connection, as seen in the figure on the left.

Formally, if the root nodes of the trees to which the first and second elements belong are r_1 and r_2, respectively, we arbitrarily select r_2 and make it a child of r_1, thus merging the two trees.

For now, we settle on this approach and translate it to code. Note that, if $r_1 = r_2$, nothing needs to be done since the two elements already belong to the same set.

```
def merge(self, element1, element2):
    root1 = self.find(element1)
    root2 = self.find(element2)

    if root1 != root2:
        self.parent[root2] = root1
        self.size[root1] += self.size[root2]
```

N.B. The size of the set to which an element `e` belongs is not `size[e]` but `size[find(e)]`.

On another note, it may be best to avoid naming the function `union` since it is a keyword in languages like C++ (although not in Python).

⚠️ Not So Fast

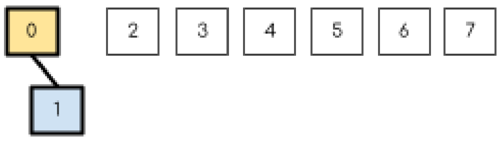
We now have a working implementation of a disjoint-set data structure. Notice that the efficiency of our methods relies heavily on the height of the tree. While we tried to factor in the height when we were coming up with our union method, our approach is, in truth, still naïve. Can you think of a sequence of operations that will result in a skewed tree?

In the next sections, we are going to introduce two landmark optimization techniques that can significantly improve the runtime of the find and union operations — ultimately bringing their (amortized) complexity down to almost-constant time!

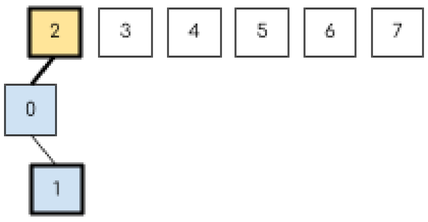
💡 OPTIMIZATION #1: Union by Rank

Using our current implementation of union, let us simulate the sequence of method calls listed below. Assume that we are starting with eight disjoint singletons.

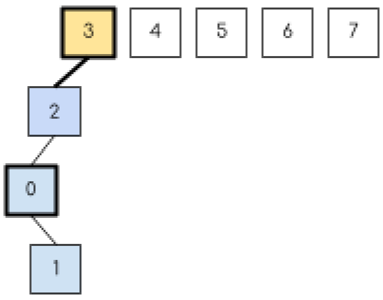
`union(0, 1)`



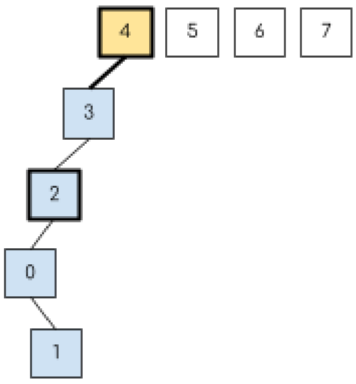
`union(2, 1)`



`union(3, 0)`



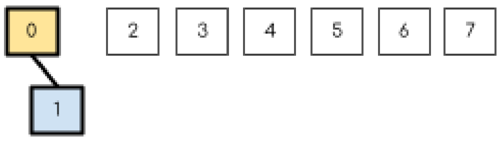
`union(4, 2)`



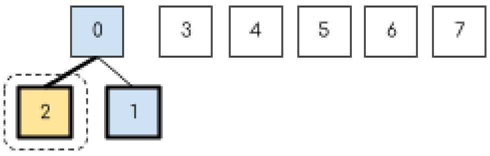
The weakness of our present approach becomes evident: since the tree can be skewed, the worst-case time complexity of both the find and union operations degrades to $\Theta(n)$, where n is the number of elements. Ergo, instead of our current scheme, it would be better to select the root of the shorter tree and turn it into a child of the root of the taller tree.

Let us retrace the same sequence of union operations following this new idea (the shorter tree is enclosed in a dashed box).

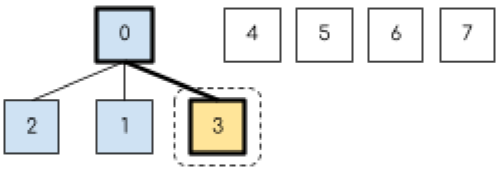
`union(0, 1)`



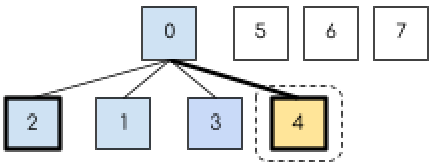
`union(2, 1)`



`union(3, 0)`



`union(4, 2)`



Even after four successive union operations, the tree is as short as it can possibly be.

This strategy of attaching the shorter tree to the taller tree in the interest of balancing the height is known as union by rank. But why did we bother to introduce a new term when we could have simply said "height"? At this point in our discussion, rank and height are, indeed, identical. But, when we apply our next optimization technique (path compression), we will encounter cases wherein the tree's rank is not necessarily the same as its height (and we do not want to burden ourselves with keeping track of the *exact* height).

> The rank is a loose upper bound or an overestimate of the tree's height.
> It is a value (a heuristic) that is always greater than or equal to the height.

If the ranks of the two trees are equal, then we just arbitrarily attach the tree to which the first element belongs to the tree to which the second element belongs.

Note that we have to create an array to store the ranks. Hence, we have to revise both the `__init__` and `merge` methods:

```python
def __init__(self, num_elements):
    self.parent = [i for i in range(num_elements)]
    self.size = [1 for _ in range(num_elements)]
    self.rank = [0 for _ in range(num_elements)]

def merge(self, element1, element2):
    root1 = self.find(element1)
    root2 = self.find(element2)

    if root1 != root2:
        if self.rank[root1] > self.rank[root2]:
            self.parent[root2] = root1
            self.size[root1] += self.size[root2]
        else:
            self.parent[root1] = root2
            self.size[root2] += self.size[root1]

            if self.rank[root1] == self.rank[root2]:
                self.rank[root2] += 1
```

Prepared by ***Mark Edward M. Gonzales***

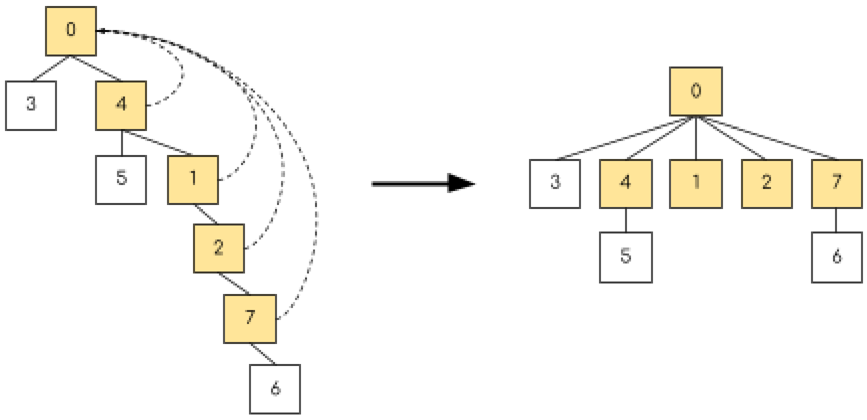
Although most textbook implementations use union by rank[4], an alternative is to perform union by size, especially if we would like to avoid the cost of creating another array just to store the ranks. In any case, both approaches yield the same worst-case time complexity; each find and merge operation runs in $\Theta(\log n)$ time.

If you are interested in proving the above-mentioned logarithmic time complexity, it may be helpful to start with this observation: If the trees S_1 and S_2 have heights h_1 and h_2, respectively, then attaching S_2 to S_1 results in a tree of height $\max\left(h_1, h_2 + 1\right)$.

💡 OPTIMIZATION #2: Path Compression

In the find operation, we "climb up" the tree to reach the root. Path compression takes the procedure up a notch by "flattening" the tree as we "climb up." Formally, suppose r is the root; we directly set the parent of each visited node to r, thereby shortening the path.

The visualization below illustrates the process of finding the root of the tree to which 7 belongs vis-à-vis the resulting tree after applying path compression (the visited nodes are shaded for emphasis):



Incorporating path compression can be accomplished by changing only a single line of code in our existing implementation:

```
def find(self, element):
    if self.parent[element] != element:
        self.parent[element] = self.find(self.parent[element])

    return self.parent[element]
```

Note that path compression shortens the height of the tree but does not change its rank, tying into our earlier remark that the rank is an overestimate of the height.

In a 1984 paper, Robert E. Tarjan and Jan van Leeuwen showed that, if we start with an empty disjoint-set data structure, an intermixed sequence of m find operations and $n - 1$ union operations (where $m \geq n$) has a time complexity of $O(m \log n)$ if path compression alone (i.e., without union by rank or size) is employed.

[4] This is mainly for pedagogical reasons. The time complexity analysis of union by rank is more instructional than that of union by size.

⌚ FINAL TIME COMPLEXITY

The task of computing the time complexity of the find and union operations when path compression is combined with union by rank (or size) is challenging. In fact, it relates to an interesting historical sidenote. Assuming an intermixed sequence of m find operations and $n - 1$ union operations (where $m \geq n$):

- In a 1972 paper, Michael J. Fisher (one of the proponents of the disjoint-set data structure) showed that its time complexity is $O(m \log \log n)$ time.
- In a 1973 paper, John E. Hopcraft and Jeffrey D. Ullman provided a better upper bound: $O(m \log^* n)$, where $\log^*$ (read as "log star") is the iterated logarithm.
- In a 1975 paper, Robert E. Tarjan derived an even tighter upper bound: $O(m\,\alpha(m, n))$, where $\alpha(m, n)$ is the inverse Ackermann function.

💭 Iterated Logarithm, Inverse Ackermann Function?

For a mathematical treatment of disjoint-set data structures, including formal proofs and definitions, you may refer to this slide deck from Stanford University and these lecture notes from the Massachusetts Institute of Technology.

Key Takeaways

Using path compression and union by rank (or size) brings the amortized time complexity of each operation down to $\Theta(\alpha(n))$, where n is the number of elements in the data structure and $\alpha(n)$ is the inverse Ackermann function[5]. Let us dissect this claim:

- In amortized time complexity[6], we look at the bigger picture. Instead of separately analyzing each operation, we average the cost in view of a *sequence* of operations. Even if a certain operation is expensive — as long as it is seldom performed — then its cost is "canceled out" over time.

  In our disjoint-set data structure, the individual time complexity of each operation is actually $O(\log n)$; thus, performing a series of n operations clocks a runtime of $O(n \log n)$. This is correct, but is this a *fair* assessment? Whenever we perform a find, path compression "flattens" the tree, thereby making succeeding operations faster. Amortized time complexity seeks to take this nuance into account.

- What is $\Theta(\alpha(n))$? Defining the inverse Ackermann function is outside the scope of this handout. For now, it suffices to know that this function's growth is remarkably, truly, and exceedingly slow to the point that it can be approximated as $O(1)$.

  In brief, for all practical purposes, the amortized time complexity of each find and union operation is $O(1)$. An almost-constant time complexity is the Holy Grail of computer science!

[5] In a 1989 paper, Michael Fredman and Michael Saks proved that this is the fastest possible amortized time complexity for *any* implementation of a disjoint-set data structure. This means that our implementation, which achieves this complexity, is as optimal as it can possibly be.

[6] The term is derived from the concept of *amortization* in accounting, where the value of an intangible asset or a loan is periodically lowered over time.

IMPLEMENTATION

Unlike priority queues, which are implemented via the `heapq` module, the Python Standard Library — as well as the C++ Standard Library and the Java Utility Package — does not include any built-in implementation of disjoint-set data structures.

In other words, whenever a task requires a disjoint-set data structure, we have to write our own implementation. It is, therefore, crucial to **understand** the logic and the optimization techniques that we discussed in the previous section.

⚠️ Caveat Against Memorization

While memorization can be helpful to a certain extent (after all, in a time-pressured contest, we have to code swiftly and conserve our energies for the more complex tasks), relying on rote learning and reducing programming to a "plug-and-play" of memorized algorithms foster a dangerous mindset that insults computational thinking.

For our exercises (and for your future programs), feel free to use, share, and modify our implementation of a disjoint-set data structure:

```
class DisjointSet:
    def __init__(self, num_elements):
        self.parent = [i for i in range(num_elements)]
        self.size = [1 for _ in range(num_elements)]
        self.rank = [0 for _ in range(num_elements)]

    def find(self, element):
        if self.parent[element] != element:
            self.parent[element] = self.find(self.parent[element])

        return self.parent[element]

    def merge(self, element1, element2):
        root1 = self.find(element1)
        root2 = self.find(element2)

        if root1 != root2:
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
                self.size[root1] += self.size[root2]
            else:
                self.parent[root1] = root2
                self.size[root2] += self.size[root1]

                if self.rank[root1] == self.rank[root2]:
                    self.rank[root2] += 1
```

ILLUSTRATIVE EXAMPLES

Let us now take a look at a selection of contest problems that creatively feature the usage of disjoint-set data structures.

Contest Virtual Friends

September 2008 University of Waterloo Programming Contest

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	Sample Output
1 3 Fred Barney Barney Betty Betty Wilma	2 3 4

📝 Discussion

To make our simulation more meaningful, let us augment the problem's sample input and add three new characters, namely `Jules`, `Jana`, and `Jill` (friends are marked with the same color):

Friendships	Social Networks					
Fred Barney	Fred	Barney				
Barney Betty	Fred	Barney	Betty			
Jules Jana	Fred	Barney	Betty	Jules	Jana	
Betty Wilma	Fred	Barney	Betty	Jules	Jana	Wilma
Jules Jill	Fred	Barney	Betty	Jules	Jana	Wilma	Jill

The visualization suggests that, if we view the social networks as disjoint sets, making friends is equivalent to the union operation. To expound a bit more, it may be helpful to point out that:

- The number of people in the social network of the two people who have just become friends is equal to the number of elements in the set to which they belong. As mentioned earlier in this handout, the size of the set to which an element `e` belongs is not `size[e]` but `size[find(e)]`.

- It is imperative to map each name to a unique integer. An efficient approach is to use a dictionary to store the name-integer mappings. If a name is not yet in the dictionary, insert it and map it to the number of items in the dictionary prior to its insertion. Otherwise, retains the name's original mapping.

  In the table below, a shaded row means that the name is already in the dictionary:

Names (Keys) in the Dictionary	Name Read	Integer Mapping
-	Fred	0
Fred	Barney	1
Fred, Barney	Barney	1
Fred, Barney	Betty	2
Fred, Barney, Betty	Jules	3
Fred, Barney, Betty, Jules	Jana	4
Fred, Barney, Betty, Jules, Jana	Betty	2
Fred, Barney, Betty, Jules, Jana	Wilma	5
Fred, Barney, Betty, Jules, Jana, Wilma	Jules	3
Fred, Barney, Betty, Jules, Jana, Wilma	Jill	6

The core method can be implemented as follows (`friendships` is a list of tuples, with the two people who have just become friends as the elements of each tuple):

```
def virtual_friends(friendships):
    social_networks = DisjointSet(200000)
    people = {}

    for friends in friendships:
        if friends[0] not in people:
            people[friends[0]] = len(people)
        if friends[1] not in people:
            people[friends[1]] = len(people)

        person1 = people[friends[0]]
        person2 = people[friends[1]]

        social_networks.merge(person1, person2)
        print(social_networks.size[social_networks.find(person1)])
```

Contest Graduation (Skolavslutningen)

2019 Programmeringsolympiadens onlinekval
(Swedish Olympiad in Informatics)

The school administration has encountered a problem with the upcoming school graduation, a problem which they hope you can help them resolve. During the graduation ceremony, students will stand in N rows with M students in each row. The administration wants the graduation to be as colorful as possible and will therefore hand out hats in different colors for the students.

For the lineup to look good, it's important that all the students in the same column have the same color of hat. So that no one feels left out, it is also important that all students in the same class have the same hat color. The row and column for each student has already been decided, but not the hat color. The administration needs your help with assigning hat colors to the students so that the graduation is as colorful as possible.

Write a program that, given how the students will be lined up, calculates the maximum number of unique hat colors that can be assigned to the students.

Input

The first line contains three integers N, M $(1 \leq N,\ M \leq 700)$ and K $(1 \leq K \leq 26)$ — the number of lines, the number of columns, and the number of classes.

Then N lines follow, each with M characters, describing how the students will be lined up in the graduation. The character in row i and column j is an uppercase letter between A and the K^{th} letter of the alphabet — the class to which the student in row i, column j belongs. It is guaranteed that each class has at least one student.

Output

Print an integer — the maximum number of unique hat colors that can be assigned to the students so that all the students in the same column have the same hat color, and also all students in the same class have the same hat color.

Sample Input	Sample Output

`2 3 2`
`AAB`
`ABB`
	`1`

`2 2 3`
`AC`
`BC`
	`2`

`2 3 3`
`ABC`
`ABC`
	`3`

`3 5 5`
`ABECE`
`BCDAE`
`CADBD`
	`2`

Prepared by ***Mark Edward M. Gonzales***

Explanation of Sample Cases

In the first sample case, the second column has one student from class A and one student from class B. Since both of these students must have the same hat color, everyone in class A must have the same color hat as everyone in class B. The conclusion is that all students must have the same color, so the answer is 1.

In the second sample case, classes A and B must have the same color, as there is a student from each of these two classes in the first column. Class C, on the other hand, can use a different color on their hats. Therefore, the answer is 2.

In the third sample case, we can give each class its own color, since there are no two students from different classes in the same column. Thus, the answer is 3.

In the last sample case, we can assign one color to all students from classes A, B and C, and a different color to all students from classes D and E. So the answer is 2.

📝 **Discussion**

We frame the problem so that students with the same hat color belong to the same set.

Ergo, satisfying the condition that students in the same column share the same hat color translates to taking the union of the students belonging to the same column. To maximize the number of unique hat colors, columns are treated as disjoint sets; they are merged with other columns (sets) only if there are students belonging to the same class.

Using the last sample input, let us run a simulation of this idea, which also prods us to process the input in column-wise fashion:

Column	Operation	Result
1	`union(A, B)`	A B
	`union(B, C)`	A B C
2	`union(B, C)`	A B B C C
	`union(C, A)`	A B B C C A
3	`union(E, D)`	A B E B C D C A
	`union(D, D)`	A B E B C D C A D
4	`union(C, A)`	A B E C B C D A C A D
	`union(A, B)`	A B E C B C D A C A D B
5	`union(E, E)`	A B E C E B C D A E C A D B
	`union(E, D)`	A B E C E B C D A E C A D B D

The maximum number of unique colors is equal to the number of disjoint sets in our data structure. We can either modify our implementation of the `DisjointSet` class to include an attribute for keeping track of the number of disjoint sets, or opt to create a method similar to the one below:

```python
def get_num_disjoint_sets(data_structure):
    disjoint_sets = set()
    for element in data_structure.parent:
        disjoint_sets.add(data_structure.find(element))

    return len(disjoint_sets)
```

Similar to the previous problem, we have to map the classes to integers. Since the classes are characters, the easiest way would be to take the difference between their ASCII[7] value and the ASCII value of `A`. For instance, the ASCII value of `A` is 65, so it maps to 65 – 65 = 0. Meanwhile, the ASCII value of `Z` is 90, so it maps to 90 – 65 = 25.

In Python, the ASCII value of a character can be obtained via the `ord()` function. The core method can, thus, be implemented as follows (`students` is a list storing each line of input, excluding the first line):

```python
def graduation(students, num_classes):
    num_rows = len(students)
    num_columns = len(students[0])

    colors = DisjointSet(num_classes)

    for j in range(num_columns):
        for i in range(1, num_rows):
            student1 = ord(students[i][j]) - ord('A')
            student2 = ord(students[i - 1][j]) - ord('A')

            colors.merge(student1, student2)

    return get_num_disjoint_sets(colors)
```

The code above has a small peculiarity. Why do we have to pass the value of `num_classes` as an argument? Would it be incorrect to simply initialize the number of singletons in the disjoint-set data structure `colors` to 26 (the maximum number of classes)?

[7] ASCII stands for the American Standard Code for Information Interchange. Here is a copy of the ASCII table.

Contest Swap to Sort

2018 ICPC Singapore Preliminary Contest
(International Collegiate Programming Contest)

You are given an array $A[1 \ldots N]$ with integers in decreasing order and a list of pairs (a_1, b_1), (a_2, b_2), …, (a_k, b_k). You wish to sort the array A in increasing order, each turn you choose an i (i can be chosen multiple times) and swap $A[a_i]$ with $A[b_i]$. Determine whether this is possible.

Input

The first line contains two integers, representing N and K respectively ($1 \le N, K \le 10^6$).
The following K lines each contains two integers, representing a_i and b_i respectively ($1 \le a_i < b_i \le N$).

Output

Output "`Yes`" if it is possible to sort the array in increasing order, "`No`" otherwise.

Sample Input	Sample Output

5 2
1 5
2 4
	Yes

5 4
1 4
2 3
4 5
1 5
	No

📝 Discussion

Consider the task of transforming the list $A = [9, 7, 4, 3, 2, 1]$ to $[1, 2, 3, 4, 7, 9]$. The most straightforward way is shown below (note that the indexing starts at 1):

Operation	Result					
Swap $A[1]$ and $A[6]$.	1	7	4	3	2	9
Swap $A[2]$ and $A[5]$.	1	2	4	3	7	9
Swap $A[3]$ and $A[4]$.	1	2	3	4	7	9

The key insight is that it is possible to sort the list in increasing order only if:

- $A[1]$ can be swapped with $A[N]$.
  The first integer can be swapped with the last integer.

- $A[2]$ can be swapped with $A[N-1]$.
  The second integer can be swapped with the second to last integer.

and so on …

Claim 1. It is possible to sort the list in increasing order only if $A[p]$ can be swapped with $A[N - p + 1]$ for $p = 1, 2, 3, \ldots, \lfloor \frac{N}{2} \rfloor$. We only need to consider up to $\lfloor \frac{N}{2} \rfloor$ since swapping is symmetric.

To clarify, the swapping does not have to be direct — we can take a roundabout sequence of swaps as long as, in the end, the $A[p]$ can be swapped with the $A[N - p + 1]$. Consider the following sample input (without the first line) and simulation:

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

Operation	Result					
Swap $A[1]$ and $A[6]$.	1	7	4	3	2	9
Swap $A[2]$ and $A[4]$.	1	3	4	7	2	9
Swap $A[2]$ and $A[3]$.	1	4	3	7	2	9
Swap $A[2]$ and $A[5]$.	1	2	3	7	4	9
Again: Swap $A[2]$ and $A[4]$.	1	7	3	2	4	9
Again: Swap $A[2]$ and $A[5]$.	1	4	3	2	7	9
Again: Swap $A[2]$ and $A[4]$.	1	2	3	4	7	9

It may be easy to miss that we are allowed to repeat steps. As stipulated in the problem description: "... each turn you choose an i (i can be chosen multiple times) and swap $A[a_i]$ with $A[b_i]$" (emphasis added).

How does this relate to disjoint-set data structures? Admittedly, the connection may not be too intuitive, so it might be better to take a more investigative or inductive approach. Focus on the 4th row of the preceding table (the "cut-off" of the sample input):

Swap $A[2]$ and $A[5]$.	1	2	3	7	4	9

For the list to be increasing, the objective is to be able to swap $A[4] = 7$ with $A[5] = 4$:

- **Which of our allowed swaps feature $A[4]$ and $A[5]$?** There are two: swap $A[2]$ and $A[4]$, and swap $A[2]$ and $A[5]$.

- **How do we know that it is possible to swap $A[4]$ and $A[5]$ using our allowed swaps?** This is the part where things get tricky. Observe that the pertinent allowed swaps are "linked" together by $A[2]$. If they were entirely "disjoint," then it would clearly be impossible to establish a sequence of steps to swap $A[4]$ **and** $A[5]$.

  One way to frame this observation is that the indices, namely 2, 4, and 5, belong to the same set. Pursuing this idea further, swapping $A[a_i]$ and $A[b_i]$ can be thought of as taking the union of the sets to which a_i and b_i belong.

Claim 2. $A[p]$ can be swapped with $A[q]$ only if:

- The input explicitly tells us to swap $A[p]$ with $A[q]$; or
- p and q belong to the same set.

Before proceeding, try to convince yourself of the veracity of this claim and experiment with some more test cases.

By combining claims 1 and 2, the core method can be implemented as follows (`swaps` is a list of tuples, with a_i and b_i as the elements of each tuple):

```python
def swap_to_sort(swaps, N):
    swappable_indices = DisjointSet(N + 1)

    for (a_i, b_i) in swaps:
        swappable_indices.merge(a_i, b_i)

    can_swap_to_sort = "Yes"
    for p in range(1, N // 2 + 1):
        if swappable_indices.find(p) != swappable_indices.find(N - p + 1):
            can_swap_to_sort = "No"
            break

    return can_swap_to_sort
```

***Sidenote**: During the actual contest, only 61 out of 158 teams (roughly 38.61%) managed to submit a correct solution that executes within the 2-second runtime limit.*

🧑‍💻 EXERCISES

1. Kruskal's Algorithm. Proposed by the American mathematician Joseph B. Kruskal in a 1956 paper published in the *Proceedings of the American Mathematical Society*, it is used to find the minimum spanning tree of a weighted undirected graph. Write an efficient implementation of Kruskal's algorithm using a disjoint-set data structure.

   For a high-level walkthrough (as well as some hints), you may refer to pages 71 to 103 of this slide deck from Stanford University.

2. Network Connections (University of Valladolid [UVa] Online Judge 793)
3. Bridges and Tunnels (September 2011 University of Waterloo Programming Contest)
4. Association for Control Over Minds (2015 ACM ICPC Singapore Regional Contest)
5. Almost Union-Find (Rujia Liu's Present 3: A Data Structure Contest Celebrating the 100th Anniversary of Tsinghua University)

Exercises 2 to 5 were selected from the list of practice problems suggested by S. Halim and F. Halim in *Competitive Programming 3: The New Lower Bound of Programming Contests.*

-- return 0; --

*Prepared by **Mark Edward M. Gonzales***