Question - 1

Tom & Jerry in a Maze

After decades of chasing Jerry, Tom wants to make peace. Being skeptical, Jerry hides in a $n \times m$ maze. Tom decides the best way to make Jerry take him seriously is to collect all of the cheese pieces in the maze and give them to Jerry as a gift. A *move* is considered to be a single step from the current position to some adjacent position in the maze. Tom can move in all four directions within the maze: up (↑), down (↓), left (←), and right (→).

Each cell in *maze* is labeled as follows:

- A path cell is represented by a *0*.
- A blocked cell (wall) is represented by a *1*.
- A cheese cell is represented by a *2*. Tom can move through a cheese cell just as he would move through a regular path cell.

Tom is initially positioned at the left topmost cell with coordinates *(0, 0)*.

Complete the *minMoves* function in your editor. It has *3* parameters:

1. A *2D* array of integers, *maze*, denoting the maze where Jerry is hiding.
2. An integer, *x*, denoting the *x*-coordinate for Jerry's location.
3. An integer, *y*, denoting the *y*-coordinate for Jerry's location.

Your function must return an integer denoting the minimum number of moves that it will take for him to collect all the cheese and deliver it to Jerry at *(x, y)*; if the task is not possible, return *−1*.

Constraints

- $1 \le n, m \le 100$
- *0 ≤ the number of cheese pieces ≤ 10*
- $1 \le x \le n$
- $1 \le y \le m$

▸ **Input Format For Custom Testing**

▾ **Sample Case 0**

Sample Input For Custom Testing

maze = {{0, 2, 0}, {0, 0, 1}, {1, 1, 1}}

x = 1

y = 1

Sample Output

```
2
```

Explanation

The shortest path Tom can take to pick up all the cheese and deliver it to Jerry is *(0, 0) → (0, 1) → (1, 1)*. Because this involves *2* moves, we return *2*.

▸ **Sample Case 1**

▸ **Sample Case 2**

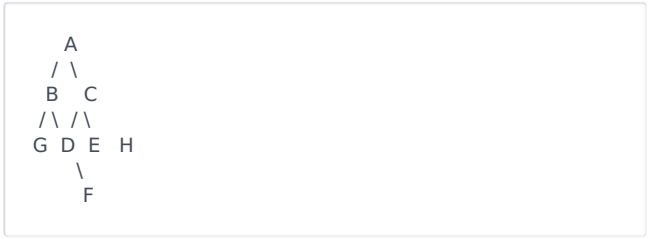
Question - 2

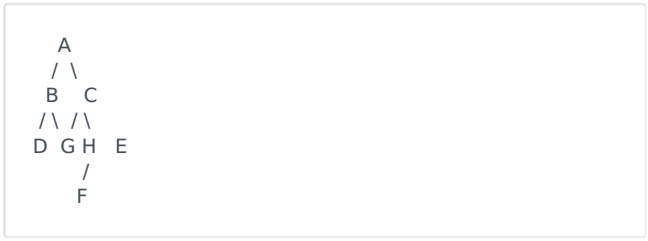
Is this a tree?

You are given a binary tree as a sequence of parent-child pairs. For example, the tree represented by the node pairs below:

```
(A,B) (A,C) (B,G) (C,H) (E,F) (B,D) (C,E)
```

may be illustrated in many ways, with two possible representations below:





Below is the recursive definition for the S-expression of a tree:

```
S-exp(node) = ( node->val (S-exp(node->first_child))
(S-exp(node->second_child))), if node != NULL
                    = "", node == NULL
  where, first_child->val < second_child->val
(lexicographically smaller)
```

This tree can be represented in a S-expression in multiple ways. The lexicographically smallest way of expressing this is as follows:

```
(A(B(D)(G))(C(E(F))(H)))
```

We need to translate the node-pair representation into an S-expression (lexicographically smallest one), and report any errors that do not conform to the definition of a binary tree.

The list of errors with their codes is as follows:

Error Code	Type of error
E1	More than 2 children
E2	Duplicate Edges
E3	Cycle present
E4	Multiple roots
E5	Any other error

Input Format:
Input will consist of on line of parent-child pairs. Each pair consists of two node names separated by a single comma and enclosed in parentheses. A single space separates the pairs.

Output:
Output the given tree in the S-expression representation described above.
There should be no spaces in the output.

Constraints:
1. There is no specific sequence in which the input (parent,child) pairs are represented.

2. The name space is composed of upper case single letter (A-Z) so the maximum size is 26 nodes.
3. Error cases are to be tagged in the order they appear on the list. For example, if one input pair raises both error cases 1 and 2, the output must be E1.

Sample Input #00

```
(B,D) (D,E) (A,B) (C,F) (E,G) (A,C)
```

Sample Output #00

```
(A(B(D(E(G))))(C(F)))
```

Sample Input #01

```
(A,B) (A,C) (B,D) (D,C)
```

Sample Output #01

```
E3
```

Explanation
Node **D** is both a child of **B** and a parent of **C,** but **C** and **B** are both child nodes of **A**. Since **D** tries to attach itself as parent to a node already above it in the tree, this forms a cycle.

Question - 3
Approximate Matching

Consider three strings: *text*, *prefixString*, and *suffixString*. For each substring, *sub*, of *text*, we define the *text_score* as follows:

- *prefix_score = the highest n* such that the first *n* characters of *sub* are equal to the last *n* characters of *prefixString* and occur in the same exact order.
- *suffix_score = the highest n* such that last *n* characters of *sub* are equal to the first *n* characters of *suffixString* and occur in the same exact order.
- *text_score = prefix_score + suffix_score*

For example, if *sub = "nothing"*, *prefixString = "bruno"*, and *suffixString = "ingenious"*:

- *prefix_score = 2* because *sub = "nothing"* and *prefixString = "bruno"* both have a substring, *"no"*, that is common to the beginning of *sub* and the end of *prefixString*, and *"no"* has length *n = 2*.
- *suffix_score = 3* because *sub = "nothing"* and *suffixString = "ingenious"* both have a substring, *"ing"*, that is common to the end of *sub* and the beginning of *suffixString*, and *"ing"* has length *n = 3*.
- *text_score = prefix_score + suffix_score = 2 + 3 = 5*

Complete the *calculateScore* function in the editor below. It has three parameters:

1. A string, *text*.
2. A string, *prefixString*.
3. A string, *suffixString*.

The function must return a string denoting the non-empty substring of *text* having a maximal *text_score*. If there are multiple such substrings, choose the lexicographically smallest substring.

Input Format

Locked stub code in the editor reads the following input from stdin and passes it to the function:
The first line contains a string denoting *text*.
The second line contains a string denoting *prefixString*.
The third line contains a string denoting *suffixString*.

Constraints

- *text*, *prefix*, and *suffix* contain lowercase English alphabetic letters (i.e., *a* through *z*) only.
- *1 ≤ |text|, |prefix|, |suffix| ≤ 50*, where *|s|* denotes the number of characters in some string *s*.
- It is guaranteed that there will always be a substring of *text* that, at minimum, matches either of the following:
  - One or more characters at the end of *prefixString*.
  - One or more characters at the beginning of *suffixString*.

Sample Input 0

```
nothing
bruno
ingenious
```

Sample Output 0

```
nothing
```

Explanation 0
This example is explained above.

Sample Input 1

```
ab
b
a
```

Sample Output 1

```
a
```

Explanation 1
Given *text = "ab"*, our possible substrings are *sub = "a"*, *sub = "b"*, and *sub = "ab"*.

- *sub = "a"*
  - *prefixString = "b"*: The beginning of *sub* doesn't match the end of *prefixString*, so *prefix_score = 0*.
  - *suffixString = "a"*: The last character of *sub* matches the first character of *suffixString*, so *suffix_score = 1*.
  - *text_score = prefix_score + suffix_score = 0 + 1 = 1*
- *sub = "b"*
  - *prefixString = "b"*: The first character of *sub* matches the last character of *prefixString*, so *prefix_score = 1*.
  - *suffixString = "a"*: The end of *sub* doesn't match the beginning of *suffixString*, so *suffix_score = 0*.
  - *text_score = prefix_score + suffix_score = 1 + 0 = 1*
- *sub = "ab"*
  - *prefixString = "b"*: The beginning of *sub* doesn't match the end of *prefixString*, so *prefix_score = 0*.
  - *suffixString = "a"*: The last character of *sub* matches the first character of *suffixString*, so *suffix_score = 1*.
  - *text_score = prefix_score + suffix_score = 0 + 1 = 1*

All of these have a *text_score* of *1*, so we return the lexicographically smallest one (i.e., *"a"*).

Question - 4
Zombie Clusters

There are zombies in Seattle. Liv and Ravi are trying to track them down to find out who is creating new zombies in an effort to prevent an apocalypse. Other than the patient-zero zombies (who became so by mixing MaxRager and tainted Utopium), new people only become zombies after being scratched by an existing zombie. Zombiism is transitive. This means that if zombie *0* knows zombie *1* and zombie *1* knows zombie *2*, then zombie *0* is connected to zombie *2* by way of knowing zombie *1*. A zombie *cluster* is a group of zombies who are directly or indirectly linked through the other zombies they know, such as the one who scratched them or supplies who them with brains.

The diagram showing connectedness will be made up of a number of binary strings, characters 0 or 1. Each of the characters in the string represents whether the zombie associated with a row element is connected to the zombie at that character's index. For instance, a zombie 0 with a connectedness string '110' is connected to zombies 0 (itself) and zombie 1, but not to zombie 2. The complete matrix of zombie connectedness is:

```
110
110
001
```

Zombies 0 and 1 are connected. Zombie 2 is not.

Your task is to determine the number of connected groups of zombies, or *clusters*, in a given matrix.

Note: Method signatures may vary depending on the requirements of your chosen language.

Function Description

Complete the function *zombieCluster* in the editor below. The function must return an integer representing the number of zombie clusters counted.

zombieCluster has the following parameter(s):

zombies[z_0,...z_{n-1}]: an array of strings of binary digits z_i representing connectedness of zombies

Constraints

- $1 \leq n \leq 300$
- $0 \leq i < n$
- $|zombies| = n$
- Each z_i contains a binary string of n zeroes and ones. It is a square matrix.

▶ **Input Format for Custom Testing**

▼ **Sample Case 0**

Sample Input 0

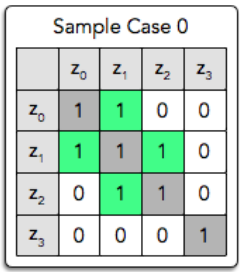
```
4
1100
1110
0110
0001
```

Sample Output 0

```
2
```

In the diagram below, the squares highlighting a known connection between two different zombies are highlighted in green. Because each zombie is already aware that they are personally a zombie, those are highlighted in grey.

Explanation 0



We have $n = 4$ zombies numbered z_0 through z_3. There are 2 pairs of zombies who directly know each another: (z_0, z_1) and (z_1, z_2). Because of zombiism's transitive property, the set of zombies $\{z_0, z_1, z_2\}$ is considered to be a single zombie cluster. The remaining zombie, z_3, doesn't know any other zombies and is considered to be his own, separate zombie cluster ($\{z_3\}$). This gives us a total of 2 zombie clusters.

▸ **Sample Case 1**

Question - 5

IP Address Validation

IPv4 was the first publicly used Internet Protocol. It used *4*-byte addresses and permitted 2^{32} distinct values. The typical format for an *IPv4* address is *A.B.C.D* where *A*, *B*, *C*, and *D* are integers in the inclusive range between *0* and *255*.

IPv6, with *128* bits, was developed to permit the expansion of the address space. These addresses are represented by eight colon-separated sixteen-bit groups, where each sixteen-bit group is written using *1* to *4* hexadecimal digits. Leading zeroes in a section are often omitted from an address, meaning that the groups *0* is identical to *0000* and group *5* is identical to *0005*. Some examples of valid *IPv6* addresses are `2001:0db8:0000:0000:0000:ff00:0042:8329` and `3:0db8:0:01:F:ff0:0042:8329`.

Given *n* strings of text that *may or may not be* valid Internet Protocol (IP) addresses, we want to determine whether each string of text is:

- An *IPv4* address.
- An *IPv6* address.
- *Neither* an *IPv6* address nor an *IPv4* address.

Complete the *checkIPs* function in the editor below. It has one parameter: an array of strings, *ip_array*, where each element *i* denotes a string of text to be checked. It must return an array of strings where each element *i* contains the answer for ip_i; each answer must be whichever of the following case-sensitive terms

is appropriate:

- `IPv4` if the string is a valid *IPv4* address.
- `IPv6` if the string is a valid *IPv6* address.
- `Neither` if the string is not a valid *IPv4* or *IPv6* address.

Input Format

Locked stub code in the editor reads the following input from stdin and passes it to the function:
The first line contains an integer, *n*, denoting the number of elements in *ip_array*.
Each line *i* of the *n* subsequent lines (where $0 \leq i < n$) contains a string describing ip_i.

Constraints

- $1 \leq n \leq 50$
- $1 \leq ip_i \leq 500$
- It is guaranteed that any string containing a valid *IPv4* or *IPv6* address has no leading or trailing whitespace.

Output Format

The function must return an array of strings where each element *i* contains the string `IPv4`, `IPv6`, or `Neither`, denoting that ip_i was an *IPv4* address, an *IPv6* address, or *Neither* (i.e., not an address at all). This is printed to stdout by locked stub code in the editor.

Sample Input 0

```
2
This line has junk text.
121.18.19.20
```

Sample Output 0

```
Neither
IPv4
```

Explanation 0

We must check the following $n = 2$ strings:

1. ip_0 = *"This line has junk text."* is not a valid *IPv4* or *IPv6* address, so we return `Neither` in index *0* of our return array.
2. ip_1 = *"121.18.19.20"* is a valid *IPv4* address, so we return `IPv4` in index *1* of our return array.

Sample Input 1

```
1
2001:0db8:0000:0000:0000:ff00:0042:8329
```

Sample Output 1

```
IPv6
```

Explanation 1

We only have $n = 1$ value to check. Because ip_0 = *"2001:0db8:0000:0000:0000:ff00:0042:8329"* is a valid *IPv6* address, we return `IPv6` in index *0* of our return array.

Question - 6

Cutting Metal Surplus

The owner of a metal rod factory has a surplus of rods of arbitrary lengths. A local contractor offers to buy any of the factory's surplus as long as all the rods have the same exact

length, referred to as *saleLength*. The factory owner can increase the number of sellable rods by cutting each rod zero or more times, but each cut has a cost denoted by *costPerCut* and any leftover rods having a length other than *saleLength* must be discarded for no profit. The factory owner's total profit for the sale is calculated as:

$$totalProfit = totalUniformRods \times saleLength \times salePrice - totalCuts \times costPerCut$$

where *totalUniformRods* is the number of sellable rods, *saleLength* is the uniform length of the rods being sold,*salePrice* is the per-rod price that the contractor agrees to pay, and *totalCuts* is the total number of times the rods needed to be cut.

Complete the function in the editor below. It has three parameters:

1. An integer, *costPerCut*, denoting the cost incurred each time any rod is cut.
2. An integer, *salePrice*, denoting the amount of money the contractor will pay for each rod of length *saleLength*.
3. An array of *n* integers, *lengths*, where the value of each element *i* denotes the initial length of a metal rod.

The function must find the optimal *saleLength* such that the factory owner's profit is maximal, and then return an integer denoting the maximum possible profit.

Input Format

Locked stub code in the editor reads the following input from stdin and passes it to the function:
The first line contains an integer denoting *costPerCut*.
The second line contains an integer denoting *salePrice*.
The third line contains an integer, *n*, denoting the number of elements in *lengths*.
Each line *i* of the *n* subsequent lines (where $0 \le i < n$) contains an integer describing $lengths_i$.

Constraints

- $1 \le n \le 50$
- $1 \le lengths_i \le 10^4$
- $1 \le salePrice, costPerCut \le 1000$

Output Format

The function must return an integer denoting the maximum possible profit the factory owner can make from the sale. This is printed to stdout by locked stub code in the editor.

Sample Input 0

```
1
10
3
26
103
59
```

Sample Output 0

```
1770
```

Explanation 0

Since *costPerCut = 1* is very inexpensive, a large number of cuts can be made to reduce the number of wasted pieces. The optimal rod length for maximizing profit is *6*, and the rods are cut like so:

- `lengths[0] = 26`: Cut off a piece of length *2* and discard it, resulting in a rod of length *24*. Then cut this rod into *4* pieces of length *6*.
- `lengths[1] = 103`: Cut off a piece of length *1* and discard it, resulting in a rod of length *102*. Then cut this rod into *17*

pieces of length *6*.
- `lengths[2] = 59` : Cut off a piece of length *5* and discard it, resulting in a rod of length *54*. Then cut this rod into *9* pieces of length *6*.

After performing *totalCuts = (1 + 3) + (1 + 16) + (1 + 8) = 30* cuts, there are *totalUniformRods = 4 + 17 + 9 = 30* pieces of length *saleLength = 6* that can be sold at *salePrice = 10* dollars. This yields a total profit of *salePrice × totalUniformRods × saleLength − totalCuts × costPerCut = 10 × 30 × 6 − 30 × 1 = 1770*, so the function returns *1770*.

Sample Input 1

```
100
10
3
26
103
59
```

Sample Output 1

```
1230
```

Explanation 1

Since *costPerCut = 100*, cuts are expensive and must be minimal. The optimal rod length for maximizing profit is *51*, and the rods are cut like so:
- `lengths[0] = 26` : Discard this rod entirely.
- `lengths[1] = 103` : Cut off a piece of length *1* and discard it, resulting in a rod of length *102*. Then cut this rod into *2* pieces of length *51*.
- `lengths[2] = 59` : Cut off a piece of length *8* and discard it, resulting in a rod of length *51*.

After performing *totalCuts = (0) + (1 + 1) + (1) = 3* cuts, there are *totalUniformRods = 0 + 2 + 1 = 3* pieces of length *saleLength = 51* that can be sold at *salePrice = 10* dollars each. This yields a total profit of *salePrice × totalUniformRods × saleLength − totalCuts × costPerCut = 10 × 3 × 51 − 3 × 100 = 1230*, so the function returns *1230*.

Question - 7

Adorable Strings

We consider a string consisting of one or more lowercase English alphabetic letters (`[a-z]`), digits (`[0-9]`), colons (`:`), forward slashes (`/`), and backward slashes (`\`) to be *adorable* if the following conditions are satisfied:
- The first letter of the string is a lowercase English letter.
- Next, it contains a sequence of *zero or more* of the following characters: lowercase English letters, digits, and colons.
- Next, it contains a forward slash.
- Next, it contains a sequence of *one or more* of the following characters: lowercase English letters and digits.
- Next, it contains a backward slash.
- Next, it contains a sequence of *one or more* lowercase English letters.

Given some string, *s*, we define the following:
- *s[i..j]* is a substring consisting of all the characters in the inclusive range between index *i* and index *j* (i.e., *s[i]*, *s[i + 1]*, *s[i + 2]*, ..., *s[j]*).
- Two substrings, $s[i_1..j_1]$ and $s[i_2..j_2]$, are said to be *distinct* if either $i_1 \neq i_2$ or $j_1 \neq j_2$.

Complete the *adorableCount* function in the editor below. It has

one parameter: an array of *n* strings, *words*. The function must return an array of *n* positive integers where the value at each index *i* denotes the total number of *distinct*, *adorable substrings* in $words_i$.

Input Format
Locked stub code in the editor reads the following input from stdin and passes it to the function:
The first line contains an integer, *n*, denoting the number of elements in *words*.
Each line *i* of the *n* subsequent lines (where $0 \leq i < n$) contains a string describing $words_i$.

Constraints
- $1 \leq n \leq 50$
- Each $words_i$ consists of one or more of the following characters: lowercase English alphabetic letters (`[a-z]`), digits (`[0-9]`), colons (`:`), forward slashes (`/`), and backward slashes (`\`) only.
- The length of each $words_i$ is no more than 5×10^5.

Output Format
The function must return an array of *n* positive integers where the integer at each index *i* denotes the total number of *distinct*, *adorable* substrings in $words_i$. This is printed to stdout by locked stub code in the editor.

Sample Input 0

```
6
w\\/a/b
w\\/a\b
w\\/a\b
w::/a\b
w::/a\b
w:/a\bc::/12\xyz
```

Sample Output 0

```
0
0
0
0
1
8
```

Explanation 0
Let's call our return array *ret*. We fill *ret* as follows:
- *word = "w\\/a/b"* has no adorable substring, so *ret[0] = 0*.
- *word = "w\\/a\b"* has no adorable substring, so *ret[1] = 0*.
- *word = "w\\/a\b"* has no adorable substring, so *ret[2] = 0*.
- *word = "w::/a\b"* has no adorable substring, so *ret[3] = 0*.
- *word = "w::/a\b"* has one adorable substring, *word[0..6] = "w::/a\b"*, so *ret[4] = 1*.
- *word = "w:/a\bc::/12\xyz"* has the following eight adorable substrings:
  1. *word[0..5] = w:/a\b*
  2. *word[0..6] = w:/a\bc*
  3. *word[5..13] = bc::/12\x*
  4. *word[5..14] = bc::/12\xy*
  5. *word[5..15] = bc::/12\xyz*
  6. *word[6..13] = c::/12\x*
  7. *word[6..14] = c::/12\xy*
  8. *word[6..15] = c::/12\xyz*

  This means *ret[5] = 8*.

We then return *ret = [0, 0, 0, 0, 1, 8]*

Question - 8
Matching Tokens

We define the following:

- There are *friends_nodes* friends numbered from *1* to *friends_nodes*.
- There are *friends_edges* pairs of friends, where each (x_i, y_i) pair of friends is connected by a shared integer *token* described by $friends\_weight_i$.
- Any two friends, x_i and y_i, can be connected by zero or more tokens because if friends x_i and y_i share token t_i and friends y_i and z_i also share token t_i, then x_i and z_i are also said to share token t_i.

Find the maximal product of x_i and y_i for any directly or indirectly connected (x_i, y_i) pair of friends such that x_i and y_i share the maximal number of tokens with each other.

Complete the *maxTokens* function in the editor. It has four parameters:

Name	Type	Description
friends_nodes	integer	The number of friends.
friends_from	integer array	Each *friends_from[i]* (where $0 \le i <$ *friends_edges*) denotes the first friend in pair *(friends_from[i], friends_to[i])*.
friends_to	integer array	Each *friends_to[i]* (where $0 \le i <$ *friends_edges*) denotes the second friend in pair *(friends_from[i], friends_to[i])*.
friends_weight	integer array	Each *friends_weight[i]* (where $0 \le i <$ *friends_edges) denotes the ID number of a token shared by both friends_from[i] and friends_to[i].*
Note: *friends_edges* is the number of pairs of friends that directly share a token.		

The function must return an integer denoting the maximal product of x_i and y_i such that x_i and y_i are a pair of friends that share the maximal number of tokens with each other.

Input Format
The first line contains two space-separated integers describing the respective values of *friends_nodes* and *friends_edges*.
Each line *i* of the *friends_edges* subsequent lines (where $0 \le i <$ *friends_edges*) contains three space-separated integers describing the respective values of $friends\_from_i$, $friends\_to_i$, and $friends\_weight_i$.

Constraints

- $2 \le friends\_nodes \le 100$
- $1 \le friends\_edges \le min(200, (friends\_nodes \times (friends\_nodes - 1)) / 2)$
- $1 \le friends\_weight_i \le 100$
- $1 \le friends\_from_i, friends\_to_i \le friends\_nodes$

- $1 \leq friends\_weight_i \leq friends\_edges$
- $friends\_from_i \neq friends\_to_i$
- Each pair of friends can be connected by zero or more types of tokens.

Output Format

Return an integer denoting the maximal product of x_i and y_i such that x_i and y_i are a pair of friends that share the maximal number of tokens with each other.

Sample Input 0

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

Sample Output 0

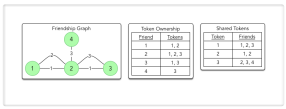
```
6
```

Explanation 0

Each pair of $n = 4$ friends is connected by the following tokens:

- Pair *(1, 2)* shares *2* tokens (i.e., tokens *1* and *2*)
- Pair *(1, 3)* shares *1* token (i.e., token *1*)
- Pair *(1, 4)* shares *0* tokens
- Pair *(2, 3)* shares *2* tokens (i.e., tokens *1* and *3*)
- Pair *(2, 4)* shares *1* token (i.e., token *3*)
- Pair *(3, 4)* shares *1* token (i.e., token *3*)

The pairs connected by the maximal number of tokens are *(1, 2)* and *(2, 3)*. Their respective products are $1 \times 2 = 2$ and $2 \times 3 = 6$. We then return the largest of these values as our answer, which is *6*.



Question - 9

Minimum Weight Path in a Directed Graph

We define a *directed graph*, *g*, such that:

- The total number of nodes in the graph is *g_nodes*.
- The nodes are numbered sequentially as *1, 2, 3, ..., g_nodes*.
- The total number of edges in the graph is *g_edges*.
- Each edge connects two distinct nodes (i.e., no edge connects a node to itself).

- The weight of the edge connecting nodes *g_from[i]* and *g_to[i]* is *g_weight[i]*.
- The edge connecting nodes *g_from[i]* and *g_to[i]* is directed. In other words, it describes a path only in the direction *g_from[i]* → *g_to[i]*.

We define the *weight* of a path from node *1* to node *g_nodes* to be the sum of all edges traversed on that path.

Complete the *minCost* function in the editor below. It has four parameters:

1. An integer, *g_nodes*, denoting the number of nodes in graph *g*.
2. An array of integers, *g_from*, where each *g_from[i]* denotes the starting (source) node of the i^{th} directed edge in graph *g*.
3. An array of integers, *g_to*, where each *g_to[i]* denotes the ending (target) node of the i^{th} directed edge in graph *g*.
4. An array of integers, *g_weight*, where each *g_weight[i]* denotes the weight of the i^{th} directed edge in graph *g*.

You must find the path from node *1* to node *g_nodes* having the minimum possible weight. You can add extra directed edges having weight *1* (one) between any two distinct nodes that are not already connected by an edge. The function must return an integer denoting the minimum possible weight of any path from node *1* to node *g_nodes*.

Input Format

Locked stub code in the editor reads the following input from stdin and passes it to the function:
The first line contains two space-separated integers describing the respective values of *g_nodes* and *g_edges*.
Each line *i* of the *g_edges* subsequent lines contains three space-separated integers describing the respective values of *g_from[i]*, *g_to[i]*, and *g_weight[i]*.

Constraints

- $3 \le g\_nodes \le 10^3$
- $1 \le g\_edges \le min(10^4, {}^{(g\_nodes \times (g\_nodes - 1))}/_{2})$
- $1 \le g\_weight[i] \le 10^6$

Output Format

The function must return an integer denoting the minimum weight of any possible path (including one created by adding the optional additional directed edges) from node *1* to node *g_nodes*. This is printed to stdout by locked stub code in the editor.

Sample Input 0

```
2 1
1 2 3
```

Sample Output 0

```
3
```

Explanation 0

A directed edge already exists from node *1* to node *2* and the path *1* → *2* is the minimum cost path, so the function returns *3*.

Sample Input 1

```
3 1
1 2 3
```

Sample Output 1

```
1
```

Explanation 1
As graph *g* has no edge between node *1* and node *3*, we can add an extra edge from node *1* to node *3* having weight *1*. Thus, the path *1 → 3* is the minimum weight path and the function returns *1*.

Sample Input 2

```
4 4
1 2 3
1 3 3
1 4 3
2 1 3
```

Sample Output 2

```
3
```

Explanation 2
A directed edge already exists from node *1* to node *4* and the path *1 → 4* is the minimum cost path, so the function returns *3*.

Question - 10
Money Collection

Julia is collecting money from *n* classmates for a trip. Each classmate is assigned a unique ID number from *1* to *n*, and each classmate *i* is prepared to donate exactly *i* dollars to the trip fund. For example, classmate *1* will donate *1* dollar, classmate *3* will donate *3* dollars, and so on. Julia plans to visit each classmate *in order* (i.e., from *1* to *n*) to collect their money, but she may also *refuse* to take their money because she is superstitious and does not ever want the current sum of collected money to be equal to *k*. Given *n* and *k*, what is the maximum amount of money she can collect?

Complete the *maxMoney* function in the editor below. It has two parameters:
1. An integer, *n*, denoting the number of classmates.
2. An integer, *k*, denoting Julia's unlucky number.

The function must return an integer denoting the *maximum* amount of money Julia can collect by visiting her classmates in order of sequential ID number and ensuring that the current sum of money collected is never equal to *k*. As the answer could be large, return *answer % (10^9 + 7)*.

Input Format
Locked stub code in the editor reads the following input from stdin and passes it to the function:
The first line contains an integer, *n*, denoting the number of classmates.
The second line contains an integer, *k*, denoting Julia's unlucky number.

Constraints
- $1 \le n \le 2 \times 10^9$
- $1 \le k \le 4 \times 10^{15}$

Output Format
The function must return an integer denoting the *maximal* amount of money Julia can collect by visiting her classmates in order of sequential ID number and ensuring that the current sum of money collected is never equal to *k*. As the answer could be large, return *answer % (10^9 + 7)*. This is printed to stdout by locked stub code in the editor.

Sample Input 0

```
2
2
```

Sample Output 0

```
3
```

Explanation 0

Julia visits the following sequence of $n = 2$ classmates:

1. Julia collects 1 dollar from classmate 1 to get $sum = 1$.
2. Julia collects 2 dollars from classmate 2 to get $sum = 1 + 2 = 3$; observe that she collected a maximal amount of money and avoided having exactly $k = 2$ dollars.

Sample Input 1

```
2
1
```

Sample Output 1

```
2
```

Explanation 1

Julia visits the following sequence of $n = 2$ classmates:

1. Julia will not collect 1 dollar from classmate 1 because $k = 1$ and she refuses to have a $sum \equiv k$ at any time.
2. Julia moves on and collects 2 dollars from classmate 2 to get $sum = 0 + 2 = 2$.

Sample Input 2

```
3
3
```

Sample Output 2

```
5
```

Explanation 2

Julia must skip some classmate because collecting from all her classmates will result in a $sum \equiv k = 3$ when she collects from the second classmate. There are two ways for her to visit all $n = 3$ classmates:

- She can collect 1 dollar from classmate 1 to get $sum = 1$. Next, she can refuse to collect 2 dollars from classmate 2 to avoid having a sum equal to k. Next, she can collect 3 dollars from classmate 3 to get $sum = 1 + 3 = 4$.
- She can refuse to collect 1 dollar from classmate 1, meaning that $sum = 0$. Next, she can collect 2 dollars from classmate 2 to get $sum = 0 + 2 = 2$. Next, she can collect 3 dollars from classmate 3 to get $sum = 2 + 3 = 5$.

Because we want the maximum amount of money that Julia can collect from her sequentially-numbered classmates without ever having a sum equal to k, we return 5 as our answer.

Question - 11

Super Stack

Implement a simple *stack* that accepts the following commands and performs the operations associated with them:

- `push k` : Push integer *k* onto the top of the stack.
- `pop` : Pop the top element from the stack.
- `inc e k` : Add *k* to each of the bottom *e* elements of the stack.

Function Description

Complete the function *superStack* in the editor below. The function must create an empty stack and perform each of the operations in order. After performing each operation, print the value of the stack's *top* element on a new line. If the stack is empty, print `EMPTY` instead.

superStack has the following parameter(s):

operations[operations[0],...operations[n-1]]: an array of strings

Constraints

- $1 \leq n \leq 2 \times 10^5$
- $-10^9 \leq k \leq 10^9$
- $1 \leq e \leq |S|$, where $|S|$ is the size of the stack at the time of the operation.
- It is guaranteed that *pop* is never called on an empty stack.

▸ Input Format for Custom Testing

▾ Sample Case 0

Sample Input 0

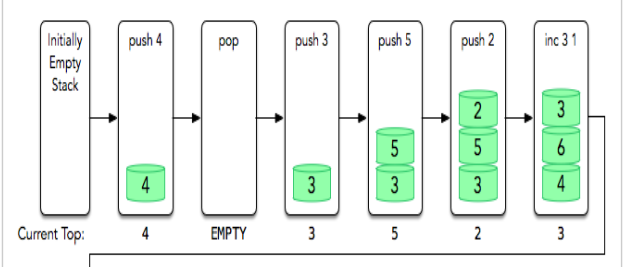
```
12
push 4
pop
push 3
push 5
push 2
inc 3 1
pop
push 1
inc 2 2
push 4
pop
pop
```

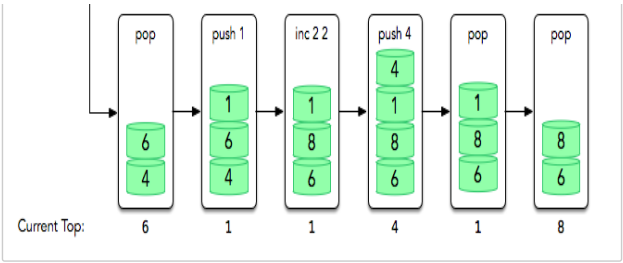
Sample Output 0

```
4
EMPTY
3
5
2
3
6
1
1
4
1
8
```

Explanation 0

The diagram below depicts the stack after each operation:





After each operation, we print the value denoted by *Current Top* on a new line.

In other words, we have an empty stack, *S*, we express as an array where the leftmost element is the *bottom* of the stack and the rightmost element is its *top*. We perform the following sequence of *n = 12* operations as given in the *operations* array:

1. `push 4`: Push *4* onto the top of the stack, so *S = [4]*. We then print the top (rightmost) element, *4*, on a new line.
2. `pop`: Pop the top element from the stack, so *S = []*. Because the stack is now empty, we print *EMPTY* on a new line.
3. `push 3`: Push *3* onto the top of the stack, *S = [3]*. Print 3, and the top of the stack after each of the following operations.
4. `push 5`: Push *5* onto the top of the stack, *S = [3, 5]*.
5. `push 2`: Push *2* onto the top of the stack, *S = [3, 5, 2]*.
6. `inc 3 1`: Add *k = 1* to bottom *e = 3* elements of the stack, *S = [4, 6, 3]*.
7. `pop`: Pop the top element from the stack, *S = [4, 6]*.
8. `push 1`: Push *1* onto the top of the stack, *S = [4, 6, 1]*.
9. `inc 2 2`: Add *k = 2* to bottom *e = 2* elements of the stack, *S = [6, 8, 1]*.
10. `push 4`: Push *4* onto the top of the stack, *S = [6, 8, 1, 4]*.
11. `pop`: Pop the top element from the stack, *S = [6, 8, 1]*.
12. `pop`: Pop the top element from the stack, *S = [6, 8]*.

Question - 12

Slot Machine 2.0

You are testing an experimental slot machine with multiple spinning number wheels that shift positions at random. To make it even more fun, the spinning wheels are not the same because each wheel can have any number of stops from 1 to 9. If it has ***f*** stops, then its stops are numbered from *1* to ***f***. After every spin, every wheel will show one stop number in the slot machine's window, and each stop has the same probability of occurring on that wheel. You run a series of up to 50 test spins and write down all the digits visible in the window. Now you have to do some analysis on the results. More specifically, you are calculating the minimum number of possible stops on each of wheels that can produce that particular series of results.
For this problem, you have to return the sum of those wheel stops.

Input Format
The code to parse the input and print the output is already provided. You have to complete the function **slotGame** in the language chosen, which takes an array of strings of digits and returns an integer.

Constraints:

- The test results will contain between 1 and 50 spin notes, inclusive.
- Each spin note will contain between 1 and 50 digits,

inclusive.
- All notes contain the same number of characters.
- Each digit in each element will be between '1' through '9'.

Sample Input 0
4
137
364
115
724
Sample Output 0
14

Explanation 0
The numbers showing are:
1, 3 and 7 in the first spin
3, 6 and 4 in the second spin
1, 1 and 5 in the third spin
7, 2 and 4 in the fourth spin.
There are three wheels on our slot machine. One wheel needs to have at least 7 stops to produce the first and the fourth spin. Another wheel needs to have at least 4 stops to produce the second and fourth spin. A third wheel needs at least 3 stops to produce the second spin. The answer is 7 + 4 + 3 = 14 minimum total stops across all the wheels. Any fewer stops would have failed to give the results in the input.

Sample Input 1
4
1112
1111
1211
1111
Sample Output 1
5

Explanation 1
One wheel with 2 stops and three wheels with 1 stop each could produce the above results for a minimum total of 5 stops.

Question - 13
Maximizing Profit from Stocks

Your algorithms have become so good at predicting the market that you now know what the share price of Silly Purple Toothpicks Inc. (SPT) will be for the next ***N*** minutes.
Each minute, your high frequency trading platform allows you to either buy one share of SPT, sell any number of shares of SPT that you own, or not make any transaction at all.
Your task is to find the maximum profit you can obtain with an optimal trading strategy?

Constraints
- 1 <= T <= 10
- $1 \le N \le 5x10^5$
- All share prices are between 1 and 10^5

Input Format
The first line contains the number of test cases ***T***. ***T*** test cases follow, each consisting of two lines.
The first line of each test case contains a number ***N***. The next line contains ***N*** integers, denoting the predicted price of WOT shares for the next ***N*** minutes.

Output Format
Output ***T*** lines, each containing the maximum profit which can be obtained for the corresponding test case.

Sample Input 0

```
3
3
5 3 2
3
1 2 100
4
1 3 1 2
```

Sample Output 0

```
0
197
3
```

Explanation 0
For the first case, you cannot make any profit because the share price never increases.
For the second case, you can buy one share on the first two minutes, and sell both of them on the third minute.
For the third case, you can buy one share on the first minute, sell one on the second minute, buy one share on the third minute, and sell one share on fourth minute to get a total profit of 3.

Question - 14
Valid Binary Search Trees

A *binary tree* is a multi-node data structure where each node has, at most, two child nodes and one stored value. It may either be:

- An empty tree, where the root is *null*.
- A tree with a non-null root node that contains a value and two subtrees, *left* and *right*, which are also binary trees.

A binary tree is a *binary search tree (BST)* if all the non-null nodes exhibit two properties:

- Each node's left subtree contains only values that are lower than its own stored value.
- Each node's right subtree contains only values that are higher than its own stored value.

A *pre-order traversal* is a tree traversal method where the *current node* is visited first, then the *left subtree*, and then the *right subtree*. The following pseudocode parses a tree into a list using pre-order traversal:

- If the root is null, output the null list.
- For a non-null node:
  1. Make a list, *left*, by pre-order traversing the left subtree.
  2. Make a list, *right*, by pre-order traversing the right subtree.
  3. Output the stored value of the non-null node, append *left* to it, then append *right* to the result.

For more detail, see the diagram in the *Explanation* section below.

You have to write a program to test whether a traversal history could describe a path on a valid BST. For each query, it should print `YES` on a new line if the path could be in a valid BST, or `NO` if it could not.

Constraints

- $1 \leq q \leq 10$
- $1 \leq n \leq 100$

▸ Input Format

▾ Sample Case 0

Sample Input 0

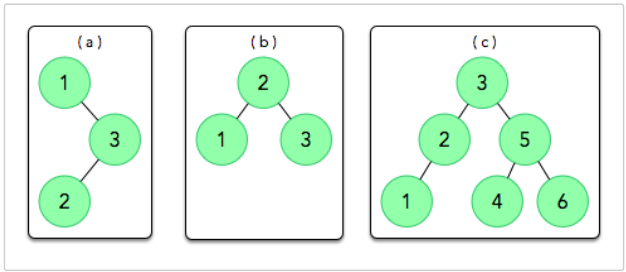
```
5
3
1 3 2
3
2 1 3
6
3 2 1 5 4 6
4
1 3 4 2
5
3 4 5 1 2
```

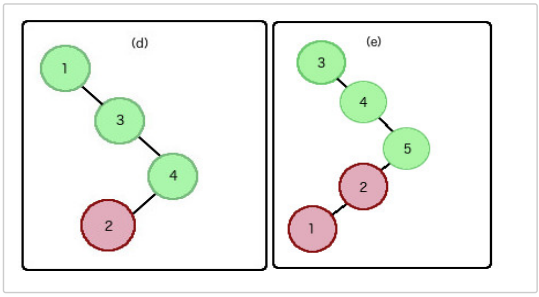
Sample Output 0

```
YES
YES
YES
NO
NO
```

Explanation 0

The diagram below depicts the test for a valid binary search tree for the five queries. Figures (d) and (e) do not represent a valid BST.





We perform the following $q = 5$ queries:

1. Diagram *(a)*'s pre-order traversal matches the pre-order traversal in the first query, `1 3 2`, so we print `YES` on a new line to indicate that the traversal matches a valid BST.
2. Diagram *(b)*'s pre-order traversal matches the pre-order traversal in the second query, `2 1 3`, so we print `YES` on a new line to indicate that the traversal matches a valid BST.
3. Diagram *(c)*'s pre-order traversal matches the pre-order traversal in the first query, `3 2 1 5 4 6`, so we print `YES` on a new line to indicate that the traversal matches a valid BST.
4. The fourth query, `1 3 4 2`, is not a pre-order traversal of a binary search tree. We know that the root is *1* because that is the first value in the list. For the second value to be *3*, it must be the right child of*1*. For the third value to be *4*, it must be the right child of*3*. For *2* to be the last value in the traversal, it would have to be the left child of *4*; however, this would break the order property of a binary search tree because a value less than *3* would be in *3*'s right subtree. Thus, we print `NO` on a new line.
5. The fifth query, `3 4 5 1 2`, is not a pre-order traversal of a

5. The fifth query, `3 4 5 1 2`, is not a pre-order traversal of a binary search tree. We know that the root is *3* because that is the first value in the list. For the second value to be *4*, it must be the right child of *3*. For the third value to be *5*, it must be the right child of *4*. For the fourth value to be *1*, it must be the left child of *5*; however, this would break the order property of a binary search tree because a value less than *4* would be in *4*'s right subtree. Thus, we print `NO` on a new line.

Question - 15

PowerSum

You are given two integers, ***l*** and ***r***. Find the number of integers ***x*** such that ***l ≤ x ≤ r***, and ***x*** is a *Power Number*.

A *Power Number* is defined as an integer that can be represented as sum of *two* powers, i.e.

- $x = a^p + b^q$,
- a, b, p and q are all integers,
- $a, b \geq 0$, and
- $p, q > 1$.

Complete the function *countPowerNumbers* which takes the following arguments :

Name	Type	Description
l	Positive integer	The lower range for finding power sum
r	Positive integer	The upper range for finding power sum

The above function should return the count of power numbers in the given range.

Constraints:

- $0 \leq l \leq r \leq 5 \times 10^6$

Input Format:

The locked code stub in the editor reads the following input from stdin:

The first line contains the value of *l*. The next line contains the value of *r*.

Output Format:

Your function must return a single integer representing the required result.

▼ Sample Case 0

Sample Input

```
0
1
```

Sample Output

```
2
```

Explanation

0 and *1* both are *Power Numbers*.

- $0 = 0^2 + 0^2$,
- $1 = 0^2 + 1^2$.

▼ Sample Case 1

Sample Input

```
25
30
```

Sample Output

```
5
```

Explanation

Except *30*, all are *Power Numbers.*

- $25 = 5^2 + 0^2$,
- $26 = 5^2 + 1^2$,
- $27 = 3^3 + 0^2$,
- $28 = 3^3 + 1^2$,
- $29 = 5^2 + 2^2$.

Question - 16
Nuclear Rods

A core meltdown just happened at a local nuclear plant! They need to move all of their rods to lead isolation chambers using specialized, radiation hardened robots. There is a cost of moving the rods associated with the square root of the number of rods lifted. Lifting four rods costs twice the cost of lifting two rods because $4^{1/2} = 2$ and $1^{1/2} = 1$. Any remaining fraction, such as $3^{1/2} \approx 1.732$ is raised to the next integer, i.e. $ceiling(3^{1/2}) = 2$.

In this challenge, you will be given a list of fused pairs of nuclear fuel rods. Once you have determined all of the fused groups, calculate the total cost of moving the rods to isolation.

For example, assume you have *10* fuel rods labeled *1 - 10*. Pairs of rods are *[1, 3], [1, 4]* and *[3,6]*. There are two groups of fused rods: *{1, 3, 4}* and *{3,6}*. The first group costs $ceiling(3^{1/2}) = 2$ units to move. The second costs $ceiling(2^{1/2}) = 2$ units. The remaining *5* rods each cost $ceiling(1^{1/2}) = 1$ unit to move giving us a total of 2 + 2 + 5 = 9 units cost.

Function Description

Complete the function *minimalCost* in the editor below. The function must return an integer denoting the cost of isolating all of the rods.

minimalCost has the following parameter(s):

n: the number of rods

pairs[pairs[1],...pairs[m]]: an array of space-separated integer pairs denoting fused rod numbers, *p* and *q*

Constraints

- $2 \le n \le 10^5$
- $1 \le m \le 10^5$
- $1 \le p, q \le n$
- $p \ne q$

▸ **Input Format for Custom Testing**

▾ **Sample Case 0**

Sample Input 0

```
4
2
1 2
1 4
```

Sample Output 0

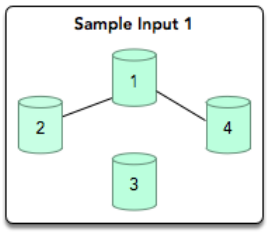
```
3
```

Explanation 0

The following arguments are passed to the function:

1. *n = 4*
2. *pairs = [[1 2], [1 4]]*

The diagram below depicts the configuration of rods:



The cost for removing each group is as follows:

1. Set *{1, 2, 4}: c = ceil(sqrt(3)) = 2*
2. Set *{3}: c = ceil(sqrt(1)) = 1*

When we sum all values of *c*, we get *2 + 1 = 3* as our answer.

▸ **Sample Case 1**

Question - 17

Reduced Fraction Sums

Consider two fractions in the form `a/b` and `c/d`, where `a`, `b`, `c`, and `d` are integers. Given a string describing an arithmetic expression that sums these two fractions in the form `a/b+c/d`, compute the sum and fully reduce the resultant fraction to its simplest form.

For example:

- The expression `1/2+1/6` evaluates to *4/6*, which we reduce to the string `2/3`.
- The expression `7/10+13/10` evaluates to *20/10*, which we reduce to the string `2/1`.

Function Description

Complete the function *reducedFractionSums* in the editor below. The function must return an array of strings representing the fully reduced fractions.

reducedFractionSums has the following parameter(s):

expressions[expressions[0],...expressions[n-1]]: an array of strings in the form `a/b+c/d`.

Constraints

- *1 ≤ n ≤ 500*
- *1 ≤ a, b, c, d ≤ 2000*

▸ **Input Format for Custom Testing**

▾ **Sample Case 0**

Sample Input 0

```
5
722/148+360/176
978/1212+183/183
358/472+301/417
780/309+684/988
258/840+854/686
```

Sample Output 0

```
2818/407
365/202
145679/98412
4307/1339
1521/980
```

Explanation 0

We perform the following *n = 5* calculations:

0. `722/148+360/176 = 127072/26048+53280/26048 = 180352/26048 = 2818/407`.
1. `978/1212+183/183 = 978/1212 + 1212/1212 = 2190/1212 = 365/202`.
2. `358/472+301/417` evaluates to the reduced fraction `145679/98412`.
3. `780/309+684/988` → `4307/1339`.
4. `258/840+854/686` → `1521/980`.

Return the array *["2818/407", "365/202", "145679/98412", "4307/1339", "1521/980"]*.

▸ **Sample Case 1**

Question - 18

Maximum Difference in an Array

You are given an array of integers and must compute the maximum difference between any item and any lower indexed smaller item for all the possible pairs, i.e., for a given array a find the maximum value of *a[j] - a[i]* for all *i, j* where *0 ≤ i < j < n and a[i] < a[j]*. If there are no lower indexed smaller items for all the items, then return -1.

For example, given an array [1, 2, 6, 4], you would first compare 2 to the elements to its left. 1 is smaller, so calculate the difference 2 - 1 = 1. 6 is bigger than 2 and 1, so calculate the differences 4 and 5. 4 is only bigger than 2 and 1, and the differences are 2 and 3. The largest difference was 6 - 1 = 5.

Function Description

Complete the function *maxDifference* in the editor below. The function must return an integer representing the maximum difference in *a*.

maxDifference has the following parameter(s):

a[a_0,...a_{n-1}]: an array of integers

Constraints

- $1 \leq n \leq 2 \times 10^5$
- $-10^6 \leq a[i] \leq 10^6 \ \forall i \in [0, n-1]$

▸ **Input Format For Custom Testing**

▾ **Sample Case 0**

Sample Input 0

```
7
2
3
10
2
4
8
1
```

Sample Output

```
8
```

Explanation

$n = 7$, $a = [2, 3, 10, 2, 4, 8, 1]$

Differences are calculated as:

- 3 - [2] = [1]
- 10 - [3, 2] = [7, 8]
- 4 - [2, 3, 2] = [2, 1, 2]
- 8 - [4, 2, 3, 2] = [4, 6, 5, 6]

The maximum is found at 10 - 2 = 8.

▸ **Sample Case 1**

Question - 19

Username Disparity

Given two usernames, the degree of similarity is defined as the length of the longest prefix common to both strings. In this challenge, you will be given a string. You must break the string to create ever shorter suffixes, then determine the similarity of the suffix to the original string. Do this for each suffix length from the length of the string to 0, and cumulate the results.

As an example, consider the string 'ababa'. Compare all suffixes to the original string.

Discard	Suffix	Similarity	Length
''	'ababa'	'ababa'	5
'a'	'baba'	''	0
'ab'	'aba'	'aba'	3
'aba'	'ba'	''	0
'abab'	'a'	'a'	1
'ababa'	''	''	0

So our sum is 5 + 0 + 3 + 0 + 1 + 0 = 9

Function Description
Complete the function *usernameDisparity* in the editor below. The function must return an integer array of the sums of the similarities for each test case.

usernameDisparity has the following parameter(s):
inputs[inputs$_0$,...inputs$_{n-1}$]: an array of username strings to process

Constraints
$1 \leq T \leq 10$
$1 \leq |s| \leq 10^5$
The string contains only letters in the range ascii[a-z].

▸ **Input Format For Custom Testing**

▾ **Sample Case 0**

Sample Input For Custom Testing

```
1
ababaa
```

Sample Output

```
11
```

Explanation 1

T = 1
S = ababaa

The suffixes of the string are *ababaa, babaa, abaa, baa, aa and a.* The similarities of each of these strings with the string *ababaa* are *6,0,3,0,1,1* respectively.
The sum is calculated *6 + 0 + 3 + 0 + 1 + 1 = 11.*

▸ **Sample Case 1**

Question - 20
Hosts and the Total Number of Requests

In this challenge, you will write a program to analyze a log file and summarize the results. You will be given a text file of an http requests log and must list the number of requests from each host. Output should be directed to a file as described in the Program Description below.

The format of the log file, a text file with a `.txt` extension, follows. Each line contains a single log record with the following columns (in order):

1. The *hostname* of the *host* making the request.
2. This column's values are missing and were replaced by a hyphen.
3. This column's values are missing and were replaced by a hyphen.
4. A timestamp enclosed in square brackets following the

format `[DD/mmm/YYYY:HH:MM:SS -0400]`, where `DD` is the day of the month, `mmm` is the name of the month, `YYYY` is the year, `HH:MM:SS` is the time in *24*-hour format, and `-0400` is the time zone.

5. The *request*, enclosed in quotes (e.g., `"GET /images/NASA-logosmall.gif HTTP/1.0"`).
6. The *HTTP response code*.
7. The total number of *bytes* sent in the response.

▸ Example log file entry

Function Description

Your function must create a unique list of hostnames with their number of requests and output to a file named `records_`*filename* where *filename* is replaced with the input *filename*. Each hostname should be followed by a space then the number of requests and a newline. Order doesn't matter.

Constraints

- The log file has a maximum of 2×10^5 lines of records.

▸ Input Format

▾ Sample Case 0

Sample Input 0

```
hosts_access_log_00.txt
```

Sample Output 0

Given *filename = "hosts_access_log_00.txt"*, we process the records in `hosts_access_log_00.txt` and create an output file named `records_hosts_access_log_00.txt` containing the following rows:

```
burger.letters.com 3
d104.aa.net 3
unicomp6.unicomp.net 4
```

Explanation 0

The log file `hosts_access_log_00.txt` contains the following log records:

```
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400]
"GET /shuttle/countdown/ HTTP/1.0" 200 3985
burger.letters.com - - [01/Jul/1995:00:00:11 -0400] "GET
/shuttle/countdown/liftoff.html HTTP/1.0" 304 0
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET
/images/NASA-logosmall.gif HTTP/1.0" 304 0
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET
/shuttle/countdown/video/livevideo.gif HTTP/1.0" 200 0
d104.aa.net - - [01/Jul/1995:00:00:13 -0400] "GET
/shuttle/countdown/ HTTP/1.0" 200 3985
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400]
"GET /shuttle/countdown/count.gif HTTP/1.0" 200 40310
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400]
"GET /images/NASA-logosmall.gif HTTP/1.0" 200 786
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400]
"GET /images/KSC-logosmall.gif HTTP/1.0" 200 1204
d104.aa.net - - [01/Jul/1995:00:00:15 -0400] "GET
/shuttle/countdown/count.gif HTTP/1.0" 200 40310
d104.aa.net - - [01/Jul/1995:00:00:15 -0400] "GET
/images/NASA-logosmall.gif HTTP/1.0" 200 786
```

When we consolidate the data above, we confirm the following:

1. The host `unicomp6.unicomp.net` made *4* requests.
2. The host `burger.letters.com` made *3* requests.
3. The host `d104.aa.net` made *3* requests.