Data Structure and Algorithm, Spring 2021

Homework 3

Purple Correction Date: 05/27/2021 12:00

Orange Correction Date: 05/25/2021 18:00

Red Correction Date: 05/24/2021 20:00

Due: 23:59:00, Saturday, June 19, 2021

TA E-mail: dsa_ta@csie.ntu.edu.tw

Rules and Instructions

- Any form of cheating, lying, or plagiarism will not be tolerated. Students can get zero scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.
- In Homework 3, the problem set contains 6 problems and is divided into two parts, the non-programming part (Problems 1, 2, 3) and the programming part (Problems 4, 5, 6). The total score of these six problems are 500 points. **Note that due to the current pandemic situation, the score of HW3 is capped at 400 points.** That is, your-final-score $= \min\{400, \text{your-raw-score}\}$. The implication is that you may choose to work on only 80% of HW3 without any significant impact on your score. Please take advantage of this special policy to lower your own work load in the next few weeks.
- For problems in the non-programming part, you should combine your solutions in ONE PDF file. Your file should generally be legible with a white/light background—using white/light texts on a dark/black background is prohibited. Your solution must be as simple as possible. At the TAs' discretion, solutions which are too complicated will be regarded as incorrect. Moreover, if you would like to use any theorem which is not mentioned in the classes, please include its proof in your solution.
- The PDF file for the non-programming part should be submitted to Gradescope as instructed, and you should use Gradescope to tag the pages that correspond to each subproblem to facilitate the TAs' grading. Failure to tagging the correct pages of the subproblem can cause losing part or all of the scores on the subproblem.
- For the programming part, you should have visited the *DSA Judge* (https://dsa-2021.csie.org) and familiarized yourself with how to submit your code via the judge system in Homework 0.1126.
- For problems in the programming part, you should write your code in C programming

language, and then submit the code via the judge system. You can submit up to 5 times per day for each problem. The judge system will compile your code with

```
gcc main.c -static -O2 -std=c11
```

- Discussions on course materials and homework solutions are encouraged. But you should write the final solutions alone and understand them fully. Books, notes, and Internet resources can be consulted, but not copied from. For *each non-programming problem*, you have to specify the references (the Internet URL you consulted with or the people you discussed with) on the first page of your solution for that problem; for *each programming problem*, you have to specify the references on the first lines (comments) of your code (`main.c`) for that problem.

- Since everyone needs to write the final solutions alone, there is absolutely no need to lend your homework solutions and/or source codes to your classmates at any time. In order to maximize the level of fairness in this class, lending and borrowing homework solutions are both regarded as dishonest behaviors and will be punished according to the honesty policy.

- The score of the part that is submitted after the deadline will get some penalties according to the following rule:

$$\text{LateScore} = \max\left(0, \frac{86400 \cdot 5 - \text{DelayTime (sec.)}}{86400 \cdot 5}\right) \times \text{OriginalScore}$$

- If you have questions about HW3, please go to the course forum and discuss (*strongly preferred*, which will provide everyone a better learning experience). If you really need an email answer, please follow the rules outlined below to get a fast response:

  - The subject should contain two tags, `"[HW3]"` and `"[Px]"`, specifying the problem where you have questions. For example, `"[HW3][P1] Can swap in subproblem 7 be done in constant time?"`. Adding these tags allows the TAs to track the status of each email and to provide faster responses to you.
  - If you want to provide your code segments to us as part of your question, please upload it to Gist or similar platforms and provide the link. Screenshots or code segments directly included in the email is discouraged and may not be reviewed.

Non-Programming Part

Problem 1 - Hash (60 pts)

1. (10 pts) Suppose Arvin stores n keys in a hash table of size n^2 using a **uniform** hash function $h(k)$. What is the probability that there is any collision ?

2. (10 pts) Assume that we have a **uniform** hash function $h(k)$ which maps input k into hash space P. Arvin wants to generate a set of magic passwords using the hash function $h(k)$. What is the expected number of times to query $h(k)$, in order to get $\frac{|P|}{4}$ unique password ?

3. (20 pts) We have learned open addressing from the lecture video. Define two hash functions $h_1(k) = k \mod m$ and $h_2(k) = 1 + (k \mod (m-1))$. Please insert the keys $\{18, 34, 9, 37, 40, 32, 89\}$ in the given order into a hash table of length $m = 11$. Assume the hash table is empty initially. Please specify in a step-by-step manner how the keys are inserted into the hash table using open addressing with (1) linear probing with hash function $h_1(k)$, (2) double hashing with primary hash function $h_1(k)$ and secondary hash function $h_2(k)$.

   Please fill in the table below, showing the content of the hash table after each insertion.

   - Open addressing with linear probing

keys to be inserted \ index	0	1	2	3	4	5	6	7	8	9	10
18								18			
34											
9											
37											
40											
32											
89											

   - Open addressing with double hashing

keys to be inserted \ index	0	1	2	3	4	5	6	7	8	9	10
18								18			
34											
9											
37											
40											
32											
89											

4. (20 pts) Cuckoo hashing is a hashing technique which guarantees O(1) worst-case query time. It uses two tables T_1 and T_2 of the same size, and two hash functions $h_1(k)$ and $h_2(k)$. To place an element x into the hash table, start by inserting x into T_1 at position $h_1(x)$. If a collision happens, the element x originally stored at position $h_1(x)$ in T_1 is moved to T_2 at position $h_2(y)$. In case of another collision, the element z stored at position $h_2(y)$ in T_2 is again moved to position $h_1(z)$ in T_1. Repeat these steps until the moved element can be placed in a position without collision.

   However, in practice, it is possible for this insertion process fails by entering an infinite loop. If this happens, all elements would be removed from the two tables and rehashed with two new hash function $h_1'(k)$ and $h_2'(k)$. However, in this question, the inserted sequence does not introduce this condition, and thus you **DO NOT** need to consider this,

   Given two tables of size 7 each and two hash functions $h_1(k) = k \bmod 7$ and $h_2(k) = \lfloor \frac{k}{7} \rfloor \bmod 7$. Insert elements $[6, 31, 2, 41, 30, 45, 44]$ into the hash table using cuckoo hashing in the given order. Draw the content of the two hash tables after each insertion in a step-by-step manner by filling in the table below.

   - Table T_1, using $h_1(k)$

keys to be inserted \ index	0	1	2	3	4	5	6
6							
31							
2							
41							
30							
45							
44							

   - Table T_2, using $h_2(k)$

keys to be inserted \ index	0	1	2	3	4	5	6
6							
31							
2							
41							
30							
45							
44							

Problem 2 - String matching (60 pts)

In the following subproblems, if the question asks for an algorithm, your answer should include:

(a) Your algorithm in the form of pseudo-code and a brief explanation of its correctness.

(b) Analysis of time complexity and space complexity of your algorithm. Better complexity would result in the higher points, but correct algorithm would give you base points.

Consider a string $S[1..N]$ of length N. You are given Q queries ($Q \approx N$), where there are three numbers l_1, l_2, n. The query is to determine if string $S[l_1..l_1 + n - 1]$ equals to string $S[l_2..l_2 + n - 1]$ (i.e., the respective substrings starting at l_1 and l_2 and of length n match each other).

1. (10 pts) Design an algorithm to respond to these queries. Note that when analyzing the complexity, you should take the time of pre-processing and responding to queries (if any) into consideration.

Now you are given another string $S[1..N]$ of length N. Please compute a function $x(i) = \max\{p \mid S[1..p] == S[i..i + p - 1]\}$ for $1 \leq i \leq N$, and store these values $x(i)$ in an array X.

2. (10 pts) Given a string $S =$ "*bcdabcde*", compute $x(i)$ for $1 \leq i \leq 8$.

3. (20 pts) Design an algorithm that calculate the content of array X, given the input string S.

4. (20 pts) Now that you have finished the problems above, try to solve of a slightly different version of the string matching problem taught in class: find the number of occurrences of pattern p in string t. Utilize the array X created from the previous subproblems to construct an efficient algorithm to solve the problem.

Problem 3 - Having Fun with Disjoint Sets (80 pts)

In subproblem 1 and 2, you can use a disjoint set data structure with linked list representation and **union by size** technique. You can use the following functions without any detailed explanation:

- `MAKE-SET(x)`: Create a set with one element `x`.
- `UNION(x, y)`: Merge the set that contains `x` and the set that contains `y` into a new set, and delete the original two sets that contains `x` and `y`, respectively.
- `FIND-SET(x)`: Find out the ID of the set that contains `x`.

1. (15 pts) Giver loves graph theory. Now, he's interested in `bipartite graph`. A bipartite graph is a simple, undirected, connected graph $G = (V, E)$ whose set of vertices V can be split into **two disjoint and independent** vertices set A, B, such that every edge in E connects a vertex in A and a vertex in B. Now, Giver wants you to implement three functions:

   - `INIT(N)`: Initialize the graph with N vertices. Vertices are numbered from 0 to $N - 1$.
   - `ADD-EDGE(x, y)`: Add an edge which connects vertex x and vertex y. You can assume that $0 \leq x, y \leq N - 1, x \neq y$.
   - `IS-BIPARTITE()`: Return `TRUE` is the graph is a bipartite graph, return `FALSE` otherwise

   Giver will perform `INIT(N)` first, then perform the other two operations M times. Please use disjoint set data structure to solve this problem (implement the above 3 functions) in $O(N + M \log N)$ time. Note that Giver can call the other two operations **in any order**. The following is an example that may help you solve this problem:

   - `INIT(5)`: Initialize the graph with 5 vertices.
   - `IS-BIPARTITE()`: You should return `TRUE`. This is a special case where there is no edge in the graph, but still considered bipartite.
   - `ADD-EDGE(2, 3)`: Add an edge which connects vertices $2, 3$.
   - `ADD-EDGE(3, 4)`: Add an edge which connects vertices $3, 4$.
   - `IS-BIPARTITE()`: You should return `TRUE`, since the graph so far is bipartite. This is because V can be split into $\{2, 4\}$ and $\{0, 1, 3\}$, and then any of the edges connect a vertex in the first vertex set and a vertex in the other.

- `ADD-EDGE(2, 4)`: Add an edge which connects vertices $2, 4$.
- `IS-BIPARTITE()`: You should return `FALSE`, since the graph is not bipartite anymore. This means that no split among the vertices can be found, such that any of the edges connect a vertex in the first vertex set and a vertex in the other.

(Hint: Bipartite graph is 2-colorable. That is to say, you can color the vertices in `black` and `white`, such that any edge connects a black vertex and a white vertex.)

2. (15 pts) Now, Giver finds out that the disjoint set data structure has more applications! Giver observes that there are N people playing `Paper, Scissor, Stone`. Giver also finds out that, each of the N people will always have the same hand (either paper, scissor, or stone) and never changes. In addition, Giver's friend, Robert, reports some relations between those N people to Giver. But Giver finds out that Robert might give him some contradicting relations. Now, Giver wants you to implement the following four functions:

   - `INIT(N)`: Giver sees N people. They are numbered from 0 to $N - 1$.
   - `WIN(a, b)`: Person a defeats person b. (Paper defeats stone, scissor defeats paper, and stone defeats scissor).
   - `TIE(a, b)`: Person a and person b have the same hand.
   - `IS-CONTRADICT()`: Return `TRUE` is there's a contradict relation; return `FALSE` otherwise.

   Giver will perform `INIT(N)` first, then call the other three functions M times. Please use the disjoint set data structure to solve this problem (implement the above 4 functions) in $O(N + M \log N)$ time. Note that Giver can call the other three functions **in any order**. The following is an example that may help you solve this problem:

   - `INIT(5)`: Giver sees five people.
   - `WIN(1, 2)`: Person 1 defeats person 2.
   - `WIN(2, 3)`: Person 2 defeats person 3.
   - `IS-CONTRADICT()`: You should return `FALSE`, as there exist possible assignments of hands to people satisfying previous relations. For example, Person 1 can have `Paper`, person 2 has `Stone`, and person 3 has `Scissor`.
   - `TIE(2, 4)`: Person 2 has the same hand as person 4.
   - `WIN(4, 1)`: Person 4 defeats person 1.
   - `IS-CONTRADICT()`: You should return `TRUE`, since it is impossible for person 4 to defeat person 1 if person 1 defeats person 2.

Now, Giver wants to have fun with disjoint sets data structure using simple undirected graph again. He wants to implement the following functions:

- `INIT(N)`: Initialize the graph with N vertices. Vertices are numbered from 0 to $N-1$. Currently, there are no edges in the graph yet.
- `ADD-EDGE(x, y)`: Add an edge that connects vertex x and vertex y. Assume there is no edge connecting vertex x and y.
- `SHOW-CC()`: return the number of the connected components in this graph.
- `UNDO()`: Undo the last `ADD-EDGE()` operation. This operation is equivalent to removing the last added edge. When this operation is executed, assume there is at least one edge in the graph.

After spending 11.26 hours, Giver finally finished the implementation of the above four functions. He uses a `stack` to store the changes of the variables. When performing `UNDO()` operation, he pops all the changed variables, and restore the variables into their original values. You can see his original implementation here: https://www.csie.ntu.edu.tw/~b07902132/djs.c. He implemented it with linked-list representation but without any optimization techniques.

If there are N vertices in the graph, one `INIT` operation and followed by M other operations are performed in total, the time complexity would be $O(N + MN)$, which is too slow.

3. (15 pts) Now, Giver learns that he can apply `path compression` technique on his disjoint set implementation. You can see his implementation here: https://www.csie.ntu.edu.tw/~b07902132/djs_path_compression.c.

   Prove or disprove that, the complexity of the above implementation is $O(N + M \log N)$.

4. (15 pts) Now, Giver learns that he can apply `union by size` technique on his disjoint set implementation (without `path compression`). You can see his implementation here: https://www.csie.ntu.edu.tw/~b07902132/djs_union_by_size.c.

   Prove or disprove that, the complexity of the above implementation is $O(N + M \log N)$.

   (Hint: You can use everything that is proved in lecture, slide, or in the textbook. To disprove the time complexity, you can simply show a counter-example)

Finally, Giver thinks that you must observe **the beauty of disjoint set**. Here's the last challenge that Giver wants to give you.

He would like you to design a disjoint set implementation with the following functions:

- `MAKE-SET(x)`: Create a set with one element `x`.

- `UNION(x, y)`: Merge the set that contains `x` and the set that contains `y` into a new set, and delete the original set(s) that contains `x, y`.
- `SAME-SET(x, y)`: Return `TRUE` if element x and element y belong to the same set. Return `FALSE` otherwise.
- `ISOLATE(k)`: Element k will be isolated from the set it belongs to. That is to say, remove k from that set and form a set by itself.

Giver will perform M operations in total. You can assume that $0 \leq x, y, k \leq M - 1$.
Here's an example that may help you solve this subproblem.

- `MAKE-SET(1)`
- `MAKE-SET(2)`
- `SAME-SET(1, 2)`: You should return `FALSE`.
- `UNION(1, 2)`
- `SAME-SET(1, 2)`: You should return `TRUE`.
- `MAKE-SET(3)`
- `UNION(1, 3)`
- `SAME-SET(2, 3)`: You should return `TRUE`
- `ISOLATE(2)`
- `SAME-SET(1, 3)`: You should return `TRUE`
- `SAME-SET(2, 3)`: You should return `FALSE`

5. (20 pts) Design a disjoint set implementation that can support the above functions. If we perform M operations in total, its complexity should be $O(M\alpha(M))$.

   Note that if your time complexity is not $O(M\alpha(M))$, you can still get some partial points.

Congratulations, you have finished all the disjoint set challenges set by Giver!

Problem 4 - Too Many Emails (100 pts)

Time Limit	:	3 s
Memory Limit	:	1024 MB

Problem Description

Lingling recently receive her Ph.D. degree and became a professor that has always been her dream job. But some chores bother her. There are too many emails sent to her every day! Also, the mail system that she uses is very old. The system sometimes adds garbled text into her emails, and has very little storage space.

To solve the first problem, Lingling makes some observations and found that garbled text exhibits some patterns:

1. It's a substring of an email, and each email only has at most one garbled text.
2. Garbled text is known to have some specific characters. Some might occur for multiple times.

Lingling comes up with an idea that can locate garbled text in an email. First, based on recent experience, she writes down a string of garbled text characters, denoted G. In G, assume we have n different characters, $c_1, c_2, \ldots, c_n$, with the number of occurrences in G denoted as $o_1, o_2, \ldots, o_n$, respectively. For example, if G="TGET", then we have c_1='T', c_2='G', c_3='E', and $o_1 = 2, o_2 = 1, o_3 = 1$. To locate the garbled text within the email text D, we look for a substring $D[i..j], 1 \leq i, j \leq |D|$, such that the number of occurrences of $c_1, c_2, \ldots, c_n$ within $D[i..j]$, denoted $p_1, p_2, \ldots, p_n$, can satisfy the inequalities $p_1 \geq o_1, p_2 \geq o_2, \ldots, p_n \geq o_n$. Note that we want to find a minimum length garbled text $D[i..j]$ satisfying the above conditions, in order to retain the most email content. Then $D[i..j]$, identified as garbled text in the email, can be removed from D, obtaining the email without garbled text, denoted as D'.

For example, given email text D ="DSA is so GaRBledTExTeasy:D" and garbled text string G ="TGET". To see if substring $D[11, 21]$ ="GaRBledTExT" is a garbled text, we evaluate the number of occurrences of c_1='T', c_2='G', c_3='E' within $D[11, 21]$, and obtain $p_1 = 2, p_2 = 1, p_3 = 1$. Since they satisfy $p_1 \geq o_1, p_2 \geq o_2, p_3 \geq o_3$, $D[11, 21]$ is considered garbled text. Moreover, we cannot find any other substring of D satisfying these conditions and has a length smaller than $D[11, 21]$. Finally, we remove $D[11, 21]$ from the original D, obtaining the final email text D' ="DSA is so easy:D".

The next step is to save the space to store the email. The main idea is to break D' into k substrings, called blocks, denoted as $b_1, b_2, \ldots, b_k$. Each block is a substring of D', i.e., $b_i = D'[s_i..s_{i+1} - 1], 1 \leq s_i \leq |D'|, s_1 = 1, s_{k+1} = |D'| + 1$, but can have different lengths. Most importantly, we have $b_i == b_{k-i+1}$, for $1 \leq i \leq k$. In this case, the email system only needs to store b_i but does not need to store b_{k-i+1}, for $1 \leq i \leq \lfloor k/2 \rfloor$. This can effectively save up to 50% of the length of the email text. Note that the system prefers breaking the email into small blocks. Therefore, you are requested to break the email into **as many blocks as possible**.

For example, D'="ABDCDAB" can be broken into b_1 ="AB",b_2 ="D",b_3 ="C",b_4 ="D", b_5 ="AB", saving space to store b_4 and b_5.

Can you write a program to help Lingling remove garbled text from the email, and then find a way to break the email into k blocks to save the storage space?

Input

The first line contains an integer T, indicating the number of test cases. For each test case, there are two lines, which contain string D and string G respectively, representing the email text and the garbled text string. D and G only contain English characters (upper- and lower-case letters should be treated as different characters). If there are multiple garbled texts of the same length, remove the left most one.

Output

For each test case, print string D' with garbled text removed and using '|' to indicate where to break D' into blocks. Things that need to be pay attention to include:

1. There are cases where the garbled text conditions are not satisfied with any of the substrings of D. In this case, you do not need to delete any character from D to obtain D'.

2. There are cases where D cannot be broken into blocks which satisfied the block conditions. In this case, you do not need to add any '|' to D'.

3. You can assume that length of the output is always larger than or equal to 1.

Constraints

1. $1 \leq T \leq 10^2$

2. $1 \leq |G| \leq |D| \leq 10^5$

Subtask 1 (20 pts)

- $1 \leq |G| \leq |D| \leq 10^3$

Subtask 2 (15 pts)

- D does not contain any character in G.

Subtask 3 (15 pts)

- D cannot be broken into more than one blocks. (No '|' in the output)

Subtask 4 (50 pts)

- No other constraints.

Sample Input 1

```
1
HelloWorldolleH
ee
```

Sample Output 1

```
H|H
```

Sample Input 2

```
2
DSARANDOMTEXTISSOHARD
RTTX
DSARANDOMTEXTISSOEASY
RXTT
```

Sample Output 2

```
D|SAISSOHAR|D
DSAISSOEASY
```

Sample Input 3

```
3
OkayOkayOkay
x
Nooo
oo
JJJJJJ
j
```

Sample Output 3

```
Okay|Okay|Okay
No
J|J|J|J|J|J
```

Problem 5 - Alice's Bookshelf (100 pts)

Time Limit	:	7 s
Memory Limit	:	1024 MB

Problem Description

Alice loves reading books and has a bookshelf. For each book in her bookshelf, there's a number, indicating the corresponding priority. You have to maintain the sequence of the books' priority with operations specified in the Input section.

Input

The first line contains two numbers N, Q ($1 \leq N, Q \leq 8 \times 10^5$), indicating the number of books in the initial bookshelf and the number of operations.
The second line contains N integers, representing the priority of books in the bookshelf.
Each of the following Q lines contains one of the following operations, which can be identified with the first number in each line:

1. `1 p k`: Insert a book with priority p *after* the k-th position. Note that the position index starts from 1. If $k = 0$, then insert it as the first book in the bookshelf.
2. `2 k`: Delete the k-th book.
3. `3 l r p`: Increase the priorities of the books between the positions l and r by p (including books at position l and r, and $l \leq r$.) Note that p may be negative.
4. `4 l r`: Query the largest priority of books between the positions l and r (including books at position l and r, and $l \leq r$.)
5. `5 l r`: Reverse the order of books between the positions l and r (including books at position l and r, and $l \leq r$.)
6. `6`: Remove the book with the largest priority.

You can assume that the given priorities can be stored in a 32-bit `int` variable and all operations are valid. Note that the bookshelf may become empty. When performing operation 6, if there are multiple books with the largest priority, you should remove the book with smallest position index.

Output

For each operation 4 in the sequence of given operations, the largest priority of books between the position l and r (inclusive) has to be printed out. You don't have to print anything for any other operation.

Subtask 1 (10 pts)

- $N, Q \leq 1000$.

Subtask 2 (20 pts)

- Only operation 1, 2, 4.

Subtask 3 (15 pts)

- No operation 5, 6.

Subtask 4 (10 pts)

- No operation 6.

Subtask 5 (15 pts)

- No operation 5.

Subtask 6 (10 pts)

- $N, Q \leq 10^5$.

Subtask 7 (20 pts)

- No other constraints.

Sample Input 1

```
5 5
9 5 1 -5 -3
2 5
1 5 4
4 3 4
6
4 2 2
```

Sample Output 1

```
1
1
```

Sample Input 2

```
6 10
-4 3 2 4 -2 4
3 2 4 -3
1 6 0
3 2 3 1
3 5 5 1
1 -3 3
3 4 7 5
2 3
2 6
2 2
4 2 5
```

Sample Output 2

```
7
```

Sample Input 3

```
10 10
-9 8 3 -8 -4 -8 -2 1 5 3
2 9
1 8 6
1 -7 10
4 4 6
6
4 1 4
1 3 4
2 3
1 9 9
4 5 9
```

Sample Output 3

```
-4
3
8
```

Hint

As you have to insert and delete the priority of books, you may be considering using a self-balancing binary search tree to maintain the sequence of priority. Now, how do you add *extra* information to the tree so that you can handle the other operations efficiently?

Problem 6 - Recover Graph (100 pts)

Time Limit	:	1 s
Memory Limit	:	1024 MB

Problem Description

Robert, a clever student, is learning graph theory in the DSA course. Commonly, people use adjacency lists to store the graph. Robert is diligent, so he tries to implement adjacent lists after class. Formally, Robert implements it with the pseudo code in the following:

Algorithm 1: Implementation of adjacency lists

```
Function BuildAdjList(numVertices, edgeList):
    adjList[1..numVertices] ← empty lists
    for edge (x, y) in edgeList do
        insert y into adjList[x]
        insert x into adjList[y]
    end
    return adjList[1.. numVertices]
```

However, after getting the adjacency list, Robert forgets the original list. He would like to verify the correctness of his implementation. Help Robert to recover the original list of edges, in the original given order.

Input

The first line contains only one integer N ($1 \leq N \leq 10^5$), representing the number of vertices in the graph. The vertices are indexed from 1 to N.

Each of the next N lines contains the description of one of the adjacency lists. The first integer num_i in the i^{th} line represents the length of the list of vertex i. Then, the following num_i numbers represents the neighbors of vertex i in the list. You can assume that $0 \leq \sum num_i \leq 4 \cdot 10^5$. It is guaranteed that without considering the order of vertices in the adjacency lists, the graph is simple, that is, no self-loops and multiple edges exist.

Output

You should determine if there exist any possible list of edges satisfying the constraints given in the input. If no answers exists, print "No" to the output (without quotation marks).
Otherwise, print "Yes" in the first line (without quotation marks). In this case, in the following lines, print two integers on the i^{th} line, representing the i^{th} edge in the list of edges. If there are more then one list of edges satisfying the constraints, you may print any of them.

Subtask 1 (30 pts)

- $N \leq 1000$.

Subtask 2 (70 pts)

- No other constraints.

Sample Input 1

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

Sample Output 1

```
Yes
1 2
2 3
1 3
```

Sample Input 2

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

Sample Output 2

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

Sample Input 3

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

Sample Output 3

```
No
```