You are given two strings of the same length `s1` and `s2` and a string `baseStr`.

We say `s1[i]` and `s2[i]` are equivalent characters.

- For example, if `s1 = "abc"` and `s2 = "cde"`, then we have `'a' == 'c'`, `'b' == 'd'`, and `'c' == 'e'`.

Equivalent characters follow the usual rules of any equivalence relation:

- **Reflexivity:** `'a' == 'a'`.
- **Symmetry:** `'a' == 'b'` implies `'b' == 'a'`.
- **Transitivity:** `'a' == 'b'` and `'b' == 'c'` implies `'a' == 'c'`.

For example, given the equivalency information from `s1 = "abc"` and `s2 = "cde"`, `"acd"` and `"aab"` are equivalent strings of `baseStr = "eed"`, and `"aab"` is the lexicographically smallest equivalent string of `baseStr`.

Return *the lexicographically smallest equivalent string of* `baseStr` *by using the equivalency information from* `s1` *and* `s2`.

**Example 1:**

```
Input: s1 = "parker", s2 = "morris", baseStr = "parser"
Output: "makkek"
Explanation: Based on the equivalency information in s1 and s2, we can group their characters as [m,p], [a,o], [k,r,s], [e,i].
The characters in each group are equivalent and sorted in lexicographical order.
So the answer is "makkek".
```

**Example 2:**

```
Input: s1 = "hello", s2 = "world", baseStr = "hold"
Output: "hdld"
Explanation: Based on the equivalency information in s1 and s2, we can group their characters as [h,w], [d,e,o], [l,r].
So only the second letter 'o' in baseStr is changed to 'd', the answer is "hdld".
```

**Example 3:**

```
Input: s1 = "leetcode", s2 = "programs", baseStr = "sourcecode"
Output: "aauaaaaada"
Explanation: We group the equivalent characters in s1 and s2 as [a,o,e,r,s,c], [l,p], [g,t] and [d,m], thus all letters in baseStr except 'u' and 'd' are transformed to 'a', the answer is "aauaaaaada".
```

Input :- s1 = "parker" (0 1 2 3 4 5)
s2 = "morris"

baseStr = "parser"

Output :- "makker"

question says we can exchange same corresponding letter of two strings. (meaning both are equivalent)
meaning inplace of s1[0], we can write s2[0]
Ex: p/m.

we need to find a equivalent to base str. which is lexicographically the smallest.

"Parser" ⟶ baseStr.
↓ ↓ ↓ ↓ ↓ ↓
M a k k e k

as s1[i] is equivalent to s2[i]

P equivalent to M — s1[0] = s2[0]
and M < P (lexicographically) ∴ replace. (P with M)

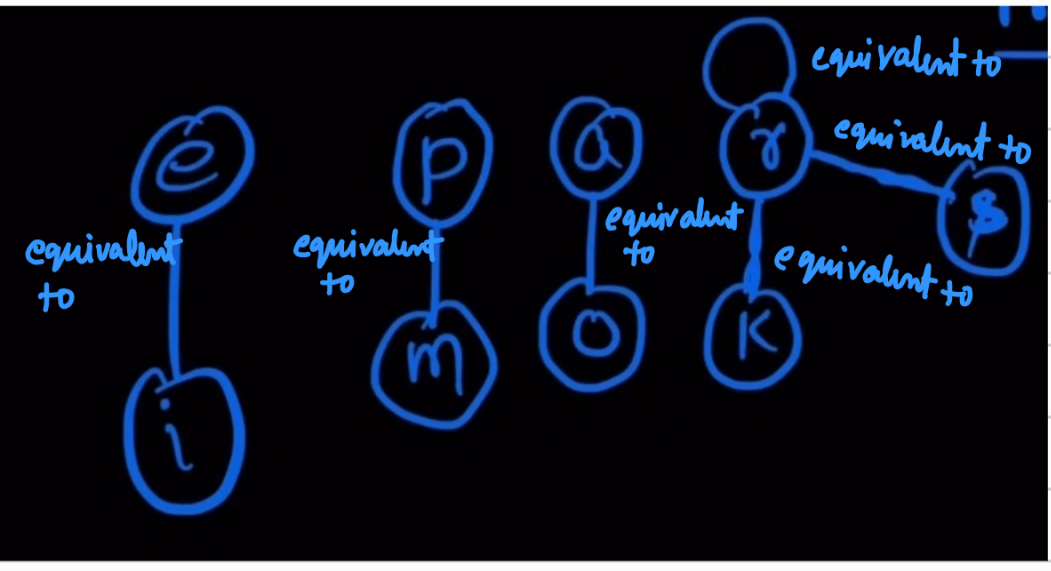
a equivalent to o — s1[1] = s2[1]
and a < o (lexicographically) ∴ ~~replace~~ don't replace.

r equivalent to r, K, b — s1[2] ("r") = s2[2] ("r") | s1[3] ("K") = s2[3] ("r")
and K < r (lexicographically) ∴ replace. (r with K) — s1[5] ("r") = s2[5] ("s")

s equivalent to r, K — s1[5] ("r") = s2[5] ("s")
and K < s (lexicographically) ∴ replace. (S with K)

why r? as s equivalent to r if a ≡ b
& r " " k & b ≡ c
∴ s " " k also . then a ≡ c } given in q . transitive rule .

"e" "i"
e equivalent to i s1[4] = s2[4]
and e < i (lexicographically) ∴ ~~replace~~ . don't replace .




r had 3 equivalent char r, s, k
∴ apply dfs on r & get min ( r, s, k )

Similarly we do this for all char,

⇒ 1. we need to make a mapping using s1 & s2
∴ let's create an adjacency list .

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| s1 = | p | a | r | k | e | r |
| s2 = | m | o | r | r | i | s |

P → M
M → P

a → o

o → a

r → r, k, s

k → r

s → r

e → i

i → e

2. take each char & apply dfs.
( for each char there will be dfs call )

baseStr = " P a r s e r "

↓

dfs.

minChar = P

check from P where we can go

∴ go to adj list.

P → m

m < P ∴ update min char.

minChar = ~~P~~ m.

m → P

already visited.

∴ create a vis array, if not visited then only visit.

∴ minchar = m

baseStr = " P a r s e r "

↓

m

Similarly for "a"

dfs on a.

minChar = a

a → o

a < o ∴ no updation.

o → a

alr visited.

∴ minChar = a

baseStr = "P a r s e r"

Similarly for "r"

dfs on r.

minChar = r

r → r

alr vis.

r → k

k < r ∴ update

minChar = k

k → r

alr vis.

r → s

r < k ∴ no updation

minChar = k.

baseStr = "P a r s e r"

Similarly for "s"

minChar = s

s → r

r < s ∴ update

minChar = $\infty$ r

r → r

alr vis.

r → K

K < r ∴ update

minChar = $\infty$ $\infty$ K

K → r

alr vis

r → s.

alr vis

∴ minChar = K.

baseStr = " P a r s e r "

m a K K

⋮

Similarly the dry run continues ....

for each char in baseStr there will be dfs call.

if baseStr.length = m.

& dfs time comp. O(V+E)

∴ Time Comp = m(V+E)

```cpp
//Approach-1 (DFS)
//T.C : O(m* ( V+E)), we try DFS m times
//S.C : O(V+E)
class Solution {
public:
    // dfs.
    char DFS(unordered_map<char, vector<char>> &adj, char curr, vector<int>& visited) {
        visited[curr-'a'] = 1;  → mark the char vis.
                  to find index.
        char minChar = curr;    → initially minChar is the curr char itself.

        for(char &v : adj[curr]) {

            if(visited[v-'a'] == 0)  → if not vis.
                minChar = min(minChar, DFS(adj, v, visited));  → min of every dfs call. (min char will get updated)
        }

        return minChar;  → return min char.
    }

    string smallestEquivalentString(string s1, string s2, string baseStr) {
        int n = s1.length();  → find out len → s1 & s2 len equal (given in Q)
        unordered_map<char, vector<char>> adj;  → adj list → for mapping equivalent chars.

        for(int i = 0; i<n; i++) {
            char u = s1[i];
            char v = s2[i];

            adj[u].push_back(v);
            adj[v].push_back(u);
        }   → filling adj list.


        int m = baseStr.length();  → baseStr len
        string result;  → to store ans.

        for(int i = 0; i<m; i++) {  → for every char in baseStr dfs will be called
            char ch = baseStr[i];  → char in baseStr.

            vector<int> visited(26, 0);  → visited arr → given in Q small case alphabets :. 26

            result.push_back(DFS(adj, ch, visited));  → putting the result of dfs call in answer
        }

        return result;  → printing answer.
    }
};
```

dfs calls return min char