DNA Genome Sequencing

locked

In this challenge, you need to implement the Trie data structure which is used for efficient pattern matching of strings. Given a set S of strings such that no string in S is a prefix of another string, the Trie data structure is organized in the form of a tree such that each internal node except the root is labelled with a character from strings in S, and it has exactly $|S|$ external nodes, each corresponding to a string in S. Once the trie is constructed, checking whether a string t is present in set S can be done in $O(|t|)$, i.e. linear in the size of t. Please refer to [Goodman] Chapter 12, Section 12.5.1 for more information on Tries.

We will use Tries to efficiently check whether a DNA genome is infected by a disease. A k-mer DNA genome is a sequence of strings, each of length k, formed using characters from `A` to `Z` (i.e. capitalized english alphabet). A disease is characterized by a set of malignant patterns S, where each pattern is a string of length k. Each pattern $s \in S$ is associated with a frequency $f(s)$ which will be a positive integer. We say that a DNA genome is infected by a disease if for each malignant pattern s of the disease, the DNA genome contains at least $f(s)$ instances of s.

Write a program to efficiently check using tries whether a DNA genome is infected by a disease.

Further, a disease may also mutate over time, where a mutation happens in the following manner:

A mutation is characterized by a map $M : F \rightarrow T$, which maps each string in F to a string of the same size in T. Each string in F will have length at most k, and it is guaranteed that no string in F will be a prefix of any other string in F. Now, to apply mutation M to a disease, we take each malignant pattern $s \in S$ of the disease, and check if a string in F is a prefix of s. If yes, then the matching prefix f is replaced by $M(f)$. After the mutation, the disease will have a new (mutated) set of malignant patterns.

Write a program to efficiently perform mutations using tries. Here, you will have to use strings in F to generate the trie, and then search for prefixes in each malignant disease pattern. Finally, generate another trie using the mutated disease pattern to check whether a DNA genome is infected by the mutated disease. Assume that the frequencies associated with each malignant pattern remain unchanged after the mutation.

It is recommended to use an object-oriented design, which will significantly simplify your implementation. Note that each tree node in the trie can have at most 26 children, and hence you will have to maintain a vector of `Node *` pointers at each internal node. External nodes can store additional information (frequencies/mutated strings).

Input Format

- The first line `k n p IsMutated` where
  - `k` is the length of each string in the DNA genome sequence (also the length of each malignant pattern).
  - `n` is the number of strings in the DNA genome sequence
  - `p` is the number of malignant patterns of the disease
  - `IsMutated` is either 0 or 1. If `IsMutated` is 1, then the disease goes through a mutation, otherwise not.
- The second line contains the entire DNA genome sequence, space-separated.
- Next, there will be `p` lines, each containing one malignant pattern s of the disease, followed by frequency $f(s)$.
- If `IsMutated` was 1, then the next line contains `m`, the number of target prefixes F to be mutated.

- Following this, there will be `m` lines, each containing a string in F, followed by its mutation in T.

Constraints

$1 \leq k \leq 10$

$1 \leq n, p, m \leq 10^5$

Complexity Constraints: Trie generation has no requirement. However, searching for pattern should now be O(nk). That is, linear in size of the DNA.

Output Format

- You need to print the indices of strings in the DNA genome sequence which match a malignant pattern of the disease, in increasing order. Assume that indexing starts from 0. All indices should be printed in one line, space separated. If there are no matches, print `No match found`.
- On the second line, you should print `Yes` if the DNA genome is infected by the disease. Otherwise, print `No`.
- If `IsMutated` was 1, then the third line should contain the indices of strings in the DNA genome sequence which match a malignant pattern of the mutated disease, in increasing order. If there are no matches, print `No match found`.
- If `IsMutated` was 1, then the fourth line should contain `Yes` if the DNA genome is infected by the mutated disease. Otherwise, print `No`.

Sample Input 0

```
4 8 2 0
ACGT ACGT AGGT ACGT ACGT GTAC GTAC ACCT
ACGT 2
GCTA 1
```

Sample Output 0

```
0 1 3 4
No
```

Sample Input 1

```
4 8 2 0
ACGT ACGT AGGT ACGT ACGT GCTA GTAC ACCT
ACGT 2
GCTA 1
```

Sample Output 1

```
0 1 3 4 5
Yes
```

Sample Input 2

```
10 10 2 1
AACGCTAGTT AACGCTAGTT AXCGCTAGTT ABCGCTAGTT AACGCTAGTT ABCGCTAGTT AACGCTAGTT AACGCTAGTT AACGCTAGTT AACGCTAGTT
AACGCTAGTT 1
AXCGCTAGTT 2
1
AXCGCTAGTT ABCGCTAGTT
```

Sample Output 2

```
0 1 2 4 6 7 8 9
No
0 1 3 4 5 6 7 8 9
Yes
```

Sample Input 3

```
4 8 2 0
ACGT ACGT AGGT ACGT ACGT GTAC GTAC ACCT
ABGT 2
GATA 1
```

Sample Output 3

```
No match found
No
```

Sample Input 4

```
5 7 2 1
ACGTA ACGTA AGGTC ACGTC ACGTA GTACG GTACA
ACGTG 2
GCTAC 1
1
ACGTG AGGTC
```

Sample Output 4

```
No match found
No
2
No
```

Sample Input 5

```
5 7 2 1
ACGTA ACGTA AGGTC ACGTC ACGTA GTACG GTACA
ACGTG 1
GCTAC 1
2
ACGTG AGGTC
GCTAC ACGTA
```

Sample Output 5

```
No match found
No
0 1 2 4
Yes
```

Sample Input 6

```
6 6 2 0
ACGTAT ACGTAT AGGTCG ACGTCA ACCTAG GTAGAC
ACGTAT 2
GCTACA 1
```

Sample Output 6

```
0 1
No
```

Sample Input 7

```
7 10 3 0
ACGTGTT ACGTGTT GTTACGT GTTACGA GTTACGT GTTACGT GTTACGT GTTACGT ACGTGTA GTTACGT
ACGTGTT 2
GTTACGT 2
ACGTGTA 1
```

Sample Output 7

```
0 1 2 4 5 6 7 8 9
Yes
```

Sample Input 8

```
12 10 2 1
KDDPPPKCCDEC KDDPPPKCCDEC KDDPKCPDPPKC KDDPKCPDPPKC KDDPPPKCCDEC KDDPPPKCCDEC KDDPKCPDPPKC KDDPPPKCCDEC
KDDPKCPDPPKC KDDPKCPDPPKC
KDDPPPKCCDEC 3
KDDPKCPDPPKC 2
1
KDDPPPKCCDEC KDDPKCPDPPKC
```

Sample Output 8

```
0 1 2 3 4 5 6 7 8 9
Yes
2 3 6 8 9
Yes
```

Sample Input 9

```
3 120 8 1
AGA
AGC
AGG
AGT
ATA
AAC
ATG
AAT
CAA
AAT
CAG
AAT
CCA
AGG
CCG
CCT
CGA
AGG
CGG
CGT
CTA
CTC
CTG
CAA
AAA
AAC
AAG
AAT
```

```
ACA
ACC
ACG
ACT
AGA
AGC
AGG
AGT
ATA
AAC
ATG
AAT
CAA
AAT
CAG
AAT
CCA
AGG
CCG
CCT
CGA
AGG
CGG
CGT
CTA
CTC
CTG
CAA
AAA
AAC
AAG
AAT
ACA
ACC
ACG
ACT
AGA
AGC
AGG
AGT
ATA
AAC
ATG
AAT
CAA
AAT
CAG
AAT
CCA
AGG
CCG
CCT
CGA
AGG
CGG
CGT
CTA
CTC
CTG
CAA
AAA
AAC
AAG
AAT
ACA
ACC
ACG
ACT
AGA
AGC
AGG
AGT
ATA
AAC
ATG
```

```
AAT
CAA
AAT
CAG
AAT
CCA
AGG
CCG
CCT
CGA
AGG
CGG
CGT
CTA
CTC
CTG
CAA
AAA 1
AAC 2
AAG 3
CTG 3
CTC 3
CCA 2
CCG 2
AAT 4
3
AAA ATA
AAC CAA
AAG AGG
```

Sample Output 9

```
5 7 9 11 12 14 21 22 24 25 26 27 37 39 41 43 44 46 53 54 56 57 58 59 69 71 73 75 76 78 85 86 88 89 90 91 101 103
105 107 108 110 117 118
Yes
2 4 7 8 9 11 12 13 14 17 21 22 23 27 34 36 39 40 41 43 44 45 46 49 53 54 55 59 66 68 71 72 73 75 76 77 78 81 85
86 87 91 98 100 103 104 105 107 108 109 110 113 117 118 119
Yes
```

Submissions: 89
Max Score: 100
Difficulty: Medium

Rate This Challenge:
☆☆☆☆☆

More

C++20

```cpp
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
class Charc     //This is a class representing each node of the trie
{
    public:
    char cH;    //The character of the node
    int dP;     //The depth of the node (shortest distance from the root of the trie)
    vector<Charc*> mP;   //A vector containing a adjacency array containing only the children nodes(null if no child)
    int fR;     //The frequency of a particular string(only for terminal nodes here)
    vector<int> indeX;   //vector containing the indices of the string in the input dna sequence array
    Charc(char dd, int gg)  //constructor
    {
        cH = dd;
        vector<Charc*> v(26);
        mP = v;
        for (int ii=0; ii<26; ii++)
        {
            mP[ii] = NULL;
        }
        dP = gg;
        fR = 1;
        vector<int> vV;
        indeX = vV;
    }
    Charc()    //constructor
    {   cH = 'A';
        fR = 1;
        vector<Charc*> v(26);
        mP = v;
        for (int ii=0; ii<26; ii++)
        {
            mP[ii] = NULL;
        }
        dP = 0;
        vector<int> vV;
        indeX = vV;
    }
    void ch_F(int xyZ){fR = xyZ;}   //changes the frequency
    void ch_mP(Charc* cR, int yY){mP[yY] = cR;}  //assigns/changes a particular index of mP to a particular node address
    void ch_dP(int dD){dP = dD;}   //changes/assigns the depth
    void ch_cH(char tT){cH = tT;}  //changes/assigns the character value
    void ch_iN(int uU) {indeX.push_back(uU);}   //inserts an string index in the input array to the vector indeX
};

void Dfs1(Charc* rooT, int kK, bool* bB) //This function is just to find if any terminal descendant of a given node is a non null node
{
    if (rooT!=NULL)
    {
        if (rooT->dP==kK)
        {
            (*bB) = false;
        }
        else
        {
            for (int ii=0; ii<26; ii++)
            {
                Dfs1(rooT->mP[ii],kK, bB);
            }
        }
    }
}

void Dfs(Charc* rooT, Charc* rooT1, int kK, vector<int>& v, bool* bB) //This function is to find the set of all matching pairs amongst the given dna genome sequence and malignant pattern
{
    Charc* g1 = rooT;
    Charc* g2 = rooT1;
    for (int ii=0; ii<26; ii++)
    {
        if (g1->mP[ii]!=NULL&&g2->mP[ii]!=NULL)
        {
            if (g1->mP[ii]->dP==kK&&g2->mP[ii]->dP==kK)
            {
                for (auto zZ:g1->mP[ii]->indeX)
                {
                     v.push_back(zZ);
                }
                if (g1->mP[ii]->fR<g2->mP[ii]->fR)
                {
                    (*bB) = false;
                }
            }
            else
            {
                Dfs(rooT->mP[ii], rooT1->mP[ii], kK, v, bB);
            }
        }
        else if (g1->mP[ii]==NULL&&g2->mP[ii]!=NULL)
        {
            Dfs1(g2->mP[ii], kK, bB);
        }
    }
}

void Merge(Charc* rooT, Charc* rooT1, int xX, int kK)  //This is to merge the descendant trie patterns of the malignant species with the mutated one after replacing the malignant matching prefix with the mutated one's
{
    if (kK>xX)
    {
        return;
    }
    else if (kK==xX)
    {
        rooT1->ch_F(rooT->fR + rooT1->fR);
        return;
    }
    for (int ii=0; ii<26; ii++)
    {
        if (rooT->mP[ii]!=NULL)
        {
            if (rooT1->mP[ii]==NULL)
            {
                rooT1->ch_mP(rooT->mP[ii], ii);
                rooT1->mP[ii]->ch_dP(rooT1->dP + 1);
            }
            else
            {
                if (kK==xX-1)
                {
                    rooT1->mP[ii]->ch_F(rooT1->mP[ii]->fR + rooT->mP[ii]->fR);
                }
                else
                {
                    Merge(rooT->mP[ii], rooT1->mP[ii], xX, kK+1);
                }
            }
        }
    }
    return;
}

int main()
{
    int nN, mM, kK, iM;
    cin>>kK>>nN>>mM>>iM;
    Charc* rooT = new Charc('A', 0);
    char cT;
    for (int ii=0; ii<nN; ii++)  //Taking the dna genome sequence as input and constructing the trie
    {
        int gG=1;
        Charc* g = rooT;
        for (int jj=0; jj<kK; jj++)
        {
            cin>>cT;
            if (g->mP[cT - 'A']==NULL)
            {
                Charc* charC = new Charc(cT, gG);
                g->ch_mP(charC, (int)(cT-'A'));
                if (jj==kK-1)
                {
                    g->mP[cT-'A']->ch_F(1);
                    g->mP[cT-'A']->ch_iN(ii);
                }
            }
            else
            {
                if (jj==kK-1)
                {
                    int myV = g->mP[cT-'A']->fR;
                    myV++;
                    g->mP[cT-'A']->ch_F(myV);
                    g->mP[cT-'A']->ch_iN(ii);
                }
            }
            g = g->mP[cT-'A'];
            gG++;
        }
    }

    Charc* rooT1 = new Charc('A', 0);
    for (int ii=0; ii<mM; ii++)  //Taking the malignant species as input and constructing the trie
    {
        int gG=1;
        Charc* g = rooT1;
        for (int jj=0; jj<kK; jj++)
        {
            cin>>cT;
            if (g->mP[cT-'A']==NULL)
            {
                Charc* charC = new Charc(cT, gG);
                g->ch_mP(charC, (int)(cT-'A'));
                if (jj==kK-1)
                {
                    int hh;
                    cin>>hh;
                    g->mP[cT-'A']->ch_F(hh);
                }
            }
            else
            {
                if (jj==kK-1)
                {
                    g->mP[cT-'A']->ch_F(0);
                }
            }
            g = g->mP[cT-'A'];
            gG++;
        }
    }
    bool bB = true;
    vector<int> v1;
    Dfs(rooT, rooT1, kK, v1, &bB);
    sort(v1.begin(), v1.end());
    if (v1.empty())
    {
        cout<<"No match found"<<endl;
    }
    else
    {
        for (int ii=0; ii<v1.size(); ii++)
        {
            cout<<v1[ii]<<" ";
        }
        cout<<endl;
    }
    if (bB==true)
    {
        cout<<"Yes"<<endl;
    }
    else
    {
        cout<<"No"<<endl;
    }

    if (iM==1)  // If mutation happens then we proceed as below
    {
        int fF;
        cin>>fF;
        string s1, s2;
        while(fF--)  //We take the mutated versions of the malignant species as input, as a pair
of the malignant species prefix with the corresponding mutated prefix, to be replaced with
        {
            cin>>s1>>s2;
            Charc* lL = rooT1;
            Charc* lL1 = rooT1;
            int rQ = 0;
            Charc* g = rooT1;
            bool bV = true;
            for (int ii=0; ii<s1.size(); ii++)
            {
                if (g->mP[s1[ii] - 'A']!=NULL)     // We keep on checking if the malignant
species prefix is present
                {
                    if (ii == s1.size()-1)
                    {
                        lL = g->mP[s1[ii] - 'A'];
                        lL1 = g;
                        rQ = s1[ii] - 'A';
                    }
                    g = g->mP[s1[ii] - 'A'];
                }
                else     //if not present we don't make any change
                {
                    bV = false;
                    break;
                }
            }
            if (bV = true)    //if present we replace that prefix with the mutated version
            {
                g = rooT1;
                lL1->mP[rQ] = NULL;
                for (int ii=0; ii<s1.size(); ii++)
                {
                    if (g->mP[s2[ii] - 'A']==NULL)
                    {
                        Charc* hY = new Charc(s2[ii], ii+1);
                        g->ch_mP(hY,(int)(s2[ii] - 'A'));

                        if (ii == s1.size()-1)
                        {
                            hY->ch_F(0);
                        }
                    }
                    g = g->mP[s2[ii] - 'A'];
                }
                Merge(lL, g, kK-s1.size(), 0);   //We modify the descendent trie of the remaining suffix of the malignant species by merging them with the suffix(if present) of the mutated version
            }
        }
        bool bB = true;
        vector<int> v1;
        Dfs(rooT, rooT1, kK, v1, &bB);
        sort(v1.begin(), v1.end());
        if (v1.empty())
        {
            cout<<"No match found"<<endl;
        }
        else
        {
            for (int ii=0; ii<v1.size(); ii++)
            {
                cout<<v1[ii]<<" ";
            }
            cout<<endl;
        }
        if (bB==true)
        {
            cout<<"Yes"<<endl;
        }
        else
        {
            cout<<"No"<<endl;
        }
    }
    return 0;
}

```

Line: 1 Col: 1

Upload Code as File   ☐ Test against custom input   Run Code   Submit Code

Interview Prep | Blog | Scoring | Environment | FAQ | About Us | Support | Careers | Terms Of Service | Privacy Policy |