Century Permutations and Patterns

locked

by pruthvishalcodi1

Editorial by PruthvishE

First things first, we are going to need the positions where each pattern is present in the string. So, we can just find all match positions using KMP algorithm (or any other string matching algorithm) and store the positions of matches for each pattern. For our purpose, we can preprocess and build an NXT*[j] table telling the position of the first match of ith pattern at or after jth position, since it makes sense to use the first match due to optimality.

The Naive solution for this problem shall be to iterate over all permutations of patterns and check for every permutation whether that permutation is matchable or not. This solution has complexity O(M!∗N) and will definitely time out for M=14.

Let us use Meet in the middle trick here.

Let us split each permutation into two equal (or nearly equal) parts. We can see that first M/2 patterns may be any subset of all patterns, so we can try all subsets of M patterns which is of size M/2. Now, These M/2 patterns may appear in any order and each shall be corresponding to a different permutation. So, Let us iterate over all permutation of all elements of current subsets. We can do the same for remaining M−M/2 patterns.

Now comes the interesting part.

Suppose we have a permutation of M / 2 patterns (Calling it left permutation) (the ones with on bit in bitmask) and a permutation of M − M / 2 patterns (calling it right permutation) (the ones with off bit in bitmask). How can we check if any pair of left and right permutation form a matchable permutation?

Let us suppose that p is the first position in the string S such that All the first M/2 patterns are present in String S[0,p] in order of left permutation such that no two patterns overlap. Also suppose that q is the last position in the string S such that all M−M/2 patterns are present in S[q,N−1] in order of right permutation such that no two patterns overlap.

The combination of these two permutations is valid if and only if p < q , since only that way all M patterns would be present in S without overlap for current permutation.

But trying every pair of permutation for every bitmask is essentially the same as iterating over all permutation since we check each permutation individually.

Statistics

Difficulty: Medium

Time Complexity: $O(CMM/2 * (N + (M/2)! * M) + M * N + sum(|Pi|))$

Required Knowledge: KMP String Matching, Meet in the middle and Partial Sums.

Publish Date: Jul 06 2019

But we can use an observation here. If a right permutation is valid with a left permutation with end position p , it shall also be valid with any position j ≤ p .

Hence, for every bitmask with M / 2 bits set, we can iterate over all permutations of M / 2 patterns and using prefix sum arrays, count the number of left permutations which end before or at a given position p for all positions p . Now iterating over all right permutations, we can easily count the number of left permutations which can be paired with current right permutation. We can increase our answer by the number of such left permutations for each right permutations and print the answer.

For ease of implementation, we can Build NXT table in the same manner, and then reverse both S and all patterns and find the position of matches on these reversed strings. That way, we can easily find the rightmost position such that all patterns of right permutation are present at or after that position without overlap, working in the same manner as we work with NXT table.

 Set by PruthvishE

Problem Setter's code :

```
#include <bits/stdc++.h>
using namespace std;

#define pb push_back

using ll = long long;
using ii = pair<int, int>;

const int N = 1e5 + 5, K = 14;
string S, P[K];
int n, k;
int nxt[K][N], rnxt[K][N];
int dp[1005][1 << K];

vector<int> FAIL(string pat) {

    int m = pat.size();
    vector<int> F(m + 1);
    int i = 0, j = -1;
    F[0] = -1;

    while (i < m) {
        while (j >= 0 && pat* != pat[j])
            j = F[j];
        i++, j++;
        F* = j;
    }

    return F;
}

vector<int> KMP_Search(string txt, string pat) {

    vector<int> F = FAIL(pat);
    int i = 0, j = 0;
    int n = txt.size(), m = pat.size();
    vector<int> ret;
    while (i < n) {
        while (j >= 0 && txt* != pat[j])
            j = F[j];
        i++, j++;
        if (j == m) {
            ret.pb(i - j);
            j = F[j];
        }
    }

    return ret;
}
```

```
vector<vector<int>> Match, rMatch;

void buildNext(int * arr, vector<int> matches) {
    //first matching >= i
    for (int i = 0; i <= n; i++)
        arr* = n;
    for (auto x : matches)
        arr[x] = x;
    for (int i = n - 2; i >= 0; --i)
        arr* = min(arr*, arr[i + 1]);
}
int calc[N];
int solve(int idx, int mask) {
    if (idx > n)
        return 0;
    if (mask == (1 << k) - 1)
        return 1;
    int &ret = dp[idx][mask];
    if (~ret)
        return ret;
    ret = 0;
    for (int j = 0; j < k; j++)
        if (mask >> j & 1 ^ 1) {
            ret += solve(nxt[j][idx] + P[j].size(), mask | (1 << j));
        }
    return ret;
}
void Stress1() {
    ///solution for subtask 2
    ///first brute force dp[index][mask];
    ///use next[index]
    ///can solve subtasks 1-2
    memset(dp, -1, sizeof dp);
    cerr << solve(0, 0) << '
';
}
int getFirst(vector & x) {
if (x.empty())
return n;
return x[0];
}
int getLast(vector & x) {
if (x.empty())
return -1;
return x.back();
}

void Stress2() {
    ///Solution for subtask 1
    if (k == 1) {
        cerr << !KMP_Search(S, P[0]).empty() << '
';
} else if (k == 2) {
vector x1 = KMP_Search(S, P[0]);
vector x2 = KMP_Search(S, P1 2);
int ans = 0;
if (getFirst(x1) + (int) P[0].size() <= getLast(x2))
ans++;
if (getFirst(x2) + (int) P1 2.size() <= getLast(x1))
ans++;
cerr << ans << '
';
} else {
vector x3 3;
x[0] = KMP_Search(S, P[0]);
x1 2 = KMP_Search(S, P1 2);
x2 6 = KMP_Search(S, P2 6);
int ans = 0;
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 3; j++) {
if (i == j)
continue;
for (auto v : x[3 - i - j]) {
//    cout << i << ' ' << getFirst(x*) << ' ' << j << ' ' << getLast(x[j])
```

```
 << ' ' << v << '
';
if (getFirst(x*) + (int) P*.size() <= v
&& v + (int) P[3 - i - j].size() <= getLast(x[j])) {
ans++;
break;
}
}
}
}
cerr << ans << '
';
}
}
int main() {
ios_base::sync_with_stdio(0);
cin.tie(0);

    cin >> n >> k;
    cin >> S;

    for (int i = 0; i < k; i++)
        cin >> P*;

    ///get matchings and build next array for them
    ///where nxt[j]* = next matching position for the j-th pattern which is
more than or equal i
    for (int i = 0; i < k; i++) {
        Match.pb(KMP_Search(S, P*));
        buildNext(nxt*, Match.back());
    }

    reverse(S.begin(), S.end());

    ///do the same as above for reverse strings and patterns
    for (int i = 0; i < k; i++) {
        reverse(P*.begin(), P*.end());
        rMatch.pb(KMP_Search(S, P*));
        buildNext(rnxt*, rMatch.back());
        reverse(P*.begin(), P*.end());
    }

    reverse(S.begin(), S.end());

    ll ans = 0;
    for (int mask = 0; mask < (1 << k); mask++)
        if (__builtin_popcount(mask) == k / 2) {
            ///process the normal
            ///get the indexes
            ///brute force on all permutations and find the minimum suffix
            ///needed to get all of these matched for each permutation
            ///the use partial sum to pre-process the results
            vector<int> v;
            for (int i = 0; i < k; i++)
                if (mask >> i & 1) {
                    v.pb(i);
                }
            memset(calc, 0, sizeof calc);
            do {
                int cur = 0;
                for (auto x : v) {
                    if (rnxt[x][cur] == n)
                        goto fin1;
                    cur = rnxt[x][cur] + P[x].size();
                }

                ///i have the last cur digits covered
                calc[cur]++;

                fin1: ;
            } while (next_permutation(v.begin(), v.end()));

            ///partial sum
            for (int i = 1; i <= n; i++)
                calc* += calc[i - 1];
```

```
            ///solve the flip
            v.clear();
            int flip = ((1 << k) - 1) ^ mask;
            for (int i = 0; i < k; i++)
                if (flip >> i & 1) {
                    v.pb(i);
                }
            ///get the indexes brute force on them find the minimum
            ///prefix to cover these patterns find all suffixes in the prev
ious calculation
            do {
                int cur = 0;
                for (auto x : v) {
                    if (nxt[x][cur] == n)
                        goto fin2;
                    cur = nxt[x][cur] + P[x].size();
                }
                ans += calc[n - cur];
                fin2: ;
            } while (next_permutation(v.begin(), v.end()));

        }

    cout << ans << '';

    return 0;
}
```