Let's Pull Some Strings

md.mottakin.chowdhury

July 2018

1 Problems

It is all about *strings*. **Strings**. **Strings** make up the universe.

1.1 Problem 1

What is the largest n such that a given string can be written as a concatenation of exactly n substring? For example, answer for *ababab* is 3 as it can be written as concatenation of 3 *ab*.

Solution: We can us KMP for this problem. After calculating the failure function, the answer can be found directly:

```
if(len%(len-pi[len-1]-1)==0)
    ans=len/(len-pi[len-1]-1);
else ans=1;
```

1.2 Problem 2

You are given n strings. Print all the substrings that occur in at least more than half of the input strings.

Solution: We concatenate all the n strings using a distinct non-alphabet character between each two strings. Now build the suffix array. For each suffix in the suffix array, we check previous n suffixes and find out if they are from at least greater than $n/2$ different strings. If they are, we find the length of LCP of current suffix to the suffix at which we get the match happen.

```
for(int i=n/2+1; i<=len; i++)
{
    int cnt=0, last=-1, match=0;
    bool found[105]; ms(found,false);
    // found tracks how many different strings have we found previously
    for(int j=i; j>=1; j--)
    {
        if(!found[from[sa[j]]]) match++, found[from[sa[j]]]=true;
        cnt++;
        if(match>n/2)
        {
            // so from sa[i] to sa[j], n/2+1 different input strings are there
            // so we need to check the lcp between sa[j] to sa[i]
            last=sa[j];
            break;
        }
        if(cnt==n) break;
    }

    if(match>n/2)
```

```
    {
        // taking the lcp to find maximum length
        int L=min(rak[sa[i]],rak[last]), R=max(rak[sa[i]],rak[last]);
        ret=max(ret,askRMQ(L+1,R));
        // look[] initialized to -1. It means that we should check the lcp
        // between sa[i] and look[i] to print answers
        look[i]=last;
    }
}
```

1.3 Problem 3

Given a string S and Q queries. In each query you are given an integer F. You need to find out number of substrings of S which occur at least F times in S.

Solution: After constructing the suffix array and then calculating the *height* array, we will find two arrays for each index of *height*[i] - $L[i]$ and $R[i]$. $(L[i], R[i])$ is the largest interval such that minimum of height array over the interval is equal to H, then it means that we have got a substring of length H, which repeats exactly $R[i] - L[i]$ times.

Now, we will iterate on the all possible values of *height* array. For each such value, we have stored its occurrences in vector $occ[val]$. Indices in $occ[val]$ has corresponding $L[index]$ and $R[index]$. We know the minimum in range $(L[i], R[i])$ is val. We will add its contribution to the corresponding position of the answer.

```
FOR(i,2,len+1)
{
    occ[height[i]].pb(i);
}
// calculating L[idx] and R[idx]. For R[idx], we iterate the height array reverse way
// aux[idx] is the corresponding L[idx] or R[idx]
void simulate(stack<pii> &st, int idx, int *aux, bool flag, int len)
{
    while(!st.empty() && st.top().first>=height[idx])
    st.pop();

    if(!st.empty()) aux[idx]=st.top().second;
    else
    {
        if(flag) aux[idx]=1;
        else aux[idx]=len+1;
    }

    st.push({height[idx],idx});
}
void solveRest(int len)
{
    FOR(i,1,len+1) // possible values in height array
    {
        // ptr is to ensure that overcount for same length does not happen
        int ptr=0;
        FOR(j,0,occ[i].size())
        {
            int idx=occ[i][j];
            int val=i;
            if(idx<ptr) continue;
```

```
                int lptr=L[idx];
                int rptr=R[idx];
                int minv=inf;
                // Not to add overlapping substrings again
                if(lptr!=1) minv=min(minv,i-height[lptr]);
                if(rptr!=len+1) minv=min(minv,i-height[rptr]);
                val=min(val,minv);
                // adding answer to the corresponding position
                // substring of length 'val' occurs (rptr-lptr) times
                // and all of its substrings
                ans[rptr-lptr]+=(ll)val*(rptr-lptr);
                ptr=rptr;
            }
        }
    }
```

1.4 Problem 4

Similar to previous problem. But this time we only need to find summation of the squares of occurrences of each distinct substring of the given string.

Solution: Notice that $(rptr - lptr)$ in previous code means number of occurrences of a substring of length val. So we add $val * (rptr - lptr)^2$ to our answer. In this case, substrings occurring only one time are not considered. To calculate that, we first consider all the substrings, there are $len * (len + 1)/2$ of them. Then subtract the number of substrings that occur more than once, there are $val * x$ for each substring of length val. We add this subtracted resulting value to the answer.

1.5 Problem 5

You are given a string s. Each substring of this string is a concatenation of one or more substrings. Print the maximum value k where k is the number of times you can concatenate a substring of s to build a larger substring of s. For example, for *babbabaabaabaabab*, we output 4 as *aba* can be concatenated four times to build a substring of this string.

```
    int query(int x, int y)
    {
        int L=rak[x];
        int R=rak[y];
        if(L>R) swap(L,R);
        return askRMQ(L+1,R);
    }
int solve(int n)
{
    int ret=0;
    for(int len=1; len<=n; len++)
    {
        for(int j=0; j+len<n; j+=len)
        {
            int curr=query(j,j+len);
            int k=j-(len-curr%len);
            curr=curr/len+1;
            if(k>=0 && query(k,k+len)>=len)
            {
                curr++;
            }
```

```
            ret=max(ret,curr);
        }
    }
    return ret;
}
```

1.6 Problem 6

You are given n strings. Print the length of the longest substring that occurs in all the strings at least twice and the occurrences do not overlap.

Solution: We do binary search on answer. For some possible answer len, we linearly go through the all suffixes in the sorted order and for each such suffix, we find out the farthest suffix such that the lcp between these two is greater or equal to our current length. Now, we check the minimum and maximum position of each strings that occurr inside this range. We can use rmq to find these values. If difference between all the minimum and maximum positions for each string is greater equal our current length, then we can achieve this answer.