Coding Club
IIT Guwahati

CRACKING THE PLACEMENT TESTS

A WEEKLY GUIDE CONTAINING TOPICS, QUESTIONS AND TRICKS COMMONLY ENCOUNTERED IN PLACEMENT TESTS

JULY 17, 2020
CODING CLUB, IIT GUWAHATI

Coding Club
IIT Guwahati

WEEK 3

STRINGS

A data structure so extensively used that it is taught as a separate chapter in every CP reference book under the topic of String Manipulation.

Under this topic we will cover various subtopics namely

1. Basics of String
2. Knuth Morris Pratt algorithm
3. String Hashing
4. Trie data structure
5. stringstream
6. Rabin Karp for string Matching
7. Suffix array [OPTIONAL, but RECOMMENDED]
8. Z algorithm [OPTIONAL]
9. Manacher algorithm [OPTIONAL]

There are highly powerful data structures (e.g. Suffix Automaton) and advanced algorithms (Aho Corasick) but these aren't relevant for the current scenario. I am excluding it, but the interested people may check it out.

BASICS

String Class in C++

Strings in C

Storage for Strings in C

Array of Strings in C++ (3 Different Ways to Create)

C++ string class and its applications

std::string::append vs std::string::push_back() vs Operator += in C++ [IMP]

C program to find second most frequent character

C Program to Sort an array of names or strings

C++ Program to remove spaces from a string

C++ program to concatenate a string given number of times

Comparing two strings in C++

Convert string to char array in C++

Extract all integers from string in C++

Searching for a pattern in a text

Using find()

```
int func(string str, string pattern){
    int ctr=0;
    size t found = str.find(pattern);
    if (found != string::npos){
        ctr++;
        label:
        found = str.find(pattern, found+1);
        if (found != string::npos){
            ctr++;
            goto label;
        }
    }
    return ctr;
}
```

Using naïve pattern matching

https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/

Worst Case Time complexity is quadratic O(n*m)

Reducing the time complexity to linear using KMP

1. CLRS [very detailed explanation given]

- Using DFA [CLRS]: 995-1002
- KMP algorithm [CLRS]: 1002-1006

2. cp-algorithms
   https://cp-algorithms.com/string/prefix-function.html

Coding Club
IIT Guwahati

IMPLEMENTATION as provided by Steven Halim in CP3

```
#include <cstdio>
#include <cstring>
#include <time.h>
using namespace std;

#define MAX N 100010

char T[MAX N], P[MAX N]; // T = text, P = pattern
int b[MAX N], n, m; // b = back table, n = length of T, m = length of P

void naiveMatching() {
  for (int i = 0; i < n; i++) { // try all potential starting indices
    bool found = true;
    for (int j = 0; j < m && found; j++) // use boolean flag `found'
      if (i + j >= n || P[j] != T[i + j]) // if mismatch found
        found = false; // abort this, shift starting index i by +1
    if (found) // if P[0 .. m - 1] == T[i .. i + m - 1]
      printf("P is found at index %d in T\n", i);
} }

void kmpPreprocess() { // call this before calling kmpSearch()
  int i = 0, j = -1; b[0] = -1; // starting values
  while (i < m) { // pre-process the pattern string P
    while (j >= 0 && P[i] != P[j]) j = b[j]; // if different, reset j using b
    i++; j++; // if same, advance both pointers
    b[i] = j; // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4
} }           // in the example of P = "SEVENTY SEVEN" above

void kmpSearch() { // this is similar as kmpPreprocess(), but on string T
  int i = 0, j = 0; // starting values
  while (i < n) { // search through string T
    while (j >= 0 && T[i] != P[j]) j = b[j]; // if different, reset j using b
    i++; j++; // if same, advance both pointers
    if (j == m) { // a match found when j == m
      printf("P is found at index %d in T\n", i - j);
      j = b[j]; // prepare j for the next possible match
} } }

int main() {
  strcpy(T, "I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN");
  strcpy(P, "SEVENTY SEVEN");
  n = (int)strlen(T);
  m = (int)strlen(P);

  //if the end of line character is read too, uncomment the line below
  //T[n-1] = 0; n--; P[m-1] = 0; m--;

  printf("T = '%s'\n", T);
  printf("P = '%s'\n", P);
  printf("\n");

  clock t t0 = clock();
  printf("Naive Matching\n");
  naiveMatching();
  clock t t1 = clock();
  printf("Runtime = %.10lf s\n\n", (t1 - t0) / (double) CLOCKS_PER_SEC);

  printf("KMP\n");
  kmpPreprocess();
  kmpSearch();
  clock_t t2 = clock();
  printf("Runtime = %.10lf s\n\n", (t2 - t1) / (double) CLOCKS_PER_SEC);

  printf("String Library\n");
  char *pos = strstr(T, P);
  while (pos != NULL) {
    printf("P is found at index %d in T\n", pos - T);
    pos = strstr(pos + 1, P);
  }
  clock_t t3 = clock();
  printf("Runtime = %.10lf s\n\n", (t3 - t2) / (double) CLOCKS_PER_SEC);

  return 0;
}
```

STRING HASHING

Comparing two 64-bit integers takes O(1) time whereas comparing two strings takes O(number_of_characters) time. What if we can make a number as a representative of the current string. Then all we have to do is compare two numbers which takes O(1) time. This is the key idea behind String Hashing.

There are different hash functions available. But here we will learn polynomial rolling hash function.

https://cp-algorithms.com/string/string-hashing.html

NOTE: Since the number of strings is infinite but the number of 32-bit or 64-bit integers is limited, two different strings can map to the same number. To avoid such instances,

1. use a random prime number apart from the common ones (1000000007 and 998244353) for hashing which will reduce the chances that the test cases that your code will be judged on contain a test case which contains a hash collision.
2. use pair hash (i.e. compute two different hashes for the strings and perform some algebraic operation(e.g xor) on the two hashes to get the final hash)

```
const int p1 = 31;
const int mod1 = 1e9 + 9;

long long compute_hash(string const& s) {
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % mod1;
        p_pow = (p_pow * p1) % mod1;
    }
    return hash_value;
}
```

```

const int p2 = 137;
const int mod2 = 1e9 + 7;

long long compute_hash2(string const& s) {
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % mod2;
        p_pow = (p_pow * p2) % mod2;
    }
    return hash_value;
}

struct pair_hash
{
  template <class T1, class T2>
  std::size_t operator () (std::pair<T1, T2> const &pair) const
  {
          std::size_t h1 = std::hash<T1>()(pair.first);
          std::size_t h2 = std::hash<T2>()(pair.second);

          return h1 ^ h2;
  }
};
```

NOTE: The prime number chosen for hashing should always be greater than the size of alphabet for the string.

SEARCHING FOR WORDS INSIDE A DICTIONARY

For efficient searching of words inside a dictionary, we use a trie.

This blog beautifully explains the benefits of a trie over a set of strings.

https://www.topcoder.com/community/competitive-programming/tutorials/using-tries/

Problem demonstrating the advantages and speed of a trie over set of strings:

https://cses.fi/problemset/task/1731

This can be solved using hashing too but the difference in running times will itself demonstrate the usefulness of a trie. Try it out.

I am attaching an AC code of the above problem for reference.

```cpp
#include <bits/stdc++.h>
using namespace std;
#define M1 1000000007
#define ll long long
#define REP(i,a,b) for(ll i=a;i<b;i++)
#define REPI(i,a,b) for(ll i=b-1;i>=a;i--)
#define F first
#define S second
#define MP make pair
#define V(a) vector<a>

struct trie
{
    map<char,struct trie *> mp;
};

static const ll N=1000005;
static string s,t;
static ll n,dp[N],pt=1;
static struct trie v[N],*x;

int main()
{
    ios::sync with stdio(0);
    cin.tie(0);
    cout.tie(0);

    cin>>s>>n;
    REP(i,0,n)
    {
        cin>>t;
        x=&v[0];
        REPI(j,0,t.size())
        {
            if(!x->mp.count(t[j]))
            {
                x->mp.insert(MP(t[j],&v[pt]));
                pt++;
            }
            x=x->mp[t[j]];
        }
        x->mp.insert(MP('$',&v[0]));
    }
    dp[0]=1;
    REP(i,1,s.size()+1)
    {
        dp[i]=0;
        x=&v[0];
        REPI(j,0,i)
        {
            if(!x->mp.count(s[j]))
            {
                break;
            }
            x=x->mp[s[j]];
            if(x->mp.count('$'))
            {
                dp[i]+=dp[j];
                dp[i]%=M1;
            }
        }
    }
    cout<<dp[s.size()];

    return 0;
}
```

STRINGSTREAM

https://www.geeksforgeeks.org/stringstream-c-applications/

important concept

makes code very short in problems where it is used

will come across some problems where it can be used while doing string questions on interviewbit.

WEEK 3

RABIN KARP FOR STRING MATCHING

https://cp-algorithms.com/string/rabin-karp.html

A linear time string matching algorithm based on string hashing.

SUFFIX ARRAY [OPTIONAL, but RECOMMENDED]

https://codeforces.com/edu/course/2/lesson/2

best tutorial on the topic. Taught by Pavel Mavrin (pashka)

https://cp-algorithms.com/string/suffix-array.html

Z ALGORITHM [OPTIONAL]

https://cp-algorithms.com/string/z-function.html

MANACHER ALGORITHM [OPTIONAL]

https://cp-algorithms.com/string/manacher.html

PROBLEMS

https://www.interviewbit.com/courses/programming/topics/strings/ [MUST]

https://www.geeksforgeeks.org/string-data-structure/

https://leetcode.com/tag/string/

NOTE: finding problems on specific topics is advised. Since there are thousands of problems on strings, it will be beneficial to practice problems topic wise. It can be easily googled.

cses.fi

Word Combinations

String Matching

Finding Borders

Finding Periods

Minimal Rotation

Longest Palindrome