# Leetcode struct and algorithm

**验证回文串**

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

**说明：**本题中，我们将空字符串定义为有效的回文串。

**示例 1:**

**输入:** "A man, a plan, a canal: Panama"

**输出:** true

**示例 2:**

**输入:** "race a car"

**输出:** false

class Solution {

public:

bool isPalindrome(string s)

{

int n=s.length();

if(n==0)

{

return true;

}

string temp="";

for(int i=0;i<n;i++)

{

if(s[i]>='0' && s[i]<='9' || s[i]>='a' && s[i]<='z')

{

temp+=s[i];

}

if(s[i]>='A' && s[i]<='Z')

{

temp+=(s[i]+32);

}

}

int m=temp.size();

for(int i=0;i<m/2;i++)

{

if(temp[i]!=temp[m-1-i])

{

return false;

}

}

return true;

}

};

**分割回文串**

给定一个字符串 *s*，将*s*分割成一些子串，使每个子串都是回文串。

返回 *s* 所有可能的分割方案。

**示例:**

**输入:** "aab"

**输出:**

[

["aa","b"],

["a","a","b"]

]

class Solution {

public:

bool isPali(string s){

for(int i = 0; i < s.length() / 2; i ++)

if(s[i] != s[s.length() - i - 1])

return false;

return true;

}

void dfs(vector<vector<string>> &ans, vector<string> &tmp, int n, string s){

if(n == s.length()){

ans.push\_back(tmp);

return;

}

for(int i = n; i < s.length(); i ++){

if(isPali(s.substr(n, i - n + 1))){

tmp.push\_back(s.substr(n, i - n + 1));

dfs(ans, tmp, i + 1, s);

tmp.pop\_back();

}

}

}

vector<vector<string>> partition(string s) {

vector<vector<string>> ans;

vector<string> tmp;

dfs(ans, tmp, 0, s);

return ans;

}

};

**单词拆分**

给定一个**非空**字符串 *s* 和一个包含**非空**单词列表的字典 *wordDict*，判定 *s* 是否可以被空格拆分为一个或多个在字典中出现的单词。

**说明：**

* 拆分时可以重复使用字典中的单词。
* 你可以假设字典中没有重复的单词。

**示例 1：**

**输入:** s = "leetcode", wordDict = ["leet", "code"]

**输出:** true

**解释:** 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

**示例 2：**

**输入:** s = "applepenapple", wordDict = ["apple", "pen"]

**输出:** true

**解释:** 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。

  注意你可以重复使用字典中的单词。

**示例 3：**

**输入:** s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]

**输出:** false

动态规划:

class Solution {

public:

bool wordBreak(string s, vector<string>& wordDict)

{

vector<bool> dp(s.size() + 1, false);

dp[0] = true;

for(int i = 1; i <= s.size(); i++)

for(int j = 0; j < i; j++)

{

if(dp[j] && find(wordDict.begin(), wordDict.end(), s.substr(j, i - j)) != wordDict.end())

// //从位置j开始，截取i-j长度的字符串，如果没有找到，那么将返回最后一个元素的后一个位置的迭代器

{

dp[i] = true;

break;

}

}

return dp[s.size()];

}

};

**实现 Trie (前缀树)**

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

**示例:**

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // 返回 true

trie.search("app"); // 返回 false

trie.startsWith("app"); // 返回 true

trie.insert("app");

trie.search("app"); // 返回 true

**说明:**

* 你可以假设所有的输入都是由小写字母 a-z 构成的。
* 保证所有输入均为非空字符串。

Trie树，又称单词查找树或键树，是一种树形结构，是一种哈希树的变种。

典型应用是

1. 用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。

2. 用于前缀匹配，比如我们在搜索引擎中输入待搜索的字词时，搜索引擎会给予提示有哪些前缀。

它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。缺点就是空间开销大

208. Implement Trie (Prefix Tree)这道题是实现一个前缀树，作为基础题啦

Add and Search Word - Data structure design这道题是把前缀树做一个简单的变形

472. Concatenated Word

[212. Word Search II](https://leetcode.com/problems/word-search-ii/#/description)

[421. Maximum XOR of Two Numbers in an Array](https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/#/description)

有如下特点：

1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符。

2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。

3. 每个节点的所有子节点包含的字符都不相同。

4. 如果字符的种数为n，则每个结点的出度为n，这也是空间换时间的体现，浪费了很多的空间。

5. 插入查找的复杂度为O(n)，n为字符串长度。

class TrieNode {

public:

//因为题目中是说字符都是小写字母。所以只用26个子节点就好

TrieNode \*child[26];

bool isWord;

TrieNode() : isWord(false){

for (auto &a : child) a = nullptr;

}

}; //这个是前缀树的每个节点的构造,其中isWord表示是否有以这个节点结尾的单词

//下面这个就是前缀树所包含的操作了

class Trie {

private:

TrieNode \*root;

public:

/\*\* Initialize your data structure here. \*/

Trie() {

root = new TrieNode();

}

/\*\* Inserts a word into the trie. \*/

//插入操作

void insert(string word) {

TrieNode \* nptr = root;

for (int i = 0; i<word.size(); i++){

//每次判断接下来的这个节点是否存在,如果不存在则创建一个

if (nptr->child[word[i] - 'a'] == NULL)

nptr->child[word[i] - 'a'] = new TrieNode();

nptr = nptr->child[word[i] - 'a'];

}

nptr->isWord = true;

}

/\*\* Returns if the word is in the trie. \*/

//搜索操作，判断某一个字符串是否存在于这个字典序列中

bool search(string word) {

if (word.size() == 0)

return false;

TrieNode \*nptr = root;

for (int i = 0; i<word.size(); i++){

if (nptr->child[word[i] - 'a'] == NULL)

return false;

nptr = nptr->child[word[i] - 'a'];

}

//判断是否有以当前节点为结尾的字符串

return nptr->isWord;

}

/\*\* Returns if there is any word in the trie that starts with the given prefix. \*/

//判断是否存在以prefix为前缀的字符串，其实跟search操作几乎一样啦，只不过最后返回的时候不用判断结尾节点是否为一个叶子结点

bool startsWith(string prefix) {

if (prefix.size() == 0)

return false;

TrieNode \*nptr = root;

for (int i = 0; i<prefix.size(); i++){

if (nptr->child[prefix[i] - 'a'] == NULL)

return false;

nptr = nptr->child[prefix[i] - 'a'];

}

return true;

}

};

**二进制求和**

给定两个二进制字符串，返回他们的和（用二进制表示）。

输入为**非空**字符串且只包含数字 1 和 0。

**示例 1:**

**输入:** a = "11", b = "1"

**输出:** "100"

**示例 2:**

**输入:** a = "1010", b = "1011"

**输出:** "10101"

char\* addBinary(char\* a, char\* b)

{

if(strlen(a)<strlen(b))

{

char\* c=a;

a=b;

b=c;

}

int m=strlen(a);

int n=strlen(b);

for(int i=0;i<m/2;i++)

{

char x=a[i];

a[i]=a[m-1-i];

a[m-1-i]=x;

}

for(int i=0;i<n/2;i++)

{

char x=b[i];

b[i]=b[n-1-i];

b[n-1-i]=x;

}

char\* res=(char\*)malloc(sizeof(char)\*(m+2));

int temp=0;

int ss;

int k=0;

for(int i=0;i<n;i++)

{

ss=(a[i]-'0')+(b[i]-'0')+temp;

if(ss>=2)

{

res[k]=ss%2+'0';

k++;

temp=1;

}

else

{

res[k]=ss+'0';

k++;

temp=0;

}

}

for(int i=n;i<m;i++)

{

ss=(a[i]-'0')+temp;

if(ss>=2)

{

res[k]=ss%2+'0';

k++;

temp=1;

}

else

{

res[k]=ss+'0';

k++;

temp=0;

}

}

if(temp==1)

{

res[k]='1';

k++;

}

for(int i=0;i<k/2;i++)

{

char x=res[i];

res[i]=res[k-1-i];

res[k-1-i]=x;

}

res[k]='\0';

return res;

}

C++

class Solution {

public:

string addBinary(string a, string b)

{

string res="";

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

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

if(a.size()<b.size())

{

string c=a;

a=b;

b=c;

}

int m=a.size();

int n=b.size();

int temp=0;

for(int i=0;i<n;i++)

{

int bit=(a[i]-'0')+(b[i]-'0')+temp;

if(bit>=2)

{

res+=(bit%2+'0');

temp=1;

}

else

{

res+=(bit+'0');

temp=0;

}

}

for(int i=n;i<m;i++)

{

int bit=(a[i]-'0')+temp;

if(bit>=2)

{

res+=(bit%2+'0');

temp=1;

}

else

{

res+=(bit+'0');

temp=0;

}

}

if(temp==1)

{

res+='1';

}

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

return res;

}

**实现strStr()**

实现 [strStr()](https://baike.baidu.com/item/strstr/811469) 函数。

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回  **-1**。

**部分匹配表**

**输入:** haystack = "hello", needle = "ll"

**输出:** 2

**示例 2:**

**输入:** haystack = "aaaaa", needle = "bba"

**输出:** -1

**说明:**

当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0 。这与C语言的 [strstr()](https://baike.baidu.com/item/strstr/811469) 以及 Java的 [indexOf()](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#indexOf(java.lang.String)) 定义相符。

KMP算法

部分匹配表：

1. void getNext(char \* p, int \* next)
2. {
3. next[0] = -1;
4. int i = 0, j = -1;
6. while (i < strlen(p))
7. {
8. if (j == -1 || p[i] == p[j])
9. {
10. ++i;
11. ++j;
12. next[i] = j;
13. }else{
14. j = next[j];
15. }
16. }

int KMP(char \* t, char \* p)

1. {
2. int i = 0;
3. int j = 0;
5. while (i < strlen(t) && j < strlen(p))
6. {
7. if (j == -1 || t[i] == p[j])
8. {
9. i++;
10. j++;
11. }
12. else
13. j = next[j];
14. }
16. if (j == strlen(p))
17. return i - j;
18. else
19. return -1;
20. }

**Sunday算法**

由Daniel M.Sunday在1990年提出，它的思想跟BM算法很相似：1

只不过Sunday算法是从前往后匹配，在匹配失败时关注的是主串中参加匹配的最末位字符的下一位字符。

如果该字符没有在模式串中出现则直接跳过，即移动位数 = 模式串长度 + 1；

否则，其移动位数 = 模式串长度 - 该字符最右出现的位置(以0开始) = 模式串中该字符最右出现的位置到尾部的距离 + 1。

下面举个例子说明下Sunday算法。假定现在要在主串”substring searching”中查找模式串”search”。

整个过程，我们只移动了两次模式串就找到了匹配位置，缘于Sunday算法每一步的移动量都比较大，效率很高

const int maxNum = 1005;

int shift[maxNum];

int Sunday(const string& T, const string& P) {

int n = T.length();

int m = P.length();

// 默认值，移动m+1位

for(int i = 0; i < maxNum; i++) {

shift[i] = m + 1;

}

// 模式串P中每个字母出现的最后的下标

// 所对应的主串参与匹配的最末位字符的下一位字符移动到该位，所需要的移动位数

for(int i = 0; i < m; i++) {

shift[P[i]] = m - i;

}

// 模式串开始位置在主串的哪里

int s = 0;

// 模式串已经匹配到的位置

int j;

while(s <= n - m) {

j = 0;

while(T[s + j] == P[j]) {

j++;

// 匹配成功

if(j >= m) {

return s;

}

}

// 找到主串中当前跟模式串匹配的最末字符的下一个字符

// 在模式串中出现最后的位置

// 所需要从(模式串末尾+1)移动到该位置的步数

s += shift[T[s + m]];

}

return -1;

}