**二分搜索**

**模板一 – lower bound**

定义 lower bound 为在给定升序数组中大于等于目标值的最小索引

这里的有序数组是nums

public static int lowerBound(int[] nums, int target) {

if (nums == null || nums.length == 0) return -1;

int lb = -1, ub = nums.length;

while (lb + 1 < ub) {

int mid = lb + (ub - lb) / 2;

if (nums[mid] < target) {

lb = mid;

} else {

ub = mid;

}

}

return lb + 1;

}

**模板二 - 最优解**

我们还可以利用二分搜索求最优解（最大值/最小值），通常这种题中只是隐含了『有序数组』，需要我们自己构造。

这里构造的有序数组是 int i = 0; i < 100;

有  *N* 条绳子，它们的长度分别为 *L*​*i*​​. 如果从它们中切割出 *K* 条长度相同的绳子的话，这  *K* 条绳子每条最长能有多长？答案保留到小数点后两位。

public static double solve(double[] nums, int K) {

double lb = 0.00, ub = 10e5 + 0.01;

// while (lb + 0.001 < ub) {

for (int i = 0; i < 100; i++) {

double mid = lb + (ub - lb) / 2;

if (C(nums, mid, K)) {

lb = mid;

} else {

ub = mid;

}

}

return lb;

}

public static boolean C(double[] nums, double seg, int k) {

int count = 0;

for (double num : nums) { //每一个num代表1整条绳子

count += Math.floor(num / seg);

}

//count为所有绳子切每段seg 的整数和

return count >= k; //判断是否大于k

}

}

方法C 只做一件事，给定数组nums, 判断是否能切割出K 条长度均为seg 的绳子。while 循环中使用lb + 0.001 < ub, 不能使用0.01, 因为计算mid 时有均值的计算，对于double 型数值否则会有较大误差。

**样例**

将binary search 问题转化成：寻找第一个或者最后一个，该target元素出现的位置的问题，,

def binary\_search(self, array, target):

if not array:

return -1

start, end = 0, len(array) - 1

while start + 1 < end:

mid = (start + end) / 2

if array[mid] == target:

start = mid

elif array[mid] < target:

start = mid

else:

end = mid

if array[start] == target:

return start

if array[end] == target:

return end

return -1

**最大公约数**

常用的方法为辗转相除法，也称为欧几里得算法。

a 6 b 4

gcd(4, 2) = gcd(2, 0)

**Java**

public static long gcd(long a, long b) {

return (b == 0) ? a : gcd(b, a % b);

}

更优的洗牌算法**Knuth-Durstenfeld Shuffle**

**每次从未处理的数据中随机取出一个数字，然后把该数字放在数组的尾部，即数组尾部存放的是已经处理过的数字**。

该算法类似于插入排序

从数组的最后一个数（下标为i）开始，进行随机取余（除数为i+1,确保下标不过界）

将得到的下标对应的元素和最后一个数交换

将最后一个数不在视为数组中的元素，继续循环

直到只剩下第一个元素为止

def shuffle(lis):

for i in range(len(lis) - 1, 0, -1):

p = random.randrange(0, i + 1)

lis[i], lis[p] = lis[p], lis[i]

return lis

r = shuffle([1, 2, 2, 3, 3, 4, 5, 10])

print(r)

判断两个字符串是否互为变位词

Given s = "abcd", t = "dcab", return true.  
Given s = "ab", t = "ab", return true.  
Given s = "ab", t = "ac", return false.

**题解1 - hashmap 统计字频**

def anagram(self, s, t):

return collections.Counter(s) == collections.Counter(t)

另一直接的解法是对字符串先排序，若排序后的字符串内容相同，则其互为变位词。

def anagram(self, s, t):

return sorted(s) == sorted(t)

变形

问B中的所有字符是否都在A中：比如B="AABC"包含两个「A」，而A="ABCD"只包含一个「A」，故返回false.

For A = "ABCD", B = "ACD", return true.

For A = "ABCD", B = "AABC", return false.

def compareStrings(self, A, B):

letters = collections.defaultdict(int)

for a in A:

letters[a] += 1

for b in B:

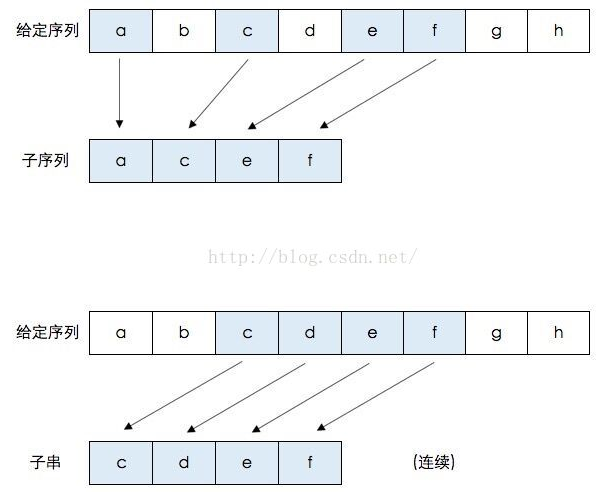
letters[b] -= 1

if b not in letters or letters[b] < 0:

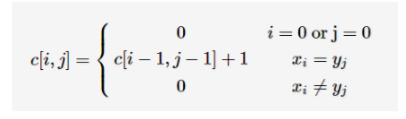
return False

return True

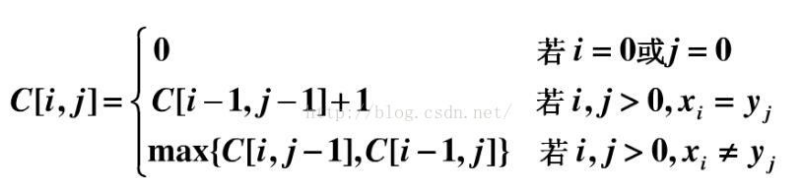
 首先需要科普一下，最长公共子序列（longest common sequence）和最长公共子串（longest common substring）不是一回事儿。什么是子序列呢？即一个给定的序列的子序列，就是将给定序列中零个或多个元素去掉之后得到的结果。什么是子串呢？给定串中任意个连续的字符组成的子序列称为该串的子串。给一个图再解释一下：

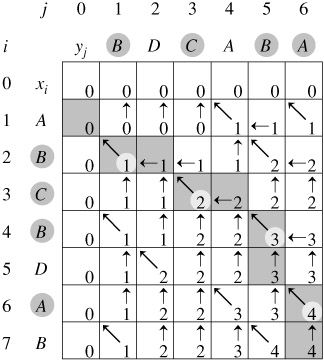


最长公共子串：



最长公共子序列：





int longestSubSeq(string x, string y) {

int \*\*dp = new int\*[x.size() + 1];

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

dp[i] = new int[y.size() + 1];

}

for (int i = 0; i <= x.size(); i++) dp[i][0] = 0;

for (int i = 0; i <= y.size(); i++) dp[0][i] = 0;

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

for (int j = 1; j <= y.size(); j++) {

if (x[i] == y[j]) {

dp[i][j] = dp[i - 1][j - 1] + 1;

}

else {

dp[i][j] = dp[i - 1][j] > dp[i][j - 1] ? dp[i - 1][j] : dp[i][j - 1];

}

}

}

return dp[x.size()][y.size()];

}

**Rotate String 旋转字符串**

给定一个字符串和一个偏移量，根据偏移量旋转字符串(从左向右旋转)

样例   
对于字符串 “abcdefg”.

offset=0 => “abcdefg”   
offset=1 => “gabcdef”   
offset=2 => “fgabcde”   
offset=3 => “efgabcd”

常见的翻转法应用题，仔细观察规律可知翻转的分割点在从数组末尾数起的offset位置。先翻转前半部分，随后翻转后半部分，最后整体翻转。

def rotateString(self, A, offset):

if A is None or len(A) == 0:

return A

offset %= len(A)

before = A[:len(A) - offset]

after = A[len(A) - offset:]

# [::-1] means reverse in Python

A = before[::-1] + after[::-1]

A = A[::-1]

return A

**Valid Palindrome**

给定字符串，判断该字符串是否是“回文”。只考虑字母和数字，并忽略字母大小写。   
注：空字符串也是回文

解法1：逐个比较首尾字符是否相等最为便利，即常见的『两根指针』技法。此题忽略大小写：  
def isPalindrome(self, s):

if not s:

return True

l, r = 0, len(s) - 1

while l < r:

# find left alphanumeric character

if not s[l].isalnum():

l += 1

continue

# find right alphanumeric character

if not s[r].isalnum():

r -= 1

continue

# case insensitive compare

if s[l].lower() == s[r].lower():

l += 1

r -= 1

else:

return False

#

return True

解法2：

def isPalindrome(self, s):

cleanlist = [c for c in s.lower() if c.isalnum()]

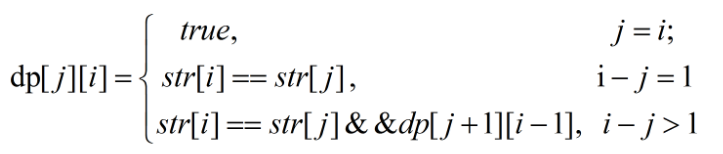
return cleanlist == cleanlist[::-1]

最长回文子串；

https://blog.csdn.net/shineboyxxb/article/details/52079360

https://www.cnblogs.com/leavescy/p/5878336.html

解法二： 动态规划



1. **class** Solution {
2. **public**:
3. string longestPalindrome(string s) {
4. **const** **int** n=s.size();
5. **bool** dp[n][n];
6. fill\_n(&dp[0][0],n\*n,**false**);
7. **int** max\_len=1; //保存最长回文子串长度
8. **int** start=0;//保存最长回文子串起点
9. **for**(**int** i=0;i<s.size();++i)
10. {
11. **for**(**int** j=0;j<=i;++j)
12. {
13. **if**(i-j<2)
14. dp[j][i]=(s[i]==s[j]);
15. **else**
16. dp[j][i]=(s[i]==s[j] && dp[j+1][i-1]);
17. **if**(dp[j][i] && max\_len<(i-j+1))
18. {
19. max\_len=i-j+1;
20. start=j;
21. }
22. }
23. }
24. **return** s.substr(start,max\_len);
25. }
26. };

解法三：中心扩展法

中心扩展就是把给定的字符串的每一个字母当做中心，向两边扩展，这样来找最长的子回文串。算法复杂度为O(N^2)。

但是要考虑两种情况：

1、像aba，这样长度为奇数。

2、想abba，这样长度为偶数。

public static String findLongestPalindrome2(String s){

int len = s.length();

int maxlength = 0;

int start = 0;

// 类似于aba这种情况，以i为中心向两边扩展

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

int j = i - 1;

int k = i + 1;

while(j >= 0 && k < len && s.charAt(j) == s.charAt(k)){

if(k - j + 1 > maxlength){

maxlength = k - j + 1;

start = j;

}

j --;

k ++;

}

}

// 类似于abba这种情况，以i，i+1为中心向两边扩展

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

int j = i;

int k = i + 1;

while(j >= 0 && k <len && s.charAt(j) == s.charAt(k)){

if(k - j + 1 > maxlength){

maxlength = k - j + 1;

start = j;

}

j --;

k ++;

}

}

if(maxlength > 0)

return s.substring(start, start + maxlength);

return null;

}

解法4 马拉车 有点复杂不看了