Nothing is Harder, Except Math and …

2021年8月25日 8:49

Kadane，KMP， Morris遍历

==========

// Kadane
给与一个数组，求数组中最大连续 子数组的和。
暴力算法：

```
        for i in range(0,length):
            for j in range(i+1,length+1):
                sub = nums[i:j]
                sub_sum = sum(sub)
                if sub_sum>MAX:
                    MAX = sub_sum
        return MAX
```

优化到O(n^2):

```
        for i in range(0,length):
            sum_sub = 0
            for j in range(i, length):
                sum_sub += nums[j]
                if sum_sub>MAX:
                    MAX = sum_sub
        return MAX
```

上面的代码中，我们是以 某个节点为开头的所有子序列：[a],[a,b],[a,b,c]，然后[b],[b,c]。
这样，在计算的时候，需要对所有子数组之和进行计算和比较。

改变对子数组的遍历方式，以 子序列的结束节点为基准，先遍历以某个结点为结束的所有子序列。[a,b],[b] [a,b,c],[b,c],[c] 等。
这样，我们想获得 以c为结束点的子序列的信息时，可以利用之前的以b为结束点的子序列信息， 已有[a,b]的情况下，加上c就是[a,b,c]。 sum[i] = sum[i - 1] + arr[i]

新的遍历方式可以产生递推关系，使得 当前问题的解 可以在 先前问题的解的基础上 获得。

dp关键有3点，定义子问题，递推基(问题规模在最简单的情况下的解是什么)，递推关系(如何通过之前的子问题的解来获得当前解)

常见的定义子问题的方式有2种，定义目标问题为子问题，定义非目标问题为子问题，目标问题的解可以通过所保存的所有子问题的解来获得。

先尝试第一种方式来定义子问题，我们将问题抽象问array[0, n-1]的最大子数组之和maxSubSum(n-1)，n代表数组长度。子问题就是求 array[0,i]的最大子数组之和maxSubSum(i)，我们用数组dp来记录子问题的解。递推基为 dp[0] = arr[0].递推关系需要考虑maxSubSum(i-1)和maxSubSum(i)的关系。这样的递推关系很难获得。例如，数组[-2,1,-3,4,-1,2,1,-5,4]递推基为dp[0]=-2，而dp[1]=1, dp[2]=1, dp[3]=4, dp[3]和dp[2]之间的关系并不明确。

所以尝试使用第二种方式来定义子问题。我们将子问题定义为 求以i为终止下标的子数组之和的最大值，这样最终可以通过比较 以下标0为终止下标的子数组的最大值，为下标1为终止下标的子数组的最大值，以下标2为终止下标的子数组的最大值。。以下标n-1为终止下标的子数组的最大值。递推基为dp[0]=arr[0],递推关系：如果dp[i-1]<0, arr[i]>0,则dp[i]=arr[i]， 如果dp[i-1]<0,arr[i]<0,则dp[i]=arr[i],如果dp[i-1]>0, arr[i]<0,dp[i]=dp[i-1]+arr[i]， 如果dp[i-1]>0,arr[i]>0,则dp[i]=dp[i-1]+arr[i]。最后，原始问题的解就是max(dp)
递推关系可以简化为： dp[i]=max(dp[i-1],arr[i],arr[i])
时间O(n),空间O(n)

```
    dp[0] = nums[0]
    for i in range(1, length):
        dp[i] = max(dp[i-1]+nums[i], nums[i])
    return max(dp)
```

kadane是在动态规划的基础上进一步优化，使用一根指针保存以i为结尾的子数组和的最大值，另一根指针保存迄今为止的子数组和的最大值。
时间O(n),空间O(1)

```
    max_ending_here = max_sub_sum = nums[0]
    for i in range(1,length):
        max_ending_here = max(max_ending_here+nums[i],nums[i])
        max_sub_sum = max(max_ending_here, max_sub_sum)
    return max_sub_sum
```

实际应用场景
计算机视觉中，通过kadane算法来检测图像中最亮区域的最高分数子序列

有些其他的算法题可以转化为最大子数组之和的问题，用kadane算法求解。如LT121。

LT121来了：

```
        int maxCur = 0, maxSoFar = 0;
        for(int i = 1; i < prices.length; i++) {
            maxCur = Math.max(0, maxCur += prices[i] - prices[i-1]);
            maxSoFar = Math.max(maxCur, maxSoFar);
        }
        return maxSoFar;
```

https://leetcode.com/problems/best-time-to-buy-and-sell-stock/discuss/39038/Kadane's-Algorithm-Since-no-one-has-mentioned-about-this-so-far-%3A)-(In-case-if-interviewer-twists-the-input)

==========

// Fenwick

==========

// Segment

==========

// KMP
LPS which is Longest Prefix also Suffix

i	0	1	2	3	4	5	6	7	8
S[i]	A	B	A	B	C	A	B	A	B
LPS	0	0	1	2	0	1	2	3	4

i	0	1	2	3	4	5	6	7	8
S[i]	A	A	B	A	A	B	A	A	A
LPS	0	1	0	1	2	3	4	5	2

。。以当前char作为结尾的 所有 subStr 中 是 整个string 的 suffix的 且 长度最长的 那个 subStr 的长度 就是 LPS 的值。。并且suffix 还不能包含 当前char (第二个例子中i=1时，subStr可以是AA，但是LPS 是 1)。

```
    private int[] computeKMPTable(String pattern) {
        int i = 1, j = 0, n = pattern.length();
        int[] lps = new int[n];
        while (i < n) {
            if (pattern.charAt(i) == pattern.charAt(j)) {
                lps[i++] = ++j;
            } else {
                if (j != 0) j = lps[j - 1]; // try match with longest prefix
suffix
                else i++; // don't match -> go to next character
            }
        }
        return lps;
    }
```

。。上面是构造 LPS
。。尾部(j) 操作。
。。j代表 前面某个字符的 lps长度。

以上面的第二个LPS例子为例：
i1 j0 : if 成立，所以 lps[1] = 1
i2 j1: if 不成立， 里面的if成立， j=lps[0]=0
i2 j0: if不成立， 里面的if不成立， i++
i3 j0: if 成立，lps[3] = 1
i4 j1: if 成立， lps[4] = 2
l5 j2: if成立， lps[4] = 3
i6 j3: 成立
i7 j4: 成立
i8 j5: if不成立，内层if成立， j = lps[4] = 2
i8 j2: 成立。

if成立能理解。

j = lps[j - 1];

lps保存的是以 这个下标为结尾的substr集合 中 最长的 可以作为 整个string的prefix的 那个substr的长度
j是 上一个char的 最长lps长度。
假设当前下标是 x， 并且此时 不满足 if。 x-1的时候是满足 if的。 并且 x-1的lps是j
说明 s[0, j-1] 等于 s[x-j, x-1] .
现在不满足if。
降级到lps[j-1] 。。。 不知道为什么要 lps[j-1]。。。

```
        int[] lps = computeKMPTable(needle);
        int i = 0, j = 0, n = haystack.length(), m = needle.length();
        while (i < n) {
            if (haystack.charAt(i) == needle.charAt(j)) {
                ++i; ++j;
                if (j == m) return i - m; // found solution
            } else {
                if (j != 0) j = lps[j - 1]; // try match with longest prefix
suffix
                else i++; // don't match -> go to next character of `haystack`
string
            }
        }
```

。上面是使用。真正的搜索 首次匹配。

```
        int[] lps = new int[n];
        for (int i = 1, j = 0; i < n; i++) {
            while (j > 0 && pattern.charAt(i) != pattern.charAt(j)) j =
```

```
lps[j - 1];
            if (pattern.charAt(i) == pattern.charAt(j)) lps[i] = ++j;
        }
```

。另一种 创建 lps的方法。

```
        vector<int> lps(n, 0);
        for (int i = 1, len = 0; i < n;) {
            if (needle[i] == needle[len]) {
                lps[i++] = ++len;
            } else if (len) {
                len = lps[len - 1];
            } else {
                lps[i++] = 0;
            }
        }
```

。。另一种创建 lps

==========

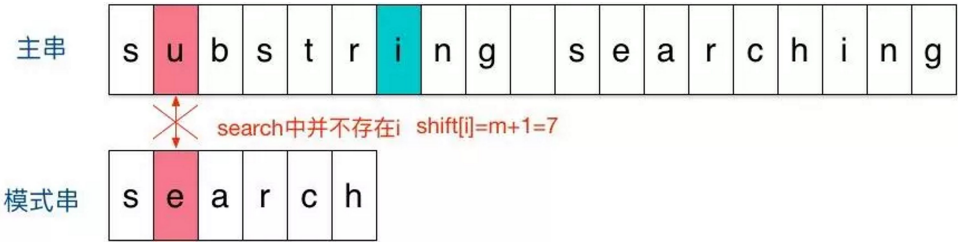
// Sunday
字符串匹配- Sunday

KMP不常用，BM常用，Sunday在其基础上做了一些改动。

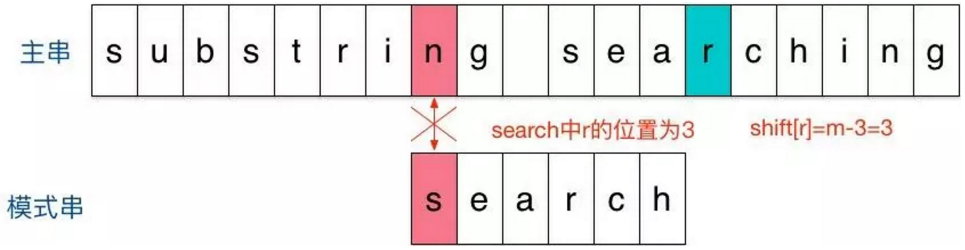
从前往后扫描模式串，思路更像是对 坏字符 策略的升华。关注的是主串中 参与匹配的最末字符(并非正在匹配的)的下一位。
。。只有BM是从后往前，其他都是 从前往后。

Sunday只有一个启发策略

当遇到不匹配的字符时，如果 关注的字符 没有在模式串中出现则直接跳过。即移动位数=子串长度+1



当遇到不匹配的字符时，如果 关注的字符 在模式串中也存在，则移动位数=模式串长度-该字符最右出现的位置(以0开始) 或 移动位数=模式串中该字符最右出现的位置到尾部的距离+1



缺点：

主串: baaaabaaaabaaaabaaaa

模式串: aaaaa

此时时间复杂度 O(m*n)

。。。？上面的例子为什么是 m*n？ 第一次比较就不相等，此时第二个b 在 模式串中并不存在，则会直接移动 5+1个位置啊。 估计 模式串应该是 baaaaa

Sunday算法的移动取决于子串，但这个子串重复很多的时候，就非常糟糕。

时间复杂度

KMP	O(m+n)
BM	O(m/n) - O(m*n)
Sunday	O(m/n) - O(m*n)

实际使用中，Sunday 比 BM略优。

==========

// BM 算法 // Boyer-Moore

与KMP从前往后扫描模式串不同，BM算法是从后往前对模式串进行扫描与主串进行匹配的。

核心是2个启发策略：

1 坏字符算法

当出现一个坏字符时，BM算法向右移动模式串，让模式串中最靠右的对应字符与坏字符相对，然后继续匹配。

坏字符算法有2种情况：

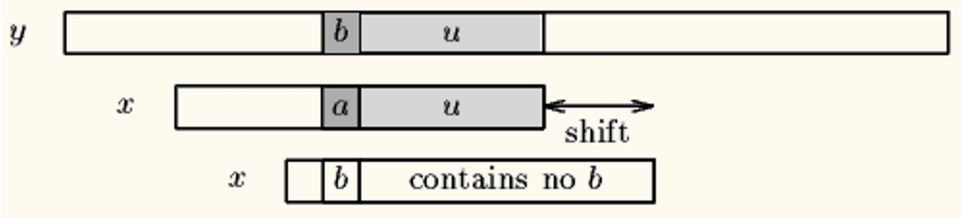
1 模式串中有对应的坏字符时，让模式串中最靠右的对应字符与坏字符相对，(BM不可能走回头路，因为走回头路，移动距离就是负数，肯定不是最大移动步数了)

。。为什么不是最左的。感觉最右的话，前面的b或许还有机会。不，应该没有了。下次的坏字符就不是这个b了。。

。。还有，如果是 xxaxxxbxxx这种，a不匹配的情况下，用b岂不是倒退了？可能是括号里的，不可能回头路吧，可能是必须大于等于1？

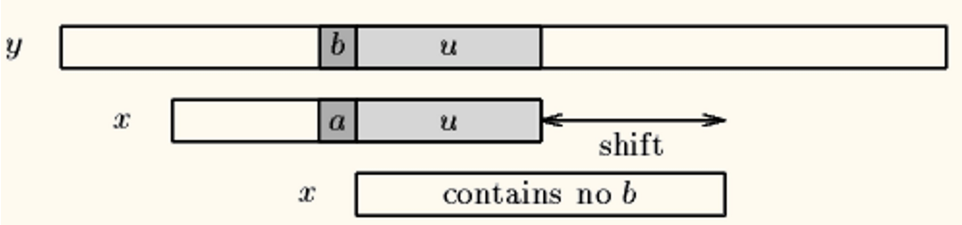
。。不，BM算法 从后往前的。就是 模式串从后往前，整体还是从前往后。

。。所以下面的 u 是成功匹配部分。



2 模式串中不存在坏字符，那么直接右移整个模式串长度的步数

。。看图，不是整个模式串长度，是 已匹配的长度。

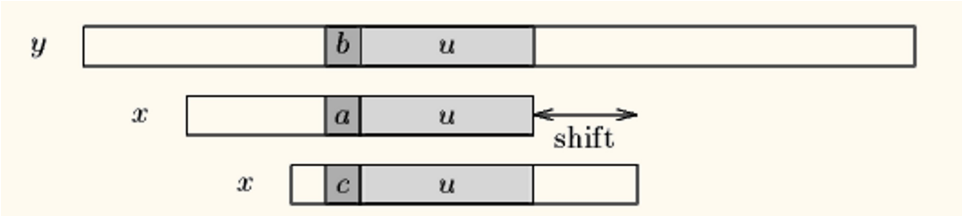


2 好后缀算法

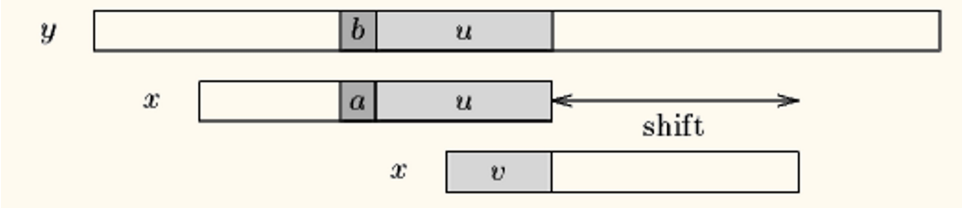
如果程序匹配了一个好后缀，并且在模式串中还有另外一个相同的后缀或后缀的部分，那把下一个后缀或部分移动到当前后缀位置。

即，模式串的后u个字符和主串已经匹配了，但是接下来的一个字符不匹配，如果说后u个字符在模式串其他位置也出现过或部分出现，我们将模式串右移到前面的u个字符或部分和最后u个字符或部分相同的位置，如果说后u个字符在模式串其他位置完全没有出现，那么就直接右移整个模式串。

1 模式串中有子串和好后缀完全匹配，则将最靠右的那个子串移动到好后缀的位置继续进行匹配。



2 如果不存在和好后缀完全匹配的子串，则在好后缀中找具有如下特征的最长子串，使得P[m-s, m] = P[0, s]



3 如果完全不存在和好后缀匹配的子串，则右移整个模式串。

3 移动规则

每次向右移动模式串的距离是 max(shift(好后缀), shift(坏字符))

时间复杂度

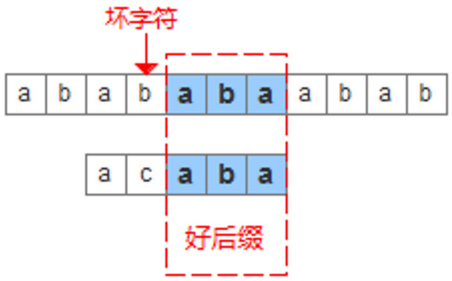
KMP: O(m+n)

BM: O(m/n) - O(m*n)

。。好后缀，坏字符，这种要预先计算，不，应该是cache的懒计算。

BM算法，从后往前扫描模式串使得它更好地利用了"后缀"，BM算法的启发策略也使得模式串可以更加有效率的移动。

经典的BM算法其实是对后缀蛮力匹配算法的改进。为了实现更快移动模式串，BM算法定义了两个规则，好后缀规则和坏字符规则

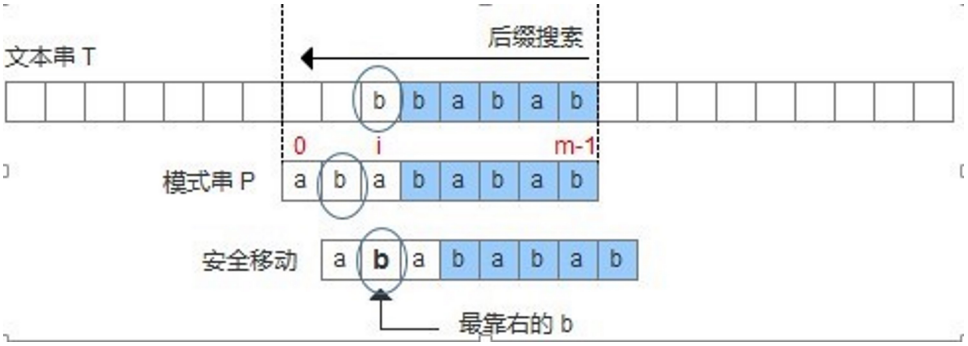


利用好后缀和坏字符可以大大加快模式串的移动距离，不是简单的++j，而是j+=max(shift(好后缀), shift(坏字符))

shift(坏字符)分为两种情况

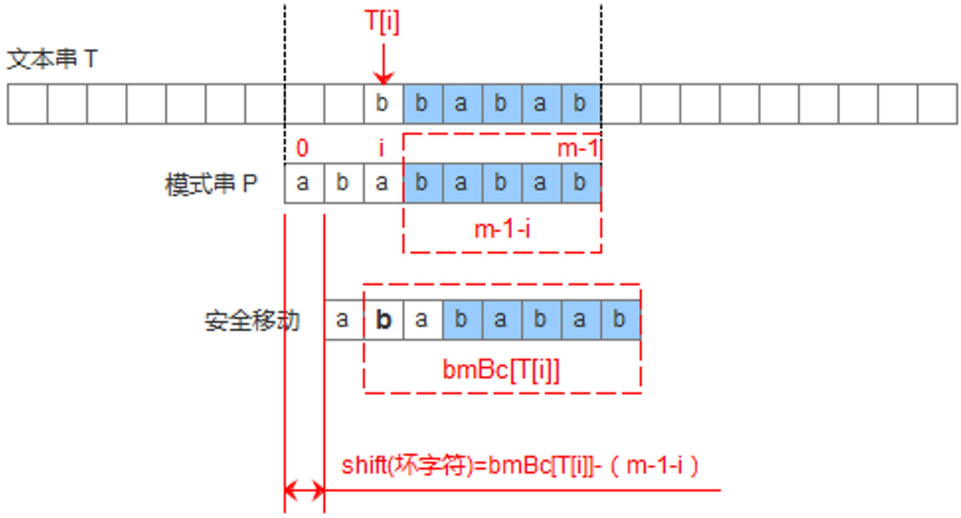
坏字符没出现在模式串中，这时可以把模式串移动到坏字符的下一个字符

坏字符出现在模式串中，这时可以把模式串第一个出现的坏字符和母串的坏字符对齐，当然，这样可能造成模式串倒退移动



此处配的图是不准确的，因为显然加粗的那个b并不是”最 靠右的”b。而且也与下面给出的代码冲突！论文的意思是最右边的。

为了用代码来描述上述的两种情况，设计一个数组bmBc['k']，表示坏字符‘k’在模式串中出现的位置距离模式串末尾的最大长度，那么当遇到坏字符的时候，模式串可以移动距离为：shift(坏字符) = bmBc[T[i]]-（m-1-i)。




```
void preBmBc(char *x, int m, int bmBc[]) {
   int i;
   for (i = 0; i < ASIZE; ++i)
      bmBc[i] = m;
   for (i = 0; i <= m - 1; ++i)
      bmBc[x[i]] = m - i - 1;
}
```

ASIZE是指字符种类个数，为了方便起见，就直接把ASCII表中的256个字符全表示了，哈哈，这样就不会漏掉哪个字符了。

第一个for循环处理上述的第一种情况，这种情况比较容易理解就不多提了。第二个for循环，bmBc[x[i]]中x[i]表示模式串中的第i个字符。bmBc[x[i]] = m - i - 1;也就是计算x[i]这个字符到串尾部的距离。

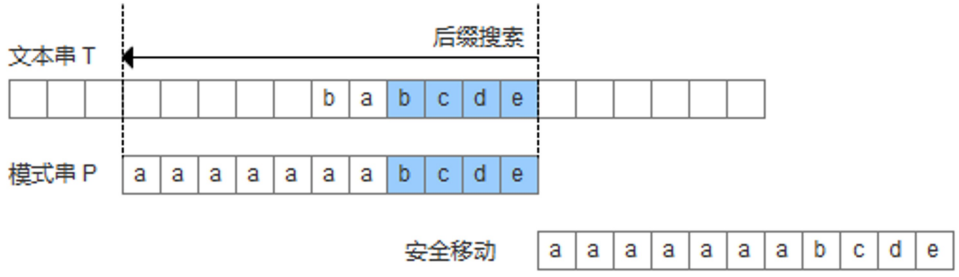
为什么第二个for循环中，i从小到大的顺序计算呢？哈哈，技巧就在这儿了，原因在于就可以在同一字符多次出现的时候以最靠右的那个字符到尾部距离为最终的距离。当然了，如果没在模式串中出现的字符，其距离就是m了。

shift（好后缀）分为三种情况

模式串中有子串匹配上好后缀，此时移动模式串，让该子串和好后缀对齐即可，如果超过一个子串匹配上好后缀，则选择最靠左边的子串对齐。

模式串中没有子串匹配上后后缀，此时需要寻找模式串的一个最长前缀，并让该前缀等于好后缀的后缀，寻找到该前缀后，让该前缀和好后缀对齐即可

模式串中没有子串匹配上后后缀，并且在模式串中找不到最长前缀，让该前缀等于好后缀的后缀。此时，直接移动模式到好后缀的下一个字符



为了实现好后缀规则，需要定义一个数组suffix[]，其中suffix[i] = s 表示以i为边界，与模式串后缀匹配的最大长度，如下图所示，用公式可以描述：满足P[i-s, i] == P[m-s, m]的最大长度s。

```
void suffixes(char *x, int m, int *suff)
{
    suff[m-1]=m;
    for (i=m-2; i>=0; --i){
        q=i;
        while(q>=0&&x[q]==x[m-1-i+q])
            --q;
        suff[i]=i-q;
    }
}
```

有了suffix数组，就可以定义bmGs[]数组，bmGs[i] 表示遇到好后缀时，模式串应该移动的距离，其中i表示好后缀前面一个字符的位置（也就是坏字符的位置），构建bmGs数组分为三种情况，分别对应上述的移动模式串的三种情况

```
void preBmGs(char *x, int m, int bmGs[]) {
   int i, j, suff[XSIZE];
```

```
   suffixes(x, m, suff);
   for (i = 0; i < m; ++i)
      bmGs[i] = m;
   j = 0;
   for (i = m - 1; i >= 0; --i)
      if (suff[i] == i + 1)
         for (; j < m - 1 - i; ++j)
            if (bmGs[j] == m)
               bmGs[j] = m - 1 - i;
   for (i = 0; i <= m - 2; ++i)
      bmGs[m - 1 - suff[i]] = m - 1 - i;
}
```

BM算法

```
void BM(char *x, int m, char *y, int n) {
   int i, j, bmGs[XSIZE], bmBc[ASIZE];

   /* Preprocessing */
   preBmGs(x, m, bmGs);
   preBmBc(x, m, bmBc);

   /* Searching */
   j = 0;
   while (j <= n - m) {
      for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
      if (i < 0) {
         OUTPUT(j);
         j += bmGs[0];
      }
      else
         j += MAX(bmGs[i], bmBc[y[i + j]] - m + 1 + i);
   }
}
```

==========

// Morris Traversal

时间O(n)，空间O(1)

利用树的 叶节点 的 左右子节点 为空 来压缩空间。

如果cur无左孩子，cur向右移动（cur=cur.right）
如果cur有左孩子，找到cur左子树上最右的节点，记为mostright

如果mostright的right指针指向空，让其指向cur，cur向左移动（cur=cur.left）
如果mostright的right指针指向cur，让其指向空，cur向右移动（cur=cur.right）

可以实现 pre/in/post order 遍历。
后序遍历比较复杂 。。。都很复杂，后序特别复杂。。

。。感觉就是 把 下一个该访问的 节点 放到 前一个节点的 子节点上。
。。后序遍历 最后一个 是 父节点，左右子节点都是非空的， 所以 特比复杂。
。。先序 中序， 先序最后一个是 右叶子节点， 它的子节点是空的。 中序最后一个是 。。还是 右叶子节点。。

==========

==========

// Rabin-Karp 2个人的名字。
==========

//Fisher-Yates Algorithm and Knuth Shuffle
==========

// flood-fill

==========

// sweep line

==========

// Manacher 最长回文 O(n)

在进行Manacher算法时，字符串都会进行一个字符处理，比如输入的字符为acbbcbds，用“#”字符处理之后的新字符串就是#a#c#b#b#c#b#d#s#。

回文半径和回文直径：因为处理后回文字符串的长度一定是奇数，所以回文半径是包括回文中心在内的回文子串的一半的长度，回文直径则是回文半径的2倍减1。比如对于字符串"aba"，在字符 'b' 处的回文半径就是2，回文直径就是3。

最右回文边界R：在遍历字符串时，每个字符遍历出的最长回文子串都会有个右边界，而R则是所有已知右边界中最靠右的位置，也就是说R的值是只增不减的。

回文中心C：取得当前R的第一次更新时的回文中心。由此可见R和C时伴生的。

半径数组：这个数组记录了原字符串中每一个字符对应的最长回文半径。

悟了一半：
用arr[]保存 以 下标为中心的 最大 回文
就是 如果本次要检查的 下标A 在 之前的最大回文B里，那么 就按照B的中心对称到 B的左半部分，dp下。
如果 下标A 在最大回文B外， 那么 硬算。
如果 下标A在 B内，但是还可以超过B，复用B内的，然后B外的硬算。
。。但是如果最大回文 只是 3个字符， 那么 A不好弄啊。就是 最大回文B，为什么是哪样的

```
//开始从左到右遍历
for (int i = 0; i < len; i++) {
    //第一步直接取得可能的最短的回文半径，当i>R时，最短的回文半径是1，反之，
    最短的回文半径可能是i对应的i'的回文半径或者i到R的距离
    pArr[i] = R > i ? min(R - i, pArr[2 * C - i]) : 1;
    //取最小值后开始从边界暴力匹配，匹配失败就直接退出
    while (i + pArr[i]<len && i - pArr[i]>-1) {
        if (chaArr[i + pArr[i]] == chaArr[i - pArr[i]]) {
            pArr[i]++;
        }
        else {
            break;
        }
    }
    //观察此时R和C是否能够更新
    if (i + pArr[i] > R) {
        R = i + pArr[i];
        C = i;
    }
    //更新最大回文半径的值
    maxn = max(maxn, pArr[i]);
```

```
}
。。。复制的
-----------------------------------------

if i >= R: # Case 1
    d[i] = 0
    从d[i]逐步继续扩展，求d[i]
else:
    if d[i'] < R - i: # Case 2
        d[i] = d[i']
    else if d[i'] = R - i: # Case 3
        d[i] = d[i']
        从d[i]逐步继续扩展，求d[i]
    else: # Case 4
        d[i] = R - i

---------------------
    for i in range(len(s)):
        if i<MaxRight:
            RL[i]=min(RL[2*pos-i], MaxRight-i)
        else:
            RL[i]=1
        #尝试扩展，注意处理边界
        while i-RL[i]>=0 and i+RL[i]<len(s) and s[i-RL[i]]==s[i+RL[i]]:
            RL[i]+=1
        #更新MaxRight,pos
        if RL[i]+i-1>MaxRight:
            MaxRight=RL[i]+i-1
            pos=i
        #更新最长回文串的长度
        MaxLen=max(MaxLen, RL[i])
    return MaxLen-1




-----------------------------------

https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/




===================================
```

// LT1334
//Floyd: 14ms
//Dijkstra: 32ms
//SPFA: 64ms
//Bellman: 251ms

// Prim, Kruskal

==========

Moore Voting Algorithm

==========

==========

Hierholzer's algorithm

希尔霍尔策算法是数学家卡尔·希尔霍尔策在1873年提出的一种寻找欧拉回路的算法 。比起另一种著名的Fleury算法而言，希尔霍尔策算法更加高效，能够达到图的总边数的线性次复杂度。

下面给出来自Harris的《Combinatorics and Graph Theory》中对该算法的描述 ：现给出一个欧拉图G，求欧拉回路。

选定G中一个环，称其为R1，标记R1的边，并记i为1。
如果Ri已经包含G中所有边，则停止搜索，显然Ri已经是一个欧拉环路。

否则，取Ri中一个点vi，满足vi有一条未被标记的边，记作ei。
从vi和ei出发，寻找一个环Qi，标记Qi上的所有边
使用Qi，创建一条新的环Ri+1
i的值加一，并回到步骤二，如此重复。

==========

```
    // int lengthOfLIS(vector<int>& nums) {
    //     vector<int> sub;
    //     for (int x : nums) {
    //         if (sub.empty() || sub[sub.size() - 1] < x) {
    //             sub.push_back(x);
    //         } else {
    //             auto it = lower_bound(sub.begin(), sub.end(), x); // Find
the index of the smallest number >= x
    //             *it = x; // Replace that number with x
    //         }
    //     }
    //     return sub.size();
    // }
```

==========

==========

372. Super Pow

Your task is to calculate a^b mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.

中国余数定理 (Chinese Remainder Theorem)

费马小定理（。。&& 百度 联想的 其他定理。。。）

https://leetcode.com/problems/super-pow/discuss/84475/Fermat-and-Chinese-Remainder

If the modulus weren't 1337 = 7 * 191 but a prime number p, we could use Fermat's little theorem to first reduce the exponent to e = b % (p-1) and then compute the result as ae % p. Oh well, we can do it for 1337's prime factors 7 and 191 and then combine the two results with the Chinese remainder theorem. I'll show my derivation of the magic constants 764 and 574 after the solutions below.

1337 only has two divisors 7 and 191 exclusive 1 and itself, so judge if a has a divisor of 7 or 191, and note that 7 and 191 are prime numbers, phi of them is itself - 1, then we can use the Euler's theorem, see it on wiki https://en.wikipedia.org/wiki/Euler's_theorem, it's just Fermat's little theorem if the mod n is prime.

see how 1140 is calculated out:
phi(1337) = phi(7) * phi(191) = 6 * 190 = 1140

==========

==========

水塘抽样(Reservoir Sampling)

蓄水池抽样算法(Reservoir Sampling)

如果接收的数据量小于m，则依次放入蓄水池。
当接收到第i个数据时，i >= m，在[0, i]范围内取以随机数d，若d的落在[0, m-1]范围

内，则用接收到的第i个数据替换蓄水池中的第d个数据。
重复步骤2。

当处理完所有的数据时，蓄水池中的每个数据都是以m/N的概率获得的。

当i<=m时，数据直接放进蓄水池，所以第i个数据进入过蓄水池的概率=1。
当i>m时，在[1,i]内选取随机数d，如果d<=m，则使用第i个数据替换蓄水池中第d个数据，因此第i个数据进入过蓄水池的概率=m/i。
当i<=m时，程序从接收到第m+1个数据时开始执行替换操作，第m+1次处理会替换池中数据的为m/(m+1)，会替换掉第i个数据的概率为1/m，则第m+1次处理替换掉第i个数据的概率为(m/(m+1))*(1/m)=1/(m+1)，不被替换的概率为1-1/(m+1)=m/(m+1)。依次，第m+2次处理不替换掉第i个数据概率为(m+1)/(m+2)...第N次处理不替换掉第i个数据的概率为(N-1)/N。所以，之后第i个数据不被替换的概率=m/(m+1)*(m+1)/(m+2)*...*(N-1)/N=m/N。
当i>m时，程序从接收到第i+1个数据时开始有可能替换第i个数据。则参考上述第3点，之后第i个数据不被替换的概率=i/N。
结合第1点和第3点可知，当i<=m时，第i个接收到的数据最后留在蓄水池中的概率=1*m/N=m/N。结合第2点和第4点可知，当i>m时，第i个接收到的数据留在蓄水池中的概率=m/i*i/N=m/N。综上可知，每个数据最后被选中留在蓄水池中的概率为m/N。

维护一个大小为M的数组. 记当前接收的是第N个数据(从1开始).

如果N<=M, 直接插入
如果N>M, 就取一个1~N之间的随机数index. 如果index在1~M之间, 则用新接收的数据替换第index个数据; 否则丢弃.

分布式的蓄水池抽样

假设有K个机器, 每个机器维护大小为M的数组, 并记录该机器接受的数据总数Ni.

当机器获取新数据时, 进行单机的蓄水池抽样.
当进行采样时, 重复M次以下操作:
取随机数d在[0,1)之间, 记N=Sum(Ni | i=1...K)
若d<N1/N则从第一个机器上等概率抽取一个元素.
若N1/N<=d<(N1+N2)/N则从第二个机器上等概率抽取一个元素
依此类推.

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========