

# Nothing is Harder, Except Math and ...

2021年8月25日 8:49

Kadane, KMP, Morris遍历

<https://queue.acm.org/topics.cfm>

=====

// 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**，我们将问题抽象为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數子序列

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 prefix 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)。

。。以当前char 的前一个char 作为结尾的所有 substr 中是 string 的前缀的最长的长度就是 LPS 的值。

。。就是匹配到当前的 char 不相等时, 此时知道前面的 substr 如果是 string 的前缀, 那么就可以跳过前缀的判断, 直接从这个前缀后面开始。

。。下面的形参是 pattern。

。。主要是绿色的 j = lps[j-1] 。。 dp 了已有的结果, 进入这个分支代表 j-1 是匹配的, [j] 不等于 [i], 现在要设置 [i] 的值, 但是 [i] != [j], 但是能肯定的是 [i-1]==[j-1] (如果 j > 0 的话。), 所以有一个 j!=0 的判断。。所以 dp 了一个 kmp 算法。尝试用 [j-1] 为结尾的所有 substr 中是 string 的前缀的最长的 substr 来尝试继续匹配。

```

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

i5 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
```

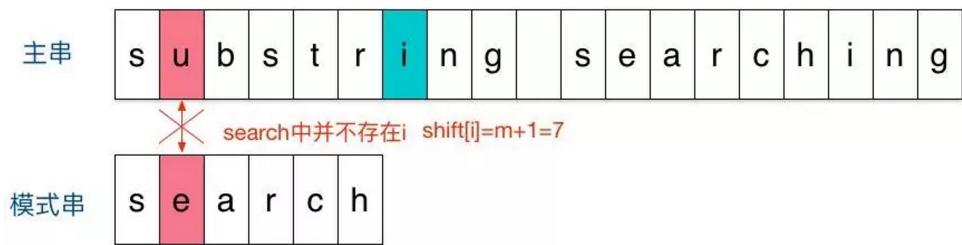
KMP不常用，BM常用，Sunday在其基础上做了一些改动。

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。

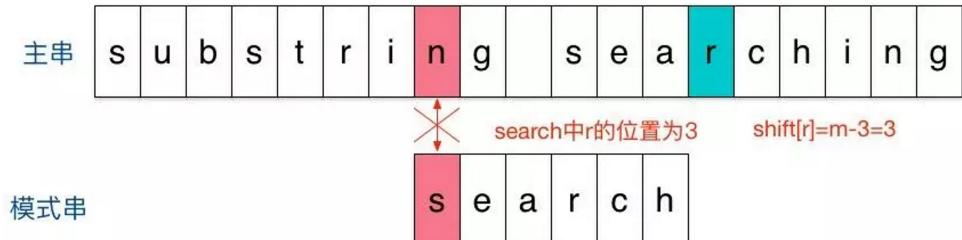
。。只有BM是从后往前，其他都是 从前往后。

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 baaaaaa

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。

LT0028

```

int lta(string haystack, string needle)
{
    int arr[123] = { 0 };
    int sz1 = haystack.size();
    int sz2 = needle.size();
    if (sz2 > sz1)
        return -1;
    for (int i = 0; i < sz2; ++i)
    {
        arr[needle[i]] = sz2 - i; // 最后出现的位置 到 模式串的尾巴 + 1.
    }
}

```

```

for (int i = 'a'; i <= 'z'; ++i)
    if (arr[i] == 0)
        arr[i] = sz2 + 1;      // 没有出现 则等于 模式串.size + 1

for (int i = 0; i < sz1; )
{
    for (int j = 0; j < sz2; ++j)
    {
        if (haystack[i + j] != needle[j])
            goto AAA;
    }
    return i;

AAA:
    i += (i + sz2 < sz1) ? arr[haystack[i + sz2]] : 1;          // 检
查 i + sz2 这个 char
}
return -1;
}

```

=====

// 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：

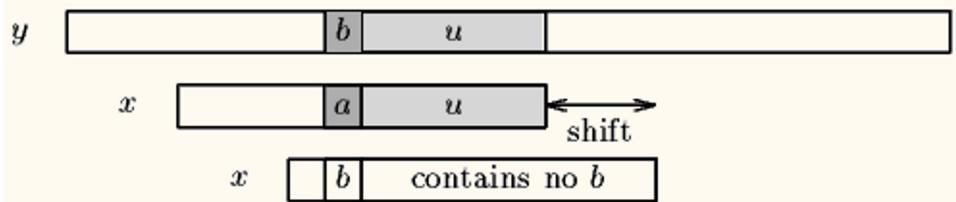
1 模式串中有对应的坏字符时，让模式串中最靠右的对应字符与坏字符相对，(BM不可能走回头路，因为走回头路，移动距离就是负数，肯定不是最大移动步数了)

。。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？

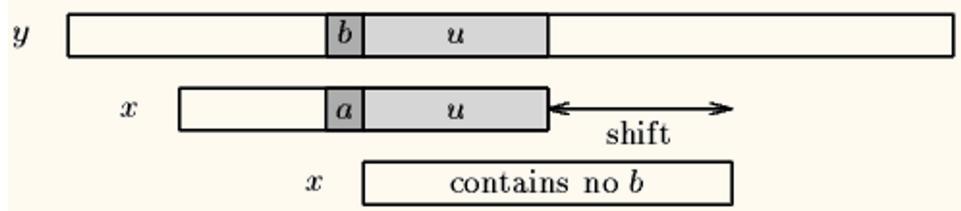
。。不，BM算法 从后往前的。就是 模式串从后往前，整体还是从前往后。

。。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。

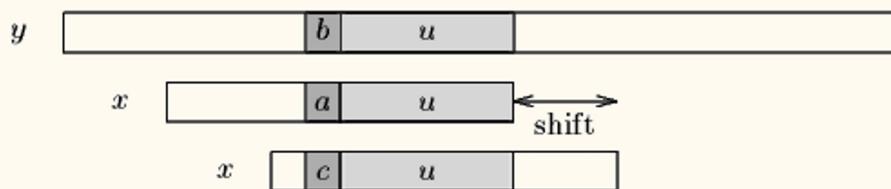


## 2 好后缀算法

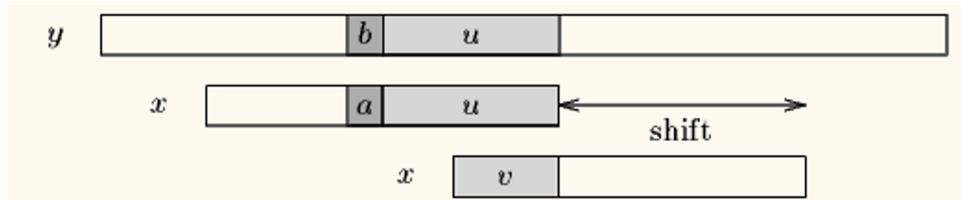
如果程序匹配了一个好后缀，并且在模式串中还有另外一个相同的后缀或后缀的部分，那把下一个后缀或部分移动到当前后缀位置。

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  $\max(\text{shift(好后缀)}, \text{shift(坏字符)})$

时间复杂度

KMP:  $O(m+n)$

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

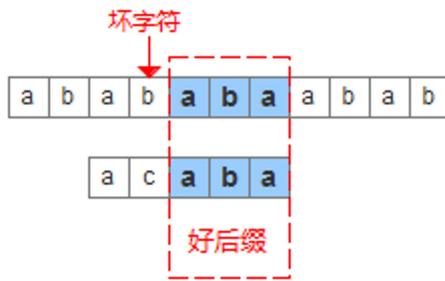
。。好后缀，坏字符，这种要预先计算，不，应该是cache的懒计算。

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



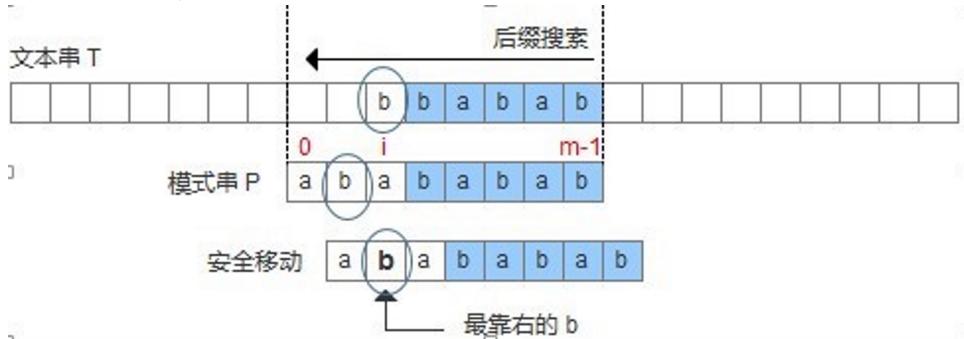
利用好后缀和坏字符可以大大加快模式串的移动距离，不是简单的 $++j$ ，而是 $j+=\max(\text{shift}(\text{好后缀}), \text{shift}(\text{坏字符}))$

$\text{shift}(\text{坏字符})$ 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

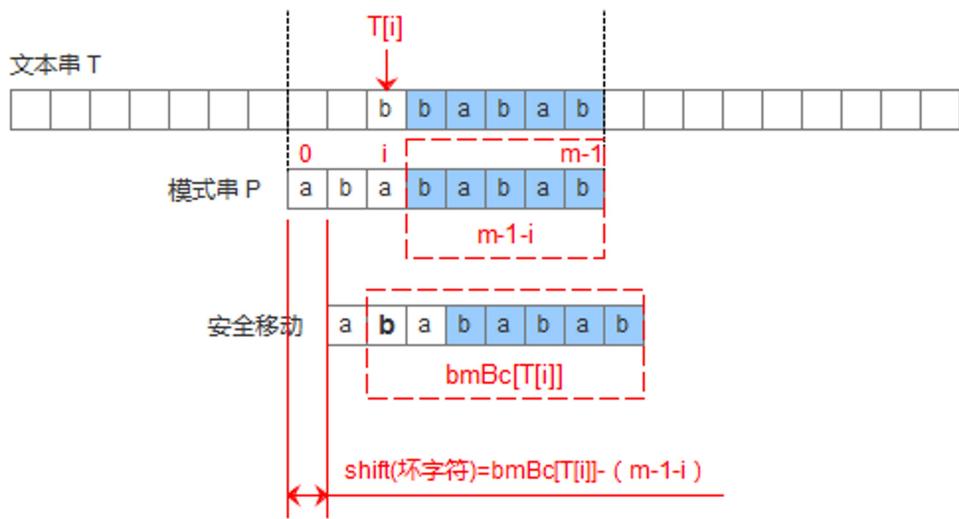


坏字符出现在模式串中，这时可以把模式串第一个出现的坏字符和母串的坏字符对齐，当然，这样可能造成模式串倒退移动



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 $\text{bmBc}['k']$ ，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： $\text{shift}(\text{坏字符}) = \text{bmBc}[T[i]] - (m-1-i)$ 。



```
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

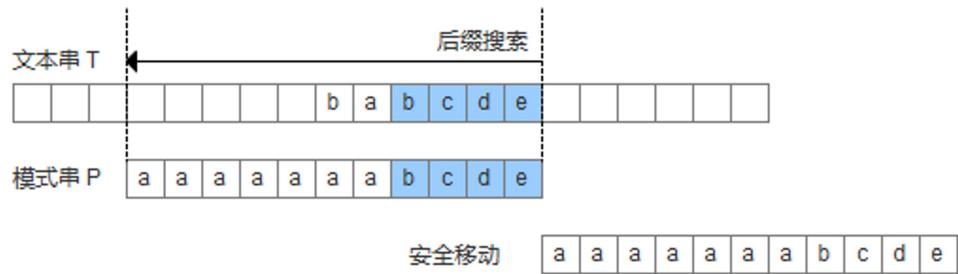
模式串中有子串匹配上好后缀，此时移动模式串，让该子串和好后缀对齐即可，如果超过一个子串匹配上好后缀，则选择最靠左边的子串对齐。



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);  
    }  
}
```

---

<https://www.cnblogs.com/Philip-Tell-Truth/p/5185267.html>

## BM算法

BM算法利用了串的后缀信息，用了2个类似KMP的Next数组的表来存储模式串的信息，一个是坏字符表，一个是好后缀表。

### 坏字符表有2种情况

1. 目标串中出现了模式串中没有的字符，此时模式串直接整体对齐到这个字符的后方，继续比较。  
。。BM是从后往前对比的，所以出现了一个不匹配的位置，并且target在这个位置上的char并没有出现在pattern中，那么就将pattern的第一个char放到不匹配位置的后一个char上，然后开始对pattern从后往前对比。
2. 目标串中出现了不匹配的字符A，并且不和其他子串形成后缀（这种情况也可以理解为好后缀表的len == 1的特殊情况），则把模式串中的最右的A和目标串的A对齐。  
。。到底是模式串最右，还是模式串当前不匹配位置左侧的最右？应该是后者啊。  
。。主要是形成后缀。。到底是什么？  
。。根据代码来看，是前者，就是模式串最右的出现。当然还有好后缀的帮助，不

过感觉直接设置当前位置左侧的最右，能跳过的更多啊。这种会导致不正确吗？。。。不会，而且肯定能跳过更多，因为如果模式串最右出现在当前不匹配的右侧，那么会导致 pattern 回退，这肯定没有意义的，所以这种应该靠好后缀来规避。那么就只能，，，不，想简单了，想得是构建当前不匹配位置的左侧的最右，只需要遍历的时候前缀就可以了，但是不是，因为有很多字符，所以在当前不匹配位置上把所有 target 中出现的字符就需要计算出在左侧的最右。不可能的，特别是中文字符太多了。也不是，可以靠 map，毕竟出现在左侧的是可以靠遍历一次的，target 中再多，只要不出现在 pattern 中，就没有意义，都是跳过整个 pattern 长度。如果 target 中多，就。。。不过 map 的 size 可以确保  $\leq$  pattern 的 size。不过需要 pattern.size() 个 map，除非有一些能保存历史记录的 map 结构（但即使有也是应该是靠时间换空间吧，而且意义不大，map 的话本身就不保存冗余数据的，不像 arr[][]，三角矩阵就浪费一半）。但无论如何，意义不大，1 是最坏情况  $\text{size} \times \text{size}$  的空间，2 是 map 终究不如 array。

```
typedef int Postion;
typedef char * _String;
void BuildBads(_String pattern, const int p_len, int *const BadS_List)
{
    fill(BadS_List, BadS_List + 256, p_len);
    for (int i = 0; i < p_len; i++)
        BadS_List[pattern[i]] = p_len - 1 - i;
}
```

。这个和之前的 preBmBc。都是全部初始化为 pattern 的 size。然后遍历 pattern，而且由于是从 0 到 size 的遍历，所以最后数组中保存的是最后一次出现的结果，所以是最右侧的出现。

## 好后缀表

1. 如果模式串中存在已经匹配成功的后缀，则把目标串和后缀对齐，然后从模式串的尾部开始往前匹配。
2. 如果无法找到匹配好的后缀，那么我们就找一个匹配的最长的前缀，让目标串与最长的前缀对齐（如果这个前缀存在的话）
3. 如果无法找到前缀，并且也没有好后缀，则直接移动模式串的 size 的长度。

构建好后缀的思路，有2种

第一个是  $O(n^3)$  的方法

1. 先定义一个 pre[i] 数组，含义和 KMP 的最长公共长度表差不多，不同在于：pre[i] 存储的是从  $pattern[k, k+sz-1-i]$  等于  $pattern[i, sz-1]$  的且  $pattern[k] \neq pattern[i]$  的最大的 k 值（相当于后缀的匹配），在构建 pre 表的时候我们顺便把最大的前缀长度记录起来，我们可以把 pre 全部初始化为 sz，然后一个个枚举就可以了，然后我们构建 good\_list，根据 good\_list 的性质，我们可以得到：
  - a. 如果  $pre[i] \neq sz$ ，则说明是第一种情况，我们直接把目标串前移  $sz-1-pre[i]$  个单位
  - b. 如果  $pre[i] = sz$ ，这里分为2种情况：
    - i. 如果前缀存在，则我们把目标串与前缀对齐（但前提是匹配的个数已经比前缀的长度大），则移动  $sz-1-i-c$  个单位（c 就是前缀长度）；
    - ii. 如果不存在前缀，或匹配的长度不够前缀的长度，就直接移动  $sz-1-i$

就可以了。

于是就有了下面这个很难懂的代码，注意这里把 pre[i] 和 good\_list 写一起了，因为 pre 只用一次。

时间复杂度 $O(n^3)$  (2个for 加一个 memcpy)

注意 这个算法 Good\_List 最后一位 固定是1

```
void BuildGoods_Slow(_String pattern, const int p_len, int *const Goods_List)
{
    //以 $O(n^3)$ 的时间构建好后缀表
    int max_suffix_length = 0;

    fill(Goods_List, Goods_List + p_len, p_len);
    Goods_List[p_len - 1] = 1; //最后一位固定是1

    for (Postion i = p_len - 1; i > 0; i--)
    {
        if (Inffix_Suffix_Compare(pattern, pattern + i, p_len - i))
            max_suffix_length = p_len - i; //记录最长的后缀
        for (Postion j = 1; j < i; j++)
        {
            if (Inffix_Suffix_Compare(pattern + j, pattern + i, p_len - i)
                && pattern[i - 1] != pattern[j - 1]) //一定要是不等于的时候才记录，最长后缀的最大k值
                Goods_List[i - 1] = j - 1;
        }
    }

    for (Postion i = 0; i < p_len; i++)
    {
        if (Goods_List[i] != p_len)
            Goods_List[i] = p_len - (Goods_List[i] + 1); //下标是从0开始的，而且要对齐后缀
        else //Goods_List[i]==p_len
        {
            Goods_List[i] += p_len - (1 + i); //下标是从0开始的
            if (max_suffix_length != 0 && p_len - 1 - i >= max_suffix_length)
                Goods_List[i] -= max_suffix_length;
        }
    }
}

bool Inffix_Suffix_Compare(_String sx, _String sy, const int len)
{
    for (int i = 0; i < len; i++)
        if (sx[i] != sy[i])
            return false;
    return true;
}
```

第二个是很巧妙的算法，算法时间复杂度 $O(n^2)$

1. 我们首先定义一个 suff 数组，这个数组和 pre 的定义是一样的，但是说法可能不太一样，suff[i] 表示以 i 为边界，与模式串后缀匹配的最大长度。

有了 suff 数组，我们直接定义 Good\_List 数组，实现方式与第一种算法的类似：

1. 模式串中有子串匹配上好后缀
2. 模式串中没有子串匹配上好后缀，但找到一个最大前缀
3. 模式串没有子串匹配上好后缀，但找不到一个最大前缀

```
void BuildGoods_Fast(_String pattern, const int p_len, int *const Goods_List)
{
    int *suff = new int[p_len];
    //构建suff表.........................
    suff[p_len - 1] = p_len;
    for (Postion i = p_len - 2; i >= 0; i--)
    {
        Postion k = i;
        while (k >= 0 && pattern[k] == pattern[p_len - 1 - (i - k)])
            k--;
        suff[i] = i - k;//设定最长后缀的位置
    }
}
```

。。。p\_len - 1 - (i - k) 自己代入下，第一次的时候 k=i，就是最后一个元素，后来每次 k--，所以每次前移一位，所以这个while是从后往前找最大后缀。一边是从最后一个元素，一边是从 i 开始，都是往前走。

```
//.........................
fill(Goods_List, Goods_List + p_len, p_len);
for (Postion i = p_len - 1, j = 0; i >= 0; i--)
    if (suff[i] == i + 1)
        for (; j < p_len - (i + 1); j++)
            if (Goods_List[j] == p_len)
                Goods_List[j] = p_len - (j + 1);
//以上代码是符合Good_List的第三和第二规则，复杂度是O(n^2)
//不用排最后一个了，因为最后一个肯定是直接移动p_len的
for (Postion i = 0; i < p_len - 1; i++)
    Goods_List[p_len - 1 - suff[i]] = p_len - (1 + i);
//Goods_List的值一直更新最小的，移动更小的位置达到更好的匹配效果
delete suff;
}
```

首先我们来研究怎么创建怎么构建 suff 数组，根据 suff 数组的定义，其实根据定义我们就知道，我们只用从后往前枚举  $pattern[i-k+1, i] == pattern[sz-1-k+1, sz-1]$ ，从而  $suff[i] == k$ （也就是得到了 i 位置的最长后缀长度），这里的 时间复杂度是  $O(n^2)$ ，这就是代码 5-12 行做的事情。

接下来我们要设定 good\_list，我们知道，如果我们要保证不漏掉任何一个匹配的可

能，那么首先目标串的移动要尽可能地少(其实坏字符表的构建也是一样的)，我们每次的操作都要保证移动 good\_list 的值一定要是所有的可能最小的。而我们再来看好后缀表移动的距离，我们发现移动的距离是情况1  $\leq$  情况2  $\leq$  情况3，而情况3相当于k=0的特殊情况，所以现在我们先来设定情况2和情况3。

代码14行，也就是fill那一段，就是对情况3的实现，一旦情况3发生，我们直接让目标串移动sz的距离重新匹配。时间复杂度O(n)。

代码15-19行，这是一段时间复杂度O(n)的代码（虽然有2个for，但是good\_list的每个位置最多只会改变一次），而suff[i] == i+1也就证明pattern[0, i]==pattern[sz-1-i, sz-1]，也就是情况2的前缀情况，所以我们直接让这个移动sz-1-j个位置对其前缀。

最后代码20-23，也就是对情况1的实现而suff[i]对于不同的i有可能是一样的，但是我们想让good\_list最小，所以我们就把i从小往大枚举，最后的good\_list就是最小的。这里的时间复杂度是O(n)。

最后我们来实现BM的主算法，其实BM主算法和KMP的主算法的思想是差不多的，只是BM算法利用了坏字符表和好后缀表进行跳转。当某个字符匹配成功的时候，我们把目标串和模式串都往前移动一个位置继续匹配（因为匹配是从后往前匹配的），如果匹配失败，我们就把目标串失配的那个位置向后移动这个失配字符在坏字符表和好后缀表中的最大的那个值。（为什么这里又不是最小了呢？因为我们坏字符表和好后缀表的位置都是已经保证了移动正确性为前提的了，现在我们想要算法更加的高效，那么目标串的移动肯定要尽量大一点的了。）

```
bool BmSearch(_String target, _String pattern)
{
    int t_len = strlen(target), p_len = strlen(pattern);
    int *BadS_List = new int[256];
    int *Goods_List = new int[p_len];

    BuildBads(pattern, p_len, BadS_List);
    //BuildGoods_Slow(pattern, p_len, Goods_List);
    BuildGoods_Fast(pattern, p_len, Goods_List);

    Postion i = p_len - 1, j = p_len - 1;

    while (j < p_len)
    {
        while(j > 0 && target[i] == pattern[j])
        {
            i--;
            j--;
        }
        if (j == 0 && target[i] == pattern[j])
        {
            delete BadS_List, Goods_List; //找到一个就可以了
            return true;
        }
    }
}
```

```
i += Goods_List[j] > BadS_List[target[i]] ? Goods_List[j] :  
BadS_List[target[i]];  
j = p_len - 1;  
}  
delete BadS_List, Goods_List;  
return false;  
}
```

=====

// 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，比如输入的字符串为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/>

```
s[0] = '$'; s[++m] = '#';
for (b = 1; ss[b] != '\0'; ++b) {
    s[++m] = ss[b];
    s[++m] = '#';
}
s[++m] = '?';
for (int i = 1; i < m; ++i) {
    if (maxid > i) p[i] = min(maxid-i, p[2*id-i]);
    else p[i] = 1;
    while (s[i-p[i]] == s[i+p[i]]) p[i]++;
    if (i + p[i] > maxid) {
        maxid = i + p[i];
        id = i;
    }
}
```

。 。 。

最远可达子串的右侧是 maxid。这个子串的 中心是 id。这个id 必然是 遍历过的 i，所以 必然 小于 当前遍历的 i。

如果 maxid 大于 当前遍历的 i。那么 可以 按照 id 进行 (轴)对称，可以 确定 i 关于id 的对称点 i2，然后 复用 i2的 最大回文长度，因为 id 是回文，所以 i 和 i2

附近的 substr 是 倒序相同的。。 i 对于 id 的 对称点 是  $id - (i-id) = 2*id - i$ ，  
所以 p[2\*id-1] 是一个选项。

另外一个选项是 maxid-i， 这个是为了防止 i2 很长，假设 maxid-1 是 i2 的最后一个字符，那么此时  $p[i2] == p[2*id-i] == maxid-1-i2 = maxid-1-2*id+i$

那么 i 的回文的最右侧就是， $i + maxid - 1 - 2*id + i$ ，由于 i 大于 id，这个长度是大于 maxid 的，而大于 maxid 的字符还没有被使用过，所以无法确保是回文的一部分。 所以需要限制最大为 maxid - i

。 。 。

=====

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

```
// Prim, Kruskal
```

=====

Moore Voting Algorithm

=====

---

## 欧拉图

具有欧拉回路的图 被称为欧拉图

具有欧拉路径 但不具有 欧拉回路的图 被称为 半欧拉图

半欧拉图 充要条件： 连通图，且仅有2个 奇度点。

---

## 欧拉回路

通过图(无向或有向图) 中所有边 且 每条边仅通过一次的通路，相应的回路被 称为欧拉回路。

如果图G 中一个路径 包括每个边 恰好一次，则该路径被称为 欧拉路径 Euler path

如果一个 回路 是欧拉路径，则称为欧拉回路 Euler circuit

。。回路就是 起点==终点的路径

---

### 无向图存在欧拉回路的充要条件

一个无向图存在欧拉回路，当且仅当该图 所有 顶点度数 都为偶数，且该图是连通图。

---

### 有向图存在欧拉回路的充要条件

所有顶点的 入度等于出度，且是连通图。

---

## 求欧拉回路的思路

循环的找到出发点。从某个节点开始，然后查出一个 从这个节点出发 最后回到这个节点的环。如果环中 某个节点 还有边 没有被遍历，则以这个节点为起点，未遍历的边为方向，找 从这个节点出发 最终回到这个节点的 环。直到所有边都被遍历。

。。首先，不能有 不成环的边，这种边 是无法 构成回路的，最多只能构成 欧拉路径。

。。就是先找一个环，然后 环上 存在 某些点，它们有边没有被遍历，那么 选一个，然后 以这个 点为开始，再找环，而且要边的vst(或者st\*100000+en)，不然可能和 找到的第一个环 有部分重叠，那么就 不是欧拉路径了。

。。就是找到一个环，中间有节点 还存在没有被遍历的边，那么就 走一个半环 走到那个节点，然后 走那个节点的环，然后 再走剩下的 半环。

。。就是下面的算法。

。。

。。欧拉回路 所有的边 必然在环中，所以 如果 找不到环，那就可以直接退出了。

。。还要判断下是否 全部边都被遍历了。可能是一个 非连通图。

## 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。

---

图的环游(tour)是指经过图的每条边至少一次的闭途位。欧拉环游是经过每条边恰好一次的环游。一个图若包含欧拉环游，则称为欧拉图(Eulerian graph)。

弗勒里(B. H. Fleury) 在1883 年给出了在欧拉图中找出一个欧拉环游的多项式时间算法，称为弗勒里算法

输入：一个连通偶图 G 和 G 中任意一个指定项点 u

输出：从 u 出发的 G 的一个欧拉环游

1、令 W: =u, x: =u, F: =G

2、while F不为空集

    3、选一条 F 中的边 e，其中 e 不是 F 的一条割边；如果 F 中的边都是割边，那么任选一条边 e

    4、用 uWxey 替换 uWx，用 y 替换 x，用 F\{e\} 替换 F

    5、end while

    6、返回 W

。。。都是数学符号。。写不了。

其算法核心就是沿着一条迹往下寻找，先选择非割边，除非这个点的邻边都是割边。这样得到一条新的迹，然后再继续往下寻找，直到把所有边找完。遵循这样一个原则就可以找出图的一个欧拉环游来。

在有向图中也可以类似地定义有向环游、有向欧拉环游、有向欧拉图和有向欧拉迹的概念。

割边：连通图G，e是其中一条边，G-e 形成的图不是连通图，则 e 是 G 的一条割边。

割点：

```
=====

// 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  $a^b \bmod 1337$  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  $a \equiv p \pmod{1337}$ . 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](https://en.wikipedia.org/wiki/Euler%27s_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  $i \leq m$  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  $d \leq m$ ，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  $i \leq m$  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  $i \leq m$  时，第  $i$  个接收到的数据最后留在蓄水池中的概率 =  $m/N = m/N$ 。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  $N \leq M$ ，直接插入

如果  $N > M$ ，就取一个  $1 \sim N$  之间的随机数  $index$ 。如果  $index$  在  $1 \sim M$  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  $N_i$ 。

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  $N = \text{Sum}(N_i \mid i=1 \dots K)$

若  $d < N_1/N$  则从第一个机器上等概率抽取一个元素。

若  $N_1/N \leq d < (N_1+N_2)/N$  则从第二个机器上等概率抽取一个元素

依此类推。

---

=====

Tarjan 算法 – Robert Tarjan

线性时间求解有向图强连通分量

如果 2 个顶点可以相互到达，则称 2 个顶点 强连通（strongly connected）。如果有向图  $G$  的每 2 个顶点都强连通，称  $G$  是一个 强连通图。

有向图的极大强连通子图，称为 强连通分量

Tarjan算法是用于求 有向图 的 强连通分量的。

Robert Tarjan 还发明了 求 双连通分量 的 Tarjan 算法。

时间复杂度 $O(N+M)$

tarjan算法是基于对图的 深度优先搜索的 算法，每个强连通分量是 搜索树中的一颗子树。搜索时，把当前搜索树中未处理的 节点 加入一个栈，回溯时 可以判断 栈顶 到栈中的节点 是否为 一个强连通分量。

定义 $DFN(u)$  为节点 $u$  搜索的 次序编号 (时间戳)， $Low(u)$  是  $u$  或 $u$ 的子树能追溯到的最早的 栈中节点的次序号。

当  $DFN(u) == Low(u)$  时，以 $u$ 为根的 搜索子树上所有节点是一个 强连通分量。

1. 当首次搜索到点 $u$  时， $DFN[u] = Low[u] = time$ ;
2. 每当搜索到一个点，把该点压入栈顶
3. 当 $u$  和  $v$ 有边相连时：
  - a. 如果 $v$  不在栈中 (树枝旁)， $dfs(v)$ ，然后  $Low[u] = \min\{Low[u], Low[v]\}$ ;
  - b. 如果 $v$  在栈中 (前向边 / 后向边)，此时  $Low[u] = \min\{Low[u], DFN[v]\}$ 
    - i. 当 $DFN[u] = Low[u]$ 时，将它及它之上的元素弹出栈，这些弹出栈的节点 构成一个 强连通分量
    - ii. 继续搜索，直到图被遍历完。

```
tarjan(u){  
    DFN[u]=LOW[u]=++Index           // (1) 为节点u设定次序编号和Low初值  
    Stack.push(u)                   // (2) 将节点u压入栈中  
    for each (u, v) in E            // (3) 枚举每一条边  
        if (v is not visted)         // (4) 如果节点v未被访问过  
            tarjan(v)               // (5) 继续向下找  
            LOW[u] = min(LOW[u], LOW[v]) // (6)  
        else if (v in Stack)         // (7) 如果节点v还在栈内  
            LOW[u] = min(LOW[u], DFN[v]) // (8)  
  
        if (DFN[u] == LOW[u])          // (9) 如果节点u是强连通分量的根  
            repeat                    // (10) 循环出栈,直到u=v  
                v = Stack.pop          // (11) 将v退栈, 为该强连通分量中一个顶点  
                print v               // (12) 输出v  
            until (u == v)           // (13) 循环终止条件u=v  
}
```

=====

Kosaraju算法

基于对 有向图及其逆有向图2次dfs，时间复杂度也是 $O(M+N)$ 。

与Trajan相比，Kosaraju可能更直观一些。但是Tarjan只对原图进行一次dfs，不需要建立逆图，更简洁。

在实际的测试中，Tarjan的效率 比Kosaraju 高 30% 左右。

Kosaraju的解释和实现都比较简单，为了找到强连通分量，首先对图G进行dfs，计算出各

顶点完成搜索的时间f；然后计算图的逆图GT，对逆图也进行dfs，但是这里搜索时顶点的访问次序不是按照顶点标号的大小，而是按照各顶点的 f 值 从大到小的次序；逆图dfs所得的森林就是 对应的 强连通区域。

对原图G进行深度优先遍历，记录每个节点的离开时间num[i]

选择具有最晚离开时间的顶点，对反图GT进行遍历，删除能够遍历到的顶点，这些顶点构成一个强连通分量

如果还有顶点没有删除，继续步骤2，否则算法结束

=====

Tarjan算法 – Robert Tarjan

LCA

对于有根树T 的2个节点u, v, 最近公共祖先LCA(T,u,v) 表示一个节点x, 满足 x是u和v的祖先 且x的深度尽可能大(。。。深度应该浅吧)。在这里，一个节点也可以是它自己的祖先。

另一种理解方式是把 T 理解为一个 无向无环图，而LCA(T,u,v) 即 u到v的最短路径上深度最小的点。

。。。这个路径不能重复的。

利用并查集 优越的 时空复杂度，我们可以实现LCA问题的  $O(n+Q)$  算法，Q表示查询次数。

Tarjan算法基于dfs，对于新搜索到的 节点，首先创建由 这个节点构成的集合，再对当前节点的每一个子树进行搜索，每搜索完一颗子树，则可以确定子树内的LCA询问都已解决。其他的LCA询问的结果必然在这个子树之外，这时把子树所形成的集合与当前节点的集合合并，并将当前节点设置为这个集合的祖先。

之后继续搜索下一棵子树，直到当前节点的所有子树都搜索完。这时把当前节点也设置为已检查，同时可以处理有关当前节点的LCA查询，如果有 一个 当前节点 到 节点v 的询问，且v已经被检查过，则由于进行的是 dfs，当前节点 与 v的 LCA 一定还没有被检查，而这个 LCA 的包含 v 的子树一定已经搜索过了，那么这个 LCA一定是 v 所在集合的 祖先。

=====

=====

## Dijkstra

一个顶点到其它各个顶点的最短路径。

解决有权图中最短路径。 不能存在 负权边

从起始点开始，采用 贪心策略，每次 遍历 未访问的节点中 距离起始点最近的 节点。

具体过程：

1. 需要一个辅助数组D，它的每个元素表示 起始点 到 其它节点的最短长度。
2. 辅助数组D的初始化：
  - a. 如果从 节点 $v_i$  到 起始点 有直接边 相连，则  $D[i]$  的值就是 边的权值。 没有直接边相连，则  $D[i]$  值是 无穷大。
3. 辅助数组D中最小值，就是 从起始点出发 到其它某个节点 的最短路径。
4. 找到 某个其它节点后，更新 与 这个节点 直接相连的 节点的 最小长度。
5. 再取 (除了起始点和已找到的节点外的) 最小值
6. 然后 再 根据取到的节点 刷新 辅助数组D 中直接相连的 节点的 最小长度。

=====

## Prim

1). 输入：一个加权连通图，其中顶点集合为V，边集合为E；

2). 初始化：  $V_{new} = \{x\}$ ， 其中x为集合V中的任一节点（起始点），  $E_{new} = \{\}$ , 为空；

3). 重复下列操作，直到 $V_{new} = V$ ：

- a. 在集合E中选取权值最小的边 $\langle u, v \rangle$ ，其中u为集合 $V_{new}$ 中的元素，而v不在 $V_{new}$ 集合当中，并且 $v \in V$ （如果存在有多条满足前述条件即具有相同权值的边，则可任意选取其中之一）；
  - b. 将v加入集合 $V_{new}$ 中，将 $\langle u, v \rangle$ 边加入集合 $E_{new}$ 中；
- 4). 输出：使用集合 $V_{new}$ 和 $E_{new}$ 来描述所得到的最小生成树。

这里不需要更新

=====

## Kruskal

时间复杂度为 $O(e \log e)$  ( $e$ 为网中的边数)，所以，适合于求边稀疏的网的最小生成树

假设连通网 $G = (V, E)$ ，令最小生成树的初始状态为只有 $n$ 个顶点而无边的非连通图

$T = (V, \{\})$ ，概述图中每个顶点自成一个连通分量。在E中选择代价最小的边，若该边依附的顶点分别在T中不同的连通分量上，则将此边加入到T中；否则，舍去此边而选择下一条代价最小的边。依此类推，直至T中所有顶点构成一个连通分量为止。

=====

SPFA

//LT1631

//Using BFS is actually an improvement of Bellman – Ford, called Shortest Path Faster Algorithm,

[https://en.wikipedia.org/wiki/Shortest\\_Path\\_Faster\\_Algorithm](https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm)

//SPFA removes the unnecessary edge relaxations, thus reducing the time complexity, but worst case still the same as Bellman – Ford.

=====

Floyd

=====

Bellman–Ford

=====

Suffix Array

对字符串所有后缀进行排序后得到的数组。

用于全文索引，数据压缩算法，生物信息学。

后缀数组， $SA[i]$ 存放排名第*i*大的后缀首字符下标

一维数组，保证  $\text{suffix}(SA[i]) < \text{suffix}(SA[i+1])$

也就是将 字符串S 的n个后缀 从小到大排序之后，把排好序的后缀的 开头位置依次放入SA中。

名次数组，`rank[i]` 存放 `suffix(i)` 的优先级  
保存的是 `suffix(i)` 在所有后缀中 从小到大排列的 名次

。 。 SA是排序后，保存 排序前 这个substring的起始下标。。。就是 `sz1 - substr.size()`。  
。 。 rank保存的是 排序前 这个substring，在排序后的 数组中的 次序。

SA：排第几的是谁。

rank：你排第几。

SA和rank是互逆运算。只要计算出SA，就可以在O(n)中算出 rank。

height数组：保存的是 `suffix(i)` 和 `suffix(i - 1)` 的最长公共前缀的长度。也就是排名相邻的2个后缀的最长公共前缀。

。 。 这里的排名相邻是指SA相邻？主要是前面说 `suffix(i)`，感觉是排序前的 相邻 substr的 最长公共前缀。不不不，相邻的substr的 公共前缀是 非常 稀少的，没有太大意义的。

构建suffix array，主要就是要构建 SA， rank， height数组。

如何构建SA数组：

1. 倍增算法：O(nlogn)
2. DC3算法：O(n)
3. skew算法：不常用

这里介绍DC3算法：

1. 将后缀分成2步走，然后对第一部分的后缀排序。  
字符的编号从0开始  
将后缀分成 2部分：  
    第一部分是后缀k ( $k \% 3 \neq 0$ )  
    第二部分是后缀k ( $k \% 3 == 0$ )
2. 利用 步骤1 的结果，对第二部分的后缀排序
3. 将 步骤1 和 2 的结果合并，就完成了 对所有后缀的排序。

求出了所有后缀的排序，有什么用呢？主要是用于求它们之间的最长公共前缀(LCP, longed common prefix)

求出SA数组后，根据 `rank[sa[i]] = i`， rank数组可以在 O(n) 中求出。

如何求出height数组呢？

令  $LCP(i, j)$  为 第*i*小的后缀 和 第*j*小的后缀 (即 `suffix(SA[i])` 和 `suffix(SA[j])` ) 的最长公共前缀的长度，则有如下2个性质：

- a. 对于任意  $i \leq k \leq j$ ，有  $LCP(i, j) = \min(LCP(i, k), LCP(k, j))$
- b.  $LCP(i, j) = \min(i < k \leq j)(LCP(k-1, k))$

令`height[i] = LCP(i-1, i)`，即`height[i]`代表第*i*小的后缀 与 第*i-1*小的后缀的 LCP，则求  $LCP(i, j)$  就变成了 求 `height[i+1] height[i]` 之间的 RMQ，使用RMQ 算法就可以了，复杂度是 预处理  $O(nlogn)$ ，查询 $O(1)$ 。

。 。 ?

。 。 。 代码很难的。

---

## 倍增算法

1. 根据字典顺序 对每个后缀的 第一个字符 排序，得到了第一次的排序
  2. 对每个后缀的 前2个字符 进行排序，得到了 第二次的排序
  3. 对每个后缀的 前4个字符进行排序（这就是倍增）。 但是我们这次排序的仍然时2个字符。后缀k的前4个字符 看成 由 后缀k的 前2个字符 和 后缀k+2 的前2个字符 组成。所以如果要把 后缀m 和 后缀n 进行比较，那么应该首先 比较 后缀m 的前2个字符 和 后缀 n 的前2个字符，如果相同，再比较 后缀m+2 的 前2个字符 和 后缀n+2 的前2个字符。 所以 一直排序的是 二元组，这就是为什么要 使用 基数排序。
  4. 如果所有的排名都不同的话，就不需要再倍增了
- 。 。 感觉像桶排序。先根据第一个char进行排序，然后根据第二个char。  
。 。 不过这里，可以有dp。 桶排序没有。
- 。 。 代码是 一个for 中 7个for。 。 。

---

## LCP array

---

## RMQ 算法

range min/max query，区间最值查询

解决多个区间最值查询的算法。

----

假设有一个长度N的数组arr，求a-b之间的最大/最小值。

最简单的就是二维数组，预先计算好 每个ab对的 max/min。但是内存非常浪费。

RMQ算法就是巧妙地运用 2的幂数/对数 解决这个问题的。

设  $dp[i][j]$  是以  $i$  为起点， $(1 \ll j)$  长度的最大值。

这样，需要求  $a-b$  的最大值时，只需要求  $\max(dp[a][\text{某个长度}], dp[b-(1 \ll \text{某个长度})+1][\text{某个长度}])$ ，其中 某个长度 =  $\log_2(b-a+1)$

```
void query(int a, int b, int &maxV) {
    int r = mlog2(b - a + 1);
    maxV = max( dpMax[a][r], dpMax[b - (1<<r) + 1][r] );
}
```

---

```
using namespace std;
```

```
#define MAXN 50005
```

```
int h[MAXN];
int dpMin[MAXN][16];
int dpMax[MAXN][16];
```

```
int mlog2(int x) {
    double xx = x;
    double v1 = log(xx);
    double v2 = log(2.0);
    return v1/v2;
}
```

```
void genDP(int len){
    for(int i=0;i<len;i++){
        dpMax[i][0] = h[i];
        dpMin[i][0] = h[i];
    }
    for(int j=1;(1<<j)<len;j++){
        for( int i=0;i+(1<<j)-1<len;i++ ){
            int r = i+(1<<(j-1));
            dpMax[i][j] = max(dpMax[i][j-1],dpMax[r][j-1]);
            dpMin[i][j] = min(dpMin[i][j-1],dpMin[r][j-1]);
        }
    }
}
```

```
void query(int a,int b,int &maxV,int &minV){
    int r = mlog2(b-a+1);
    maxV = max(dpMax[a][r],dpMax[b-(1<<r)+1][r]);
```

```

minV = min(dpMin[a][r],dpMin[b-(1<<r)+1][r]);
}

int main(){
    int N,Q;
    int t;
    scanf("%d%d",&N,&Q);
    for( int i=0;i<N;i++ ){
        scanf("%d",&t);
        h[i] = t;
    }
    genDP(N);
    int a,b;
    int maxV,minV;
    for( int i=0;i<Q;i++ ){
        scanf("%d%d",&a,&b);
        if ( a==b ){
            printf("0\n");
        }else{
            query(a-1,b-1,maxV,minV);
            printf("%d\n",maxV-minV);
        }
    }
    return 0;
}

```

=====

=====

=====

Travelling Salesman Problem

哈密顿回路(Hamiltonian cycle) 哈密顿图(Hamiltonian Path) 旅行推销员问题

(Travelling salesman problem)

=====

Rolling Hash

=====

=====

线段树

<https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/>

。。。这个是 sum of given range，还有一个是 range min query。都差不多，就是 merge 的行为的不同

考虑如下问题 来理解 segment tree

有一个数组 arr[0 … n-1]。我们需要做到：

1. 计算 [l, r] 区间内的 元素 和
2. 修改某个元素的值

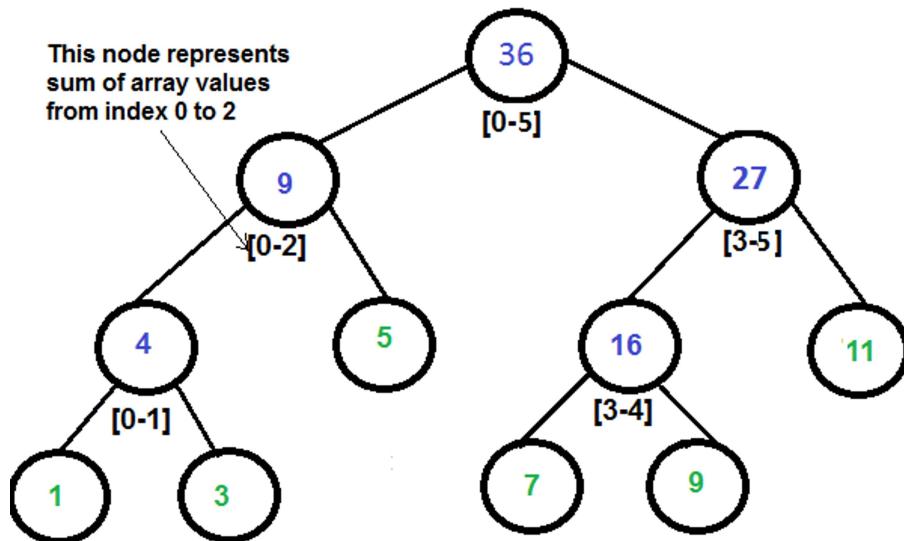
方案

1. 遍历[l, r] 来累加 sum; 这样，操作1是 O(n)，操作2是 O(1)
2. 前缀和数组，保存从下标0 到当前下标的 sum。这样，操作1是 O(1)，操作2是 O(n)。
3. 最有效的方法是 使用 Segment Tree。2个操作都是 O(logn)

Segment tree 的描述

1. 叶子节点 是 输入的arr 的元素
2. 每个 内部节点 表示 叶子节点的 merge。**这个 merge 对于不同的问题 是不同的。**  
对于目前的问题，这个 merge 是 该内部节点 下属的 所有 叶子节点 的 sum。
3. 树 的 数组形式 来 表示 Segment tree。对于 下标 i的节点，左child 是在 下标

$i \times 2 + 1$ , 右child 是  $i \times 2 + 2$ , parent 是  $(\text{floor}((i-1) / 2))$



Segment Tree for input array {1, 3, 5, 7, 9, 11}

。。不过这个没有办法转成数组。因为不是 完全二叉树。

从输入的数组 构造 Segment Tree

我们从一个 segment: arr[0, n-1] 开始。每次 我们 divide 当前的 segment 成 2个 (只要 segment 还没有变成长度1), 然后 对 2个新的 segment 再次使用相同的计算, 对于每个这样的 segment, 我们在对应的node 上保存 sum。

构造出的 segment tree 的 除了最后一层外, 每层 都是满的。这样, tree 会是一个 full 二叉树, 因为 我们divide segment 成2个。由于 构造的tree 是一个 含有n 个叶子 的 full 二叉树, 所以 会有 n-1 个 内部节点, 所以一共有  $2*n - 1$  个节点。

。。国内外对 full binary tree (满二叉树) 的定义不同, 国内是 正好一个等腰三角形。国外是: 每个节点 要么是叶子节点, 要么 有2个child。

输入的arr 的 segment tree 的 高度是多少

线段树的高度是  $\text{ceil}(\log N)$ 。由于 树 是通过arr 来表达的, 且 parent 和 child node 之间的 关系 是固定的, 所以 申请的 空间是  $(2 * 2^{\text{ceil}(\log N)}) - 1$ 。  
。。也就是  $2^{\text{ceil}(\log N)} + 1 - 1$

查询 给定range 的 sum

```
int getSum(node, l, r)
{
    if the range of the node is within l and r
        return value in the node
    else if the range of the node is completely outside l and r
        return 0
    else
        return getSum(node's left child, l, r) +
               getSum(node's right child, l, r)
```

```
}
```

上面的实现中，有3个分支

1. 如果 当前node 的range 被 [l, r] 完全包围，那么 直接 把 节点的值 加到 ans
2. 如果 range 和 [l, r] 没有重叠，那么 ans + 0
3. 如果部分重叠， 递归 left 和 right child

更新值

和 tree的构造，查询 一样，update 也是 递归的。 我们收到一个 需要更新的 index，需要加上 diff 值。

我们从 root 开始，增加 diff 值 到 任何 range 包含 index 的 node。

。。。更下面有个位运算实现的，更短。不过 下标是从 1 开始的。

。。这里是 从0 开始， 所以  $i*2+1, i*2+2$   
。。从1开始的话，  $i*2, i*2+1$

下面是代码实现

```
#include <bits/stdc++.h>
using namespace std;

int getMid(int s, int e) { return s + (e - s)/2; }

int *constructST(int arr[], int n)
{
    int x = (int)(ceil(log2(n)));
    int max_size = 2*(int)pow(2, x) - 1;
    int *st = new int[max_size];

    constructSTUtil(arr, 0, n-1, st, 0);

    return st;
}

int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) +
              constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}
```

```

int getSum(int *st, int n, int qs, int qe)
{
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        cout<<"Invalid Input";
        return -1;
    }
    return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// st : segment tree
// si : current node's index in array
// ss, se : 当前node代表的 range
// qs, qe : query range
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    if (qs <= ss && qe >= se)
        return st[si];

    if (se < qs || ss > qe)
        return 0;

    int mid = getMid(ss, se);
    return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
           getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    if (i < 0 || i > n-1)
    {
        cout<<"Invalid Input";
        return;
    }

    int diff = new_val - arr[i];

    arr[i] = new_val; // 这个是arr, 下面更新的是 st。 ok的。

    updateValueUtil(st, 0, n-1, i, diff, 0);
}

void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    if (i < ss || i > se)
        return;

```

```

st[si] = st[si] + diff;
if (se != ss)
{
    int mid = getMid(ss, se);
    updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
    updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
}
}

int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    int *st = constructST(arr, n);

    cout<<"Sum of values in given range = "<<getSum(st, n, 1, 3)<<endl;

    updateValue(arr, st, n, 1, 10);

    cout<<"Updated sum of values in given range = "
        <<getSum(st, n, 1, 3)<<endl;
    return 0;
}

```

---

<https://www.geeksforgeeks.org/segment-tree-efficient-implementation/>

考虑如下问题 来理解 segment tree， 不使用递归。

。。不使用递归的意思是，construct, update, query segment tree 的时候 不是 递归的。

有一个数组 arr[0 … n-1]。我们需要做到：

1. 计算 [l, r] 区间内的 元素 和
2. 修改某个元素的值

方案

1. 每次遍历[l, r] 区间来计算 sum, 这样的话，上面的第一个操作是 O(n)， 第二个操作是 O(1)。
2. 创建数组来保存 从 i开始的后缀数组的和。 第一个操作是 O(1)， 第二个操作是 O(n)。
3. 有没有一种 都是 O(logn) 的呢？ 我们可以使用 segment tree

考虑下面的数组和 segment tree

---

。。注意这里 下标是从 1 开始的。

1 : [0, 16)		3 : [8, 16)		7 : [12, 16)	
4 : [0, 4)	2 : [0, 8)	5 : [4, 8)	6 : [8, 12)	13 : [10, 12)	14 : [12, 14)
8 : [0, 2)	9 : [2, 4)	10 : [4, 6)	11 : [6, 8)	12 : [8, 10)	15 : [14, 16)

可以看到 原始数组 在底部，是一个下标从0开始的 16个元素 组成的arr。

树一共有31个node，叶子节点(保存了原始数组的元素) 从 node16 开始。

所以我们可以方便地 使用  $2 \times N$  大小的 数组 来构造 segment tree ( $N$  是 原始数组的长度)。

叶子节点从 数组的下标N 开始，到  $2 \times N - 1$  结束。因此，原数组 下标为 i 的元素，在 segment tree 中 下标是  $N + i$ 。

现在来计算 child(原文是 parent)， 我们从下标  $N - 1$  开始，向前移动。

对于 下标 i，left child 会在  $2 \times i$ ，right child 会在  $2 \times i + 1$ 。所以  $2 \times i$  和  $2 \times i + 1$  的 node 上的值 被组合到 node i 上。

在上图中，我们可以查询  $[l, r)$  的区间。

我们将使用 位运算 来实现这些 乘法和加法。

```
#include <bits/stdc++.h>
using namespace std;

const int N = 100000;
int n; // array size
int tree[2 * N];

void build( int arr[])
{
    // 添加原数组
    for (int i=0; i<n; i++)
        tree[n+i] = arr[i];

    // 通过计算child 来merge，注意下标从 1 开始。不然  $0 \ll 1$  还是0 。
    // 这里原文是 build the tree by calculating parents
    // 但是这里是 已知parent 为 i，计算child 啊。
    for (int i = n - 1; i > 0; --i)
        tree[i] = tree[i<<1] + tree[i<<1 | 1];
}

void updateTreeNode(int p, int value)
{
    // 先写 p += n 不好么。
    tree[p+n] = value;
    p = p+n;

    //  $i^1$  是 找到 和 i配对的那个，可能大于，可能小于。
    // 因为 parent 是  $i \gg 1$ ，所以 child 是  $(i \gg 1) \ll 1$  和  $((i \gg 1) \ll 1) + 1$ 
    // 这2个值 和 i,  $i^1$  是相同集合
    for (int i=p; i > 1; i >>= 1)
```

```

        tree[i>>1] = tree[i] + tree[i^1];
    }

// ... 宋体的11真的差不多。。00 ...
// l1 00 i1
// 11 00 i1
// 宋体字号11时，11就明显一点
// 而且不知道为什么，使用consolas, dejavu sans mono时，中文是
Microsoft YaHe, 英文是选定的字体。可能是字库不全，然后就使用默认的 YaHe
了。也不是，是 consolas, dejavu sans mono 不会修改中文的字体。。

// function to get sum on interval [l, r)
int query(int l, int r)
{
    int res = 0;

    // l 和 r 变成 segment tree 中的叶子节点的下标
    // 如果 &l == true, 说明 >>1 时会被舍弃，所以需要 +
    // 注意 先用 l, 然后 + 1。先-1, 然后用 r
    // 先-1, 然后r 是因为 [l, r)
    // 还有 root 下标是1, child 是 xxx0, xxx1 是成对出现的。
    // 所以当 l&1 时，说明左侧多了一个不成匹配的单个值 xxx1, 所以先+上,
    // 然后从range中 移除这个单个值
    // 当 r&1 时，说明右侧多了一个不成匹配的单个值 xxx0, 所以加上这个0。
    for (l += n, r += n; l < r; l >>= 1, r >>= 1)
    {
        if (l&1)
            res += tree[l++];

        if (r&1)
            res += tree[-r];
    }

    return res;
}

int main()
{
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    n = sizeof(a)/sizeof(a[0]);

    build(a);

    cout << query(1, 3)<<endl;

    updateTreeNode(2, 1);

    cout << query(1, 3)<<endl;

    return 0;
}

```

---

<https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/>

在第一个 sum of range 的 segment tree 中，update方法 用来更新 数组中的一个值，这个update 会导致 segment tree 中的 多次更新，因为 被更新的值 在 segment tree 的node 的range 中。

下面的简单的逻辑

1. 从 segment tree 的root 开始。
2. 如果 被更新的值 不在 node 的range 中，则return
3. 如果在 range中，则 更新node，并且 递归 children。

下面是 第一个 sum of range 中使用的 update 代码

```
void updateValueUtil(int tree[], int ss, int se, int i, int diff, int si)
{
    if (i < ss || i > se)
        return;

    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}
```

如果update 是对于 一段下标range 进行的，该怎么处理？

例如，对于 下标 2 到 7 的元素 全部 + 10。

上面的代码 会对 每个下标 调用 update操作。 我们可以避免 多次调用， 通过 编写一个 updateRange() 方法

。。是range内全部加一个固定值。

```
// si 是 segment tree 中的 当前节点
// ss, se, 是 当前node 存储的 原数组元素 下标的 起止
// us, ue, update range 的起止
// diff, update range中 每个元素的 add 的值。
void updateRangeUtil(int si, int ss, int se, int us, int ue, int diff)
{
    // out of range
    if (ss>se || ss>ue || se<us)
        return ;
```

```

// Current node is a leaf node
if (ss==se)
{
    // Add the difference to current node
    tree[si] += diff;
    return;
}

// If not a leaf node, recur for children.
int mid = (ss+se)/2;
updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

// Use the result of children calls to update this
// node
tree[si] = tree[si*2+1] + tree[si*2+2];
}

```

### Lazy Propagation - An optimization to make range updates faster

当有许多update range时，我们可以 推迟一些 update (避免 update的 递归调用)，只有在需要时 才执行 update

segment tree 中的 node 保存了 对index range 的 query 的结果。

如果 node 的 range 在 update操作的range 的内部，那么 这个 node 的所有 后代都需要被 update。

例如，一个node 保存了 下标3-5的sum，那么 如果有一个 update range 是2-5，那么这个node 和它所有后代node 都要被更新。

使用 lazy propagation，我们只更新 这个node，推迟 对这个node的后代node 的 update，通过 保存 update info 到 单独的 被称为 lazy node 的 节点。

我们创建一个 lazy[] 来表示 lazy node。 长度 和 segment tree 的 数组的长度一样。

初始化为 全0。lazy[i] == 0 意味着 node i 没有 待执行的 update。 非0 意味者 对这个节点的 任何查询之前 需要把 这个值 加到 节点 i上。

更新us 到 ue 的值

1. 如果当前 node 有待执行的 update，那么把 待执行的update 加到 当前node。
2. 如果当前node 的range 完全包含于 update range
  - a. 更新当前node
  - b. 推迟 对child 的update，通过设置 child node 的 lazy value。
3. 如果当前node 的range 和 update range 有 重叠
  - a. 递归 left, right child
  - b. 使用 left, right 返回的值，更新 当前 node 的值。

query方法 是否有变化？

由于 我们修改了 update 方法 来延迟，如果对 还没有update 的 node 进行query 会出

现问题。

所以我们需要修改 query 方法。现在的 getSumUtil 方法先检查是否有待执行的 update，如果有，那么 update node，在 update 执行完后，就和之前的 getSumUtil 一样。

```
#include <stdio.h>
#include <math.h>
#define MAX 1000

int tree[MAX] = {0}; // To store segment tree
int lazy[MAX] = {0}; // To store pending updates

// si: 当前node 在 segment tree 的array表达 中的index
// ss, se, 当前node 存储的 元素下标的 start 和 end
// us, ue, update range 的 start 和 end
// diff, 我们需要add 到 us 到 ue 的 值。
void updateRangeUtil(int si, int ss, int se, int us, int ue, int diff)
{
    // 如果当前node 的 lazy value 非0, 那么说明 存在 待执行的 update。
    // 所以我们要确保 在 newupdate 前, 待执行的 update 被执行完。因为 本节点
    // 的值 会被 parent 用到。
    // (看这个方法的 最后一行, 用到了child 的 值, 所以 本节点的parent 会用到
    // 本节点的值)
    if (lazy[si] != 0)
    {
        // 当前节点上 执行 待执行的 update。
        tree[si] += (se-ss+1)*lazy[si];

        // 如果有child, 需要把 lazy 的值 下放到 child上。
        if (ss != se)
        {
            lazy[si*2 + 1] += lazy[si];
            lazy[si*2 + 2] += lazy[si];
        }

        lazy[si] = 0;
    }

    // out of range // 这行怎么不是 第一行。。
    if (ss>se || ss>ue || se<us)
        return ;
}

// 觉得不太对啊。
// 首先 out of range 应该是第一行。
// 第二, 应该先判断 是否被完全覆盖, 然后 检查 lazy[si]吧。
// 假设 lazy[si] 不为0, 如果 si 依然被 update range 完全覆盖, 并不需要 执行
// pending的update 啊。只需要把 本次的 diff 在 加到 lazy[si] 上就可以了啊。
// ? 这里多做了一层(就是 diff 保存到 本node 就可以了, 但是这里保存到了 child
```

上。)。 不知道有什么含义。

//

// 这里是为了 query 的时候，因为 照这里的写法， query 的时候 只需要 判断  
lazy[si] 是否为0， 为0 ， 就可以直接取 tree[si]， 然后返回。

// 如果diff 保存到 本节点， 那么在 query 的时候 就需要 执行这里的 部分操作： 把  
diff 的值 \* range size， 加到 tree[si] 上， 然后把 diff 保存到 child 上， 然后返  
回。

// 不够lazy啊。 可能只有 update， 没有query。。

//

// 不， 不能延迟到 query时， 因为这里 最下面 使用了 tree[left-child] +  
tree[right-child] 来更新自己 。 所以 只要进入了这个 方法， 就必须确保 tree[si]  
是最新的， 不然 退出这个方法， 回到 上层， tree[child] 不是最新的， 会导致  
tree[si] 错误。

// 当前node 被 update range 完全覆盖。

if (ss>=us && se<=ue)

{

// Add the difference to current node  
tree[si] += (se-ss+1)\*diff;

// same logic for checking leaf node or not

if (ss != se)

{

    lazy[si\*2 + 1] += diff;  
    lazy[si\*2 + 2] += diff;

}

    return;

}

// 剩下的清空： 被部分覆盖。

int mid = (ss+se)/2;

updateRangeUtil(si\*2+1, ss, mid, us, ue, diff);

updateRangeUtil(si\*2+2, mid+1, se, us, ue, diff);

// 使用child 的值 来更新自己

    tree[si] = tree[si\*2+1] + tree[si\*2+2];

}

// us, ue, update range 的 start, end

void updateRange(int n, int us, int ue, int diff)

{

    updateRangeUtil(0, 0, n-1, us, ue, diff);

}

// si, 当前node 在 segment tree 中的index， 最初传0进来， 因为root的index 是0

// ss, se, 当前node 代表的 index 的 start 和 end

// qs, qe, query range 的 start 和 end

int getSumUtil(int ss, int se, int qs, int qe, int si)

```

{
    // 如果 当前node 有 pending 的update, 需要确保在 query 前, 执行完 pending
    // 的 update
    if (lazy[si] != 0)
    {
        tree[si] += (se-ss+1)*lazy[si];
        if (ss != se)
        {
            lazy[si*2+1] += lazy[si];
            lazy[si*2+2] += lazy[si];
        }
        lazy[si] = 0;
    }

    // Out of range
    if (ss>se || ss>qe || se<qs)
        return 0;

    // 当前node 的range 被query range 覆盖
    if (ss>=qs && se<=qe)
        return tree[si];

    int mid = (ss + se)/2;
    return getSumUtil(ss, mid, qs, qe, 2*si+1) +
           getSumUtil(mid+1, se, qs, qe, 2*si+2);
}

int getSum(int n, int qs, int qe)
{
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n-1, qs, qe, 0);
}

// 为[ss, se] subarr 构造 segment tree, 节点是 si。
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return ;

    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }
}
```

```

    }

    int mid = (ss + se)/2;
    constructSTUtil(arr, ss, mid, si*2+1);
    constructSTUtil(arr, mid+1, se, si*2+2);

    tree[si] = tree[si*2 + 1] + tree[si*2 + 2];
}

void constructST(int arr[], int n)
{
    constructSTUtil(arr, 0, n-1, 0);
}

int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    constructST(arr, n);

    printf("Sum of values in given range = %d\n",
           getSum(n, 1, 3));

    updateRange(n, 1, 5, 10);

    printf("Updated sum of values in given range = %d\n",
           getSum( n, 1, 3));

    return 0;
}

```

<https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction>

## Persistent Segment Tree

把segment tree 进行持久化，这样可以查看 各个 update 前的情况。

```

struct node
{
    int val;

    node* left, *right;

    node() {}
    node(node* l, node* r, int v)

```

```

{
    left = 1;
    right = r;
    val = v;
}
};

// input array
int arr[MAXN];

// root pointers for all versions
node* version[MAXN];

```

每次更新原数组的一个元素，segment tree 会修改  $\log N$  个节点，所以 每次update 都新建  $\log N$  个节点，保存为 version。

。。代码没有复制。

=====

## 主席树

。。就是上面的 Persistent Segment Tree

=====

## 树状数组

<https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/>

### Binary Indexed Tree or Fenwick Tree

考虑下面的问题 来理解 binary indexed tree。

我们有一个数组  $arr[0 .. n-1]$ ， 我们希望：

1. 计算前  $i$  个元素的 sum
2. 修改某个元素的值

### 方案

1. 不做任何预处理，query的时候 遍历前  $i$  个
2. prefix sum, update的时候需要遍历  $i$  后的元素

我们能做到 query 和 update 都是  $O(\log n)$  吗？

一种有效的方法是 segment tree。它的 query 和 update 都是  $O(\log n)$ 。

另一种方法是 Binary Indexed Tree，query 和 update 也是  $O(\log n)$ 。相比 segment tree，BIT 需要更少的空间，更容易实现。

BIT 使用数组来表达。假设数组是 `BITree[]`，每个 BIT 的节点 保存了 原数组的一些元素的 sum。BIT 数组的 长度等于 原数组。

构造

我们初始化 `BITree` 数组中的 每个元素为0。然后 对每个下标 调用 `update` 方法。

操作

`getSum(x)`: 返回 子数组  $[0 \dots x]$  的 sum。

// 使用 通过 原数组`arr[0..n-1]` 构造的 `BITree[0..n]` 来 计算 原数组`[0..x]` 的sum

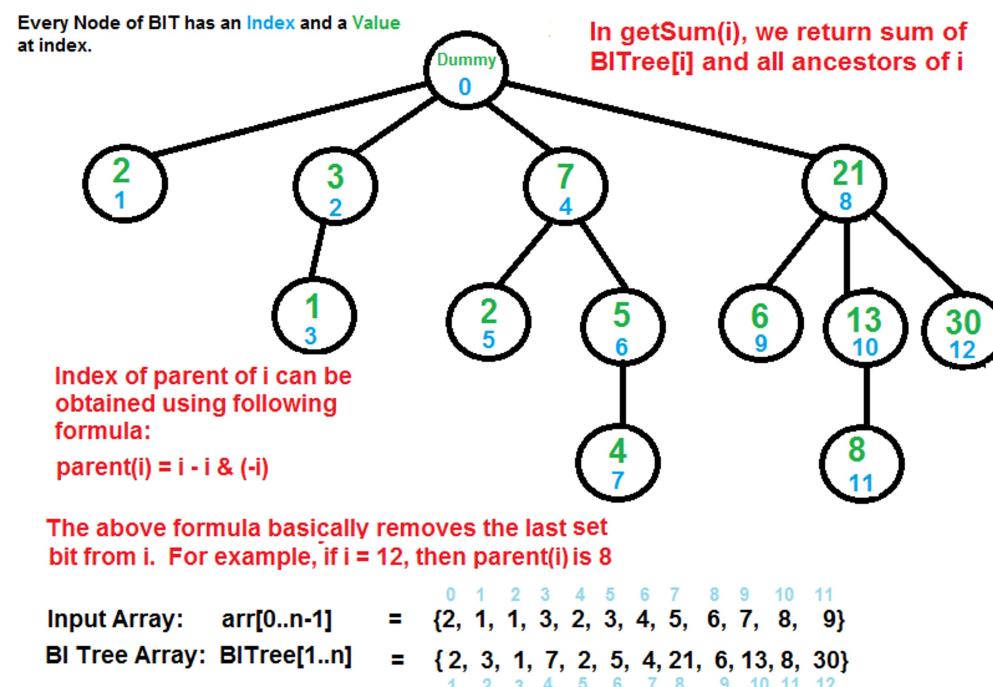
1. 初始化 sum 为0，当前index 为 `x + 1`

2. 当 当前index > 0 时，做：

a. 增加`BITree[index]` 到 sum

b. 移动到 `BITree[index]` 的 parent，parent 可以通过 移除 当前index 的二进制的 最后一个 1 来 获得。如，`index = index - (index & (-index))`

3. 返回 sum



**View of Binary Indexed Tree to understand `getSum()` operation**

`BITree[0]` 是 dummy node。

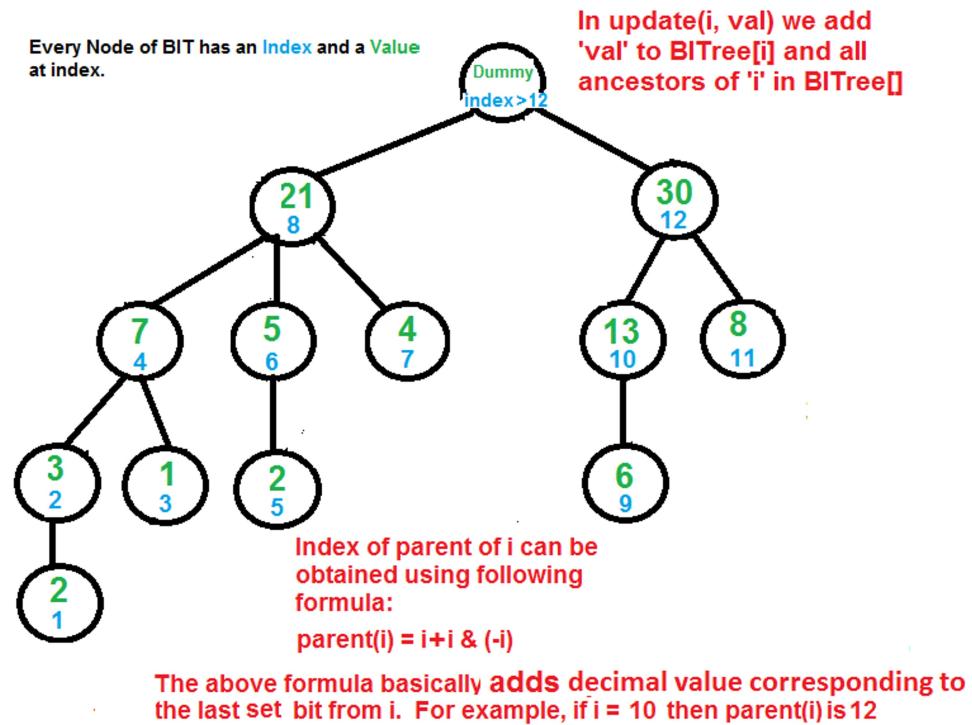
`BITree[y]` 是 `BITree[x]` 的parent，当且仅当 y 可以通过 移除 x的二进制的 最后一个 1 而获得，即  $y = x - (x \& (-x))$ 。

`BITree[y]` 的 child node `BITree[x]` 保存了  $[y, x]$  的sum

`update(x, val)` : 更新BIT，通过执行 `arr[index] += val`

// 注意, update(x, val) 不会修改 arr[], 只修改 BITree[]。

1. 初始化当前 index 为 x + 1
2. 当 当前 index <= n 时, 执行:
  - a. 增加val 到 BITree[index]
  - b. 移动到 BITree[index] 的 下一个元素, 下一个元素可以通过 增加 当前 index 的 最后一个 bit 1 来 实现, 如, index = index + (index & (-index))



Contents of arr[] and BITree[] are same as above diagram for getSum()

View of Binary Indexed Tree to understand update() operation

◦ ◦ ◦  
10 = 8 + 2  
00001010

-10  
11110101  
11110110

& = 0000 0010

◦ ◦ ◦

◦ ◦ 下标的顺序是 树的后续遍历。

How does Binary Indexed Tree work?

原理是基于 一个事实: 所有正数 都可以 表达为 2的次方的 和。  
例如, 19 可以表达为 16 + 2 + 1。

BITree 中的 每个 node 保存了 n个元素的 和 (n 是 2的次方)

例如，在第一张图中(getSum 的图)，前12个元素的 sum 可以通过 最后4个元素(9-12) + 上 第8个元素(从1-8) 来获得。

数字n 的二进制 的 bit 数量是  $O(\log n)$ 。因此，我们 遍历最多  $O(\log n)$  个node，在 getSum 和 update 操作中。构造BITree 的复杂度是  $O(n \log n)$ ，因为它 对每个 元素 调用一次 update()。

下面是BIT的实现

```
#include <iostream>
using namespace std;

// n: input array 的 元素个数
// BITree[0..n], BIT的数组形式
// arr[0..n-1], input array
int getSum(int BITree[], int index)
{
    int sum = 0;

    // BITree 中的下标 比 arr 中的下标 多1
    index = index + 1;

    while (index>0)
    {
        sum += BITree[index];
        index -= index & (-index);
    }
    return sum;
}

void updateBIT(int BITree[], int n, int index, int val)
{
    // BITree 比 arr 的下标 多 1
    index = index + 1;

    while (index <= n)
    {
        BITree[index] += val;
        index += index & (-index);
    }
}

int *constructBITree(int arr[], int n)
{
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update()
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    return BITree;
}

// Driver program to test above functions
```

```

int main()
{
    int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(freq)/sizeof(freq[0]);
    int *BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is " << getSum(BITree, 5);

    freq[3] += 6;
    updateBIT(BITree, n, 3, 6); // 上面是 [3] = 6, 这里也是 3, 6

    cout << "\nSum of elements in arr[0..5] after update is
" << getSum(BITree, 5);

    return 0;
}

```

计算range sum

$\text{rangeSum}(l, r) = \text{getSum}(r) - \text{getSum}(l-1)$ .

---

[https://cp-algorithms.com/data\\_structures/fenwick.html](https://cp-algorithms.com/data_structures/fenwick.html)

。。这个 BIT, 是 从0开始的。

设  $f$  是一些 group 操作 (具有元和逆元的集合上的 二元操作),  $A$  是长度  $N$  的数组。

Fenwick tree 是一种数据结构, 可以做到:

1. 在  $O(\log N)$  时间内 计算  $[1, r]$  区间内的  $f$ 函数 的值。
2.  $O(\log N)$  时间内 更新  $A$  中的一个元素的值
3. 需要  $O(N)$  空间
4. 方便地 使用和编码, 特别是在 多维数组的帮助下。

Fenwick Tree 最常见的 应用是 计算 range 的sum。

Fenwick Tree 也被称为 Binary Indexed Tree, 简写 BIT

第一次出现于 1994年的 "A new data structure for cumulative frequency tables" 论文, Peter M. Fenwick

为了简单起见, 我们假设  $f$  只是一个 sum 方法。

给定数组  $A[0 \dots N-1]$ , BIT 是一个数组  $T[0 \dots N-1]$ , 其中的每个元素等于  $A$ 数组的  $[g(i), i]$  区间的 sum。

Note: 这里的BIT 从下标0 开始。许多人使用 下标1开始的 BIT。因此在 implementation 章节，你会看到一个 下标1开始的 实现。2个版本 的 时间 空间 复杂度都是一样的。

对于上面提到的2个操作的一些伪代码： 获得 A 中[0, r] 的元素的和，更新 Ai：

```
def sum(int r):
    res = 0
    while (r >= 0):
        res += t[r]
        r = g(r) - 1
    return res

def increase(int i, int delta):
    for all j with g(j) <= i <= j:
        t[j] += delta
```

sum 的行为如下：

1. 增加  $[g(r), r]$  的 sum 到 result
2. 然后，跳到 range  $[g(g(r) - 1), g(r) - 1]$ 。然后增加这个range的 sum 到 result。
3. 一直跳，直到 从  $[0, g(\dots g(r) - 1) \dots -1]$  跳到  $[g(-1), -1]$ ， stop。

increase的行为如下：

1. 满足  $g(j) \leq i \leq j$  的  $[g(j), j]$  增加 delta，即  $t[j] += \text{delta}$ 。因此 我们更新 T 中 所有和  $A_i$  相关的 range 的值。

显然，sum 和 increase 的 复杂度都依赖于 g 函数。有许多种 g函数，只要  $0 \leq g(i) \leq i$ , for all i。例如  $g(i) = i$  是可以的，这个会导致  $T = A$ 。我们也可以使  $g(i) = 0$ ，这个 会导致 T 变成 prefix sum array。

Fenwick 算法的核心部分 就是，使用了一个 特别定义的 function g，来使得 2个操作都是  $O(\log N)$

定义g(i)

$g(i)$  的计算 使用了下面的 简单操作： 替换  $i$  的尾部(后缀) 1 为 0，直到  $i$  变成 0。

换句话说，如果  $i$  的二进制 的最低位 是0，那么  $g(i) = i$ 。如果 最低位 是 1，那么 把 后缀 1 全部 翻转 为 0。

例如：

```
g(11) = g(0b1011) = 0b1000 = 8
g(12) = g(0b1100) = 0b1100 = 12
g(13) = g(0b1101) = 0b1100 = 12
g(14) = g(0b1110) = 0b1110 = 14
g(15) = g(0b1111) = 0b0000 = 0
```

一个简单的实现是 使用 位操作：  $g(i) = i \& (i + 1)$ 。  
 $\dots + 1$  会使得 后缀1 全部变成 0，且 最后出现的 0 会变成1，然后  $\&$  就可以把

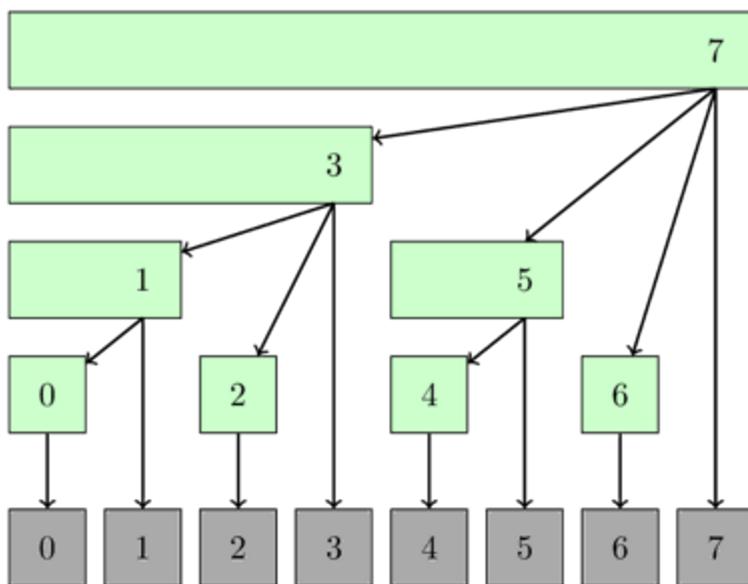
后缀 1 全部消掉了。

现在，我们只需要找到一个 方法 来遍历 所有 满足  $g(j) \leq i \leq j$  的  $j$ 。  
容易看到：我们可以 找到所有的  $j$ ，通过：以  $i$  为开始，然后 翻转 最后一个 值为0的bit。 我们称这个 操作是  $h(j)$ 。 例如，对于  $i=10$ :

```
10 = 0b0001010  
h(10) = 11 = 0b0001011  
h(11) = 15 = 0b0001111  
h(15) = 31 = 0b0011111  
h(31) = 63 = 0b0111111
```

也存在一个 简答的方法 来执行  $h$ :  $h(j) = j | (j + 1)$

下面的图片展示了 一个 BIT。



## Implementation

### Finding sum in one-dimensional array

这里提供了2个构造器，一个是新建一个arr，一个是使用原数组

```
struct FenwickTree {  
    vector<int> bit; // binary indexed tree  
    int n;  
  
    FenwickTree(int n) {  
        this->n = n;  
        bit.assign(n, 0);  
    }  
  
    FenwickTree(vector<int> a) : FenwickTree(a.size()) {  
        for (size_t i = 0; i < a.size(); i++)  
            add(i, a[i]);  
    }  
};
```

```

int sum(int r) {
    int ret = 0;
    for (; r >= 0; r = (r & (r + 1)) - 1)
        ret += bit[r];
    return ret;
}

int sum(int l, int r) {
    return sum(r) - sum(l - 1);
}

void add(int idx, int delta) {
    for (; idx < n; idx = idx | (idx + 1))
        bit[idx] += delta;
}
};


```

Finding minimum of  $[0, r]$  in one-dimensional array

显然，没有简单的方法来找到  $[l, r]$  区间的 min。

BIT 只能回答  $[0, r]$  区间的查询。另外，每当值被更新时，新的值必须小于当前值。这2个限制是因为 min 运算和整数集不能形成 group，因为没有逆元素。

```

struct FenwickTreeMin {
    vector<int> bit;
    int n;
    const int INF = (int)1e9;

    FenwickTreeMin(int n) {
        this->n = n;
        bit.assign(n, INF);
    }

    FenwickTreeMin(vector<int> a) : FenwickTreeMin(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            update(i, a[i]);
    }

    int getmin(int r) {
        int ret = INF;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret = min(ret, bit[r]);
        return ret;
    }

    void update(int idx, int val) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] = min(bit[idx], val);
    }
};


```

Note: 这是可能的：使用 Fenwick tree 来处理任意的 min query 和任意的 update。论文《Efficient Range Minimum Queries using Binary Indexed Trees》([http://ioinformatics.org/oi/pdf/v9\\_2015\\_39\\_44.pdf](http://ioinformatics.org/oi/pdf/v9_2015_39_44.pdf)) 描述了这种实现。但是，需要维护一个 second binary indexed tree，因为一个 tree 无法保存数组中所有元素的值。

### Finding sum in two-dimensional array

很容易就可以 实现 Fenwick Tree for 多维数组。

```
struct FenwickTree2D {
    vector<vector<int>> bit;
    int n, m;

    // init(...) { ... }

    int sum(int x, int y) {
        int ret = 0;
        for (int i = x; i >= 0; i = (i & (i + 1)) - 1)
            for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
                ret += bit[i][j];
        return ret;
    }

    void add(int x, int y, int delta) {
        for (int i = x; i < n; i = i | (i + 1))
            for (int j = y; j < m; j = j | (j + 1))
                bit[i][j] += delta;
    }
};
```

### 下标从1开始的 实现

我们修改了  $T[]$  和  $g()$  的定义。 我们希望  $T[i]$  保存 sum of  $[g(i)+1, i]$ 。 这会使得 代码有点不同。

```
def sum(int r):
    res = 0
    while (r > 0):
        res += t[r]
        r = g(r)
    return res

def increase(int i, int delta):
    for all j with g(j) < i <= j:
        t[j] += delta
```

$g(i)$  的计算是:  $i$  的二进制的 最低位(最后一个) 1 被 翻转为 0。

$$g(7) = g(0b111) = 0b110 = 6$$

$$g(6) = g(0b110) = 0b100 = 4$$

$$g(4) = g(0b100) = 0b000 = 0$$

最低位的 1 可以通过  $i \& (-i)$  来获得，所以 操作可以展开为:  $g(i) = i - (i \& (-i))$

不难看出，你需要 以  $i, h(i), h(h(i)) \dots$  的顺序 来修改  $T[j]$  的值， 当你 想更新  $A[j]$  时，  $h(i)$  的定义:  $h(i) = i + (i \& (-i))$

你可以看到，这个实现的主要好处是：二元操作可以相互补充。

```

struct FenwickTreeOneBasedIndexing {
    vector<int> bit; // binary indexed tree
    int n;

    FenwickTreeOneBasedIndexing(int n) {
        this->n = n + 1;
        bit.assign(n + 1, 0);
    }

    FenwickTreeOneBasedIndexing(vector<int> a)
        : FenwickTreeOneBasedIndexing(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int idx) {
        int ret = 0;
        for (++idx; idx > 0; idx -= idx & -idx)
            ret += bit[idx];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    void add(int idx, int delta) {
        for (++idx; idx < n; idx += idx & -idx)
            bit[idx] += delta;
    }
};

```

### Range operations

BIT 可以支持下面的range 操作:

1. point update 和 range query
2. range update 和 point query
3. range update 和 range query

#### 1. Point Update and Range Query

普通的 BIT 就可以。

#### 2. Range Update and Point Query

使用简单的技巧，我们可以执行相反的操作： range increase 和 单个值的查询。

将 BIT 初始化为全0。假设我们想要增加[1, r] 每个元素的值 by x。我们在 BIT 上执行 2次 point update，一次是 add(1, x)，一次是 add(r+1, -x)

如果我们想要获得 A[i] 的值，我们只需要 使用 普通的 range sum 方法 就可以获得 prefix sum。

为了证明正确性，让我们聚焦于 上面的 increase 操作，如果  $i < 1$ ，那么 2次 update 操作 不会对 query 造成影响，我们得到 sum 0。如果  $i$  在  $[1, r]$  区间内，那么我们得到 答案  $x$ ，因为 第一次的 update 操作，如果  $i > r$ ，那么 第二次的 update 会 撤销

第一次的 update 的结果。

下面的实现是 下标从1开始的。

```
void add(int idx, int val) {
    for (++idx; idx < n; idx += idx & -idx)
        bit[idx] += val;
}

void range_add(int l, int r, int val) {
    add(l, val);
    add(r + 1, -val);
}

int point_query(int idx) {
    int ret = 0;
    for (++idx; idx > 0; idx -= idx & -idx)
        ret += bit[idx];
    return ret;
}
```

Note: 这个也可以 增加单个值 A[i]， 通过 range\_add(i, i, val)

### 3. Range Updates and Range Queries

为了支持 range update 和 range query，我们需要 2个BIT，一个B1[], 一个B2[]，用0 初始化。

假设 我们想要增加[l, r] 区间的每个值 by x。类似上面的方法， 我们执行2次 point update 在 B1 上： add(B1, l, x) 和 add(B1, r+1, -x) 。 我们也更新B2，具体后面说。

```
def range_add(l, r, x):
    add(B1, l, x)
    add(B1, r+1, -x)
    add(B2, l, x*(l-1))
    add(B2, r+1, -x*r))
```

在 range update (l, r, x) 后， range sum query 应该返回下面的值

```
sum[0, i] =
    1. 0, i<l
    2. x*(i-(l-1)), l<=i<=r
    3. x*(r-l+1), i>r
```

我们可以把 range sum 写成 2项 的差，我们使用 B1 代表第一项， B2代表第二项。对于这2项的查询 的差 就是 [0, i] 的 prefix sum

```
sum[0, i] = sum(B1, i)*i - sum(B2, i) =
    1. 0*i - 0, i<l
    2. x*i-x*(l-1), l<=i<=r
    3. 0*i-(x*(l-1)-x*r), i>r
```

最后一个表达式就是所需的项， 所以， 我们可以使用B2 来移除 额外的项，当我们执行乘

法  $B1[i] * i$  时

我们可以执行任意的 range sum, 通过计算  $l-1$  和  $r$  的前缀和, 然后 获得 差

```
def add(b, idx, x):
    while idx <= N:
        b[idx] += x
        idx += idx & -idx

def range_add(l,r,x):
    add(B1, l, x)
    add(B1, r+1, -x)
    add(B2, l, x*(l-1))
    add(B2, r+1, -x*r)

def sum(b, idx):
    total = 0
    while idx > 0:
        total += b[idx]
        idx -= idx & -idx
    return total

def prefix_sum(idx):
    return sum(B1, idx)*idx - sum(B2, idx)

def range_sum(l, r):
    return prefix_sum(r) - prefix_sum(l-1)
```

=====

Splay

=====

分块

=====

莫队

=====

珂朵莉树

=====

哈夫曼树

=====

费马小定理

=====

裴蜀定理

又称 贝祖定理, Bézout's lemma

若  $a, b$  是整数, 且  $d = \gcd(a, b)$ , 那么对于 任意整数  $x, y$ ,  $ax + by$  一定是  $d$  的倍数, 特定地, 一定存在  $x, y$ , 使得  $ax + by = d$  成立。

一个重要推论:  $a, b$  互质的 充要条件 是 存在整数  $x, y$ , 使得  $ax + by = 1$   
。。2, 3 互质 :  $2*(-1) + 3*1 = 1$

---

n个整数间 的裴蜀定理

设  $a_1, a_2, a_3 \dots a_n$  为 n 个整数,  $d$  是它们的最大公约数, 那么存在整数  $x_1, x_2 \dots x_n$  使得  $a_1 * x_1 + a_2 * x_2 + \dots + a_n * x_n = d$  成立。

特别地, 如果  $a_1, a_2 \dots a_n$  存在 2 个数是互质的 (不必两两互质), 则存在 整数  $x_1, x_2 \dots x_n$  使得  $a_1 * x_1 + a_2 * x_2 + \dots + a_n * x_n = 1$  成立。

---

裴蜀可以推广到任意的 主理想环 上。设环 A 是 主理想环,  $a$  和  $b$  是 环中元素,  $d$  是它们的一个 最大公约元, 那么存在 环中元素  $x$  和  $y$  使得:  $ax + by = d$ 。这是因为 主理想环 中,  $a$  和  $b$  的最大公约元 被定义为 理想  $aA + bB$  的生成元。

。。。 主理想环 - 理想 - 主理想 - 环论 - 非结合代数 - 域论 - 泛函分析  
。。。 环 - 四元数 - 乘法不符合交换律 - 超复数 - 自旋(量子力学) - 幺正厄米复矩阵  
(幺正矩阵, 厄米矩阵, 复矩阵)  
。。。。。

---

=====

威尔逊定理(数论四大定理之一)

判断自然数 是否是 素数 的 充要条件: 当且仅当  $p$  为素数时:  $(p - 1)! \equiv -1 \pmod{p}$   
。。。 mod(负, 正)=正 mod(正, 负)=负。

充分性

1

4

>4 的完全平方数

$$\begin{aligned} p &= k^2 \\ 2k-p &= 2k - k^2 \\ &= 2k - k^2 - 1 + 1 \\ &= -(k-1)^2 + 1 < 0 \end{aligned}$$

所以  $k < p$ ,  $2k < p$  成立

$$\begin{aligned} (p-1)! &= 1 * 2 * \dots * k * \dots * 2k * (p-1) \\ &= k * 2k * n \\ &= 2n k^2 \\ &= 2np \end{aligned}$$

所以  $(p-1)! \equiv 0 \pmod{p}$

>4 的非完全平方数

$p$  必然等于 2 个不等数  $a, b$  的乘积

$$(p-1)! = 1 * 2 * \dots * a * \dots * b * \dots * (p-1)$$

$$\begin{aligned}
 &= a*b*n \\
 &= np \\
 \text{所以 } (p-1)! &\equiv 0 \pmod{p}
 \end{aligned}$$

必要性

证明：如果p是质数，则p可以整除  $(p-1)! + 1$

○ ○ ○

数论四大定理

## 白人定理 威尔逊定理

威尔逊定理

歐拉定理  
孙子定理

费马小定理

费马小定理

如果p是一个质数，整数a不是p的倍数，则有  $a^{(p-1)} \equiv 1 \pmod{p}$

是欧拉定理的一个特殊情况。

三九一

设  $a, m$  是 正整数，且  $\gcd(a, m) = 1$ ，则有

由 (m) 成为 对模 m 缩系的元素个数

。。缩系，应该就是： 小于  $m$  且 和  $m$  互质的数 的集合。 （。。1也是的）。。。 不是 小于  $m$  的素数 的集合。 比如 5和4 互质。 或者说 是  $\gcd = 1$  就是互质。

当  $m$  是素数时，变为 费马小定理

应用

用来简化幂的模运算。

比如计算  $7^{222}$  的个位数，实际上求的是  $7^{222}$  被10除的余数。7和10互质( $\gcd=1$ )，且  $\phi(10) = 4$ 。由欧拉定理知  $7^4 \equiv 1 \pmod{10}$ 。所以  $7^{222} = (7^4)^{55} * 7^{22} \equiv 1^{55} * 7^2 \equiv 49 \equiv 9 \pmod{10}$

## 孙子定理，中国余数定理

一元线性同余方程组

$$x \equiv a_1 \pmod{m_1}$$

$$x \equiv a_2 \pmod{m_2}$$

$$x \equiv a_3 \pmod{m_3}$$

...

$$x \equiv a_n \pmod{m_n}$$

中国剩余定理说明：假设整数  $m_1, m_2 \dots m_n$  两两互质，则对任意整数  $a_1, a_2, \dots, a_n$ ，上面的方程组有解，并且通解可以通过如下方式获得：

设  $M = m_1 * m_2 * \dots * m_n$ ，并设  $M_i = M/m_i$ ，即除  $m_i$  意外  $n-1$  个数的乘积。

设  $t_i = M_i^{-1} \pmod{m_i}$ ，即  $M_i$  模  $m_i$  的数论倒数（ $t_i$  为  $M_i$  模  $m_i$  意义下的逆元） $M_i t_i \equiv 1 \pmod{m_i}$ 。

方程组的通解： $x = a_1 t_1 M_1 + a_2 t_2 M_2 + \dots + a_n t_n M_n + kM$ 。  $k$  是整数。

在模  $M$  的意义下，只有一个通解  $x = (a_1 t_1 M_1 + \dots + a_n t_n M_n) \pmod{M}$

。。。余数  $a_1 * (M / m_i) * (M_i^{-1})$

$$x \equiv 2 \pmod{3}$$

$$x \equiv 3 \pmod{5}$$

$$x \equiv 2 \pmod{7}$$

$$M = 105$$

$$M_1 = 35$$

$$M_2 = 21$$

$$M_3 = 15$$

$$a_1 = 2$$

$$a_2 = 3$$

$$a_3 = 2$$

---

逆元

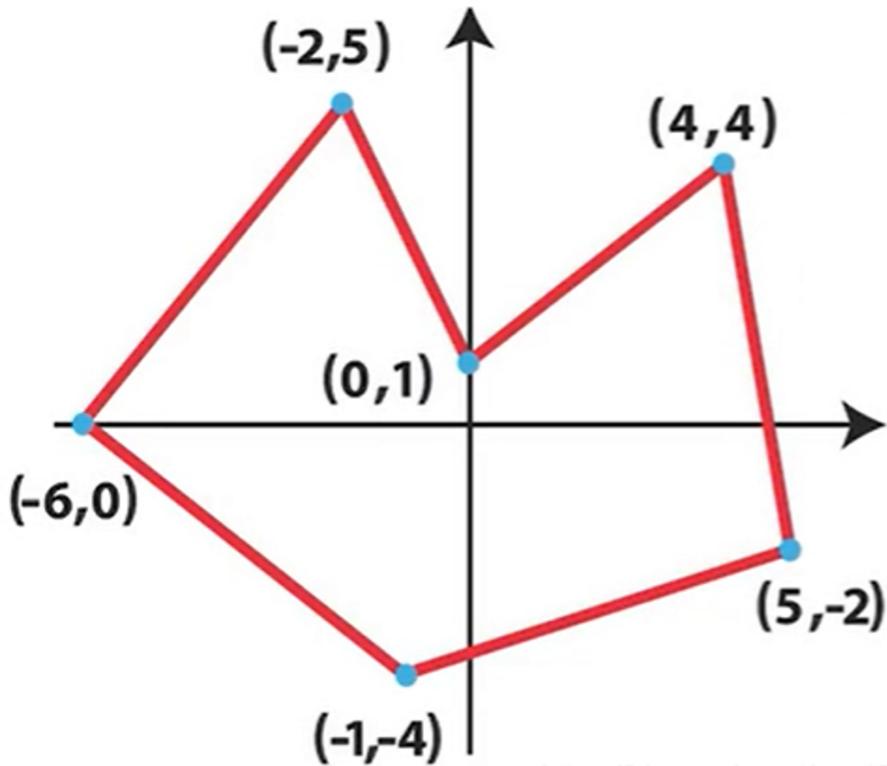
。。。

扩展欧几里得算法

---

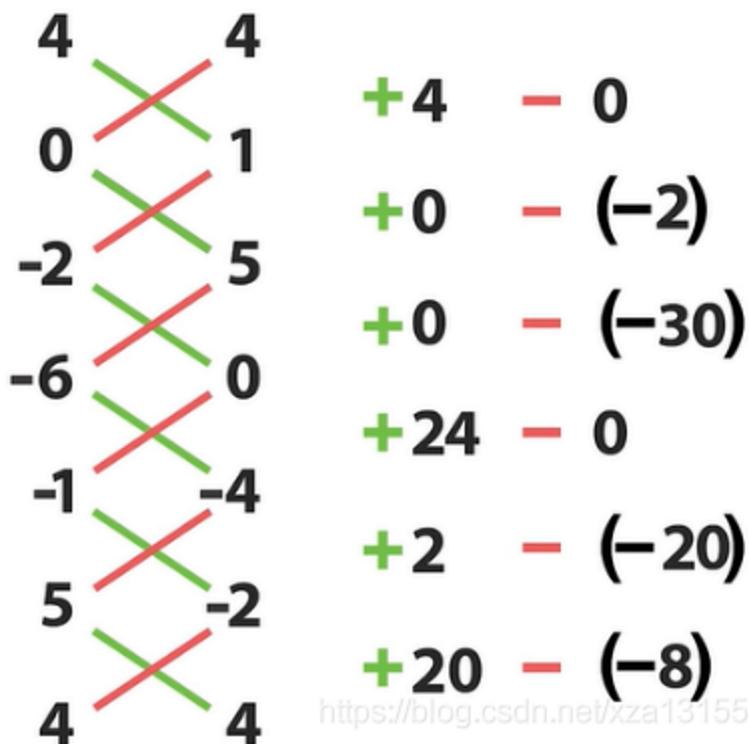
鞋带公式

求多边形面积



<https://blog.csdn.net/xza13155>

选择一个顶点，然后按照 逆时针 顺序读取坐标，最后回到起点。  
按照类似 系鞋带的顺序 将坐标 串联起来。



都是相乘，  $\text{sum}(\text{绿色}) - \text{sum}(\text{红色}) = 110$ ， 多边形的面积就是  $110/2=55$

这个是把 多边形 切分成 多个 三角形， 然后 向量计算 三角形面积。 都是 都需要/2， 所以 最后 /2

---

$$S_{\text{三角形}} = 0.5 * ((x_1*y_2 + x_2*y_3 + x_3*y_1) - (y_1*x_2 + y_2*x_3 + y_3*x_1))$$

现在 多边形上 取一条边， 加上原点(0, 0) 就组成 三角形， 套入上面的公式， 有原点(0, 0) (假设x1, y1 是原点)， 所以就等于  $1/2 * (x_2*y_3 - x_3*y_2)$   
就是上面的 绿线相乘 - 红线相乘。然后/2

然后按照顺序遍历所有的边。计算每个三角形

---

由A-->B-->C-->A 按逆时针方向转。(行列式书写要求) 设三角形的面积为S ， 则  
 $S = (1/2) * (\text{下面行列式})$

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

$$S = (1/2) * (x_1y_2 + x_2y_3 + x_3y_1 - x_1y_3 - x_2y_1 - x_3y_2)$$

即用三角形的三个顶点坐标求其面积的公式为：  $S = (1/2) * (x_1y_2 + x_2y_3 + x_3y_1 - x_1y_3 - x_2y_1 - x_3y_2)$ 。

---

差分数组

---

频繁对数组的 区间[i, j] 中每个元素做加减法。

比如， 对区间[a, b]中每个元素 +3， 然后 对[a+1, b-1]中每个元素 -2

差分数组是一个和原数组等长的数组， 其第i个元素表示 原数组[i]和[i-1]的差值， 即 原数组[i] - 原数组[i-1]

由原数组 得到差分数组

```
i == 0 : diff[0] = nums[0];
```

```
i != 0 : diff[i] = nums[i] - nums[i - 1];
```

由差分数组 得到原数组

```
i == 0: nums[0] = diff[0];  
i != 0: nums[i] = nums[i - 1] + diff[i];
```

给区间[i , j]增加val

```
diff[i] += val;  
if (j + 1 < diff.size())  
    diff[j + 1] -= val;
```

---

给你一个数组(或全0的数组)，然后多个区间，进行加减，最后求 数组和。

给你多个区间[x, y]，分别告诉你小于x获得多少，[x, y] 获得多少，大于x获得多少，  
然后 求 在哪里能获得最大值，最大值多少

多个区间[x, y, z, a, b, c]，<x多少，[x, y]多少，[y, z]多少，[z, a]多少，[a, b]多少，  
[b, c]多少，大于c多少。

---

=====

=====

欧几里得算法

辗转相除法

求2个数的最大公约数

---

=====

扩展欧几里得算法

旨在解决一个问题：求  $ax + by = \gcd(a, b)$ ，其中a, b为常数 的整数解

收集 辗转相除法 中产生的 式子，倒回去，可以得到  $ax + by = \gcd(a, b)$  的整数解。

扩展欧几里得算法 可以用于 计算 模逆元

```
int exgcd(int a, int b, int &x, int &y) {
```

```

//x为a的解, y为b的解
if (b==0) {//到达尽头
    x=1, y=0;
    //y可赋任意整数值
    return a;//返回最大公因数
}
int d=exgcd(b, a%b, x, y);
//此时的x, y为下一层的解
int temp=y;
y=x-a/b*y;//把y变成当前层解
x=temp;//把x变成当前层解
return d;
}
=====

=====

https://www.geeksforgeeks.org/range-minimum-query-for-static-array/

```

Range Minimum Query (Square Root Decomposition and Sparse Table)  
。平方根分解 与 稀疏表

我们有一个数组 arr[0…n-1]， 我们需要高效地 找到 L R 区间内的 min value。比如下面的情况

Input: arr[] = {7, 2, 3, 0, 5, 10, 3, 12, 18}  
query[] = [0, 4], [4, 7], [7, 8]

Output: min of [0, 4] is 0  
min of [4, 7] is 3  
min of [7, 8] is 12

一种方法是 遍历 L 到 R 来找到 min， 不需要额外的 时间空间，每次查询 O(n)，  
另一种是 segment tree，需要 额外的 O(n)的时间和空间 来构建 segment tree， 每次查询 O(logN)

Can we do better if we know that the array is static?

#### Method 1 (Simple Solution)

创建一个 2维数组， [i][j] 保存的就是 range[i, j] 的 min 。 query 时间复杂度是

$O(1)$ 。但预处理时 需要  $O(n^2)$  时间和空间

### Method 2 (Square Root Decomposition)

我们可以使用 平方根分解 来降低 上面方法需要的 空间。

预处理

1. 将  $[0, n-1]$  划分为  $\sqrt{n}$  个。。。 Divide the range  $[0, n-1]$  into different blocks of  $\sqrt{n}$  each。。。 不知道 是  $\sqrt{n}$  个block，还是 每个block  $\sqrt{n}$  个元素。。。 还有 是 上取整，还是下取整。。。 看第二步，应该是 每个 block  $\sqrt{n}$  个元素。但是不一定 正好是 平方数啊。  $\sqrt{n}$  个 和 每个  $\sqrt{n}$  都差不多，区别就是 最后一个block 中元素 是否能超过  $\sqrt{n}$
2. 计算每个  $\sqrt{n}$  大小的 block 的 min，并且保存下 结果。

预处理的 时间复杂度是  $O(\sqrt{n})^2 = O(n)$ ， 空间复杂度  $O(\sqrt{n})$



query:

1. 要查询 L 到 R 的 min，我们 获得 被 L R 范围 完全覆盖的 多个block 的 min。对于左侧 和右侧的 边界的block，可能被部分覆盖，我们 遍历它们 来搜索 min。  
时间复杂度  $O(\sqrt{N})$

更多信息看: <https://www.geeksforgeeks.org/sqrt-square-root-decomposition-technique-set-1-introduction/>

下面的代码来上上面的链接

```
void update(int idx, int val)
{
    int blockNumber = idx / blk_sz;
    block[blockNumber] += val - arr[idx];
    arr[idx] = val;
}

int query(int l, int r)
{
    int sum = 0;
```

```

        while (l < r and l % blk_sz != 0 and l != 0)
        {
            sum += arr[l];
            l++;
        }
        while (l + blk_sz - 1 <= r)
        {
            sum += block[l / blk_sz];
            l += blk_sz;
        }
        while (l <= r)
        {
            sum += arr[l];
            l++;
        }
    return sum;
}

// Fills values in input[]
void preprocess(int input[], int n)
{
    int blk_idx = -1;
    blk_sz = sqrt(n);
    for (int i=0; i < n; i++)
    {
        arr[i] = input[i];
        if (i % blk_sz == 0)
        {
            blk_idx++;
        }
        block[blk_idx] += arr[i];
    }
}

```

。看起来是 每个block 最多  $\sqrt{n}$  个， 不过 最后一个block，如果没有满，则不会被用到。 query的时候 由于  $l + \text{blk\_size}$  大于  $r$ ，所以进不了 最后一个 不满的block

### Method 3 (Sparse Table Algorithm)

上面的方法 只使用了  $O(\sqrt{n})$  的空间， 但是 每次query 使用  $O(\sqrt{n})$  的时间。

sparse table 支持 query  $O(1)$ ， 额外空间  $O(n \log n)$

基本思想是 预处理 每个  $2^j$  长度的 subarr，计算它的 min，  $j$  的范围是  $[0, \log n]$ 。

和method 1一样， 我们需要一个 lookup 数组，  $\text{lookup}[i][j]$  表示的是 从  $i$  开始，长度为  $2^j$  的subarr 的min。例如，  $\text{lookup}[0][3]$  保存了  $[0, 7]$  range 的min (从下标0开始，长度是  $2^3$ )

预处理：

lookup表中元素 如何生成? 方法很简单, bottom-up的方式, 使用 上次计算的值 来 fill table。

例如, 为了 找到 range[0, 7] 的min, 我们会使用到 2个range 来计算min: range [0, 3], range[4, 7]。

基于上面的例子, 下面是 伪代码

```
// If arr[lookup[0][2]] <= arr[lookup[4][2]],  
// then lookup[0][3] = lookup[0][2]  
If arr[lookup[i][j-1]] <= arr[lookup[i+2^(j-1)][j-1]]  
    lookup[i][j] = lookup[i][j-1]  
  
// If arr[lookup[0][2]] > arr[lookup[4][2]],  
// then lookup[0][3] = lookup[4][2]  
Else  
    lookup[i][j] = lookup[i+2^j-1][j-1]
```

。 。 计算[i][j], 用到了 [i][j-1] 和 [i+2^(j-1)][j-1]

7	2	3	0	5	10	3	12	18
0	1	2	3	4	5	6	7	8

arr[]

0 1 3 3	
1 1 3 3	lookup[i][j] contains index of
2 3 3 _	minimum in range from arr[i] to
3 3 3 _	arr[i + 2^j - 1]
4 4 6 _	
5 6 6 _	
6 6 _ _	
7 7 _ _	
8 _ _ _	

## lookup[][]

query:

对于每个[L, R], 我们需要 使用 2的次方 长度的 range。 每次使用 最接近的 2的次方。 我们需要做 最多一次的 比较 (比较 2个 2的次方长度的 range的 min 的 大小)。  
一个range 开始于L, 终于 L + 最大的2的次方。

另一个range 终于R, 开始于 R - 相同的最大的2的次方 + 1。

例如, 如果给定的range 是 [2, 10], 那么我们比较 [2, 9], [3, 10] 的 min。

。 。 range [2, 9] 就是 lookup[2, 3], range[3, 10] 就是 lookup[3, 3]

上面例子的 伪代码

```

// For (2,10), j = floor(Log2(10-2+1)) = 3
j = floor(Log(R-L+1))

// If arr[lookup[0][3]] <= arr[lookup[3][3]],
// then RMQ(2,10) = lookup[0][3]
If arr[lookup[L][j]] <= arr[lookup[R-(int)pow(2,j)+1][j]]
    RMQ(L, R) = lookup[L][j]

// If arr[lookup[0][3]] > arr[lookup[3][3]],
// then RMQ(2,10) = lookup[3][3]
Else
    RMQ(L, R) = lookup[R-(int)pow(2,j)+1][j]

```

我们只做了一次 比较，所以 时间复杂度是  $O(1)$ 。

```

// C++代码，完成 range min
// query O(1)
// 额外空间 和 预处理时间 都是  $O(n \log n)$ 
#include <bits/stdc++.h>
using namespace std;
#define MAX 500

// [i][j] 保存了 i为第一个元素， $2^j$  为长度的 subarr 的min
// 应该是 [n][logn]，这里固定下来，来使得代码简单一些。
int lookup[MAX][MAX];

// Structure to represent a query range
struct Query {
    int L, R;
};

// bottom-up 地 fill 数组
void preprocess(int arr[], int n)
{
    // 长度1 的subarr 的 min
    for (int i = 0; i < n; i++)
        lookup[i][0] = i;

    // 间隔 从小到大 开始计算 min
    for (int j = 1; (1 << j) <= n; j++)
    {
        // 以每个下标为开始， 计算  $2^j$  长度的 subarr 的 min
        for (int i = 0; (i + (1 << j) - 1) < n; i++)
        {
            if (arr[lookup[i][j - 1]] < arr[lookup[i + (1 << (j - 1))][j - 1]])
                lookup[i][j] = lookup[i][j - 1];
            else
                lookup[i][j] = lookup[i + (1 << (j - 1))][j - 1];
        }
    }
}

// Returns minimum of arr[L..R]
int query(int arr[], int L, int R)
{
    int j = (int)log2(R - L + 1);

```

```

if (arr[lookup[L][j]] <= arr[lookup[R - (1 << j) + 1][j]])
    return arr[lookup[L][j]];
else
    return arr[lookup[R - (1 << j) + 1][j]];
}

// Prints minimum of given
// m query ranges in arr[0..n-1]
void RMQ(int arr[], int n, Query q[], int m)
{
    // Fills table lookup[n][Log n]
    preprocess(arr, n);

    // One by one compute sum of all queries
    for (int i = 0; i < m; i++)
    {
        int L = q[i].L, R = q[i].R;

        // Print sum of current query range
        cout << "Minimum of [" << L << ", "
              << R << "] is "
              << query(arr, L, R) << endl;
    }
}

// Driver code
int main()
{
    int a[] = { 7, 2, 3, 0, 5, 10, 3, 12, 18 };
    int n = sizeof(a) / sizeof(a[0]);
    Query q[] = { { 0, 4 }, { 4, 7 }, { 7, 8 } };
    int m = sizeof(q) / sizeof(q[0]);
    RMQ(a, n, q, m);
    return 0;
}

```

=====

LSM tree

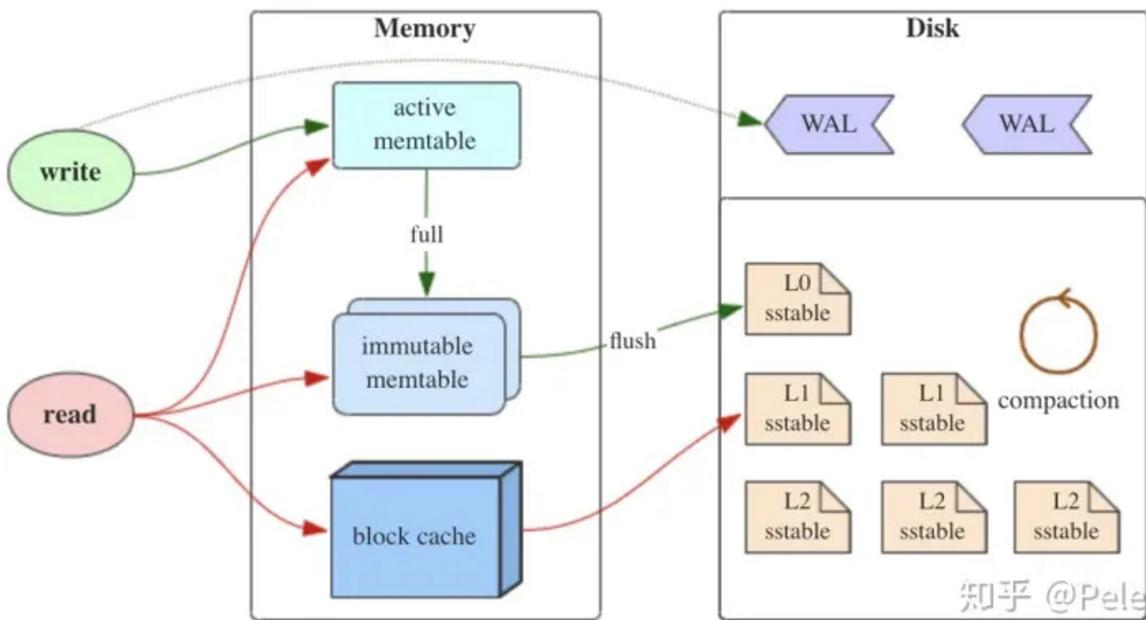
<https://zhuanlan.zhihu.com/p/181498475>

LSM树(log structured merge tree) 会给 初识者 一个错误的印象。

实际上，LSM 并不像 B+树，红黑树 那样 是一颗 严格的 树状数据结构，它其实是一种存储结构，目前 HBase, LevelDB, RocksDB 这些 NoSQL 存储 都是 采用LSM树

LSM树的核心特点是利用顺序写来提高写性能，但因为分层（此处分层是指分为内存和文件2部分）的设计会稍微降低读性能，但是通过牺牲小部分读性能换来高性能写，使得LSM树称为非常流程的存储结构

## LSM树核心思想



如上所示，有3个重要组成部分

### 1. MemTable

是内存中的数据结构，用于保存最近更新的数据，会按照 Key 有序地组织这些数据，LSM树对于具体如何有序地组织数据并没有明确的数据结构定义。例如 Hbase 使用跳表来保证内存中 Key 的有序。因为数据暂时保存在内存中，内存并不是可靠的存储，所以通常会通过 WAL(write-ahead logging，预写式日志) 的方式来保证数据的可靠性。

### 2. Immutable MemTable

当MemTable达到一定大小后，会转化为 Immutable MemTable，这个是将 MemTable 转化为 SSTable 的一种中间状态。写操作由新的 MemTable 处理，在转存过程中不阻塞数据更新操作。

### 3. SSTable (Sorted String Table)

有序键值对集合，是LSM树在磁盘中的数据结构，为了加快SSTable的读取，可以通过建立Key的索引以及布隆过滤器来加快Key的查找。

LSM树会将所有数据插入、修改、删除等操作记录保存在内存中，当此类操作达到一定的数据量后，再批量地顺序写入到磁盘中。这和B+树不同，B+树数据的更新会直接在原数据处修改对应的值，但是LSM树的更新是日志式的，一次数据的更新是通过append一条更新日志来完成的。这样设计的目的是为了顺序写，不断地将Immutable MemTable刷新到磁盘即可，而不用去修改以往的SSTable中Key的值，保证了顺序写。

因此 在不同的 SSTable中，可能存在 相同key 的记录，当然 最新的 那条 记录 才是准确的。这样的设计 虽然 大大提高了 写性能，但是 带来了一些问题

1. 元余存储，对于 某个Key，除了最新的那条以外，其他的记录 都是冗余的，但是仍然占据了 存储空间。因此需要 进行 Compact操作（合并多个 SSTable）来清除冗余数据。
2. 读取时 需要从 最新的 倒着查。最坏情况要查询完 所有的 SSTable，这里可以通过 前面提到的 index, bloom filter 来优化查找速度

### LSM的Compact策略

从上面可以看出，Compact 操作 是十分关键的操作，苟泽 SSTable 数量会 不断膨胀。这里主要介绍 2 种 基本策略。

先介绍3个比较重要的 概念，实际上 不同的策略 就是 围绕 这3个 概念 之间 做出 权衡 和 取舍。

1. 读放大：读取数据时 实际读取的数据量 大于 真正的数据量。例如 在LSM树中 需要先在 MemTable 查看 当前 key 是否存在，不存在 则继续从 SSTable 中寻找
2. 写放大：写入数据时 实际写入的数据量 大于真正的 数据量。例如，在 LSM树 中写入时 可能触发 Compact 操作，导致 实际写入的 数据量 远大于 该 Key 的数据量
3. 空间放大：数据实际占用的 磁盘空间 比数据真正的大小更多。 就是上面提到的冗余存储。

### size-tiered 策略

保证每层 SSTable 的大小相近，同时限制每层SSTable 的数量。

每层限制 SSTable 为 N，当每层 SSTable 达到N后，触发 Compact 操作 合并这些 SSTable，将 合并后的 结果写入到 下一层 成为一个 更大的 SSTable。

由此可以看出，当层数达到一定数量时，最底层(。。是从上往下合并的 ) 的 单个 SSTable 的大小会变得 非常大。

并且 该策略 导致 空间放大比较严重。即使在 同一层的SSTable，每个 key 的记录是可能存在 多份的，只有当该层的 SSTable 执行 compact 时 才会消除 这些 key的 冗余记录。

。。只要不是 最顶层，不可能 空间放大。。。不，可能是 第一层 3个 合并了，放到 第二层中，然后 第一层又 有了 3个，并且 第一层中 和 第二层中 有相同的 key，然后 第一层进行合并，放到 第二层，那么 第二层 就有重复了。

。。 所以 同层 会重复。 不同层也有重复。

### leveled策略

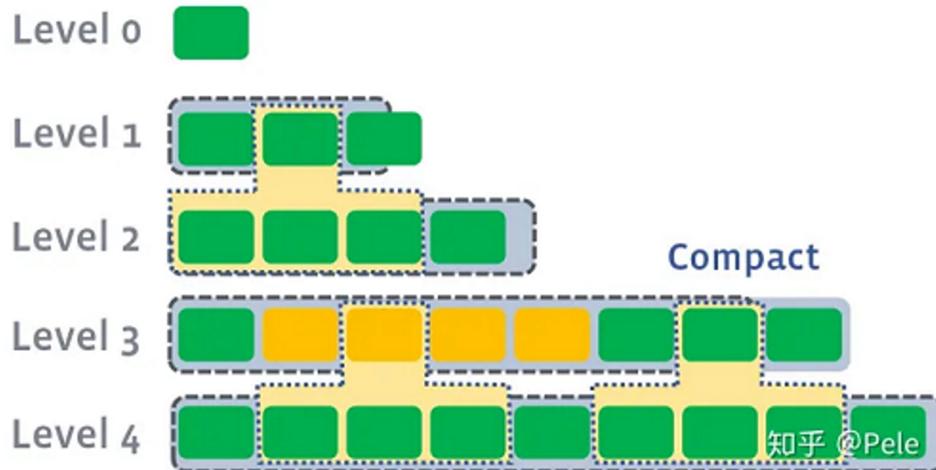
也是分层的思想，每一层限制 总文件 的大小。

但是和 size-tiered 不同的是，leveled 会将 每层 切分成 多个 大小相近的 SSTable。这些 SSTable 在这一层 是 全局有序的，意味着，一个key 在每层 至多只有一条记录，不存在冗余记录。

当第一层L1 的 总大小 超过 大小限制时，会从 L1中 选择至少一个文件，然后把 它跟 L2 有交集的部分(非常关键) 进行 合并，生成的文件 会放在 L2

如果L1 的 第二个SSTable 的 key 范围 覆盖了 L2中 前3个 SSTable, 那么就需要在 L1的第二个 和 L2的 前3个 SSTable 中 执行 compact 操作  
如果L2 在 compact后 超过了 限制, 那么就重复之前的操作: 选至少一个文件, 然后合并到 下一层。

多个不相干的合并是可以并发进行的。



leveled策略相较于 size-tired 策略来说, 每层内 key 是不会重复的, 即使最坏的情况, 除开最底层外, 其余层 都是 重复key, 按照相邻层 大小比例 为 10 来算, 冗余占比也很小, 因此空间放大问题得到缓解。但是 写放大问题 更加突出。举一个 最坏的场景, 如果 某层的 某个SSTable 的 key 范围跨度非常大, 覆盖了 下一层的 所有 key 的范围, 那么 进行 compact时 将涉及 下一层的 全部数据。

---

=====

用卡特兰数来求出栈序列个数

---

=====

## 指派问题，匈牙利算法

需要完成n个任务，正好有n个人，每个人的专长不同，所以完成任务的代价不同。  
应该为人 指派什么问题，使得完成 n 项任务的 总代价最小。

这类问题，根据 人员和代价(收益) 建立矩阵，称为 效率矩阵 或系数矩阵，其中元素  $c_{ij} > 0$  表示 第i个人 完成第j个任务的 效率(或时间，成本等)

代价矩阵有一个性质，如果从指派问题的 系数矩阵的 某行(列) 各元素 分别减去 或者 加上常数 k， 其最优任务 分解问题不变。

匈牙利算法 实际上有2个算法，分别解决 指派问题 和 二分图最大匹配求解问题。此处指的是 解决指派问题的 匈牙利算法。

### 第一步

矩阵经过变换，在各行各列中都出现 0 元素。

使指派问题的 系数矩阵变换，在 各行各列中 都出现 0 元素

从 系数矩阵的 每行元素减去 该行的最小元素

从所得系数矩阵的 每列元素 减去 该列的最小元素。

如果某行(列) 已有 0 元素，那就不必再 减了。

每行每列 最小元素 非负

### 第二步

进行试指派，以寻求最优解。为此，按以下步骤进行

经第一步变换后，系数矩阵中 每行每列 都已有了 0 元素， 但需要找出 n 个 独立的 0 元素。如果能找到，就以这些 独立 0 元素 对应 解矩阵  $(x_{ij})$  中 的元素 为1， 其余为0，就得到最优解。

步骤为

1. 从 只有一个0元素的 行 开始，给这个 0 元素加圈，记做@，这表示 对这 行 所代表的人，只有一种 任务可以指派。然后 划去@ 所在 列 的其他 0元素，记做Φ。这表示 这列所代表的 任务已经指派完，不需要考虑其他人
2. 只有 一个0元素的 列 的 0元素加圈，记做@，然后 划去 @所在的行的 0 元素，记做 Φ。
3. 反复进行1,2，直到所有 0元素都被 圈出 和 划掉。
4. 如果 仍有 没有 画圈的 0元素，且 同行(列) 的 0元素至少有2个。这可 以用不同的方案 去试探。 从剩余0元素 最少的 行(列) 开始，比较这行 各 0 元素所在列中0元素的数目，选择0元素少的 那列的 0元素 加圈。 然后划掉同行同列的 其他0元素。 反复进行，直到所有0元素 都已圈出 和 划掉为止。
5. 如果 @元素 的数目m 等于 矩阵的阶数n，那么 指派问题的 最优解已经 得到，如果 m < n， 则进入下一步。

### 第三步

( $m < n$  时的处理方法)： 作最少的直线 覆盖 所有0元素， 以确定 该系数矩阵中

能找到 最多的 独立元素数。

为此按以下步骤进行：

1. 对没有@ 的行 打 √ 号；
2. 对已打√ 的行中 所有 包含@元素 的 列 打 √。
3. 再对 打√的 列 中 含有@元素的 行 打√。
4. 重复2, 3, 直到得不出新的 √。
5. 对没有 √ 的行 画一条横线， 对有 √ 的 列画一条 纵线，这样 就得到了 覆盖所有 0元素的 最少直线数 l。 如果  $l < n$ , 说明必须 变换当前 系数矩阵，才能找到 n 个独立的 0元素，因此需要转第四步；如果  $l=n$ , 而  $m < n$ , 则 应该回到 第二步的4, 另行试探。

#### 第四步

对矩阵进行变换的目的是增加0元素。

为此，在没有被直线覆盖的部分 找出最小元素，然后在 打√ 行 各元素都减去这个 最小元素，而在打√ 列 的各元素 都加上这个 最小元素，以保证原来的 0 元素不变。

这样得到 新的 系数矩阵（它的最优解 和 原问题相同）。如果得到 n 个独立的 0 元素，则已经得到最优解，否则 回到 第三步重复进行。

=====

#### 匈牙利算法 - 二分图最大匹配

用于求解 无权二分图的 最大匹配

二分图，有2个点集，集合内部没有边相连，集合之间有边相连，如果存在这样的划分，则此图为一个二分图。二分图的一个等价定义：不含有〔含奇数条边的环〕的图。

匹配，在图论中，匹配 是一个边的集合，其中 任意两条边 没有公共顶点。

匹配点，匹配边，非匹配点，非匹配边

最大匹配，一个图 的所有匹配中，所含匹配边数最多的匹配，称为这个图的 最大匹配。

完美匹配，如果一个图的某个匹配中，所有的顶点都是匹配点，那么它就是一个完美匹配。显然，完美匹配一定是最小匹配（完美匹配的 每个点都已经匹配，添加一条新的

匹配边 一定会与现有的匹配边冲突)

交替路，从一个 未匹配点出发，依次经过非匹配边，匹配边，非匹配边。。。形成的 路径。

增广路，从一个 未匹配点出发，走交替路，如果途经另一个未匹配点（出发点不算），则这条交替路称为 增广路

增广路有一个重要特点：非匹配边 比 匹配边多一条。因此，研究增广路的意义是 改进匹配。只要把增广路中的 匹配边 和 非匹配边 的身份交换即可。由于 中间的匹配节点不存在 其他相连的匹配边，所以这样做不会破坏匹配的性质。交换后，图中的匹配边数目 比原来 多了一条。

我们可以通过不断地 找增广路 来增加匹配中的 匹配边 和 匹配点。找不到增广路时，达到最大匹配(这是增广路定理)。 匈牙利算法正是这么做的。

### 匈牙利树

一般由 BFS 构造。 从一个未匹配点出发运行bfs（唯一的限制是，必须走交替路），直到不能再扩展为止。

### 匈牙利算法要点如下

1. 从左边第一个顶点开始，挑选未匹配点进行搜索，寻找增广路
  1. 如果经过一个 未匹配点，说明寻找成功。更新路径信息，匹配边数+1，停止搜索
  2. 如果一直没有找到增广路，则不再从这个点开始搜索。事实上，此时搜索后会形成一颗匈牙利树。我们可以永久性地把它从图中删去，而不影响结果。
2. 由于找到增广路之后需要 沿着路径更新匹配，所以我们需要一个结构来记录路径上的点。DFS版本通过 函数调用 隐式地使用栈，而BFS使用 prev 数组

对于稀疏图，BFS 明显快于DFS。 稠密图，不相上下。

### 最大匹配数，最大匹配的匹配边的数目

最小点覆盖数，选取最少的点，使任意一条边至少有一个端点被选择

最大独立数，选取最多的点，使任意所选2点均不相连

最小路径覆盖数，对于一个 DAG (有向无环图)，选取最少条路径，使得每个顶点属于且仅属于一条路径。路径长可以为0 (即单个点)

定理1：最大匹配数 = 最小点覆盖数 (这是 Konig 定理)

定理2：最大匹配数 = 最大独立数

定理3：最小路径覆盖数 = 顶点数 - 最大匹配数

### Konig 定理

由匈牙利数学家柯尼希 (D. Konig) 于1913年首先陈述的定理。

定理的内容：在0-1矩阵中，1的最大独立集合最小覆盖包含的元素个数相同，等价地，二分图中的最大匹配数等于这个图中的最小点覆盖数。

---

匈牙利算法的核心在于：在 A 集合中选择一个点，然后 将与其相连的 B 中的点 依次对照，如果 B 中的点尚未匹配，那就将这2个点进行匹配，然后遍历 A 中下一个点，继续访问与其相连的 B 中的点，如果 B 中的点 已经被匹配了，那么就尝试 递归地 将与B中这个点 相匹配的 A 中的点 换一个 匹配对象。  
这其实就是在 寻找 增广路。

---

---

=====

CF 1633 E  
<https://www.wenjiangs.com/doc/zapnony9>

Spanning Tree 生成树  
是图G 的子集，使用最少的边 覆盖了所有顶点。因此，生成树没有 环，即 移出任意一条边后 会变成 非连通图。

---

最小生成树  
Kruskal  
Prim

---

=====

ternary search

三分查找用来确定函数在凹/凸区间上的极值点

=====

组合

```
// big number, nCr
// p MUST be prime and less than 2^63
uint64_t inverseModp(uint64_t a, uint64_t p) {

    uint64_t ex = p - 2, result = 1;
    while (ex > 0) { // 这个就是 快速幂 来求 逆元
        if (ex % 2 == 1) {
            result = (result * a) % p;
        }
        a = (a * a) % p;
        ex /= 2;
    }
    return result;
}

// p MUST be prime
uint32_t nCrModp(uint32_t n, uint32_t r, uint32_t p)
{
    if (r > n - r) r = n - r;
    if (r == 0) return 1;
    if ((n / p - (n - r) / p) > r / p) return 0;

    uint64_t result = 1; //intermediary results may overflow 32 bits

    for (uint32_t i = n, x = 1; i > r; --i, ++x) {
        if (i % p != 0) {
            result *= i % p;
            result %= p;
        }
        if (x % p != 0) { // 费马小定理: x 不是 p 的倍数
            result *= inverseModp(x % p, p);
            result %= p;
        }
    }
}
```

```
        return result;  
    }
```

=====

=====

$$(a/b) \bmod m = (a \bmod(m * b)) / b$$

证：

设  $a/b \bmod m = x$

则：

$$a/b = km + x$$

$$a = kbm + bx$$

$$a \bmod bm = bx$$

$$a \bmod bm / b = x$$

。。对大数的 组合排列，没有用，因为  $bm$  是一个大数，无法计算的。

-----  
通过逆元求解

费马小定理：如果  $p$  是质数，且  $a$  不是  $p$  的倍数，则  $a^{(p-1)} \equiv 1 \pmod{p}$

。。如果  $a$  是  $p$  的倍数，则  $a^{(x)} \bmod p = 0$  ( $x > 0$ )

逆元：如果存在  $x$ ，使得  $ax \equiv 1 \pmod{p}$ ，那么  $x$  就是  $a$  的逆元。

$a^{(p-1)} = a * a^{(p-2)} = 1 \pmod{p}$ ，所以  $a^{(p-2)}$  就是  $a$  的逆元。

下面的1 使用了  $b^{(c-1)} \bmod c = 1$

$(a/b) \bmod c$

$$\Leftrightarrow a * b^{(-1)} \bmod c$$

$$\Leftrightarrow a * (b^{(c-1)}) * b^{(-1)} \bmod c$$

$$\Leftrightarrow a * b^{(c-2)} \bmod c$$

一般  $c$  很大，所以要用快速幂

=====

Lucas卢卡斯定理

$$C(n, m) \% p = C(n/p, m/p) * C(n \% p, m \% p) \% p$$

条件:  $n, m \leq 10^{18}$ ,  $p$  为素数,  $p \leq 10^5$

```
11 C(11 n, 11 m) {  
    11 ans=1;  
    for(11 i=1; i<=m; i++) {  
        ans=ans*(n-m+i)/i; // 这里可以 mod 吧  
    }  
    return ans;  
}  
int Lucas(11 n, 11 m) {  
    if(n==0) return 1;  
    return C(n%p, m%p)*Lucas(n/p, m/p)%p;  
}  
。。 $n \% p$  以后, 再递归调用 Lucas 也没用, 因为不会变小了, 所以需要硬算。
```

$$c(m, n) = c(m-1, n-1) + c(m-1, n)$$

ST表, (sparse table, 之前有)

用于解决 可重复贡献问题

可重复贡献问题: 对于运算  $op$ , 运算的性质满足  $x op x = x$ , 则对应的 区间查询就是一个 可重复贡献问题, 例如, 最大值满足  $\max(x, x) = x$ , 最大公因数满足  $\gcd(x, x) = x$ , 因此 RMQ 和 GCD 就是一个 可重复贡献的问题。

但是 区间和 就不满足 这个性质, 因为 在 求解 区间和的过程中 采用的 预处理区间会发生在 重叠, 导致 重叠部分被重复计算, 因此 对于  $op$  操作 还需要 满足 结合律 才能使用 ST 表 进行求解。

题目: 给定  $n$  个数, 有  $m$  个查询, 对于每个查询, 你需要回答  $[l, r]$  区间内的 最大值。

暴力算法是  $O(n^2)$ 。

ST表 基于 倍增思想, 可以做到  $O(n \log n)$  预处理,  $O(1)$  回答。但是 不支持修改。所以 ST 表 是一种 离线的 数据结构

基于倍增思想，我们考虑如何求出 区间最值。可以发现，如果按照一般的 倍增流程，每次跳  $2^i$  步的话，询问时的 复杂度 仍然是  $O(\log n)$ ，并没有比 线段树 更优，而且 预处理 比 线段树 慢。

我们发现，区间最值 是一个 可重复贡献 的问题。即使 用来求解的 预处理 区间 有 重叠部分，只要这些 区间的 并 是 所求的区间，最终计算出来的 结果就是 正确的。

以最大值为例，设  $\text{arr}[i][j]$  表示整个数列 A 中 下标在  $[i, i+2^{j-1}]$  区间中的 最大值。递推的 边界是  $\text{arr}[i][0] = A[i]$ ，即数列A 在区间  $[i, i]$  中的最大值。

### 预处理

在递推时，我们把 子区间 的长度 成倍增加，于是 就可以得到下面的 递推表达式

$$\text{arr}[i][j] = \max(\text{arr}[i][j-1], \text{arr}[i+2^{j-1}][j-1])$$

得到代码：

```
inline void prework() {
    for(int i=1;i<=n;i++)
        f[i][0]=a[i];
    int t = log(n)/log(2) + 1;
    for(int j=1;j < t;j++) { // 必须 先枚举 倍增次数。
        for(int i=1;i+(1<<j)-1<=n;i++) {
            f[i][j]=max(f[i][j-1],f[i+(1<<(j-1))][j-1]);
        }
    }
}
```

### 查询

在查询任意区间  $[l, r]$  的最大值时，先计算出 一个  $k$ ，满足

$$2^k \leq r-l+1 < 2^{k+1}$$

也就是  $2$ 的 $k$ 次幂 小于 区间长度的前提下的 最大的  $k$ 。

左侧：

$$k \leq \log(r-l+1)/\log 2$$

右侧：

$$k > \log(r-l+1)/\log 2 - 1$$

所以  $k$  的上界就是  $\log(r-l+1)/\log 2$

因此， $[l, r]$  之间的最大值就是：

$$\max(f[l][k], f[r-(1<<k)+1][k])$$

```
inline int query(int l, int r) {
    int k=log(r-l+1)/log(2);
    return max(f[l][k], f[r-(1<<k)+1][k])
}
```

---

std::lg (它是  $O(1)$  的)

`_lg`在gcc上的实现是调的`_builtin_clz`, 编译出来直接是单独的汇编指令  
这函数生成一条汇编bsr指令, 有原生的支持

=====

DSU

UF

```
struct DSU {
    std::vector<int> f, siz;
    DSU(int n) : f(n), siz(n, 1) { std::iota(f.begin(), f.end(), 0); }
    int leader(int x) {
        while (x != f[x]) x = f[x] = f[f[x]];
        return x;
    }
    bool same(int x, int y) { return leader(x) == leader(y); }
    bool merge(int x, int y) {
        x = leader(x);
        y = leader(y);
        if (x == y) return false;
        siz[x] += siz[y];
        f[y] = x;
        return true;
    }
    int size(int x) { return siz[leader(x)]; }
};
```

=====

轮廓线DP

$n*m$  的棋盘, 放置  $1*2$  的 骨牌。 求能摆满棋盘的方案。

我们只需要统计 边界 上 格子的状态 就可以进行转移了。

1. 这个格子 向上放, 需要保证 它上面的格子是空的
2. 这个格子 向左放, 需要保证它左边的格子为空, 且 上面的格子已经填满
3. 不放, 需要保证 它上面的格子被填满。

总结：（这里使用了 滚动数据）

$dp[tmp][k \wedge p[j]] += dp[tmp^1][k], k \& p[j]$

$dp[tmp][k \mid\mid p[j-1]] += dp[tmp^1][k], (j > 1) \&\& !(k \& p[j-1]) \&\& (k \& p[j])$

$dp[tmp][k \mid\mid p[j]] += dp[tmp^1][k], (i > 1) \&\& (k \& p[j])$

。。 $k, p, j$  分别代表了什么？

。。根据  $\mid\mid$  说明 第二维 是 bool。但是  $tmp$  应该是 滚动数组的，所以 后面那维应该是 棋盘的 长或宽。

可能是  $\mid$ ，而不是  $\mid\mid$ ？

下面以 行数 远远大于 列数 作为前提（这样的话 滚动数组的 第二维 应该是 列数）

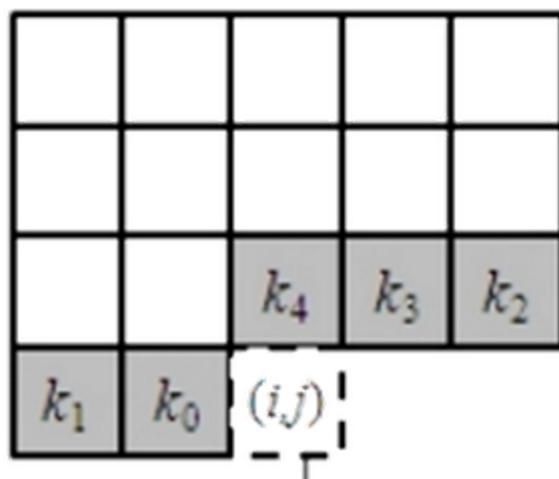
感觉  $k$  是代表 当前格子的 列值。

公式中的 逗号 后面应该是 条件，满足条件的时候 才执行前面的  $+=$ 。

根据 公式 上面的文字描述，感觉应该是 一一对应的。

但是  $p, j$  是什么， $p$  应该代表 本行。。估计  $p = grid[i]$ ？ $grid$  是  $n*m$  的棋盘。

。。文章有一张图，当时没有在意。。看  $k$  的下标。。是“蠕动数组”。。但是  $tmp$  是什么？感觉图文 不匹配。



。。。还真是 蠕动的数组。。下面的 讲得比较清

---

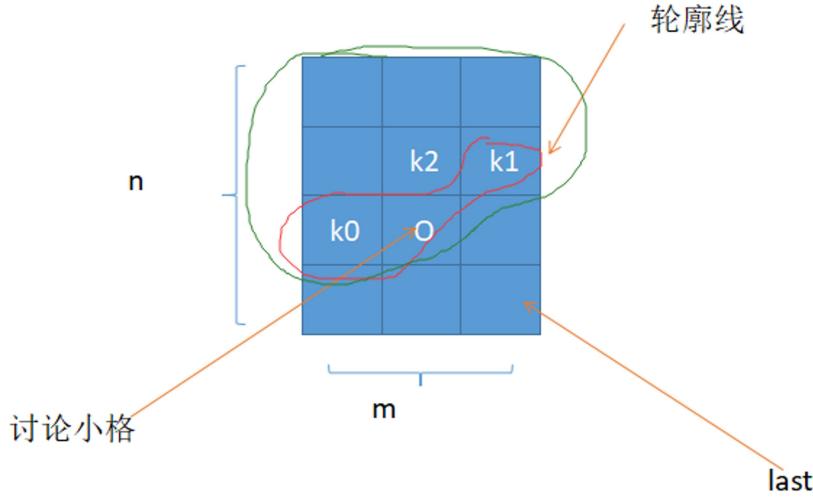
## 轮廓线DP

适用范文：较窄的棋盘。按 整行 或整列 无法进行 状态转移。而是把 轮廓线 作为 状态的一部分。

我们定义 0 为 非覆盖，1为覆盖。

对于每一个小格，轮廓线 包含： 该小格 和 该小格前面且确定该小格后状态还不确定的小格。

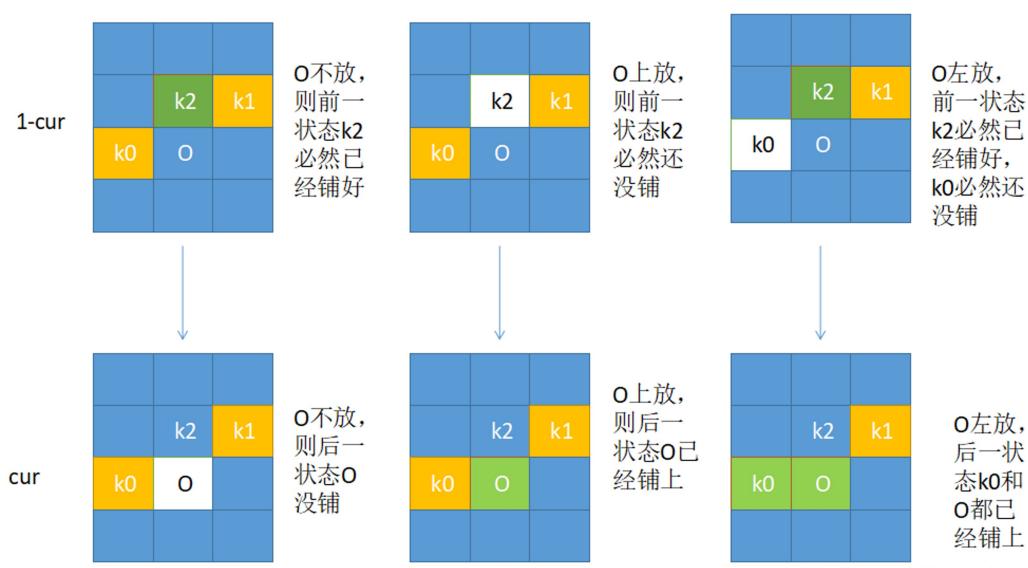
。。就是 该小格 + 该小格这行前面的格子 + 该小格上方的后面的格子。一行格子。不包括  $k2$



<https://wangpeiyi.blog.csdn.net>

因此，总共有  $2^m$  个状态（就是这个格子 被覆盖 或 没被覆盖）。所以 对于每个小格，我们需要分配  $2^m$  个状态，因此：

1. 定义  $dp[cur][S]$ ：当前所讨论 小格 轮廓线 内 状态为 S 时（例子中  $S = k_1k_2\dots$ ）（2进制），当前小格 和 当前小格前面的所有小格（绿色圈内区域）的总共铺放 方法数
2. 目标状态：  $dp[last][2^m - 1]$
3. 状态转移：
  1. 选择：对每个小格，由于我们只讨论 其 对前面区域的影响，所以可以选择：不放，左放，上放：
    1. 不放：  $O = 0$ ; 前一状态的  $k_2$  为 1。（。。就是当前位置是 0（代表没有覆盖），当前位置的上方（本例中为  $k_2$ ）必须已经 覆盖（即 1））
    2. 左放： 当前小格不能在第一列；  $k_0 = 0 = 1$ ; 前一状态  $k_0=0$ ,  $k_2=1$ 。
    3. 上放： 当前小格不能在第一行；  $O=1$ ; 前一状态的  $k_2 = 0$ 。



<https://wangpeiyi.blog.csdn.net>

2. 滚动数组：当前小格是在 前一小格的基础上讨论，所以采用 滚动数组：
  1.  $dp[x][S]$ ： 表示 前一个小格的 状态

2.  $dp[1-x][S]$ : 表示后一个 小格的状态。  $x$  属于  $\{0, 1\}$
3. 根据 转移的选择 及 条件, 有 状态转移方程式:
  1.  $A = dp[1 - cur][(s >> 1) | (1 << (m-1))]$
  2.  $B = dp[1 - cur][( (s >> 1) | (1 << (m-1))) \& ((1 << m) - 2)]$
  3.  $C = dp[1 - cur][ s >> 1 ]$

$dp[cur][S] =$  当前  $S$  满足不铺:  $A$   
                   当前  $S$  满足左铺:  $B$   
                   当前  $S$  满足上铺:  $C$   
                   当前  $S$  满足左铺+上铺:  $B+C$

这里  $A$  对应 不放,  $B$  对应左放,  $C$  对应上放。

初始状态:  $dp[0][2^m - 1] = 1$ , 其余为 0。(这里可以想象: 由于 第0行, 不能上方, 所以 -1 行 就 必须全部 被覆盖, 这样 第 0 行 就会 因为 -1 行 被覆盖了, 所以 无法 上放)。

初始状态 对应的 是 第 -1 行的 最后一个 小格。

```
while (cin >> m >> n)
{
    if (m + n == 0)
        return 0;
    if (n < m)
        swap(n, m);

    memset(dp, 0, sizeof(dp));
    cur = 0;
    dp[cur][(1<<m) - 1] = 1;
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < m; ++j)
        {
            cur ^= 1;
            memset(dp[cur], 0, sizeof(dp[cur]));
            for (int s = 0; s < (1 << m); ++s)
            {
                // 不放
```

。。  $s$  是增加本格子后的状态, 所以如果增加本格子后, 最后一位(即本格子) 是0, 即没有覆盖, 则取 上一个状态的  $s>>1 | (1 << (m-1))$

。。  $s>>1$  代表, 通过 本状态 获得 上次的状态, 因为 本状态 只是 在 上次的 状态上 去掉头, 然后 append 本格子(这里是0), 所以 要 获得 上一个 状态, 就需要 把本状态 后移一位, 这样 获得了 去掉头的上次状态,

然后由于 本格子不放, 那么就意味着 本格子上方的格子 必须被覆盖, 所以 上次的 状态 的 头 就是 被覆盖的。 所以  $| (1 << (m-1))$ 。

。。因为 本格子不放, 所以 本格子不会 上放, 所以 上方 的格子 必须已经被覆盖了, 不然 没有办法 再覆盖 上方的格子了 (因为 本格子不会放了)。

```
if (!(s & 1))
```

```
dp[cur][s] += dp[1 - cur][(s>>1) | (1 << (m - 1))];  
else  
{
```

// 左放

。。不能是第0列。

。。不过这个条件 没有看懂  $s \gg 1 \& 1$ , 这个应该是 要求 上一个状态的 最后一位是1, 即 被覆盖, 如果上一个状态的 最后一位 是1, 本格子 没有办法 左放啊。

。。左放, 也要求 本格子上方 的格子 已经被覆盖, 所以  $s \gg 1 | 1 \ll (m-1)$

。。但是 感觉是  $!(s \gg 1 \& 1)$  才对啊, 这样的话: 由于这里是 else, 所以 保证 本格子 被覆盖。

。。不不不, s 是 本格子处理后的 状态, 所以 确实 是要求 本格子 前面的 格子 (在处理(左放)完后) 被覆盖, 所以确实 是  $s \gg 1 \& 1$ 。

。。但是 dp 的 下标就不对啊。应该是 要求 上一个状态: 头被覆盖  $\&\&$  尾巴为0 啊  $\&\&$  本次的头也被覆盖。。。而现在  $s \gg 1 | (1 \ll m-1)$  的 尾巴 必然是 1, 因为 if 里的判断。

。。尾巴为0 就意味着 尾巴的 上方的 格子 必然 是 1。 (根据 不放 里的逻辑)。

。。所以应该是要求 上次 尾巴0  $\&\&$  本格子上方 已覆盖。

。。不, s 是 滚动的, 没有办法 限制 上次的 尾巴。而且 s 是 处理后的, 所以 处理后 上次的尾巴(这次的倒数第二) 必然是 1。不, 本次是1 不代表 之前的状态是1。

。。我感觉应该是  $dp[1-cur][(s \gg 2) \ll 1 | (3 < (m-2))]$  就是上个状态 的末尾 是0 且 开头2位是1 。 上个状态末尾0 代表 本次格子的 处理前 左侧是0, 这样就可以 进行 左放 处理, 并且 上个状态开头2个1 , 代表 本次格子 和 本次格子左侧, 这2个格子的 上方 都是 被覆盖了。

。。应该是 我认为的 这个。 可以看下面的 那段代码, 可以看到 左放的时候, 是要 求 之前的状态 尾巴 是0, 之后的状态 尾巴是 11。 所以 这里 我应该 搜索 上个状态 的尾巴 为0 的。 所以 要  $(s \gg 2) \ll 1$  , 这样 上个状态 尾巴是 0, 然后 再加上 头 是 11 。

。。找半天, 找不到一个 标准/通用的。。各种写法 都有。太难了。

```
if (j  $\&\&$  (s >> 1 & 1))  
    dp[cur][s] += dp[1 - cur][(s >> 1) | (1 << (m - 1))];
```

// 上放

。。只要不是 第0行, 都可以尝试 上放。 上放 就要求 上次状态的 头是0, 所以 这里 就是  $s \gg 1 \& (1 \ll (m-1) - 1) == s \gg 1$

```
if (i)  
    dp[cur][s] += dp[1 - cur][s >> 1];  
}  
}  
}  
}
```

```

void update(int a, int b) {
    dp[cur][b] += dp[cur^1][a]; //更新状态方案数
}
。。update() 第一个参数是原状态，第二个参数是新状态。

for(int k = 0; k < (1<<m); k++) { //枚举当前状态
    //当前和上都放 上有空位就不能往左，也不能不放
    if(i && !(k&(1<<(m-1)))) //不是第一行，且正上方为空
    {
        update(k, ((k<<1)^1)&mask); //新状态尾部置1
    } else
    {
        //当前和左放
        if(j && !(k&1)) //不是第一列并且左边为空
            update(k, ((k<<1)^3)&mask); //新状态尾两个11
        //不放
        update(k, (k<<1)&mask);
        //掩码是只取低m位作为状态
    }
}

```

---

```

54         for(int k = 0; k < (1<<m); k++) {
55             update(k, k<<1);
56             if(i && !(k&(1<<(m-1)))) update(k, (k<<1)^((1<<m)^1));
57             if(j && !(k&1)) update(k, (k<<1)^3);
58         }
59     }

38 void update(int a, int b) {
39     if(b & (1<<m)) dp[cur][b^(1<<m)] += dp[1-cur][a];
40 }

```

---

```

void update(int a, int b)
{
    if(b&(1<<m))
    {
        d[cur][b^(1<<m)]+=d[1-cur][a];
    }
}

```

```

for (int k=0;k<(1<<m);k++)
{
    update(k,k<<1); //不放
    if(i&&! (k&(1<<m-1)))
        update(k, (k<<1)^(1<<m)^1); //竖着放
    if(j&&! (k&1))
        update(k, (k<<1)^3); //横着放
}

```

。。很多都是 根据现在的状态 来 生成 下一个状态。  
 。。最开始那个 是根据现在的状态 来推算 上一个状态。

=====

## 状态压缩DP

传统DP 都是基于 整数的，比如 背包问题： 定义状态  $dp[i][j]$ ， 背包容量 为  $j$  时 前  $i$  件物品的 最大收益。 这里  $i$  取整数。

对于 状态压缩dp， 动态规划 是基于 集合的， 但是 我们使用 二进制 来将 这个集合 压缩成 一个 整数，这个 过程就是 状态压缩。

状态压缩 DP 常常用到 位运算 来 模拟 对集合的 操作

&  
|  
~  
^

## 旅行商问题

给定  $n$  个顶点 组成的 带权 有向图 的 距离矩阵  $d(i, j)$  ( $INF$  代表没有边)。要求从 顶点 0 出发，经过每个 顶点 恰好一次 后 再回到 顶点0。问 所经过 的 边 的总权重 的 最小值 是 多少？

限制：

$2 \leq n \leq 15$

$0 \leq d(i, j) \leq 1000$

定义  $dp[S][v]$  : 当已拜访节点 集合为  $S$ , 且 当前位置为  $v$  时, 回到 位置0 还需要 经过 的最短路径。

目标态  $dp[0][0]$  : 拜访所有城市再回到 0 所经过 的最短路径。

状态转移:  $dp[S][v] = \min\{dp[S \text{ 交集并 } \{u\}][u] + d(v, u) \mid u \text{ 不属于 } S\}$

。。我怎么觉得 不太对呢。

。。感觉是  $d(u, v)$  吧? 就是  $dp[S \text{ 包含 } u][u] + d(u, v)$ , 且  $v$  不属于  $S$  ?

。。这样的话 就是 所有 以  $u$  为 终点 的 路径 再次 向  $v$  走, 这些走法中 最min的。 这里 需要 遍历  $u$  的。

。。不是, 看下面的, 说的是  $S$  递减。。 而且上面说了, 是 还需要 经过的 最短距离。

更新策略: 由状态转移方程式可知, 所有的  $dp[S][v]$  都由  $dp[S'][u](S' > S)$  转移而来, 故我们按照  $S'$  递减更新。

初始化:  $dp[2^n - 1][0] = 0$ : 已经全部拜访, 并且 当前位置为 0, 因此 还需要 经过路径 长度 为0。

核心代码:

```
void solve(int n)
{
    /*
    n: 需要拜访的城市数量
    road[i][j]: 城市i到城市j的距离, 若没有路, 则为INF。
    dp[i][j]: 当前到达城市节点集合为i, 且当前位于城市j, 剩余的路程长度。
    */

    // 初始化dp数组
    for(int i=0; i<1<<n; i++)
        for(int j=0; j<n; j++)
            dp[i][j] = inf;

    dp[(1<<n)-1][0] = 0;
    for(int i=(1<<n)-2; i>=0; i--)
    {
        for(int v=0; v<n; v++)
        {
            for(int u=0; u<n; u++)
            {
                // 使用移位运算和按位与判断元素是否存在于集合
                if(!(i >> u & 1))
                {
                    // 使用按位或运算模拟集合求并
                    // 由于没有路初始化为inf, 因此没有判断v和u间是否存在路。
                    dp[i][v] = min(dp[i][v], dp[i | (1 << u)][u] + road[v])
                }
            }
        }
    }
}
```

$[u]) ;$   
 $\} \quad \}$   
 $\} \quad \}$   
 $\} \quad \}$

## Travelling by Stagecoach

旅行家计划乘坐马车旅行。他所在的国家有  $m$  个城市，在城市间有若干道路相连。从某个城市沿某条道路到相邻的城市需要坐马车。坐马车需要车票，每用一张车票只可以 通过一条道路。每张车票上都记录了马的匹数，从一个城市移动到另一个城市的所需时间等于城市之间道路的长度除以马的数量。

这位旅行家一共有  $n$  张车票，第  $i$  张车票上的马的匹数是  $t_i$ 。一张车票只能使用一次，并且换乘时间可以忽略。求从城市  $a$  到城市  $b$  所需的最短时间。无法到达则输出 Impossible。

限制

$$1 \leq n \leq 8$$

$$2 \leq m \leq 30$$

$1 \leq a, b \leq m$  ( $a \neq b$ )

$$1 \leq i \leq 10$$

1 <= 道路长度 <= 100

定义  $dp[T][v]$ : 当前所剩票的集合  $T$ , 且位于城市  $v$ , 要到达  $b$  还需要的最小花费。

目标态  $dp[2^k - 1][a]$  : 从  $a$  出发, 且有题目给定的票数, 到达  $b$  的最小花费。

状态转移:  $dp[T][v] = \min\{ dp[T - t][u] + road[v][u] / t \}, t \in T, u \in v$  的邻居节点集合。

更新策略：由状态转移方程式 可知， $T$  是用  $T'$  ( $T' < T$ ) 更新的，因此按  $T'$  递增更新。

初始化.

$dP[*][h] \equiv 0$ , 已到达  $h$ , 还需要花费 0

$dp[0][x] = \inf$ ,  $x \neq b$ 时; 即如果不是终点, 且没有票, 则无法到达, 设置为  $\inf$ 。

核心代码

```
void solve(int n){
```

/\*

1代表还有该票  
0代表没有该票

$dp[i][j]$ : 剩下车票状态  $j$ , 现在在城市  $i$  到达  $b$  还需要的花费

```

*/
for(int i=0; i<=m; i++) {
    dp[0][i] = INF;
}

for(int i=0; i< (1<<n); i++)
    dp[i][b] = 0;

for(int s=1; s<1<<n; s++)
{
    for(int v=1; v<=m; v++)
    {
        for(int u=1; u<=m; u++)
        {
            if(grad[v][u] != INF)
            {
                for(int t=0; t<n; t++)
                {
                    if(s >> t & 1)
                        dp[s][v] = min(dp[s][v], dp[s & ~(1 << t)][u] +
grad[v][u] / T[t]);
                }
            }
        }
    }
}
}

```

## 1\*2 骨牌 铺砖问题

对铺好的转 进行编码:

横放: 2个格子都是1

竖放: 上面是0, 下面是 1

可以证明 编码 和 铺装方案 是一一对应的。

。 。 等于就是: 0 代表 它和下面的1 是 一块 竖放的 砖。 其他的 1都是代表 横放的 砖。

递推:

根据编码, 我们知道:

铺砖 的上一排 和 下一排 一定有 对应关系, 必须按 一定规则 才算合法;  
最后一排砖 肯定全 1。

假设我们已经知道：

倒数第二排 所有 编码对应的 铺砖方法总数

最后一排 对应 编码的 所有 倒数第二排 合法编码。

那么我们就能得到总数，即为 所有倒数第二排 合法编码 总数之和。

而要求 倒数第二排的数量，我们又需要 倒数第三排的数量 以及 对应 合法关系，因此 逐层递推。

求解对应关系：

如何 简便求解 对应的合法关系？ 考虑两排格子。对第二排的当前格子，我们有三种 铺放方式： 右铺，不铺，上铺。

这样 对下一排的 编码方式 进行 深度优先搜索， 我们就可以求出 所有的上下两排 对应 合法 编码。具体的，先将 两排 (top, down) 都初始化为 0。

1. 右铺： 则  $\text{top} = (\text{top} \ll 2) | 3$ ,  $\text{down} = (\text{down} \ll 2) | 3$
2. 上铺： 则  $\text{top} = (\text{top} \ll 1)$ ,  $\text{down} = (\text{down} \ll 1) | 1$
3. 不铺： 则  $\text{top} = (\text{top} \ll 1) | 1$ ,  $\text{down} = (\text{down} \ll 1)$

可以证明，如果不是 正好 铺完长度  $m$  的铺法，则是不合法的，反之则合法。（最后一格不能选择 右铺）

方便起见，我们人为设置 第0排 全1，并将其 数量置为 1， 因为 这样设置 第0排 符合第一排 的 不能上铺的设定。

。。第-1排。

DP 步骤

定义  $dp[i][E]$  : 第  $i$  行 编码为  $E$  时， 前  $i$  行 所有 铺砖 方法的总数  
目标态 :  $dp[n][2^m - 1]$ , 最后一行 全 1 的铺法总数

状态转移:  $dp[i][Edown] = \sum \{ dp[i-1][Etop] , Etop \text{ 属于 } N, N \text{ 是所有 和 } Edown \text{ 合法匹配的 上一行 } \}$

更新策略: 按  $i$  从小到大更新

初态:  $dp[0][2^m - 1] = 1$ , 其余  $dp[0][*] = 0$

代码:

```
int ok_for_top_down[MAX][2];
//ok_for_top_down[i][0]表示第i个合法上下排的上
//ok_for_top_down[i][0]表示第i个合法上下排的下排

11 dp[12][MAX];
int n, m;
11 cnt;
void dfs_get_all_ok(int c, int top, int down)
{
    // 不合法
    if(c > m)
        return;
```

```

// 合法
if(c == m)
{
    ok_for_top_down[cnt][0] = top;
    ok_for_top_down[cnt][1] = down;
    cnt++;
}

// 右铺
dfs_get_all_ok(c+2, (top << 2) | 3, (down << 2) | 3);
// 上铺
dfs_get_all_ok(c+1, (top << 1), (down << 1) | 1);
// 不铺
dfs_get_all_ok(c+1, (top << 1) | 1, (down << 1));
}

int main()
{
    while(cin >> n >> m)
    {
        if(n + m == 0)
            break;
        if(n < m)
            swap(n, m);
        cnt = 0;
        dfs_get_all_ok(0, 0, 0);

        memset(dp, 0, sizeof(dp));
        dp[0][(1<<m)-1] = 1;

        for(int i=1; i<=n; i++)
        {
            for(int k=0; k<cnt; k++)
            {
                int top = ok_for_top_down[k][0];
                int down = ok_for_top_down[k][1];
                dp[i][down] += dp[i-1][top];
            }
        }
        cout << dp[n][(1<<m)-1] << endl;
    }
    return 0;
}

```

=====

=====

=====

=====

=====

=====

=====

=====