

Date: 2014-11-14

Title: T1 : 旋转数组的最小值

Published: true

Type: post

Excerpt:

待字闺中全部题解

T1 : 旋转数组的最小值

```
int minVal(int A[], int n)
{
    if(A == NULL || n <= 0 ) return -1;
    int left=0,right=n-1;
    int mid = left; //处理有序的情况
    // ** left 为左边大的递增数组的尾巴, right为右边的小的递增数组的尾部, 一直夹逼! ! ! **
    while(A[left] >= A[right])
    {
        if(left+1==right) return A[right];
        int mid=left+(right-left)/2;
        if(A[left] == A[mid] && A[mid] == A[right]) return
        *min_element(A+left,A+right+1); //无法判断, 只能遍历
        if(A[left] <= A[mid]) left=mid; //有可能是mid, 所以left=mid
        else if(A[mid] <= A[right]) right=mid;
    }
    return A[mid];
}
```

T2: 子数组的最大乘积

```
int maxProduct(int A[], int n) {
    int global = INT_MIN;
    int local_min = A[0], local_max = A[0];
    global = local_max;
    for(int i=1;i<n;++i)
    {
        int a = local_min * A[i];
        int b = local_max * A[i];
        local_min = min(a,min(b,A[i]));
        local_max = max(a,max(b,A[i]));
        global = max(global,local_max);
    }
    return global;
}
```

T3：最长01子串

给定一个数组，数组中只包含0和1。请找到一个最长的子序列，其中0和1的数量是相同的

```
string longest01(const string &s)
{
    const int n = s.size();
    if(n < 2) return "";
    vector<int> sub(n,0);
    //求部分和
    sub[0] = ((s[0]=='0')?(-1):1);
    for(int i=1;i<n;++i)
        sub[i] = sub[i-1] + ((s[i]=='0')?(-1):1);
    for(auto a: sub) cout << a << " "; cout << endl;
    string ans;
    unordered_map<int,P> ex; //key : subSum value, value: pair (
minIndex, maxIndex)
    for(int i=0;i<n;++i)
    {
        auto it = ex.find(sub[i]);
        if(it!=ex.end()) //update the minIndex and maxIndex
        {
            (*it).second.first = min((*it).second.first,i);
            (*it).second.second = max((*it).second.second,i);
        }
        else
        {
            if(sub[i] == 0) /**注意此处对于0的特殊处理**
                ex.insert(make_pair(sub[i],make_pair(-1,i)));
            else
                ex.insert(make_pair(sub[i],make_pair(i,i)));
        }
    }
    for(auto a:ex)
        cout << a.first << " " << a.second.first << " " << a.second.second
<< endl;
    for(auto a:ex)
    {
        int l =a.second.first, r = a.second.second;
        if(r-l >= ans.size()) ans = s.substr(l+1,r-l);
    }
    return ans;
}
```

来源: <<http://xuqing.gitcafe.com/2014/10/28/Longest-01-sub-string/>>

T4：最小区间

k个有序的数组，找到最小的区间范围使得这k个数组中，每个数组至少有一个数字在这个区间范围内

```
struct node //node 自己定义，不建议用pair!!!
{
    int val;
    int idx;
    int offset;
    node(int _val = 0 , int _idx = -1, int _offset = -1):val(_val),
    idx(_idx),offset(_offset){}
};

struct cmp:public binary_function<bool,const node &, const node &>
{
public:
    bool operator()(const node &lhs, const node &rhs)
    {
        return lhs.val > rhs.val;
    }
};

typedef pair<int,int> P;

P minInterval(const vector<vector<int> > &A)
{
    const int m = A.size();
    if(m < 2) return P(0,0);

    vector<int> capa;
    for(int i=0;i<m;++i) capa.push_back(A[i].size());

    priority_queue<node,vector<node> ,cmp> pq;
    int curMax = INT_MIN;
    for(int i=0; i < m;++i )
    {
        pq.push(node(A[i][0],i,0));
        curMax = max(curMax,A[i][0]);
    }
    int ans = curMax - pq.top().val;
    int l = pq.top().val, r = curMax;
    while(1)
    {
        node cur = pq.top();pq.pop();
        int val = cur.val, idx = cur.idx, off = cur.offset;
        if(curMax - val < ans)
        {

```

```
        ans = curMax-val;
        l = val;
        r = curMax;
    }
    if(off >= capa[idx]-1) return P(l,r);
    int next = A[idx][off+1];
    if(next > curMax) curMax = next;
    node tmp(next,idx,off+1);
    pq.push(tmp);
}
return P(l,r);
}
```

T5 : Magic Index & 寻找特殊数字

1. 分析：没有重复元素时，与二分搜索代码完全一致
2. 有重复元素时。需要递归求解左右两个部分。但是边界部分是可以缩小的。

```

//不含有重复元素，就是简单的二分搜索
int magicIndex(const vector<int> &A)
{
    const int n=A.size();
    if(n==0) return -1;
    int left=0,right=n-1;
    while(left <= right)
    {
        int mid=left+(right-left)/2;
        if(A[mid] == mid) return mid;
        else if (A[mid] > mid)right=mid-1;
        else left=mid+1;
    }
    return -1;
}

//含有重复元素，那么我们就需要递归的对左右进行二分搜索
int binary_search(const vector<int> &A, int left, int right)
{
    int l=-1,r=-1;
    if(left <= right)
    {
        int mid=left+(right-left)/2;
        if(A[mid]==mid) return mid;
        else if(A[mid] < mid)
        {
            l=binary_search(A,left,min(A[mid],mid-1));
            if(l != -1) return l;
            r=binary_search(A,mid+1,right);
        }
        else
        {
            l=binary_search(A,left,mid-1);
            if(l != -1) return l;
            r=binary_search(A,max(mid+1,A[mid]),right);
        }
    }
    return (l==-1)?r:l;
}

int magicIndexWithRepeat(const vector<int> &A)
{
    const int n = A.size();
    return binary_search(A,0,n-1);
}

```

T6: 数字游戏

盒子中有 n 张卡片，上面的数字分别为 k_1, k_2, \dots, k_n 。你有4次机会，每抽一次，记录下卡片上的数字，再将卡片放回盒子中。如果4个数字的和等于 m 。则你就赢得游戏，否则就是输。直觉上，赢的可能性太低了。请你给出程序，判断是否有赢的可能性。尽量提高方法的效率。

最坏时间复杂度 $O(n^4) \implies O(n^3 \log n) \implies O(n^2 \log n)$

```
bool bi_search(const vector<int> &A, const int target)
{
    int l = 0, r = A.size()-1;
    while(l <= r)
    {
        int mid = l + (r-l)/2;
        if(A[mid] == target) return true;
        if(A[mid] > target) r = mid-1;
        else l = mid+1;
    }
    return false;
}

bool add4Number(const vector<int> & A, const int sum)
{
    const int n = A.size();
    vector<int> twoSum(n*n, 0);
    int i = 0;
    for(auto a: A)
        for(auto b: A)
            twoSum[i++] = a + b;
    sort(twoSum.begin(), twoSum.end());
    for(auto a: twoSum)
        if(bi_search(twoSum, sum-a)) return true;
    return false;
}
```

T7: 24Game

给你一个包含整数 $1 \dots n$ 的集合 S 。接下来进行 $n-1$ 次操作，每次操作从集合 S 中选取两个数，在加、减、乘三种运算中选取一种，将结果放回再集合 S 。在 $n-1$ 次操作完成后，集合 S 中只剩下一个数。求问

一种取数和运算策略，使最后的结果为24。

假设 $n = k$ ，并且有 $f(k)$ ，使得 $f(k) \Rightarrow 24$ 。那么，易得，对于 $f(k + 2) \Rightarrow f(k) * ((k + 2) - (k + 1)) \Rightarrow 24$ 。

$((k + 2) - (k + 1)) \Rightarrow 24$ 。

此时，我们只需要找到 $f(k) \Rightarrow 24$ ，就可以证明所有 $f(k + 2x) \Rightarrow 24$ 。聪明的同学们必然可以手动算出 $f(4) = 24$ 的过程，这可以证明对于所有 $k \geq 4$ 的偶数，总有 $f(k) \Rightarrow 24$ 。 $f(5) \Rightarrow 24$ 略微复杂一点，但是也不在话下。这样我们的结论已经扩展到了 $k \geq 4$ 的所有整数。又由于在 $k = [1, 2, 3]$ 时， $f(k)$ 必须不能推出24。我们的结论已经推广到了所有正整数。此题得证。

T8：（归纳法的应用） 已知一个最大堆的中序遍历序列，要求恢复该最大堆。

比如现在已知数列 $A[1...k]$ 的树已经形成了，现在加入新的节点

$A[k+1]$ 。分为两种情况：

- 如果 $A[k+1] < A[k]$ ，那么很显然， $A[k+1]$ 是 $A[k]$ 的右孩子。毫无疑问。这样既能保证中序遍历的相邻，又能保证树是最大堆。
- 如果 $A[k+1] > A[k]$ ，那么由于中序遍历序列的先后关系，那么 $A[k]$ 一定是 $A[k+1]$ 的左子树中的节点，那么 $A[k+1]$ 的左孩子就是 $A[k]$ 的最大的不超过 $A[k+1]$ 的节点。（为了实现这样的遍历，利用栈来保存 $A[k]$ 节点的祖先节点们，显然祖先节点是有序的）。

解析： 栈内保存的是 $A[k]$ 节点的祖先们

- 在插入 $A[k+1]$ 节点的时候，如果 $A[k+1]$ 是小于 $A[k]$ 的，直接加入成为 $A[k]$ 的右孩子，此时的栈顶就是 $A[k]$ ，并且 $A[k]$ 不会被弹出，再压入 $A[k+1]$ 。此时的栈就是 $A[k+1]$ 的祖先节点们。
- 如果 $A[k+1]$ 是小于 $A[k]$ 的，那么就需要找到 $A[k]$ 的祖先中不大于 $A[k+1]$ 的最大的节点，这些候选的节点都在栈内了，直接弹出，直到栈为空，或者是栈顶元素大于 $A[k+1]$ 。
- 如果栈为空，那么说明此时的已有的子树应该全部是 $A[k+1]$ 的左子树，并且更新root节点即可。
- 如果此时的栈顶是大于 $A[k+1]$ 的，那么说此时栈顶节点的右子树要退化为 $A[k+1]$ 节点的左子树（因为访问的先后关系），那么更新 $A[k+1]$ 的左子树，并且把以 $A[k+1]$ 为根的子树作为栈顶元素的右子树。 **

```

TreeNode* buildMaxHeap(const vector<int> &A)
{
    const int n = A.size();
    if(n == 0) return nullptr;
    TreeNode *root = nullptr;
    stack<TreeNode *> s;
    bool ins = false;
    TreeNode *ptr = nullptr;
    for(int i=0;i<n;++i)
    {
        ptr = nullptr;
        ins = false;
        TreeNode *now = new TreeNode(A[i]);

        while(!s.empty())
        {
            ptr = s.top();
            if(ptr->val > A[i])
            {
                now->left = ptr->right;
                ptr->right = now;
                s.push(now);
                ins = true;
                break;
            }
            s.pop();
        }
        if(ins) continue; //add node end....
        now->left = ptr;
        s.push(now);
        root = now;
    }
    return root;
}

```

T9: 统计数组

给定数组A，大小为n，数组元素为1到n的数字，不过有的数字出现了多次，有的数字没有出现。请给出算法和程序，统计哪些数字没有出现，哪些数字出现了多少次。能够在 $O(n)$ 的时间复杂度， $O(1)$ 的空间复杂度要求下完成么？

遍历数组3次的做法：（利用除数记录数组原始的值，利用余数来记录次

数)

1. $A[i] = A[i] * n;$
2. $A[A[i]/n]++$
3. $A[i] = A[i] \% n;$

注意处理数组元素完全一样的情况，这样的情况上面的方法是处理不了的，如果所有的A中的最后的元素都是0，那么就说明所有的n个元素是完全一样的，那么就直接将 $A[A[0]] = n$ ，返回即可

遍历数组两次的方法：（利用余数记录数组原始的值，利用除数来记录次

数)

1. $A[A[i]\%n] += n;$
2. $A[i] = A[i] / n;$

//重点解析一下这个代码：我们需要关注的问题是：

1. 数组的元素的范围是1 .. n, 那么就需要出现A[n]来记录元素n出现的次数, 因此首先给A.push一个元素0 (表示计数, 初值当然是0)。
2. 数组的元素在乘以n的时候可能会出现越界, 这也是需要考虑的, 下面的代码是假设不越界的情况。

```
void count(vector<int> &A) // 数组A最终存放的就是每个下标对应的数字的出现次数
{
    const int n = A.size();
    A.push_back(0); //这个位置用来记录n出现的个数。
    for(int i=0;i<n;++i)
        if(A[i] > n || A[i] <= 0)
            cout << "Illegal input..." << endl;
    bool label = false; // label 就是用来标记所有的数都是一样的情况
    for(int i=0;i<n;++i)A[i] *= n;
    for(int i=0;i<n;++i)A[A[i]/n]++;
    for(int i=0;i<n;++i){ A[i] = A[i]%n; if(A[i]) label = true; }
}
```

// 这里的余数表示元素本身的值, 显然余数的值应该小于n, 于是现将所有的元素值-1. 然后再计算每个元素出现的次数, 最后再相应的将所有的元素右移一个位置, 那么对应位置i的出现次数就是A[i]的值, A[0] 最后赋成0即可。

```
void count2(vector<int> &A)
{
    const int n = A.size();
    A.push_back(0);
    for(int i=0;i<n;++i)
        if(A[i] > n || A[i] <= 0)
            cout << "Illegal input..." << endl;

    for(int i=0;i<n;++i) //comput 0 -- n-1
        A[i]-=1;

    for(int i=0;i<n;++i) A[A[i]%n] += n;
    for(int i=0;i<n;++i) A[i] = A[i]/n;
    int tmp = A[n];
    for(int i=n;i>0;--i)A[i] = A[i-1];
    A[0] = tmp;
```

T10: 子串排列

给定两个字符串A和B, 判断A中是否包含由B中字符重新排列成的新字符串。例如: A=abcdef, B=ba, 结果应该返回true。因为ba的排列ab, 是A的子串。

依次检查字符串A的所有的长度为Len(B)的子串，然后比较排序两者是否是变位词即可。

```
bool occur(string s, string t)
{
    const int m = s.size(), n = t.size();
    if(n > m) return false;
    string tmp;
    sort(t.begin(),t.end());

    for(int i=0;i <= m-n+1; ++i)
    {
        tmp = s.substr(i,n);
        sort(tmp.begin(),tmp.end());
        if(tmp == t) return true;
        auto it = find(tmp.begin(),tmp.end(),s[i]);
        if(i+n >= m ) break; //这里必须考虑越界的问题
        *it = s[i+n];
    }
    return false;
}
```

T11：树的高度

有一个棵树，不一定是二叉树，有n个节点，编号为0到n-1。有一个数组A，数组的索引为0到n-1，数组的值A[i]表示节点i的父节点的id，根节点的父节点id为-1。给定数组A，求得树的高度。

树的问题，一般使用递归来求解，但是，存在着大量的重复运算，那么很简单，用一个备忘录记录下已经计算了的节点的高度。

```

int helper(const vector<int> &A, const int idx, vector<int> &height)
{
    if(height[idx] != -1 )return height[idx];
    if(A[idx] == -1) {height[idx] = 1; return 1;}
    height[idx] = 1 + helper(A,A[idx], height);
    return height[idx];
}

vector<int> printHeight(const vector<int> &A)
{
    const int n = A.size();
    vector<int> height(n,-1);
    for(int i = 0 ; i < n ; ++i)
        helper(A,i,height);
    return height;
}

```

T12: 123排序（荷兰国旗问题）

快速排序的partition的思想，注意循环不变式的定义即可。

```

void sort123(int *A, const int n)
{
    if(A == nullptr || n < 2) return;
    int p1=0,p2=n-1;
    int k = 0;
    while(k <= p2)
    {
        if(A[k] == 1) {swap(A[k++],A[p1++]);}
        else if(A[k] == 3){swap(A[k],A[p2--]);}
        else ++k;
    }
}

```

T13: 删除字符串

删除字符串中的“b”和“ac”，需要满足如下的条件：字符串只能遍历一次并且不能够使用额外的空间。

涉及到这些字符串处理的问题，其实都是状态机的问题，至于状态机，自己不熟。那就直接按照题目的要求来编写代码。

如果当前的字符是b，那么删除即可，如果当前的字符是'a',那么还要考虑下面的字符是不是c，如果是c，删除即可，如果不是，那么保留a，依次循环往复即可。

//代码有BUG!! 自己写的，下面贴的是对的

```
//删除'b' 和 "ac"
void deleteChars_ans(char *s)
{
    cout << "s : " << s << endl;

    if(s == nullptr) return;
    int i = 0 ; //下一个合法字符已经存放的位置
    int PREV = false; // 前一个字符是A的情况
    int k = 0;
    for(k = 0 ; s[k] != '\0'; ++k)
    {
        cout << s[k] << endl;
        if(PREV == false && s[k] != 'a' && s[k] != 'b')
        {
            s[i++] = s[k];
        }
        if(PREV && s[k] != 'c')
        {
            s[i++] = 'a';
            if(s[k] != 'a' && s[k] != 'b')
            {
                s[i++] = s[k];
            }
        }
        PREV = (s[k] == 'a');
        if(i > 1 && s[i-2] == 'a' && s[i-1] == 'c')
            i-=2;
    }

    if(PREV) // 说明还有最后一个'a'需要写到合理的位置
        s[i++] = 'a';
    s[i] = '\0';
}
```

T14: 最小公倍数

给定 n, 求最小的整数 M, 使得它能够被 1-n 中的所有数整除。 其实

就是求 1-n 的最小公倍数

也就是将每一个数质因子分解，然后求所有的质因子的最高次幂的乘积即可，例如：

1 2 3 4 5 6 == > 1 2 3 2^2, 5, 2*3 ==>那么分别提取这些质因子的最高次幂为：2^2 * 3 * 5 = 60 .

同样需要注意的是溢出的问题。

```
int minMultiply(const int n)
{
    if(n <= 2) return n;
    vector<int> vec(n,0);
    for(int i=0;i<n;++i) vec[i] = i + 1; //generate the [1...n]
    int ret = 1;
    for(int i=1;i<n;++i) //vec[0] = 1, we can ignore
    {
        for(int j = i+1; j < n ; ++j)
        {
            if(vec[j] % vec[i] == 0) vec[j] /= vec[i];
        }
        ret *= vec[i];
    }
    return ret;
}
```

T15: 最长回文

直接上manacher算法吧，其他的算法见博客
<http://blog.csdn.net/xuqingict/article/details/39718893>

```

string Manacher(const string &s)
{
    cout << "enter..." << endl;
    const int n = s.size();
    string tmp;
    tmp += '$';
    for(auto a : s){tmp += '#'; tmp += a;}
    tmp += '#';
    cout << tmp << endl;
    const int m = tmp.size();
    vector<int> P(m,0);
    int idx = -1, right = -1;
    for(int i=1;i<m;++i)
    {
        P[i] = (i < right)?(min(P[2*idx-i],right-i)):1;
        while(tmp[i+P[i]] == tmp[i-P[i]]) P[i]++;
        if(i + P[i] > right)
        {
            right = i + P[i];
            idx = i;
        }
    }
    auto pos = max_element(P.begin(),P.end());
    int len = *pos-1;
    string ret;
    int i = pos-P.begin();
    ret += tmp[i];
    cout << "middle : " << ret << endl;
    int k = 1;
    //construct the palindrome
    while(len)
    {
        ret += tmp[i+k];
        ret = tmp[i-k] + ret;
        --len;
        ++k;
    }
    //trim '#'
    k = 0;
    for(auto a: ret)
        if(a != '#') ret[k++] = a;
    ret.resize(k);
    return ret;
}

```

T16 : 丢失的数字

给定一个无序的整数数组，怎么找到第一个大于0，并且不在此数组

的整数。比如[1,2,0] 返回 3, [3,4,-1,1] 返回 2。最好能O(1)空间和O(n)时间

```
//A[i] 保存的是 (i+1) 的值!!
int firstMissingDigit(vector<int> &A) {
    const int n = A.size();
    for(int i=0;i<n;++i)
    {
        while(A[i] != (i+1))
        {
            //如果A[i]的范围不是[1,n-1]内部的话，那么不能直接操作!!!
            if(A[i] < 1 || A[i] >= n || A[i] == A[A[i]-1]) break;
            swap(A[i],A[A[i]-1]);
        }
    }
    for(int i=0;i<n;++i)if(A[i] != (i+1)) return (i+1);
    return n+1;
}
```

T17：给定字符串，输出括号是否匹配，例如，

```
"()" yes;
")(" no;
"(abcd(e)" no;
"(a)(b)" yes。
```

题目描述很简单，要求使用递归来写。不能出现循环语句。


```

bool valid(const string &s)
{
    if(s.empty()) return true;
    const int n = s.size();
    if(n < 2) return false;
    stack<char> left;
    for(auto a : s)
    {
        if(a != '(' && a != ')') continue;
        if(a == '(') left.push(a);
        else
        {
            if(left.empty()) return false;
            left.pop();
        }
    }
    return left.empty();
}

bool helper(const string &s, const int idx, const int max_index, int
&left)
{
    if(s[idx] != '(' && s[idx] != ')') return
helper(s,idx+1,max_index,left);
    if(idx > max_index) return left == 0;
    if(s[idx] == '(')
    {
        ++left;
    }
    else
    {
        if(left == 0) return false;
        --left;
    }
    return helper(s,idx+1,max_index,left);
}

bool valid_2(const string &s)
{
    const int n = s.size();
    int left = 0;
    bool ret = helper(s,0,n-1,left);
    return ret;
}

```

**T18 : 对一个字符串按照回文进行分割，例如
aba|b|bbabb|a|b|aba就是字符串**

ababbbabbababa的一个回文分割，每一个字串都是一个回文。请找到可以分割的最少的字串数

```
int minCut(string s) {
    const int n = s.size();
    if(n < 2) return 0; //if length < 2. return 0 directly.
    vector<vector<bool> > f(n,vector<bool>(n,false)); //f[i...j]
means whether the s[i...j] is palindrome or not.
    for(int i=n-1;i>=0;--i)
    {
        f[i][i] = true;
        for(int j=i+1;j<n;++j)
        {
            if(i+1 == j) f[i][j] = (s[i] == s[j]);
            else f[i][j] = (f[i+1][j-1] && s[i] == s[j]);
        }
    }

    vector<int> g(n+1,0); //g[i] means the minimum cut number of
partitioning s[0...i-1].
    g[0] = -1;
    //previous i chars
    for(int i=2;i<n+1;++i)
    {
        g[i] = i-1;
        for(int j=0;j<i;++j) // j is also length.
            if(f[j][i-1]) g[i] = min(g[i],g[j]+1);
    }
    return g[n];
}
```

T19：数字翻译

翻译数字串，类似于电话号码翻译：给一个数字串，比如12259，映射到字母数组，比如，1 -> a， 2-> b， ... ， 12 -> l， ... 26-> z。那么， 12259 -> lyi 或 abbei 或 lbei 或 abyj。输入一个数字串，判断是否能转换成字符串，如果能，则打印所有可能的转换成的字符串。动手写写吧。

```
char dict[27];
```

```

ostream_iterator<char> out(cout, "\\t");

//can be translated or not
bool translated(const string &s)
{
    const int n = s.size();
    if(n == 0 ) return true;
    vector<bool> f(n+1, false);
    f[0] = true;
    f[1] = (s[0] != '0');
    for(int i=2; i<=n; ++i) // length
    {
        int tmp = atoi(s.substr(i-2, 2).c_str());
        f[i] = (f[i-1] && s[i-1] != '0') || (f[i-2] && (tmp >= 10 && tmp
<= 26));
    }
    return f[n];
}

//the number of decode ways
bool valid(const string &s)
{
    int t = atoi(s.c_str());
    return (s[0] != '0') && (t > 0 && t <= 26);
}

//count the number of decode ways
int numDecodings(string s) {
    const int n = s.size();
    if(n == 0) return 0;
    vector<int> f(n+1, 0);
    f[0] = 1;
    f[1] = (s[0] != '0');
    for(int i=2; i<=n; ++i)
    {
        if(s[i-1] != '0') f[i] = f[i-1];
        if(valid(s.substr(i-2, 2))) f[i] += f[i-2];
    }
    return f[n];
}

//dfs to print all result
void helper( const string &s, int cur_index, const int max_index, string
cur, vector<string> &ret)
{
    if(cur_index > max_index) {ret.push_back(cur); return ;}
    //one digit
    if(s[cur_index] == '0') return; //illegal
    cur += dict[s[cur_index] - '0'];
    helper(s, cur_index+1, max_index, cur, ret);
    cur.pop_back();
    //two digit
    int tmp = atoi(s.substr(cur_index, 2).c_str());

```

```

        if(tmp >= 10 && tmp <= 26)
        {
            cur += dict[tmp];
            helper(s,cur_index+2,max_index,cur,ret);
        }
    }
    vector<string> helper(const string &s)
    {
        const int n = s.size();
        if(n == 0 ) return vector<string>();
        vector<string> ret;
        string cur;
        helper(s,0,n-1,cur,ret);
        return ret;
    }

    void print(const string &s , int cur_index, string &cur, vector<string>
    &ret, const vector<vector<int> > parent)
    {
        if(cur_index == 0)
        {reverse(cur.begin(),cur.end());ret.push_back(cur);return;}
        string tmp = cur;
        for(auto idx : parent[cur_index])
        {
            cur = tmp;
            //cur = s.substr(idx,cur_index-idx) + cur;
            int t = atoi(s.substr(idx,cur_index-idx).c_str());
            cur += dict[t];
            print(s,idx,cur,ret,parent);
        }
    }

    //DP to record the parent of each position(at least two parent...)
    vector<string> printAll(const string &s)
    {
        const int n = s.size();
        if(n == 0) return vector<string>();
        vector<vector<int> > f(n+1,vector<int>()); //record the parent of
each solution
        //f[i] : the last index of s[i-1]
        vector<bool> g(n+1,false);
        g[0] = true;
        g[1] = (s[0] != '0');
        f[1].push_back(0);
        for(int i=2;i<=n;++i)
        {
            //one digit
            if(g[i-1] && s[i-1] != '0')
            {
                g[i] = true;
                f[i].push_back(i-1);
            }
        }
    }

```

```

    }

    //two digit
    int tmp = atoi(s.substr(i-2,2).c_str());
    if( g[i-2] && (tmp >= 10 && tmp <= 26))
    {
        g[i] = true;
        f[i].push_back(i-2);
    }
}
for(int i=0;i<f.size();++i)
{
    for(auto a: f[i])
        cout << a << " ";
    cout << endl;
}
vector<string> ret;
string cur;
print(s,n,cur,ret,f);
return ret;
}

```

T20：糖果问题

Description : N个孩子站成一排，每个人分给一个权重。按照如下
的规则分配糖果：

每个孩子至少有一个糖果
所分配权重较高的孩子，会比他的邻居获得更多的糖果
问题是，最少需要多少个糖果？

权重的分布一定是一个折线（直线是特殊的折线），那么波谷的孩子分得的糖果一定是1，他旁边比他高的依次加1即可。但是需要考虑到来自左右两边的约束。那么扫描数组两边，因为每一个孩子分得的糖果数是取决于他的左右两个孩子的。

```

int minCandy(const vector<int> &A)
{
    const int n = A.size();
    if(n < 2) return n;
    vector<int> l(n,0),r(n,0);
    l[0]=1;
    for(int i=1;i<n;++i)
    {
        if(A[i]>A[i-1]) l[i] = l[i-1]+1;
        else l[i]=1;
    }
    r[n-1]=1;
    for(int i=n-2;i>=0;--i)
    {
        if(A[i] > A[i+1])r[i] = r[i+1]+1;
        else r[i]=1;
    }

    int ret=0;
    for(int i=0;i<n;++i)ret+=max(l[i],r[i]);
    return ret;
}

```

T21: 分词问题

给定字符串，以及一个字典，判断字符串是否能够拆分为字典中的单词。例如，字段为{hello, world}，字符串为hellohelloworld，则可以拆分为hello,hello,world，都是字典中的单词
DP求解即可。

```

bool dictPartition(const string &s,const unordered_set<string>
&dict)
{
    const int n = s.size();
    if(n == 0) return false;
    vector<bool> f(n+1,false);
    f[0]=true;
    for(int i=1; i<=n;++i) //len
    {
        for(int j=0;j<i;++j)
        {
            auto it = dict.find(s.substr(j,i-j));
            if(f[j] && it != dict.end()) {f[i]=true;break;}
        }
    }
    return f[n];
}

```

T22: LIS 最大独立子集

含义如下：给定一棵二叉树，找到满足如下条件的最大节点集合：

集合中的任意两个节点之间，都没有边。

DP，跟最大娱乐值的那题很像。定义递推式： $f[x]$ ：以 x 节点为根节点的子树的最大独立集（包含 x ）。 $g[x]$ ：以 x 为根节点的子树的最大独立集（不包含 x ）。

注意题目中的最大独立集应该指的是独立集的元素的和，而不是个数，如果是个数的话，那么也是可以解的，每个节点的值是1就对了。

```

struct fg
{
    int f;
    int g;
    fg(int _f=0,int _g=0):f(_f),g(_g){}
}; // f : include x, g: exclude x

void helper(TreeNode *root, unordered_map<TreeNode *,fg> & exist, int
&cnt)
{
    if(root == nullptr || exist.find(root) != exist.end()) return;
    if(exist.find(root->left) == exist.end())
        helper(root->left,exist,cnt);
    if(exist.find(root->right) == exist.end())
        helper(root->right,exist,cnt);
    //contain x
    assert(exist.find(root->left) != exist.end());
    assert(exist.find(root->right) != exist.end());
    fg l = exist[root->left];
    fg r = exist[root->right];
    int f = root->val + l.g + r.g;
    int g = max(l.f,l.g) + max(r.f,r.g);
    if(f > g) ++cnt;
    exist.insert(make_pair(root,fg(f,g)));
}

int maxLIS(TreeNode *root)
{
    if(root == nullptr) return 0;
    unordered_map<TreeNode*,fg> exist;
    int cnt = 0; //record the size of LIS set
    exist.insert(make_pair(nullptr,fg(0,0)));
    helper(root,exist,cnt);
    int ret = max(exist[root].f, exist[root].g);
    cout << "size : " << cnt << endl;
    return ret;
}

```

T23: 拷贝链表

有一个链表，每一个节点除了next指针指向一下节点以外，又多出了一个指针random，指向链表中的任何一个节点，包括null。请给出方法完成链表的深拷贝。


```

ListNode *copyList(ListNode *head)
{
    if(head == nullptr) return nullptr;
    //copy node and insert
    ListNode *tmp = head;
    while(tmp)
    {
        ListNode *node = new ListNode(tmp->val);
        node->next = tmp->next;
        tmp->next = node;
        tmp = node->next;
    }
    //copy random node
    tmp = head;
    while(tmp)
    {
        if(tmp->random) tmp->next->random = tmp->random->next;
        tmp = tmp->next->next;
    }
    //split
    ListNode dummy(0);
    ListNode *new_head = &dummy;
    tmp = head;
    while(tmp)
    {
        new_head->next = tmp->next;
        tmp->next = tmp->next->next;
        new_head = new_head->next;
        new_head->next = nullptr;
        tmp = tmp->next;
    }
    return dummy.next;
}

```

T24：城市的环形路有n个加油站，第i个加油站的油量用gas[i]来表示，你有如下的一辆车：

它的油缸是无限量的，初始是空的
 它从第i个加油站到第i+1个加油站消耗油量为cost[i]
 现在你可以从任意加油站开始，路过加油站可以不断的加油，问是否能够走完环形路。如果可以返回开始加油站的编号，如果不可以返回-1。注意，解决方案保证是唯一的。

Observation:

如果说从某一个点*i*出发，在点*k*处出现了油不足的情况，那么在 $[i, k]$ 之间的所有的点都是不满足条件的。

如果说消耗的油的总量大于加油站的油的储量，那么很显然不存在这样的点；下面这句很重要，如果说油的储量大于油的消耗量，那么一定存在这样的点。（这句需要好好理解，可用反证法来证明）。

```
int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
    const int n = gas.size();
    assert(n == cost.size());
    int global = 0 ; //record the diff between cost and gas
    int start = 0 ; // start index
    int cur = 0 ; //current gas
    for(int i = 0; i<n; ++i)
    {
        cur += (gas[i]-cost[i]);
        global += (gas[i] - cost[i]);
        if(cur < 0)
        {
            start = i+1;
            cur = 0;
        }
    }
    return (global<0)?(-1):(start);
}
```

T25 : 最大乘积

Description : 一根绳子，长度为*n*米。将其切成几段，每一段的长度都是整数。请给出一种切法，使得切成的各段绳子之间的乘积是最大的。注意，最少要切一下的。

```

int maxCutLen(const int n)
{
    if(n <= 0 )return 0;
    if(n == 1) return 0; //illegal input
    vector<int> f(n+1,0); //f[i]: 长度为i的绳子至少切一刀的最长的长度。
    f[0] = 0;
    f[1] = 1;
    f[2] = 1;
    for(int i=3;i<=n;++i) //length
        for(int j=1;j<=i/2;++j) // enumerate the length of the first
            segment
                f[i] = max(f[i],max(j*f[i-j], (i-j)*j)); // (j-i)*j :
cut once
    return f[n];
}

```

只需要将长度 n 表示为 $3x+2y=n$ ，并且 3 尽可能的多，这样的 3^x+2^y 是最大的。不得不赞叹，这确实是一个很巧妙的方法。大家可以通过例子，验证几个。为什么只有 3 和 2 呢？长度为 4 的，就是 2×2 ， 5 以上的，都可以分解为 $3x+2y$ ，并且 $3^x+2^y > 5$ 以上的数字。这个题目要求是整数，如果取消这个限制呢？

T26: T25的拓展，最大字符串乘积

输入数字组成的字符串，取 k 切分后的最大 K 乘积

例如：如“123”切为2分，则两项最大乘积是 $12 * 3 = 36$

T27: 最少插入字符

给定字符串，可以通过插入字符，使其变为回文。求最少插入字符的数量

```

int minInsert(const string &s)
{
    const int n = s.size();
    if(n < 2) return 0;
    vector<int> prev(n,0), next(n,0);
    for(int i=n-1;i>=0;--i)
    {
        for(int j=i+1;j<n;++j)
        {
            next[j] = min(prev[j],next[j-1]) + 1;
            if(s[i] == s[j]) { next[j] = min(next[j],prev[j-1]);}
        }
        prev.swap(next);
    }
    return prev[n-1];
}

```

T28：数对数目

给定两个数组X和Y，元素都是正数。请找出满足如下条件的数对的数目。

$x^y > y^x$ ，即x的y次方>y的x次方

x来自X数组，y来自Y数组

对于这样的指数的运算，我们首先想到的是化简，那么取log是化简指数运算的利器。

两边同时取对数有：对于式子： $y \log(x) > x \log(y)$ ，x和y都是正数，则进一步有：两边同时除以xy，则： $\log(x)/x > \log(y)/y$ 。这个式子，看起来也复杂，但是，x和y都在各自的一边，要简单的多。

Algorithm:

+ 数组X与Y分别计算 $\log x/x$ 和 $\log y/y$ 。然后排序，再对于每一个x，在Y中进行二分查找，从而得到比该数字小的位置，然后输出即可

```
int numPair(const vector<int> &A, const vector<int> &B)
{
    const int m = A.size(), n = B.size();
    if (m == 0 || n == 0) return 0;
    vector<double> tA (m,0), tB(n,0);
    for(int i=0;i<m;++i)
        tA[i] = log(A[i])/A[i];
    for(int i=0;i<n;++i)
        tB[i] = log(B[i])/B[i];
    sort(tA.begin(),tA.end());
    sort(tB.begin(),tB.end());

    //count
    int ret = 0;
    for(auto a: tA)
    {
        auto it = upper_bound(tB.begin(),tB.end(),a);
        ret += (it-tB.begin());
    }
    return ret;
}
```

补充： lower_bound 与 upper_bound的代码：

```

        template<class ForwardIterator, typename T>
        ForwardIterator my_lower_bound(ForwardIterator first,
        ForwardIterator last, T target)
        {
            while(first != last)
            {
                ForwardIterator mid = first + (last-first)/2;
                if(target > *mid) first = mid+1;
                else last=mid;
            }
            return first;
        }

        template<class ForwardIterator , typename T>
        ForwardIterator my_upper_bound(ForwardIterator first,
        ForwardIterator last, T target)
        {
            while(first != last)
            {
                ForwardIterator mid=first + (last-first)/2;
                if(target >= *mid) first=mid+1;
                else last=mid;
            }
            return first;
        }

```

T29：谁多谁少

盒子A有10个红球，盒子B有10个绿球。进行如下的操作：

- + 随机从A中拿三个球放入B中
- + 随机从B中拿三个球放入A中

问题是，在哪一个盒子中，会出现一个颜色的球比另一个颜色的球更多？该如何分析？

首先这个题目的目标大家理解了么？这里有个小坑。两个盒子，肯定都是一种颜色的球比另一种颜色的球多，但是哪个更多呢？关键在这个“更”字，直接上，感觉是B的同学，也是对“更”字的关注不够。

仔细、逐步分析，让我们看看实际是什么样的：第一步，从A中，随机选择三个红球放入B中，这里没有什么变数。看第二步：从B中，取三个球，放入A中。那么有多少个红球被拿到A中呢？

所以，最终的答案是一样最多的，**没有哪个盒子里是更多的**。

B中拿出红球个数	A中情况	B中情况	哪个更多
0	7红球, 3绿球	7绿球, 3红球	一样多
1	8红球, 2绿球	8绿球, 2红球	一样多
2	9红球, 1绿球	9绿球, 1红球	一样多
3	10红球, 0绿球	10绿球, 0红球	一样多

T30：选择旅游国家

有一个待选国家的列表，以及国家的相对热门程度，请给出一个算法，随机选择一个国家，并且保证，越是热门的国家，随机选择它的可能性就越高。

每当我们遇到一个题目的时候，都要对题目进行充分的理解，有哪些条件，目标是什么。这个题目看完之后，我们能够得到两个要点：

随机选择一个国家

越是热门，选择的可能性、概率就越高

我们怎么做到这个呢？如何充分理解这个呢？还是通过一个例子来进行：

```
| 国家 |      A | B | C | D |
| --- | --- | --- | --- | --- |
| 热度 | 1 | 2 | 5 | 2 |
```

随机选择一个国家，意味着，如果热度相同，则被选择的概率是相同的，更进一步，都可以表示为 $2/10$ 。依次类推，A被选择的概率是 $1/10$ ，热度最小，则概率最小。并且，概率之间的比，和热度之间的比是相同的。

那么，我们以怎么样的方法，来保证，选择A的概率是 $1/10$ ，B和D的概率是 $2/10$ ，C的概率是 $5/10$ ？我们稍作变换，将上面的表格，转换为如下的表格：

```
| A | B | B | C | C | C | C | C | C | D | D |
| - | - | - | - | - | - | - | - | - | - | - |
```

我们只要保证，选择上面表格中每一个元素的概率是相同的，就可以得到A，B，C，D的概率值。如何保证呢？两种情况：

- + 当国家数以及热度都是固定时，比如上面的总数10，随机0-9的数字，即可。
- + 当国家数以及热度都是不固定时，则需要蓄水池抽样算法。

T30：色子问题

n个色子，每个色子m面，每一面的值分别是1-m。你将n个色子同时抛，落地后将所有朝上面的数字加起来，记为sum。给定一个数字x，如果 $\text{sum} > x$ ，则你赢。给定n，m，x，求你赢的概率。

```
+ 1 <= n <= 100
+ 1 <= m <= 10
+ m <= x < n * x
```

这种题目一般都是产生式模型，其实也就是时间换空间的做法。

```
void helper(const int n, const int m, int cur_index, int sum,
vector<int> &cnt)
{
    if (cur_index > n)
    {
```



```

        cnt[sum]++;
        return;
    }
    int tmp = sum;
    for(int i=1;i<=m;++i)
    {
        sum = tmp;
        sum += i; // add current dice's index
        helper(n,m,cur_index+1,sum,cnt);
    }
}

double win2(const int n, const int m, const int x)
{
    if(x < n || x > m*n) return 0;
    vector<int> cnt(n*m+1,0);
    helper(n,m,1,0,cnt);
    int cn = accumulate(cnt.begin()+x,cnt.end(),0);
    int sum = accumulate(cnt.begin(),cnt.end(),0);
    cout << cn << " " << sum << endl;
    return static_cast<double>(cn)/sum;
}

//循环版本
double win(const int n, const int m, const int x)
{
    if(x < n || x > m*n) return 0.0; // illegal x
    vector<int> prev(n*m+1,0);
    vector<int> next(n*m+1,0);
    //init
    for(int i=1;i<=m;++i) prev[i]=1;

    for(int k = 2;k <= n ; ++k) //dice count
    {
        //update each elem in next
        vector<int>(n*m+1,0).swap(next);
        for(int i=k;i<= m*k; ++i) //the updated index is in a range
        {
            //add prev m elems
            for(int j=1; j<=i && j<=m;++j)
            {
                next[i] += prev[i-j];
            }
        }
        prev.swap(next);
    }

    int cn = 0;
    for(int i=x;i<=m*n+1;++i)
        cn += prev[i];
    int sum = accumulate(prev.begin(),prev.end(),0);

```

```
        return static_cast<double>(cn)/sum;
    }
```

T31: 巧妙变换（完美洗牌）

输入数组[a1,a2,...,an,b1,b2,...,bn]，构造函数，使得输出为，
[a1,b1,a2,b2,...,an,bn]。

注意：方法要是in-place的。

本问题是矩阵的原地reverse的问题。矩阵的size为 $2 * n$ 。

```
void shuffle(char *A, const int R, const int C)
{
    if(A == nullptr || R <= 0 || C <= 0 ) return ;
    if(R == 1 || C == 1 ) return;
    const int size = R*C-1;
    vector<bool> B(size+1,false);
    B[0] = B[size] = true ; //the first and last
    for(int i=1;i<size;++i)
    {
        if(B[i]) continue;
        int beg = i ;
        char tmp = A[beg];
        int next = i;
        do
        {
            next = (i*R)%size;
            swap(A[next],tmp);
            B[i] = true;
            i = next;
        }while(next != beg);
    }
    return;
}
```

T32：赛马

想必田忌赛马的故事，大家都耳熟能详。但是，大家知道Google的童鞋们是怎么赛马的么？

不过，首先，大家要先尝试一下：有25匹马，每次只能五匹一起

跑，那么最少跑几次，才能确定前三甲呢？

7次 = （分成5组，每组5匹，决出5匹马） + 5个第一名比赛一次（得1，2，3）（此时的1就是冠军了） + （2，3 + 1所在的队列的2，3名 + 2所在的队列的第二名）。

T33：巧妙排序 + 荷兰国旗

排序只有1，2，3三个元素的数组，不能统计1，2，3的个数

更加巧妙一点的方法，那就是将数组中的3个元素分别看作质数2，3，

5，...，并将这个质数乘起来，那么积的结果的2，3，5，。。。的因子的个数，依次填入之前的数字即可了。

```
void sort123(int *A, const int n)
{
    if(n < 2) return;

    int p1 = 0, p2=0, p3=n-1;
    for( ; p2 <= p3; ++p2)
    {
        if(A[p2] == 1)
        {
            swap(A[p1++], A[p2]);
        }
        else if(A[p2] == 3)
        {
            swap(A[p2], A[p3]);
            --p3;
        }
    }
}
```

T34：数组波谷

一个数组 $A[1...n]$ ，满足 $A[1] \geq A[2]$ ， $A[n] \geq A[n-1]$ 。 $A[i]$ 被成为波谷，意味着： $A[i-1] \geq A[i] \leq A[i+1]$ 。请给出一个算法，找到数组中的一个波谷。 $O(n)$ 的方法，是很直接，有更快的方法么？

既然存在更好的方法，那么必然是 $O(\log n)$ ，自然想到了用二分查找。

Obsevation: 数组的两端有 $A[1] \geq A[2]$, $A[n] \geq A[n-1]$, 也就是说整个数组是一个V字型, 那么必然存在波谷。那么每次取mid, 根据mid与其前后两个元素之间的关系, 那么我们可以递归的查找某一边!!!

```
+ if A[mid-1] >= A[mid] <= A[mid+1], 找到波谷, 返回A [mid] ;  
+ if A[mid-1] >= A[mid] >= A[mid+1] , 那么在右边肯定存在一个波谷, 二分查找又半边即可。  
+ if A[mid-1] <= A[mid] >= A[mid+1], 任选一边即可。  
+ if A[mid-1] <= A[mid] <= A[mid+1], 那么在左边一定存在波谷, 二分查找左半边即可。
```

```
int minValue(const vector<int> &A)  
{  
    const int n = A.size();  
    if(n < 2 || A[0] <= A[1] || A[n-1] <= A[n-2])  
    {  
        return -1;  
    }  
    int left=0, right=n-1;  
    while(left<=right)  
    {  
        int mid = left+(right-left)/2;  
        if(left == mid) return A[left] < A[right] ? left : right;  
        if(A[mid-1] >= A[mid] && A[mid] <= A[mid+1]) return mid;  
  
        if(A[mid-1] >= A[mid] && A[mid] >= A[mid+1]) left = mid+1;  
        else if(A[mid] >= A[mid-1] && A[mid+1] >= A[mid]) right=mid-1;  
        else left=mid+1;  
    }  
    return -1;  
}
```

T35:给定一根木棍, 长一米, 把它随机折成3段, 问最短的一段期望为多长, 最长的期望多长?

此类题目无非是求积分, 可以画图, 求积分, 即可。

这题随机变量看似3个，但是其实两个就可以了。设在X, Y处把棍子折断，X, Y都在[0,1]区间内均匀分布。

我们不妨设 $X < Y$ ，那么三段的长度分别为 $X, Y-X, 1-Y$ （米）。如果X是最短的，那么满足

$$X < Y - X$$

$$X < 1 - Y$$

期望很简单，代入联合分布，以及这些条件得：

$$\int_0^1 \int_0^{1-x} 2x \, dy \, dx$$

这里，主要是由给出的条件，分别给出X, Y的上下界。我们先对Y积分，Y的上下界由1) , 2) 很容易给出，那么X的呢？一般可画出X, Y的坐标图，以及边界条件1) 2) , 我们可知X最小为0，当1) , 2) 两条曲线相交时取到最大值1/3. 计算积分公式得出1/54. 这里我们假定了X, Y-X, 1-Y中，X最小。Y-X最小以及1-Y最小的情形类似跟这种情况对称，变化下符号就一样了。另外Y>X情况也是一样，所以，我们加总这六种情况，得出期望值为： $6 \times 1/54 = 1/9$.

计算最大的一段的价值类似，可先假定一组条件，然后根据对称性，简单求和。

我们还是先假定 $X < Y$ 然后X为其中最大段，那么

$$X > Y - X$$

$$X > 1 - Y$$

这种情况，画出图来，容易确定积分

上限界限，从而列出公式：

$$\int_0^{1/3} \int_{1-x}^1 2x \, dy \, dx + \int_{1/3}^1 \int_0^{1-x} 2x \, dy \, dx$$

等于 $1/54 + 1/12$, 同样，根据对称性，乘以6，最终得11/18.

T36 : 有两个色子，一个是正常的，六面分别1-6的数字；另一个六面都是空白的。现在有0-6的数字，请给出一个方案，将0-6中的任意数字涂在空白的色子上，使得当同时扔两个色子时，以相等的概率出现某一个数字。如果一个色子是1，另一个色子是2，则出现的数字是3。依次类推。

首先，深入理解题目。两个色子，一个色子上1到6，是正常的，可以理解为，随手一扔，每个数字出现的概率是相同的，都是1/6；另一个呢？空白的，不过我们可以自己涂上0-6的数字，包括0和6。然后扔完了之后，一个色子上面出现a，另一个色子出现b，最终把a+b作为一个数字。有多少个不同的数字呢？假设有n个，则题目要求是每一个出现的概率都是1/n。

n的取值都有那些呢？1到12都可以。加入，就是1-12的数字，该如何涂空白的色子，保证概率相等呢？两个色子，每个六面，扔起来，一共36种可能，如果出现12个数字，并且，每个数字是等概率的，则 $36/12=3$ ，每个可能会出现三次。当，第一个色子，扔得数字是1时，第二个色子要有三次是0才能保证1出现了三次。同理，当第一个色子扔的是6，要得到三次12的数字，则第二个色子要有三次是6。则，空白的色子，必须涂三个0，三个6。我们来证明，每一个概率都是 $3/36=1/12$ ：

	第一个色子	第一个色子概率	第二个色子	所得数字
1	1/6	$p(0)=p(6)=1/2$	$p(1) = p(7) = 1/2 * 1/6 = 1/12$;	
2	1/6	$p(0)=p(6)=1/2$	$p(2) = p(8) = 1/2 * 1/6 = 1/12$;	
3	1/6	$p(0)=p(6)=1/2$	$p(3) = p(9) = 1/2 * 1/6 = 1/12$;	
4	1/6	$p(0)=p(6)=1/2$	$p(4) = p(10) = 1/2 * 1/6 = 1/12$;	
5	1/6	$p(0)=p(6)=1/2$	$p(5) = p(11) = 1/2 * 1/6 = 1/12$;	
6	1/6	$p(0)=p(6)=1/2$	$p(6) = p(12) = 1/2 * 1/6 = 1/12$;	

则， $p(1)=p(2)=p(3)=p(4)=p(5)=p(6)=p(7)=p(8)=p(9)=p(10)=p(11)=p(12)=1/12$

最后的取值范围，还可以是其他的么？我们已经知道正常的色子，哪一面出现的概率都是1/6；能不能充分利用这个呢？只需要空白的色子，六面都是一个数字就可以了。 $p = 1/6 * 1$ 最终每个数字出现的概率都是1/6。

T37：给定一个数组，我们可以找到两个不相交的、并且是连续子数组A和B，A中的数字和为sum(A)，B中的元素和为sum(B)。找到这样的A和B，满足sum(A) - sum(B)的绝对值是最大的。例如：[2, -1 -2, 1, -4, 2, 8]划分为A=[-1, -2, 1, -4]，B=[2, 8]，最大的值为16

```

    void minSub(const vector<int> &A, vector<int> &minL, vector<int>
&maxL, vector<int> &minR, vector<int> &maxR)
    {
        const int n = A.size();

        minL[0] = A[0];
        maxL[0] = A[0];

        //get the minL and maxL first
        int min_tmp = A[0], max_tmp = A[0];
        for(int i=1;i<n;++i)
        {
            //A[0...i];
            if(min_tmp > 0 ) min_tmp = 0;
            min_tmp += A[i];
            minL[i] = min(minL[i-1],min_tmp);
            if(max_tmp < 0 ) max_tmp = 0;
            max_tmp += A[i];
            maxL[i] = max(maxL[i-1],max_tmp);
        }
        //get the minR and maxR
        minR[n-1]=A[n-1],maxR[n-1]=A[n-1];
        min_tmp = A[n-1], max_tmp = A[n-1];
        for(int i=n-2;i>=0;--i)
        {
            //A[0...i];
            if(min_tmp > 0 ) min_tmp = 0;
            min_tmp += A[i];
            minR[i] = min(minR[i+1],min_tmp);
            if(max_tmp < 0 ) max_tmp = 0;
            max_tmp += A[i];
            maxR[i] = max(maxR[i+1],max_tmp);
        }
    }

    //minL[i] :the minimum subarray of A[0...i].
    int diff(const vector<int> &A)
    {
        const int n = A.size();
        if(n < 2) return 0; //illegal
        vector<int> minL(n,0),maxL(n,0),minR(n,0),maxR(n,0);
        minSub(A,minL,minR,maxL,maxR);
        int ret = INT_MIN;
        for(int i=1;i<n;++i)
        {
            //cut [0...i-1] and [i..n-1]
            int s1 = abs(minL[i-1]-maxR[i]);
            int s2 = abs(maxL[i-1]-minR[i]);
            ret = max(ret,max(s1,s2));
        }
        return ret;
    }

```

}

T38 : 大家都知道facebook用户都是双向的好友，a是b的好友，那么b一定是a的好友，现在给定一个用户列表，其中有些用户是好友，有些不是，请判断，这些用户是否可以划分为两组，并且每组内的用户，互相都不是好友。如果能，请给出这个划分。

+ 例子1: 用户: {1, 2, 3} 好友关系: 1-2, 2-3 划分: {1,3} {2}
+ 例子2: 用户{1,2,3,4} 好友关系: 1-2, 2-3, 3-4, 4-1 划分: {1, 3}{2, 4}

根据二分图的特性，一条边上的两个点，肯定是属于不同的组。如果它们出现在同一个组中，肯定就不是二分图了。怎么判断，

一条边上的两个点，分属于不同的组呢？我们需要遍历图，如果找到一条边，两个节点，都在同一组，则不是二分图；如果图遍历完成之后，没有找到这样的边，则是二分图。我们在遍历的过程中，我们需要区分，一条边的两个节点分属于不同的组，这里我们用到了染色法。

核心思想如下：从某一个点开始，将这个节点染色为白色，并且开始广度优先遍历，找到与其相邻的节点，如果是二分图，相邻节点的颜色

都应该不同。如果是黑色，则不变；如果是无色，则染成黑色；如果是白色，也就是同色，程序退出。当图遍历完毕时，没有相邻节点同色的，则是二分图，标记为白色和黑色的两组就是一个划分。来看两个例子，第一个图中的例子2：

步骤	遍历节点	相邻节点	队列
1	1:黑	2:黑, 4:黑	2, 4
2	2:黑	1:白, 3:白	4, 3
3	4:黑	1:白, 3:白	3
4	3:白	2:黑, 4:黑	空

使用BFS.

T39 : 构造最大数

给定只包含正数的数组，给出一个方法，将数组中的数拼接起来，得到的数，是最大的。 例如：
[4, 94, 9, 14, 1] 拼接之后，所得最大数为：9944141。

分析：

下述分析最重要的一点就是说根据数组的第一位将数组分成几个组，每个组内按照第一位，第二位的顺序排序下去即可。如果组内只有一个元素，那么不管即可。

初看这个题目，肯定是要排序的。按照从左到右的第一个位置的数字，从大到小进行排序。如题目中的例子，结果是： 94 9 4 14 1 直接拼接为9494141显然不是最大的。那很自然的想法，就是考虑从左到右第二位的数字。考虑9和94的第二个数字，9没有第二个数字如何处理？这里有一个小技巧，就是用9补位。为什么用9呢？994>949，因为4<9，所以，拼接的方式 9 + “” + 94. 为了方便我们排序，所以用9进行了补位。关于上述的用9来补位，意味着什么呢？比如数字94，9的组合有949和994，那么显然最终我们希望的是什么呢，我们希望的是数字9在94的前面！！！这里便是下面的简单的quicksort的解法的基础！！！*。

再看一个例子：56,54,5。第一次排序结果为：

56 54 5

第二次呢？

56 55 54

此时要注意，第二55，是由5补位而来的。

再看一个复杂一点的例子，包含了：1位数，2位数，3位数：96 9 95 556 56 55 5 554 54 3 2 1

第一次排序：

96 9 95 556 56 55 5 554 54 3 2 1

96 9 95 556 56 55 5 554 54 3 2 1

第二次排序：

9 96 95 56 556 55 5 554 54 3 2 1

99 96 95 56 556 55 55 554 54 3 2 1

上面一行，是原始数字，下面一行，经过补位了。

第三次排序

9 96 95 56 556 55 5 554 54 3 2 1 99 96 95 565 556 555 555 554 545 3 2 1

9 96 95 56 556 55 5 554 54 3 2 1

99 96 95 565 556 555 555 554 545 3 2 1

第四次排序

9 96 95 56 556 55 5 554 54 3 2 1 99 96 95 565 556 555 555 554 545 3 2 1

9 96 95 56 556 55 5 554 54 3 2 1

99 96 95 565 556 555 555 554 545 3 2 1

处理最后三个个位数即可。则最终的结果是：996955655655554554321

上面的算法，排序处理自左向右第一位，然后处理第二位，一次类推，如果某一组内，即第一个数字相同的一组，长度不同时，采用第一位进行补位。

```
/compare lhsrhs rhslhs
bool cmp(int lhs, int rhs)
{
    cout << "compare : " << lhs << " " << rhs << endl;
    char tmp1[20],tmp2[20];
    sprintf(tmp1,"%d",lhs);
    sprintf(tmp2,"%d",rhs);
    cout << lhs << " " << tmp1 << endl;
    cout << rhs << " " << tmp2 << endl;
```

```

        int len1 = strlen(tmp1),len2=strlen(tmp2);
        if(len1 == len2) return lhs > rhs;

        //len1 != len2
        //add tmpw to tmp1, and add tmp1 to tmp2
        char temp[20] = {0};
        strcat(temp,tmp1);
        strcat(tmp1,tmp2);
        strcat(tmp2,temp);

        cout << "compare : " << tmp1 << " " << tmp2 << endl;

        if(strcmp(tmp1,tmp2) > 0) return true;
        return false;
    }
int partition(int *A, int left, int right)
{
    int pivot = A[right];
    int i=left;
    for(int k=left;k<right;++k)
    {
        if(cmp(A[k],pivot)) swap(A[i++],A[k]);
    }
    swap(A[i],A[right]);
    return i;
}

void quickSort(int *A, int left, int right)
{
    if(left >= right) return;
    int idx = partition(A,left,right);
    quickSort(A,left,idx-1);
    quickSort(A,idx+1,right);
}

void constructMaxNum(int *A, int n)
{
    if(A == nullptr || n <= 0 ) return ;
    quickSort(A,0,n-1);

    cout << "ret : ";
    for(int i=0;i<n;++i)cout<< A[i];
    cout << endl;
}

```

T40 : Morris 遍历

先序，中序，后序的Morris遍历的实现：

先序:

```
vector<int> preorderTraversal(TreeNode *root) {
    if(root == nullptr) return vector<int>();
    vector<int> ret;
    while(root)
    {
        if(root->left == nullptr)
        {
            ret.push_back(root->val);
            root = root->right;
            continue;
        }
        TreeNode *node = root->left;
        for(;node->right != nullptr && node->right != root; node=node-
>right);
        if(node->right == nullptr) //visit firstly
        {
            node->right = root;
            ret.push_back(root->val);
            root = root->left;
        }
        else //visited
        {
            node->right = nullptr;
            root = root->right;
        }
    }
    return ret;
}
```

中序

```

vector<int> inorderTraversal(TreeNode *root) {
    if(root == nullptr) return vector<int>();
    vector<int> ret;
    while(root)
    {
        if(root->left == nullptr)
        {
            ret.push_back(root->val);
            root = root->right;
            continue;
        }
        auto node = root->left;
        for(;node->right != nullptr && node->right != root;node=node-
>right);
        if(node->right == nullptr)
        {
            node->right = root;
            root=root->left;
        }
        else
        {
            node->right = nullptr;
            ret.push_back(root->val);
            root = root->right;
        }
    }
    return ret;
}

```

后序：（后序的版本比较复杂，添加了附加节点。）

```

void reverse(TreeNode *from, TreeNode *to)
{
    if(from==to) return;
    TreeNode *head=nullptr,*next=nullptr;
    while(from)
    {
        next=from->right;
        from->right=head;
        head=from;
        from=next;
    }
}
void reverse_print(TreeNode *from, TreeNode *to, vector<int> &ret)
{
    reverse(from,to);
    TreeNode *tmp=to;
    while(tmp)

```

```

    {
        ret.push_back(tmp->val);
        tmp=tmp->right;
    }
    reverse(to,from);
}
public:
    //Morris
    vector<int> postorderTraversal(TreeNode *root) {
        if(root == nullptr) return vector<int>();
        vector<int> ret;
        TreeNode dummy(0);
        dummy.left = root;
        root = &dummy;
        while(root)
        {
            if(root->left == nullptr)
            {
                root = root->right;
                continue;
            }
            auto node = root->left;
            for(;node->right && node->right != root; node=node->right);
            if(node->right == nullptr)
            {
                node->right = root;
                root=root->left;
            }
            else
            {
                node->right = nullptr;
                reverse_print(root->left,node,ret);
                root = root->right;
            }
        }
        return ret;
    }
}

```

T41: 小岛的故事

题目：一个岛上有100个人，其中有5个红眼睛，95个蓝眼睛。这个岛有三个奇怪的宗教规则。1. 他们不能照镜子，不能看自己眼睛的颜色。2. 他们不能告诉别人对方的眼睛是什么颜色。3. 一旦有人知道了自己是红眼睛，他就必须在当天夜里自杀。某天，有个旅行者到了这个岛。由于不知道这里的规矩，所以他在和全岛人一起的时候，不留神就说了一句话：你们这里有红眼睛的人。假设这个岛上的人足够聪明，每个人都可以做出缜密的逻辑推理。

问题一：请问这个岛上将会发生什么？

答案：第五天五个红眼睛一起自杀死了。思路：当旅行者说了这句话后，岛上所有人都知道有红眼睛存在了，那么会怎么办呢？假设岛上只有1个红眼睛，那么他马上知道自己是红眼睛，

当天就自杀了。

假设岛上有两个红眼睛，那么每个红眼睛都先假定自己不是红眼睛，那么他看到的岛上只有一个红眼睛，他会判断当天这个红眼睛就会死，可是当天没有事发生。于是他知道岛上不止一个红眼睛，而自己是另一个，于是双双在第二天自杀。依次类推5个红眼睛在第五天自杀了。

问题二：那么旅行者存在的意义是什么呢？毕竟一开始岛上每个人都知道岛上存在红眼睛。只要从岛上所有人都知道存在有红眼睛开始，这5个红眼睛，就会在第五天选择自杀，等旅行者来岛上，岛上的红眼睛早死光了。

答案：假设岛上只有一个红眼睛，这个红眼睛看到的是99个蓝眼睛，所以这个红眼睛没必要自杀，所以一个红眼睛是可以稳定存在的。

假设岛上有两个红眼睛，虽然岛上每个人都知道存在红眼睛，但红眼睛也不会自杀。因为每个红眼睛都假定自己是蓝眼睛，他看到的是1个红眼睛和98个蓝眼睛，由于一个红眼睛是可以稳定存在的，所以这时其中的一个红眼睛是不能断定自己也是红眼睛的。

同理5个红眼睛可以稳定存在。

总结：旅行者出现的意义在于，他说了那句话，大家的推理方式就变了。

T42: 忘我之乘积

给你一个数组A[1..n]，请你在O(n)的时间里构造一个新的数组

B[1..n]，使得 $B[i]=A[1]A[2]...A[n]/A[i]$ 。你不能使用除法运算。

```
vector<int> compute(const vector<int> &A)
{
    const int n = A.size();
    vector<int> B(n,1);
    B[0] = A[0];
    for(int i=1;i<n-1;++i)
        B[i] = B[i-1]*A[i];
    B[n-1] = B[n-2];
    int c = A[n-1];
    cout << "subArr : " ;
    for(auto b:B)cout << b << " "; cout << endl;
    for(int i=n-2;i>=1;--i)
    {
        B[i] = B[i-1]*c;
        c *= A[i];
    }
    B[0] = c;
    return B;
}
```

T43: 只使用加法实现减法，乘法和除法。

```

//implement the subtraction, multiplication, division with only add.

/*
 * implement subtraction
 */

//negative a
int negative(int a)
{
    int neg=0;
    int d=(a>0)?-1:1;
    while(a)
    {
        neg += d;
        a += d;
    }
    return neg;
}

int sub(int a, int b)
{
    int negb = negative(b);
    return a + negb;
}

//implement multiplication

int multiply(int a, int b)
{
    if( a < b ) return multiply(b,a); //multiplicand > multiplier is much
    faster as the times of add operation called is less.
    int sum = 0;
    for(int i=abs(b);i>0;--i)
    {
        sum += a;
    }
    if(b < 0) sum = negative(sum);
    return sum;
}

//implement division
int divide(int a, int b)
{
    if(b == 0 ) throw runtime_error("divisor is zero...\n");
    int absa=abs(a),absb=abs(b);
    int sum=0;
    int ret=0;
    while(sum + absb <= absa)
    {
        ret += 1;
    }
}

```



```
        sum += absb;
    }
    if((a < 0 && b < 0) || (a>0 && b>0))
        return ret;
    return negative(ret);
}
```

T44:海枯石烂

两个玩家，一堆石头，假设多于100块，两人依次拿，最后拿光者赢，

规则是： + 1. 第一个人不能一次拿光所有的； + 2. 第一次拿了之后，每人每次最多只能拿对方前一次拿的数目的两倍。求先拿者必胜策略，如果有的话。怎么证明必胜。

分析

这是斐波那契博弈，当且仅当石头个数是斐波那契数的时候先手必败。

证明：

让我们用第二数学归纳法证明：为了方便，我们将 n 记为 $f[i]$ 。//这里的 $f[i]$ 的意义应该是第 i 个斐波拉切数！！

首先 当 $i=2$ 时，因为不能全部去完，先手只能取1颗，显然必败，结论成立。

其次，假设当 $i \leq k$ 时，结论成立。则当 $i=k+1$ 时， $f[i] = f[k] + f[k-1]$ 。则我们可以把这一堆石子看成两堆，简称 k 堆和 $k-1$ 堆。（一定可以看成两堆，因为假如先手第一次取的石子数大于或等于 $f[k-1]$ ，则后手可以直接取完 $f[k]$ ，因为 $f[k] < 2*f[k-1]$ ）

对于 $k-1$ 堆，由假设可知，不论先手怎样取，后手总能取到最后一颗。下面我们分析一下后手最后取的石子数 x 的情况。

如果先手第一次取的石子数 $y \geq f[k-1]/3$ ，则这堆所剩的石子数小于 $2y$ ，即后手可以直接取完，此时 $x = f[k-1] - y$ ，则 $x \leq 2/3 * f[k-1]$ 。

我们来比较一下 $2/3f[k-1]$ 与 $1/2f[k]$ 的大小。即 $4f[k-1]$ 与 $3f[k]$ 的大小，对两值作差后不难得出，后者大。

所以我们得到， $x < 1/2 * f[k]$ 。

即后手取完 $k-1$ 堆后，先手不能一下取完 k 堆，所以游戏规则没有改变，则由假设可知，对于 k 堆，后手仍能取到最后一颗，所以后手必胜。

即 $i=k+1$ 时，结论依然成立。

那么，当 n 不是Fibonacci数的时候，情况又是怎样的呢？

这里需要借助“Zeckendorf定理”（齐肯多夫定理）：任何正整数可以表示为若干个不连续的Fibonacci数之和。

关于这个定理的证明，感兴趣的同学可以在网上搜索相关资料，这里不再详述。

分解的时候，要取尽量大的Fibonacci数。

比如分解85：85在55和89之间，于是可以写成 $85=55+30$ ，然后继续分解30,30在21和34之间，所以可以写成 $30=21+9$ ，

依此类推，最后分解成 $85=55+21+8+1$ 。

则我们可以把 n 写成 $n = f[a_1] + f[a_2] + \dots + f[a_p]$ 。 ($a_1 > a_2 > \dots > a_p$)

我们令先手先取完 $f[a_p]$ ，即最小的这一堆。由于各个 f 之间不连续，则 $a_{p-1} > a_p + 1$ ，则有 $f[a_{p-1}] > 2 * f[a_p]$ 。即后手只能取 $f[a_{p-1}]$ 这一堆，且不能一次取完。

此时后手相当于面临这个子游戏（只有 $f[a_{p-1}]$ 这一堆石子，且后手先取）的必败态，即先手一定可以取到这一堆的最后一颗石子。

同理可知，对于以后的每一堆，先手都可以取到这一堆的最后一颗石子，从而获得游戏的胜利。

T45：最多连续数的子集（并查集的应用。）

给一个整数数组，找到其中包含最多连续数的子集，比如给：15, 7, 12, 6, 14, 13, 9, 11，则返回: 5:[11, 12, 13, 14, 15]，最简单的方法是sort然后scan一遍，但是要 $O(n \lg n)$ 。有什么 $O(n)$ 的方法吗？

Leetcode上的Longest Consecutive Sequence，建立hash表，分别记录每个元素的左边界和右边界，然后取最大的即可。

```

int longestConsecutive(vector<int> &num) {
    const int n = num.size();
    if(n < 2) return n;
    map<int,int> lbound,rbound;
    int ret=0;
    for(auto x:num)
    {
        if(lbound.find(x) != lbound.end()) continue;
        lbound[x]=rbound[x]=x;
        //modify lbound
        if(lbound.find(x-1) != lbound.end())
        {
            lbound[x] = lbound[x-1];
            rbound[lbound[x]] = x;
        }
        //modify rbound
        if(rbound.find(x+1) != rbound.end())
        {
            rbound[x] = rbound[x+1];
            lbound[rbound[x+1]] = lbound[x];
            rbound[lbound[x]] = rbound[x+1];
        }

        ret = max(ret,rbound[x]-lbound[x]+1);
    }
    return ret;
}

```

T46：单链表之和

两个单链表（singly linked list），每一个节点里面一个0-9的数字，输入就相当于两个大数了。然后返回这两个数的和（一个新list）。这两个输入的list长度相等。

要求是：

- 1. 不用递归。
- 1. 要求算法在最好的情况下，只遍历两个list一次，最差的情况下两遍。

首先题意要清楚，链表等长，于是不用再paddle了。再次如果该链表是从低位到高位，那么很简单，直接遍历一遍即可，并且最终的进位在数组的末尾；如果链表的顺序是从高位到低位的话，

那么也就是说我们需要递归的来进行。

```
//head1 and head2 have same length.
int len(ListNode *head)
{
    int len = 0;
    while(head){++len;head=head->next;}
    return len;
}
ListNode *helper(ListNode *lhs, ListNode *rhs, int &carry)
{
    if(lhs==nullptr && rhs == nullptr) return nullptr;
    ListNode *ret = helper(lhs->next,rhs->next,carry);
    int sum = lhs->val+rhs->val+carry;
    carry = sum/10;
    int a = sum-carry*10;
    ListNode *node = new ListNode(a);
    node->next = ret;
    ret = node;
    return ret;
}
ListNode *addList(ListNode *head1, ListNode *head2)
{
    int len1 = len(head1), len2 = len(head2);
    if(len1 != len2)
    {
        cout << "This function is not available the lists with different
length..." <<endl;
        return nullptr;
    }
    int carry = 0;
    ListNode *ret = helper(head1,head2,carry);
    if(carry)
    {
        ListNode *node=new ListNode(1);
        node->next = ret;
        ret = node;
    }
    return ret;
}
```

T47：在C中你如何调整比特来找出一个机器内存内的堆栈向上或向下增长？

定义两个函数，一个函数内部定义一个局部变量，并且调用另外的

一个函数，在那个函数中同样定义一个局部变量，查看这两个局部变量的地址大小即可。

T48：哪些比特在TCP/IP三个包中是握手？

想到三次握手，SYN，SYN + ACK，ACK。

A：它是SYN、SYN+ACK、ACK。源想发送数据到目的地，第一步它发送SYN去问目的地是否它准备好接收数据，然后目的地想去回答它是否准备好接收比特流于是它发送SYN+ACK，然后源需要对该目的地来确认它得到的是请求于是源返回ACK。

TCP是面向连接的而UDP不是。TCP：连接建立->数据传输->连接终止。发起主机（客户端）发送一个同步(SYN标记设置)包发起一个连接。任何SYN包拥有一个序号，该序号在TCP段头是一个32位区。比如说这个事物的该序号值是x。其它主机接收数据包，从客户端记录了x的序号，并回复一个应答同步(SYN-ACK)。该应答号在TCP段头是一个32位区，它包含这个主机想要接收(x+1)的随后序号。该主机还发起了一个返回事物，这包含一个带有它自身值为y的发起序号的TCP段。发起主机答复以下一个序号(x+1)和一个简单的值为y+1的应答号，它是其它主机的序号值+1。

Q：快速乘法(Karatsuba)算法是什么？

A：它的算法是处理N位的两个大正数相乘，算法复杂度 $O(N^2)$ 。思想主要是将乘法转化为加法来计算以减少计算复杂度。

Q：告诉我当你输入www.google.com进入网页浏览器时发生什么

A1：解释DNS通过UDP端口53的运作等等，IP地址由DNS返回(有时ISP缓存这个URL->IP映射，这节省访问DNS的开销)，HTTP GET请求是发送到google.com服务器上，它返回HTML格式的google.com网页。

A2：Google的DNS服务器进行负载平衡，让用户能够最快地访问Google的内容。实现这靠发送用户一群没有重载的IP地址，并在地理上接近他们。每群有几千台服务器，并由该进群中的硬件在被连接到的集群进一步负载平衡，以发送请求到最少负载的网络服务器。

Q：N线程.. n资源..你使用什么协议以确保不会发生死锁？（dead lock）

A1：避免死锁用锁级别，每个线程将有不同的级别，高级别线程将能锁低级别线程而不是相反。A2：避免一个时间有多余一个的锁。A3：获得锁总是以同一次序。(get the lock with the same order.) A4：收缩同步段。

Q：你可以得到一个应用程序源码，当它运行时就会崩溃。在一个调试器中运行10次后，你发现在同一点它不再崩溃。该程序是单线程

并只使用C标准库。什么编程错误会导致这个崩溃？你如何测试每一个？

A1: 它将需要耗费时间或者随机数或者它做了记忆中讨厌的事或者它在Windows下运行。

A2: 可能在代码中有多条递归，它们导致一个堆栈溢出，导致“超出内存”异常。A3: 可能的原因可能是一个未初始化的指针/变量被用。

Q: 这是一个代码片段，如何使得该代码抛出

ArrayIndexOutOfBoundsException异常？

改代码本身是没有问题的，这个时候就需要考虑多线程的影响。

```
private static String[] list = null;

public static String[] getArray(size) {
    list= new String[size];
    for (int i = 0; i < size; i++) {
        list[i] = "a" + i;
    }
    return list;
}
```

A: 如果两个线程并行执行还用值10和7调用这个方法，第一个线程将创建10维数组，然后第二个线程将重新初始化7维同样的静态数组，接着第一个线程将运行这个数组从0到9还放值在里面，然而数组维数现在是7于是在list[7]它将抛出该异常。

Q: 写一函数f(a,b)，它带有两个字符串参数并返回一串字符，该字符串只包含在两个串中都有的并按照在a中的顺序。写一个版本算法复杂度 $O(N^2)$ 和一个 $O(N)$

```
A: private static String match(String a, String b) {  
  
    String result = "";  
  
    Set lettersSet = new HashSet(26);  
    for (int i=0; i  
lettersSet.add(new Character(b.charAt(i)));  
}  
  
    for (int i=0; i  
  
    if (lettersSet.contains(new Character(a.charAt(i)))) {  
  
        result=result+a.charAt(i);  
  
    }  
  
    }  
  
    return result;  
  
}
```

Q (Count and say ...):

1 1 1 2 1 1 2 1 1 1 1 1 2 2 1 下一个数是什么? (3 1 2 2 1 1)

序列的下一个数是什么10, 9, 60, 90, 70, 66, ?

A) 96 B) 10000000000000000000000000000000\

00000000000000000000000000000000\ C)

Either of the above D) None of the above

分析：

ten, nine, sixty, ninety, seventy, sixty-six, 每个数比前面的数多一个字母, 10有3个字母, 9有4个字母等。既然sixty-six(66)有8个字母下个数字需要有9个字母, 它就是ninety-six(96)。

Q: 有8块石头它们彼此相似除了一个比其它更重。为找出它，给你一个天平。找出最重的石头需要最少的称量次数是多少？

A: 划分石头成子集象(3,3,2)。使用天平去称量(3,3)，如果该天平保持平衡，则重的是在剩下的2个中，再次使用天平从这2个中找出那个重（总计称量

2次)。如果天平不平衡在称量(3,3)时,选石头子集比另一边更重的,把它分成子集(1,1,1),使用天平称量首先的2个,若保持平衡,剩下的石头则是更重的,否则高翘那边的就是更重的(总计称量2次)。

Q: 只给你二个鸡蛋,你能上100层楼,你想知道鸡蛋的硬度。鸡蛋可能很硬或很脆弱,如果鸡蛋从第m层掉下而没破裂,而从第m+1层掉下就破裂了,那么这个鸡蛋的硬度就是m。你需要找出这个m和在最坏情况下最少试验次数。(经典鸡蛋问题)

A: 计算机学生可能会首先用第一个鸡蛋做二分搜索($O(\log N)$)再用第二个递增做线性搜索 ($O(N)$),最后必将用线性搜索结束因为用第二个鸡蛋时你无法确定最高一层。因此,问题变为如何使用第一个鸡蛋来减少线性搜索。

于是如果第一个蛋破裂在最高点我们要扔x-1次并且我们必须从x层高扔第一个蛋。现在如果第一个蛋的第一次扔没有破裂,如果第一个蛋在第二次扔破了我们要扔x-2次第二个蛋。假如16是答案,我需要扔16次才能找到答案。来验证一下是否可以从16层开始扔,首先从16层扔如果它破裂了,我们尝试所有其下的楼层从1到15;如果没破我们还能扔15次,于是我们将从32层(16+15+1)再扔。原因是如果它在32层破裂我们能尝试其下所有楼层从17到31最坏扔第二个蛋14次(总共能扔16次了)。如果32层并没破,我们还剩下能扔13次,依此类推得:

1 + 15 16 如果它在16层破裂,从1到15层最坏扔15次第二个蛋

1 + 14 31 如果它在31层破裂,从17到30层最坏扔14次第二个蛋

1 + 13 45.....

1 + 12 58

1 + 11 70

1 + 10 81

1 + 9 91

1 + 8 100 在最后我们能轻易地做到因为我们有足够多扔的次数来完成任务

从上表我们能看到最佳的一个在最后一步将需要0次线性搜索。

能把上述规律写为: $(1+p) + (1+(p-1)) + (1+(p-2)) + \dots + (1+0) \geq 100$.

令 $1+p=q$ 上述式子变为 $q(q+1)/2 \geq 100$, 对100解答得到 $q=14$ 。

扔第一个蛋从层14, 27, 39, 50, 60, 69, 77, 84, 90, 95, 99, 100直到它破裂, 再开始扔第二个蛋。最坏情况只需14次。

T48:4. Given a byte array, which is an encoding of characters. Here is the rule:

- a. If the first bit of a byte is 0, that byte stands for a one-byte character;
- a. If the first bit of a byte is 1, that byte and its following byte together stand for a two-byte character. Now implement a function to decide if the last character is a one-byte character or a two-byte character. Constraint: You must scan the byte array from the end to the start.

分析: We need to check the previous bytes which are started with 1 continuously to decide the last byte.

we count the continuous bytes started with 1 from the 2-nd byte from the end.

- if the count is odd, the last byte is one-byte character.
- if the count is even, the last byte is two-byte character.

```
void decide(vector<uchar> &A)
{
    const int n = A.size();
    if( n < 2)
    {
        cout << "One-byte character..." << endl;
        return;
    }
    int cnt = 0; //count the continuous bytes started with 1 from the 2-nd
byte from the end.
    for(int j=n-2;j>=0;--j)
    {
        if(A[j] & 0x80) cnt += 1;
        else break;
    }
    if(cnt && cnt % 2)
        cout << "two-byte character..." << endl;
    else
        cout << "one-byte character..." << endl;
}
```

T49: 数组排序

给定一个数组，数组中有 n 个元素。数组有这样一个特性：每个数的位置索引与排序后该数的位置索引小于 k ($k < n$)。请问有没有时间复杂度低于 $n \log n$ 的算法来对这个数组进行排序。

考虑排序后的数组的第一个元素可能是原数组的 $[0, k-1]$ 范围内的元素；第二个元素是原数组中的 $[0 \dots k]$ 范围内的元素；第三个元素是原数组中的 $[0 \dots k+1]$ ，以此类推，可以一个大小为 k 的最小堆来得到一个有序的元素集合。

//时间复杂度为 $O(n \log k)$

```
void sorted(vector<int> &A, int k)
{
    const int n = A.size();
    if(k <= 0 || k > n || n < 2)
    {
        // cout << "Input k is illegal..." << endl;
        return;
    }
    priority_queue<int, vector<int>, greater<int> > pq;
    for(int i=0; i<k; ++i) pq.push(A[i]);
    for(int i=k; i<n; ++i)
    {
        A[0] = pq.top(); pq.pop();
        pq.push(A[i]);
    }
    int i=0;
    while(!pq.empty()) { A[n-k-i] = pq.top(); pq.pop(); }
    return ;
}
```

T50: Given a list of words, find two strings S & T such that:

- a. S & T have no common character
- b. S.length() * T.length() is maximized

Follow up: how to optimize and speed up your algorithm.

Analysis:

T51 : n 位数字

给定数字n，请给出方法，打印出所有的位数是n的数字，并且每一个数字，每一位从左到右依次变大。例如：n=3时
(123,124,125...789).

这里的算法比较简单，只需要一次输出就可以了。

```
void helper(int cur_digit, int cur_cnt, const int max_cnt, string &cur)
{
    if(cur_cnt == max_cnt)
    {
        cout << cur << endl;
        return;
    }
    cur.push_back('0' + cur_digit);
    helper(cur_digit+1, cur_cnt+1, max_cnt, cur);
    cur.pop_back();
}

void print(const int n)
{
    if(n <= 0 ) return;
    string cur;
    //the first digit
    for(int i=1; i<=10-n; ++i)
        helper(i, 0, n, cur);
}
```

T52:子矩阵的GCD

给定n*n的矩阵，我们需要查询任意子矩阵中所有数字的GCD，请给出一种设计思路，对矩阵进行预处理，加速查询。额外的空间需要在 $O(n^2)$ 内，预处理的时间复杂度，要尽可能的小。

T53:

An array : 1 3 0 2 4 9

input: dest-node: A0

output: all the source nodes: (A1, A3, A4)

Each element in this array means the steps it can take. Each element can go left or right. So

A[1] and A[4] can reach A[0]. A[1] can reach A[4], A[4] can reach A[0], so A[1] can reach A[0].
Output the index of element which can reach A[0].

这里的reach是说刚刚好使用A[i]步。