面试算法题总结

排序算法

冒泡排序

冒泡排序是两两进行比较，将较大值往后移动。

最差时间复杂度	$O(n^2)$
最优时间复杂度	$O(n)$
平均时间复杂度	$O(n^2)$
最差空间复杂度	总共$O(n)$，需要辅助空间$O(1)$

```
class BubbleSort {
public:
    int* bubbleSort(int* A, int n) {
        if(n <= 0)
            return NULL;
        for(int i=0; i<n-1;++i){
            for(int j=0; j<n-i-1;++j){
                // 从前往后比较，将最大值移动到末尾
                if(A[j]>A[j+1]){
                    swap(A[j],A[j+1]);
                }
            }
        }
        return A;
    }
};

// 改进版冒泡算法
int* BubbleSortOptimz(int* A, int n){
    int i, j;
    bool flag = true;
    for (int i = 0; i < n-1 && flag; i++){
        // 若 flag为false则退出循环
        flag = false;
        for (j = 0; j < n-1-i; --j){
            if (A[j] > A[j+1]){
                // 实现递增排序
                swap(A, j, j + 1);
                // 如果有数据交换，则flag是true
                flag = true;
            }
        }
    }
    return A;
}
```

选择排序

选择排序是每次在给定范围内搜索最小值，然后将其交换到当前范围的头部，所以第一次搜索范围是整个数组，搜索最小值交换到第一个位置，接着从第二个位置开始搜索整个数组的第二小的数值交换到第二个位置，依次进行。所以整个过程，交换次数只有n-1次。

简单选择排序的最大特点就是交换移动数据次数相当少。分析其时间复杂度发现，无论最好最差的情况，比较次数都是一样的，都需要比较$\sum_{i=1}^{n-1}(n-i)=(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$次。对于交换次数，最好的时候是交换0次，而最差的情况是$n-1$次。因此，总的时间复杂度是$O(n^2)$，并且在最好，最差和平均情况下时间复杂度都是这个，即$O(n^2)$。虽然与冒泡排序一样的时间复杂度，但是其性能上还是略好于冒泡排序，在n值较少时，选择排序会比冒泡排序快。

```
class SelectionSort {
public:
    int* selectionSort(int* A, int n) {
        // write code here
        int i, j, min;
        for(i=0; i < n-1; ++i){
            min = i;
            // 在后面的元素中进行比较，并保存最小元素的索引值
            for(j= i+1; j < n; ++j)
                if(A[min] > A[j])
                    min = j;
            // 进行交换，选择排序每次外层循环中只进行一次交换
            int tmp = A[min];
            A[min] = A[i];
            A[i] = tmp;
        }
        return A;
    }
};
```

插入排序

选择第i个元素，依次跟前面的元素进行比较，当前面的元素比它大，则将更大的元素往后移动，不断比较直到遇到小于它的元素。

直接插入排序算法是需要有一个保存待插入数值的辅助空间，辅助空间是$O(1)$。

在时间复杂度方面，最好的情况是待排序的表本身就是有序的，如{2,3,4,5,6}，比较次数则是$n-1$次，然后不需要进行移动，时间复杂度是$O(n)$。

最差的情况就是待排序表是逆序的情况，如{6,5,4,3,2},此时需要比较$\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$次，而记录的移动次数也达到最大值$\sum_{i=2}^{n}(i+1) = \frac{(n+4)(n-1)}{2}$次。

如果排序记录是随机的，那么根据概率相同的原则，平均比较和移动次数约为$\frac{n^2}{4}$。因此，可以得出直接插入排序算法的时间复杂度是$O(n^2)$。同时也可以看出，直接插入排序算法会比冒泡排序和简单选择排序算法的性能要更好一些。

插入排序不适合对于数据量比较大的排序应用。但是，如果需要排序的数据量很小，例如，量级小于千，那么插入排序还是一个不错的选择。 插入排序在工业级库中也有着广泛的应用，在STL的sort算法和stdlib的qsort算法中，都将插入排序作为快速排序的补充，用于少量元素的排序（通常为8个或以下）.

```
class InsertionSort {
public:
    int* insertionSort(int* A, int n) {
        // write code here
        if(n <= 0)
            return A;
        int tmp;
        for(int i=1; i<n; i++){
            tmp = A[i];
            int j = i-1;
            // 如果 i 前面的元素都比A[i]大，则依次往后移动，直到一个小于A[i]的位置
            while(j >= 0 && A[j] > tmp){
                A[j+1] = A[j];
                --j;
            }
            A[j+1] = tmp;
        }
        return A;
    }
};
```

归并排序

递归实现：将n长度的数组分解成n/2长度，接着继续对半分解，直到得到长度为1的子数组，并两两进行比较，然后合并成一个更长的数组，依次合并，最终得到原数组，并且是一个排序好的数组。

归并排序的效率是比较高的，设数列长为N，将数列分开成小数列一共要$logN$步，每步都是一个合并有序数列的过程，时间复杂度可以记为$O(N)$，故一共为$O(N*logN)$。因为归并排序每次都是在相邻的数据中进行操作，所以归并排序在$O(N*logN)$的几种排序方法（快速排序，归并排序，希尔排序，堆排序）也是效率比较高的。

归并排序的时间复杂度是$O(nlogn)$，并且无论是最好、最坏还是平均都是同样的时间性能。另外，在归并过程中需要与原始记录序列同样数量的存储空间存放归并结果，并且递归时需要深度为log_2n的栈空间，因此空间复杂度是$O(n+logn)$。

另外，归并排序是使用两两比较，不存在跳跃，所以归并排序是一个稳定的排序算法。

总体来说，归并排序是一个比较占用内存，但效率高且稳定的算法。

```
class MergeSort {
public:
    int* mergeSort(int* A, int n) {
        if(n <= 0)
            return NULL;
        int* tmp = new int[n];
        MergeSortInner(A,0, n-1,tmp);
        delete[] tmp;
        return A;
    }
    void MergeSortInner(int* A, int start, int end,int* tmp){
        if(start < end){
            int mid = (start+end)/2;
            // 令左边有序
            MergeSortInner(A,start, mid, tmp);
            // 令右边有序
            MergeSortInner(A,mid+1, end, tmp);
            // 合并
            merge(A, start, mid, end, tmp);
        }
    }
    void merge(int* A, int start, int mid, int end, int* tmp){
        // 定义左边开始和结尾位置
        int i = start, ie = mid;
        // 右边起始和结束位置
        int j = mid+1, je = end;
        // 进行比较，然后在tmp数组中进行排序
        int k=0;
        while(i <= ie && j <= je){
            if(A[i] < A[j]){
                tmp[k++] = A[i++];
            }else{
                tmp[k++] = A[j++];
            }
        }
        // 复制剩下的元素
        while(i <= ie)
            tmp[k++] = A[i++];
        while(j <= je)
            tmp[k++] = A[j++];
        // 将tmp排序好的部分复制回原数组中
        for(int l=0; l<k; ++l)
            A[start+l] = tmp[l];
    }
};
```

迭代实现版本：

迭代实现则是自底向上，先依次比较长度为1的子数组，然后升级到长度为2，依次增加子数组长度，直到到达数组长度为止。

非递归版本的归并排序算法避免了递归时深度为$log_2 n$的栈空间，空间复杂度是$O(n)$，并且避免递归也在时间性能上有一定的提升。应该说，使用归并排序时，尽量考虑用非递归方法。

```
// 迭代实现归并排序
int* mergeSort(int* A, int n) {
    // write code here
    int *tmp = new int[n];

    for (int size = 1; size < n; size *= 2){
        // size 是控制每个分块的大小，先从1开始，每次加倍增加，即从1，2，4...直到整个数组长度为止
        for (int start = 0; start < n; start += size + size){
            int k = start;
            // 设置一个分块的起始、中间和结束位置，通过size来控制
            int left = start, mid = min(left + size, n), right = min(left + size + size, n);

            int start1 = left, end1 = mid;
            int start2 = mid, end2 = right;
            // 通过tmp数组来对分块内的两个部分进行排序
            while (start1 < end1 && start2 < end2)
                tmp[k++] = (A[start1] < A[start2]) ? A[start1++] : A[start2++];
            while (start1 < end1)
                tmp[k++] = A[start1++];
            while (start2 < end2)
                tmp[k++] = A[start2++];
            // 将排序好的部分赋回原数组A
            for (int j = left; j < right; ++j)
                A[j] = tmp[j];
        }
    }
    delete[] tmp;
    return A;
}
```

快速排序

快速排序的时间性能取决于快速排序递归的深度。在最优情况下，`Partition()`每次都划分得很均匀，如果排序n个关键字，其递归树的深度技术$\lfloor log_n \rfloor + 1$，即需要递归$log_2 n$次，其时间复杂度是$O(nlogn)$。而最坏的情况下，待排序的序列是正序或逆序，得到的递归树是斜树，最终其时间复杂度是$O(n^2)$。

平均情况可以得到时间复杂度是$O(nlogn)$，而空间复杂度的平均情况是$O(logn)$。但是由于关键字的比较和交换是跳跃进行的，所以快速排序也是不稳定排序。

最差时间复杂度	$O(n^2)$
最优时间复杂度	$O(nlogn)$
平均时间复杂度	$O(nlogn)$
最差空间复杂度	根据实现的方式不同而不同

```
class QuickSort {
public:
    int* quickSort(int* A, int n) {
        if(n <= 0)
            return NULL;
        qSort(A,0,n-1);
        return A;
    }
    void qSort(int* A, int start, int end){
        if(start < end){
            // 取第一个值为基准值
            int privot = A[start];
            int low = start;
            int high = end;
            while(low < high){
                while(high > low && A[high] >= privot)
                    // 从后往前寻找小于privot的元素
                    --high;
                if(low < high)
                    A[low++] = A[high];
                while(low < high && A[low] <= privot)
                    ++low;
                if(low < high)
                    A[high--] = A[low];
            }
            A[low] = privot;
            qSort(A,start, low-1);
            qSort(A, low+1, end);
        }
    }
};
```

堆排序

堆排序的运行时间主要是消耗在初始构造堆和在重建堆时的反复筛选上。

在构建堆的过程中，因为是从完全二叉树的最下层最右边的非叶结点开始构建，将它与其孩子进行比较和若有必要的交换，对每个非叶结点，最多进行两次比较和互换操作，这里需要进行这种操作的非叶结点数目是$\lfloor \frac{n}{2} \rfloor$个，所以整个构建堆的时间复杂度是$O(n)$。

在正式排序的时候，第i取堆顶记录重建堆需要用$O(logi)$的时间(完全二叉树的某个结点到根结点的距离是$\lfloor log_2 i \rfloor + 1$)，并且需要取$n-1$次堆顶记录，因此，重建堆的时间复杂度是$O(nlogn)$。

所以，总体上来说，堆排序的时间复杂度是$O(nlogn)$。由于堆排序对原始记录的排序状态并不敏感，因此它无论最好、最坏和平均时间复杂度都是$O(nlogn)$。同样由于记录的比较与交换是跳跃式进行，堆排序也不是稳定的排序算法。

另外，由于初始构建堆需要的比较次数较多，因此，它并不适合待排序序列个数较少的情况。

```
void hSort(int *A, int i, int n){
    int left = i * 2 + 1;
    while (left < n){
        // 对比左右子树，选择更大的数值
        if (left + 1 < n && A[left] < A[left + 1])
            left++;
        // 再与父结点进行比较
        if (A[i] > A[left])
            return;
        else{
            int tmp = A[left];
            A[left] = A[i];
            A[i] = tmp;
            // 此时就继续往下比较，用left作为父结点，进行比较
            i = left;
            left = i * 2 + 1;
        }
    }
}

int* heapSort(int* A, int n) {
    // write code here
    // 首先开始建立一个最大堆，从最后一个非叶结点开始
    for (int i = n / 2 - 1; i >= 0; --i)
        hSort(A, i, n);
     // 再开始进行堆排序，最大堆实现升序排序
    for (int i = n - 1; i >= 0; --i){
        // 每次先将堆顶元素和末尾元素交换，再进行堆排序
        int tmp = A[0];
        A[0] = A[i];
        A[i] = tmp;
        // 每次将最大数值放到堆顶，交换后，最大值就放到数组末尾了。
        hSort(A, 0, i);
    }
    return A;
}
```

递归版本的堆排序的代码只需要修改`hSort`函数：

```
// 递归版本的堆排序
void hSort(int* A, int start, int end){
        int left = start*2+1;
        int right = left+1;
        int largest = start;
        if(left < end && A[start] < A[left])
            largest = left;
        if(right < end && A[largest]<A[right])
            largest = right;
        if(largest != start){
            swap(A[start],A[largest]);
            hSort(A, largest, end);
        }
    }
```

希尔排序

希尔排序的时间复杂度与增量序列的选取有关，例如希尔增量时间复杂度为$O(n^2)$，而Hibbard增量的希尔排序的时间复杂度为$O(N^{\frac{5}{4}})$，但是现今仍然没有人能找出希尔排序的精确下界。

```
class ShellSort {
public:
    int* shellSort(int* A, int n) {
        // write code here
        // 选择初始增量为数组长度的一半
        int gap = n/2;
        while(gap > 0){
            for(int i=gap; i<n; ++i){
                // 起始位置从gap开始
                int tmp = A[i];
                int j = i-gap;
                // 每次比较的增加或减少都是 gap
                while(j >= 0 && A[j] > tmp){
                    A[j+gap] = A[j];
                    j -= gap;
                }
                A[j+gap] = tmp;
            }
            // 逐步缩小增量，直到等于1为止。
            gap /= 2;
        }
        return A;
    }
};
```

计数排序

当输入的元素是**n** 个**0**到**k**之间的整数时，它的运行时间是 $O(n+k)$。计数排序不是比较排序，排序的速度快于任何比较排序算法。

```
class CountingSort {
public:
    int* countingSort(int* A, int n) {
        if(n <= 0)
            return NULL;
        int maxV = getMax(A,n)+1;
        // 创建长度为maxV的桶
        vector<int> res(maxV,0);
        for(int i=0; i<n;++i){
            // 统计每个数值在数组中出现的次数
            res[A[i]]++;
        }
        // 排序
        int k=0;
        for(int i=0; i<maxV;++i){
            for(int j=0; j<res[i];++j)
                A[k++] = i;
        }
        return A;
    }
    int getMax(int* A, int n){
        // 返回最大值
        int res = A[0];
        for(int i=1; i<n; ++i)
            res = (A[i] > res)? A[i]:res;
        return res;
    }
};
```

基数排序

基数排序的时间复杂度是$O(kn)$，其中n是排序元素个数，k是数字位数。

注意这不是说这个时间复杂度一定优于$O(n \cdot log(n))$，因为k的大小一般会受到n的影响。 用排序n个不同整数来举例，假定这些整数以B为底，这样每位数都有B个不同的数字，k就一定不小于$logB(n)$。由于有B个不同的数字，所以就需要B个不同的桶，在每一轮比较的时候都需要平均$n \cdot log2(B)$ 次比较来把整数放到合适的桶中去，所以就有：

$$k \geq logB(n)$$

每一轮(平均)需要 $n \cdot log_2(B)$ 次比较

所以，基数排序的平均时间T就是：

$$T \geq logB(n) \cdot n \cdot log2(B) = log2(n) \cdot logB(2) \cdot n \cdot log2(B) = log2(n) \cdot n \cdot logB(2) \cdot log2(B) = n \cdot log2(n)$$

所以和比较排序相似，基数排序需要的比较次数：$T \geq n \cdot log2(n)$。 故其时间复杂度为 $\Omega(n \cdot log2(n)) = \Omega(n \cdot logn)$ 。

```
class RadixSort {
public:
    int* radixSort(int* A, int n) {
        if(n <= 0)
            return NULL;
        int maxBits_ = maxBits(A,n);
        // 相除的最大数
        int bits = pow(10, maxBits_-1);
        // 建立10个桶，对应位数是0-9
        vector<int> res[10];
        int c = 1;
        while(c <= bits){
            for(int i=0; i<n; ++i){
                // 遍历数组，依次求取对应位数上的数值并存放到对应的桶中
                int lgd = (A[i] / c) %10;
                res[lgd].push_back(A[i]);
            }
            // 重新排序
            int k=0;
            for(int i=0; i<10; ++i){
                for(int m=0; m<res[i].size(); ++m)
                    A[k++] = res[i][m];
                res[i].clear();
            }
            // 准确计算更高一位的数值
            c *= 10;
        }
        return A;
    }
    // 统计数组中元素的最大位数
    int maxBits(int* A, int n){
        int _max = 0;
        for(int i=0;i<n; ++i){
            int d = 0;
            int a = A[i];
            // 统计当前元素的位数
            while(a){
                a /= 10;
                ++d;
            }
            _max = (d>_max)? d:_max;
        }
        return _max;
    }
};
```

桶排序

桶排序最好情况下使用线性时间$O(n)$，很显然桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为 其它部分的时间复杂度都为$O(n)$;很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

可以证明，即使选用插入排序作为桶内排序的方法，桶排序的平均时间复杂度为线性。

```
void BucketSort(int * arr, int n)
{
    vector<int> bucket[10];
    for (int i = 0; i < n; i++)
    {
        int temp = arr[i];
        int flag = (int)(arr[i] / 10); //flag标识小数的第一位
        bucket[flag].push_back(temp); //用二维数组的每个向量来存放小数第一位相同的数据
        int j = bucket[flag].size() - 1;

        /* 利用插入排序对每一行进行排序 */
        for (; j > 0 && temp < bucket[flag][j - 1]; --j)
        {
            bucket[flag][j] = bucket[flag][j - 1];
        }
        bucket[flag][j] = temp;
    }

    /* 所有数据重新排序 */
    int k = 0;
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j< bucket[i].size(); j++)
        {
            arr[k++] = bucket[i][j];
        }
    }
}
```

排序算法性质特点总结

- 时间复杂度方面，最好、平均、最差情况都相同的有选择排序($O(n^2)$)、归并排序($O(nlogn)$)、堆排序($O(nlogn)$)
- 稳定性方面，不稳定的排序算法是快速排序、选择排序、堆排序和希尔排序

1. 小范围排序

题目描述如下：

> 已知一个几乎有序的数组，几乎有序是指，如果把数组排好顺序的话，每个元素移动的距离可以不超过k，并且k相对于数组来说比较小。请选择一个合适的排序算法针对这个数据进行排序。
>
> 给定一个int数组**A**，同时给定A的大小**n**和题意中的**k**，请返回排序后的数组。
>
> 测试样例：
>
> > [2,1,4,3,6,5,8,7,10,9],10,2
> >
> > 返回：[1,2,3,4,5,6,7,8,9,10]

解法如下：

```
class ScaleSort {
public:
    vector<int> sortElement(vector<int> A, int n, int k) {
        // write code here
        // 建立一个最小堆
        vector<int>  myheap(A.begin(), A.begin() + k);
        buildHeap(myheap, k);
        // 每次弹出堆顶，并加入下一个元素，即i+k
        for (int i = 0; i != n - k; i++){
            A[i] = myheap[0];
            myheap[0] = A[i + k];
            heapSort(myheap, 0, k);
        }
        // 对于剩下的n-k 到 n-1位置的元素的处理
        for (int i = 0; i != k; i++){
            // 起始位置是从 n-k 开始的
            A[i + n - k] = myheap[0];
            // 堆内将堆顶和堆后面的元素依次交换
            myheap[0] = myheap[k - 1 - i];
            heapSort(myheap, 0, k - i);
        }
        return A;

    }
    // 建堆
    void buildHeap(vector<int>& a, int k){
        for(int i=k/2 -1; i>=0; --i)
            heapSort(a, i, k);
    }
    void heapSort(vector<int>& A, int i, int n){
        int left = i * 2 + 1;
        while (left < n){
            // 对比左右子树，选择更小的数值
            if (left + 1 < n && A[left] > A[left + 1])
                left++;
            // 再与父结点进行比较
            if (A[i] < A[left])
                return;
            else{
                int tmp = A[left];
                A[left] = A[i];
                A[i] = tmp;
                // 此时就继续往下比较，用left作为父结点，进行比较
                i = left;
                left = i * 2 + 1;
            }
        }
    }
};
```

上述解法主要使用了堆排序，先建立一个长为`k`的最小堆，每次将堆顶按顺序放回到原数组中，然后要注意`n-k`位置后的元素个数会不足k个。并且堆排序要使用非递归形式，递归形式的空间复杂度是$O(logN)$。

2. 重复值判断

题目描述如下：

> 请设计一个高效算法，判断数组中是否有重复值。必须保证额外空间复杂度为O(1)。
>
> 给定一个int数组**A**及它的大小**n**，请返回它是否有重复值。
>
> 测试样例：
>
> > [1,2,3,4,5,5,6],7
> >
> > 返回：true

解法如下：

```
class Checker {
public:
    bool checkDuplicate(vector<int> a, int n) {
        // write code here
        // 建立一个最小堆
        buildHeap(a, n);
        // 排序
        for(int i=0; i<n; ++i){
            swap(a[n-i-1], a[0]);
            hSort(a, 0, n-i-1);
        }
        // 判断是否存在重复数值
        for(int i=1; i < n; ++i){
            if(a[i-1] == a[i])
                return true;
        }
        return false;
    }
    void buildHeap(vector<int>& a, int n){
        for(int i= n/2-1; i >= 0; --i)
            hSort(a, i, n);
    }
    void hSort(vector<int>& A, int i, int n){
        int left = i * 2 + 1;
        while (left < n){
            // 对比左右子树，选择更小的数值
            if (left + 1 < n && A[left] > A[left + 1])
                left++;
            // 再与父结点进行比较
            if (A[i] < A[left])
                return;
            else{
                int tmp = A[left];
                A[left] = A[i];
                A[i] = tmp;
                // 此时就继续往下比较，用left作为父结点，进行比较
                i = left;
                left = i * 2 + 1;
            }
        }
    }
};
```

另一种解法，不满足空间复杂度的是如下：

```
class Checker {
public:
    bool checkDuplicate(vector<int> a, int n) {
        // write code here
        int min = a[0], max=a[0];
        // 查找数组中的最大值和最小值
        for(int i=1; i<n; ++i){
            min = (min < a[i])? min:a[i];
            max = (max > a[i])? max:a[i];
        }
        // 建立一个新的数组
        vector<int> tmp(max-min+1);
        // 统计每个索引值的数量，如果大于1，说明有重复值
        for(int i=0; i<n; ++i){
            if(++tmp[a[i]-min]>1)
                return true;
        }
        return false;
    }
};
```

这是一种新建一个数组，用于统计数组中每个数出现次数。

3. 有序数组合并

题目如下：

> 有两个从小到大排序以后的数组A和B，其中A的末端有足够的缓冲空容纳B。请编写一个方法，将B合并入A并排序。
>
> 给定两个有序int数组**A**和**B**，A中的缓冲空用0填充，同时给定A和B的真实大小int **n**和int **m**，请返回合并后的数组。

解法如下：

```
class Merge {
public:
    int* mergeAB(int* A, int* B, int n, int m) {
        // write code here
        // 分别定义指向A和B数组尾元素位置的变量
        int pa = n-1, pb = m-1;
        // 定义指向A末尾刚好是两个数组大小的位置
        int p = n+m-1;
        while(pa>=0 && pb >= 0){
            // 通过比较，将较大数放在数组A末端
            A[p--] = (A[pa] >= B[pb])? A[pa--]:B[pb--];
        }
        // 如果数组B还有未进行比较的，直接复制到数组A前面
        while(pb >= 0)
            A[p--] = B[pb--];
        return A;
    }

};
```

这里采用从后往前进行比较来合并数组，并且注意如果数组A还有未进行比较的，可以直接跳过，而如果是数组B，则需要将其复制到数组A前面的相应位置。

4. 三色排序

题目描述如下：

> 有一个只由0，1，2三种元素构成的整数数组，请使用交换、原地排序而不是使用计数进行排序。
>
> 给定一个只含0，1，2的整数数组**A**及它的大小，请返回排序后的数组。保证数组大小小于等于500。
>
> 测试样例：
>
> > [0,1,1,0,2,2],6
> >
> > 返回：[0,0,1,1,2,2]

解法如下：

```
class ThreeColor {
public:
    vector<int> sortThreeColor(vector<int> A, int n) {
        // 数组左边设0区，右边设2区，把0放到数组0区，2放到数组2区，当遍历到2区时结束
        int start = 0, end = n-1;
        for(int i=0; i<n; ++i){
            if(i > end)
                break;
            if(A[i] == 0 && start < i)
                //交换后，下次判断还要为当前位置
                swap(A[i--], A[start++]);
            else if(A[i] == 2 && i < end)
                swap(A[i--], A[end--]);
        }
        return A;
    }
};
```

这是在左边设置0区，右边设置2区，即分别将0和2放到这两个区域，同时注意交换后还要判断交换过来的数值，以及这两个区的索引和当前进行循环的索引`i`大小的判断。

5. 有序矩阵查找

题目如下：

> 现在有一个行和列都排好序的矩阵，请设计一个高效算法，快速查找矩阵中是否含有值x。
>
> 给定一个int矩阵**mat**，同时给定矩阵大小**n**x**m**及待查找的数**x**，请返回一个bool值，代表矩阵中是否存在x。所有矩阵中数字及**x**均为int范围内整数。保证**n**和**m**均小于等于1000。
>
> 测试样例：
>
> > [[1,2,3],[4,5,6],[7,8,9]],3,3,10
> >
> > 返回：false

解法如下：

```
class Finder {
public:
    bool findX(vector<vector<int> > mat, int n, int m, int x) {
        if(n <= 0 || m <= 0 || mat.size() <= 0)
            return false;
        if(mat[0].size() <= 0)
            return false;
        // 从矩阵右上角开始查找
        int r = 0, c = m-1;
        while(r < n && c >= 0){
            if(mat[r][c] == x)
                return true;
            else if(mat[r][c] > x)
                --c;
            else
                ++r;
        }
        return false;
    }
};
```

上述解法是从矩阵右上角开始进行查找的。

6. 最短子数组

题目如下：

> 对于一个数组，请设计一个高效算法计算需要排序的最短子数组的长度。
>
> 给定一个int数组**A**和数组的大小**n**，请返回一个二元组，代表所求序列的长度。(原序列位置从0开始标号,若原序列有序，返回0)。保证A中元素均为正整数。
>
> 测试样例：
>
> > [1,4,6,5,9,10],6
>
> > 返回：2

解法如下：

```
class Subsequence {
public:
    int shortestSubsequence(vector<int> A, int n) {
        int max=A[0], min=A[n-1];
        int rt=0, lt=0;
        // 从左到右遍历保存最大的数值，并记录比最大数要小的位置
        for(int i=0; i<n; ++i){
            max = (A[i] > max)? A[i] : max;
            rt = (A[i] < max)? i:rt;
        }
        // 从右到左遍历数组，保存最小值，并记录比最小值大的位置
        for(int i=n-1; i>=0; --i){
            min = (A[i] < min)? A[i]:min;
            lt = (A[i] > min)? i : lt;
        }
        return (rt != lt ? (rt-lt+1):0);
    }
};
```

先从左到右查找最大数，并记录比最大数小的位置，然后从右到左查找最小数，记录比最小数大的位置，判断两个位置是否相等，如果是，返回0，否则，则相减并加1。

7. 相邻两数最大差值

题目如下：

> 有一个整形数组A，请设计一个复杂度为O(n)的算法，算出排序后相邻两数的最大差值。
>
> 给定一个int数组**A**和**A**的大小**n**，请返回最大的差值。保证数组元素多于1个。
>
> 测试样例：
>
> > [1,2,5,4,6],5
> >
> > 返回：2

解法如下：

```
class Gap {
public:
    int maxGap(vector<int> A, int n) {
        int maxNum = getMax(A,n);
        // 建立一个桶
        vector<int> bucket(maxNum+1,0);
        // 统计数组中不同数值的个数
        for(int i=0; i<n; ++i){
            ++bucket[A[i]];
        }
        // 计算空缺的桶的索引值的差值
        int res = 0;
        int tmp = 0;
        for(int j=1; j<=maxNum; ++j){
            if(bucket[j] > 0){
                // 更新当前最大差值
                res = (res > tmp)? res:tmp;
                tmp = 0;
            }else{
                ++tmp;
            }
        }
        // 如果res!=0,需要加1
        return (res !=0)?(res+1) : res;
    }
    int getMax(vector<int> A, int n){
        int max=A[0];
        for(int i=1; i<n; ++i)
            max = (max > A[i])? max:A[i];
        return max;
    }
};
```

这是利用桶排序的思路，先根据数组最大值`maxNum`创建一个长度是`maxNum+1`的桶，然后根据数组元素的数值作为索引，找到其在桶中的位置，让对应位置的桶计数加1，之后通过空缺的桶的连续个数来得到数组相邻两数的最大差值。

字符串

1. 词语变形

题目如下：

> 对于两个字符串A和B，如果A和B中出现的字符种类相同且每种字符出现的次数相同，则A和B互为变形词，请设计一个高效算法，检查两给定串是否互为变形词。
>
> 给定两个字符串**A**和**B**及他们的长度，请返回一个bool值，代表他们是否互为变形词。
>
> 测试样例：
>
> > "abc", 3, "bca", 3
>
> > 返回：true

解法如下：

```
class Transform {
public:
    bool chkTransform(string A, int lena, string B, int lenb) {
        // 分别建立两个数组作为哈希表
        char da[256] = {0}, db[256] = {0};
        // 分别统计两个数组的字符种类和出现个数
        for(int i=0; i<lena; ++i)
            ++da[A[i]];
        for(int j=0; j<lenb; ++j)
            ++db[B[j]];
        for(int k=0; k<256; ++k){
            if(da[k] != db[k])
                return false;
        }
        return true;
    }
};
```

这个方法主要是使用哈希表来分别统计两个数组的字符种类和出现次数，然后一一比较即可。

另一种解决方法，只建立一个哈希表如下：

```
bool chkTransform(string A, int lena, string B, int lenb) {
        // write code here
        vector<int> m(256,0);
        bool result = true;
        for(int i=0; i<= lena; ++i)
            ++m[A[i]];
        for(int i=0; i<= lenb; ++i)
            --m[B[i]];
        for(int i=0; i<= lena; ++i){
            if(m[A[i]] != 0)
                return false;
        }

        return result;
    }
```

2. 两串旋转

> 如果对于一个字符串A，将A的前面任意一部分挪到后边去形成的字符串称为A的旋转词。比如A="12345",A的旋转词有"12345","23451","34512","45123"和"51234"。对于两个字符串A和B，请判断A和B是否互为旋转词。
>
> 给定两个字符串**A**和**B**及他们的长度**lena**，**lenb**，请返回一个bool值，代表他们是否互为旋转词。
>
> 测试样例：
>
> > cdab",4,"abcd",4
> >
> > 返回：true

解法如下：

```
bool chkRotation(string A, int lena, string B, int lenb) {
        // 先判断两个字符串是否长度相同
        if(lena != lenb)
            return false;
        if(A.size() <= 0 && B.size() > 0)
            return false;
        if(A.size() > 0 && B.size() <= 0)
            return false;
        // 拼接两个字符串A
        string str = A + A;
        int lens = lena + lena;
        int b_index = 0;
        for(int i=0; i < lens && b_index < lenb; ++i)
            // 遍历新字符串，寻找字符串B
            if(str[i] == B[b_index])
                ++b_index;
        return (b_index == lenb);
    }
```

上述解法是先判断两个字符串是否相等，相等的情况下，拼接两个字符串A得到一个新字符串，在这个新字符串中寻找是否存在字符串B。

另一个简洁的代码如下：

```
bool chkRotation(string A, int lena, string B, int lenb) {
        // write code here
        if(lena==lenb){
        //find函数的返回值是整数，假如字符串存在包含关系，其返回值必定不等于npos，但如果字符串不存在包含关系，
         //那么返回值就一定是npos
            string C=A+A;
            if(C.find(B)!= string::npos){
               return  true;
            }
        }
        return false;
    }
```

或者如下解法：

```
class ReverseEqual {
public:
    bool checkReverseEqual(string s1, string s2) {
        if(s1.size() == 0 || s2.size() == 0)
            return false;
        // 拼接两个s1
        string str = s1+s1;
        // 如果能在新的str中找到s2，则表示s2是s1循环移位得到的
        if(str.find(s2) == -1)
            return false;
        return true;
    }
};
```

3. 句子的逆序

> 对于一个字符串，请设计一个算法，只在字符串的单词间做逆序调整，也就是说，字符串由一些由空格分隔的部分组成，你需要将这些部分逆序。
>
> 给定一个原字符串**A**和他的长度，请返回逆序后的字符串。
>
> 测试样例：
>
> > "dog loves pig",13
> >
> > 返回："pig loves dog"

解法如下：

```
string reverseSentence(string A, int n) {
        if(A.size() <= 0)
            return NULL;
        // 先翻转整个字符串
        reverse(A.begin(), A.end());
        int pBegin = 0, pEnd = 0;
        // 再逐个单词翻转
        while(pBegin < n){
            if(A[pBegin] == ' '){
                // 开头指针指向空格，两个指针同时增加
                ++pBegin;
                ++pEnd;
            }else if(A[pEnd] == ' ' || A[pEnd] == '\0'){
                // 尾指针指向空格或者结束符，都表示遇到一个单词
                reverse(A.begin()+pBegin, A.begin()+pEnd);
                pBegin = pEnd;
            }
            else{
                ++pEnd;
            }
        }
        return A;
    }
```

上述解法首先是旋转整个字符串，然后再逐个单词旋转。

4. 字符串移位

> 对于一个字符串，请设计一个算法，将字符串的长度为len的前缀平移到字符串的最后。
>
> 给定一个字符串**A**和它的长度，同时给定**len**，请返回平移后的字符串。
>
> 测试样例：
>
> > "ABCDE",5,3
> >
> > 返回："DEABC"

解法如下：

```
string stringTranslation(string A, int n, int len) {
        if(A.size() <= 0)
            return "";
        // 先旋转前len个字符
        reverseString(A, 0, len-1);
        // 然后选择剩余字符串
        reverseString(A, len, n-1);
        // 最后选择整个字符串
        reverseString(A, 0, n-1);
        return A;
    }

    void reverseString(string& A, int start, int end){
        while(start < end){
            char tmp = A[start];
            A[start] = A[end];
            A[end] = tmp;
            ++start;
            --end;
        }
    }
```

首先选择需要平移的前`len`个字符，然后平移剩余字符串，最后再整个字符串进行翻转即可。

5. 拼接最小字典序

> 对于一个给定的字符串数组，请找到一种拼接顺序，使所有小字符串拼接成的大字符串是所有可能的拼接中字典序最小的。
>
> 给定一个字符串数组**strs**，同时给定它的大小，请返回拼接成的串。
>
> 测试样例：
>
> > ["abc","de"],2
> >
> > "abcde"

解法如下：

```
string findSmallest(vector<string> strs, int n) {
        if(strs.size() <= 0)
            return "";
        // 通过组合进行排序
        sort(strs.begin(),strs.end(),compare);
        string result;
        for(int i=0;i<n;++i)
            result += strs[i];
        return result;
    }

    static bool compare(const string& a, const string& b){
        string str1 = a+b;
        string str2 = b+a;
        return (str1 < str2)? true:false;
    }
```

上述解法是定义一个比较函数，对两个字符串按不同顺序拼接后进行排序，根据排序后的结果来进行递增排序。

6. 合法括号序列判断

> 对于一个字符串，请设计一个算法，判断其是否为一个合法的括号串。
>
> 给定一个字符串**A**和它的长度**n**，请返回一个bool值代表它是否为一个合法的括号串。
>
> 测试样例：
>
> > "(()())",6
> >
> > 返回：true
>
> 测试样例：
>
> > "()a()()",7
> >
> > 返回：false
>
> 测试样例：
>
> > "()(()()",7
> >
> > 返回：false

解法如下：

```
bool chkParenthesis(string A, int n) {
        // 记录左右括号数量的差值
        int num = 0;
        for(int i=0; i<n; ++i){
            if(A[i] == '(')
                ++num;
            else if(A[i] == ')')
                --num;
            else
                // 非法字符
                return false;
            if(num < 0)
                // 表示右括号多于左括号
                return false;
        }
        return (num == 0);
    }
```

使用一个整型变量记录左右括号数量的差值，根据差值判断是否合法。

7. 空格替换

> 请编写一个方法，将字符串中的空格全部替换为“%20”。假定该字符串有足够的空间存放新增的字符，并且知道字符串的真实长度(小于等于1000)，同时保证字符串由大小写的英文字母组成。
>
> 给定一个string **iniString** 为原始的串，以及串的长度 int **len**, 返回替换后的string。
>
> 测试样例：
>
> > "Mr John Smith”,13
> >
> > 返回："Mr%20John%20Smith"
> >
> > ”Hello World”,12
> >
> > 返回：”Hello%20%20World”

解法如下：

```
class Replacement {
public:
    string replaceSpace(string iniString, int length) {
        if(iniString.size() <= 0)
            return "";
        int numBlanks=0;
        // 计算空格的数量
        for(int i=0; i<length;++i)
            if(iniString[i] == ' ')
                ++numBlanks;
        // 新长度
        int newLength = length + 2*numBlanks;
        string res(newLength,' ');
        int i = length-1, j = newLength-1;
        while(i >= 0){
            // 开始替换空格
            if(iniString[i] == ' '){
                res[j--] = '0';
                res[j--] = '2';
                res[j--] = '%';
            }else{
                res[j--] = iniString[i];
            }
            --i;
        }
        return res;
    }
};
```

需要先计算空格数量，然后新建一个新长度的字符串，并分别从两个字符串末尾开始往前遍历。

8. 最长无重复字符子串

> 对于一个字符串,请设计一个高效算法，找到字符串的最长无重复字符的子串长度。
>
> 给定一个字符串**A**及它的长度**n**，请返回它的最长无重复字符子串长度。保证A中字符全部为小写英文字符，且长度小于等于500。
>
> 测试样例：
>
> > "aabcb",5
> >
> > 返回：3

解法如下：

```
int longestSubstring(string A, int n) {
        //空间复杂度O（n），时间复杂度O（n）
        vector<int> map(26,-1);//保存当前字符出现的前一个位置
        vector<int> ans(n,0);//保存以i位置字符结尾的每最长无重复子串的长度
        int ret = 0;

        ans[0] = 1;
        map[A[0] - 'a'] = 0;
        for (int i = 1; i < n; i++) {
            // 计算当前位置的最长无重复子串是比较当前字符的上一个位置
            //和前一个字符位置得到的无重复子串长度的起始位置，取两者中最大值
            ans[i] = i - max(i - ans[i - 1], map[A[i] - 'a']+1) + 1;
            // 更新当前字符出现的最新位置
            map[A[i] - 'a'] = i;
            if (ret < ans[i])
                ret = ans[i];
        }

        return ret;
    }
```

上述解法使用了一个哈希表，用于记录当前字符出现的上个位置；再用一个数组记录以当前位置为字符结尾的最长无重复子串长度。每次循环的时候，最长无重复子串长度是根据当前字符上一个位置，以及前一个位置的最长无重复子串长度和当前位置差值，取这两个数值的最大值。

队列和栈

1. 可查询最值的栈

> 定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的min函数

解法如下：

```
class Solution {
private:
    stack<int> data;
    stack<int> min_d;
public:
    void push(int value) {
        if(min_d.size() <= 0 || value < min_d.top()){
            // 当待插入数值小于min_d栈顶元素或者min_d是空，压入min_d中
            min_d.push(value);
        }else{
            min_d.push(min_d.top());
        }
        data.push(value);
    }
    void pop() {
        if(data.size() > 0 && min_d.size() > 0){
            data.pop();
            min_d.pop();
        }
    }
    int top() {
        return data.top();
    }
    int min() {
        return min_d.top();
    }
};
```

采用两个栈实现，一个栈保存插入的数据，一个栈保存当前最小值。

2. 双栈队列

> 编写一个类,只能用两个栈结构实现队列,支持队列的基本操作(push，pop)。
>
> 给定一个操作序列**ope**及它的长度**n**，其中元素为正数代表push操作，为0代表pop操作，保证操作序列合法且一定含pop操作，请返回pop的结果序列。
>
> 测试样例：
>
> > [1,2,3,0,4,0],6
> >
> > 返回：[1,2]

解法如下：

```
class TwoStack {
private:
    stack<int> s1;
    stack<int> s2;
public:
    vector<int> twoStack(vector<int> ope, int n) {
        if(ope.size() <= 0)
            return vector<int>();
        vector<int> res;
        for(int i=0; i<n; ++i){
            if(ope[i] == 0)
                res.push_back(pop());
            else
                push(ope[i]);
        }
        return res;
    }

    void push(int val){
        // s1专门用于保存插入的数据
        s1.push(val);
    }
    int pop(){
        if(s2.empty()){
            // s2是空的时候，将s1的元素都压入s2，并弹出s2的栈顶
            while(!s1.empty()){
                int tmp = s1.top();
                s1.pop();
                s2.push(tmp);
            }
        }
        int res = s2.top();
        s2.pop();
        return res;
    }
};
```

用两个栈实现队列，其中栈1专门用来保存`push`的数值，而栈2则是用于`pop`操作，它是将栈1中的元素都压入，这样栈2弹出的顺序就是所有元素压入的顺序，实现了队列的先进先出的特性。

3. 栈的反转

> 实现一个栈的逆序，但是只能用递归函数和这个栈本身的pop操作来实现，而不能自己申请另外的数据结构。
>
> 给定一个整数数组**A**即为给定的栈，同时给定它的大小**n**，请返回逆序后的栈。
>
> 测试样例：
>
> > [4,3,2,1],4
> >
> > 返回：[1,2,3,4]

解法如下：

```
class StackReverse {
public:
    vector<int> reverseStack(vector<int> A, int n) {
        int top = n-1;
        reverse(A, top);
        return A;
    }
    int pop(vector<int>& A, int& top){
        int val = A[top--];
        return val;
    }
    void push(vector<int>& A, int& top, int val){
        A[++top] = val;
    }
    // 移除栈顶元素，并返回它
    int removeBottom(vector<int>& A, int& top){
        int res = pop(A,top);
        if(top < 0){
            return res;
        }else{
            int bottom = removeBottom(A, top);
            // 压回非栈底元素
            push(A, top, res);
            return bottom;
        }
    }
    // 反转栈
    void reverse(vector<int>& A, int& top){
        if(top < 0){
            return;
        }else{
            // 得到栈底元素
            int res = removeBottom(A, top);
            reverse(A, top);
            // 将栈底元素放到最前
            push(A, top, res);
        }
    }
};
```

上述解法主要是`removeBottom`函数和`reverse`两个函数，前者是移除栈中的栈底元素，并返回它，而其他元素则按原来顺序存放在栈中，而`reverse`函数则是实现反转的操作。

4. 双栈排序

> 请编写一个程序，按升序对栈进行排序（即最大元素位于栈顶），要求最多只能使用一个额外的栈存放临时数据，但不得将元素复制到别的数据结构中。
>
> 给定一个int[] **numbers**(C++中为vector)，其中第一个元素为栈顶，请返回排序后的栈。请注意这是一个栈，意味着排序过程中你只能访问到第一个元素。
>
> 测试样例：

> [1,2,3,4,5]
>
> 返回：[5,4,3,2,1]

解法如下：

```
class TwoStacks {
public:
    vector<int> twoStacksSort(vector<int> numbers) {
        if(numbers.size() <= 0)
            return vector<int>();
        // 使用一个临时栈，从栈底到栈顶是降序，即最小值在栈顶
        stack<int> s;
        int i=0;
        while( i<numbers.size()){
            if(s.empty() || numbers[i] >= s.top()){
                // 临时栈为空或者大于栈顶元素，则压入
                s.push(numbers[i]);
                ++i;
            }
            else{
                // 将较小的当前元素放到前一个位置
                numbers[i-1] = numbers[i];
                // 将较大的栈顶元素放到当前位置，然后从i-1位置开始再次循环
                numbers[i] = s.top();
                s.pop();
                --i;
            }
        }
        // 将排序好的临时栈依次放回原栈中
        for(int i=0; i<numbers.size(); ++i){
            numbers[i] = s.top();
            s.pop();
        }
        return numbers;
    }
};
```

上述算法利用一个辅助栈来实现对栈元素的排序。

5. 滑动窗口

> 有一个整型数组 arr 和一个大小为 w 的窗口从数组的最左边滑到最右边,窗口每次向右边滑一个位置。 返回一个长度为n-w+1的数组res，res[i]表示每一种窗口状态下的最大值。 以数组为[4,3,5,4,3,3,6,7]，w=3为例。因为第一个窗口[4,3,5]的最大值为5，第二个窗口[3,5,4]的最大值为5，第三个窗口[5,4,3]的最大值为5。第四个窗口[4,3,3]的最大值为4。第五个窗口[3,3,6]的最大值为6。第六个窗口[3,6,7]的最大值为7。所以最终返回[5,5,5,4,6,7]。
>
> 给定整形数组**arr**及它的大小**n**，同时给定**w**，请返回res数组。保证**w**小于等于**n**，同时保证数组大小小于等于500。
>
> 测试样例：
>
> [4,3,5,4,3,3,6,7],8,3

> 返回：[5,5,5,4,6,7]

解法如下：

```
vector<int> slide(vector<int> arr, int n, int w) {
        // 定义一个队列保存数组下标
        deque<int> ins;
        vector<int> res;
        if(n >= w && w >= 1){
            // 初始滑动窗口
            for(unsigned int i = 0; i<w; ++i){
                while(!ins.empty() && arr[i] >= arr[ins.back()])
                    // 队列末尾元素小于当前元素，则弹出
                    ins.pop_back();
                ins.push_back(i);
            }
            // 继续滑动窗口
            for(unsigned int i = w; i<n; ++i){
                // 保存队列头部元素
                res.push_back(arr[ins.front()]);
                while(!ins.empty() && arr[i] >= arr[ins.back()])
                    // 队列末尾元素小于当前元素，则弹出
                    ins.pop_back();
                while(!ins.empty() && ins.front() <=(int)(i-w))
                    // 队列头部超出滑动窗口范围
                    ins.pop_front();
                ins.push_back(i);
            }
            res.push_back(arr[ins.front()]);
        }
        return res;
    }
```

利用一个队列保存滑动窗口中元素的下标，这个元素是当前窗口中的最大值以及可能成为最大值的结果。因此，队列首部必然是最大值，而末尾元素如果小于新进入的元素，则会被弹出队列中。滑动过程还要注意队列头部的下标是否超过滑动窗口范围，需要及时弹出。同时考虑最后一个窗口的最大值要记得保存。

6. 数组变树

> 对于一个没有重复元素的整数数组，请用其中元素构造一棵MaxTree，MaxTree定义为一棵二叉树，其中的节点与数组元素一一对应，同时对于MaxTree的每棵子树，它的根的元素值为子树的最大值。现有一建树方法，对于数组中的每个元素，其在树中的父亲为数组中它左边比它大的第一个数和右边比它大的第一个数中更小的一个。若两边都不存在比它大的数，那么它就是树根。请设计O(n)的算法实现这个方法。
>
> 给定一个无重复元素的数组**A**和它的大小**n**，请返回一个数组，其中每个元素为原数组中对应位置元素在树中的父亲节点的编号，若为根则值为-1。
>
> 测试样例：
>
> [3,1,4,2],4
>
> 返回：[2,0,-1,2]

解法如下：

```
class MaxTree {
public:
    vector<int> buildMaxTree(vector<int> A, int n) {
        // 定义两个数组分别存放左边第一个大和右边第一个大的数
        vector<int> left(n);
        vector<int> right(n);
        vector<int> res;
        // 两个栈，用来辅助寻找左右第一个比当前元素大的数值
        stack<int> s1;
        stack<int> s2;
        for(int i=0; i<n; ++i){
            // 寻找左边第一个比A[i]大的元素
            while(!s1.empty() && A[i] > A[s1.top()]){
                // 当前元素大于栈顶元素，则弹出栈顶
                s1.pop();
            }
            if(!s1.empty()){
                // 栈顶元素就是左边第一个大的元素
                left[i] = s1.top();
            }else{
                // 否则，左边没有找到大于自己的元素
                left[i] = -1;
            }
            s1.push(i);
        }
        // 接下来寻找右边第一个比自己大的数
        for(int i=n-1; i>=0; --i){
            while(!s2.empty() && A[i] > A[s2.top()]){
                // 当前元素大于栈顶元素，则弹出栈顶
                s2.pop();
            }
            if(!s2.empty()){
                // 栈顶元素就是右边第一个大的元素
                right[i] = s2.top();
            }else{
                // 否则，右边没有找到大于自己的元素
                right[i] = -1;
            }
            s2.push(i);
        }
        // 进行比较，确定每个数组位置上父结点的位置
        for(int i=0; i<n; ++i){
            // 对比左右两个数，哪个更小
            if(left[i] == -1 && right[i] == -1)
                res.push_back(-1);
            else if(left[i] == -1)
                res.push_back(right[i]);
            else if(right[i] == -1)
                res.push_back(left[i]);
            else{
                int ind = (A[left[i]] <= A[right[i]])? left[i]:right[i];
                res.push_back(ind);

            }
```

```
        }
        return res;
    }
};
```

这道题目需要找到当前元素左右两边第一个大于自己的元素，分别用两个栈进行辅助，当前元素如果小于栈顶元素，则压入栈中，否则就一直弹出栈顶元素，直到找到大于当前元素的栈顶元素为止。最后是比较左右两边的元素，取最小值的下标。

链表

1. 环形链表插值

> 有一个整数val，如何在节点值有序的环形链表中插入一个节点值为val的节点，并且保证这个环形单链表依然有序。
>
> 给定链表的信息，及元素的值**A**及对应的**nxt**指向的元素编号同时给定**val**，请构造出这个环形链表，并插入该值。
>
> 测试样例：
>
> > [1,3,4,5,7],[1,2,3,4,0],2
> >
> > 返回：{1,2,3,4,5,7}

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class InsertValue {
public:
    ListNode* insert(vector<int> A, vector<int> nxt, int val) {
        // write code here
        if(A.size() == 0){
            ListNode* res = new ListNode(val);
            res->next = res;
            return res;
        }
        if(A.size() == 1){
            ListNode* tmp1 = new ListNode(val);
            ListNode* tmp2 = new ListNode(A[0]);
            if(A[0] > val){
                tmp1->next = tmp2;
                tmp2->next = tmp1;
                return tmp1;
            }
            else{
                tmp2->next = tmp1;
                tmp1->next = tmp2;
                return tmp2;
            }
        }
        ListNode* head = new ListNode(A[0]);
        ListNode* build = head;
        int i = 1;
        int len = A.size();
        //构建链表
        while(i < len){
            ListNode* tmp = new ListNode(A[i++]);
            build->next = tmp;
            build = build->next;
        }
        build->next = NULL;

        ListNode* pre = head;
        ListNode* cur = head->next;
        while(1){
            if(val >= pre->val && val <= cur->val){
                ListNode* tmp = new ListNode(val);
                pre->next = tmp;
                tmp->next = cur;
                return head;
            }
            pre = pre->next;
            cur = cur->next;

            if(cur == NULL)
```

```
                break;
        }
        if(val > pre->val){
            ListNode* tmp = new ListNode(val);
            pre->next = tmp;
            tmp->next = NULL;
        }
        if(val <= head->val){
            ListNode* tmp = new ListNode(val);
            //pre->next = NULL;
            tmp->next = head;
            return tmp;
        }
        return head;
    }
};
```

循环链表，首先要判断链表是否为空，如果是空，根据插入值构建的结点就直接指向自己，并返回；如果不空，就构建链表，并找到合适的位置插入，这里可能会出现没有找到位置，说明插入值比链表的数值要小或者大，如果是前者，注意返回的是插入值构建的结点，因为它是链表的头节点。

2. 删除链表的单结点

> 实现一个算法，删除单向链表中间的某个结点，假定你只能访问该结点。
>
> 给定带删除的节点，请执行删除操作，若该节点为尾节点，返回false，否则返回true

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class Remove {
public:
    bool removeNode(ListNode* pNode) {
        if(pNode->next == NULL){
            // 尾结点返回false
            return false;
        }else{
            // 复制下个结点，并删除下一个结点
            ListNode* deleteNode = pNode->next;
            pNode->val = deleteNode->val;
            pNode->next = deleteNode->next;
            delete deleteNode;
            deleteNode = NULL;
        }
        return true;
    }
};
```

$O(1)$时间删除给点结点，其实就是复制下个结点的内容到当前结点，然后删除下一个结点，这要求当前结点不是尾结点。

3. 链表的分化

> 对于一个链表，我们需要用一个特定阈值完成对它的分化，使得小于等于这个值的结点移到前面，大于该值的结点在后面，同时保证两类结点内部的位置关系不变。
>
> 给定一个链表的头结点**head**，同时给定阈值**val**，请返回一个链表，使小于等于它的结点在前，大于等于它的在后，保证结点值不重复。
>
> 测试样例：
>
> > {1,4,2,5},3
> >
> > {1,2,4,5}

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class Divide {
public:
    ListNode* listDivide(ListNode* head, int val) {
        if(head == NULL || head->next == NULL)
            return NULL;
        ListNode *minHead = NULL, *minTail = NULL;
        ListNode *maxHead = NULL, *maxTail = NULL;
        ListNode* pNode = head;
        while(pNode != NULL){
            if(pNode->val <= val){
                if(minHead == NULL){
                    minHead = pNode;
                    minTail = pNode;
                }else{
                    minTail->next = pNode;
                    minTail = pNode;
                }
            }
            if(pNode->val > val){
                if(maxHead == NULL){
                    maxHead = pNode;
                    maxTail = pNode;
                }else{
                    maxTail->next = pNode;
                    maxTail = pNode;
                }
            }
            pNode = pNode->next;
        }
        if(!minHead)
            // 只有大于val的数
            return maxHead;
        if(!maxHead)
            return minHead;
        maxTail->next = NULL;
        minTail->next = maxHead;
        return minHead;
    }
};
```

这是将链表分成两部分，一部分是小于等于阈值的链表，一部分是大于阈值的链表，然后再合并。

4. 打印两个链表的公共值

> 现有两个升序链表，且链表中均无重复元素。请设计一个高效的算法，打印两个链表的公共值部分。

> 给定两个链表的头指针**headA**和**headB**，请返回一个vector，元素为两个链表的公共部分。请保证返回数组的升序。两个链表的元素个数均小于等于500。保证一定有公共值
>
> 测试样例：
>
> > {1,2,3,4,5,6,7},{2,4,6,8,10}
>
> > 返回：[2.4.6]

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class Common {
public:
    vector<int> findCommonParts(ListNode* headA, ListNode* headB) {
        if(headA == NULL || headB == NULL)
            return vector<int>();
        vector<int> res;
        while(headA != NULL && headB != NULL){
            if(headA->val < headB->val){
                headA = headA->next;
            }else if(headA->val > headB->val){
                headB = headB->next;
            }else{
                res.push_back(headA->val);
                headA = headA->next;
                headB = headB->next;
            }
        }
        return res;
    }
};
```

主要就是相互比较，如果相等就保存下来，否则就让值更小的链表往后移动一步。

5. 链表的k逆序

> 有一个单链表，请设计一个算法，使得每K个节点之间逆序，如果最后不够K个节点一组，则不调整最后几个节点。例如链表1->2->3->4->5->6->7->8->null，K=3这个例子。调整后为，3->2->1->6->5->4->7->8->null。因为K==3，所以每三个节点之间逆序，但其中的7，8不调整，因为只有两个节点不够一组。
>
> 给定一个单链表的头指针**head**,同时给定K值，返回逆序后的链表的头指针。

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class KInverse {
public:
    ListNode* inverse(ListNode* head, int k) {
        stack<int> vals;
        ListNode* pNode = head;
        ListNode* newHead = head;
        if(head == NULL || k < 2)
            return head;
        int i=0;
        while(pNode != NULL){
            ++i;
            // 使用栈保存结点数值
            vals.push(pNode->val);
            if(i==k){
                while(!vals.empty()){
                    // 依次弹出栈中数值，相当于做了逆序
                    newHead->val = vals.top();
                    vals.pop();
                    newHead = newHead->next;
                }
                i = 0;
            }
            pNode = pNode->next;
        }
        return head;
    }
};
```

这是使用栈来保存结点数值，每次遍历k个数值后，就弹出栈中数值，此时就相当于做了逆序操作。

最优解如下：

```
class KInverse {
public:
    ListNode* inverse(ListNode* head, int k) {
        ListNode* prehead=new ListNode(0);//申请新的节点，避免边界问题
        prehead->next=head;
        ListNode* begin=head;
        ListNode* lastkend=prehead;//每k个元素为一组，中间需要链接，当前链接尾
        ListNode* khead;//每组的头结点
        ListNode* kend;//每组的尾节点
        ListNode* nextbegin;//下一组的头结点
        if(!head)
            return NULL;
        if(k<=1)
            return head;
        while(1){
        int i=1;
        while(i%k!=0&&head){
        head=head->next;
        ++i;
        }       //head为尾节点，如果尾节点为空，则不用翻转
        if(i%k==0&&head)
        {
        nextbegin=head->next;//下一组的头节点必须保存下，要不然翻转之后head->next不再是下一组头
        inverse_k(begin,k,khead,kend);//组内翻转
        lastkend->next=khead;//将翻转好的组加入链中
        lastkend=kend;//新的链末尾
        begin=nextbegin;//下一组开始
        head=nextbegin;//新的头
        }
        else
        {
        lastkend->next=nextbegin; //如果头为空，不翻转直接链接
        return prehead->next;
        }
        }
        return NULL;
    }
    void inverse_k(ListNode* head,int k,ListNode* &khead,ListNode* &kend)
        {
        ListNode* prehead=new ListNode(0);
        ListNode* pre=prehead;
        prehead->next=head;
        ListNode* next=head->next;

        while(k>2)
            {
            head->next=pre;
            pre=head;
            head=next;
            next=next->next;
            --k;
        }
        //翻转一个链需要借助三个临时变量才可以完成
```

```
            next->next=head;
            head->next=pre;
            khead=next;
            kend=prehead->next;
            kend->next=NULL;
            delete prehead;
            prehead=NULL;
        }
    };
```

这是不需要辅助空间的做法。

6. 链表指定值删除

> 现在有一个单链表。链表中每个节点保存一个整数，再给定一个值val，把所有等于val的节点删掉。
>
> 给定一个单链表的头结点**head**，同时给定一个值**val**，请返回清除后的链表的头结点，保证链表中有不等于该值的其它值。请保证其他元素的相对顺序。
>
> 测试样例：
>
> > {1,2,3,4,3,2,1},2
> >
> > {1,3,4,3,1}

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class ClearValue {
public:
    ListNode* clear(ListNode* head, int val) {
        if(head == NULL)
            return NULL;
        while(head != NULL && head->val == val)
            // 删除头结点是指定值的情况
            head = head->next;
        ListNode* pre = head;
        ListNode* node = head->next;
        while(node != NULL){
            if(node->val == val){
                pre->next = node->next;
                node = pre->next;
            }else{
                pre = node;
                node = node->next;
            }
        }
        return head;
    }

};
```

这里要注意头结点是否等于指定值的情况。

7. 链表的回文结构

> 请编写一个函数，检查链表是否为回文。
>
> 给定一个链表ListNode* **pHead**，请返回一个bool，代表链表是否为回文。
>
> 测试样例：
>
> > {1,2,3,2,1}
> >
> > 返回：true
> >
> > {1,2,3,2,3}
> >
> > 返回：false

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class Palindrome {
public:
    bool isPalindrome(ListNode* pHead) {
        if(pHead == NULL || pHead->next == NULL)
            return false;
        ListNode *p = NULL;
        ListNode *q = pHead;
        while(pHead){
            if(!p) p = new ListNode(pHead->val);
            else{
                ListNode *tmp = new ListNode(pHead->val);
                tmp->next = p;
                p = tmp;
            }
            pHead = pHead->next;
        }
         while(q){
            if(q->val != p->val) return false;
            q = q->next;
            p = p->next;
        }
        return true;
    }
};
```

上述解法是新建立一个链表，是原链表的反转顺序，然后依次比较。另一种借助栈的解法如下：

```
bool isPalindrome(ListNode* pHead) {
        // write code here
        stack<int> s;
        ListNode* p = pHead;
        while (p != NULL){
            s.push(p->val);
            p = p->next;
        }
        p = pHead;
        while (p != NULL){
            if (s.top() != p->val)    return false;
            s.pop();
            p = p->next;
        }
        return true;
    }
```

8. 复杂链表的复制

> 输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点）。

解法如下：

```
/*
struct RandomListNode {
    int label;
    struct RandomListNode *next, *random;
    RandomListNode(int x) :
            label(x), next(NULL), random(NULL) {
    }
};
*/
class Solution {
public:
    RandomListNode* Clone(RandomListNode* pHead)
    {
        cloneFirst(pHead);
        cloneRandom(pHead);
        return splitList(pHead);
    }
    // 先复制链表的数值和next指针，并插入到每个结点后面
    void cloneFirst(RandomListNode* head){
        RandomListNode* node = head;
        while(node != NULL){
            RandomListNode* clone = new RandomListNode(node->label);
            clone->next = node->next;
            // 将复制的结点先插入到原结点后面
            node->next = clone;
            // 遍历到下个结点
            node = clone->next;
        }
    }
    // 复制结点的随机结点
    void cloneRandom(RandomListNode* head){
        RandomListNode* node = head;
        while(node != NULL){
            RandomListNode* clone = node->next;
            if(node->random != NULL)
                clone->random = node->random;
            node = clone->next;
        }
    }
    // 拆分链表，变成原始链表和复制得到的链表
    RandomListNode* splitList(RandomListNode* head){
        RandomListNode* node = head;
        // 复制链表的头结点和遍历结点
        RandomListNode* cloneHead = NULL;
        RandomListNode* cloneNode = NULL;
        if(node != NULL){
            cloneHead = cloneNode = node->next;
            // 原结点指向原始的结点，而非其复制结点
            node->next = cloneNode->next;
            node = node->next;
        }
        while(node != NULL){

            cloneNode->next = node->next;
```

```
            cloneNode = cloneNode->next;
            node->next = cloneNode->next;
            node = node->next;
        }
        return cloneHead;
    }
};
```

先复制每个结点，并插入到原结点后面，然后再复制结点的`random`结点，这就完成复制结点的复制，之后就是将原结点和复制结点分开，变成两个链表。

9. 链表判环

> 如何判断一个单链表是否有环？有环的话返回进入环的第一个节点的值，无环的话返回-1。如果链表的长度为N，请做到时间复杂度O(N)，额外空间复杂度O(1)。
>
> 给定一个单链表的头结点**head**（注意另一个参数**adjust**为加密后的数据调整参数，方便数据设置，与本题求解无关)，请返回所求值。

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class ChkLoop {
public:
    int chkLoop(ListNode* head, int adjust) {
        ListNode *meetNode = check(head);
        if(meetNode == NULL)
            return -1;
        // 计算环中的节点数量
        ListNode *tmp = meetNode;
        int loopNum = 1;
        while(tmp->next != meetNode){
            tmp = tmp->next;
            ++loopNum;
        }
        tmp = head;
        // 先走loopNum步
        for(int i=0; i<loopNum;++i)
            tmp = tmp->next;
        ListNode *node = head;
        while(node != tmp){
            node = node->next;
            tmp = tmp->next;
        }
        return node->val;
    }
    // 判断是否有环
    ListNode* check(ListNode* head){
        ListNode *fast = head;
        ListNode *slow = head;
        while(fast != NULL){
            slow = slow->next;
            fast = fast->next;
            if(fast != NULL)
                fast = fast->next;
            else
                return NULL;
            if(fast == slow)
                break;
        }
        return fast;
    }
};
```

实际上一个更简洁的解法如下：

```
int chkLoop(ListNode* head, int adjust) {
        // write code here
        ListNode* slow = head;
        ListNode* fast = head;
        bool flag = false;
        while(fast&&fast->next){
            slow = slow->next;
            //个数为奇数的情况
            if(fast->next){
                fast = fast->next->next;
            }
            if(slow==fast){
                flag =true;
                break;
            }
        }
        if(!flag){
            return -1;
        }
        //找第一个入环的节点
        fast = head;
        while(slow!=fast){
            slow = slow->next;
            fast = fast->next;
        }
        return fast->val;

    }
```

10. 无环链表判相交

> 现在有两个无环单链表，若两个链表的长度分别为m和n，请设计一个时间复杂度为O(n + m)，额外空间复杂度为O(1)的算法，判断这两个链表是否相交。
>
> 给定两个链表的头结点**headA**和**headB**，请返回一个bool值，代表这两个链表是否相交。保证两个链表长度小于等于500。

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class CheckIntersect {
public:
    bool chkIntersect(ListNode* headA, ListNode* headB) {
        int a=0, b=0;
        // 计算两个链表长度
        ListNode* na = headA;
        ListNode* nb = headB;
        while(na != NULL){
            na = na->next;
            ++a;
        }
        while(nb != NULL){
            nb = nb->next;
            ++b;
        }
        ListNode* fast = headA;
        ListNode* slow = headB;
        int n = a-b;
        if(b > a){
            fast = headB;
            slow = headA;
            n = b-a;
        }
        // 更长的链表先走相差长度的步数
        for(int i=0; i<n; ++i)
            fast = fast->next;
        while(slow != NULL && fast != NULL){
            if(slow == fast)
                return true;
            slow = slow->next;
            fast = fast->next;
        }
        return false;
    }
};
```

上述解法是分别求两个链表长度，然后长的链表先走长度差值的步数，然后再同时走，这样如果有相交，才会判断得到。

下面是一个更简洁的解法，两个链表分别走到最后一个结点，如果相交，那么最后的结点必然相等。

```
bool chkIntersect(ListNode* headA, ListNode* headB) {
        while(headA->next)
            headA = headA->next;
        while(headB->next)
            headB = headB->next;
        if(headA->val == headB->val)
            return true;
        return false;
    }
```

11. 有环单链表判断相交

> 如何判断两个有环单链表是否相交？相交的话返回第一个相交的节点，不想交的话返回空。如果两个链表长度分别为N和M，请做到时间复杂度O(N+M)，额外空间复杂度O(1)。
>
> 给定两个链表的头结点**head1**和**head2**(注意，另外两个参数**adjust0**和**adjust1**用于调整数据,与本题求解无关)。请返回一个bool值代表它们是否相交。

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class ChkIntersection {
public:
     bool chkInter(ListNode* head1, ListNode* head2, int adjust0, int adjust1) {
        auto entry1 = get_loop_entry(head1);
        auto entry2 = get_loop_entry(head2);
        //均为空或有一个为空，则说明有链表为无环节点
        if (!entry1 || !entry2) return false;
         // 情况一
        if (entry1 == entry2) return true;
         // 情况二，接着往后遍历，若能遇到2的第一个入环节点则说明相交
        auto p = entry1->next;
        while (p != entry1) {
            if (p == entry2) return true;
            p = p->next;
        }
        return false;
    }
private:
    //因为是有环链表  所以一定存在入环结点
    ListNode* get_loop_entry(ListNode *head) {
        //nullptr也是空指针
        if (!head) return nullptr;
        ListNode *fast = head;
        ListNode *slow = head;
        while (fast->next && fast->next->next) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                // 找到入环结点
                fast = head;
                while (fast != slow) {
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }
        }
        return nullptr;
    }
};
```

首先寻找两个链表的入环结点，如果相等，说明相交，并且相交结点可能在入环结点的前面；如果不相等，则从其中一个入环结点开始寻找，在环中如果能遇到另一个入环结点，说明相交，否则就不相交。

12. 单链表相交判断

> 给定两个单链表的头节点head1和head2，如何判断两个链表是否相交？相交的话返回true，不想交的话返回false。
>
> 给定两个链表的头结点**head1**和**head2**(注意，另外两个参数**adjust0**和**adjust1**用于调整数据,与本题求解无关)。请返回一个bool值代表它们是否相交。

解法如下：

```
/*
struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};*/
class ChkIntersection {
public:
    bool chkInter(ListNode* head1, ListNode* head2, int adjust0, int adjust1) {
        ListNode* entry1 = get_loop_entry(head1);
        ListNode* entry2 = get_loop_entry(head2);
        if((!entry1 && entry2) || (entry1 && !entry2))
            // 一个有环，一个无环，肯定不相交
            return false;
        if(!entry1 && !entry2){
            // 无环链表判断相交的情况
            while(head1->next)
                head1 = head1->next;
            while(head2->next)
                head2 = head2->next;
            if(head1->val == head2->val)
                return true;
            return false;
        }
        // 最后一种情况就是两者都是有环链表
        if(entry1 == entry2)
            return true;
        ListNode* tmp = entry1->next;
        while(tmp != entry1){
            if(tmp == entry2)
                return true;
            tmp = tmp->next;
        }
        return false;
    }
    // 寻找可能存在的入环结点
    ListNode* get_loop_entry(ListNode *head) {
        //nullptr也是空指针
        if (!head) return nullptr;
        ListNode *fast = head;
        ListNode *slow = head;
        while (fast->next && fast->next->next) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                fast = head;
                while (fast != slow) {
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }

        }
```

```
        return nullptr;
    }
};
```

首先是寻找可能存在的入环结点，这里就分3种情况，一个有环和一个无环是肯定不相交，第二种是都无环，那就按照无环链表判断相交的方法，最后就是都有环，那么就按照有环链表判断相交的方法。

二分搜索

1. 局部最小值位置

> 定义局部最小的概念。arr长度为1时，arr[0]是局部最小。arr的长度为N(N>1)时，如果arr[0]<arr[1]，那么arr[0]是局部最小；如果arr[N-1]<arr[N-2]，那么arr[N-1]是局部最小；如果0<i<N-1，既有arr[i]<arr[i-1]又有arr[i]<arr[i+1]，那么arr[i]是局部最小。 给定无序数组arr，已知arr中任意两个相邻的数都不相等，写一个函数，只需返回arr中任意一个局部最小出现的位置即可。

解法如下：

```
class Solution {
public:
    int getLessIndex(vector<int> arr) {
         if(arr.size() <=0 )
            return -1;
        if(arr.size() == 1 || arr[0] < arr[1])
            return 0;
        int lens = arr.size();
        if(arr[lens-1] < arr[lens-2])
            return lens-1;
        int left = 0, right = lens-1;

        while(left < right){
            // 防止left+right溢出的情况
             int mid = left + (right-left)/2;
            if(arr[mid] < arr[mid-1] && arr[mid] < arr[mid+1])
                return mid;
            else if(arr[mid] < arr[mid-1])
                left = mid+1;
            else
                right = mid-1;
        }
        return -1;
    }
};
```

使用二分搜索，判断中间值跟相邻两个数值的大小比较，如果小于左值，则应该往右半部分查找；如果小于右值，则往左半部分查找；如果两个数值都小于，那么就返回当前中间值的位置。

2. 元素最左出现

> 对于一个有序数组arr，再给定一个整数num，请在arr中找到num这个数出现的最左边的位置。

> 给定一个数组**arr**及它的大小**n**，同时给定**num**。请返回所求位置。若该元素在数组中未出现，请返回-1。
>
> 测试样例：
>
> > [1,2,3,3,4],5,3
> >
> > 返回：2

解法如下：

```
class LeftMostAppearance {
public:
    int findPos(vector<int> arr, int n, int num) {
        if(n <= 0 || (n==1 && arr[0] != num))
            return -1;
        if(n==1 && arr[0] == num)
            return 0;

        int left = 0, right = n-1;
        while(left < right){
            int mid = left + (right - left)/2;
            if(arr[mid] >= num)
                right = mid;
            else if(arr[mid] < num && arr[mid+1] != num)
                left = mid;
            else if(arr[mid] < num && arr[mid+1] == num)
                return mid+1;
        }
        if(left == right && arr[right] == num)
            return right;
        return -1;
    }
};
```

3. 循环有序数组最小值

> 对于一个有序循环数组arr，返回arr中的最小值。有序循环数组是指，有序数组左边任意长度的部分放到右边去，右边的部分拿到左边来。比如数组[1,2,3,3,4]，是有序循环数组，[4,1,2,3,3]也是。
>
> 给定数组**arr**及它的大小**n**，请返回最小值。
>
> 测试样例：
>
> > [4,1,2,3,3],5
> >
> > 返回：1

解法如下：

```
class MinValue {
public:
    int getMin(vector<int> arr, int n) {
        if(n <= 0)
            return -1;
        int left = 0, right = n-1;
        int mid = left;
        while(arr[left] >= arr[right]){
            if(right-left == 1){
                mid = right;
                break;
            }
            mid = left+(right-left)/2;
            if(arr[left] == arr[mid] && arr[right] == arr[mid])
                // 只能顺序查找
                return minOrder(arr, left, right);
            if(arr[mid] >= arr[left])
                left = mid;
            else if(arr[mid] <= arr[right])
                right = mid;
        }
        return arr[mid];
    }
    int minOrder(vector<int> arr, int left, int right){
        int result = arr[left];
        for(int j=left+1; j<=right; ++j)
            result = (result < arr[j])? result:arr[j];
        return result;
    }
};
```

上述解法也是利用二分搜索，定义两个指针，分别指向数组两端位置，判断其跟中间值的大小，来缩小搜索范围，注意如果三者相等，那么就只能通过顺序查找得到最小值。

4. 最左原位

> 有一个有序数组arr，其中不含有重复元素，请找到满足arr[i]==i条件的最左的位置。如果所有位置上的数都不满足条件，返回-1。
>
> 给定有序数组**arr**及它的大小**n**，请返回所求值。
>
> 测试样例：
>
> > [-1,0,2,3],4
>
> > 返回：2

解法如下：

```
class Find {
public:
    int findPos(vector<int> arr, int n) {
        int res = -1;
        if(n <= 0)
            return res;
        if(arr[0] >= n-1 || arr[n-1] < 0)
            return res;
        int left = 0, right = n-1;
        int mid = 0;
        while(left <= right){
            mid = left + (right - left)/2;
            if(arr[left] == left)
                return left;
            if(arr[mid] < mid)
                left = mid+1;
            else if(arr[mid] > mid)
                right = mid-1;
            else{
                // 记录位置，并继续往左半部分查找
                res = mid;
                right = mid-1;
            }
        }
        return res;
    }
};
```

5. 快速N次方

> 如果更快的求一个整数k的n次方。如果两个整数相乘并得到结果的时间复杂度为O(1)，得到整数k的N次方的过程请实现时间复杂度为O(logN)的方法。
>
> 给定**k**和**n**，请返回k的n次方，为了防止溢出，请返回结果Mod 1000000007的值。
>
> 测试样例：
>
> > 2,3
> >
> > 返回：8

解法如下：

```
class QuickPower {
public:
    int getPower(int k, int N) {
        int mod = 1000000007;
        if(N == 0)
            return 1;
        if(N == 1)
            return k;
        long long result = getPower(k, N >> 1)%mod;
        result = (result*result)%mod;
        if(N & 0x1 == 1)
            // 指数是奇数
            result = result*(long long)k%mod;
        return (int)result;
    }
};
```

这也是利用了二分搜索的思想。

二叉树

1. 递归二叉树的序列打印

> 请用递归方式实现二叉树的先序、中序和后序的遍历打印。
>
> 给定一个二叉树的根结点**root**，请依次返回二叉树的先序，中序和后续遍历(二维数组的形式)。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class TreeToSequence {
public:
    vector<vector<int> > convert(TreeNode* root) {
        if(root == NULL)
            return vector<vector<int> >();
        vector<vector<int> > result;
        vector<int> tmp;
        TreeNode *node = root;
        preOrder(node, tmp);
        result.push_back(tmp);

        node = root;
        tmp.clear();
        inOrder(node, tmp);
        result.push_back(tmp);

        node = root;
        tmp.clear();
        postOrder(node,tmp);
        result.push_back(tmp);
        return result;
    }
    void preOrder(TreeNode* root, vector<int>& res){
        // 先序遍历
        if(root == NULL)
            return;
        res.push_back(root->val);
        preOrder(root->left, res);
        preOrder(root->right, res);
    }
    void inOrder(TreeNode* root, vector<int>& res){
        // 中序遍历
        if(root == NULL)
            return;
        inOrder(root->left, res);
        res.push_back(root->val);
        inOrder(root->right, res);
    }
    void postOrder(TreeNode* root, vector<int>& res){
        // 后序遍历
        if(root == NULL)
            return;

        postOrder(root->left, res);
```

```
        postOrder(root->right, res);
        res.push_back(root->val);
    }
};
```

2. 非递归二叉树的序列打印

> 请用非递归方式实现二叉树的先序、中序和后序的遍历打印。
>
> 给定一个二叉树的根结点**root**，请依次返回二叉树的先序，中序和后续遍历(二维数组的形式)。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class TreeToSequence {
public:
    vector<vector<int> > convert(TreeNode* root) {
        if(root == NULL)
            return vector<vector<int> >();
        vector<vector<int> > result;
        vector<int> v1;
        preOrder(root,v1);
        result.push_back(v1);
        v1.clear();

        inOrder(root, v1);
        result.push_back(v1);
        v1.clear();

        postOrder(root,v1);
        result.push_back(v1);

        return result;
    }
    // 先序打印
    void preOrder(TreeNode* root, vector<int>& res){
        if(root == NULL)
            return;
        stack<TreeNode* > nodes;
        auto p = root;
        while(p || !nodes.empty()){
            while(p){
                // 不断遍历左子结点,并打印
                res.push_back(p->val);
                nodes.push(p);
                p = p->left;
            }
            if(!nodes.empty()){
                // 弹出栈顶，遍历方向转向右子结点
                p = nodes.top();
                nodes.pop();
                p = p->right;
            }
        }
    }
    // 中序打印

    void inOrder(TreeNode* root, vector<int>& res){
```

```
    if(root == NULL)
        return;
    stack<TreeNode*> nodes;
    auto p = root;
    while(p || !nodes.empty()){
        while(p){
            // 不断遍历左子结点
            nodes.push(p);
            p = p->left;
        }
        if(!nodes.empty()){
            // 弹出栈顶，遍历方向变成右子结点
            p = nodes.top();
            nodes.pop();
            res.push_back(p->val);
            p = p->right;
        }
    }
}
// 后序打印1
void postOrder(TreeNode* root, vector<int>& res){
    if(root==NULL)
        return;
    // 使用两个栈
    stack<TreeNode*> s1;
    stack<TreeNode*> s2;
    s1.push(root);
    while(!s1.empty()){
        // 每次都弹出s1栈顶，并压入s2中
        TreeNode *tmp = s1.top();
        s1.pop();
        s2.push(tmp);
        // 依次压入当前结点的左右子结点
        if(tmp->right)
            s1.push(tmp->right);
        if(tmp->left)
            s1.push(tmp->left);
    }
    // s2的弹出顺序就是后序遍历顺序
    while(!s2.empty()){
        TreeNode* tmp = s2.top();
        s2.pop();
        res.push_back(tmp->val);
    }
}
// 后序打印2
void postOrder2(TreeNode* root, vector<int>& ret){
    if (!root) return;
    stack<TreeNode*> st;
    st.push(root);
    // pre表示上一个打印的结点
    TreeNode* pre = nullptr;

    // 栈顶结点
```

```
        TreeNode* cur = nullptr;
        while (!st.empty()) {
            cur = st.top();
            if ((!cur->left && !cur->right)
                || (pre && (pre == cur->left || pre == cur->right))) {
                // 当满足当前结点没有左右子结点或者上一个打印的结点是当前结点的左或右子结点
                ret.push_back(cur->val);
                st.pop();
                pre = cur;
            } else {
                if (cur->right) st.push(cur->right);
                if (cur->left) st.push(cur->left);
            }
        }
    }
};
```

先序遍历的话，首先压入根结点，并打印，然后每次遍历左子结点，并打印，然后当左子结点为空，则压入当前结点的右子结点，并继续先前的步骤，如果右子结点是空，那么就会继续弹出栈顶结点，再来判断他是否有右子结点，重复上述步骤了。

中序遍历，则每次遍历左子结点，但是不打印，直到遍历到最左的一个子结点，发现当前结点左子结点为空，那么弹出栈顶并打印，然后判断是否有右子结点，如果有，就压入它，没有就再弹出栈顶并打印，继续判断其右子结点是否存在，重复上述步骤。

后序遍历，有两种方法，第一种用两个栈，先压入根结点到第一个栈，每次循环的时候弹出栈顶，压入到第二个栈中，然后先后压入左、右两个子结点到第一个栈中，并重复上述步骤。最后栈1是空的，栈2中弹出顺序就是后序遍历的顺序了。

第二种方法只需要一个栈，定义两个变量，一个表示当前栈顶结点，一个表示上一个打印的结点。如果满足当前结点是叶子结点，也就是没有左右子结点，或者上个打印结点是当前结点的左子结点或者右子结点，那么就打印当前结点，并弹出。每次循环，都让指向栈顶结点的变量指向当前栈顶结点。如果不满足上述条件，那么就先压入右子结点，再压入左子结点。当然首先是要压入根结点。

3. 二叉树的打印

> 有一棵二叉树，请设计一个算法，按照层次打印这棵二叉树。
>
> 给定二叉树的根结点**root**，请返回打印结果，结果按照每一层一个数组进行储存，所有数组的顺序按照层数从上往下，且每一层的数组内元素按照从左往右排列。保证结点数小于等于500。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class TreePrinter {
public:
    vector<vector<int> > printTree(TreeNode* root) {
        // write code here
        if(root == NULL)
            return vector<vector<int> >();
        vector<vector<int> > res;
        queue<TreeNode*> qt;
        qt.push(root);
        // 分别定义变量记录当前层和下一层待打印结点
        int current = 1;
        int nextLevel = 0;
        vector<int> tmp;
        while(!qt.empty()){
            TreeNode* n = qt.front();
            qt.pop();
            tmp.push_back(n->val);
            if(n->left){
                qt.push(n->left);
                ++nextLevel;
            }
            if(n->right){
                qt.push(n->right);
                ++nextLevel;
            }
            --current;

            if(current==0){
                // 打印完当前层结点
                res.push_back(tmp);
                tmp.clear();
                current = nextLevel;
                nextLevel = 0;
            }
        }
        return res;
    }
};
```

使用一个队列保存结点，然后定义两个变量分别表示当前层待打印的结点数量和下一层需要打印的结点数量。

4. 二叉树的序列化

> 首先我们介绍二叉树先序序列化的方式，假设序列化的结果字符串为str，初始时str等于空字符串。先序遍历二叉树，如果遇到空节点，就在str的末尾加上“#!”，“#”表示这个节点为空，节点值不存在，当然你也可以用其他的特殊字符，“!”表示一个值的结束。如果遇到不为空的节点，假设节点值为3，就在str的末尾加上“3!”。现在请你实现树的先序序列化。
>
> 给定树的根结点**root**，请返回二叉树序列化后的字符串。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class TreeToString {
public:
    string toString(TreeNode* root) {
        if(root == NULL)
            return "#!";
        string s;
        s = to_string(root->val) + '!';
        s += toString(root->left);
        s += toString(root->right);
        return s;
    }

};
```

上述是递归版本的解法。

非递归版本如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class TreeToString {
public:
    string toString(TreeNode* root) {
        if(root == NULL)
            return "#!";
        string s;
        stack<TreeNode*> nodes;
        auto p = root;
        while(p || !nodes.empty()){
            while(p){
                nodes.push(p);
                s += to_string(p->val) + '!';
                p = p->left;
                if(!p)
                    s += "#!";
            }
            if(!nodes.empty()){
                p = nodes.top();
                nodes.pop();
                p = p->right;
                if(!p)
                    s += "#!";
            }
        }
        return s;
    }
};
```

5. 平衡二叉树判断

> 有一棵二叉树，请设计一个算法判断这棵二叉树是否为平衡二叉树。
>
> 给定二叉树的根结点**root**，请返回一个bool值，代表这棵树是否为平衡二叉树。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class CheckBalance {
public:
    bool check(TreeNode* root) {
        if(root == NULL)
            return true;
        int h=0;
        return getHeight(root,h);
    }

    bool getHeight(TreeNode* root, int& h){
        if(root == NULL)
            return true;
        int lh = h, rh = h;
        // 遍历获得左右子树的高度
        int left = getHeight(root->left, lh);
        int right = getHeight(root->right, rh);
        if(abs(lh-rh) > 1)
            return false;
        h = max(lh,rh) + 1;
        return true;
    }
};
```

上述解法是通过递归分别得到左右子树的高度，并进行判断是否符合平衡二叉树的要求。

6. 完全二叉树判断

> 有一棵二叉树,请设计一个算法判断它是否是完全二叉树。
>
> 给定二叉树的根结点**root**，请返回一个bool值代表它是否为完全二叉树。树的结点个数小于等于500。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class CheckCompletion {
public:
    bool chk(TreeNode* root) {
        if(root == NULL)
            return true;
        queue<TreeNode*> nodes;
        nodes.push(root);
        bool flag = false;
        while(!nodes.empty()){
            TreeNode *p = nodes.front();
            nodes.pop();
            if(flag){
                if(p->left != NULL || p->right != NULL)
                    // 这是在前一个结点只包含一个子结点情况下，下个结点，即当前结点却拥有子结点的情况
                    return false;
            }
            if(p->left == NULL && p->right != NULL)
                // 只有右子结点是不满足条件的
                return false;
            if(p->left == NULL || p->right == NULL)
                // 当前结点可能是叶子结点或者只有一个左子结点
                flag = true;
            if(p->left)
                nodes.push(p->left);
            if(p->right)
                nodes.push(p->right);
        }
        return true;
    }
};
```

主要是使用层次遍历，使用一个队列辅助，每次判断当前结点的子结点个数，如果遇到只有一个子结点，那么要判断是否是右子结点，如果是就不满足满二叉树条件，如果不是，那么就是只有一个左子结点，或者遇到叶子结点，那么就需要注意下一个结点必须是叶子结点才可以满足条件。

7. 折纸

> 请把纸条竖着放在桌□上，然后从纸条的下边向上□对折，压出折痕后再展 开。此时有1条折痕，突起的□向指向纸条的背□□这条折痕叫做“下”折痕 ； 突起的□向指向纸条正□的折痕叫做“上”折痕。如果每次都从下边向上□ 对折，对折N次。请从上到下计算出所有折痕的□向。
>
> 给定折的次数**n**,请返回从上到下的折痕的数组，若为下折痕则对应元素为"down",若为上折痕则为"up".

> 测试样例：
>
> > 1
>
> 返回：["down"]

解法如下：

```
class FoldPaper {
public:
    vector<string> foldPaper(int n) {
        if(n <= 0)
            return vector<string>();
        vector<string> res;
        mid(n, res, "down");
        return res;
    }
    void mid(int n, vector<string>& res, string str){
        if(n == 0)
            return;
        // 中序遍历，并且是先“down”，再有“up”
        mid(n-1, res, "down");
        res.push_back(str);
        mid(n-1, res, "up");
    }
};
```

依题意，有根结点是“down”，然后左子树都是“up”，右子树是“down”，则按照右中左的顺序遍历二叉树就是打印的顺序。

8. 寻找错误结点

> 一棵二叉树原本是搜索二叉树，但是其中有两个节点调换了位置，使得这棵二叉树不再是搜索二叉树，请找到这两个错误节点并返回他们的值。保证二叉树中结点的值各不相同。
>
> 给定一棵树的根结点，请返回两个调换了位置的值，其中小的值在前。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class FindErrorNode {
public:
    vector<int> findError(TreeNode* root) {
        if(root == NULL)
            return vector<int>();
        vector<int> nodes;
        vector<int> result(2);
        mid(root, nodes);
        int first = -1, second = -1;
        for(int i=0; i<nodes.size()-1; ++i){
            if(nodes[i] > nodes[i+1]){
                if(first == -1)
                    // 记录第一个错误的结点位置
                    first = i;
                else
                    second = i;
            }
        }
        if(second == -1){
            // 这表示两个错误结点是相邻点
            result[0] = nodes[first+1];
            result[1] = nodes[first];
        }else{
            // 表示错误结点不相邻，而且第二个结点位置应该是记录的位置再往后一位
            result[0] = nodes[second+1];
            result[1] = nodes[first];
        }
        return result;
    }
    void mid(TreeNode* root, vector<int>& res){
        // 中序遍历，非递归
        if(root == NULL)
            return;
        stack<TreeNode*> nodes;
        auto p = root;
        while(p || !nodes.empty()){
            while(p){
                nodes.push(p);
                p = p->left;
            }
            if(!nodes.empty()){
                p = nodes.top();

                nodes.pop();
```

```
                res.push_back(p->val);
                p = p->right;
            }
        }
    }
};
```

这道题目首先是要进行中序遍历得到一个升序排列的序列，然后进行遍历，查找出现降序的地方的位置，这里分两种情况，第一种是两个结点是相邻的，所以记录位置只会记录开始降序的位置，所以`second`变量在这种情况会保持初始值`-1`；第二种就是不相邻了，那么会出现一段降序的序列，然后`second`会记录到最后降序部分的初始位置，但实际上错误的结点应该还是在其后面一位，这是需要注意的地方。最后就是第一个记录的结点位置得到的结点是数值更大的。

9. 树上最远的距离

> 从二叉树的节点A出发，可以向上或者向下走，但沿途的节点只能经过一次，当到达节点B时，路径上的节点数叫作A到B的距离。对于给定的一棵二叉树，求整棵树上节点间的最大距离。
>
> 给定一个二叉树的头结点**root**，请返回最大距离。保证点数大于等于2小于等于500.

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class LongestDistance {
public:
    int findLongest(TreeNode* root) {
        return getLongest(root);
    }
    //获取树的最大距离
    int getLongest(TreeNode* root)
    {
        if(!root)
            return 0;
        int LLgt = 0;//左子树最大距离
        int RLgt = 0;//右子树最大距离
        int LNodeLgt = 0;//左子树中距离左孩子最长距离
        int RNodeLgt = 0;//右子树中距离右孩子最长距离
        if(root->left)
        {
            LLgt = getLongest(root->left);
            LNodeLgt = getNodeLgt(root->left);
        }
        if(root->right)
        {
            RLgt = getLongest(root->right);
            RNodeLgt = getNodeLgt(root->right);
        }
        return max(max(LLgt, RLgt), LNodeLgt+RNodeLgt+1);
    }
    //计算以node为头的子树距离node的最长距离
    int getNodeLgt(TreeNode* node)
    {
        if(!node)
            return 0;
        int LLgt = 0;
        int RLgt = 0;
        LLgt = getNodeLgt(node->left) + 1;
        RLgt = getNodeLgt(node->right) + 1;
        return max(LLgt, RLgt);
    }
};
```

10. 最大二叉搜索子树

> 有一棵二叉树，其中所有节点的值都不一样,找到含有节点最多 的搜索二叉子树,并返回这棵子树的头节点.

> 给定二叉树的头结点**root**，请返回所求的头结点,若出现多个节点最多的子树，返回头结点权值最大的。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/
class MaxSubtree {
public:
    TreeNode* getMax(TreeNode* root) {
        if(root == NULL)
            return NULL;
        int max, min, nums;
        return myGetMax(root, max, min, nums);
    }
    TreeNode* myGetMax(TreeNode* node,int& maxNode,int& minNode,int& numNode){
        if(node == NULL){
            maxNode = -9999999;
            minNode = 9999999;
            numNode = 0;
            return NULL;
        }
        //搜集左结点的信息
        int lmax,lmin,lnum;
        TreeNode* lnode = myGetMax(node->left,lmax,lmin,lnum);
        //搜集右结点的信息
        int rmax,rmin,rnum;
        TreeNode* rnode = myGetMax(node->right,rmax,rmin,rnum);

        //更新max,min
        maxNode = max(rmax,node->val);
        minNode = min(lmin,node->val);

        if(lmax < node->val && node->val < rmin && lnode == node->left
           && rnode == node->right){
            // 当前结点就是一个二叉搜索子树的根结点
            numNode = lnum + rnum + 1;

            return node;
        }
        // 返回节点数最多的
        if(lnum > rnum){
            numNode = lnum;
            return lnode;
        }else{
            numNode = rnum;
            return rnode;
        }
    }
};
```

11. 完全二叉树计数

> 给定一棵完全二叉树的根节点root，返回这棵树的节点个数。如果完全二叉树的节点数为N，请实现时间复杂度低于O(N)的解法。
>
> 给定树的根结点**root**，请返回树的大小。

解法如下：

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
            val(x), left(NULL), right(NULL) {
    }
};*/

class CountNodes {
public:
    int count(TreeNode* root) {
        TreeNode *tmp = root;
        int amount = 0;
        if(root->left == NULL)
            return 1;
        int high = 0;
        while(tmp){
            // 找到最左的子树，并统计树的高度
            ++high;
            tmp = tmp->left;
        }
        tmp = root->right;
        int right = 1;
        while(tmp){
            // 找到右子树的最左子树，并统计其高度
            ++right;
            tmp = tmp->left;
        }
        if(high == right){
            // 此时根结点的左子树是满二叉树，递归遍历右子树结点个数
            amount += pow(2, high-1);
            amount += count(root->right);
        }else{
            // 此时根结点的右子树是满二叉树，递归遍历左子树结点个数
             amount += pow(2, high-2);
            amount += count(root->left);
        }
        return amount;
    }
};
```

先找到最左的子结点，并且记录此时树的高度`high`，然后找到根结点右子树的最左结点，并且统计其高度`right`，比较这两个高度，如果相等，说明两者在同一高度，那么左子树是满二叉树，右子树可以通过递归遍历来统计结点数；否则，右子树是满二叉树，左子树通过递归遍历得到结点数量。

位运算

1. 交换练习

> 给定一个数组**num**，其中包含两个值，请不用任何额外变量交换这两个值，并将交换后的数组返回。
>
> 测试样例：
>
> > [1,2]
> >
> > 返回：[2,1]

解法如下：

```
vector<int> getSwap(vector<int> num) {
        // 使用异或交换两个数
        num[0] ^= num[1];
        num[1] ^= num[0];
        num[0] ^= num[1];
        return num;
}
```

使用异或方法进行交换两个数。

2. 比较练习

> 对于两个32位整数a和b，请设计一个算法返回a和b中较大的。但是不能用任何比较判断。若两数相同，返回任意一个。
>
> 给定两个整数**a**和**b**，请返回较大的数。
>
> 测试样例：
>
> > 1,2
> >
> > 返回：2

解法如下：

```
int getMax(int a, int b) {
        // 获取两个数的差值
        int c = a - b;
        // 查看符号位，如果是负数，结果是1，正数是0
        int flag = (c>>31) & 1;
        // flag是负数，返回b，是正数，则返回a
        return b*flag + a*(flag ^ 1);
}
```

解法主要是判断两个整数差值的符号位。

3. 寻找奇数出现

> 有一个整型数组A，其中只有一个数出现了奇数次，其他的数都出现了偶数次，请打印这个数。要求时间复杂度为O(N)，额外空间复杂度为O(1)。
>
> 给定整形数组**A**及它的大小**n**，请返回题目所求数字。
>
> 测试样例：
>
> > [1,2,3,2,1],5
>
> > 返回：3

解法如下：

```
int findOdd(vector<int> A, int n) {
        int res = A[0];
        // 对每个数都进行异或，最终结果就是只出现奇数次的数字
        for(int i=1; i<n; ++i)
            res ^= A[i];
        return res;
    }
```

使用异或运算，偶数次的元素会变成0，剩下的结果就是奇数次的数。

4. 奇数出现2

> 给定一个整型数组arr，其中有两个数出现了奇数次，其他的数都出现了偶数次，找到这两个数。要求时间复杂度为O(N)，额外空间复杂度为O(1)。
>
> 给定一个整形数组**arr**及它的大小**n**，请返回一个数组，其中两个元素为两个出现了奇数次的元素,请将他们按从小到大排列。
>
> 测试样例：
>
> > [1,2,4,4,2,1,3,5],8
>
> > 返回：[3,5]

解法如下：

```
class OddAppearance {
public:
    vector<int> findOdds(vector<int> arr, int n) {
        int res = 0;
        for(int i=0; i<n; ++i)
            res ^= arr[i];
        unsigned int indexOf1 = findBitsOf1(res);
        vector<int> r(2,0);
        for(int j=0; j<n; ++j){
            // 根据indexOf1位上是否为1分成两组
            if(isBit1(arr[j],indexOf1))
                r[0] ^= arr[j];
            else
                r[1] ^= arr[j];
        }
        sort(r.begin(), r.end());
        return r;
    }

    // 寻找出现1的位数
    unsigned int findBitsOf1(int n){
        int count = 0;
        while((n & 1) == 0 && count < sizeof(int) * 8){
            ++count;
            n = n >> 1;
        }
        return count;
    }
    // 判断给定的位数上是否为1
    bool isBit1(int n, unsigned int index){
        n = n >> index;
        return (n & 1);
    }
};
```

这个方法也是使用了异或，对数组的所有元素都进行异或后，得到的结果由于有两个数是奇数次，所以不为0，因此先找出第一个为1的位，根据这个将数组分成两部分，分别进行异或就可以得到结果。

排列组合

排列公式为$A_n^m = \frac{n!}{(n-m)!}$;

组合公式是$C_n^m = \frac{A_n^m}{m!} = \frac{n!}{m!(n-m)!}$。

1. 方格移动

> 在XxY的方格中，以左上角格子为起点，右下角格子为终点，每次只能向下走或者向右走，请问一共有多少种不同的走法
>
> 给定两个正整数int **x**,int **y**，请返回走法数目。保证x＋y小于等于12。
>
> 测试样例：

> 2,2
>
> 返回：2

解法如下：

```
class Robot {
public:
    int countWays(int x, int y) {
        if(x == 0 && y == 0)
            return 1;
        else if(x == 0)
            return y-1;
        else if(y == 0)
            return x-1;
        // 向下走的步数
        int cx = x-1;
        // 向右走的步数
        int cy = y-1;
         // 分别计算总的步数，然后进行组合，求C(cx+cy,cx) 或者 C(cx+cy,cy)
        return factorial(cx+cy) / (factorial(cx) * factorial(cy));
    }

    int factorial(int n){
        // 计算阶乘
        int res = 1;
        while(n > 0)
            res *= (n--);
        return res;
    }
};
```

上述解法首先求出总共需要走多少步数，也就是往下走的步数和往右走的步数，然后通过组合方法$C_{sum}^{x} = C_{sum}^{y}$,其中$sum = x + y$，也就是总的步数。

2. 站队问题

> n个人站队，他们的编号依次从1到n，要求编号为a的人必须在编号为b的人的左边，但不要求一定相邻，请问共有多少种排法？第二问如果要求a必须在b的左边，并且一定要相邻，请问一共有多少种排法？
>
> 给定人数**n**及两个人的编号**a**和**b**，请返回一个两个元素的数组，其中两个元素依次为两个问题的答案。保证人数小于等于10。
>
> 测试样例：
>
> > 7,1,2
> >
> > 返回：[2520,720]

解法如下：

```
class StandInLine {
public:
    vector<int> getWays(int n, int a, int b) {
        if(n <= 0)
            return vector<int>();
        vector<int> res;
        // n个人的排列有n! 种
        int total = factorial(n);
        // 不一定相邻的解法是n!/2
        res.push_back(total/2);
        // 一定相邻的解法是将这两个人当做一个人看，解法是(n-1)!
        int t2 = factorial(n-1);
        res.push_back(t2);
        return res;
    }
    int factorial(int n){
        // 计算阶乘
        int res = 1;
        while(n > 0)
            res *= (n--);
        return res;
    }
};
```

n个人的排列有`n!`种，而其中两个人，如a和b，他们的站队分别有两种形式，a在b左边或者a在b右边，现在第一种是要求a在b左边，所以是有`n!/2`种，而如果要求相邻，则可以将其看成是一个人，求n-1个人的排列方式，有`(n-1)!`种方法。

3. 孤傲的A

> A(A也是他的编号)是一个孤傲的人，在一个n个人(其中编号依次为1到n)的队列中，他于其中的标号为b和标号c的人都有矛盾，所以他不会和他们站在相邻的位置。现在问你满足A的要求的对列有多少种？
>
> 给定人数**n**和三个人的标号**A**,**b**和**c**，请返回所求答案，保证人数小于等于11且大于等于3。
>
> 测试样例：
>
> > 6,1,2,3
> >
> > 288

解法如下：

```
class LonelyA {
public:
    int getWays(int n, int A, int b, int c) {
        if(n <= 3)
            return 0;
        // 总排列数
        int sum = factorial(n);
        // a和b相邻-->ab/ba
        int ab = factorial(n-1)*2;
        // a和c相邻
        int ac = factorial(n-1)*2;
        // 上述两种情况中重复的情况是三者相邻
        int abc = factorial(n-2)*2;
        return sum - (ab+ac-abc);
    }
    int factorial(int n){
        // 计算阶乘
        int res = 1;
        while(n > 0)
            res *= (n--);
        return res;
    }
};
```

上述解法是分别求出a与b，a与c相邻的情况，然后用总数减去它们，这里注意a与b，c相邻都分别有两种排列，如ab或者ba，然后看做一个人，再来求排列可能，接着还需要减去三者相邻的情况。

4. 分糖果

> n颗相同的糖果，分给m个人，每人至少一颗，问有多少种分法。
>
> 给定**n**和**m**，请返回方案数，保证n小于等于12，且m小于等于n。
>
> 测试样例：
>
> > 10,3
> >
> > 返回：36

解法如下：

```
class Distribution {
public:
    int getWays(int n, int m) {
        if(n <= m && m <= 0)
            return 0;
        // 相当于n-1个空挡中放入m-1个隔板
        return factorial(n-1)/(factorial(m-1)*factorial(n-m));
    }
    int factorial(int n){
        int res = 1;
        while(n>0)
            res *= n--;
        return res;
    }
};
```

这个分糖果，n个人，有n-1个空挡，分给m个人，相当于用m-1个隔板隔开n个人，所以有C_{n-1}^{m-1}。

5. 括号序列

> 假设有n对左右括号，请求出合法的排列有多少个？合法是指每一个括号都可以找到与之配对的括号，比如n=1时，()是合法的，但是)(为不合法。
>
> 给定一个整数**n**，请返回所求的合法排列数。保证结果在int范围内。
>
> 测试样例：
>
> > 1
>
> > 返回：1

解法如下：

```
int countLegalWays(int n) {
        if(n <= 0)
            return 0;
        // 总的排列C(2n,n)
        // 非法组合的可能有C(2n,n+1)
        // 结果是卡特兰数公式，即C(2n,n)/(n+1)
        int a=1;
        int b=1;
        for(int i=1;i<=n;i++)
        {
            a*=n+i;
            b*=i;
        }
        return a/b/(n+1);
    }
```

卡特兰数公式为$\frac{C_{2n}^{n}}{n+1}$。

6. 进出栈

> n个数进出栈的顺序有多少种？假设栈的容量无限大。

> 给定一个整数**n**，请返回所求的进出栈顺序个数。保证结果在int范围内。
>
> 测试样例：
>
> > 1
> >
> > 返回：1

解法和上一题一样，也是使用卡特兰数公式解决。

7. 排队买票

> 2n个人排队买票，n个人拿5块钱，n个人拿10块钱，票价是5块钱1张，每个人买一张票，售票员手里没有零钱，问有多少种排队方法让售票员可以顺利卖票。
>
> 给定一个整数**n**，请返回所求的排队方案个数。保证结果在int范围内。
>
> 测试样例：
>
> > 1
> >
> > 返回：1

解法同样如括号序列一样，使用卡特兰数公式解决。

8. 二叉树统计

> 求n个无差别的节点构成的二叉树有多少种不同的结构？
>
> 给定一个整数**n**，请返回不同结构的二叉树的个数。保证结果在int范围内。
>
> 测试样例：
>
> > 1
> >
> > 返回：1

解法其实也是卡特兰数公式。该题目解法是通过枚举可以找到规律，即有

> $$f(0)=1, f(1)=1, f(2)=2, f(3)=5$$
> $$f(n)=f(0)*f(n-1)+f(1)*f(n-2)+f(2)*f(n-3)+\ldots+f(n-1)*f(0)$$
> $$=\frac{1}{n+1}C_{2n}^{n}$$

9. 高矮排列

> 12个高矮不同的人，排成两排，每排必须是从矮到高排列，而且第二排比对应的第一排的人高，问排列方式有多少种？
>
> 给定一个偶数**n**，请返回所求的排列方式个数。保证结果在int范围内。
>
> 测试样例：
>
> > 1
> >
> > 返回：1

这道题目也是一个使用卡特兰数公式的解法。如下所示

```
class HighAndShort {
public:
    int countWays(int n) {
        if(n<=0)
            return 0;
        int a=1;
        int b=1;
        for(int i=1;i<=n/2;++i){
            a *= n/2+i;
            b *= i;
        }
        return a/b/(n/2+1);
    }
};
```

这里需要注意要使用`n/2`，是n个人分两排。

10. 错装信封

> 有n个信封，包含n封信，现在把信拿出来，再装回去，要求每封信不能装回它原来的信封，问有多少种装法?
>
> 给定一个整数**n**，请返回装发个数，为了防止溢出，请返回结果Mod 1000000007的值。保证n的大小小于等于300。
>
> 测试样例：
>
> > 2
>
> > 返回：1

解法如下：

```
class CombineByMistake {
public:
    int countWays(int n) {
        if(n <= 1)
            return 0;
        if(n == 2)
            return 1;
        const int mod = 1000000007;
        // 避免溢出，定义long long类型
        vector<long long> res(n+1,0);
        res[2] = 1;
        for(int i=3; i<=n; ++i){
            res[i] = ((i-1)*(res[i-1]+res[i-2]))%mod;
        }
        return res[n];
    }
};
```

假设第n封信装到第i个信封,n是大于2，且不等于i，则有两种可能：

1. 第i封信装到第n个信封中，则后续是f(n-2)，此时是解决了两封信，剩下n-2封信处理；
2. 第i封信没有装到第n个信封中，则后续是f(n-1)。

而i其实有n-1种可能，所以有$f(n)=(n-1)(f(n-2)+f(n-1))$。

概率

1. 足球比赛

> 有2k只球队，有k-1个强队，其余都是弱队，随机把它们分成k组比赛，每组两个队，问两强相遇的概率是多大？
>
> 给定一个数k，请返回一个数组，其中有两个元素，分别为最终结果的分子和分母，请化成最简分数
>
> 测试样例：
>
> > 4
> >
> > 返回：[3,7]

解法如下：

```cpp
class Championship {
public:
    vector<int> calc(int k) {
        if(k <= 0)
            return vector<int>();
        vector<int> res;
        if(k < 3){
            // 这种情况最多只有1只强队
            res.push_back(0);
            res.push_back(1);
        }else{
            int numerator = 0;
            int denominator = 1;
            int i;
            for(int i=2; i<2*k; i += 2)
                // 所有可能的组合情况
                denominator *= (i+1);
            // 两强相遇的可能情况
            numerator = denominator - factorial(k+1)/factorial(2);
            i = numerator;
            while(i>1){
                // 化简
                if(denominator%i == 0 && numerator%i == 0){
                    numerator /= i;
                    denominator /= i;
                    i = numerator;
                }else
                    --i;
            }
            res.push_back(numerator);
            res.push_back(denominator);
        }

        return res;
    }
    //排列数
    int factorial(int n){
        int result = 1;
        while(0 != n)
            result *= n--;
        return result;
    }
};
```

2. 蚂蚁相遇

> n只蚂蚁从正n边形的n个定点沿着边移动，速度是相同的，问它们碰头的概率是多少？
>
> 给定一个正整数n，请返回一个数组，其中两个元素分别为结果的分子和分母，请化为最简分数。
>
> 测试样例：
>
> > 3

> 返回：[3,4]

解法如下：

```
class Ants {
public:
    vector<int> collision(int n) {
        if(n<3)
            return vector<int>();
        vector<int> res;
        int numerator = pow(2,n)-2;
        int denominator = pow(2,n);

        res.push_back(numerator/max(denominator,numerator));
        res.push_back(denominator/max(denominator,numerator));
        return res;
    }
    // 辗转相除法
    int max(int x, int y){
        if(x%y==0){
            return y;
        }
        else{
            return max(y,x%y);
        }
    }
};
```

每个蚂蚁有两种选择，所以n个蚂蚁有2^n种选择，只有都是顺时针或者逆时针才会没有蚂蚁相遇，所以就是减去这两种情况，然后化简的时候采用辗转相除法。

3. 随机函数

> 给定一个等概率随机产生1~5的随机函数，除此之外，不能使用任何额外的随机机制，请实现等概率随机产生1~7的随机函数。(给定一个可调用的Random5::random()方法,可以等概率地随机产生1～5的随机函数)

解法如下：

```
// 以下内容请不要修改
class Random5 {
public:
    static int randomNumber();
};

class Random7 {
public:
    int rand5() {
        return Random5::randomNumber();
    }
    // 以上内容请不要修改


    int randomNumber() {
        // 代码写这里,通过rand5函数随机产生[1,7]
        int x;
        while(1){
            x = rand5()-1;
            x = x*5 + rand5()-1;
            if(x <= 20)
                break;
        }
        return x%7 + 1;
    }
};
```

4. 随机01

> 给定一个以p概率产生0，以1-p概率产生1的随机函数RandomP::f()，p是固定的值，但你并不知道是多少。除此之外也不能使用任何额外的随机机制，请用RandomP::f()实现等概率随机产生0和1的随机函数。

解法如下：

```
class RandomP {
public:
    static int f();
};

class Random01 {
public:
    // 用RandomP::f()实现等概率的01返回
    int random01() {
        int num;
        do{
            num = RandomP::f();
            // 如果是生成00或者11，就继续循环，否则01就返回0,10就返回1
        }while(num == RandomP::f());
        return num;
    }
};
```

连续两次调用函数，其概率就都变成$p*(1-p)$,即变成等概率了，此时如果两次生成相同的数就继续循环，直到生成两个不同的数。

5. 随机区间函数

> 假设函数f()等概率随机返回一个在[0,1)范围上的浮点数，那么我们知道，在[0,x)区间上的数出现的概率为x(0<x≤1)。给定一个大于0的整数k，并且可以使用f()函数，请实现一个函数依然返回在[0,1)范围上的数，但是在[0,x)区间上的数出现的概率为x的k次方。

解法如下：

```
class RandomSeg {
public:
    // 等概率返回[0,1)
    double f() {
        return rand() * 1.0 / RAND_MAX;
    }
    // 通过调用f()来实现
    double random(int k, double x) {
        double m = -1;
        for(int i=0; i<k; ++i)
            m = max(m, f());
        return m;
    }
};
```

调用k次函数，并且返回较大的数值。

6. 随机数组打印

> 给定一个长度为N且没有重复元素的数组arr和一个整数M，实现函数等概率随机打印arr中的M个数。

解法如下：

```
class RandomPrint {
public:
    vector<int> print(vector<int> arr, int N, int M) {
        if(N <= 0)
            return vector<int>();
        vector<int> res(M);
        for(int i=0; i<M; ++i){
            int num = rand()%(N-i);
            res[i] = arr[num];
            // 将已经打印的数值交换到数组尾部
            swap(arr[num], arr[N-i-1]);
        }
        return res;
    }
};
```

这是类似于随机洗牌的思路，每次选择一个数打印，并将其交换到数组尾部，然后每次选择的范围自动就减1了。

7. 机器吐球

> 有一个机器按自然数序列的方式吐出球，1号球，2号球，3号球等等。你有一个袋子，袋子里最多只能装下K个球，并且除袋子以外，你没有更多的空间，一个球一旦扔掉，就再也不可拿回。设计一种选择方式，使得当机器吐出第N号球的时候，你袋子中的球数是K个，同时可以保证从1号球到N号球中的每一个，被选进袋子的概率都是K/N。举一个更具体的例子，有一个只能装下10个球的袋子，当吐出100个球时，袋子里有10 球，并且1~100号中的每一个球被选中的概率都是10/100。然后继续吐球，当吐出1000个球时，袋子里有 10 个球，并且1~1000号中的每一个球被选中的概率都是10/1000。继续吐球，当吐出i个球时，袋子里有10个球，并且1~i号中的每一个球被选中的概率都是10/i。也就是随着N的变化，1~N号球被选中的概率动态变化成k/N。请将吐出第N个球时袋子中的球的编号返回。

解法如下：

```
class Bag {
public:
    vector<int> ret;
    // 每次拿一个球都会调用这个函数，N表示第i次调用
    vector<int> carryBalls(int N, int k) {
        if(N<=k){
            // 没有达到袋子的最大容量前是装入球
            ret.push_back(N);
        }else{
            // 随机选择一个袋子中的球扔掉然后放入新的球
            int index = rand()%k;
            if(index < k)
                ret[index] = N;
        }
        return ret;
    }
};
```

动态规划

1. 找零钱

> 有数组penny，penny中所有的值都为正数且不重复。每个值代表一种面值的货币，每种面值的货币可以使用任意张，再给定一个整数aim(小于等于1000)代表要找的钱数，求换钱有多少种方法。
>
> 给定数组**penny**及它的大小(小于等于50)，同时给定一个整数aim，请返回有多少种方法可以凑成aim。
>
> 测试样例：
>
> > [1,2,4],3,3
> >
> > 返回：2

解法如下：

```
class Exchange {
public:
    int countWays(vector<int> penny, int n, int aim) {
        vector<vector<int> > dp(n);
        for(int i=0; i<n; ++i){
            // 对于求目标是0元，方法都是1种，就是不用任何货币
            dp[i].resize(aim+1, 0);
            dp[i][0] = 1;
        }
        for(int j=0; j<=aim; ++j){
            if(j>= penny[0] && j%penny[0] == 0)
                // 目标货币必须大于等于第一种货币并且是其数倍，方法也是1种
                dp[0][j] = 1;
        }
        for(int i=1; i<n; ++i){
            for(int j=0; j<aim+1; ++j){
                if(j >= penny[i]){
                    // 当目标货币大于当前使用货币值，在不使用当前货币的方法是dp[i-1][j],使用当前
                    //货币的方法是dp[i][j-penny[i]]
                    dp[i][j] = dp[i-1][j] + dp[i][j-penny[i]];
                }else
                    dp[i][j] = dp[i-1][j];
            }
        }
        return dp[n-1][aim];
    }
};
```

动态规划方法，首先是考虑暴力搜索的时候如何求解问题，也就是一个递归求解问题，然后在考虑使用记忆搜索方法，即将暴力搜索解决的时候，记录每次的答案，然后在调整求解顺序，就可以变成动态规划问题，并且再通过化简求解，得到最优解。

2. 台阶问题

> 有n级台阶，一个人每次上一级或者两级，问有多少种走完n级台阶的方法。为了防止溢出，请将结果Mod 1000000007
>
> 给定一个正整数int **n**，请返回一个数，代表上楼的方式数。保证n小于等于100000。
>
> 测试样例：
>
> > 1
>
> > 返回：1

解法如下：

```
class GoUpstairs {
public:
    int countWays(int n) {
        if(n <= 0)
            return 0;
        if(n == 1)
            return 1;
        if(n == 2)
            return 2;
        int res1 = 1;
        int res2 = 2;
        int res = 0;
        for(int i=3; i<=n; ++i){
            res = (res1+res2)%1000000007;

            res1 = res2;
            res2 = res;
        }
        return res;
    }
};
```

这个通过寻找规律可以知道有$f(n) = f(n-1) + f(n-2)$。

3. 矩阵最小路径和

> 有一个矩阵map，它每个格子有一个权值。从左上角的格子开始每次只能向右或者向下走，最后到达右下角的位置，路径上所有的数字累加起来就是路径和，返回所有的路径中最小的路径和。
>
> 给定一个矩阵**map**及它的行数**n**和列数**m**，请返回最小路径和。保证行列数均小于等于100.
>
> 测试样例：
>
> > [[1,2,3],[1,1,1]],2,3
>
> > 返回：4

解法如下：

```
class MinimumPath {
public:
    int getMin(vector<vector<int> > map, int n, int m) {
        if(n <= 0 || m <= 0)
            return 0;
        // 建立一个同样大小的数组，表示到达当前位置的最小路径和
        vector<vector<int> > dp(n);
        for(int i=0; i<n; ++i){
            dp[i].resize(m,0);
        }
        dp[0][0] = map[0][0];
        // 初始化第一行的数值，其等于左边格子最短路径和加当前数值之和
        for(int j=1; j<m; ++j)
            dp[0][j] = map[0][j]+dp[0][j-1];
        // 初始化第一列的数值
        for(int i=1; i<n; ++i)
            dp[i][0] = map[i][0] + dp[i-1][0];
        for(int i=1; i<n; ++i){
            // 从第二行第二列开始计算
            for(int j=1; j<m; ++j){
                // 当前路径最小和等于当前数值加上左边和上边路径和的最小值
                dp[i][j] = map[i][j] + min(dp[i-1][j], dp[i][j-1]);
            }
        }
        return dp[n-1][m-1];
    }
};
```

这也是动态规划的经典问题。也是先考虑到第一行和第一列的特殊情况，只有一个方向可以到达，即第一行的都是从左边走过来，第一列也是从上边走下来。其他位置都是当前数值加上左边或者上边的路径和的最小值。

4. LIS

> 这是一个经典的LIS(即最长上升子序列)问题，请设计一个尽量优的解法求出序列的最长上升子序列的长度。
>
> 给定一个序列**A**及它的长度**n**(长度小于等于500)，请返回LIS的长度。
>
> 测试样例：
>
> > [1,4,2,5,3],5
> >
> > 返回：3

解法如下：

```
class LongestIncreasingSubsequence {
public:
    int getLIS(vector<int> A, int n) {
        if(n <= 0)
            return 0;
        // dp[i]表示以i位置字符结尾的当前LIS的长度
        vector<int> dp(n);
        dp[0] = 1;
        // 记录最大的长度
        int res = 1;
        for(int i=1; i<n; ++i){
            for(int j=0; j<i; ++j){
                // 寻找比i位置数字更小的数值，并且修改当前i位置最大LIS长度
                if(A[j] < A[i])
                    dp[i] = max(dp[i], dp[j]);
            }
            // 加上当前字符，即长度加1
            ++dp[i];
            // 更新当前最大的LIS长度
            res = max(res, dp[i]);
        }
        return res;
    }
};
```

首先也是生成一个同样长度为`n`的数组dp，它代表每个位置上，以当前字符结尾的最大LIS长度，这个长度的求解是通过寻找前面的字符中，比它小的字符或者数字的位置的LIS长度，找到满足条件的长度中的最大值，并且加1，这个过程中定义一个变量，记录找到过的最大LIS长度。

5. LCS

> 给定两个字符串A和B，返回两个字符串的最长公共子序列的长度。例如，A="1A2C3D4B56”，B="B1D23CA45B6A”，”123456"或者"12C4B6"都是最长公共子序列。
>
> 给定两个字符串**A**和**B**，同时给定两个串的长度**n**和**m**，请返回最长公共子序列的长度。保证两串长度均小于等于300。
>
> 测试样例：
>
> > "1A2C3D4B56",10,"B1D23CA45B6A",12
>
> > 返回：6

解法如下：

```
class LCS {
public:
    int findLCS(string A, int n, string B, int m) {
        if(n <= 0 || m <= 0)
            return 0;
        // 定义数组dp，dp[i][j]表示字符串A[0...i]和字符串B[0...j]的最长公共子序列长度
        vector<vector<int> > dp(n);
        for(int i=0; i<n; ++i)
            dp[i].resize(m,0);
        // 初始化dp[i][0]，即字符串A[0...i]和B[0]的LCS长度，最大只能是1
        for(int i=0; i<n; ++i){
            dp[i][0] = 1;
        }
        for(int i=0; i<n; ++i){
            if(A[i] != B[0])
                dp[i][0] = 0;
            else
                break;
        }
        // 初始化dp[0][j]
        for(int j=0; j<m; ++j)
            dp[0][j] = 1;
        for(int j=0; j<m; ++j){
            if(A[0] != B[j])
                dp[0][j] = 0;
            else
                break;
        }
        // 其他位置
        for(int i=1; i<n; ++i){
            for(int j=1; j<m; ++j){
                if(A[i] == B[j]){
                    // 在A和B对应当前位置上的字符相等
                    dp[i][j] = dp[i-1][j-1] + 1;
                }else{
                    // 否则，就求解A[0...i-1]和B[0...j]的LCS长度，以及A[0...i]和B[0...j-1]的LCS长度
                    //中，最大值的一个就是dp[i][j]的数值
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
                }
            }
        }
        return dp[n-1][m-1];
    }
};
```

也是先解决特殊情况的位置，即第一行和第一列，这里最大就是1，因为分别是跟1个字符进行比较的。而其他位置有三种情况，如果当前位置，即`A[i] == B[j]`，那么就是`dp[i-1][j-1] + 1;`，否则就是A[0...i-1]和B[0...j]的LCS长度，以及A[0...i]和B[0...j-1]的LCS长度中，最大值的一个就是dp[i][j]的数值。

6. 01背包问题

> 一个背包有一定的承重cap，有N件物品，每件都有自己的价值，记录在数组v中，也都有自己的重量，记录在数组w中，每件物品只能选择要装入背包还是不装入背包，要求在不超过背包承重的前提下，选出物品的总价值最大。
>
> 给定物品的重量**w**价值**v**及物品数**n**和承重**cap**。请返回最大总价值。
>
> 测试样例：
>
> > [1,2,3],[1,2,3],3,6
> >
> > 返回：6

解法如下：

```
class Backpack {
public:
    int maxValue(vector<int> w, vector<int> v, int n, int cap) {
        // 定义数组dp,dp[i][j]表示物品数量是i，承重是j时的物品最大价值
        vector<vector<int> > dp(n+1);
        for(int i=0; i<=n; ++i)
            dp[i].resize(cap+1,0);
        for(int i=1; i<=n; ++i){
            for(int j=0; j<=cap; ++j){
                if(j >= w[i-1]){
                    // 计算放入当前物品后和不放当前物品的总价值，取最大值
                    dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i-1]]+v[i-1]);
                }else
                    dp[i][j] = dp[i-1][j];
            }
        }
        return dp[n][cap];
    }
};
```

定义数组dp,`dp[i][j]`表示物品数量是i，承重是j时的物品最大价值。每次增加一件物品的时候，都需要考虑不放入当前物品的价值，以及如果能放入当前物品，即承重没有超过最大值时，物品的总价值，取两者的最大值。

一个更简洁的解法如下；

```
int maxValue(vector<int> w, vector<int> v, int n, int cap) {
        vector<int>dp(cap+1,0);
        for(int i=0;i<n;i++)
        {
            for(int j=cap;j>=w[i];j--)
                 dp[j]=max(dp[j-w[i]]+v[i],dp[j]);
        }

        return dp[cap];
}
```

只需要建立一维数组，`dp[j]`表示的不超过承重是`j`时背包物品的总价值。

7. 最优编辑

> 对于两个字符串A和B，我们需要进行插入、删除和修改操作将A串变为B串，定义c0，c1，c2分别为三种操作的代价，请设计一个高效算法，求出将A串变为B串所需要的最少代价。
>
> 给定两个字符串**A**和**B**，及它们的长度和三种操作代价，请返回将A串变为B串所需要的最小代价。保证两串长度均小于等于300，且三种代价值均小于等于100。
>
> 测试样例：
>
> > "abc",3,"adc",3,5,3,100
> >
> > 返回：8

解法如下：

```
class MinCost {
public:
    int findMinCost(string A, int n, string B, int m, int c0, int c1, int c2) {
        // 定义数组dp,dp[i][j]表示A前i个字符变成B前j个字符的最小代价
        vector<vector<int> > dp(n+1);
        for(int i=0; i<=n; ++i)
            dp[i].resize(m+1,0);
        // dp[i][0] 表示A前i个字符变成一个空字符的最小代价，也就是删除i个字符的代价
        for(int i=0; i<=n; ++i)
            dp[i][0] = i*c1;
        // dp[0][j] 表示A从空字符变成B前j个字符的最小代价，即插入i个字符的代价
        for(int j=0; j<=m; ++j)
            dp[0][j] = j*c0;
        // 其他位置的求解
        for(int i=1; i<=n; ++i){
            for(int j=1; j<=m; ++j){
                //dp[i][j]可由dp[i-1][j] + 删除A[i],dp[i][j-1] + 插入B[i]
                // A[i] != B[j]则dp[i-1][j-1] + 修改A[i];否则d[i-1][j-1]操作而来
                if(A[i-1] == B[j-1]){
                    // 最后一个字符相等的情况
                    int sum1 = dp[i-1][j-1];
                    int sum2 = dp[i-1][j] + c1;
                    int sum3 = dp[i][j-1] + c0;
                    dp[i][j] = min(min(sum1,sum2), sum3);
                }else{
                    int sum1 = dp[i-1][j-1]+c2;
                    int sum2 = dp[i-1][j] + c1;
                    int sum3 = dp[i][j-1] + c0;
                   dp[i][j] = min(min(sum1,sum2), sum3);
                }
            }
        }
        return dp[n][m];
    }
};
```

这也是动态规划问题。首先考察A或B是空字符的情况，前者是空格字符，则需要使用插入来得到B，而后者是空字符，则需要删除A来得到B。然后普通情况，有四种可能，一种是由前i-1个字符的A通过插入得到B，或者是由A删除一个字符可以得到j-1个字符的B，然后如果`A[i-1] == B[j-1]`，那么这种情况就可以等于`dp[i-1][j-1]`，如果不相等，可以是通过替换最后一个字符得到，即`dp[i-1][j-1]+c2`。`dp[i][j]`表示的就是i个字符的A得到j个字符的

B所需要的最小代价。