**数据结构、算法与习题**

本书分三个部分：数据结构，常用算法，以及习题

数据结构有线性表（数组表示和链表表示）、栈（数组和链表表示）、队列（数组和链表表示）、二叉树（数组和链表表示）、搜索树（链表表示）、平衡二叉树（链表表示）、红黑树（链表表示）、B树（链表表示）、广度优先搜索、深度优先搜索。

线性表的操作以及复杂度：

|  |  |  |
| --- | --- | --- |
| 操作 | 描述 | 复杂度 |
| bool Isempty() | 表空，返true |  |
| T SearchElementFromIndex(int index) | 根据index查询元素 |  |
| int SearchIndexFromElement(T Element) | 根据元素查询第一次出现的索引 |  |
| int Size() | 返回表的大小 |  |
| void EraseElementFromIndex(int index) | 根据索引删除元素 |  |
| void InsertElementFromIndex(int index, T element) | 在指定的索引插入指定元素 |  |
| void ShowLineList() | 从左到右显示表的元素 |  |

**数组表示线性表：**

template<class T>

class ArrayLineList :

public linelist<T>

{

public:

//构造函数和复制构造函数与析构函数

ArrayLineList(int nIntialCapacity = 10);

ArrayLineList(ArrayLineList<T>& thelinelist);

~ArrayLineList() { delete[] m\_pelement; }

public:

//相应的方法

void checkIndex(int theindex);//判断索引是否在范围内

bool Isempty();

bool IsFull();

T SearchElementFromIndex(int index);

int SearchIndexFromElement(T Element);

int Size();

void EraseElementFromIndex(int index);

void InsertElementFromIndex(int index, T element);

void ShowLineList();

int capacity();

private:

//成员变量与方法

T\* m\_pelement;//成员元素

int m\_nArrayCapacity;//链表容量

int m\_nArrayListSize;//链表元素个数

};

**1先讨论下构造函数和析构函数**

//构造函数和复制构造函数与析构函数

ArrayLineList(int nIntialCapacity = 10);

ArrayLineList(ArrayLineList<T>& thelinelist);

~ArrayLineList() { delete[] m\_pelement; }

（1）第一个构造函数是初始化链表的容量，给数组开辟相应的内存，所以构造函数的定义就是

//构造函数

template<class T>

ArrayLineList<T>::ArrayLineList(int nIntialCapacity /\*= 10\*/)

{

if (nIntialCapacity < 1)

{

*cout* << "Initial capacity" << nIntialCapacity << " must be >0";

throw illegalParameterValue("Initial capacity is out!!");

}

//相当于初始化了类的成员

m\_nArrayCapacity = nIntialCapacity;

m\_nArrayListSize = 0;

m\_pelement = new T[m\_nArrayCapacity];

if (m\_pelement == nullptr)

{

*cout* << "ArrayLineList new memory false !" << *endl*;

}

}

（2）第二个构造函数ArrayLineList(ArrayLineList<T>& thelinelist);就是复制一个链表。

template<class T>

ArrayLineList<T>::ArrayLineList(ArrayLineList<T>& thelinelist)

{

m\_nArrayCapacity = thelinelist.m\_nArrayCapacity;

m\_nArrayListSize = thelinelist.m\_nArrayListSize;

m\_pelement = new T[m\_nArrayCapacity];//必须分配空间不然后面的copy会崩溃

if (m\_pelement == nullptr)

*cout* << "ArrayLineList new memory false !" << *endl*;

*copy*(thelinelist.m\_pelement, thelinelist.m\_pelement + m\_nArrayListSize, m\_pelement);

}

**\*学习一下copy用法**

将thelinelist.m\_pelement到thelinelist.m\_pelement + m\_nArrayListSize内容复制到 m\_pelement，并且是从开始的位置开始。

如：

#include <iostream>

#include <vector>

using namespace *std*;

int *main*()

{

int array[] = { 1,2,3,4,5 };

*vector*<int> arraybycopyVec;

arraybycopyVec.*resize*(7);//开辟了7个元素的空间

*copy*(array, array + 5, arraybycopyVec.*begin*());

for (*size\_t* i = 0;i<arraybycopyVec.*size*();i++)

*cout* << arraybycopyVec[i] << *endl*;

}

**（3）析构函数**

~ArrayLineList() { delete[] m\_pelement; }

就是要删除掉new出来的空间！

1. **相应的各种操作**
2. **判断链表是否为空bool Isempty()**

这里因为是用数组表示，那么可以直接根据类的成员变量m\_nArrayListSize来判断

template<class T>

bool ArrayLineList<T>::Isempty()

{

return m\_nArrayListSize == 0 ? true : false;

}

**（2）判断链表是否满了，这就要根据容量和链表大小来判断**

template<class T>

bool ArrayLineList<T>::IsFull()

{

return m\_nArrayListSize == m\_nArrayCapacity ? true : false;

}

如果满了就得考虑开辟新空间，

template<class T>

void changelength1D(T\*& a, int oldlength, int newlength)

{

if (newlength < 0)

throw illegalParameterValue("new length must be >=0");

T\* temp = new T[newlength];

int number = *std*::*min*(oldlength, newlength);

*copy*(a, a + number, temp);//copy的用法

delete[]a;

a = temp;

}

**注意：如果处理的数据量比较大，开辟空间频繁会导致效率下降，所以这方面的考虑在工作过程中要注意。**

1. **根据索引查询元素 T SearchElementFromIndex(int index);**

因为是数组表示的，所以可以直接通过数组的索引查询

**注意：索引和数组的下标是不同的**

template<class T>

T ArrayLineList<T>::SearchElementFromIndex(int index)

{

return m\_pelement[index];

}

1. **通过元素值查询第一次元素所在索引，如果没有则返回-1，int SearchIndexFromElement(T theElement)**

template<class T>

int ArrayLineList<T>::SearchIndexFromElement(T theElement)

{

//先查找元素所在索引

int nindex = (int)(find(m\_pelement, m\_pelement + m\_nArrayListSize, theElement) - m\_pelement);

if (nindex == m\_nArrayListSize)

return -1;

else

return nindex;

}

**\*学习一下find的系列函数以及search系列函数**

主要是find，find\_if,find\_if\_not,find\_end,find\_first\_of,adjacent\_find,search,serch\_n

**find 语法格式：**

**InputIterator find (InputIterator first, InputIterator last, const T& val);**

**其中，first 和 last 为输入迭代器，[first, last) 用于指定该函数的查找范围；val 为要查找的目标元素。该函数适用于所有序列式容器。**

这里first和last的类型是输入迭代器，返回的是一个迭代器。

当 find() 函数查找成功时，其指向的是在 [first, last) 区域内查找到的第一个目标元素；如果查找失败，则该迭代器的指向和 last 相同。  
 值得一提的是，find() 函数的底层实现，其实就是用==运算符将 val 和 [first, last) 区域内的元素逐个进行比对。这也就意味着，[first, last) 区域内的元素必须支持==运算符。

**find\_if语法格式：**

**InputIterator find\_if (InputIterator first, InputIterator last, UnaryPredicate pred);**

其中，first 和 last 都为输入迭代器，其组合 [first, last) 用于指定要查找的区域；pred 用于自定义查找规则。

**值得一提的是，由于 first 和 last 都为输入迭代器，意味着该函数适用于所有的序列式容器。甚至当采用适当的谓词函数时，该函数还适用于所有的关联式容器（包括哈希容器）。**

同时，该函数会返回一个输入迭代器，当查找成功时，该迭代器指向的是第一个符合查找规则的元素；反之，如果 find\_if() 函数查找失败，则该迭代器的指向和 last 迭代器相同。

**find\_if\_not**

find\_if\_not() 函数和 find\_if() 函数的功能恰好相反，通过上面的学习我们知道，find\_if() 函数用于查找符合谓词函数规则的第一个元素，而 find\_if\_not() 函数则用于查找第一个不符合谓词函数规则的元素。  
 **find\_if\_not() 函数的语法规则如下所示：**

**InputIterator find\_if\_not (InputIterator first, InputIterator last, UnaryPredicate pred);**

**其中，first 和 last 都为输入迭代器，[first, last) 用于指定查找范围；pred 用于自定义查找规则。**

**和 find\_if() 函数一样，find\_if\_not() 函数也适用于所有的容器，包括所有序列式容器和关联式容器。**

同样，该函数也会返回一个输入迭代器，当 find\_if\_not() 函数查找成功时，该迭代器指向的是查找到的那个元素；反之，如果查找失败，该迭代器的指向和 last 迭代器相同。

**find\_end()**

函数定义在<algorithm>头文件中，常用于在序列 A 中查找序列 B 最后一次出现的位置。

例如，有如下 2 个序列：

**序列 A：1,2,3,4,5,1,2,3,4,5  
序列 B：1,2,3**

通过观察不难发现，序列 B 在序列 A 中出现了 2 次，而借助 find\_end() 函数，可以轻松的得到序列 A 中最后一个（也就是第 2 个） {1,2,3}。

find\_end() 函数的语法格式有 2 种：

**//查找序列 [first1, last1) 中最后一个子序列 [first2, last2)**

**ForwardIterator find\_end (ForwardIterator first1, ForwardIterator last1,**

**ForwardIterator first2, ForwardIterator last2);**

**//查找序列 [first2, last2) 中，和 [first2, last2) 序列满足 pred 规则的最后一个子序列**

**ForwardIterator find\_end (ForwardIterator first1, ForwardIterator last1,**

**ForwardIterator first2, ForwardIterator last2,**

**BinaryPredicate pred);**

其中，各个参数的含义如下：

**（1）first1、last1：都为正向迭代器，其组合 [first1, last1) 用于指定查找范围（也就是上面例子中的序列 A）；**

**（2）first2、last2：都为正向迭代器，其组合 [first2, last2) 用于指定要查找的序列（也就是上面例子中的序列 B）；**

**（3）pred：用于自定义查找规则。该规则实际上是一个包含 2 个参数且返回值类型为 bool 的函数（第一个参数接收 [first1, last1) 范围内的元素，第二个参数接收 [first2, last2) 范围内的元素）。函数定义的形式可以是普通函数，也可以是函数对象。**

**实际上，第一种语法格式也可以看做是包含一个默认的 pred 参数，该参数指定的是一种相等规则，即在 [first1, last1) 范围内查找和 [first2, last2) 中各个元素对应相等的子序列；而借助第二种语法格式，我们可以自定义一个当前场景需要的匹配规则。**

同时，find\_end() 函数会返回一个正向迭代器，当函数查找成功时，该迭代器指向查找到的子序列中的第一个元素；反之，如果查找失败，则该迭代器的指向和 last1 迭代器相同。

find\_first\_of：

我们可能需要在 A 序列中查找和 B 序列中任意元素相匹配的第一个元素，这时就可以使用 find\_first\_of() 函数。

**ind\_first\_of() 函数有 2 种语法格式，分别是：**

**//以判断两者相等作为匹配规则**

**InputIterator find\_first\_of (InputIterator first1, InputIterator last1,**

**ForwardIterator first2, ForwardIterator last2);**

**//以 pred 作为匹配规则**

**InputIterator find\_first\_of (InputIterator first1, InputIterator last1,**

**ForwardIterator first2, ForwardIterator last2,**

**BinaryPredicate pred);**

其中，各个参数的含义如下：

（1）first1、last1：都为输入迭代器，它们的组合 [first1, last1) 用于指定该函数要查找的范围；

（2）first2、last2：都为正向迭代器，它们的组合 [first2, last2) 用于指定要进行匹配的元素所在的范围；

（3）pred：可接收一个包含 2 个形参且返回值类型为 bool 的函数，该函数可以是普通函数（又称为二元谓词函数），也可以是函数对象。

**有关谓词函数，读者可阅读《[C++谓词函数](http://c.biancheng.net/view/1513.html" \t "http://c.biancheng.net/view/_blank)》一节详细了解。**

find\_first\_of() 函数用于在 [first1, last1) 范围内查找和 [first2, last2) 中任何元素相匹配的第一个元素。如果匹配成功，该函数会返回一个指向该元素的输入迭代器；反之，则返回一个和 last1 迭代器指向相同的输入迭代器。  
值得一提的是，不同语法格式的匹配规则也是不同的：

（1）第 1 种语法格式：逐个取 [first1, last1) 范围内的元素（假设为 A），和 [first2, last2) 中的每个元素（假设为 B）做 A==B 运算，如果成立则匹配成功；

（2）第 2 种语法格式：逐个取 [first1, last1) 范围内的元素（假设为 A），和 [first2, last2) 中的每个元素（假设为 B）一起带入 pred(A, B) 谓词函数，如果函数返回 true 则匹配成功。

**注意，当采用第一种语法格式时，如果 [first1, last1) 或者 [first2, last2) 范围内的元素类型为自定义的类对象或者结构体变量，此时应对 == 运算符进行重载，使其适用于当前场景。**

adjacent\_find：adjacent\_find() 函数用于在指定范围内查找 2 个连续相等的元素。该函数的语法格式为：

**//查找 2 个连续相等的元素**

**ForwardIterator adjacent\_find (ForwardIterator first, ForwardIterator last);**

**//查找 2 个连续满足 pred 规则的元素**

**ForwardIterator adjacent\_find (ForwardIterator first, ForwardIterator last,**

**BinaryPredicate pred);**

其中，first 和 last 都为正向迭代器，其组合 [first, last) 用于指定该函数的查找范围；pred 用于接收一个包含 2 个参数且返回值类型为 bool 的函数，以实现自定义查找规则。

**值得一提的是，pred 参数接收的函数既可以定义为普通函数，也可以用函数对象的形式定义。有关谓词函数，读者可阅读《[C++谓词函数](http://c.biancheng.net/view/1513.html" \t "http://c.biancheng.net/view/_blank)》一节详细了解。**

**另外，该函数会返回一个正向迭代器，当函数查找成功时，该迭代器指向的是连续相等元素的第 1 个元素；而如果查找失败，该迭代器的指向和 last 迭代器相同。**

Search search() 函数定义在<algorithm>头文件中，其功能恰好和 find\_end() 函数相反，用于在序列 A 中查找序列 B 第一次出现的位置。  
例如，仍以如下两个序列为例：

**序列 A：1,2,3,4,5,1,2,3,4,5  
序列 B：1,2,3**

可以看到，序列 B 在序列 A 中出现了 2 次。借助 find\_end() 函数，我们可以找到序列 A 中最后一个（也就是第 2 个）{1,2,3}；而借助 search() 函数，我们可以找到序列 A 中第 1 个 {1,2,3}。  
**和 find\_end() 相同，search() 函数也提供有以下 2 种语法格式：**

**//查找 [first1, last1) 范围内第一个 [first2, last2) 子序列**

**ForwardIterator search (ForwardIterator first1, ForwardIterator last1,**

**ForwardIterator first2, ForwardIterator last2);**

**//查找 [first1, last1) 范围内，和 [first2, last2) 序列满足 pred 规则的第一个子序列**

**ForwardIterator search (ForwardIterator first1, ForwardIterator last1,**

**ForwardIterator first2, ForwardIterator last2,**

**BinaryPredicate pred);**

其中，各个参数的含义分别为：

（1）first1、last1：都为正向迭代器，其组合 [first1, last1) 用于指定查找范围（也就是上面例子中的序列 A）；

（2）first2、last2：都为正向迭代器，其组合 [first2, last2) 用于指定要查找的序列（也就是上面例子中的序列 B）；

（3）pred：用于自定义查找规则。该规则实际上是一个包含 2 个参数且返回值类型为 bool 的函数（第一个参数接收 [first1, last1) 范围内的元素，第二个参数接收 [first2, last2) 范围内的元素）。函数定义的形式可以是普通函数，也可以是函数对象。

实际上，第一种语法格式也可以看做是包含一个默认的 pred 参数，该参数指定的是一种相等规则，即在 [first1, last1) 范围内查找和 [first2, last2) 中各个元素对应相等的子序列；而借助第二种语法格式，我们可以自定义一个当前场景需要的匹配规则。

同时，search() 函数会返回一个正向迭代器，当函数查找成功时，该迭代器指向查找到的子序列中的第一个元素；反之，如果查找失败，则该迭代器的指向和 last1 迭代器相同。

serch\_n 和 search() 一样，search\_n() 函数也定义在<algorithm>头文件中，用于在指定区域内查找第一个符合要求的子序列。不同之处在于，前者查找的子序列中可包含多个不同的元素，而后者查找的只能是包含多个相同元素的子序列。

例子：

#include <iostream>

#include <vector>

#include <algorithm>

using namespace *std*;

int *main*()

{

*vector*<int> A{ 1,2,3,4,5,1,2,3,4 ,15,10,1,1,1};

*vector*<int> B{ 1,2,3 };

*vector*<int> C{ 1,1,1 };

*vector*<int>::*iterator* iter;

iter = *find*(A.*begin*(), A.*end*(), 15);

*cout* << \*iter << *endl*;

/\*iter = find(A.begin(), A.end(),mycompare(19));

cout << \*iter << endl;\*/

/\* iter = find\_if\_not(A.begin(), A.end(), A[2] != 10);

cout << \*iter << endl;\*/

iter = *find\_end*(A.*begin*(), A.*end*(), B.*begin*(), B.*end*());

*cout* << \*iter << *endl*;

iter = *find\_first\_of*(A.*begin*(), A.*end*(), B.*begin*(), B.*end*());

*cout* << \*iter << *endl*;

iter = *search*(A.*begin*(), A.*end*(), B.*begin*(), B.*end*());

*cout* << \*iter << *endl*;

iter = *search\_n*(A.*begin*(), A.*end*(), 3, 1);

*cout* << \*iter << *endl*;

}

链表表示线性表

1. 返回表的元素个数 int Size();

template<class T>

int ArrayLineList<T>::Size()

{

return m\_nArrayListSize;

}

1. 根据索引删除元素

void EraseElementFromIndex(int index);

template<class T>

void ArrayLineList<T>::EraseElementFromIndex(int index)

{

checkIndex(index);

*copy*(m\_pelement + index + 1, m\_pelement + m\_nArrayListSize, m\_pelement + index);

m\_pelement[--m\_nArrayListSize].~T();

}

1. 在指定索引插入指定元素

template<class T>

void ArrayLineList<T>::InsertElementFromIndex(int index, T element)

{

if (index<0 || index>m\_nArrayListSize)

throw illegalParameterValue("index is out range");

if (m\_nArrayListSize == m\_nArrayCapacity)

{

changelength1D(m\_pelement, m\_nArrayCapacity, 2 \* m\_nArrayCapacity);//扩容

m\_nArrayCapacity \*= 2;

}

*copy\_backward*(m\_pelement + index, m\_pelement + m\_nArrayListSize, m\_pelement + m\_nArrayListSize + 1);

m\_pelement[index] = element;

m\_nArrayListSize++;

}

**\*学习一下copy\_backward**

**copy\_backward** 不要被 copy\_backward() 算法的名称所误导，它不会逆转元素的顺序。它只会像 copy() 那样复制元素，但是从最后一个元素开始直到第一个元素。  
 copy\_backward() 会复制前两个迭代器参数指定的序列。第三个参数是目的序列的结束迭代器，通过将源序列中的最后一个元素复制到目的序列的结束迭代器之前，源序列会被复制到目的序列中，如图 1 所示。copy\_backward() 的 3 个参数都必须是可以自增或自减的双向迭代器，这意味着这个算法只能应用到序列容器的序列上。

**reverse\_copy()** 算法可以将源序列复制到目的序列中，目的序列中的元素是逆序的。定义源序列的前两个迭代器参数必须是双向迭代器。目的序列由第三个参数指定，它是目的序列的开始迭代器，也是一个输出迭代器。如果序列是重叠的，函数的行为是未定义的。这个算法会返回一个输出迭代器，它指向目的序列最后一个元素的下一个位置。

（8）显示链表void ShowLineList();

template<class T>

void ArrayLineList<T>::ShowLineList()

{

for (int i = 0; i < m\_nArrayListSize; i++)

{

*cout* << m\_pelement[i] << *endl*;

}

}

**链表表示线性表**

先要知道结点的方式，相当于初始化

template<class T>

struct LineListNode {

T m\_element;

LineListNode<T>\* m\_pnext;

//方法

LineListNode() {}

LineListNode(const T& theElement)

{

this->m\_element = theElement;

}

LineListNode(const T& theElement, LineListNode<T>\* thenext)

{

this->m\_element = theElement;

this->m\_pnext = thenext;

}

};

**1 先说构造函数与析构函数**

复制链表的构造函数

template<class T>

LinkLineList<T>::LinkLineList(const LinkLineList<T>& theList)

{

m\_nListSize = theList.m\_nListSize;

if (m\_nListSize == 0)

{

//链表为空

m\_pfirstNode = nullptr;

m\_nListSize = 0;

}

//链表thelist非空

LineListNode<T>\* pSourceNode = theList.m\_pfirstNode;

m\_pfirstNode = new LineListNode<T>(pSourceNode->m\_element);

pSourceNode = pSourceNode->m\_pnext;

LineListNode<T>\* ptargetNode = m\_pfirstNode;

while (pSourceNode != nullptr)

{

//复制剩余元素

ptargetNode->m\_pnext = new LineListNode<T>(pSourceNode->m\_element);

ptargetNode = ptargetNode->m\_pnext;

pSourceNode = pSourceNode->m\_pnext;

}

ptargetNode->m\_pnext = nullptr;

}

**相应的方法**

1. 根据索引查询相应的元素结点

LineListNode<T>\* SearchElementFromIndex(int index);

template<class T>

T LinkLineList<T>::SearchElementFromIndex(int index)

{

checkIndex(index);

LineListNode<T>\* currentNode = m\_pfirstNode;//使用头结点管理

for (int i = 0; i < index-1; i++)

{

currentNode = currentNode->m\_pnext;

}

//此时的currentNode是前驱

return currentNode->m\_pnext->m\_element;

}

注意：index是下标！！类似于数组的

其实我们要注意的是这个节点就是一个元素和一个指针，指针是有指向性的，这个理解是只能意会，类似于数组吧！其实。并且数值也是的序号也是自己分配的。实际是没有序号的！！

1. 根据元素查询序号

因为这个不像数组，可以使用stl的各种算法，所以只能遍历

template<class T>

int LinkLineList<T>::SearchIndexFromElement(T Element)

{

int index = 0;

LineListNode<T>\* pnode = m\_pfirstNode;

while (pnode != nullptr && pnode->m\_element != Element)

{

pnode = pnode->m\_pnext;//根据这种方式遍历

index++;

}

return index;

}

1. 根据序号删除，譬如

template<class T>

void LinkLineList<T>::EraseElementFromIndex(int index)

{

checkIndex(index);

LineListNode<T>\* pdeleteNode;

if (index == 0)

{

pdeleteNode = m\_pfirstNode;

m\_pfirstNode = m\_pfirstNode->m\_pnext;

}

else

{

LineListNode<T>\* p = m\_pfirstNode;//设置前驱

for (int i = 0; i < index-1; i++)

{

p = p->m\_pnext;

}

pdeleteNode = p->m\_pnext;//临时保存待删除结点ai的地址在pdeletNode,以备释放

p->m\_pnext = pdeleteNode->m\_pnext;//将结点\*p指向结点index的直接后续结点

}

m\_nListSize--;//长度要减少一个

delete pdeleteNode;//删除节点

pdeleteNode = nullptr;

}

1. 在指定位置插入指定的元素也是类似的

template<class T>

void LinkLineList<T>::InsertElementFromIndex(int index, T element)

{

checkIndex(index);

if (index == 0)

{

m\_pfirstNode = new LineListNode<T>(element, m\_pfirstNode);

}

else

{

//先找到前驱

LineListNode<T>\* p = m\_pfirstNode;

for (int i = 0; i < index-1; i++)

{

p = p->m\_pnext;

}

p->m\_pnext = new LineListNode<T>(element, p->m\_pnext);

}

m\_nListSize++;

}

1. 显示
2. 创建

template<class T>

LineListNode<T>\* LinkLineList<T>::ArrayToLinkList(T a[], int num)

{

LineListNode<T>\* headNode, \* currentNode;

m\_pfirstNode = headNode = new LineListNode<T>;

headNode->m\_element = a[0];

for (int i = 1; i < num; i++)

{

currentNode = new LineListNode<T>;

currentNode->m\_element = a[i];

currentNode->m\_pnext = nullptr;

headNode->m\_pnext = currentNode;

headNode = currentNode;

}

m\_nListSize = num;

return m\_pfirstNode;

}

**数据结构题解（c++）**

**2.1 数组**

**2.1.1 Remove Duplicates from Sorted Array**

**描述：**

Given a sorted array, remove the duplicates （重复项）in place such that each element appear only once and return the new length.

Do not allocate（分配） extra（额外的） space for another array, you must do this in place with constant（不变） memory（内存）.

For example, Given input array A = [1,1,2],

Your function should return length = 2, and A is now [1,2].

我的翻译：

给一个排序好的数组，删除重复项，使得每个元素只能出现一次，并且返回新的长度，不分配额外的内存。

我的理解：-初始化一个有序数组，在这个基础上删除重复的，计算长度，和处理之后的数组！

我的做法：

*size\_t* CLeeteCodeArray::RemoveDuplicatesFromSortedArray(*std*::*vector*<int>& numVec)

{

//因为是已经排序好了的，那么可以用类似冒泡方式

*size\_t* index = 0;

*size\_t* nNumVecSize = numVec.*size*();

for (*size\_t* i = 0;i<nNumVecSize;i++)

{

//就是一个一个对比，如果是相等的，那么就直接赋值替代

if (numVec[i] == numVec[++index])

{

numVec[++i] = numVec[++index];

nNumVecSize--;

}

//index++;

}

return numVec.*empty*() ? 0 : nNumVecSize;

}

**这种做法是错误的！**

*size\_t* CLeeteCodeArray::RemoveDuplicatesFromSortedArray(*std*::*vector*<int>& numVec)

{

//因为是已经排序好了的，那么可以用类似冒泡方式

*size\_t* j = 0;

*size\_t* n = numVec.*size*();

for (*size\_t* i = 0; i < n; ++i)

{

if (numVec[i] != numVec[j])

numVec[++j] = numVec[i];

}

return numVec.*empty*() ? 0 : (j + 1);

}

**错题总结！**

1. 题目的意思弄错了，这里其实是不要去除元素，就是把不同的元素先排列出来，并且计算不同的元素是多少个！
2. Size\_t 与int的区别。
3. 我感觉这个题有点像**线性表-数组描述**的基本操作。可以熟悉一下线性表的排序，去除元素等操作！在指定位置增删改的操作。可以复习一下stl中相关的算法与容器。

int CLeeteCodeArray::RemoveDuplicatesFromSortedArray(*std*::*vector*<int>& numVec)

{

//因为是已经排序好了的，那么可以用类似冒泡方式

if (numVec.*size*() == 0)

return 0;

int i = 0;

int j = 0;

while (j!=numVec.*size*())

{

if (numVec[i] == numVec[j])

{

j++;

}

else {

numVec[i+1] = numVec[j];

i++;

j++;

}

}

numVec.*erase*(numVec.*begin*() + i + 1, numVec.*end*());

return numVec.*size*();

}

int CLeeteCodeArray::RemoveDuplicatesFromSortedArray(*std*::*vector*<int>& numVec)

{

int n;

n = *distance*(numVec.*begin*(), *unique*(numVec.*begin*(), numVec.*end*()));

numVec.*erase*(numVec.*begin*() + n, numVec.*end*());

return numVec.*size*();

}

删除了排序之后的元素！

只是赋值是不会删除元素个数的！

原理就是因为是有序的，所以呢，就是一个元素一个元素比下去。比到不一样的就赋值到下一个元素。

**知识点：**

1. **distance() 函数用于计算两个迭代器表示的范围内包含元素的个数。**

**其语法格式如下：**

**template<class InputIterator>  
  typename iterator\_traits<InputIterator>::difference\_type distance (InputIterator first, InputIterator last);**

其中，first 和 last 都为迭代器，其类型可以是**输入迭代器、前向迭代器、双向迭代器以及随机访问迭代器**；该函数会返回[first, last)范围内包含的元素的个数。  
注意，first 和 last 的迭代器类型，直接决定了 distance() 函数底层的实现机制：

（1）当 first 和 last 为**随机访问迭代器**时，distance() 底层直接采用 last - first 求得 [first, last) 范围内包含元素的个数，其时间复杂度为O(1)常数阶；

（2）当 first 和 last 为**非随机访问迭代器**时，distance() 底层通过不断执行 ++first（或者 first++）直到 first==last，由此来获取 [first, last) 范围内包含元素的个数，其时间复杂度为O(n)线性阶。  
distance() 函数定义在<iterator>头文件中

#include <iterator>

using namespace std;  
2. unique()函数属于STL中比较常用函数，它的功能是元素去重。**即”删除”序列中所有相邻的重复元素(只保留一个)。此处的删除，并不是真的删除，而是指重复元素的位置被不重复的元素给占领了。由于它”删除”的是相邻的重复元素，所以在使用unique函数之前，一般都会将目标序列进行排序。返回值是一个迭代器，它指向的是去重后容器中不重复序列的最后一个元素的下一个元素。**

**unique函数通常和erase函数一起使用，来达到删除重复元素的目的。(注：此处的删除是真正的删除，即从容器中去除重复的元素，容器的长度也发生了变换；而单纯的使用unique函数的话，容器的长度并没有发生变化，只是元素的位置发生了变化)**

**2.1.2 Remove Duplicates from Sorted Array II**

**描述：**

Follow up for ”Remove Duplicates”: What if duplicates are allowed at most twice?

For example, Given sorted array A = [1,1,1,2,2,3],

Your function should return length = 5, and A is now [1,1,2,2,3]

**翻译：**略

**我的理解：**本题与上一题的区别就是允许重复两次。

那么这个题就要在上面的基础上添加一个计数器。

**我的做法：是错的！**

**正确的做法:**

int CLeeteCodeArray::RemoveDulicatesII(*std*::*vector*<int>& nums)

{

if (nums.*size*() <= 2) return nums.*size*();

int index = 2;

for (int i = 2; i < nums.*size*(); i++) {

if (nums[i] != nums[index - 2])

nums[index++] = nums[i];

}

nums.*erase*(nums.*begin*() + index, nums.*end*());

return nums.*size*();

}

我个人更喜欢下面的形似

int CLeeteCodeArray::RemoveDulicatesII(*std*::*vector*<int>& nums)

{

if (nums.*size*()< 3)

return nums.*size*();

int i = 1;

int j = 2;

while (j != nums.*size*())

{

if (nums[i-1] == nums[j])

{

j++;

}

else {

nums[++i] = nums[j];

j++;

}

}

nums.*erase*(nums.*begin*() + i, nums.*end*());

return nums.*size*();

}

**注意一下：i-1与++i之间正好是i元素！i = 1,则是第二个人元素开始！**

**特别注意下i++,++i和i+1的区别哦！前面两种可是把i值都变了，i+1没有变i值**

**假设一下，如果是允许出现三次呢？**

**这段代码应该怎么改？**

int CLeeteCodeArray::RemoveDulicatesN(*std*::*vector*<int>& nums, const int N)

{

if (nums.*size*() < N+1)

return nums.*size*();

int i = N-1;

int j = N;

while (j != nums.*size*())

{

//i-N+1 =0是初始值，j是相隔值

if (nums[i - N+1] == nums[j])

{

j++;

}

else {

nums[++i] = nums[j];

j++;

}

}

nums.*erase*(nums.*begin*() + i+1, nums.*end*());

return nums.*size*();

}

注意：这个函数是允许元素重复N次的去除重复项！

**2.1.3 Search in Rotated Sorted Array**

**描述：**

Suppose（假设） a sorted array is rotated（旋转） at some pivot（枢） unknown to you beforehand（预先）. (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume（承担） no duplicate exists in the array

**我的翻译：不知道什么意思！<----\_\_\_<------**

假设已排序的数组以事先未知的某个轴旋转。 （即0 1 2 4 5 6 7可能变成4 5 6 7 0 1 2）。 将为您提供要搜索的目标值。 如果在数组中找到，则返回其索引，否则返回-1。 您可以假定数组中不存在重复项。

**我的理解：没明白什么意思啊！，看了下代码明白了这道题是什么意思了，就是一个没有重复的数组，输入一个值，返回这个值所在位置，如果没有找到，那么就返回-1.尝试着自己写一下。当然最暴力的当然是遍历。答案给的是二分法。自己实现一下。**

**答案：**

int CLeeteCodeArray::SearchSelf(*std*::*vector*<int> numVec, int targetVal)

{

//因为使用的是二分法，那么就得有一个中间值位子

int frist = 0;

int last = numVec.*size*();

//很明显二分法的精髓就是通过变化长度来查询,那么就是位置，

//也就是说必须是frist==last那么这个位置也知道了

while (frist != last)

{

const int mid = (last + frist) / 2;

//分成三种情况了

//这三种情况怎么分呢？

//就是把这个数组通过frist,mid,last分成两段，

//在这两段里面寻找target

if (numVec[mid] == targetVal)//中间点的情况

return mid;

if (numVec[mid]>=numVec[frist])//纯粹就是分开对半，物理的分开，判断不出来是前还是后

{

if (numVec[frist] < targetVal && targetVal < numVec[mid])

last = mid;

else

frist = mid + 1;

}

else

{

if (numVec[last-1]>targetVal &&targetVal>numVec[mid])

frist = mid + 1;

else

last = mid;

}

}

//如果经过上面的循环还是无法找到，那么就返回-1

return -1;

}**这道题自己可以复现，当时的想法就是把它当着一个有序的写，因为不管有序还是无序，都能实现，那么我就拿最简单的实现，有问题，在特殊情况特殊处理的想法。当时写代码时候的思路。一定要有逻辑。我发现用键值对也可以解决这个问题。就是不知道复杂度和时间是不是允许。**

**2.1.4 Search in Rotated Sorted Array II**

**描述：**

Follow up for ”Search in Rotated Sorted Array”: What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

我的理解：这道题的条件是数组允许有重复元素。那么按我上面的那种想法就不行了，当然了用遍历的方式是可以的，先找到，将相应的下标值放入到数组中就可以。但是有没有更加好的方法呢？

bool CLeeteCodeArray::SearchDupVa(*std*::*vector*<int> nums, int target)

{

int first = 0;

int last = numVec.*size*();

*std*::*vector*<int> IndexVec;

while (first != last)

{

const int mid = first + (last - first) / 2;

if (nums[mid] == target)

return true;

if (nums[first] < nums[mid])

{

if (nums[first] <= target && target < nums[mid])

last = mid;

else

first = mid + 1;

}

else if (nums[first] > nums[mid])

{

if (nums[mid] < target && target <= nums[last - 1])

first = mid + 1;

else

last = mid;

}

**else**

**//skip duplicate one**

**first++;**

}

return false;

}

**我的想法：这种做法只能判断有还是没有，并不能做到具体位置，和多少个。除了使用遍历法，还有没有其他的方法？看下stl中关于find，search，count，binary\_search, lower\_bound,upper\_bound和equal\_range等知识！**

**（1）Find：**find() 函数本质上是一个模板函数，用于在指定范围内查找和目标元素值相等的第一个元素。可以用在很多地方！  
**如下为 find() 函数的语法格式：**

**InputIterator find (InputIterator first, InputIterator last, const T& val);**

**其中，first 和 last 为输入迭代器，[first, last) 用于指定该函数的查找范围；val 为要查找的目标元素。**

正因为 first 和 last 的类型为输入迭代器，因此该函数适用于所有的序列式容器。

另外，该函数会返回一个输入迭代器，当 find() 函数查找成功时，其指向的是在 [first, last) 区域内查找到的第一个目标元素；如果查找失败，则该迭代器的指向和 last 相同。  
 值得一提的是，find() 函数的底层实现，其实就是用==运算符将 val 和 [first, last) 区域内的元素逐个进行比对。这也就意味着，[first, last) 区域内的元素必须支持==运算符。

**2.1.5 Median of Two Sorted Arrays**

**描述**

There are two sorted arrays A and B of size m and n respectively(分别地). Find the median（中值） of the two sorted arrays. The overall（全部） run time complexity should be O(log(m + n)).

我的翻译：两个有序的数组分别为m,n个，寻找两个数组的中值，全部用的时间复杂度是log(m+n).

**我的理解与疑问：第一想法就是将两个数组合并起来，然后对新数组进行排序，想要多大的都可以。但是呢，就没有用到有序这个特性了，第二想法是两个数组逐个比对，也是可以A和B数组的一个一个比对。但是我看答案使用了第三种方法，我浏览了一下，他有个前提，就是m和n都比k/2大。假如m,n其中有一个（甚至两个都）没有k大呢？**

我自己写了前面两个!

答案的我还没明白！

**2.1.6 Longest Consecutive Sequence**

**描述：**

Given an unsorted array of integers(整数), find the length of the longest consecutive(连续的) elements sequence(序列). For example, Given [100, 4, 200, 1, 3, 2], The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(n) complexity

**我的翻译：给一无序的数组，寻找最长的连续长度。**

**我的思路和疑问：在无序的数组中寻找连续的元素长度，有个疑问，如果是重复的呢？怎么计算？我的想法是先排序，再比对。**

**答案：**

如果允许O(nlogn)的复杂度，那么可以先排序，可是本题要求O(n).

由于序列里的元素是无序的，又要求O(n)，首先要想到用哈希表。

用一个哈希表unordered\_map<int,bool>used 记录每个元素是否使用，对每个元素，以该元素为中心，往左右扩张，直到不连续为止，记录下最长的长度。

直接看答案吧！

对unordered\_map的使用很不熟悉！好好学习一下！！

2.1.7 Two Sum

描述：

Given an array of integers（整数）, find two numbers such that they add up to a specific target number. The function twoSum should return indices（序列号） of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

Input: numbers={2, 7, 11, 15}, target=9

Output: index1=1, index2=2

谷歌翻译：给定一个整数数组，找到两个数字，使它们加起来成为一个特定的目标数字。 函数twoSum应该返回两个数字的索引，以使它们加起来等于目标，其中index1必须小于index2。 请注意，您返回的答案（index1和index2）都不基于零。 您可以假设每个输入都只有一个解决方案。 输入：数字= {2，7，11，15}，目标= 9 输出：index1 = 1，index2 = 2

我的理解与思路：因为第二个序列号比第一个序列号大，那么就搞一个双层for，里面一层int j = i；开始一直到结尾。我将这两个序列号放入到数组里面返回出来。

我的答案：就是暴力，超时了，不符合要求！

正确答案：

2.1.8 3Sum

描述：

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique

Triplets(三个元素) in the array which gives the sum of zero.

Note:

• Elements in a triplet (a, b, c) must be in non-descending order. (ie, a ≤ b ≤ c)

• The solution set must not contain（包含） duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4}.

A solution set is:

(-1, 0, 1)

(-1, -1, 2)

谷歌翻译：给定n个整数组成的数组S，S中是否有元素a，b，c使得a + b + c = 0？ 找到所有独特的 数组中的Triplets（三个元素），得出的总和为零。

注意： •三元组（a，b，c）中的元素必须以降序排列。 （即a≤b≤c）

•解决方案集不得包含（包含）重复的三胞胎。 例如，给定数组S = {-1 0 1 2 -1 -4}。 一个解决方案集是： （-1，0，1） （-1，-1、2）

题意的理解：

1. 条件是查询一个数组中有三个元素满足a+b+c =0
2. 并且这个满足条件是唯一解。
3. 元素的排序是a<b<c

在前面做过一道题，去除有序的数组重复项，

**我的代码实现思路；**

第一步：先排序

第二步：去除重复项

第三步：寻找这三个元素

第四步：返回这数组

答案：

解题思路：

1. 根据第一点要求： 因为要求每个答案数组中的元素是升序排列的，所以在开头我们要对数组进行排 序。

2. 根据第二点要求： 因为不能包含重复的答案数组，所以我们要在代码里面做一切去掉重复的操作，对 于数组，这样的操作是相同的。最开始我做leetcode的时候是把所有满足条件的答案数组存起来，之后 再用map进行处理，感觉那样太麻烦了，所以这次给出的答案是不需要额外空间的。

**时间复杂度分析：**

对于这道题，因为是要找三个元素，所以怎样都要O(n2)的时间复杂度，目前我没有想出来O(n)时间复杂度 的解法。

归根结底，其实这是two pointers的想法，定位其中两个指针，根据和的大小来移动另外一个。解题中所要注意的就是一些细节问题。好了，上代码吧。

std::vector<std::vector<int>> LeetCode::ThreeSum(std::vector<int>& num)

{

std::vector<std::vector<int>>Ret;

if (num.size()<2)

{

return Ret;

}

//排序

sort(num.begin(), num.end());

//遍历

for (int i =0;i<num.size()-2;i++)

{

int j = i + 1;

int k = num.size() - 1;

while (j<k)

{

std::vector<int> curr;

if (num[i]+num[j]+num[k] == 0)

{

curr.push\_back(num[i]);

curr.push\_back(num[j]);

curr.push\_back(num[k]);

Ret.push\_back(curr);

++j;

--k;

while (j<k && num[j-1] == num[j])

{

++j;

}

while (j<k && num[k] == num[k+1])

{

--k;

}

}

else if (num[i] +num[j] +num[k]<0)

{

//小于0的话那么就要往后移动

++j;

}

else

{

//大于0说明过大，k就得减小

--k;

}

//防止重样

while (i<num.size()-1 && num[i] == num[i+1])

{

++i;

}

}

}

return Ret;

}

2.1.11 Remove Element

描述：

Given an array and a value, remove all instances（实例） of that value in place and return the new length. The order（顺序） of elements can be changed. It doesn’t matter what you leave beyond the new length.

谷歌翻译：给定一个数组和一个值，删除该值的所有实例并返回新的长度。 元素的顺序可以更改。 超出新长度的内容都没关系。

1. Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You man assume that each input would have exactly one solution.

我的翻译：确定一个数组S，一个确定的目标值，在S中找出3个元素的和与这个给定的值最小。Input只有一个有效答案。

我的理解：没理解这个题的意思。

答案：题目分析：

1. 最短距离：两个整数的最短距离是0.这点对于这道题比较重要，别忽略。

2. 这道题和3Sum几乎同出一辙，所以方便于解题，我们还是在开头要对数组进行排序，不然没法定位指针移动。

3. 另外，这道题中用到了INT\_MAX这个值，这个值和 INT\_MIN是相对应的，在很多比较求最大值最小值的情况，经常用这两个变量。

解题思路：

这道题的解题方法和3Sum几乎相同，设定三个指针，固定两个，根据和的大小移动另外一个。属于这道题 目自己的东西就是distance比较这块儿，建立一个tmp distance和min distance比较。

时间复杂度分析：

这道题目和3Sum几乎是一个思路，所以时间复杂度为O(n2)。

代码：

int LeetCode::threeSumClosest(std::vector<int>& num, int target)

{

if (num.size()<2)

{

return -1;

}

int Ret = 0;

int distance = INT\_MAX;

sort(num.begin(),num.end());

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

{

int j = i + 1;

int k = num.size() - 1;

while (j<k)

{

int tmp\_val = num[i] + num[j] + num[k];

int temp\_distance;

if (tmp\_val<target)

{

temp\_distance = target - tmp\_val;

if (temp\_distance<distance)

{

distance = temp\_distance;

Ret = num[i] + num[j] + num[k];

}

++j;

}

else if (tmp\_val>target)

{

temp\_distance = tmp\_val - target;

if (temp\_distance<distance)

{

distance = temp\_distance;

Ret = num[i] + num[j] + num[k];

}

--k;

}

else

{

Ret = num[i] + num[j] + num[k];

return Ret;

}

}

}

return Ret;

}

1. Given an array S of n integers, are there elements a, b, c and d in S such that a+b+c+d = target. Find all unique quadruplets in the array which gives the sume of target.

Note:

1. Elements in quadruplets (a, b, c, d) must be in non-descending order. (ie, a<=b<=c<=d)

2. The solution must not contain duplicates quadruplets.

翻译题：

给定一个整型数字数组num和一个目标值target，求出数组中所有的组合满足条件：

num[a]+num[b]+num[c]+num[d] = target.

并且要满足的条件是：

1. num[a] <= num[b] <= num[c] <= num[d]

2. 答案中的组合没有重复的.

我的理解：

时间复杂度是O(n3).我自己试试。

我的答案：

std::vector<std::vector<int>> LeetCode::fourSum(std::vector<int>& num, int target)

{

std::vector<std::vector<int>>Ret;

if (num.size()<2)

{

return Ret;

}

//排序

sort(num.begin(), num.end());

//遍历

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

{

int j = i + 1;

int k = num.size() - 2;

int m = num.size() - 1;

while (j < k < m)

{

std::vector<int> curr;

if (num[i] + num[j] + num[k] + num[m] == target)

{

curr.push\_back(num[i]);

curr.push\_back(num[j]);

curr.push\_back(num[k]);

curr.push\_back(num[m]);

Ret.push\_back(curr);

++j;

--k;

--m;

while (j < k < m && num[j - 1] == num[j])

{

++j;

}

while (j < k < m && num[k] == num[k + 1])

{

--k;

}

while (j < k < m && num[m] == num[m + 1])

{

--m;

}

}

else if (num[i] + num[j] + num[k] + num[m] < target)

{

++j;

}

else

{

--k;

--m;

}

while (i<num.size()-2&&num[i] == num[i+1])

{

++i;

}

}

}

return Ret;

}

//答案

*std*::*vector*<*std*::*vector*<int>> LeetCode::fourSum(*std*::*vector*<int>& num, int target)

{

*std*::*vector*<*std*::*vector*<int>>Ret;

if (num.*size*()<3)

{

return Ret;

}

*sort*(num.*begin*(), num.*end*());

for (int i =0;i<num.*size*();i++)

{

if (i>0&&num[i] == num[i-1])

{

continue;

}

for (int j = i+1;j<num.*size*()-2;++j)

{

//这里的逻辑错了!去网上找一下其他方式！！

if (j>i+1&&num[j] == num[j-1])

{

continue;

}

int k = j + 1;

int m = num.*size*() - 1;

while (k<m)

{

int sum = num[i] + num[j] + num[k] + num[m];

if (sum == target)

{

*std*::*vector*<int> curre;

curre.*push\_back*(num[i]);

curre.*push\_back*(num[j]);

curre.*push\_back*(num[k]);

curre.*push\_back*(num[m]);

Ret.*push\_back*(curre);

do

{

++k;

} while (k<m&&num[k] == num[k-1]);

do

{

m--;

} while (k<m&&num[m]==num[m+1]);

}

else if (sum<target)

{

++k;

}

else

{

--m;

}

}

}

}

return Ret;

}

10 Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2). Find the minimum element.

You may assume no duplicate exists in the array。

我的翻译：假设一个排序后的数组以我们不知道的枢轴旋转（如 0,1,2,4,5，6,7转成4,5,6,7，0,1，2），寻找到最小的值，假设数组中不存在副本。

我的理解：说是有序，但是基本上就是无序的。要时间少点，可不可以这样认为，第一个元素与后面的所有的对比，如果是比第一个元素小的话，那么就是最小的值了。

做题的思路：

先将一个数组排序，然后将其旋转，之后第一个元素与后面的元素的对比。

Vector没有排序的算法吗？

如果有的话，从新排序不就可以了？（狗头保命！！！）

我的答案：

int LeetCode::FindMin(*vector*<int>& num)

{

int size = num.*size*();

if (size == 0)

{

return 0;

}

else if (size == 1)

{

return num[0];

}

else if (size == 2)

{

return *min*(num[0],num[1]);

}

for (int i = 0; i < size-1; i++)

{

if (num[0]<num[i])

{

continue;

}

else

{

return num[i];

}

}

}

很明显它的时间是n。

我看答案有用二分法

//答案，二分法

int LeetCode::FindMins(vector<int>& nums)

{

int size = num.size();

if (size == 0)

{

return 0;

}

else if (size == 1)

{

return num[0];

}

else if (size == 2)

{

return min(num[0], num[1]);

}

int nStart;

int stop = size - 1;

while (nStart <stop-1)

{

if (nums[nStart]<nums[stop])

{

return nums[nStart];

}

int mid = (nStart + stop) / 2;

if (nums[mid]>nums[nStart])

{

nStart = mid;

}

else if (nums[mid]<nums[nStart])

{

stop = mid;

}

}

return min(nums[nStart],nums[stop]);

}

11 Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2). Find the minimum element.

**The array may contain duplicates.**

我的理解：这道题与上道题的区别就是有可能会有重复的，那么我们可以先把重复的删除掉！

答案不是这样的

我们直接看答案：

将重复的情况考虑了！

int LeetCode::FindMinss(*vector*<int>& nums)

{

int size = nums.*size*();

if (size == 0)

{

return 0;

}

else if (size == 1)

{

return nums[0];

}

else if (size == 2)

{

return *min*(nums[0], nums[1]);

}

int nStart = 0;

int stop = size - 1;

while (nStart < stop - 1)

{

if (nums[nStart] < nums[stop])

{

return nums[nStart];

}

int mid = (nStart + stop) / 2;

if (nums[mid] > nums[nStart])

{

nStart = mid;

}

else if (nums[mid] < nums[nStart])

{

stop = mid;

}

//增加的情况

else

{

//不做处理，直接跳出来

nStart++;

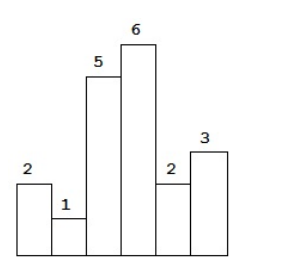
}

}

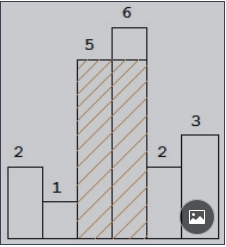
return *min*(nums[nStart], nums[stop]);

}

12 Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].



The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example, Given height = [2,1,5,6,2,3], return 10.

翻译：给定n个非负整数，表示直方图的柱高，其中每个柱宽是1，求直方图中最大的面积。最大的矩形显示在阴影区域，面积是10，假定给定的高度 = [2,1,5,6,2,3],则返回10.

我的理解：

面积是n个连在一起的柱，乘以这连在一起的几个数的最小值。必须是连续的！

答案 ：

int CLeeteCode::LargestRect(std::vector<int>& nHeight)

{

//先定义面积

int max\_area = 0;

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

{

int MinValue = nHeight[i];

for (int j = i; j < nHeight.size(); j++)

{

MinValue = std::min(nHeight[j], MinValue);

int CurArea = MinValue \* (j - i + 1);

if (CurArea > max\_area)

{

max\_area = CurArea;

}

}

}

return max\_area;

}

时间复杂度不一样！！

int CLeeteCode::LargetAreas(std::vector<int>& nheight)

{

int ret = 0;

std::stack<int>stk;

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

{

if (stk.empty()||stk.top()<=nheight[i])

{

stk.push(nheight[i]);

}

else

{

int count = 0;

while (!stk.empty()&&stk.top()>nheight[i])

{

count++;

ret = std::max(ret,stk.top()\*count);

stk.pop();

}

while (count --)

{

stk.push(nheight[i]);

}

stk.push(nheight[i]);

}

}

int count = 1;

while (!stk.empty())

{

ret = std::max(ret,stk.top()\*count);

stk.pop();

count++;

}

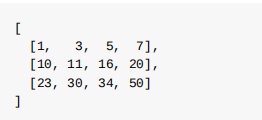
return ret;

}

1. Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

Integers in each row are sorted from left to right. The first integer of each row is greater than the last integer of the previous row. For example,

Consider the following matrix:



这道题就是类似比大小

bool CLeeteCode::searchMatrix(*std*::*vector*<*std*::*vector*<int>>& matrix, int target)

{

if (matrix.*size*() == 0)

{

return false;

}

if (matrix[0].*size*() == 0)

{

return false;

}

int rowNumber = 0;

int colNumber = matrix[0].*size*() - 1;

while (rowNumber < matrix.*size*() && colNumber >= 0)

{

if (target < matrix[rowNumber][colNumber])

{

--colNumber;

}

else if (target > matrix[rowNumber][colNumber])

{

++rowNumber;

}

else

return true;

}

return false;

}

14，Given a sorted array of integers, find the starting and ending position of a given target value. Your algorithm's runtime complexity must be in the order of O(log n).

If the target is not found in the array, return [-1, -1].

For example,

Given [5, 7, 7, 8, 8, 10] and target value 8,

return [3, 4]

std::vector<int> CLeeteCode::SearchRang(int A[], int n, int target)

{

if (n == 0)

{

return std::vector<int>({ -1,-1 });

}

std::vector<int> v;

int low = 0;

int high = n - 1;

while (low <= high)

{

int mid = (low + high) / 2;

if (A[mid] >= target)

{

high = mid - 1;

}

else

{

low = mid + 1;

}

}

if (low < n && A[low] == target)

{

v.push\_back(low);

}

else

{

return std::vector<int>({ -1,-1 });

}

low = low;//很明显这个地方有问题，有时间要看下，修改掉

high = n - 1;

while (low <= high)

{

int mid = (low + high) / 2;

if (A[mid] <= target)

{

low = mid + 1;

}

else

{

high = mid - 1;

}

}

v.push\_back(high);

return v;

}

利用二分法进行

15 Given an integer, write a function to determine if it is a power of two.

bool CLeeteCode::IsPowerOfTwo(int n)

{

if (n<1)

{

return false;

}

if (n>1&&((n&(n-1))==0))

{

return true;

}

}

16 Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing

from the array.

For example, Given nums = [0, 1, 3] return 2 .

Note: Your algorithm should run in linear runtime complexity. Could you implement it using only

constant extra space complexity?

int missingNumber(vector<int>& nums) {

int res = nums.size();

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

res ^= i^ nums[i];

}

return res;

}

还是不懂！！！这个怎么来的！！！

好好整理一下之前的16道题

17 . Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

我对题的理解：

给定一个只是包含0，和1的矩阵，求包含1的最大面积矩形。也就是说，第一，被包含的是一个矩形，并且是全部是1.

答案是利用了12题的柱状题！这个可以试着理解一下

int CLeeteCode::LargestRect(std::vector<int>& nHeight)

{

//先定义面积

int max\_area = 0;

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

{

int MinValue = nHeight[i];

for (int j = i; j < nHeight.size(); j++)

{

MinValue = std::min(nHeight[j], MinValue);

int CurArea = MinValue \* (j - i + 1);

if (CurArea > max\_area)

{

max\_area = CurArea;

}

}

}

return max\_area;

}

时间复杂度不一样！！

int CLeeteCode::LargetAreas(std::vector<int>& nheight)

{

int ret = 0;

std::stack<int>stk;

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

{

if (stk.empty()||stk.top()<=nheight[i])

{

stk.push(nheight[i]);

}

else

{

int count = 0;

while (!stk.empty()&&stk.top()>nheight[i])

{

count++;

ret = std::max(ret,stk.top()\*count);

stk.pop();

}

while (count --)

{

stk.push(nheight[i]);

}

stk.push(nheight[i]);

}

}

int count = 1;

while (!stk.empty())

{

ret = std::max(ret,stk.top()\*count);

stk.pop();

count++;

}

return ret;

}

主要的思路是一样的，但是情况的不同。仔细分析一下！

如果给的数组是升序的话

那么就是随着遍历，被压入栈中的顶，一定是小于数组的。

if (stk.empty()||stk.top()<=nheight[i])

{

stk.push(nheight[i]);

}

然后就是无序的情况了。但是使用栈的目的是为了保持这样一个升序的。

如2,1，5,6,2,3

17题答案是下面的

int CLeeteCode::maximalRectangle(*std*::*vector*<*std*::*vector*<char>>& matrix)

{

if (matrix.*size*() == 0)

{

return 0;

}

int rows = matrix.*size*();

int cols = matrix[0].*size*();

*std*::*vector*<int> temp(cols, 0);//5个0

int sum = 0;

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < cols; j++)

{

temp[j] = matrix[i][j] - '0' + (matrix[i][j] - '0') \* temp[j];

}

//sum = std::max(sum, LargestRect(temp));

sum = *std*::*max*(sum, LargetAreas(temp));

}

return sum;

}

1. Determine whether an integer is a palindrome. Do this without extra space.

回文数

一个数字题

bool CLeeteCode::IsPalindrome(int x)

{

if (x<0)

{

return false;

}

else if (x == 0)

{

return true;

}

else

{

int tmp = x;

int y = 0;

while (tmp != 0)

{

y = y \* 10 + tmp % 10;

tmp = tmp / 10;

}

if (y == x)

{

return true;

}

else

{

return false;

}

}

1. Given inorder and postorder traversal of a tree, construct the binary tree.

知道一个二叉树的中序和后序的遍历结果，构造一个二叉树

TreeNode\* CLeeteCode::buildTree(*std*::*vector*<int>& inorder, *std*::*vector*<int>& postorder)

{

*std*::*unordered\_map*<int, int>m;

if (postorder.*empty*())

{

return nullptr;

}

for (int i =0;i<inorder.*size*();i++)

{

m[inorder[i]] = i;

}

return build(inorder,0,inorder.*size*()-1,postorder,0,postorder.*size*()-1);

}

TreeNode\* CLeeteCode::build(*std*::*vector*<int>& inorder, int s0, int e0, *std*::*vector*<int>& postorder, int s1, int e1)

{

*std*::*unordered\_map*<int, int>m;

if (s0>e0||s1>e1)

{

return nullptr;

}

TreeNode\* root = new TreeNode(postorder[e1]);

int mid = m[postorder[e1]];

int num = mid - s0;

root->*left* = build(inorder,s0,mid-1,postorder,s1,s1+num-1);

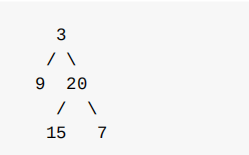
root->*right* = build(inorder, mid + 1, e0, postorder, s1 + num, e1 - 1);

return root;

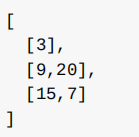
}

20 .Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example: Given binary tree {3,9,20,#,#,15,7}

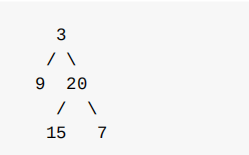


return its level order traversal as:

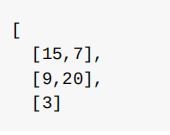


21 ..Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example: Given binary tree {3,9,20,#,#,15,7}



return its level order traversal as:



对比一下这两道题就是在返回值的时候将指颠倒！

主要是算法是一样的！这是对其算法，用递归的方式。

void CLeeteCodeTree::LevelBottom(TreeNode\* node, int level, *std*::*vector*<*std*::*vector*<int>>& res)

{

if (!node)

{

return;

}

if (res.*size*() == level)

{

res.*push\_back*({});

}

res[level].*push\_back*(node->val);

if (node->*left*)

{

LevelBottom(node->*left*, level + 1, res);

}

if (node->*right*)

{

LevelBottom(node->*left*, level + 1, res);

}

}

对于20题的使用

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeTree::leveOrder(TreeNode\* root)

{

*std*::*vector*<*std*::*vector*<int>> res;

leveloader(root, 0, res);

return res;

}

对于21题的使用

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeTree::LevelOrderBottom(TreeNode\* root)

{

*std*::*vector*<*std*::*vector*<int>>res;

LevelBottom(root,0,res);

return *std*::*vector*<*std*::*vector*<int>>(res.*rbegin*(),res.*rend*());

}

非递归的方式

20题

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeTree::LevelOrdersss(TreeNode\* root)

{

if (!root)

{

return {};

}

*std*::*vector*<*std*::*vector*<int>> res;

*std*::*queue*<TreeNode\*>q{ {root} };

while (!q.*empty*())

{

*std*::*vector*<int>oneLevel;

for (int i = q.*size*();i>0;--i)

{

TreeNode\* t = q.*front*();

q.*pop*();

oneLevel.*push\_back*(t->val);

if (t->left)

{

q.*push*(t->left);

}

if (t->right)

{

q.*push*(t->right);

}

}

res.*push\_back*(oneLevel);

}

return res;

}

21题

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeTree::LevelOrderBtm(TreeNode\* root)

{

if (!root)

{

return {};

}

*std*::*vector*<*std*::*vector*<int>> res;

*std*::*queue*<TreeNode\*>q{ {root} };

while (!q.*empty*())

{

*std*::*vector*<int>oneLevel;

for (int i = q.*size*(); i > 0; --i)

{

TreeNode\* t = q.*front*();

q.*pop*();

oneLevel.*push\_back*(t->val);

if (t->left)

{

q.*push*(t->left);

}

if (t->right)

{

q.*push*(t->right);

}

}

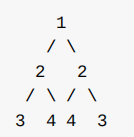
res.*insert*(res.*begin*(),oneLevel);

}

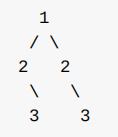
return res;

}

22.Given a binary tree, check whether it is a mirror of itself(ie, symmetric around its center) For example, this tree is symmetric:



But the following tree is not.



主要是判断一颗树是不是自己的镜像。

三个条件

1. 左右两个节点是不是一样
2. 左节点的左孩子是否是与右节点的右孩子相同
3. 左节点的右孩子是否与右节点的左孩子相同

如果都满足条件，那么就是镜像。返回true，如果不满足那么就返回false

两种解法：

递归和循环/迭代

递归：

bool CLeeteCodeTree::IsMirrorTree(TreeNode\* Root)

{

if (Root == nullptr)

{

return true;

}

return Helper(Root->left,Root->right);

}

bool CLeeteCodeTree::Helper(TreeNode\* left, TreeNode\* right)

{

if (left == nullptr && right == nullptr)

{

return true;

}

else if(left == nullptr || right == nullptr)

{

return false;

}

bool cond1 = left->val == right->val;

bool cond2 = Helper(left->left, right->right);

bool cond3 = Helper(left->right, right->left);

return cond1 && cond2 && cond3;

}

循环/迭代

bool CLeeteCodeTree::IsMirror(TreeNode\* Root)

{

if (Root == nullptr)

{

return true;

}

*std*::*queue*<TreeNode\*> q1, q2;

q1.*push*(Root->left);

q2.*push*(Root->right);

while (!q1.*empty*() && !q2.*empty*())

{

TreeNode\* node1 = q1.*front*();

q1.*pop*();

TreeNode\* node2 = q2.*front*();

q2.*pop*();

if (!node1 && !node2)

{

continue;

}

if ((node1 && !node2) || (!node1 && node2))

{

return false;

}

if (node1->val != node2->val)

{

return false;

}

q1.*push*(node1->left);

q1.*push*(node1->right);

q2.*push*(node2->left);

q2.*push*(node2->right);

}

return true;

}

1. Given two binary trees, write a function to check if they are equal or not. Two binary trees are considered equal if they are structurally identical and the nodes have the same values.

判断两棵树是不是一样的，那么就要判断两棵树的结构和相应的值都要一样。

bool CLeeteCodeTree::IsSameTree(TreeNode\* p, TreeNode\* q)

{

if (p ==nullptr &&q==nullptr)

{

return true;

}

else if (p == nullptr ||q == nullptr)

{

return false;

}

if (p->val == q->val)

{

bool left = IsSameTree(p->*left*, q->*left*);

bool rigt = IsSameTree(p->*right*, q->*right*);

return rigt && left;

}

return false;

}

24.Given a binary tree, determine if it is height-balanced. For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the

two subtrees of every node never differ by more than 1.

题目翻译: 给定一颗二叉树， 写一个函数来检测这棵树是否是平衡二叉树. 对于这个问题, 一颗平衡树的定义是其中任意节点的左右子树的高度差不大于1.

bool CLeeteCodeTree::IsBalanced(TreeNode\* root)

{

if (checkDepth(root) == -1)

{

return false;

}

else

{

return true;

}

}

int CLeeteCodeTree::checkDepth(TreeNode\* root)

{

if (!root)

{

return 0;

}

int left = checkDepth(root->*left*);

if (left ==-1)

{

return -1;

}

int right = checkDepth(root->*right*);

if (right ==-1)

{

return -1;

}

int diff = *abs*(left - right);

if (diff>1)

{

return -1;

}

else

{

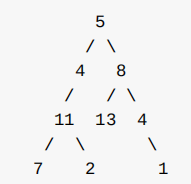
return 1 + *std*::*max*(left,right);

}

}

23.Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example: Given the below binary tree and sum = 22,



return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

题目翻译: 给定一颗二叉树和一个特定值，写一个方法来判定这棵树是否存在这样一种条件，使得从root到其中一个叶子节点的路径的和等于给定的sum值.

bool CLeeteCodeTree::hasPathSum(TreeNode\* root, int sum)

{

if (root == nullptr)

{

return false;

}

return DFS(sum, 0, root);

}

bool CLeeteCodeTree::DFS(int target, int sum, TreeNode\* root)

{

if (root == nullptr)

{

return false;

}

sum += root->val;

if (root->*left* == nullptr && root->*right* == nullptr)

{

if (sum == target)

{

return true;

}

else

{

return false;

}

bool leftPart = DFS(target,sum,root->*left*);

bool rightPart = DFS(target,sum,root->*right*);

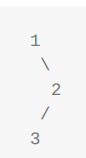
return leftPart || rightPart;

}

}

1. Given a binary tree, return the preorder traversal of its nodes' values.

For example: Given binary tree {1,#,2,3},



return [1,2,3].

Note: Recursive solution is trivial, could you do it iteratively?

给定一颗二叉树，使用迭代的方式进行前序遍历。因为不能递归，所以我们只能使用stack来保存迭代状态。对于前序遍历，根节点是最先访问的，然后是左子树，最后才是右子树。当访问到根节点的时候，我们需要将右子树压栈，这样访问左子树之后，才能正确地找到对应的右子树。

*std*::*vector*<int> CLeeteCodeTree::preorderTraversal(TreeNode\* Root)

{

*std*::*vector*<int> vals;

if (Root == nullptr)

{

return vals;

}

*std*::*vector*<TreeNode\*>nodes;

//首先将root压栈

nodes.*push\_back*(Root);

while (!nodes.*empty*())

{

TreeNode\* n = nodes.*back*();

vals.*push\_back*(n->val);

//访问了该节点，出栈

nodes.*pop\_back*();

//如果有右子树，压栈保存

if (n->*right*)

{

nodes.*push\_back*(n->*right*);

}

//如果有左子树，压栈保存

if (n->*left*)

{

nodes.*push\_back*(n->*left*);

}

}

return vals;

}

25.给定一颗二叉树，使用迭代的方式进行中序遍历。 对于中序遍历，首先遍历左子树，然后是根节点，最后才是右子树，所以我们需要用stack记录每次遍历的根节点，当左子树遍历完成之后，从stack弹出根节点，得到其右子树，开始新的遍历。

*std*::*vector*<int> CLeeteCodeTree::inorderTraversal(TreeNode\* root)

{

*std*::*vector*<int> vals;

if (root == nullptr)

{

return vals;

}

*std*::*vector*<TreeNode\*> nodes;

TreeNode\* p = root;

while (p||!nodes.*empty*())

{

//先一直遍历左子树，将根节点压栈

while (p)

{

nodes.*push\_back*(p);

p = p->*left*;

}

if (!nodes.*empty*())

{

p = nodes.*back*();

vals.*push\_back*(p->val);

//将根节点弹出，获取右子树

nodes.*pop\_back*();

p = p->*right*;

}

}

return vals;

}

26.给定一颗二叉树，使用迭代的方式进行后序遍历。

对于后序遍历，首先遍历左子树，然后是右子树，最后才是根节点。当遍历到一个节点的时候，首先我们 将右子树压栈，然后将左子树压栈。这里需要注意一下出栈的规则，对于叶子节点来说，直接可以出栈， 但是对于根节点来说，我们需要一个变量记录上一次出栈的节点，如果上一次出栈的节点是该根节点的左 子树或者右子树，那么该根节点可以出栈，否则这个根节点是新访问的节点，将右和左子树分别压栈。

*std*::*vector*<int> CLeeteCodeTree::postderTraversal(TreeNode\* root)

{

*std*::*vector*<int> vals;

if (root == nullptr)

{

return vals;

}

*std*::*vector*<TreeNode\*> nodes;

TreeNode\* pre = nullptr;

nodes.*push\_back*(root);

while (!nodes.*empty*())

{

TreeNode\* p = nodes.*back*();

//如果不判断pre，我们就无法正确出栈

if ((p->*left* == nullptr && p->*right* == nullptr)

||(pre !=nullptr &&(pre == p->*left*||pre == p->*right*)))

{

vals.*push\_back*(p->val);

nodes.*pop\_back*();

pre = p;

}

else

{

//右子树压栈

if (p->*left*!=nullptr)

{

nodes.*push\_back*(p->*left*);

}

}

}

return vals;

}

27.Given a binary tree

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL. Initially, all next pointers are set to NULL.

Note:

You may only use constant extra space. You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children). For example, Given the following perfect binary tree,

1

/ \

2 3

/ \ / \

4 5 6 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ / \

4->5->6->7 -> NULL

翻译：这题需要在一棵完全二叉树中使用next指针连接旁边的节点，我们可以发现一些规律。

如果一个子节点是根节点的左子树，那么它的next就是该根节点的右子树，譬如上面例子中的4，它的 next就是2的右子树5。

如果一个子节点是根节点的右子树，那么它的next就是该根节点next节点的左子树。譬如上面的5，它的next就是2的next（也就是3）的左子树。

void CLeeteCodeTree::connect(TreeLinkNode\* root)

{

if (!root)

{

return;

}

TreeLinkNode\* p = root;

TreeLinkNode\* first = nullptr;

while (p)

{

//记录下层第一个左子树

if (!first)

{

first = p->*left*;

}

//如果有左子树，那么next就是父节点

if (p->*left*)

{

p->*left*->*next* = p->*right*;

}

else

{

//叶子节点了，遍历结束

break;

}

//如果有next ，那么设置右子树的next

if (p->*next*)

{

p->*right*->*next* = p->*next*->*left*;

p = p->*next*;

continue;

}

else

{

//转到下一层

p = first;

first = nullptr;

}

}

}

28.What if the given tree could be any binary tree? Would your previous solution still work?

Note:

You may only use constant extra space. For example, Given the following binary tree,

1

/ \

2 3

/ \ \

4 5 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ \

4-> 5 -> 7 -> NULL

不同于上一题，这题的二叉树并不是完全二叉树，我们不光需要提供first指针用来表示一层的第一个元素，

同时也需要使用另一个lst指针表示该层上一次遍历的元素。那么我们只需要处理好如何设置last的next指针

就可以了。

27题只是28题的一个特例而已。

void CLeeteCodeTree::connect28(TreeLinkNode\* root)

{

if (!root)

{

return;

}

TreeLinkNode\* p = root;

TreeLinkNode\* first = nullptr;

TreeLinkNode\* last = nullptr;

while (p)

{

//设置下层第一个元素

if (!first)

{

if (p->*left*)

{

first = p->*right*;

}

else if (p->*right*)

{

first = p->*right*;

}

}

if (p->*left*)

{

//如果有last,则设置last的next

if (last)

{

last->next = p->*left*;

}

//last 为left

last = p->*left*;

}

if (p->*right*)

{

//如果有last，则设置last的next

if (last)

{

last->next = p->*right*;

}

//last 为right

last = p->*right*;

}

if (p->*next*)

{

p = p->*next*;

}

else

{

//转到下一层

p = first;

last = nullptr;

first = nullptr;

}

}

}

29.**Given a singly linked list** where elements are sorted in ascending order, convert it to a height balanced BST.

这题需要将一个排好序的链表转成一个平衡二叉树，我们知道，对于一个二叉树来说，左子树一定小于根

节点，而右子树大于根节点。所以我们需要找到链表的中间节点，这个就是根节点，链表的左半部分就是

左子树，而右半部分则是右子树，我们继续递归处理相应的左右部分，就能够构造出对应的二叉树了。

这题的难点在于如何找到链表的中间节点，我们可以通过fast，slow指针来解决，fast每次走两步，slow每次走一步，fast走到结尾，那么slow就是中间节点了。

？？晚上再弄

1. **Given an array** where elements are sorted in ascending order, convert it to a height balanced BST.

TreeNode\* CLeeteCodeTree::sortedArrayToBST(*std*::*vector*<int>& num)

{

return build30(num, 0, num.*size*());

}

TreeNode\* CLeeteCodeTree::build30(*std*::*vector*<int>& num, int start, int end)

{

if (start >= end)

{

return nullptr;

}

//对比一下两者之间的不同

//int mid = (start + end) / 2;

int mid = start + (end-start) / 2;

TreeNode\* node = new TreeNode(num[mid]);

node->*left* = build30(num, start, mid);

node->*right* = build30(num, mid + 1, end);

return node;

}

31.Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given

sum.

For example: Given the below binary tree and sum = 22.

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

return

[

[5,4,11,2],

[5,8,4,5]

]

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeTree::pathSum(TreeNode\* root, int sum)

{

*std*::*vector*<*std*::*vector*<int>> ret;

if (root == nullptr)

{

return ret;

}

*std*::*vector*<int> curr;

DFS31(ret,curr,sum,0,root);

return ret;

}

void CLeeteCodeTree::DFS31(*std*::*vector*<*std*::*vector*<int>>& ret, *std*::*vector*<int> curr, int sum, int temsum, TreeNode\* root)

{

if (root == nullptr)

{

return;

}

temsum += root->val;

curr.*push\_back*(root->val);

if (temsum == sum)

{

if (root->*left* == nullptr &&root->*right* ==nullptr)

{

ret.*push\_back*(curr);

return;

}

}

DFS31(ret, curr, sum, temsum, root->*right*);

DFS31(ret, curr, sum, temsum, root->*left*);

}

32 . Given a binary tree, flatten it to a linked list in-place.

For example, Given

1

/ \

2 5

/ \ \

3 4 6

The flattened tree should look like:

1

\

2

\

3

\

4

\

5

\

6

给定一颗二叉树，将其扁平化处理，我们可以看到处理之后的节点顺序其实跟前序遍历原二叉树的一致， 所以我们只需要前序遍历二叉树同时处理就可以了。

void CLeeteCodeTree::flatten(TreeNode\* root)

{

if (!root)

{

return;

}

*std*::*vector*<TreeNode\*> ns;

TreeNode dummy(0);

TreeNode\* n = &dummy;

ns.*push\_back*(root);

while (!ns.*empty*())

{

TreeNode\* p = ns.*back*();

ns.*pop\_back*();

//挂载到右子树

n->*right* = p;

n = p;

//右子树压栈

if (p->*right*)

{

ns.*push\_back*(p->*right*);

p->*right* = nullptr;

}

//左子树压栈

if (p ->*left*)

{

ns.*push\_back*(p->*left*);

p->*left* = nullptr;

}

}

}

33.Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

这题需要判断是不是一个正确的二叉搜索树，比较简单地一道题。

bool CLeeteCodeTree::isValidBST(TreeNode\* root)

{

return valid(root, *std*::*numeric\_limits*<int>::*min*(), *std*::*numeric\_limits*<int>::*max*());

}

bool CLeeteCodeTree::valid(TreeNode\* node, int minval, int maxVal)

{

if (!node)

{

return true;

}

if (node->val <= minval || node->val>= maxVal )

{

return false;

}

return valid(node->*left*, minval, node->val) &&

valid(node->*right*, node->val, maxVal);

}

34 . Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:

A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

这题需要修复一颗二叉搜索树的两个交换节点数据，我们知道对于一颗二叉搜索树来说，如果按照中序遍历，那么它输出的值是递增有序的，所以我们只需要按照中序遍历输出，在输出结果里面找到两个异常数 据（比它后面输出结果大），交换这两个节点的数据就可以了。

但是这题要求使用O(1)的空间，如果采用通常的中序遍历（递归或者栈）的方式，都需要O(N)的空间，所

以这里我们用Morris Traversal的方式来进行树的中序遍历。

Morris Traversal中序遍历的原理比较简单：

如果当前节点的左孩子为空，则输出当前节点并将其有孩子作为当前节点。

如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点，也就 是当前节点左子树的最右边的那个节点。

如果前驱节点的右孩子为空，则将它的右孩子设置为当前节点，当前节点更新为其左孩子。

如果前驱节点的右孩子为当前节点，则将前驱节点的右孩子设为空，输出当前节点，当前节点更 新为其右孩子。

重复上述过程，直到当前节点为空，递归的时候我们同时需要记录错误的节点。

那么我们如何知道一个节点的数据是不是有问题呢？对于中序遍历来说，假设当前节点为cur，它的前驱节点为pre，如果cur的值小

于pre的值，那么cur和pre里面的数据就是交换的了。

void CLeeteCodeTree::recoverTree(TreeNode\* root)

{

TreeNode\* cur = 0;

TreeNode\* pre = 0;

TreeNode\* p1 = 0;

TreeNode\* p2 = 0;

TreeNode\* preCur = 0;

bool found = false;

if (!root)

{

return;

}

cur = root;

while (cur)

{

if (!cur->left)

{

//记录p1和p2

if (preCur && preCur->val>cur->val)

{

if (!found)

{

p1 = preCur;

found = true;

}

p2 = cur;

}

preCur = cur;

cur = cur->right;

}

else

{

pre = cur->left;

while (pre->right&& pre->right! = cur)

{

pre = pre->right;

}

if (!pre->right)

{

pre->right = cur;

cur = cur->left;

}

else

{

//记录p1和p2

if (preCur->val > cur->val)

{

if (!found)

{

p1 = preCur;

found = true;

}

p2 = cur;

}

preCur = cur;

pre->right = nullptr;

cur = cur->right;

}

}

}

if (p1&&p2)

{

int t = p1->val;

p1->val = p2->val;

p2->val = t;

}

}

1. Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:

1

/ \

2 3

\

5

All root-to-leaf paths are:

["1->2->5", "1->3"]

题的翻译：

给定一棵二叉树，返回所有从根节点到叶节点的路径。

题目分析： 本题属于二叉树的遍历问题，可以用深度优先搜索来解决。

使用栈来记录遍历过程中访问过的节点。递归地访问每个节点的子节点，如果遇到叶节点，则输出记录的路径。返回上一层之前弹出栈顶元素。 C++的vector容器也能做到后进先出，所以下面的代码并没有使用

std::stack类来实现。

生成输出的字符串时，可以使用std::stringstream类来完成，类似于Java和C#中的StringBuilder。

*std*::*vector*<*std*::*string*> CLeeteCodeTree::binaryTreePaths(TreeNode\* root)

{

*std*::*vector*<*std*::*string*> result;

if (root == nullptr)

{

return result;

}

*std*::*vector*<int> path;

DFS35(root, path, result);

return result;

}

void CLeeteCodeTree::DFS35(TreeNode\* node, *std*::*vector*<int>& path, *std*::*vector*<*std*::*string*>& result)

{

if (node == nullptr)

{

return;

}

path.*push\_back*(node->val);

if (node->left == nullptr&&node->right == nullptr)

{

result.*push\_back*(generatePath(path));

}

else

{

if (node->left!= nullptr)

{

DFS35(node->left, path, result);

path.*pop\_back*();

}

if (node->right != nullptr)

{

DFS35(node->right, path, result);

path.*pop\_back*();

}

}

}

*std*::*string* CLeeteCodeTree::generatePath(*std*::*vector*<int> path)

{

*std*::*stringstream* ss;

int i;

for (i = 0;i<path.*size*()-1;i++)

{

ss << path[i] << "->";

ss << path[i];

return ss.*str*();

}

}

1. Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number. An example is the root-to-leaf path 1->2->3 which represents the number 123 . Find the total sum of all root-to-leaf numbers.

For example,

1

/ \

2 3

The root-to-leaf path 1->2 represents the number 12 . The root-to-leaf path 1->3 represents the number 13 . Return the sum = 12 + 13 = 25.

题目翻译： 给定一棵二叉树，仅包含0到9这些数字，每一条从根节点到叶节点的路径表示一个数。例如，

路径 1->2->3 表示数值123。求出所有路径表示的数值的和。 上述例子中，路径 1->2 表示数值12，路径 1->3 表示数值13。它们的和是25。

题目分析: 从根节点到叶节点的遍历方法是深度优先搜索(DFS)。解决本题只需在遍历过程中记录路径中的数字，在到达叶节点的时候把记录下来的数字转换成数值，加到和里面即可。

int CLeeteCodeTree::sumNumbers(TreeNode\* root)

{

*std*::*vector*<int> arr;

int sum = 0;

DFS36(root, arr, sum);

return sum;

}

int CLeeteCodeTree::vec2num(*std*::*vector*<int>& vec)

{

int num = 0;

for (auto n:vec)

{

num = num \* 10 + n;

}

return num;

}

void CLeeteCodeTree::DFS36(TreeNode\* node, *std*::*vector*<int>& arr, int& sum)

{

if (node == nullptr)

{

return;

}

arr.*push\_back*(node->val);

if (node->*left* == nullptr &&node->*right* == nullptr)

{

sum += vec2num(arr);

}

else

{

if (node->*left* != nullptr)

{

DFS36(node->*left*,arr,sum);

}

if (node->*right* != nullptr)

{

DFS36(node->*right*, arr, sum);

}

}

arr.*pop\_back*();

}

1. Say you have an array for which the ith element is the price of a given stock on day i. If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

这是卖股票的第一个题目，根据题意我们知道只能进行一次交易，但需要获得最大的利润，所以我们需要 在最低价买入，最高价卖出，当然买入一定要在卖出之前。 对于这一题，还是比较简单的，我们只需要遍历一次数组，通过一个变量记录当前最低价格，同时算出此次交易利润，并与当前最大值比较就可以了。

int CLeeteCodeDP::maxProfit(*std*::*vector*<int>& prices)

{

if (prices.*size*()<=1)

{

return 0;

}

int minP = prices[0];

int profit = prices[1] - prices[0];

for (int i = 2;i<prices.*size*();i++)

{

minP = *std*::*min*(prices[i - 1], minP);

profit = *std*::*max*(profit, prices[i] - minP);

}

if (profit<0)

{

return 0;

}

return profit;

}

1. Best Time to Buy and Sell Stock II

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

这题相对于上一题来说更加容易（这不知道为啥是II），因为不限制交易次数，我们在第i天买入，如果发现

i + 1天比i高，那么就可以累加到利润里面。

int CLeeteCodeDP::maxProfit38(*std*::*vector*<int>& prices)

{

int len = (int)prices.*size*();

if (len<=1)

{

return 0;

}

int sum = 0;

for (int i = 1;i<len;i++)

{

if (prices[i]-prices[i-1])

{

sum += prices[i] - prices[i - 1];

}

}

return sum;

}

1. Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

这题是三道题目中最难的一题，只允许两次股票交易，如果只允许一次，那么题目就退化到第一题了，根据第一题的算法，我们可以得到[0,1,...,i]区间的最大利润，同时在从后往前扫描得到[i,i+1,...,n-1]的最大利

润，两个相加就可以得到该题的解了。

int CLeeteCodeDP::maxProfit39(*std*::*vector*<int>& prices)

{

int len = (int)prices.*size*();

if (len <= 1)

{

return 0;

}

*std*::*vector*<int> profits;

profits.*resize*(len);

//首先我们正向遍历得到每天一次交易的最大收益

//并保存到profits里面

int minP = prices[0];

int sum = *std*::*numeric\_limits*<int>::*min*();

for (int i =1;i<len;i++)

{

minP = *std*::*min*(minP, prices[i - 1]);

profits[i] = *std*::*max*(sum, prices[i] - minP);

sum = profits[i];

}

int maxP = prices[len - 1];

int sum2 = *std*::*numeric\_limits*<int>::*min*();

//逆向遍历

for (int i = len-2;i>=0;i--)

{

maxP = *std*::*max*(maxP, prices[i + 1]);

sum2 = *std*::*max*(sum2, maxP - prices[i]);

if (sum2>0)

{

//这里我们直接将其加入profits里面，

//不需要额外保存

profits[i] = profits[i] + sum2;

sum =*std*::*max*(sum, profits[i]);

}

}

return sum > 0 ? sum : 0;

}

1. Unique Paths

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below). The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

这题是一道典型的dp问题，如果机器人要到(i, j)这个点，他可以选择先到(i - 1, j)或者，(i, j - 1)，也就是 说，到(i, j)的唯一路径数等于(i - 1, j)加上(i, j - 1)的个数，所以我们很容易得出dp方程:

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

dp[i][j]表示从点(0, 0)到(i, j)唯一路径数量。

动态规划的定义与用法，感觉很重要的样子哈哈哈

int CLeeteCodeDP::uniquePaths(const int m, const int n)

{

//初始化dp，m\*1情况全是1

int dp[m][n];

for (int i = 0; i<m; i++)

{

int dp[i][0] = 1;

}

//初始化dp，1 x n情况全为1

for (int j = 0;j<n;j++)

{

dp[0][j] = 1;

}

for (int i = 1;i<m;i++)

{

for (int j = 1;j<n;j++)

{

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

}

}

return dp[m - 1][n - 1];

}

1. Now consider if some obstacles are added to the grids. How many unique paths would there be? An obstacle and empty space is marked as 1 and 0 respectively in the grid

这题跟上一题唯一的区别在于多了障碍物，如果某一个点有障碍，那么机器人无法通过。

int CLeeteCodeDP::uniquePathsWithObstacles(*std*::*vector*<*std*::*vector*<int>>& obstacleGrid)

{

if (obstacleGrid.*empty*() || obstacleGrid[0].*empty*())

{

return 0;

}

int m = obstacleGrid.*size*();

int n = obstacleGrid[0].*size*();

int dp[m][n];

//下面初始dp的时候需要根据obstacleGrid的值来确定

dp[0][0] = (obstacleGrid[0][0] == 0 ? 1 : 0);

//我们需要注意m x 1以及1 x n的初始化

for (int i = 1;i<m;i++)

{

dp[i][0] = ((dp[i-1][0] == 1&&obstacleGrid[i][0] == 0)?1:0);

}

for (int j =1;j<n;j++)

{

dp[0][j] = ((dp[0][j-1] == 1 && obstacleGrid[0][j] == 0) ? 1 : 0);

}

for (int i = 1;i<m;i++)

{

for (int j =1;j<n;j++)

{

if (obstacleGrid[i][j] ==1)

{

dp[i][j] = 0;

}

else

{

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

}

}

}

return dp[m - 1][n - 1];

}

1. Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

这题跟前面两题差不多，所以放到这里说明了。我们使用dp[i][j]表明从(0, 0)到(i, j)最小的路径和，那么dp方 程为:

dp[i][j] = min(dp[i][j-1], dp[i - 1][j]) + grid[i][j]

int CLeeteCodeDP::minPathSum(*std*::*vector*<*std*::*vector*<int>>& grid)

{

if (grid.*empty*()||grid[0].*empty*())

{

return 0;

}

int row = grid.*size*();

int col = grid[0].*size*();

int dp[row][col];

dp[0][0] = grid[0][0];

for (int i = 1;i<row;i++)

{

dp[i][0] = dp[i - 1][0] + grid[i][0];

}

for (int j = 1;j<col;j++)

{

dp[0][j] = dp[0][j - 1] + grid[0][j];

}

for (int i = 1;i<row;i++)

{

for (int j =1;j<col;j++)

{

dp[i][j] = *std*::*min*(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];

}

}

return dp[row - 1][col - 1];

}

1. Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum. For example, given the array [−2,1,−3,4,−1,2,1,−5,4], the contiguous subarray [4,−1,2,1] has the largest sum = 6

**//我感觉和那个柱状图有点类似啊！！可以了解一下！**

这题是一道经典的dp问题，我们可以很容易的得到其dp方程，假设dp[i]是数组a [0, i]区间最大的值，那么 dp[i + 1] = max(dp[i], dp[i] + a[i + 1])

int CLeeteCodeDP::MaxSubArray(int A[], int n)

{

int sum = 0;

int m = *INT\_MIN*;

for (int i = 0;i<n;i++)

{

sum += A[i];

m = *std*::*max*(m,sum);

if (sum<0)

{

sum = 0;

}

}

return m;

}用二分法解决一下这个问题

假设数组A[left, right]存在最大区间，mid = (left + right) / 2，那么无非就是三中情况：

1. 最大值在A[left, mid - 1]里面

2. 最大值在A[mid + 1, right]里面

3. 最大值跨过了mid，也就是我们需要计算[left, mid - 1]区间的最大值，以及[mid + 1, right]的最大值，然后加上mid，三者之和就是总的最大值

我们可以看到，对于1和2，我们通过递归可以很方便的求解，然后在同第3的结果比较，就是得到的最大值

不是很明白这样做的目的，感觉不是很符合题意

int CLeeteCodeDP::maxSubArray(int A[], int n)

{

return divide(A, 0, n - 1,*INT\_MIN*);

}

int CLeeteCodeDP::divide(int A[], int left, int right, int tmax)

{

if (left>right)

{

return *INT\_MIN*;

}

int mid = left + (right - left) / 2;

//得到子区间[left, mid - 1]最大值

int lmax = divide(A, left, mid - 1, tmax);

//得到子区间[mid + 1, right]最大值

int rmax = divide(A, mid + 1, right, tmax);

tmax = *std*::*max*(tmax, lmax);

tmax = *std*::*max*(tmax,rmax);

int sum = 0;

int mlmax = 0;

//得到[left, mid - 1]最大值

for (int i = mid -1; i >= left; i--)

{

sum += A[i];

mlmax = *std*::*max*(mlmax, sum);

}

sum = 0;

int mrmax = 0;

//得到[mid + 1, right]最大值

for (int i = mid+1;i<=right;i++)

{

sum += A[i];

mrmax = *std*::*max*(mrmax, sum);

}

tmax = *std*::*max*(tmax, A[mid] + mlmax + mrmax);

return tmax;

}

1. Maxmimum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product. For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

这题是求数组中子区间的最大乘积，对于乘法，我们需要注意，负数乘以负数，会变成正数，**所以解这题 的时候我们需要维护两个变量，当前的最大值，以及最小值，最小值可能为负数，**但没准下一步乘以一个 负数，当前的最大值就变成最小值，而最小值则变成最大值了。

我们的动态方程可能这样：

maxDP[i + 1] = max(maxDP[i] \* A[i + 1], A[i + 1], minDP[i] \* A[i + 1])

minDP[i + 1] = min(minDP[i] \* A[i + 1], A[i + 1], maxDP[i] \* A[i + 1]

dp[i + 1] = max(dp[i], maxDP[i + 1])

这里，我们还需要注意元素为0的情况，如果A[i]为0，那么maxDP和minDP都为0，我们需要从A[i + 1]重新开始。

int CLeeteCodeDP::maxProduct(int A[], int n)

{

if (n==0)

{

return 0;

}

else if (n ==1)

{

return A[0];

}

int p = A[0];

int maxP = A[0];

int minP = A[0];

for (int i = 1;i<n;i++)

{

int t = maxP;

maxP = *std*::*max*(*std*::*max*(maxP \* A[i], A[i]), minP \* A[i]);

minP = *std*::*min*(*std*::*min*(t\*A[i],A[i]),minP\*A[i]);

p = *std*::*max*(maxP, p);

}

return p;

}

1. Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

爬楼，需要n步才能到达顶端，每次你可以爬1级或者2级，有多少种方法可以爬到顶端

这道题目其实就是斐波那契数列问题，题目比较简单，我们很容易就能列出dp方程

dp[n] = dp[n - 1] + dp[n - 2]

初始条件dp[1] = 1, dp[2] = 2。

int CLeeteCodeDP::climStairs(int n)

{

int f1 = 2;

int f2 = 1;

if (n == 1)

{

return f2;

}

else if (n==2)

{

return f1;

}

int fn;

for (int i = 3; i <= 3; i++)

{

fn = f1 + f2;

f2 = f1;

f1 = fn;

}

return fn;

}

1. Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to

adjacent numbers on the row below.

For example, given the following triangle

[

[2],

[3,4],

[6,5,7],

[4,1,8,3]

]

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

**这题要求我们求出一个三角形中从顶到底最小路径和，并且要求只能使用O(n)的空间。**

这题有两种解法，**自顶向下以及自底向上。**

首先来看自顶向下，根据题目我们知道，每向下一层，我们只能选择邻接数字进行累加，譬如上面第1行的数字3，它的下一行邻接数字就是6和5。

我们假设dp[m][n]保存了第m行第n个节点的最小路径和，我们有如下dp方程

dp[m + 1][n] = min(dp[m][n], dp[m][n - 1]) + triangle[m + 1][n] if n > 0

dp[m + 1][0] = dp[m][0] + triangle[m + 1][0]

**因为只能使用O(n)的空间，所以我们需要滚动计算，使用一个一位数组保存每层的最小路径和，参考 Pascal's Triangle，我们知道，为了防止计算的时候不覆盖以前的值，所以我们需要从后往前计算。**

//自上而下

int CLeeteCodeDP::minimumTotal(*std*::*vector*<*std*::*vector*<int>>& triangle)

{

int row = triangle.*size*();

if (row ==0)

{

return 0;

}

//初始化为最大值

*std*::*vector*<int>total(row, *INT\_MAX*);

total[0] = triangle[0][0];

int minTotal = *INT\_MAX*;

for (int i= 1;i<row;i++)

{

for (int j= i;j>=0;j--)

{

if (j==0)

{

total[j] = total[j] + triangle[i][j];

}

else

{

//上一层total[i]为INT\_MAX，不会影响最小值

total[j] = *std*::*min*(total[j - 1], total[j]) + triangle[i][j];

}

}

}

for (int i = 0;i<row;i++)

{

minTotal = *std*::*min*(minTotal, total[i]);

}

return minTotal;

}

//自下而上

区别于自顶向下，另一种更简单的做法就是自底向上了。dp方程为

dp[m][n] = min(dp[m + 1][n], dp[m + 1][n + 1]) + triangle[m][n]

我们仍然可以使用一位数组滚动计算。

int CLeeteCodeDP::minimumTotalToUp(*std*::*vector*<*std*::*vector*<int>>& triangle)

{

if (triangle.*empty*())

{

return 0;

}

int row = triangle.*size*();

*std*::*vector*<int>dp;

dp.*resize*(row);

//用最底层的数据初始化

for (int i = 0;i<dp.*size*();i++)

{

dp[i] = triangle[row - 1][i];

}

for (int i = row-2;i>=0;i--)

{

for (int j =0;j<=i;j++)

{

dp[j] = triangle[i][j]+*std*::*min*(dp[j],dp[j+1])

}

}

return dp[0];

}

1. Unique Binary Search Trees

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example, Given n = 3, there are a total of 5 unique BST's.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

这道题目要求给定一个数n，有多少种二叉树排列方式，用来存储1到n。

刚开始拿到这题的时候，完全不知道如何下手，但考虑到二叉树的性质，对于任意以i为根节点的二叉树，

它的左子树的值一定小于i，也就是[0, i - 1]区间，而右子树的值一定大于i，也就是[i + 1, n]。假设左子树有 m种排列方式，而右子树有n种，则对于i为根节点的二叉树总的排列方式就是m \*n。

我们使用dp[i]表示i个节点下面二叉树的排列个数，那么dp方程为:

dp[i] = sum(dp[k] \* dp[i - k -1]) 0

int CLeeteCodeDP::numsTree(int n)

{

*std*::*vector*<int> dp(n + 1, 0);

//初始化dp

dp[0] = 1;

dp[1] = 1;

for (int i = 2;i<=n;i++)

{

for (int j = 0; j < i; j++)

{

//如果左子树的个数为j，那么右子树为i - j - 1

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

}

}

return dp[n];

}

1. Unique Binary Search Trees II

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example, Given n = 3, your program should return all 5 unique BST's shown below

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

这题跟前面一题不同，需要得到所有排列的解。

根据前面我们知道，对于在n里面的任意i，它的排列数为左子树[0, i - 1]的排列数 x 右子树[i + 1, n]的排列数，所以我们只需要得到i的左子树和右子树的所有排列，就能得到i的所有排列了。而这个使用递归就能搞定。

*std*::*vector*<TreeNode\*> CLeeteCodeDP::GenerateTrees(int n)

{

return Generates(1,n);

}

*std*::*vector*<TreeNode\*> CLeeteCodeDP::Generates(int start, int stop)

{

*std*::*vector*<TreeNode\*> vs;

if (start>stop)

{

//没有子树了，返回null

vs.*push\_back*(*NULL*);

return vs;

}

for (int i =start;i<=stop;i++)

{

auto L = Generates(start, i - 1);

auto R = Generates(i + 1, stop);

//获取左子树和右子树所有排列之后，放到root为i的节点的下面

for (int j = 0;j<L.*size*();j++)

{

for (int k = 0;k<R.*size*();k++)

{

TreeNode\* n = new TreeNode(i);

n->*left* = L[j];

n->*right* = R[k];

vs.*push\_back*(n);

}

}

}

return vs;

}

1. Perfect Squares

Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n. For example, given n = 12 , return 3 because 12 = 4 + 4 + 4 ; given n = 13 , return 2 because 13 = 4 + 9 .

题目翻译： 给出一个正整数 n ，求至少需要多少个完全平方数（例如1，4，9，16……）相加能得到n。

例如，给出 n = 12 ，返回3，因为 12 = 4 + 4 + 4 。给出 n = 13 ，返回2，因为 13 = 4 + 9 。

题目分析： 乍一看题目，比较天真的想法是，先从不大于n的最大的完全平方数开始组合，如果和超过了 n，就换小一点的完全平方数。但问题是，最后如果凑不齐的话，只能添加很多1，总量上就不是最少的了。

例如12，题目中给的例子是4+4+4，需要3个完全平方数。如果从最大的开始组合，那么是9+1+1+1， 需要4个完全平方数。

从另一个角度来想，用穷举法来求解就是把不大于n的所有可能的完全平方数的组合都算出来，然后找出和 为n的组合中数量最少的那种组合。如果不大于n的完全平方数有m个的话，这个方法的时间复杂度是 O(m^m)。显然穷举法时间复杂度过大，不是可行的方法。观察到，在枚举的过程中，有一些组合显然不是最优的，比如把12拆成12个1相加。

# 另外，如果我们能够记录已经找到的最小组合，那么稍大一些的数只需 要在此基础上添加若干个完全平方数即可。这里面就包含了动态规划的思想。 具体来说，我们用一个数组来记录已有的结果，初始化为正无穷( INT\_MAX )。外层循环变量 i 从 0 到 n ，内层循环变量 j 在 i 的基础上依次加上每个整数的完全平方，超过 n 的不算。那么 i + j\*j 这个数需要 的最少的完全平方数的数量，就是数组中当前的数值，和 i 位置的数值加上一，这两者之间较小的数字。

**如果当前的值较小，说明我们已经找到过它需要的完全平方数的个数（最初都是正无穷）。否则的话，说明在 i 的基础上加上 j 的平方符合条件，所需的完全平方数的个数就是 i 需要的个数加上一。**

int CLeeteCodeDP::numSquares(int n)

{

*std*::*vector*<int> dp(n + 1, *INT\_MAX*);

dp[0] = 0;

for (int i = 0; i < n; i++)

{

for (int j =1;i+j\*j<=n;j++)

{

dp[i + j \* j] = *std*::*min*(dp[i + j \* j], dp[i] + 1);

}

}

return dp[n];

}

1. Combination

Given two integers n and k, return all possible combinations of k numbers out of 1,2,...,n.

**题目翻译:** 给定两个整型数组n和k，要求返由k个数组成的combination，其实应该叫做组合. 这个combination应该是高中里面的组合。这k个数是在n中任选k个数，由题意可得，这里的k应该小于或等于n(这个条件不要忘了做validation check哦).

**题目分析:** 我觉得应该还有不少读者困惑什么是combination，这里我们先给一个例子比如n=3， k=2的条件下， 所有可能的combination如下： [1,2], [1,3], [2,3]. 注意：[2,3]和[3,2]是相同的，我们只要求有其中一个就可以了. 所以解题的时候，我们要避免相同的组合出现.

**解题思路:** 看到这道题，首先第一想法应该是用递归来求解.如果要是用循环来求解，这个时间复杂度应该是比较恐怖了.并且，这个递归是一层一层往深处去走的，打个比方，我们一个循环，首先求得以1开始的看个数的combination，之后再求以2开始的，以此类推，所以开始是对n个数做DFS, n-1个数做DFS...所以应该是对n(n-1)...\*1做DFS. 在程序中，我们可以加一些剪枝条件来减少程序时间.

**时间复杂度:** 在题目分析中，我们提到了对于对n,n-1,...,1做DFS，所以时间复杂度是O(n!)

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeCombination::combine(int n, int k)

{

*std*::*vector*<*std*::*vector*<int>> Ret;

if (n<=0)

{

return Ret;

}

*std*::*vector*<int> Curr;

DFS50(Ret, Curr, n, k, 1);//we pass ret as reference at here

return Ret;

}

void CLeeteCodeCombination::DFS50(*std*::*vector*<*std*::*vector*<int>>& Ret, *std*::*vector*<int> curr, int n, int k, int level)

{

if (curr.*size*() == k)

{

Ret.*push\_back*(curr);

return;

}

if (curr.*size*()>k)

{

return;

}

for (int i = level; i <= n; ++i)

{

curr.*push\_back*(i);

DFS50(Ret, curr, n, k, i + 1);

curr.*pop\_back*();

}

}

1. Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

The same repeated number may be chosen from C unlimited number of times.

Note:

1. All numbers (including target) will be positive integers.

2. Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).

3. The solution set must not contain duplicate combinations.

题目翻译: 给一个数组C和一个目标值T, 找出所有的满足条件的组合：使得组合里面的数字之和等于T,并且

一些数字可以从C中午先选择。

注意:

1. 所有给定的数字均为正整数.(这意味着我们加corner case invalid check的时候要加一条，如果给定T不是正整数，我们就没必要在往下进行了)

2. 所有的答案组中要满足升序排列.

3. 最后的答案数组不能包含重复答案.

std::vector<std::vector<int>> CLeeteCodeCombination::combinationSum(std::vector<int>& candidates, int target)

{

std::vector<std::vector<int>> ret;

//corner case invalid check

if (candidates.size() ==0||target<0)

{

return ret;

}

std::vector<int>curr;

std::sort(candidates.begin(),candidates.end());

BackTracking(ret, curr, candidates, target, 0);

return ret;

}

void CLeeteCodeCombination::BackTracking(std::vector<std::vector<int>>& ret, std::vector<int> curr, std::vector<int> candidates, int target, int level)

{

if (target ==0)

{

ret.push\_back(curr);

return;

}

else if (target<0)

{

return;

}

for (int i = level;i<candidates.size();i++)

{

target -= candidates[i];

curr.push\_back(candidates[i]);

BackTracking(ret, curr, candidates, target, i);

curr.pop\_back();

target += candidates[i];

}

}

1. Combination Sum II

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. Each number in C may only be used once in the combination.

Note:

1. All numbers (including target) will be positive integers.

2. Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).

3. The solution set must not contain duplicate combinations.

题目翻译: 给定一个数组C和一个特定值T,要求找出这里面满足以下条件的所有答案：数组中数字的值加起 来等于特定和的答案. 数组中每个数字只能用一次.（同three sum和four sum的解法）

注意条件:

1. 给定数组的所有值必须是正整数.（意味着我们加corner case invalid check的时候要检查T）

2. 答案数组中的值必须为升序排列.(我们要对数组进行排序)

3. 最终答案不能包含重复数组

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeCombination::combinationSum52(*std*::*vector*<int>& num, int target)

{

*std*::*vector*<*std*::*vector*<int>> ret;

if (num.*size*() ==0||target<0)

{

return ret;

}

*std*::*vector*<int> curr;

*std*::*sort*(num.*begin*(), num.*end*());

BackTracking52(ret, curr, num, target, 0);

return ret;

}

void CLeeteCodeCombination::BackTracking52(*std*::*vector*<*std*::*vector*<int>>& ret, *std*::*vector*<int> curr, *std*::*vector*<int> num, int target, int level)

{

if (target ==0)

{

ret.*push\_back*(curr);

return;

}

else if (target<0)

{

return;

}

for (int i = level;i<num.*size*();++i)

{

target -= num[i];

curr.*push\_back*(num[i]);

BackTracking52(ret,curr,num,target,i+1);

curr.*pop\_back*();

target += num[i];

while (i<num.*size*()-1 && num[i] == num[i+1])

{

++i;

}

}

}

1. Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent. A mapping of digit to letters (just like on the telephone buttons) is given as below:



Input:Digit string "23"

Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

**题目翻译:** 给定一个字符串数字，返回这个字符串数字在电话表上可能的combination，一个map的电话键盘在上图已经给出.

**题目分析:** 这道题目的给出，具体的解题思路是和combination是相同的，不同的地方是我们要先建一个 dictionary,以方便查找.之后用combination的相同方法，对于每一个数字，在dictionary中查找它所对应的说有的数字.

**解题思路:** 我是用字符串数组来构建这个dictionary的，用于下标代表数字，例如，下标为2，我的字典就会有这种对应的关系:dic[2] = "abc".只要把给定数字字符串的每一个数字转换为int类型，就可以根据字典查找出这个数字所对应的所有字母.当然，再构建字典的时候，我们需要注意dic[0] = "", dic[1] = "".这两个特殊case，因为电话键盘并没有这两个数字相对应的字符串.

*std*::*vector*<*std*::*string*> CLeeteCodeCombination::letterCombinations(*std*::*string* digits)

{

*std*::*vector*<*std*::*string* > ret;

*std*::*vector*<*std*::*string* > dic;

*std*::*string* tmp;

dic.*push\_back*("");

dic.*push\_back*("");

dic.*push\_back*("abc");

dic.*push\_back*("def");

dic.*push\_back*("ghi");

dic.*push\_back*("jkl");

dic.*push\_back*("mno");

dic.*push\_back*("pqrs");

dic.*push\_back*("tuv");

dic.*push\_back*("wxyz");

combinationSum53(ret,tmp,digits,dic,0);

return ret;

}

void CLeeteCodeCombination::combinationSum53(*std*::*vector*<*std*::*string*>& ret, *std*::*string* tmp, *std*::*string* digits, *std*::*vector*<*std*::*string*> dic, int level)

{

if (level ==digits.*size*())

{

ret.*push\_back*(tmp);

return;

}

int index = digits[level] - '0';

for (int i =0;i<dic[index].*size*();++i)

{

tmp.*push\_back*(dic[index][i]);

combinationSum53(ret, tmp, digits, dic, level + 1);

tmp.*pop\_back*();

}

}

1. Subsets

Given a set of distinct integers, S, return all possible subsets.

Note: Elements in a subset must be in non-descending order. The solution set must not contain duplicate subsets. For example, If S = [1,2,3], a solution is:

[

[3],

[1],

[2],

[1,2,3],

[1,3],

[2,3],

[1,2],

[]

]

这题其实就是列出集合里面的所有子集合，同时要求子集合元素需要升序排列。首先我们需要对集合排序，对于一个n元素的集合，首先我们取第一个元素，加入子集合中，后面的n - 1个元素可以认为是第一个元素的子节点，我们依次遍历，譬如遍历到第二个元素的时候，后续的n - 2个元素又是第二个元素的子节点，再依次遍历处理，直到最后一个元素，然后回溯，继续处理。处理完第一个元素之后，我们按照同样的方式处理第二个元素。

譬如上面的[1, 2, 3]，首先取出1，加入子集合，后面的2和3就是1的子节点，先取出2，把[1, 2]加入子集合，后面的3就是2的子节点，取出3，把[1, 2, 3]加入子集合。然后回溯，取出3，将[1, 3]加入子集合。1处理完成之后，我们可以同样方式处理2，以及3。

*std*::*vector*<*std*::*vector*<int>> ressss;

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeCombination::subsets(*std*::*vector*<int>& S)

{

if (S.*empty*())

{

return ressss;

}

*std*::*sort*(S.*begin*(),S.*end*());

//别忘了空集

ressss.*push\_back*(*std*::*vector*<int>());

*std*::*vector*<int> v;

generate54(0,v,S);

return ressss;

}

void CLeeteCodeCombination::generate54(int start, *std*::*vector*<int>& v, *std*::*vector*<int>& S)

{

if (start ==S.*size*())

{

return;

}

for (int i =start;i<S.*size*();i++)

{

v.*push\_back*(S[i]);

ressss.*push\_back*(v);

generate54(i+1,v,S);

v.*pop\_back*();

}

}

1. Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations. For example, [1,1,2] have the following unique permutations: [1,1,2], [1,2,1], and [2,1,1].

题目翻译: 给定一个整形数组，这个数组中可能会包含重复的数字，要求我们返回的是这个数组不同的Permutations,也就是说每一种可能的permutation在最后的答案中只能出现一次.上文的例子能清晰的告诉读者不同的地方。

题目分析: 对于这道题。也是要求permutation,大体上的解题思路和Permutations是相同的，但是不同点在

哪里呢？ 不同点为：

1. 这个给定的数组中可能会含有相同的数字.

2. 最后答案不接受重复相同的答案组.

对于这两点要求，Permutations的解法是无法解决的，所以我们就要考虑怎样满足以上两个要求. 我们可以

对整个input数组进行排序，在求解答案的时候，只要一个元素的permutation求出来了，在这个元素后面和这个元素相同的元素，我们完全都可以pass掉，其实这个方法在sum和combination里面已经是屡试不爽了.

**解题思路: 除了加上对于重复答案的处理外，剩下思路同Permutation一模一样。**

**时间复杂度: O(n!)**

*std*::*vector*<*std*::*vector*<int>> CLeeteCodeCombination::premuteUnique(*std*::*vector*<int>& num)

{

*std*::*vector*<*std*::*vector*<int>>permutations;

if (num.*size*()==0)

{

return permutations;

}

*std*::*vector*<int>curr;

*std*::*vector*<bool>isVisited(num.*size*(), false);

*std*::*sort*(num.*begin*(), num.*end*());

DFS55(permutations,curr,num,isVisited);

return permutations;

}

void CLeeteCodeCombination::DFS55(*std*::*vector*<*std*::*vector*<int>>& ret, *std*::*vector*<int>curr, *std*::*vector*<int>num, *std*::*vector* < bool>isVisited)

{

if (curr.*size*() == num.*size*())

{

ret.*push\_back*(curr);

return;

}

for (int i = 0;i<num.*size*();i++)

{

if (isVisited[i] == false)

{

isVisited[i] = true;

curr.*push\_back*(num[i]);

DFS55(ret, curr, num, isVisited);

isVisited[i] = false;

curr.*pop\_back*();

while (i<num.*size*()-1&&num[i] == num[i+1])

{

++i;

}

}

}

}

1. Given a string s and a dictionary of words dict, determine if s can be segmented into a space- separated sequence of one or more dictionary words. For example, given s = "leetcode", dict = ["leet", "code"]. Return true because "leetcode" can be segmented as "leet code".

这题的意思是给出一本词典以及一个字符串，能否切分这个字符串使得每个字串都在字典里面存在。

假设dp(i)表示长度为i的字串能否别切分，dp方程如下:

dp(i) = dp(j) && s[j, i) in dict, 0

bool CLeeteCodeDP::wordBreak(*string* s, *unordered\_set*<*string*>& dict)

{

int len = (int)s.*size*();

*vector*<bool>dp(len + 1, false);

dp[0] = true;

for (int i = 1;i<=len;i++)

{

for (int j = i-1;j>=0;j--)

{

if (dp[j] &&dict.*find*(s.*substr*(j,i-j))!= dict.*end*())

{

dp[i] = true;

break;

}

}

}

return dp[len];

}

1. World Break II

Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.

For example, given s = "catsanddog", dict = ["cat", "cats", "and", "sand", "dog"].

A solution is ["cats and dog", "cat sand dog"].

这道题不同于上一题，需要我们得到所有能切分的解。这道题难度很大，我们需要采用dp + dfs的方式求

解，首先根据dp得到该字符串能否被切分，同时在dp的过程中记录属于字典的子串信息，供后续dfs使用。

首先我们使用dp[i][j]表示起始索引为i，长度为j的子串能否被切分，它有三种状态:

1. dp[i][j] = true && dp[i][j] in dict，这种情况是这个子串直接在字典中

2. dp[i][j] = true && dp[i][j] not in dict，这种情况是这个子串不在字典中，但是它能切分成更小的子串，而

这些子串在字典中

1. dp[i][j] = false，子串不能被切分

根据题意，我们需要求出所有切分的解，所以在进行dp的时候需要处理1和2这两种情况，因为对于2来说，

dp[i][j]是要继续被切分的，也就是说我们只需要关注第1种情况的子串。

当dp完成之后，我们就需要使用dfs来得到整个的解。在dp[i][j] = 1的情况下面，我们只需要dfs递归处理后

面i + j开始的子串就可以了。

*vector*<*vector*<char>> dp;

*vector*<*string*> vals;

*string* val;

*vector* <*string*> CLeeteCodeDP::wordBreakII(*string* s, *unordered\_set*<*string*>& dict)

{

int len = (int)s.*size*();

dp.*resize*(len);

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

{

dp[i].*resize*(len + 1, 0);

}

for (int i = 1;i<=len;i++)

{

for (int j =0;j<len-i+1;j++)

{

//直接存在于字典中，是第1种情况

if (dict.*find*(s.*substr*(j,i))!=dict.*end*())

{

dp[j][i] = 1;

continue;

}

//如果不存在，则看子串是不是能被切分，这是第2中情况

for (int k =1;k<i&&k<len-j;k++)

{

if (dp[j][k] &&dp[j+k][i-k])

{

dp[j][i] = 2;

break;

}

}

}

}

//不能切分，不用dfs了

if (dp[0][len] == 0)

{

return vals;

}

DFS57(s, 0);

return vals;

}

void CLeeteCodeDP::DFS57(const *string*& s, int start)

{

int len = (int)s.*size*;

if (start == len)

{

vals.*push\_back*(val);

return;

}

for (int i = 1;i<=len-start;i++)

{

if (dp[start][i] == 1)

{

int oldlen = (int)val.*size*();

if (oldlen!=0)

{

val.*append*("");

}

val.*append*(s.*substr*(start, i));

//我们从start + i开始继续dfs

DFS57(s, start + i);

val.*erase*(oldlen, *string*::*npos*);

}

}

}

1. Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up: Can you solve it without using extra space?

这道题就是判断一个链表是否存在环，非常简单的一道题目，我们使用两个指针，一个每次走两步，一个每次走一步，如果一段时间之后这两个指针能重合，那么铁定存在环了。

bool CLeeteCodeLinkedList::hasCycle(ListNode\* head)

{

if (head == nullptr||head->next == nullptr)

{

return false;

}

ListNode\* fast = head;

ListNode\* slow = head;

while (fast->next!=nullptr&&fast->next->next!=nullptr)

{

fast = fast->next->next;

slow = slow->next;

if (slow == fast)

{

return true;

}

}

return false;

}

1. Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Follow up: Can you solve it without using extra space?

紧跟着第一题，这题不光要求出是否有环，而且还需要得到这个环开始的节点。譬如下面这个，起点就是

n2。

n6-----------n5

| |

n1--- n2---n3--- n4|

我们仍然可以使用两个指针fast和slow，fast走两步，slow走一步，判断是否有环，当有环重合之后，譬如 上面在n5重合了，那么如何得到n2呢？

首先我们知道，fast每次比slow多走一步，所以重合的时候，fast移动的距离是slot的两倍，我们假设n1到 n2距离为a，n2到n5距离为b，n5到n2距离为c，fast走动距离为 a + b + c + b ，而slow为 a + b ，有方

程 a + b + c + b = 2 x (a + b) ，可以知道 a = c ，所以我们只需要在重合之后，一个指针从n1，而 另一个指针从n5，都每次走一步，那么就可以在n2重合了。

ListNode\* CLeeteCodeLinkedList::detectCycle(ListNode\* head)

{

if (head == nullptr||head->*next* == nullptr)

{

return nullptr;

}

ListNode\* fast = head;

ListNode\* slow = head;

while (fast->next!= nullptr && fast->next->next!=nullptr)

{

fast = fast->next->next;

slow = slow->next;

if (fast == slow)

{

slow = head;

while (slow!=fast)

{

fast = fast->next;

slow = slow->next;

}

return slow;

}

}

return nullptr;

}

1. Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

A: a1 → a2

↘

c1 → c2 → c3

↗

B: b1 → b2 → b3

begin to intersect at node c1.

Notes:

1 If the two linked lists have no intersection at all, return null

2 The linked lists must retain their original structure after the function returns.

3 You may assume there are no cycles anywhere in the entire linked structure.

4 Your code should preferably run in O(n) time and use only O(1) memory.

这题需要得到两个链表的交接点，也就是c1，这一题也很简单。

遍历A，到结尾之后，将A最后的节点连接到B的开头，也就是 c3 -> b1

使用两个指针fast和slow，从a1开始，判断是否有环

如果没环，在返回之前记得将 c3 -> b1 给断开

如果有环，则按照第二题的解法得到c1，然后断开 c3 -> b1

ListNode\* CLeeteCodeLinkedList::GetIntersectionNode(ListNode\* headA, ListNode\* headB)

{

if (!headA)

{

return nullptr;

}

else if (!headB)

{

return nullptr;

}

ListNode\* p = headA;

while (p->next)

{

p = p->next;

}

//将两个链表串起来

p->next = headB;

ListNode\* tail = p;

p = headA;

//fast和slow，判断是否有环

headB = headA;

while (headB->next&&headB->next->next)

{

headA = headA->next;

headB = headB->next->next;

if (headA == headB)

{

break;

}

}

if (!headA->next||!headB->next||!headB->next->next)

{

//没环，断开两个链表

tail->next = nullptr;

return nullptr;

}

//有环，得到环的起点

headA = p;

while (headA != headB)

{

headA = headA->next;

headB = headB->next;

}

//断开两个链表

tail->next = nullptr;

return headA;

}

1. Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,

Given 1->1->2, return 1->2.

Given 1->1->2->3->3, return 1->2->3

**这题要求在一个有序的链表里面删除重复的元素，只保留一个，也是比较简单的一个题目，我们只需要判**

**断当前指针以及下一个指针时候重复，如果是，则删除下一个指针就可以了。**

**ListNode\* CLeeteCodeLinkedList::deletDuplicates(ListNode\* head)**

**{**

**if (!head)**

**{**

**return head;**

**}**

**//用一个dummy节点来当做head的prev**

**ListNode dummy(0);**

**dummy.next = head;**

**ListNode\* prev = &dummy;**

**ListNode\* p = head;**

**while (p&&p->next)**

**{**

**if (p->value != p->next->value)**

**{**

**prev = p;**

**p = p->next;**

**}**

**else**

**{**

**//如果有重复，则继续遍历，直到不重复的节点**

**int values = p->value;**

**ListNode\* n = p->next->next;**

**while (n)**

**{**

**if (n->value!= values)**

**{**

**break;**

**}**

**n = n->next;**

**}**

**//删除重复节点**

**prev->next = n;**

**p = n;**

**}**

**}**

**return dummy.next;**

**}**

1. Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

这题要求合并两个已经排好序的链表，很简单的题目，直接上代码：

ListNode\* CLeeteCodeLinkedList::mergeTwoLists(ListNode\* L1, ListNode\* L2)

{

ListNode dummy(0);

ListNode\* p = &dummy;

while (L1&&L2)

{

int val1 = L1->value;

int val2 = L2->value;

//哪个节点小，就挂载，同时移动到下一个节点

if (val1<val2)

{

p->*next* = L1;

p = L1;

L1 = L1->next;

}

else

{

p->*next* = L1;

p = L1;

L1 = L1->next;

}

}

//这里处理还未挂载的节点

if (L1)

{

p->*next* = L1;

}

else if (L2)

{

p->*next* = L2;

}

return dummy.next;

}

1. Merge k Sorted Lists

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

这题需要合并k个排好序的链表，我们采用 divide and conquer 的方法，首先两两合并，然后再将先前合并的继续两两合并。时间复杂度为O(NlgN)。

ListNode\* CLeeteCodeLinkedList::mergeKLists(*vector*<ListNode\*>& lists)

{

if (lists.*empty*())

{

return nullptr;

}

int n = lists.*size*();

while (n>1)

{

int k = (n + 1) / 2;

for (int i =0;i<n/2;i++ )

{

//合并i和i + k的链表，并放到i位置

lists[i] = merge2List63(lists[i], lists[i + k]);

}

//下个循环只需要处理前k个链表了

n = k;

}

return lists[0];

}

ListNode\* merge2List63(ListNode\* n1, ListNode\* n2)

{

ListNode dummy(0);

ListNode\* p = &dummy;

while (n1&&n2)

{

if (n1->value < n2->value)

{

p->*next* = n1;

n1 = n1->next;

}

else

{

p->*next* = n2;

n2 = n2->next;

}

p = p->*next*;

}

if (n1)

{

p->*next* = n1;

}

else if (n2)

{

p->*next* = n2;

}

return dummy.*next*;

}

1. Reverse Linked List II

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example: Given 1->2->3->4->5->NULL, m = 2 and n = 4,

return 1->4->3->2->5->NULL.

Note: Given m, n satisfy the following condition: 1 ≤ m ≤ n ≤ length of list.

这题要求我们翻转[m, n]区间之间的链表。对于链表翻转来说，几乎都是通用的做法，譬如 p1 -> p2 -> p3 -> p4 ，如果我们要翻转p2和p3，其实就是将p3挂载到p1的后面，所以我们需要知道p2的前驱节点p1。伪代码如下：

//保存p3

n = p2->next;

//将p3的next挂载到p2后面

p2->next = p3->next;

//将p3挂载到p1的后面

p1->next = p3;

//将p2挂载到p3得后面

p3->next = p2;

对于上题，我们首先遍历得到第m - 1个node，也就是pm的前驱节点。然后依次遍历，处理挂载问题就可以了。

ListNode\* CLeeteCodeLinkedList::reverseBetween(ListNode\* head, int m, int n)

{

if (!head)

{

return head;

}

ListNode dummy(0);

dummy.*next* = head;

ListNode\* p = &dummy;

for (int i = 1;i<m;i++)

{

p = p->*next*;

}

//p此时就是pm的前驱节点

ListNode\* pm = p->*next*;

for (int i = m;i<n;i++)

{

ListNode\* n = pm->next;

pm->next = n->*next*;

n->*next* = p->*next*;

p->*next* = n;

}

return dummy.*next*;

}

1. Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list. If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is. You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example, Given this linked list: 1->2->3->4->5

For k = 2, you should return: 2->1->4->3->5

For k = 3, you should return: 3->2->1->4->5

这题要求我们按照每k个节点对其进行翻转，理解了链表如何翻转之后很容易处理，唯一需要注意的就是每次k个翻转之后，一定要知道最后一个节点，因为这个节点就是下组的前驱节点了。

ListNode\* CLeeteCodeLinkedList::reverseKGroup(ListNode\* head, int k)

{

if (k<=1||!head)

{

return head;

}

ListNode dummy(0);

dummy.*next* = head;

ListNode\* p = &dummy;

ListNode\* prev = &dummy;

while (p)

{

prev = p;

for (int i = 0;i<k;i++)

{

p = p->*next*;

if (!p)

{

//到这里已经不够k个没法翻转了

return dummy.*next*;

}

}

p = reverse(prev, p->*next*);

}

return dummy.*next*;

}

ListNode\* CLeeteCodeLinkedList::reverse(ListNode\* prev, ListNode\* end)

{

ListNode\* p = prev->*next*;

while (p->*next*!=end)

{

ListNode\* n = p->*next*;

p->*next* = n->*next*;

n->*next* = prev->*next*;

prev->*next* = n;

}

//这里我们会返回最后一个节点，作为下一组的前驱节点

return p;

}

1. Sort List

Sort a linked list in O(n log n) time using constant space complexity.

这题要求我们对链表进行排序，**我们可以使用divide and conquer的方式（分而治之算法）**，依次递归的对链表左右两半进行 排序就可以了。代码如下：

ListNode\* CLeeteCodeLinkedList::sortList(ListNode\* head)

{

if (head ==nullptr ||head->*next* ==nullptr)

{

return head;

}

ListNode\* fast = head;

ListNode\* slow = head;

//快慢指针得到中间点

while (fast->*next*&&fast->*next*->*next*)

{

fast = fast->*next*->*next*;

slow = slow->*next*;

}

//将链表拆成两半

fast = slow->*next*;

slow->*next* = nullptr;

//左右两半分别排序

ListNode\* p1 = sortList(head);

ListNode\* p2 = sortList(fast);

//合并

return merge66(p1, p2);

}

ListNode\* CLeeteCodeLinkedList::merge66(ListNode\* L1, ListNode\* L2)

{

if (!L1)

{

return L2;

}

else if (!L2)

{

return L1;

}

else if (!L1&&!L2)

{

return nullptr;

}

ListNode dummy(0);

ListNode\* p = &dummy;

while (L1&&L2)

{

if (L1->value<L2->value)

{

p->*next* = L1;

L1 = L1->*next*;

}

else

{

p->*next* = L2;

L2 = L2->*next*;

}

p = p->*next*;

}

if (L1)

{

p->*next* = L1;

}

else if (L2)

{

p->*next* = L2;

}

return dummy.*next*;

}

1. Insertion Sort List

Sort a linked list using insertion sort.

这题要求我们使用插入排序的方式对链表进行排序，假设一个链表前n个节点是有序，第n + 1的节点需要遍历前n个，插入到合适位置就可以了。

ListNode\* CLeeteCodeLinkedList::insertionSortList(ListNode\* head)

{

if (head == nullptr||head->*next* == nullptr)

{

return head;

}

ListNode dummy(0);

ListNode\* p = &dummy;

ListNode\* cur = head;

while (cur)

{

p = &dummy;

while (p->*next* &&p->*next*->value<=cur->value)

{

p = p->*next*;

}

ListNode\* n = p->*next*;

p->*next* = cur;

cur = cur->next;

p->*next*->*next* = n;

}

return dummy.*next*;

}

1. Rotate List

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:

Given 1->2->3->4->5->NULL and k = 2,

return 4->5->1->2->3->NULL

这题要求把链表后面k个节点轮转到链表前面。

对于这题我们首先需要遍历链表，得到链表长度n，因为k可能大于n，所以我们需要取余处理，然后将链表串起来形成一个环，在遍历 n - k % n 个节点，断开，就成了。譬如上面这个例子，k等于2，我们遍历到链表结尾之后，连接1，然后遍历 5 - 2 % 5 个字节，断开环，下一个节点就是新的链表头了。

ListNode\* CLeeteCodeLinkedList::rotateRight(ListNode\* head, int k)

{

if (!head||k ==0)

{

return head;

}

int n = 1;

ListNode\* p = head;

//得到链表长度

while (p->*next*)

{

p = p->*next*;

n++;

}

k = n - k % n;

//连接成环

p->*next* = head;

for (int i = 0;i<k;i++)

{

p = p->*next*;

}

//得到新的链表头并断开环

head = p->*next*;

p->*next* = nullptr;

return head;

}

1. Reorder List

Given a singly linked list L: L0→L1→…→Ln-1→Ln,

reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→…

You must do this in-place without altering the nodes' values.

For example,

Given {1,2,3,4}, reorder it to {1,4,2,3}.

这题比较简单，其实就是将链表的左右两边合并，只是合并的时候右半部分需要翻转一下。

主要有三步：

1. 快慢指针找到切分链表
2. 翻转右半部分
3. 依次合并

void CLeeteCodeLinkedList::reorderList(ListNode\* head)

{

if (head == nullptr||head->*next* ==nullptr)

{

return;

}

ListNode\* fast = head;

ListNode\* slow = head;

//快慢指针切分链表

while (fast->*next* != nullptr &&fast->*next*->*next*! = nullptr)

{

fast = fast->*next*->*next*;

slow = slow->*next*;

}

fast = slow->*next*;

slow->*next* = nullptr;

//翻转右半部分

ListNode dummy(0);

while (fast)

{

ListNode\* n = dummy.*next*;

dummy.*next* = fast;

ListNode\* nn = fast->*next*;

fast->*next* = n;

fast = nn;

}

slow = head;

fast = dummy.*next*;

//依次合并

while (slow)

{

if (fast!= nullptr)

{

ListNode\* n = slow->*next*;

slow->*next* = fast;

ListNode\* nn = fast->*next*;

fast->*next* = n;

fast = nn;

slow = n;

}

else

{

break;

}

}

}

1. Partition List

Given a linked list and a value x, partition it such that all nodes less than x come before nodes

greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,

Given 1->4->3->2->5->2 and x = 3,

return 1->2->2->4->3->5.

这题要求我们対链表进行切分，使得左半部分所有节点的值小于x，而右半部分大于等于x。我们可以使用两个链表，p1和p2，以此遍历原链表，如果节点的值小于x，就挂载到p1下面，反之则放到p2下面，最后将p2挂载到p1下面就成了。

ListNode\* CLeeteCodeLinkedList::partition70(ListNode\* head, int x)

{

ListNode dummy1(0), dummy2(0);

ListNode\* p1 = &dummy1;

ListNode\* p2 = &dummy2;

ListNode\* p = head;

while (p)

{

if (p->value<x)

{

p1->*next* = p;

p1 = p1->*next*;

}

else

{

p2->*next* = p;

p2 = p2->*next*;

}

p = p->*next*;

}

p2->*next* = nullptr;

p1->*next* = dummy2.next;

return dummy1.next;

}

1. Add Two Numbers

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 0 -> 8

两个链表相加的问题，需要处理好进位就成了，比较简单，直接上代码：

ListNode\* CLeeteCodeLinkedList::addTwoNumbers(ListNode\* L1, ListNode\* L2)

{

ListNode dummy(0);

ListNode\* p = &dummy;

int cn = 0;

while (L1||L2)

{

int val = cn + (L1 ? L1->value : 0) + (L2 ? L2->value : 0);

cn = val / 10;

val = val % 10;

p->*next* = new ListNode(value);

//p->next = new ListNode(val);

p = p->*next*;

if (L1)

{

L1 = L1->*next*;

}

if (L2)

{

L2 = L2->*next*;

}

}

if (cn!=0)

{

p->*next* = new ListNode(cn);

p = p->*next*;

}

return dummy.*next*;

}