一、排序算法

1.为什么要学习 O(n^2)的排序算法

基础，编码简单，易于实现，是一些简单情景的首选

在一些特殊情况下，简单的排序算法更有效

简单的排序算法思想衍生出复杂的排序算法

作为子过程，改进更复杂的排序算法

1.1 选择排序 Selection Sort O(n^2)

基本思想，将数据分为有序和无序两部分，先找到最小的元素，和没排序部分的第一个位置交换位置

然后从没排序的部分找到最小的，与第一个无序部分交换，则有序部分不断扩充，无序部分不断减少，

直到无序部分没有数据，结束。

代码：

```
#include <iostream>
#include <algorithm>
using namespace std;
template <typename T>
void selectionSort(T arr[] , int n)
{
    for(int i = 0; i < n; i++)
    {
        //寻找[i,n)区间里的最小值
        int minIndex = i;
        for(int j = i+1;j < n;j++)
        {
            if(arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }
        swap(arr[i],arr[minIndex]);
    }
}
int main()
{
    int a[10] = {10,9,8,7,6,5,4,3,2,1};
    selectionSort(a,10);
    for(int i = 0; i < 10; i++)
        cout << a[i] << endl;

    string c[4] = {"D","C","B","A"};
    selectionSort(c,4);
```

```
    for(int i = 0; i < 10; i++)
        cout << c[i] << endl;

    Student d[4] = { {"D",90}, {"C",30}, {"B",50}, {"A",80} };
    selectionSort(d,4);
    for(int i = 0; i < 10; i++)
        cout << d[i] << endl;

    return 0;
}
```

上面的程序只能对 int 数组排序，下面用模板来泛化

```
struct Student
{
    string name;
    int score;
    bool operator<(const Student &otherStudent)
    {
        return score < otherStudent.score;
    }
    friend ostream& operator<<(ostream &os,const Student &student)
    {
        os<<"Student: "<<student.name<<" "<<student.score<<endl;
        return os;
    }
};
```

随机生成算法测试用例

```
#include <ctime>
#include <iostream>
#include <cassert>
using namespace std;
namespace SortTestHelper
{
    //生成 n 个在 RangeL 和 RangeR 之间的元素的随机数组
    int* generateRandomArray(int n,int rangeL, rangeR)
    {
        assert(rangeL <= rangeR);
        int *arr = new int[n];
        srand(time(NULL));
        for(int i = 0;i < n;i++)
```

```
		{
			//生成随机数
			arr[i] = rand() % (rangeR - rangeL + 1) + rangeL;
		}
		return arr;
	}

	template<typename T>
	void printArray(T arr[], int n)
	{
		for(int i = 0;i < n;i++)
			cout << arr[i] << " ";
		cout << endl;
		return;
	}

	int* copyIntArray(int a[],int n)
	{
		int* arr = new int[n];
		copy(a,a+n,arr);
		return arr;
	}
}

int n = 10000;
int *arr = SortTestHelper::generateRandomArray(n,0,n);
selectionSort(arr,n);
SortTestHelper::printArray(arr,n);
delete[] arr;
```

测试算法的性能

```
namespace SortTestHelper
{
	//计算程序运行时间
	template<tyepname T>
	void testSort(string sortName,void(*sort)(T[],int),T arr[],int n)
	{
		clock_t startTime = clock();
		sort(arr,n);
		clock_t endTime = clock();
```

```
            assert(isSorted(arr,n));
            cout << sortName<<" :" << double(endTime - startTime) / CLOCKS_PRE_SEC << "s" << endl;
            return;
        }
        //测试排序是否成功
        template<typename T>
        bool isSorted(T arr[], int n)
        {
            for(int i = 0; i < n - 1; i++)
                if(arr[i] > arr[i + 1])
                    return false;
            return true;
        }
}

int main()
{
    int n = 10000;
    int *arr = SortTestHelper::generateRandomArray(n,0,n);
    SortTestHelper::testSort("Selection Sort",selectionSort,arr,n);
}
```

1.2 插入排序

算法思想：一个序列分无序和有序，初始的时候序列第一个是有序的，后面的都是无序的
从第二个开始，将其分别和有序的数据对比，交换，直到插入到了合适的位置。（扑克牌）

代码：

```
#include <iostream>
#include <algorithm>
#include "SortTestHelper.h"
#include "SelectionSort.h"

using namespace std;

template<typename T>
void insertionSort(T arr[], int n)
{
    for(int i = 1; i < n; i++)
    {
        //寻找第 i 个元素的插入位置
```

```
        for(int j = i; j > 0; j--)
        {
            if(arr[j] < arr[j-1])
            {
                swap(arr[j],arr[j-1]);
            }
            else
            {
                break;
            }
        }
    }
}
```

简洁版：

```
template<typename T>
void insertionSort(T arr[], int n)
{
    for(int i = 1; i < n; i++)
    {
        //寻找第 i 个元素的插入位置
        for(int j = i; j > 0 && arr[j] < arr[j-1]; j--)
        {
            swap(arr[j],arr[j-1]);
        }
    }
}
```

测试：

```
int main()
{
    int n = 10000；
    int *arr = SortTestHelper::generateRandomArray(n,0,n);
    int *arr2 = SortTestHelper::copy(arr,n);

    SortTestHelper:testSort("Insertion Sort",insertionSort,arr,n);
    a=SortTestHelper::testSort("Selection Sort",selectionSort,arr2,n);

    delete[] arr;
```

```
        delete[] arr2;
        return 0;
}
```

改进版插入排序

思想，先把无序的元素保存下来，用赋值替换 swap

```
template<typename T>
void insertionSort(T arr[], int n)
{
        T e = arr[i];
        for(int i = 1; i < n; i++)
        {
                //寻找第 i 个元素的插入位置
                for(int j = i; j > 0 && arr[j-1] > e; j--)
                {
                        arr[j]=arr[j-1];
                }
                arr[j] = e;
        }
}
```

小结：

O(n^2)的排序算法：

选择排序，插入排序，冒泡排序，希尔排序（O(n^1.3-2)）

对于近乎有序的数组，插入排序特别快，有可能比 nlogn 还快。

下面介绍 O(n*logn)的排序算法

1.3 归并排序

算法思想：将序列分成 2 部分，分别排序，然后合并。

合并的时候需要另外一个数组，和三个指针，分别指向新数组，左边有序数组和右边有序数组。

代码：

```
//合并,将 arr[l...mid]和 arr[mid+1...r]两部分进行归并
template<typename T>
void __merge(T arr[], int l,int mid, int r)
{
        T aux[r-l+1];
        for(int i = l;i <= r;i++)
        {
                aux[i-l] = arr[i];
        }
```

```
    int i = l,j=mid + 1;
    for(int k = l;k <= r;k++)
    {
        //先考虑一端已经遍历完的情况
        if(i > mid)
        {
            arr[k] = aux[j-l];
            j++;
        }
        else if(j > r)
        {
            arr[k] = aux[i-l];
            i++;
        }
        //再考虑两端都没有遍历完的情况
        else if(aux[i-l] < aux[j-l])
        {
            arr[k] = aux[i-l];
            i++;
        }
        else
        {
            arr[k] = aux[j-l];
            j++;
        }
    }
}

// 递归使用归并排序，对 arr[l...r]的范围进行排序
template<typename T>
void __mergeSort(T arr[],int l,int r)
{
    //递归停止条件（至少要有两个元素）
    if(l >= r)
        return;
    int mid = (l + r) / 2;
    __mergeSort(arr,l,mid);
    __mergeSort(arr,mid+1,r);
    __merge(arr,l,mid,r);
}
```

```
template<typename T>
void mergeSort(T arr[],int n)
{
	__mergeSort(arr,0,n-1);
}
```

归并排序需要利用额外的辅助空间来存储，利用的是分而治之，可以将 N 规模的问题变成 logN 规模的问题，一个问题是 N 复杂度，所以归并排序是 NlogN 复杂度。

归并排序优化：

```
template<typename T>
void __mergeSort(T arr[],int l,int r)
{
	//递归停止条件（至少要有两个元素）
	if(l >= r)
		return;
	int mid = (l + r) / 2;
	__mergeSort(arr,l,mid);
	__mergeSort(arr,mid+1,r);
	//如果左边最后一个比右边第一个要小，则不归并。
	if(arr[mid] > arr[mid + 1])
		__merge(arr,l,mid,r);
}
```

上面介绍的都是递归，下面用迭代来实现

```
template<typename T>
void mergeSortBU(T arr[],int n)
{
	for(int sz = 1;sz <= n;sz += sz)
	{
		for(int i = 0; i+sz<n;i += sz)
		{
			//对 arr[i...i+sz-1]和 arr[i+sz...i+sz+sz-1]进行归并
			__merge(arr,i,i+sz-1,min(i+sz+sz-1,n-1));
		}
	}
}
```

1.4 快速排序

算法思想：以某个元素为基点，处理一遍后左边的数都小于该元素，右边的数都大于该元素然后对左边和右边的分别进行上述过程。

```
//对 arr[l...r]部分进行 partition 操作
//返回 p，使得 arr[l...p-1] < arr[p];arr[p+1...r] > arr[p]
template<typename T>
int __partition(T arr[], int l, int r)
{
	swap(arr[rand()%(r-l+1)+l]); //随机化优化
	T v = arr[l];
	int j = l;
	for(int i = l + 1;i <= r; i++)
	{
		if(arr[i] < v)
		{
			swap(arr[j+1],arr[i]);
			j++;
		}
	}
	swap(arr[l],arr[j]);
	return j;
}

template<tyepname T>
void __quickSort(T arr[], int l, int r)
{
	if(l >= r)
	{
		return;
	}
	int p = __partition(arr,l,r);
	__quickSort(arr,l,p-1);
	__quickSort(arr,p+1,r);
}


template<tyepname>
void quickSort(T arr[], int n)
{
	srand(time(NULL)); //随机化优化
```

```
    __quickSort(arr, 0 ,n-1);
}
```

双路快速排序 partition

```
//对 arr[l...r]部分进行 partition 操作
//返回 p，使得 arr[l...p-1] < arr[p];arr[p+1...r] > arr[p]
template<typename T>
int __partition2(T arr[], int l, int r)
{
    swap( arr[l], arr[rand()%(r-l+1)+l] );
    T v = arr[l];

    //arr[l+1...i) <= v;arr(j...r] >= v
    int i=l+1 , j=r;
    while(true)
    {
        while(i <= r && arr[i] < v) i++;
        while(j<=l+1 && arr[j] > v) j--;
        if(i > j) break;
        swap(arr[i],arr[j]);
        i++;
        j--;
    }
    swap(arr[l],arr[j]);
    return j;
}
```

```
//三路快速排序
template<typename T>
void __quickSort3Ways(T arr[], int l, int r)
{
    if(r - l <= 15)
    {
        insertionSort(arr,l,r);
        return;
    }
    //partition
    swap(arr[l],arr[rand()%(r-l+1)+l]);
    T v = arr[l];
```

```
    int lt = l;//arr[l+1...lt] < v
    int gt = r + 1;//arr[gt...r] > v
    int i = l + 1;// arr[lt+1...i) == v
    while(i < gt)
    {
        if(arr[i] < v)
        {
            swap(arr[i],arr[lt+1]);
            lt++;
            i++;
        }
        else if(arr[i] > v)
        {
            swap(arr[i],arr[gt-1]);
            gt--;
        }
        else
        {
            i++;
        }
    }
    swap(arr[l],arr[lt]);
    __quickSort3Ways(arr,l,lt-1);
    __quickSort3Ways(arr,gt,r);

}
```

快速排序局限性：
对于近乎有序的数组，快速排序生成的递归树平衡度很差，分成的两部分一方很小，一方很大。
完全有序，退化成 O(n^2)

对于具有大量重复数据的序列，平衡度很差，分成两部分一方很少，一方很多（因为==的数据太多了）。

优化：
随机化快速排序（解决近乎有序的缺陷）
双路快速排序（解决具有大量重复数据）
三路快速排序（更加快速地解决大量重复数据）

小结：
归并排序和快速排序都利用了分治的思想

归并排序和快速排序的衍生问题

1.归并排序求逆序对的个数

2.快速排序求数组第 n 大元素（O(2n) n + n/2 + n/4 + n/8 + ... + 1）

2.堆

2.1 堆排序

堆是优先队列的实现数据结构，主要操作，入队（logN），出队(logN)（取出优先级醉倒的元素）

2.2 堆的基本实现（二叉堆 Binary heap）

条件：堆总是一颗完全二叉树（最后一层的节点必须集中在左侧）

堆中某个节点的值总是不大于其父节点的值（最大堆）

用数组来存储二叉堆（另根节点序号为 1，父节点的左右节点是 2n 和 2n+1）

则：parent(i) = i/2;

left child (i) = 2*i;

right child (i) = 2*i + 1;

代码：

```
#include <iostream>
#include <alogrithm>
#include <string>
#include <ctime>
#include <cmath>
#include <cassert>
using namespace std;
template<typename Item>
class MaxHeap
{
private:
    Item* data;
    int count;
    int capacity;
    //堆重构
    void shiftUp(int k)
    {
        while( k > 1 && data[k/2] < data[k])
        {
            swap(data[k/2],data[k]);
            k /= 2;
        }
    }
```

```
public:
    MaxHeap(int capacity)
    {
        //从索引 1 开始的，所以+1
        data = new Item[capacity + 1];
        count = 0;
        this->capacity = capacity;
    }
    int size()
    {
        return count;
    }
    bool isEmpty()
    {
        return count == 0;
    }
    ~MaxHeap()
    {
        delete[] data;
    }

    void insert(Item    item)
    {
        assert(count + 1 <= capacity);
        data[count+1] = item;
        count++;
        shiftUp(count);
    }
};
int mian()
{
    MaxHeap<int> maxheap = MaxHeap<int>(100);


    return 0;
}
```

排序算法总结：

	平均时间复杂度	原地排序(是否需要借助其他空间)	额外空间	是否稳定
插入排序	O(n^2)	是	O(1)	稳定
归并排序	O(nlogn)	否	O(n)	稳定

快速排序	O(nlogn)	是	O(logn)	不稳定
堆排序	O(nlogn)	是	O(1)	不稳定

稳定的排序：
不稳定的排序：

排序算法的稳定性 stable（可以通过自定义比较函数，让排序算法不存在稳定性问题）
稳定排序：对于相等的元素，在排序后，原来靠前的元素依然靠前。
相等元素的相对位置没有发生改变。

2.3 索引堆 index heap
2.4 和堆相关的问题
    动态选择优先级最高的任务执行
    从 1000000 个选出前 100 大的元素（Nlog(M)）
    d 叉堆
    ShiftUp 和 ShiftDown 中使用赋值操作替换 swap 操作
    表示堆的数组从 0 开始索引
    没有 capacity 的限制，动态的调整堆中数组的大小
5.1 二分查找法
    限制：对于有序数列，才能使用二分查找法
    代码：

```
template<typename T>
int binarySearch(T arr[],int n,T target)
{
    //在 arr[l...r]之中查找 target
    int l = 0, r = n - 1;
    while(l <= r)
    {
        //int mid = (l + r)/2;若 l 和 r 都是最大的 int，则会产生溢出 bug
        int mid = l + (r - l)/2;
        if(arr[mid] == target)
            return mid;
        if(target < arr[mid])
            r = mid - 1;
        else
            l = mid + 1;
    }
    return -1;
}
```

5.2 二分搜索树

优势：查找表的实现--字典数据结构

key1 value1

key2 value2

...

	查找元素	插入元素	删除元素
普通数组	O(n)	O(n)	O(n)
顺序数组	O(logn)	O(n)	O(n)
二分搜索树	O(logn)	O(n)	O(n)

高效：不仅可查找数据；还可以高效地插入，删除数据-动态维护数据

定义：二叉树（不一定是完全二叉树）

每个节点的键值大于左孩子

每个节点的键值小于又孩子

以左右孩子为根的子树仍为二分搜索树

代码：

```
template<tyepname Key,typename Value>
class BST
{
private:
    struct Node{
        Key key;
        Value value;
        Node* left;
        Node* right;

        Node(Key key,Value value){
            this->key = key;
            this->value = value;
            this->left = this->right = NULL;
        }
        Node(Node *node){
            this->key = node->key;
            this->value = node->value;
            this->left = node->left;
            this->right = node->right;
        }
    };
    Node *root;
```

```
    int count;
public:
    BST(){
        root = NULL;
        count = 0;
    }
    ~BST(){
        // TODO: ~BST()
        destroy(root);
    }
    int size(){
        return count;
    }
    bool isEmpty()
    {
        return count == 0;
    }
    void insert(Key key,Value value)
    {
        root = insert(root,key,value);
    }
    bool contain(Key key)
    {
        return contain(root,key);
    }
    Value* search(Key key){
        return search(root,key);
    }
    //前序遍历
    void preOrder(){
        preOrder(root);
    }
    //中序遍历
    void inOrder(){
        inOrder(root);
    }
    //后序遍历
    void postOrder(){
        postOrder(root);
    }
```

```
//层次遍历
void levelOrder(){
	queue<Node*> q;
	q.push(root);
	while(!q.empty()){
		Node *node = q.front();
		q.pop();
		cout << node->key<<endl;
		if(node->left)
			q.push(node->left);
		if(node->right)
			q.push(node->right);
	}
}
//寻找最小值
Key minimun(){
	assert(count != 0);
	Node* minNode = minimun(root);
	return minimun->key;
}
//寻找最大值
Key maxumun(){
	assert(count != 0);
	Node* maxNode = maxumun(root);
	return maxumun->key;
}
//删除最小元素
void removeMin(){
	if(root)
		root = removeMin(root);
}
//删除最大节点
void removeMax(){
	if(root)
		root = removeMax(root);
}
//删除节点
void remove(Key key){
	root = remove(root,key);
}
```

```
private:
	Node* insert(Node *node,Key key,Value value){
		if(node == NULL)
		{
			count++;
			return new Node(key,value);
		}
		if(key == node->key)
			node->value = value;
		else if(key < node->key)
			node->left = insert(node->left,key,value);
		else
			node->right = insert(node->right,key,value);
		return node;
	}
	bool contain(Node* node,Key key){
		if(node == NULL)
			return false;
		if(key == node->key)
			return true;
		else if(key < node->key)
			return contain(node->left,key);
		else
			return contain(node->right,key);
	}
	Value* search(Node* node,Key key){
		if(node == NULL)
			return NULL;
		if(key == node->value)
			return &(node->value);
		else if(key < node->key)
			return search(node->left,key);
		else
			return search(node->right,key);
	}
	void preOrder(Node* node){
		if(node != NULL){
			cout << node->key<<endl;
			preOrder(node->left);
			preOrder(node->right);
```

```
	}
}
void inOrder(Node* node){
	if(node != NULL){
		inOrder(node->left);
		cout << node->key<<endl;
		inOrder(node->right);
	}
}
void postOrder(Node* node){
	if(node != NULL){
		cout << node->key<<endl;
		postOrder(node->left);
		postOrder(node->right);
	}
}
void destory(Node* node){
	if(node != null){
		destory(node->left);
		destory(node->right);
		delete node;
		count--;
	}
}
Node* minimun(Node* node){
	if(node->left == NULL)
		return node;
	return node->left;
}
Node* maximun(Node* node){
	if(node->right == NULL)
		return node;
	return node->right;
}
Node* removeMin(Node* node){
	if(node->left == NULL){
		Node* rightNode = node->right;
		delete node;
		count--;
		return rightNode;
```

```
        }
        node->left = removeMin(node->left);
        return node;
}
Node* removeMax(Node* node){
        if(node->right == NULL){
                Node* leftNode = node->left;
                delete node;
                count--;
                return leftNode;
        }
        node->right = removeMax(node->right);
        return node;
}
//返回删除节点后新的二分搜索树的根
Node* remove(Node* node,Key key){
        if(node == NULL)
                return NULL;
        if(key < node->key){
                node->left = remove(node->left,key);
                return node;
        }
        else if(key > node->key){
                node->right = remove(node->right,key);
                return node;
        }
        else{
                if(node->left == NULL){
                        Node *rightNode = node->right;
                        delete node;
                        count--;
                        return rightNode;
                }
                if(node->right == NULL){
                        Node *leftNode = node->left;
                        delete node;
                        count--;
                        return leftNode;
                }
                Node *delNode = node;
```

```
            Node *successor = new Node(minimun(node->right));
            count++;
            successor->right = removeMin(node->right);
            successor->left = node->left;

            delete delNode;
            count--;
            return successor;
        }
    }
};
```

5.5 二分查找树的遍历(深度优先)

前序遍历：中左右

中序遍历：左中右（会是排好序的）

后序遍历：左右中

前中后遍历命名是根据中间节点的访问顺序，先访问的就是前

5.6 二分查找树的层次遍历（广度优先）

要引入队列的概念：先把根节点入队，然后出队，输出，然后将其左右子节点入队，然后出队一个数，之后循环

5.7 二分搜索树 删除最小值和最大值（见代码）

5.8 二分搜索树 删除节点（见代码）

删除左右都有孩子的节点 d

找到 s = min(d->right);

s 是 d 的后继节点

s->right = delMin(d->right)

s->left = d->left

删除 d,s 是新的子数的根

5.9 二叉搜索树的局限性

按照顺序将元素插入树中，会退化成链表。

同一份数据，生成的树不相同。

优化：平衡二叉树搜索树（红黑树，2-3Tree，AVL Tree，Splay Tree）

有两个子树，并且左右子树的高度差不大于 1

6.1 并查集基础

可以非常高效的解决称之为连接问题的问题

如：网络中节点之间的连接状态

--网络是个抽象的概念：用户之间形成的网络

--数学中的连接状态

连接问题和路径问题

6.2 Union Find

对于一组数据，主要支持两个动作

union(p,q):合并，连接

find(p):查

用来回答问题

isConnected(p,q)

并查集的基本数据表示

0 1 2 3 4 5 6 7 8 9

0 0 0 0 0 1 1 1 1 1

表示 0-4 是互相连接的，5-9 是互相连接的

代码：(最简单的版本)

```
class UnionFind{
private:
	int *id;
	int count;
public:
	UnionFind(int n){
		count = n;
		id = new int[n];
		for(int i = 0; i < n; i ++)
			id[i] = i;
	}
	~UnionFind(){
		delete []id;
	}
	int find(int p){
		assert(p >= 0 && p < count)
		return id[p];
	}
	bool isConnected(int p ,int q){
		return find(p) == find(q);
	}
	void unionElements(int p, int q){
		int pId = find(p);
		int qId = find(q);
		if(pId == qId)
			return;
		for(int i = 0;i < count; i++)
		{
			if(id[i] = pId)
				id[i] = qId;
```

```
        }
    }
};
```

代码：（常规版本，每个子元素都指向一个父元素）

```
class UnionFind{
private:
    int *parent;
    int count;
public:
    UnionFind(int count){
        this->count = count;
        parent = new int[count];
        for(int i = 0; i < count; i ++)
            parent[i] = i;
    }
    ~UnionFind(){
        delete []parent;
    }
    int find(int p){
        assert(p>=0 && p<count);
        while(p != parent[p])
            p = parent[p];
        return p;
    }
    bool isConnected(int p,int q){
        return find(p) == find(q);
    }
    void unionElements(int p,int q){
        int pRoot = find(p);
        int qRoot = find(q);

        if(pRoot == qRoot)
            return;
        parent[pRoot] = qRoot;
    }
};
```

6.4 基于 size 的优化

```
class UnionFind{
private:
```

```
	int *parent;
	int* sz;	//sz[i]表示以 i 为根的集合的元素个数
	int count;
public:
	UnionFind(int count){
		this->count = count;
		parent = new int[count];
		sz = new int[count];
		for(int i = 0; i < count; i ++){
			parent[i] = i;
			sz[i] = 1;
		}
	}
	~UnionFind(){
		delete []parent;
		delete []sz;
	}
	int find(int p){
		assert(p>=0 && p<count);
		while(p != parent[p])
			p = parent[p];
		return p;
	}
	bool isConnected(int p,int q){
		return find(p) == find(q);
	}
	void unionElements(int p,int q){
		int pRoot = find(p);
		int qRoot = find(q);

		if(pRoot == qRoot)
			return;

		if(sz[pRoot] < sz[qRoot]){
			parent[pRoot] = qRoot;
			sz[qRoot] += sz[pRoot];
		}
		else{
			parent[qRoot] = pRoot;
			sz[pRoot] += sz[qRoot];
```

```
        }
    }
};
```

6.5 基于 rank 的优化（根据集合的层数，rank[i]表示根节点为 i 的树的高度）

```
class UnionFind{
    private:
        int *parent;
        int* rank; //rank[i]表示以 i 为根的集合的树的层数
        int count;
    public:
        UnionFind(int count){
            this->count = count;
            parent = new int[count];
            rank = new int[count];
            for(int i = 0; i < count; i ++){
                parent[i] = i;
                rank[i] = 1;
            }
        }
        ~UnionFind(){
            delete []parent;
            delete []rank;
        }
        int find(int p){
            assert(p>=0 && p<count);
            while(p != parent[p])
                parent[p] = parent[parent[p]];//路径压缩
                p = parent[p];
            return p;
        }
        bool isConnected(int p,int q){
            return find(p) == find(q);
        }
        void unionElements(int p,int q){
            int pRoot = find(p);
            int qRoot = find(q);

            if(pRoot == qRoot)
                return;
```

```
        if(rank[pRoot] < rank[qRoot]){
            parent[pRoot] = qRoot;
        }
        else if(rank[pRoot] > rank[qRoot]){
            parent[qRoot] = pRoot;
        }
        else{
            rank[pRoot] = qRoot;
            rank[qRoot] += 1;
        }
    }
};
```

7.1 图论 Graph Theory

节点（Vertex）

边（Edge）

图的分类：无向图、有向图

无权图、有权图

简单图：没有自环边（自己连自己）和平行边（两个节点不止一条边）的图。

7.2 图的表示

邻接矩阵（Adjacency Matrix）

```
    0   1   2   3            0----1
0   0   1   0   0                / |
1   1   0   1   1               /  |
2   0   1   0   1            3----2
3   0   1   1   0
```

邻接表

```
0   1
1   0 2 3
2   1 3
3   1 2
```

邻接表适合表示稀疏图

邻接矩阵适合表示稠密图

代码：

```
//稠密图---邻接矩阵
class DenseGraph{
private:
    int n,m;
    bool directed;//有向图或者无向图
```

```
	vector<vector<bool>> g;
public:
	DenseGraph(int n,bool directed){
		this->n = n;
		this->m = 0;
		this->directed = directed;
		for(int i = 0;i < n; i++)
			g.push_back(vector<bool>(n,false));
	}
	~DenseGraph(){
	}
	int V(){return n;}
	int E(){return m;}
	void addEdge(int v,int w){
		assert(w>=0 && w<n);
		assert(v>=0 && v<n);
		if(hasEdge(v,w))
			return;
		g[v][w] = true;
		if(!directed)
			g[w][v] = true;
		m++;
	}
	bool hasEdge(int v,int w){
		assert(w>=0 && w<n);
		assert(v>=0 && v<n);
		return g[v][w];
	}
};
```

//稀疏图 -- 邻接表

缺点：不添加平行边，会造成 add 操作编程 O(n)复杂度

```
class SpareGraph{
private:
	int n , m;
	bool directed;
	vector<vector<int>> g;

public:
	SpareGraph(int n, bool directed){
```

```
        this->n = n;
        this->m = 0;
        this->directed = directed;
        for(int i = 0; i < n; i++)
            g.push_back(vector<int>());
    }
    ~SpareGraph(){}
    int V(){return n;}
    int E(){return m;}
    void addEdge(int v,int w){
        assert(v >= 0 && v < n);
        assert(w >= 0 && w < n);
        //会包含平行边
        g[v].push_back(w);
        if(!directed && v!=w)
            g[w].push_back(v);
        m++;
    }
    bool hasEdge(int v,int w){
        assert(v >= 0 && v < n);
        assert(w >= 0 && w < n);
        for(int i = 0;i < g[v].size();i++)
            if(g[v][i] == w)
                return true;
        return false;
    }
    class adjIterator{
    private:
        SpareGraph &G;
        int v;
        int index;
    public:
        adjIterator(SpareGraph &graph, int v):G(graph){
            this->v = v;
            this->index = 0;
        }
        int begin(){
        index = 0;
            if(G.g[v].size())
                return G.g[v][index];
```

```
            return -1;
        }
        ine next(){
            index++;
            if(index < G.g[v].size())
                return G.g[v][index];
        }
        bool end(){
            return index >= G.g[v].size();
        }
    };
};
```

7/3 相邻节点迭代器

遍历临边

7.4 图的算法框架

输入：第一行表示图有多少个节点和边，第二行开始是两个节点相连边

```
13  13
0   5
4   3
0   1
9   12
6   4
5   4
0   2
11  12
9   10
0   6
7   8
9   11
5   3
```

代码：

```
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <cassert>
using namespace std;
template<typename Graph>
class ReadGraph{
public:
```

```
    ReadGraph(Graph &graph, const string &filename){
        ifstream file(filename);
        string line;
        int V,E;

        assert(file.is_open());
        assert(getline(file,line));
        stringstream ss(line);
        ss >> V >> E;

        assert(V == graph.V());
        for(int i=0;i<E;i++){
            assert(getline(file,line));
            stringstream ss(line);
            int a,b;
            ss>>a>>b;
            assert(a>=0 && a<V);
            assert(b>=0 && b<V);
            graph.addEdge(a,b);
        }
    }
};

int main(){
    string filename = "test.txt";
    SpareGraph g1(13,false);
    ReadGraph<SpareGraph> readGraph1(g1,filename);
}
```

7.5 深度优先遍历和联通分量

某节点出发，输出第一个相连节点，在遍历这个节点的相连节点，知道都遍历完，返回。

代码：

```
template <typename Graph>
class Component{
private:
    Graph &G;
    bool *visited;
    int ccount;     //联通分量
    int *id;    //节点属于哪个联通分量
    void dfs(int v){
```

```
            visited[v] = true;
            id[v] = ccount;
            tyepname Graph::adjIterator adj(G,v);
            for(int i = adj.begin();!adj.end();i=adj.next()){
                if(!visited[i])
                    dfs(i);
            }
        }
    public:
        Component(Graph &graph):G(graph){
            visited = new bool[G.V()];
            id = new int[G.V()];
            ccount = 0;
            for(int i = 0;i < G.V(); i++){
                visited[i] = false;
                id[i] = -1;
            }

            for(int i = 0; i <G.V() ; i++)
                if(!visited[i]){
                    dfs(i);
                    ccount++;
                }
        }
        ~Component(){
            delete[] visited;
            delete[] id;
        }
        int count(){
            return ccount;
        }
        bool isConnected(int v,int w){
            assert(v >= 0 && v < G.V());
            assert(2 >= 0 && 2 < G.V());
            return id[v] == id[w];
        }

    };
```

7.6 寻路

获得两点间的一条路径(深度优先遍历找路径)

稀疏图（邻接表）：O(V+E)

稠密度（邻接矩阵）：O(V^2)

```
class Path{
private:
	Graph &G;
	int s;
	bool* visited;
	int* from;

	void dfs(int v){
		visited[v] = true;

		tyepname Graph::adjIterator adj(G,v);
		for(int i = adj.begin();!adj.end();i=adj.next()){
			if(!visited[i]){
				from[i] = v;
				dfs(i);
			}
		}
	}
public:
	Path(Graph &graph,int s):G(graph){
		//算法初始化
		assert(s>=0 && s<G.V());
		visited = new bool[G.V()];
		from = new int[G.V()];
		for(int i = 0;i < G.V(); i++){
			visited[i] = false;
			from[i] = -1;
		}
		this->s = s;

		//寻路算法
		dfs(s);
	}
	~Path(){
		delete[] visited;
		delete[] from;
	}
```

```
    bool hasPath(int w){
        assert(w>=0 && w<G.V());
        return visited[w];
    }
    void path(int w,vector<int> &vec){
        stack<int> s;
        int p = w;
        while(p!=-1){
            s.push(p);
            p = from[p];
        }

        vec.clear();
        while(!s.empty()){
            vec.push_back(s.top());
            s.pop();
        }
    }
    void showPath(int w){
        vector<int> vec;
        path(w,vec);
        for(int i = 0;i < vec.size();i++){
            cout << vec[i];
            if(i == vec.size() - 1)
                cout << endl;
            else
                cout << "->";
        }
    }
};
```

7.7 广度优先遍历和最短路径（利用队列，和树的广度优先一致）

广度优先遍历求出了无权图的最短路径，复杂度和深度优先是一样的

稀疏图（邻接表）：O(V+E)

稠密度（邻接矩阵）：O(V^2)

```
template <typename Graph>
class ShortestPath{
private:
    Graph &G;
    int s;
    bool *visited;
```

```
    int *from;
    int *ord;
public:
    ShortestPath(Graph &graph, int s):G(graph){
        //算法初始化
        assert(s>=0 && s<graph.V());

        visited = new bool[graph.V()];
        from = new int[graph.V()];
        ord = new int[graph.V()];
        for(int i = 0; i < graph.V();i++){
            visited[i] = false;
            from[i] = -1;
            ord[i] = -1;
        }
        this->s = s;
        queue<int> q; //辅助数据结构
        //无向图最短路径算法
        q.push(s);
        visited[s] = true;
        ord[s] = 0;
        while(!q.empty()){
            int v = q.front();
            p.pop();
            typename Graph::adjIterator adj(G,v);
            for(int i = adj.begin();!adj.end();i=adj.next())
                if(!visited[i]){
                    q.push(i);
                    visited[i] = true;
                    from[i] = v;
                    ord[i] = ord[v] + 1;
                }
        }
    }
    bool hasPath(int w){
        assert(w>=0 && w<G.V());
        return visited[w];
    }
    void path(int w,vector<int> &vec){
        stack<int> s;
```

```
            int p = w;
            while(p!=-1){
                s.push(p);
                p = from[p];
            }

            vec.clear();
            while(!s.empty()){
                vec.push_back(s.top());
                s.pop();
            }
        }
        void showPath(int w){
            vector<int> vec;
            path(w,vec);
            for(int i = 0;i < vec.size();i++){
                cout << vec[i];
                if(i == vec.size() - 1)
                    cout << endl;
                else
                    cout << "->";
            }
        }
        int length(int w){
            assert(w>=0 && w<G.V());
            return ord[w];
        }
    };
```

7.8 迷宫生成，扫雷，PS 抠图-----更到无权图的应用

8.1 有权图 Weighted Graph

邻接表,需要两个，一个是节点编号，一个是权。封装成 Edge 类

0 {to:1,w:0.12}

1 {to:0,w:0.12},{to:2,w:0.34},{to:3,w:0.52}

2 {to:1,w:0.34},{to:3,w:0.28}

3 {to:1,w:0.52},{to:2,w:0.28}

代码：边

```
template<tyepname Weight>
class Edge{
```

```
private:
    int a,b;
    Weight weight;
public:
    Edge(int a,int b,Weight weight){
        this->a = a;
        this->b = b;
        this->weight = weight;
    }
    Edge(){}
    ~Edge(){}
    int v(){return a;}
    int w(){return b;}
    Weight wt(){return weight;}
    int other(int x){
        assert(x==a || x==b);
        return x==a?b:a;
    }
    friend ostream& operator<<(ostream &os,const Edge &e){
        os<<e.a<<"-"e.b<<": "<<e.weight;
        return os;
    }
    bool operator<(Edge<Weight>& e){
        return weight < e.wt();
    }
    bool operator<=(Edge<Weight>& e){
        return weight <= e.wt();
    }
    bool operator>(Edge<Weight>& e){
        return weight > e.wt();
    }
    bool operator>=(Edge<Weight>& e){
        return weight >=( e.wt();
    }
    bool operator==(Edge<Weight>& e){
        return weight == e.wt();
    }
};
```

邻接矩阵

```
template<tyepname Weight>
class DenseGraph{
private:
    int n,m;
    bool directed;//有向图或者无向图
    vector<vector<Edge<Weight> *>> g;
public:
    DenseGraph(int n,bool directed){
        this->n = n;
        this->m = 0;
        this->directed = directed;
        for(int i = 0;i < n; i++)
            g.push_back(vector<Edge<Weight> *>(n,NULL));
    }
    ~DenseGraph(){
        for(int i = 0;i < n;i++)
            for(int j = 0;j < n; j++)
                if(g[i][j] != NULL)
                    delete g[i][j];
    }
    int V(){return n;}
    int E(){return m;}
    void addEdge(int v,int w,Weight weight){
        assert(w>=0 && w<n);
        assert(v>=0 && v<n);
        if(hasEdge(v,w)){
            delete g[v][w];
            if(!directed)
                delete g[w][v];
            m--;
        }
        g[v][w] = new Wdge<Weight>(v,w,weight);
        if(!directed)
            g[w][v] = new Wdge<Weight>(v,w,weight);
        m++;
    }
    bool hasEdge(int v,int w){
        assert(w>=0 && w<n);
        assert(v>=0 && v<n);
        return g[v][w] != NULL;
```

```
    }
};
```

//稀疏图 -- 邻接表

缺点：不添加平行边，会造成 add 操作编程 O(n)复杂度

```
template<tyepname Weight>
class SpareGraph{
private:
    int n , m;
    bool directed;
    vector<vector<Edge<Weight> *>> g;

public:
    SpareGraph(int n, bool directed){
        this->n = n;
        this->m = 0;
        this->directed = directed;
        for(int i = 0; i < n; i++)
            g.push_back(vector<Edge<Weight> *>());
    }
    ~SpareGraph(){}
    int V(){return n;}
    int E(){return m;}
    void addEdge(int v,int w,Weight weight){
        assert(v >= 0 && v < n);
        assert(w >= 0 && w < n);
        //会包含平行边
        g[v].push_back(new Edge(v,w,weight));
        if(!directed && v!=w)
            g[w].push_back(new Edge(w,v,weight);
        m++;
    }
    bool hasEdge(int v,int w){
        assert(v >= 0 && v < n);
        assert(w >= 0 && w < n);
        for(int i = 0;i < g[v].size();i++)
            if(g[v][i]->other(v) == w)
                return true;
        return false;
    }
```

```
    class adjIterator{
    private:
        SpareGraph &G;
        int v;
        int index;
    public:
        adjIterator(SpareGraph &graph, int v):G(graph){
            this->v = v;
            this->index = 0;
        }
        Edge<Weight> * begin(){
            index = 0;
            if(G.g[v].size())
                return G.g[v][index];
            return NULL;
        }
        Edge<Weight> * next(){
            index++;
            if(index < G.g[v].size())
                return G.g[v][index];
            return NULL;
        }
        bool end(){
            return index >= G.g[v].size();
        }
    };
};
```

8.2 最小生成树问题和切分原理 Minimum Span Tree

生成树不但连接了所有节点，同事所有边权重相加最小

实际生活中的应用：电缆布线设计，网络设计，电路设计

针对带权无向图，针对连通图

问题：找 V-1 条边

连接 V 个顶点

总权值最小

切分：把图中的节点分成两部分，称为一个切分

横切边：如果一个遍的两个端点，属于切分不同的两边，这个边称为横切边

切分定理：给定任意切分，横切边中权值最小的边必然属于最小生成树

8.3 Prim 算法的第一个实现（Lazy Prim）

（1）初始化 U={v},以 v 到其他顶点的所有边为候选边；

（2）重复以下步骤（n-1）次，使得其他（n-1）个顶点被加入到 U 中：

1.从侯选边中挑选权值最小的边加入 TE，设该边在 V-U 中的顶点是 k，将 k 加入 U 中；

2.考察当前 V-U 中所有顶点 j，修改侯选边，若边（k，j）的权值小于原来和顶点 j 关联的侯选边，则用边（k，j）取代后者作为侯选边

```
template<typename Graph, typename Weight>
class LasyPrimMST{
private:
    Graph &G;
    MinHeap<Edge<Weight>> pq;
    bool *marked;
    vector<Edge<Weight>> mst;
    Weight mstWeight;

    void visit(int v){
        assert(!marked[v]);
        marked[v] = true;

        typename Graph::adjIterator adj(G,v);
        for(Edge<Weight>* e = adj.begin();!adj.end();e=adj.next())
            if(!marked[e->other(v)])
                pq.insert(*e);
    }
public:
    LasyPrimMST(Graph &graph):G(graph),pq(MinHeap<Edge<Weight>>(graph.E())){
        marked = new bool[G.V()];
        for(int i = 0;i < G.V();i++)
            marked[i] = false;
        mst.clear();

        //Lazy Prim O(ElogE)
        visit(0);
        while(!pq.empty()){
            Edge<Weight> e = pq.extractMin();
            if(marked[e.v()] == marked[e.w()])
                continue;
            mst.push_back(e);
            if(!marked[e.v()])
                visit(e.v());
```

```
                else
                    visit(e.w());
            }

            mstWeight = mst[0].wt();
            for(int i = 1;i < mst.size();i++)
                mstWeight += mst[i].wt();
        }
        ~LasyPrimMST(){
            delete[] marked;
        }
        vector<Edge<Weight>> mstEdges(){
            return mst;
        }
        Weight result(){
            return mstWeight;
        }
    };
```

8.4 Prim 优化 O(ElogV)

```
    template<typename Graph, typename Weight>
    class PrimMST{
    private:
        Graph &G;
        IndexMinHeap<Weight> ipq;
        vector<Edge<Weight>*> edgeTo;
        bool *marked;
        vector<Edge<Weight>> mst;
        Weight mstWeight;

        void visit(int v){
            assert(!marked[v]);
            marked[v] = true;

            typename Graph::adjIterator adj(G,v);
            for(Edge<Weight>* e = adj.begin();!adj.end();e=adj.next())
                int w = e->other(v);
                if(!marked[w]){
                    if(!edgeTo[w]){
                        ipq.insert(w,e->wt());
```

```
                    edgeTo[w] = e;
                }
                else if(e->wt() < edgeTo[w]->wt()){
                    edgeTo[w] = e;
                    ipq.change(w,e->wt());
                }
            }
        }
    public:
        PrimMST(Graph &graph):G(graph),ipq(IndexMinHeap<double>(graph.V())){
            marked = new bool[G.V()];
            for(int i = 0;i < G.V();i++){
                marked[i] = false;
                edgeTo.push_back(NULL);
            }
            mst.clear();
            //Prim
            visit(0);
            while(!ipq.empty()){
                int v = ipq.extractMinIndex();
                assert(edgeTo[v]);
                mst.push_back(*edgeTo[v]);
                visit(v);
            }
            mstWeight = mst[0].wt();
            for(int i = 1;i < mst.size();i++)
                mstWeight += mst[i].wt();
        }
        ~PrimMST(){
            delete[] marked;
        }
        vector<Edge<Weight>> mstEdges(){
            return mst;
        }
        Weight result(){
            return mstWeight;
        }
    };
```

8.6 Krusk 算法

Kruskal 算法是基于贪心的思想得到的。首先我们把所有的边按照权值先从小到大排列，接着按照顺序选取每条边

，如果这条边的两个端点不属于同一集合，那么就将它们合并，直到所有的点都属于同一个集合为止。至于怎么合并到一个集合，那么这里我们就可以用到一个工具——-并查集。换而言之，Kruskal 算法就是基于并查集的贪心算法。

使用 Union Find 快速判断环

```
template<typename Graph, tyepname Weight>
class KruskMST{
private:
    vector<Edge<Weight>> mst;
    Weight mstWeight;
public:
    KruskMST(Graph &graph){
        MinHeap<Edge<Weight>> pq(graph.E());
        for(int i = 0;i < graph.V();i++){
            typename Graph::adjIterator adj(graph,i);
            for(Edge<Weight>* e = adj.begin();!adj.end();e=adj.next()){
                if(e->v() < e->w())
                    pq.insert(*e);
            }
        }
        UnionFind uf(graph.V());
        while(!pq.isEmpty() && mst.size() < graph.V() - 1){
            Edge<Weight> e = pq.extractMin();
            if(uf.isConnected(e.v(),e.w()))
                continue;
            mast.push_back(e);
            uf.unionElements(e.v(),e.w());
        }
        mstWeight = mst[0].wt();
        for(int i = 1;i < mst.size();i++)
            mstWeight += mst[i].wt();

    }
    ~KruskMST(){}
    vector<Edge<Weight>> mstEdges(){
        return mst;
    }
    Weight result(){
        return mstWeight;
    }
};
```

8.7 最小生成树问题

Lazy Prim O(ElogE)

Prim O(ElogV)

Kruskal O(ElogE)

如果横切边有相等的边的时候：

根据算法的具体实现，每次选择一个边

此时，图存在多个最小生成树

9.1 最短路径问题和松弛操作

应用：路径规划

广度优先遍历：

生成最短路径树，解决了单源最短路径

松弛操作：即更新两点的最短路径；原来用一根橡皮筋连接 a、b 两点，现在有一点 v 到 b 的距离更短，则把橡皮筋的 a 点换成 v 点，使得 v、b 连接在一起。这样缓解橡皮筋紧绷的压力，使其变得松弛，即松弛操作

松弛操作是最短路径求解的核心。

9.2 dijkstra 单源最短路径算法

前提：图中不能有负权边

复杂度 O(Elog(V))

```
template<typename Graph, tyepname Weight>
class Dijkstra{
private:
	Graph &G;
	int s;
	Weight *distTo;
	bool *marked;
	vector<Edge<Weight>*> from;
public:
	Dijkstra(Graph &graph, int s):G(graph){
		this->s = s;
		distTo = new Weight[G.V()];
		marked = new bool[G.V()];
		for(int i = 0;i < G.V();i++){
			distTo[i] = Weight();
			marked[i] = false;
			from.push_back(NULL);
		}

		indexMinHeap<Weight> ipq(G.V());
		//Dijkstra
		distTo[s] = Weight();
		marked[s] = true;
```

```
        ipq.insert(s,distTo[s]);
        while(ipq.isEmpty()){
            int v = ipq.extractMinIndex();
            //distTo[v] 就是 s 到 v 的最短距离
            marked[v] = true;
            //Relaxation
            typename Graph::adjIterator adj(G,v);
            for(Edge<Weight>* e = adj.begin();!adj.end();e=adj.next()){
                int w = e->other(v);
                if(!marked[w]){
                    if(from[w] == NULL || distTo[v] + e->wt() < distTo[w])
                    distTo[w] = distTo[v] + e->wt();
                    from[w] = e;
                    if(ipq.contain(w))
                        ipq.change(w,distTo[w]);
                    else
                        ipq.insert(w,distTo[w]);
                }
            }
        }
    }
    ~Dijkstra(){delete[] marked;}
    Weight shortestPathTo(int w){
        return distTo[w];
    }
    bool hasPath(int w){
        return marked[w];
    }
    void shortestPath(int w, vector<Edge<Weight>> &vec){
        stack<Edge<Weight>*> s;
        Edge<Weight> *e = from[w];
        while(e.v() != e->w()){
            s.push(e);
            e = from[e.v()];
        }

        while(!s.empty()){
            e = s.top();
            vec.push_back(*e);
            s.pop();
```

```
            }
        }
        void showPath(int w){
            assert(w>= 0 && w < G.V());
            vector<Edge<Weight>> vec;
            shortestPath(w,vec);
            for(int i = 0;i < vec.size();i++){
                cout << vec[i].v()<<"->";
                if(i == vec.size()-1)
                    cout << vec[i].w()<<endl;
            }
        }
    };
```

9.4 负权边和 BellMan-Ford 算法

拥有负权环的路径没有最短路径

Bellman-Ford 算法：

如果一个图没有负权环，从一个点到另外一个点的最短路径，最多经过所有的 V 个顶线，有 V-1 条边。

否则，存在定点经过了两次，即存在负权环。

前提：图中不能有负权环

Bellman-Ford 算法可以判断图中是否有负权环

复杂度：O(EV)

算法：

对所有的店进行 V-1 次松弛操作，理论上就找到了从原点到其他所有点的最短路径

如果好可以继续松弛，说明原图中存在负权环。

9.5 实现 Bellman-Ford 算法

```
template <typename Graph, typename Weight>
class BellmanFord
{
private:
    Graph &G;
    int s;
    Weight *distTo;
    vector<Edge<Weight>*> from;
    bool hasNegativeCycle;

    bool detectNegativeCycle()
    {
        for(int i = 0;i < G.V();i++)
            {
                typename Graph::adjIterator adj(G,i);
```

```
                for(Edge<Weight>* e = adj.begin();!adj.end();e=adj.next())
                {
                    if(!from[e->w()] || distTo[e->v()] + e->wt() < distTo[e->w()])
                    {
                        return true;
                    }
                }
            }
            return false;
        }
public:
    BellmanFord(Graph &graph, int s):G(graph)
    {
        this->s = s;
        distTo = new Weight[G.V()];
        for(int i = 0; i < G.V(); i++)
        {
            from.push_back(NULL);
        }
        //BellmanFord
        distTo[s] = Weight();

        for(int pass = 1;pass < G.V();pass++)
        {
            for(int i = 0;i < G.V();i++)
            {
                typename Graph::adjIterator adj(G,i);
                for(Edge<Weight>* e = adj.begin();!adj.end();e=adj.next())
                {
                    if(!from[e->w()] || distTo[e->v()] + e->wt() < distTo[e->w()])
                    {
                        distTo[e->w()] = distTo[e->v()] + e->wt();
                        from[e->w()] = e;
                    }
                }
            }
        }

        hasNegativeCycle = detectNegativeCycle();
    }
```

```
        ~BellmanFord()
        {
            delete[] distTo;
        }
        bool negativeCycle()
        {
            return hasNegativeCycle;
        }
        Weight shorttestPathTo(int w)
        {
            assert(w >= 0 && w < G.V());
            assert( !hasNegativeCycle );
            return distTo[w];
        }
        bool hasPathTo(int w)
        {
            assert(w >= 0 && w < G.V());
            return from[w] != NULL;
        }
        void shorttestPath(int w, vector<Edge<Weight>> &vec)
        {
            assert(w >= 0 && w < G.V());
            assert( !hasNegativeCycle );
            stack<Edge<Weight>*> s;
            Edge<Weight> *e = from[w];
            while(e->v() != this->s)
            {
                s.push(e);
                e = from[e->v()];
            }
            while(!s.empty())
            {
                e = s.top();
                vec.push_back(*e);
                s.pop();
            }
        }

    };
```

9.6 小结：

dijkstra　　　无负权边　　　　有无无向图均可　O(ElogV)
Bellman-Ford　无负权环　　　　有向图　　　　　O(VE)
利用拓扑顺序　　有向无环图 DAG　有向图　　　　　O(V+E)

所有对最短路径算法
Floyed 算法（利用到了动态规划），处理无负权环的图
最长路径算法

10.总结
线性（排序）--->>>树形结构---->>>图形结构
线性问题（排序）
O(n^2)　选择排序　　插入排序
O(nlogn)　归并排序　　快速排序　　三路快排(partition->随机化->大量重复元素)
O(nlogn)　堆排序

树形问题
堆(Heap) 堆排序　优先队列　　索引堆(Prim,Dijkstra)
二叉查找树(Binary Search Tree)　解决查找问题
二分查找法　动态维护：插入，删除，查找，遍历，顺序相关问题
并查集(Union Find)　　基于 rank 的优化->路径压缩　　Kruskal

图论问题
图的表示：邻接表和邻接矩阵　有向图和无向图　有权图和无权图
图的遍历：DFS 深度优先,BFS 广度优先
联通分量，Flood Fill，寻路
最小生成树问题:Prim Kruskal
最短路径问题：Dijkstra Bellman-Ford

更多算法问题：
数据结构相关
比如：双向队列，斐波那契堆，红黑树，区间树，KD 树...
具体领域相关
数学；数论，计算几何
图论；网络流
算法设计相关
分治：归并排序；快速排序；树结构
贪心：选择排序；堆；Kruskal;Prim;Dijkstra
递归回溯：树的遍历；图的遍历
动态规划：Prim;Dijkstra