第五章 数组与字符串

C++简明双链教程（李昕著，清华大学出版社）

作者：李昕

PPT制作者：杨述敏

目录

01 一维数组的定义和初始化

02 一维数组的应用

03 二维数组

04 C++的字符串

05 C风格的字符串

06 题单

01 一维数组的定义和初始化

1.1 一维数组的定义

在程序设计中，为了便于程序处理，通常把具有相同类型的若干变量按有序的形式组织在一起，这些按序排列的同类数据元素的集合称为数组。**语法格式**如下：

数据类型 数组名[元素数量];

```
int a[10];
```

此行定义了一个包含10个int类型元素的数组，数组名为a，这10个元素分别是a[0], a[1], ...a[9]。

特别强调，C/C++中区分定义语句和非定义语句。

[]=
- 数组定义　数组中元素的个数
  - 例如 int a[10];
- 数组使用　数组元素的下标
  - 例如 a[1]=3;

知识点

索引	要点	正链	反链
T511	掌握数组的基本定义，索引从0开始，中括号在定义语句和非定义语句中的含义		T531,T541
	洛谷：U270927(LX501)		

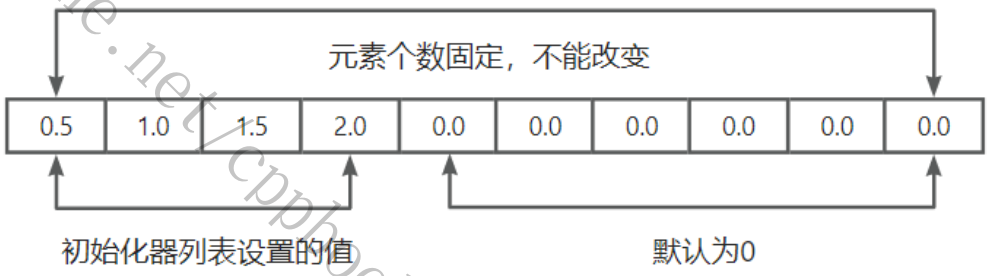
1.2 一维数组的初始化

定义后用大括号中的数值对各个元素依次进行赋值。**数值个数不能超过数组定义时的元素数量**。如果用全部数值进行了初始化，元素数量在定义时可以省略。

```
1  int main(){
2      int a[5] = {1, 2, 3, 4, 5};
3      double b[] = {7.1, 8.2, 9.3};
4      double c1[10] = {0.5,1.0,1.5,2.0};
5      long long c2[100] = {0};
6      long long c3[100] = {};
7      int d[50];
8  }
```

数组b根据初始化数值的数量确定元素个数为3。

数组c1的前4个数据与初始化列表对应，根据C/C++的规则，**部分初始化时，未赋值元素为0**，因此c1[4]以后的元素为0。



```
1  int main(){
2      int a[5] = {1, 2, 3, 4, 5};
3      double b[] = {7.1, 8.2, 9.3};
4      double c1[10] = {0.5,1.0,1.5,2.0};
5      long long c2[100] = {0};
6      long long c3[100] = {};
7      int d[50];
8  }
```

根据部分初始化规则，可以用第5行的方法将所有数值初始化为全0。这是利用了规则，并不存在全1或其他数值的全部初始化操作。

采用第6行的方式，也可以将数组全部初始化为0。

如果没有初始化，数组中所有元素的值是不确定的。变量定义时，只会分配空间，没有自动赋值为0的操作。例如数组d中的所有值是不确定的。

定义动态数组

元素数量必须是非负整数，可以是常量，也可以是变量。**如果定义数组时元素数量为常量，称为静态数组**，在编译时分配存储空间，因为存储空间确定，所以可以进行初始化；**如果元素数量是变量，称为动态数组**，在运行时分配存储空间，编译时不能确定存储空间的大小。从C++11开始，动态数组可以进行初始化。

```
int main(){
    int n;
    cin>>n;
    int arr[n]={1,2};          //动态数组，从C++11开始支持初始化
    for(int i=0;i<n;i++)
        cout<<arr[i]<<' ';
}
```

样例输入	样例输出
100	5050

知识点

索引	要点	正链	反链
T512	掌握数组的初始化方法，尤其是部分初始化的作用；理解动态数组		
	洛谷：U271197(LX503)		

1.3 一维数组的内存模型

从存储角度，当定义一个数组a时，编译器根据指定的元素个数和元素的类型分配确定大小（元素类型大小*元素个数）的一块内存，并把这块内存的名字命名为 a，名字 a 一旦与这块内存匹配就不能再改变。由此可知，**一个数组中所有元素的存储空间是连续的**。对于一个数组float a[]={1.2, 2.3, 3.4, 4.5, 5.6}，各元素的相关数据如下：

```
#include<iostream>
using namespace std;
int main (  )
{
    float a[]={1.2, 2.3, 3.4, 4.5,5.6};
    cout<<"下标";
    for(int i=0;i<5;i++)
        cout<<"\ta["<<i<<']';
    cout<<endl;
    cout<<"值";
    for(int i=0;i<5;i++)
        cout<<'\t'<<a[i];
    cout<<endl;
    cout<<"地址";
    for(int i=0;i<5;i++)
        cout<<'\t'<<&a[i];
    cout<<endl;
    cout<<"a+i";
    for(int i=0;i<5;i++)
        cout<<'\t'<<a+i;
    cout<<endl;
}
```

下标	a[0]	a[1]	a[2]	a[3]	a[4]
值	1.2	2.3	3.4	4.5	5.6
地址	0x61fe00	0x61fe04	0x61fe08	0x61fe0c	0x61fe10
a+i	0x61fe00	0x61fe04	0x61fe08	0x61fe0c	0x61fe10

表格第3行说明，**所有元素的内存地址连续**，间隔为sizeof(float)。

表格第3、4行说明，a+i与&a[i]相同，都是表示第i个元素的地址。这是因为**数组名代表了数组首元素的地址**，简称首地址，即a+0=a=0x61fe00。

下标	a[0]	a[1]	a[2]	a[3]	a[4]
值	1.2	2.3	3.4	4.5	5.6
地址	0x61fe00	0x61fe04	0x61fe08	0x61fe0c	0x61fe10
a+i	0x61fe00	0x61fe04	0x61fe08	0x61fe0c	0x61fe10

a+1中的1不代表一个字节，而是表示一个元素的空间，即sizeof(float)。因此**第i个元素的地址为a+i*sizeof(float)**。

总而言之，数组可以通过偏移快速定位第i个元素。偏移在计算机中是一个非常快速的基本运算，这也是数组能够进行快速访问的根本原因。

知识点

索引	要点	正链	反链
T513	数组的连续内存分配模型，通过偏移快速定位元素是数组的突出优势。 理解数组的物理空间和有效元素个数是不同的。	T341	T521,T525, T528,T542, T621

1.4 数组的基本运算

C/C++中的数组虽然可以看做一个整体，但并不是一种独立存在的数据类型。按照语法规定，不能整体赋值、整体比较、整体输入输出。**当需要进行赋值或比较或输入输出时，需要通过循环逐个元素的进行。**

例题5.1

输入n个同学的成绩，输出其中小于平均分的成绩。

样例输入	样例输出
5 7 6 5 3 1	3 1

本例题中所有元素要使用两遍，当得到平均值后，必须第二次遍历数组，因此必须使用数组记录每个元素。

当需要对每个元素与平均值进行比较时，必须逐个元素进行比较，C/C++中没有提供整体比较的语法支持。

```
1   #include<iostream>
2   using namespace std;
3   int main (  )
4   {
5       int n;
6       cin>>n;
7       int score[n];
8       double sum=0;
9       for(int i=0;i<n;++i){
10          cin>>score[i];
11          sum += score[i];
12      }
13      for(int i=0;i<n;++i)
14          if(score[i]<sum/n)
15              cout<<score[i]<<' ';
16  }
```

- 第7行，元素数量是变量，数组是一个动态数组，不可以直接进行初始化。只能通过第9-12行的循环，逐个进行赋值。

知识点

索引	要点	正链	反链
T514	只有多个数据反复利用时，才需要数组；单次使用多个数据尽量不用数组。 数组不能整体赋值、整体比较、整体输入输出，必须与循环结合。		
	洛谷：U270929(LX502), U271201(LX504), U271200(LX505)		

1.5 数组作为函数参数

数组名a存储了首地址，并不存储元素的数量，只是从首地址开始，通过偏移访问各个元素。因此当把数组作为函数的参数时，实参数组会把它的地址传递给形参数组，但数组的元素数量并不会被传递。因此**数组作为函数的参数时，必须同时传递数组地址和数组中元素的数量，否则无法知道数组的有效范围**。

样例输入	样例输出
5 7 3 8 1 9	3

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

int argmin(int n,int arr[]){     //数组的元素数量n和数组的首地址
    int min = 0;                 //默认下标为0的元素最小
    for(int i=1;i<n;++i)
        if(arr[i]<arr[min])      //当前元素和最小值元素进行比较
            min = i;             //min保留最小值的下标
    return min;
}

int main (  )
{
    int n;
    cin>>n;
    int a[n+10];
    for(int i=0;i<n;++i)
        cin>>a[i];
    cout<<argmin(n,a)<<endl;
    return 0;
}
```

```
1   #include<iostream>
2   using namespace std;
3
4   int argmin(int n,int arr[]){    //数组的元素数量n和数组的首地址
5       int min = 0;                //默认下标为0的元素最小
6       for(int i=1;i<n;++i)
7           if(arr[i]<arr[min])     //当前元素和最小值元素进行比较
8               min = i;            //min保留最小值的下标
9       return min;
10  }
11
12  int main (  )
13  {
14      int n;
15      cin>>n;
16      int a[n+10];
17      for(int i=0;i<n;++i)
18          cin>>a[i];
19      cout<<argmin(n,a)<<endl;
20      return 0;
21  }
```

- 由左侧代码可以看出，**把数组作为函数的参数时，必须同时传递元素的数量和数组的地址**。

- 关注第4行的形参arr，可以看到[]为空。数组定义时，理论上[]中应该注明元素的数量。但是arr是形参，在argmin函数被调用之前，它没有存储空间。在argmin函数被调用之后，**它的作用就是存放实参的首地址，因此元素数量对它没有任何意义**。因此[]为空，或者写成[0]或其他任何整数，对程序运行都没有任何影响。由此可以进一步了解，**数组名不包含任何的元素数量信息**。

- 将int arr[]改写成auto arr，根据实参赋值决定形参arr的数据类型，也是可以的。

知识点

索引	要点	正链	反链
T515	数组作为函数的参数，只是传递首元素地址，与实参共享存储空间。		T625
	洛谷：U271215(LX512)		
T516	掌握求数组极值及极值对应下标的方法。		
	洛谷：U271202(LX506)		

02 一维数组的应用

2.1 数组的插入与删除

数组的存储空间一定是连续的，因此对于非尾部数据的插入和删除是无法物理实现的，只能通过逻辑方式满足需求。**删除时，将被删除元素右侧的所有元素向前平移，插入时将所有元素向后平移，留出插入空间。**

样例输入	样例输出
5 2 1 2 3 4 5	1 2 4 5 1 2 37 4 5

```cpp
#include<iostream>
using namespace std;
bool remove(int n,int arr[],int pos){   //删除长度为n的数组的第pos个元素
    if(pos<0 || pos>=n)  return false;
    for(int i=pos;i<n-1;++i)    //正序遍历
        arr[i]=arr[i+1];        //删除位置右侧的值向左移动
    return true;
}
bool insert(int n,int arr[],int pos,int val){//在长度为n的数组的第pos个位置插入新元素val
    if(pos<0)  return false;
    if(pos>n)   pos = n;        //如果插入位置过大，把数据添加到数组的末尾
    for(int i=n-1;i>=pos;--i)   //必须倒序循环，保证数据不被覆盖
        arr[i+1] = arr[i];      //插入位置右侧的值向右移动
    arr[pos] = val;             //将新值放到空白处
    return true;
}
void print(int n,int arr[]){
    for(int i=0;i<n;++i)
        cout<<arr[i]<<(i<n-1?' ':'\n');//控制输出间隔
}
int main (  )
{
    int n,index;
    cin>>n>>index;
    int values[n];
    for(int i=0;i<n;++i)
        cin>>values[i];
    n-=remove(n,values,index);
    print(n,values);
    n+=insert(n,values,index,37);
    print(n,values);
}
```

```cpp
#include<iostream>
using namespace std;
bool remove(int n,int arr[],int pos){   //删除长度为n的数组的第pos个元素
    if(pos<0 || pos>=n)  return false;
    for(int i=pos;i<n-1;++i)     //正序遍历
        arr[i]=arr[i+1];         //删除位置右侧的值向左移动
    return true;
}
bool insert(int n,int arr[],int pos,int val){//在长度为n的数组的第pos个位置插入新元素val
    if(pos<0)  return false;
    if(pos>n)   pos = n;         //如果插入位置过大，把数据添加到数组的末尾
    for(int i=n-1;i>=pos;--i)    //必须倒序循环，保证数据不被覆盖
        arr[i+1] = arr[i];       //插入位置右侧的值向右移动
    arr[pos] = val;              //将新值放到空白处
    return true;
}
void print(int n,int arr[]){
    for(int i=0;i<n;++i)
        cout<<arr[i]<<(i<n-1?' ':'\n');//控制输出间隔
}
int main (  )
{
    int n,index;
    cin>>n>>index;
    int values[n];
    for(int i=0;i<n;++i)
        cin>>values[i];
    n-=remove(n,values,index);
    print(n,values);
    n+=insert(n,values,index,37);
    print(n,values);
}
```

- **插入时一定要倒序移位**，因为正序移位会出现元素覆盖和数据丢失。请自行尝试体验。

- 插入或删除成功返回true，否则返回false。第28和30行借助true/false和1/0的对应关系，对数组元素数量进行修正。也就是说，**数组的物理存储空间不会改变，只是从逻辑上认为元素的数量发生变化**。

```cpp
#include<iostream>
using namespace std;
bool remove(int n,int arr[],int pos){   //删除长度为n的数组的第pos个元素
    if(pos<0 || pos>=n)  return false;
    for(int i=pos;i<n-1;++i)     //正序遍历
        arr[i]=arr[i+1];         //删除位置右侧的值向左移动
    return true;
}
bool insert(int n,int arr[],int pos,int val){//在长度为n的数组的第pos个位置插入新元素val
    if(pos<0)  return false;
    if(pos>n)   pos = n;         //如果插入位置过大，把数据添加到数组的末尾
    for(int i=n-1;i>=pos;--i)    //必须倒序循环，保证数据不被覆盖
        arr[i+1] = arr[i];       //插入位置右侧的值向右移动
    arr[pos] = val;              //将新值放到空白处
    return true;
}
void print(int n,int arr[]){
    for(int i=0;i<n;++i)
        cout<<arr[i]<<(i<n-1?' ':'\n');//控制输出间隔
}
int main (  )
{
    int n,index;
    cin>>n>>index;
    int values[n];
    for(int i=0;i<n;++i)
        cin>>values[i];
    n-=remove(n,values,index);
    print(n,values);
    n+=insert(n,values,index,37);
    print(n,values);
}
```

- **对于插入而言，一定要保证物理存储空间足够用**，不要在插入时发生下标超上限的现象。

- 第18-19行对数据分隔显示做了一个示范。对于在线评测系统，多出一个空格，可能会导致整个题目被判错，一定严格遵守题目的输出规范和要求。

知识点

索引	要点	正链	反链
T521	数组只能对单个元素做逻辑插入和删除，注意循环移位时的元素覆盖问题。	T513	

2.2数组与循环的联动

有时在题目中没有明显需要数组的提示，可以采用数组记录已知的数据，利用数组可以跟循环联动的特点，极大简化程序的流程。

例题5.2

把1月1日当做第1天，当用户输入年份和第n天时，输出第n天是几月几日？

样例输入	样例输出
2022 33	2月2日

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int year,n;
7       cin>>year>>n;
8       int month[12]={31,28,31,30,31,30,31,31,30,31,30,31};
9       auto leapyear = [](int year){return year%400==0||(year%4==0 && year%100!=0);};
10      month[1]+=leapyear(year);
11      int i=0;
12      while(n>month[i])
13          n-=month[i++];
14      cout<<i+1<<"月"<<n<<"日"<<endl;
15      return 0;
16  }
```

- 第8行定义了一个数组，记录了每个月的天数。

- 第9行赋值号右侧是一个匿名函数，[]表示后面定义了一个函数，参数和函数体的写法和普通函数定义相同。**对于简单的函数，或只需要使用一次的函数，可以采用匿名函数的方式进行定义**。

- 将定义后的匿名函数赋值给auto变量leapyear，此时的leapyear就是一个函数。auto表示根据赋值号右侧的内容自动解析变量的类型。此处的leapyear被解析为函数。

例题5.2

把1月1日当做第1天，当用户输入年份和第n天时，输出第n天是几月几日？

样例输入	样例输出
2022 33	2月2日

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int year,n;
7       cin>>year>>n;
8       int month[12]={31,28,31,30,31,30,31,31,30,31,30,31};
9       auto leapyear = [](int year){return year%400==0||(year%4==0 && year%100!=0);};
10      month[1]+=leapyear(year);
11      int i=0;
12      while(n>month[i])
13          n-=month[i++];
14      cout<<i+1<<"月"<<n<<"日"<<endl;
15      return 0;
16  }
```

- 第10行利用true/false和1/0的对应关系，将闰年的2月修改为29天。
- 第12-13行，利用数组和循环的联动，快速简单地定位了月份和日期。
- 第13行的++在i后面，表示**先执行当前表达式的运算，然后再对i进行加1操作**。即先执行n-=month[i]，再执行i++。将两条语句简化成了一条语句，但是功能完全相同。

知识点

索引	要点	正链	反链
T522	单循环与数组搭配使用，嵌套循环与二维数组搭配使用。		T874
	洛谷：U271204(LX507), U271209(LX508)		
T523	掌握匿名函数的基本使用方法，理解这种形式，不做重点掌握。		

2.3 尺取法

尺取法又称双指针法，用来解决序列的区间问题，是一种常见的优化技巧。分为反向扫描法和同向扫描法。

反向扫描法(左右指针法)：设定两个指针i和j，分别指向数组的头和尾，i和j方向相反，i从头向尾，j从尾向头，在中间集合。虽然设定两个指针，但是对同一个数组同时遍历，算法复杂度为$O(n)$。

同向扫描法(快慢指针法)：设定两个指针i和j，同时指向数组的头或尾，i和j移动方向相同，但偏移速度不同。关键是一个指针处的修改不能影响另外一个指针的遍历。

例题5.3

给定一个数组arr，判断数组是否对称。

```
1   bool sym(int n,int arr[])
2   {
3       for(int i=0,j=n-1;i<j;++i,--j)
4           if(arr[i]!=arr[j])
5               return false;
6       return true;
7   }
```

注意第3行两个变量同时变化遍历数组的方法。

随堂练习5.1

给定一个数组，采用尺取法将数组逆序。

ps：注意交换要在数组的中间停止，否则会把已经逆序的数组重新修改为正序。

例题5.4

给定一个数组arr，要求删除其中的指定值val。

方案一：基于2.1节中的删除元素方法，每次删除一个值，算法复杂度为$O(n^2)$。

方案二：新建一个数组，将原数组中的有效值添加到新数组中。以空间换时间，算法复杂度降为$O(n)$。

第二种方法更好一些。但在一些特殊情况下，要求在原数组上删除指定元素，不允许建立新数组。仔细分析，一个删除后的数组，元素的数量一定小于或等于原数组，因此可以**设置两个索引i和index，i遍历原数组，index遍历保留的元素**。因为index小于等于i，因此index处的赋值不会影响i的遍历。

例题5.4

给定一个数组arr，要求删除其中的指定值val。

```
1   int remove(int n,int arr[],int val)
2   {
3       int index = 0;
4       for(int i=0;i<n;++i){
5           if(arr[i]!=val){
6               arr[index] = arr[i];
7               index++;
8           }
9       }
10      return index;
11  }
```

- index小于等于i，因此第6行的赋值操作对第4行正在进行的遍历操作不会造成任何影响。

- 一次性删除所有指定的值val。函数最后返回index，代表了保留元素的个数。

- 第八章将要提到的STL的remove算法与以上代码的想法完全一致。既不需要创建新空间，算法复杂度也降低到$O(n)$。

随堂练习5.2

移除一个数组中的重复元素。

ps：依次遍历每个元素，删除其后继元素中与其值相同的元素。

例题5.5

给定两个按非递减顺序的整数数组 nums1 和 nums2，元素数量分别为 m 和 n 。合并 nums2 到 nums1 中，使合并后的数组同样保持非递减顺序。nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素。

方案一：将nums2直接拼接到nums1的尾部，然后采用快速排序，重新达成非递减顺序，算法复杂度为$O((m+n)log(m+n))$。并没有用到原数组已经有序的条件。

方案二：新建一个数组，依次将两个数组中符合条件的数据添加到新数组中。以空间换时间，算法复杂度降为$O(m+n)$。

当不允许建立新空间时，主要的问题是一个数据可能未被处理，就被新数据覆盖。如果倒序遍历，先让索引p指向最后一个元素的位置m+n-1，这样p一定大于等于m或n，因此得到以下方法：

例题5.5

给定两个按非递减顺序的整数数组 nums1 和 nums2，元素数量分别为 m 和 n 。合并 nums2 到 nums1 中，使合并后的数组同样保持非递减顺序。nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素。

```
1  void merge(int nums1[], int m, int nums2[], int n) {
2      int p=m--+(--n);
3      while(m>=0&&n>=0){      //或while(m+1&&n+1)
4          nums1[p--] = nums1[m]>nums2[n]?nums1[m--]:nums2[n--];
5      }
6      while(n>=0){            //或while(n+1)
7          nums1[p--] = nums2[n--];
8      }
9  }
```

- 第2行将p指向数据尾部，注意m--和--n的使用，之所以使用--n，是因为p应该等于m+n-1，所以要先减1。（知识点：T268）
- p一定大于等于m或n，因此第4行的赋值操作对第3行正在进行的遍历操作不会造成任何影响。
- 第6-8行的循环是为了处理nums1已经被处理完毕，但是nums2还有残留数据，这些数据必须迁移到nums1中。如果nums1还有残留，那么正好处于应有的位置，不需要处理。

例题5.5

给定两个按非递减顺序的整数数组 nums1 和 nums2，元素数量分别为 m 和 n 。合并 nums2 到 nums1 中，使合并后的数组同样保持非递减顺序。nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素。

如果nums1已经处理完毕，可以只处理nums2，因此可以将以上代码中的两个循环简化成一个循环。具体代码如下：

```
1  void merge(int nums1[], int m, int nums2[], int n) {
2      int p=m--+(--n);
3      while(n>=0){          //或while(m+1&&n+1)
4          nums1[p--] = m>=0&&nums1[m]>nums2[n]?nums1[m--]:nums2[n--];
5      }
6  }
```

知识点

索引	要点	正链	反链
T524	掌握尺取法多指针反向或同向扫描法，掌握多变量方式对序列的遍历，能够把对称判断、原地删除和合并等方法作为解题模板。	T475,T476	T547

2.4 空间换时间

有时需要在两个以上的多维度上对数据进行遍历，或多单一维度进行多重遍历时，计算复杂度高。通过把数组作为中间媒介，可以实现降维，把嵌套循环简化为并列循环，甚至单循环，能够极大地降低算法的复杂度。这是一种用空间换时间的思路。

例题5.6

长度为L的长江路上有一排树。如果把长江路看成一个0~L的数轴，则数轴上的每个整数0,1,2,...L都种有一棵树。由于长江路部分区域要建地铁。这些区域用它们在数轴上的起始点（整数）和终止点（整数）表示，区域之间可能有重合的部分。现在要把建地铁区域的树(包括区域端点)移走，计算移走后路上还有多少棵树。

样例输入	样例输出
10 3 6 5 7 10 10	5

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

int main (  )
{
    int L;
    cin>>L;
    bool tree[L+1];
    for(auto &e:tree)    e = 1;
    int left,right;
    while(cin>>left>>right)
        for(int i=left;i<=right;++i)    //将删除区域的值修改为0
            tree[i] = 0;
    int sum = 0;
    for(auto e:tree)
        sum += e;
    cout<<sum<<endl;
    return 0;
}
```

【问题分析】从简单的思维出发，判断每棵树是否在给定的所有范围里。假定共有R个范围，在树和范围两个维度进行嵌套遍历，时间复杂度为O(L*R)。这种嵌套循环复杂度高，而且容易出错。仔细分析，每棵树只有保留或移走两种状态，可以通过数组记录状态变化。更重要的是，以数组为媒介，可以将嵌套循环拆解为并列循环，时间复杂度降为O(L+R)。

例题5.6

长度为L的长江路上有一排树。如果把长江路看成一个0~L的数轴，则数轴上的每个整数0,1,2,...L都种有一棵树。由于长江路部分区域要建地铁。这些区域用它们在数轴上的起始点（整数）和终止点（整数）表示，区域之间可能有重合的部分。现在要把建地铁区域的树(包括区域端点)移走，计算移走后路上还有多少棵树。

样例输入	样例输出
10 3 6 5 7 10 10	5

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int L;
7       cin>>L;
8       bool tree[L+1];
9       for(auto &e:tree)     e = 1;
10      int left,right;
11      while(cin>>left>>right)
12          for(int i=left;i<=right;++i)     //将删除区域的值修改为0
13              tree[i] = 0;
14      int sum = 0;
15      for(auto e:tree)
16          sum += e;
17      cout<<sum<<endl;
18      return 0;
19  }
```

- 第8行，tree是一个动态数组，不能进行直接初始化。第9行用范围for的形式将数组全部初始化为1。**注意这里要改变每个元素的值，因此e采用引用形式。**
- 第14-16行不采用判断，直接将所有数值求和得到剩余树的数量。
- 第9行和第15-16行遍历所有的树，第11-13行遍历所有的范围。这两个遍历的中间媒介是数组tree，两个遍历形成并列关系而不是嵌套关系。

例题5.7

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？其中$1 \leq n \leq 90$。

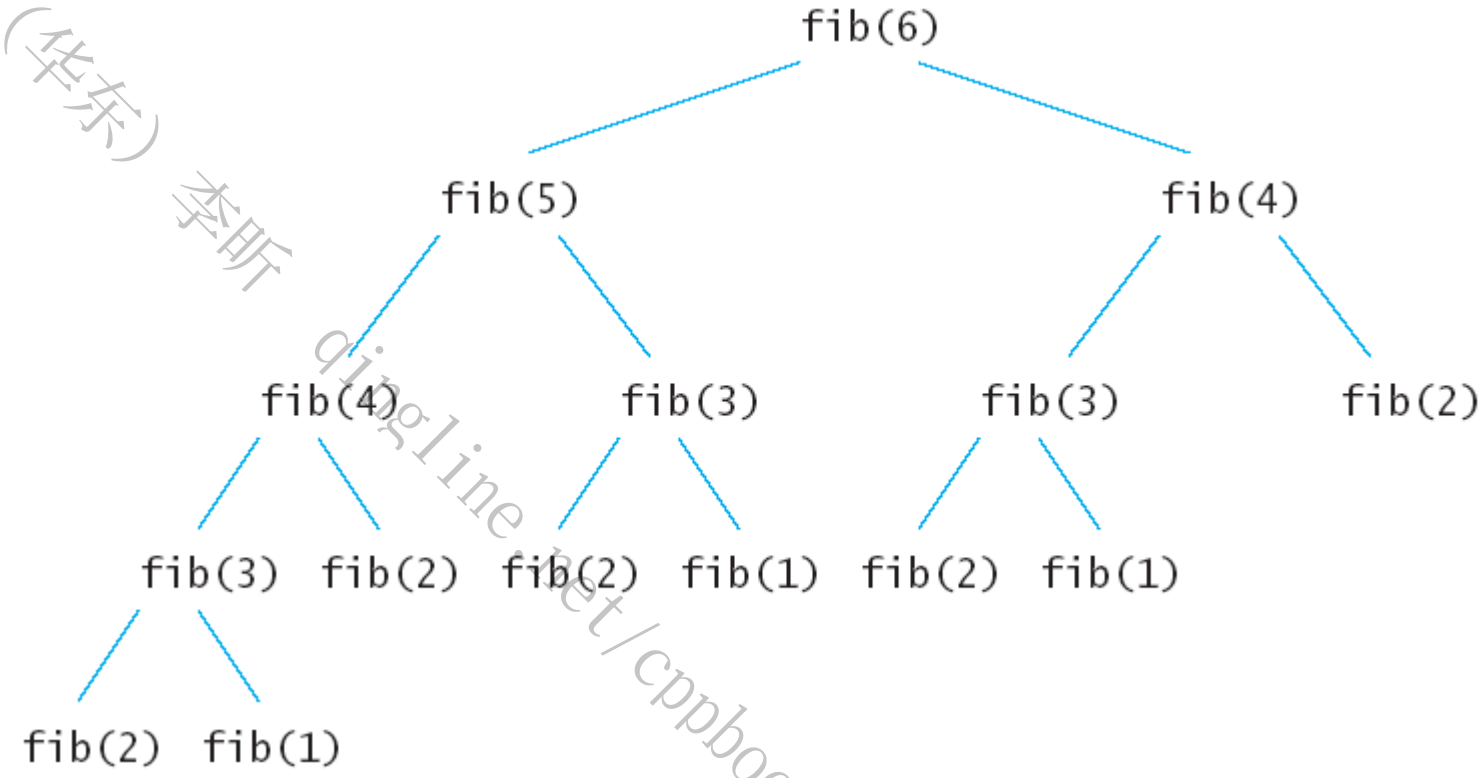
仔细分析题目，如果上到第n-2阶台阶共有f(n-2)种方法，上到第n-1阶台阶共有f(n-1)种方法，则f(n)=f(n-1)+f(n-2)，这其实是一个斐波那契数列。采用递归求解非常简单。

```cpp
#include<iostream>
using namespace std;
long long fib(int n){
    if(n==1||n==2)return n;
    return fib(n-1)+fib(n-2);
}
int main ()
{
    int n;
    cin>>n;
    long long num = fib(n);
    cout<<num<<endl;
    return 0;
}
```

例题5.7

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？其中$1 \leq n \leq 90$。

整个程序一目了然，但是当输入为50时，在codeblocks上的运行时间为55.5秒。耗时的主要原因是重复计算，以求 fib(6) 为例，见下图。n 越大，重复的越多，耗时就越长。



例题5.7

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？其中$1 \leq n \leq 90$。

事实上，如果已经计算过fib(n)，可以存储起来，下次用到的时候直接使用，就可以极大的加快计算速度。这是一个比较经典的以空间换时间的操作。这种方法称为**带备忘录的递归方法**，可以解决绝大部分递归超时问题。

```
1   #include<iostream>
2   using namespace std;
3   long long ret[100]={0,1,2};
4   long long fib(int n){
5       if(ret[n])    return ret[n];
6       ret[n]=fib(n-1)+fib(n-2);
7       return ret[n];
8   }
9   int main ()
10  {
11      int n;
12      cin>>n;
13      long long num = fib(n);
14      cout<<num<<endl;
15      return 0;
16  }
```

- 第3行给定了一个全局变量数组，并初始化了fib(1)和fib(2)的值，其余默认为0。**全局变量的生命周期贯穿整个程序的运行，可以在任意位置使用。**
- 第6行将每次计算的结果保存到数组ret中的对应位置上。计算时第5行先进行判断，如果fib(n)已经计算过（不为0），直接返回结果，这样极大地加快了计算速度。

例题5.7

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？其中$1 \leq n \leq 90$。

全局变量可以在任意处使用，可能会造成程序的混乱，因此并不建议使用。以下代码采用static静态变量。

```
1  long long fib(int n){
2      static long long ret[100]={0,1,2};
3      if(ret[n])    return ret[n];
4      ret[n]=fib(n-1)+fib(n-2);
5      return ret[n];
6  }
```

第2行定义的ret是静态变量，**只能在函数内部使用**。但是其**生命周期与全局变量一致，贯穿整个程序的运行过程**。第2行的代码在反复调用fib函数的过程中会多次执行到该位置，但是只在第一次执行的时候起作用，后继执行到该位置，会自动忽略第2行语句，因此**静态变量的定义和初始化只会执行一次**，其效果与全局变量完全相同。

知识点

索引	要点	正链	反链
T525	以空间消耗换取时间效率是算法优化的基本方法。 利用全局变量或静态变量构建数组，实现递归的快速计算，注意静态变量的使用。	T513	T526, T528

2.5 打表法

在一些题目中，某些计算过程需要反复使用，造成了时间的严重消耗，可能造成超时问题。遇到这种情况，可以一次性计算所有可能输入对应的的结果，并保存到数组中，之后直接查询。这种方式主要对每个可能的计算只操作一遍，从而达到了节省时间的目的。这种利用数组的方式称为打表法。注意**这个技巧只适用于输入的值域不大的问题，否则可能会导致内存超限、时间超限等问题。**

例题5.8

给定n个不同的非负整数，求这些数中有多少对整数的值正好相差1。

【输入】
第一行包含一个整数n（1≤n≤1000），表示给定非负整数的个数。
第二行包含n个给定的非负整数，每个整数不超过10000。
【输出】
这n个非负整数中有多少对整数的值正好相差1。

样例输入	样例输出
6 10 2 6 3 7 8	3

【样例说明】相差为1的整数对包括(2,3), (6,7), (7,8)

朴素的方法是逐个枚举，使用双重嵌套循环，时间复杂度为$O(n^2)$。也可以先排序，然后检测相邻元素是否符合题目规定，时间复杂度为 $O(nlogn)$，即排序的复杂度。根据题目说明，每个非负整数最大不超过10000，**最佳方式是创建一个元素个数为10000的数组，将所有数据标定出来，再进行相邻元素检测**。时间复杂度降为$O(n)$。

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

int main (  )
{
    int n;
    cin>>n;
    bool cnt[10010]= {0};                //给定一些冗余空间，防止边界错误
    int x,min=10000,max=0,ans;
    for(int i=0; i<n; ++i){
        cin>>x;
        if(x<min)   min = x;             //求最小值
        if(x>max)   max = x;             //求最大值
        cnt[x] = 1;                      //对应x的位置有数值
    }
    for(int i=min+1;i<=max;++i)          //遍历所有有效数值
        ans += (cnt[i]+cnt[i-1]==2);     //如果相邻两个元素都有效，则相加必为2
    cout<<ans<<endl;
    return 0;
}
```

例题5.8

给定n个不同的非负整数，求这些数中有多少对整数的值正好相差1。

【输入】
第一行包含一个整数n（1≤n≤1000），表示给定非负整数的个数。
第二行包含n个给定的非负整数，每个整数不超过10000。
【输出】
这n个非负整数中有多少对整数的值正好相差1。

样例输入	样例输出
6 10 2 6 3 7 8	3

【样例说明】相差为1的整数对包括(2,3), (6,7), (7,8)

- 本题目借助数组，虽然浪费了一定的空间，但是极大降低了算法的复杂度。当对cnt进行赋值后，实际上就是按空间顺序完成了排序，又不需要排序那么复杂。

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

int main (  )
{
    int n;
    cin>>n;
    bool cnt[10010]= {0};                //给定一些冗余空间，防止边界错误
    int x,min=10000,max=0,ans;
    for(int i=0; i<n; ++i){
        cin>>x;
        if(x<min)   min = x;             //求最小值
        if(x>max)   max = x;             //求最大值
        cnt[x] = 1;                      //对应x的位置有数值
    }
    for(int i=min+1;i<=max;++i)          //遍历所有有效数值
        ans += (cnt[i]+cnt[i-1]==2);     //如果相邻两个元素都有效，则相加必为2
    cout<<ans<<endl;
    return 0;
}
```

例题5.8

给定n个不同的非负整数，求这些数中有多少对整数的值正好相差1。

【输入】
第一行包含一个整数n（1≤n≤1000），表示给定非负整数的个数。
第二行包含n个给定的非负整数，每个整数不超过10000。
【输出】
这n个非负整数中有多少对整数的值正好相差1。

样例输入	样例输出
6 10 2 6 3 7 8	3

【样例说明】相差为1的整数对包括(2,3), (6,7), (7,8)

- 第8行元素只有10000个，但是边界是最容易出问题的地方，因此额外定义了10个空间。

- 第16行的遍历，可以是0~10000，但在输入过程中求解了最大值和最小值，缩小了遍历的范围。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int n;
7       cin>>n;
8       bool cnt[10010]= {0};                   //给定一些冗余空间，防止边界错误
9       int x,min=10000,max=0,ans;
10      for(int i=0; i<n; ++i){
11          cin>>x;
12          if(x<min)   min = x;                //求最小值
13          if(x>max)   max = x;                //求最大值
14          cnt[x] = 1;                         //对应x的位置有数值
15      }
16      for(int i=min+1;i<=max;++i)             //遍历所有有效数值
17          ans += (cnt[i]+cnt[i-1]==2);        //如果相邻两个元素都有效，则相加必为2
18      cout<<ans<<endl;
19      return 0;
20  }
```

例题5.8

给定n个不同的非负整数，求这些数中有多少对整数的值正好相差1。

【输入】
第一行包含一个整数n（1≤n≤1000），表示给定非负整数的个数。
第二行包含n个给定的非负整数，每个整数不超过10000。
【输出】
这n个非负整数中有多少对整数的值正好相差1。

样例输入	样例输出
6 10 2 6 3 7 8	3

【样例说明】相差为1的整数对包括(2,3), (6,7), (7,8)

- 第17行可以修改为**ans += cnt[i]&&cnt[i-1]**，同样表示两个相邻元素都为1时计数加1。也可以修改为**ans += cnt[i]&cnt[i-1]**，只有两个相邻元素都为1时，“位与”运算的结果才能为1。位运算是底层运算，计算效率最高。

```
1  #include<iostream>
2  using namespace std;
3
4  int main (  )
5  {
6      int n;
7      cin>>n;
8      bool cnt[10010]= {0};                //给定一些冗余空间，防止边界错误
9      int x,min=10000,max=0,ans;
10     for(int i=0; i<n; ++i){
11         cin>>x;
12         if(x<min)   min = x;             //求最小值
13         if(x>max)   max = x;             //求最大值
14         cnt[x] = 1;                      //对应x的位置有数值
15     }
16     for(int i=min+1;i<=max;++i)          //遍历所有有效数值
17         ans += (cnt[i]+cnt[i-1]==2);     //如果相邻两个元素都有效，则相加必为2
18     cout<<ans<<endl;
19     return 0;
20 }
```

随堂练习5.3

有N个非零且各不相同的整数。请你编一个程序求出它们中有多少对相反数(a和-a为一对相反数)。

【输入】

第一行包含一个正整数N (1≤N≤500)。

第二行为N个用单个空格隔开的非零整数，每个数的绝对值不超过1000，保证这些整数各不相同。

【输出】

只输出一个整数， 即这N个数中包含多少对相反数。

样例输入	样例输出
5 1 2 3 -1 -2	2

例题5.9

给定n个整数，求第i~j之间所有数据的和。

【输入】
第一行包含一个正整数N (1≤N≤10000)。
第二行为N个用单个空格隔开的整数，每个数小于10^5。
从第三行开始，每行输入两个整数i和j，1≤i≤j≤10000。
【输出】
输出从输入第三行开始每行指定范围的所有整数的和。

样例输入	样例输出
6 10 2 6 3 7 8 2 4 3 6	11 24

本题从表面上看是一个简单的数据求和问题，但是求和范围可能有重叠，重叠部分如果范围较大，次数较多时，就会造成严重的时间浪费，从而出现超时问题。采用打表法进行解决。

```
1  #include<iostream>
2  using namespace std;
3
4  int main (  )
5  {
6      int n,x;
7      cin>>n;
8      int table[10010]= {0};
9      for(int i=0; i<n; ++i){
10         cin>>x;
11         table[i+1] = table[i]+x;
12     }
13     int i,j;
14     while(cin>>i>>j)
15         cout<<table[j]-table[i-1]<<endl;
16     return 0;
17 }
```

- 第8行建立了一个数组，保存在第9~12行中输入数据的累积和。
- 第15行中，对应范围的两个累积和相减，得到这个范围内所有数据的和。这种方法的**最大优势**在于**对于重复的范围只计算了一次，避免了时间的反复消耗**。

例题5.10

计算小于给定非负整数n的所有素数的个数。0 ≤ n ≤ 5 * 106，当n为0或1时，对应结果为0。

样例输入	样例输出
10	4

【样例说明】

小于10的所有素数为2,3,5,7。

如果对范围内的每个数据进行素数判断，会造成极大的时间浪费。**利用打表法，从2开始向后遍历，将所有数据的倍数标记为非素数**，这样统计起来非常简单。

```
1  #include<iostream>
2  using namespace std;
3
4  int main (  )
5  {
6      int n;
7      cin>>n;
8      int res = 0;
9      bool prime[n+10];
10     for(auto &e:prime)  e=true;
11     for (int i = 2; i < n; ++i){
12         if (prime[i]) {
13             ++res;
14             for (int j = i+i; j < n; j+=i)
15                 prime[j] = false;
16         }
17     }
18     cout<<res<<endl;
19     return 0;
20 }
```

- 第9行建立了一个数组保存所有可能的候选答案，通过第10行全部初始化为true。
- 第11-17行从小到大遍历所有可能的候选答案，**将素数i的所有倍数全部标记为false**。剩余的就全部为素数。

虽然从表面上看是一个嵌套循环，时间复杂度应该为$O(n^2)$，但是仔细分析，**第14行的循环是跳跃的，整个程序执行完毕后，数组中的每个元素只被第14行的循环访问一次**。

知识点

索引	要点	正链	反链
T526	打表法是数组应用的最重要方法之一，需要重点掌握。 空间消耗不能过大，一般在10^6以内，如果题目中没有缩减范围则不能用打表法。 尽量减少数组遍历的范围。	T525	T833,T871,T872,T873
	洛谷：U271211(LX509), U271212(LX510)		

2.6 排序

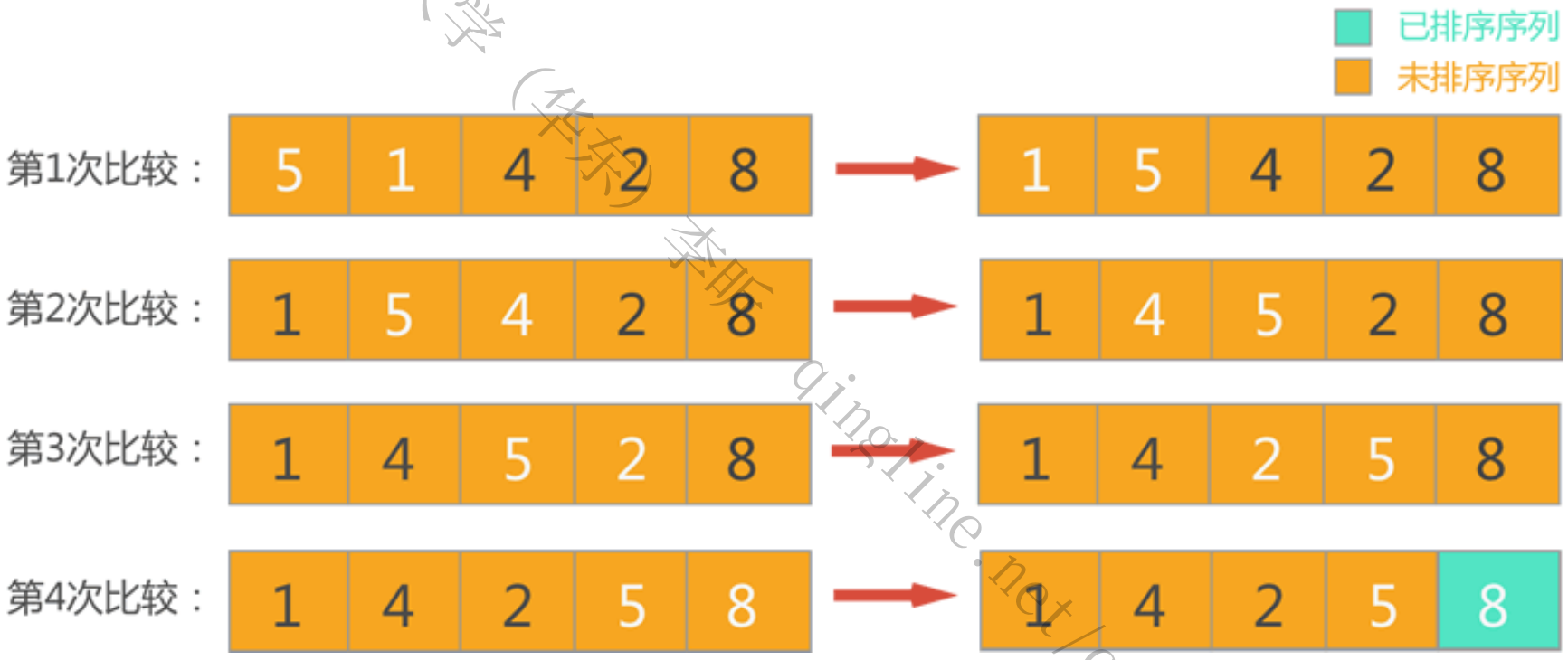
排序是多数值运算中的基本操作，一般分为升序和降序。排序的方法有很多，经典排序方法包括冒泡法、选择法、插入法和归并排序法等。这些算法的动画演示可以参见网站: https://visualgo.net/zh/sorting。

ps: visualgo是一个非常好的算法动画演示平台，很多常用算法都在该网站有动画形式展现。

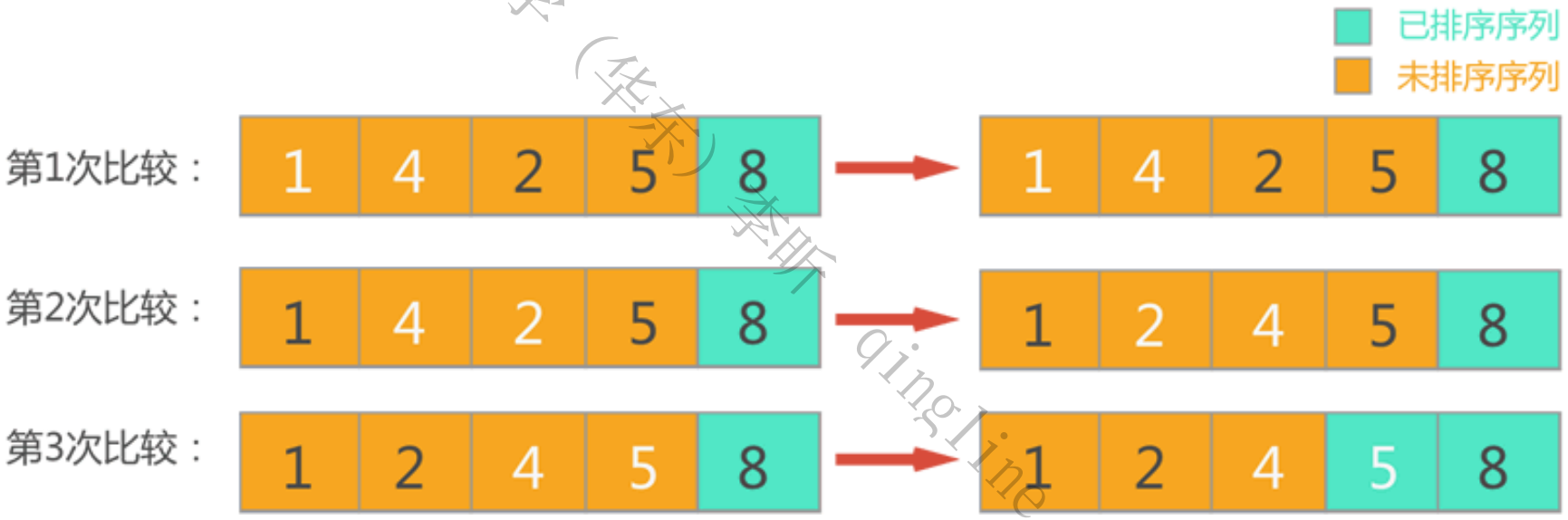
2.6.1 冒泡排序

冒泡排序是一种简单的排序算法，也是一种稳定排序算法。**重复遍历要排序的元素，依次比较两个相邻的元素，如果顺序错误就进行交换，直到没有相邻元素需要交换，完成排序**。该算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端（升序或降序排列），就如同碳酸饮料中二氧化碳的气泡最终会上浮到顶端一样，故名“冒泡排序”。

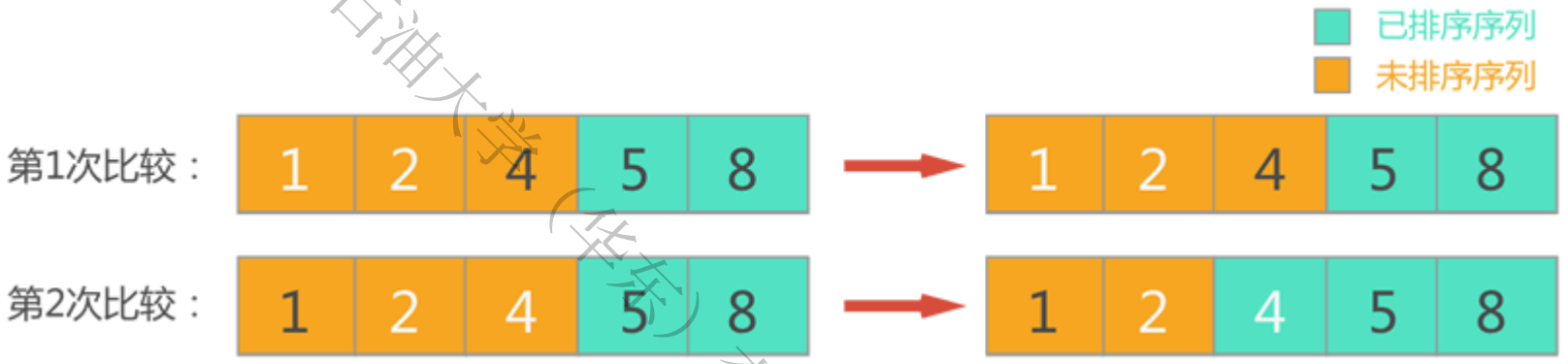
假设对待排序序列(5,1,4,2,8)进行升序排列，冒泡排序第一轮排序将最大元素置于最后：



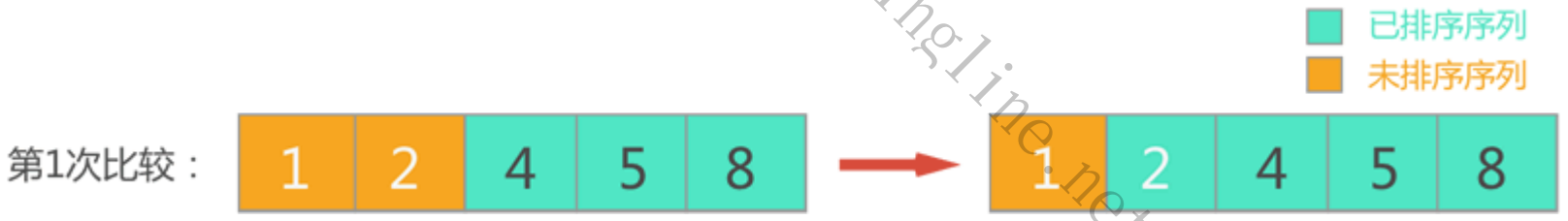
第二轮待排序序列只包含前 4 个元素，将其中最大元素放置在待排序序列尾部：



第三轮待排序序列只包含前 3个元素，将其中最大元素放置在待排序序列尾部：



第四轮只剩下2个元素，对其进行顺序调整，完成排序。



对于n个元素的排序，需要比较n-1轮，对于第i轮排序，比较n-i次。该算法**时间复杂度为$O(n^2)$**。冒泡排序的具体代码如下：

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

void bubble_sort(int n,int a[]) {
    for (int i = 0; i < n; i++) {
        //对待排序序列进行冒泡排序
        for (int j = 0; j + 1 < n - i; j++) {
            //相邻元素进行比较，当顺序不正确时，交换位置
            if (a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
        cout<<"第"<<i+1<<"轮排序：";
        for(int j = 0; j< n; ++j)
            cout<<a[j]<<' ';
        cout<<endl;
    }
}

int main(){
    int a[] = { 5,1,4,2,8 };
    bubble_sort(sizeof(a)/sizeof(int),a);
    return 0;
}
```

2.6.2 其他典型排序方法

选择法排序：与冒泡法基本流程相同，但是每次比较的时候不进行交换，只是记录最优值的下标，每轮交换一次，将一个极值放到有序位置。交换的次数比冒泡法少。

插入法排序：每次取出一个元素，放到已有有序数组的对应位置上。类似打牌时一边摸牌，一边排序。

快速排序：每轮随机选取一个元素作为基准，将所有元素分为比基准大和小两组，分别放到基准的左侧和右侧（降序），然后分别对两组采用相同的方法进行处理。这样排序轮次降低为$log_2 n$。C++中默认提供的sort函数就是采用快速排序，使用如下：

样例输入	样例输出
4 5 1 7 6	1 5 6 7 7 6 5 1

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

bool cmp(int a,int b){
    return a>b;
}

int main()
{
    int n;
    cin >> n;
    int a[n+3];
    for (int i=0;i<n;++i){
        cin >> a[i];
    }
    sort(a,a+n);                    //升序
    //reverse(a,a+n);               //逆序
    for (int i=0;i<n;++i){
        cout << a[i] << " ";
    }
    cout<<endl;
    sort(a,a+n,cmp);                //按照cmp函数指定的规则进行排序，此处为降序
    for (int i=0;i<n;++i){
        cout << a[i] << " ";
    }
    return 0;
}
```

- 第5-7行，自定义的比较函数cmp，返回值为布尔类型，两个参数的数据类型与数组中元素的数据类型相同。
- 函数cmp中，a和b代表数组中的两个元素。函数体中定义两个元素的比较规则。例如第6行，当a>b时返回true，因此第23行形成降序。
- sort**默认按照升序排列**，如果需要降序，将排列好的数组调用reverse进行逆序即可。

一般情况下，称某个**排序算法稳定**，指的是**当待排序序列中有相同的元素时，它们的相对位置在排序前后不会发生改变**。在NOI竞赛中，经常考察算法的稳定性，下表列出常用排序算法的时间复杂度和稳定性。

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r 代表关键字的基数，d 代表长度，n 代表关键字的个数						

知识点

索引	要点	正链	反链
T527	排序是基本算法，理解冒泡法、选择法、插入法和快速排序的基本思想和时间效率；能使用algorithm库中的sort函数对数组进行快速排序，能自定义比较规则。	T312	
	洛谷：U271214(LX511)		

2.7 动态规划

动态规划（Dynamic Programming, DP）是求解多阶段决策问题最优化的一种算法技术。为了解决复杂问题，它**将大问题分解为相对简单的子问题，大问题的最优解取决于子问题的最优解**。

如果一个问题，能够把所有可能的答案穷举出来，并且穷举出来后，发现存在重叠子问题，就可以考虑使用动态规划。重叠子问题是指求解大问题时需要多次重复求解小问题，它曾在例题5.7中被提及。

以例题5.7为例，讲解使用**动态规划解题的步骤**。

第一步：穷举分析

假设爬到第n级台阶共有f(n)种爬法。

当台阶数n为1时，只有一种爬法，f(1) = 1。

当台阶数n为2时，有两种爬法。第一种是直接爬两级，第二种是先爬一级然后再爬一级。f(2) = 2。

当台阶数n为3时，要么是先爬到第二级台阶然后再爬一级，要么是先爬到第一级台阶然后再直接爬两级。因此f(3) = f(2) + f(1) = 3。

当台阶数n为4时，要么是先爬到第三级台阶然后再爬一级，要么是先爬到第二级台阶然后再直接爬两级。因此f(4) = f(3) + f(2) = 5。

以此类推。

第二步：确定边界

通过穷举分析，发现当台阶数n是1或2时，能够直接求得有多少种爬法，即f(1) = 1, f(2) = 2。当台阶数$n \geq 3$时，已经呈现出规律：f(n) = f(n-1) + f(n-2)。因此 f(1) = 1, f(2) = 2 就是爬楼梯问题的边界。

第三步：找规律，确定最优子结构

最优子结构是指大问题的最优解可以由其子问题的最优解有效地构造出来。当台阶数$n \geq 3$时，有f(n) = f(n-1) + f(n-2)，因此 f(n-1) 和 f(n-2) 就是 f(n) 的最优子结构。

第四步：写出状态转移方程

通过前面三个步骤，得到状态转移方程如下：

$$f(n) = \begin{cases} 1 & n = 1 \\ 2 & n = 2 \\ f(n-1) + f(n-2) & n \geq 3 \end{cases}$$

使用**动态规划解题的模板**如下，以例题5.7为例：

```cpp
// 动态规划解题的模板
// dp[0][0][...] = 边界值;
// for(状态1: 所有状态1的值){
//     for(状态2: 所有状态2的值){
//         for(...){
//          状态转移方程;
//         }
//     }
// }
#include<iostream>
using namespace std;
int main ()
{
    int n;
    cin>>n;
    int a[n] = {1,2};                   //设定边界值
    for(int i=2;i<n;i++)                //遍历所有状态1的值
        a[i] = a[i-1]+a[i-2];           //转移方程
    cout<<a[n-1];
    return 0;
}
```

例题5.11

给你一个整数数组 nums ，找到其中最长严格递增子序列(LIS)的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

【输入】

第一行，整数数组nums的元素个数。

第二行，整数数组nums，整数之间用空格分隔。

其中，1 <= nums.length <= 2500

-10^4 <= nums[i] <= 10^4

【输出】

最长严格递增子序列的长度。

样例输入	样例输出
8 10 9 2 5 3 7 101 18	4

【题目分析】

第一步：穷举分析

以样例输入中的数组[10, 9, 2, 5, 3, 7, 101, 18]为例，进行穷举分析：

当nums只有一个元素10时，最长严格递增子序列是{10}，长度是1。

当nums加入一个元素9时，最长严格递增子序列是{10}或{9}，长度是1。

当nums再加入一个元素2时，最长严格递增子序列是{10}或{9}或{2}，长度是1。

当nums再加入一个元素5时，最长严格递增子序列是{2, 5}，长度是2。

当nums再加入一个元素3时，最长严格递增子序列是{2, 5}或{2, 3}，长度是2。

当nums再加入一个元素7时，最长严格递增子序列是{2, 5, 7}或{2, 3, 7}，长度是3。

当nums再加入一个元素101时，最长严格递增子序列是{2, 5, 7, 101}或{2, 3, 7, 101}，长度是4。

当nums再加入一个元素18时，最长严格递增子序列是{2, 5, 7, 101}或{2, 3, 7, 101}或{2, 5, 7, 18}或{2, 3, 7, 18}，长度是4。

第一步：穷举分析

分析以上过程可得，在以数组每个元素结尾的最长严格递增子序列组成的集合中，元素最多的即为数组的最长严格递增子序列。因此**原问题转化成先求出以每个元素结尾的最长严格递增子序列集合，再求最大长度**。创建一个整型数组 dp，用 dp[i] 表示以 nums[i] 结尾的最长严格递增子序列的长度，得到下表。

下标 i	0	1	2	3	4	5	6	7
nums[i]	10	9	2	5	3	7	101	18
dp[i]	1	1	1	2	2	3	4	4

事实上，**只要在前面找到 nums[j]<nums[i]，以 nums[j] 结尾的严格递增子序列加上 nums[i] 即可得到以 nums[i] 结尾的严格递增子序列**。显然，可能形成多种新的子序列，选择最长的子序列，即为以 nums[i] 结尾的最长严格递增子序列。

第二步：确定边界

对于某个数组，dp[0] = 1，dp[1] = 2或1， 因此边界就是 dp[0] =1。

第三步：找规律，确定最优子结构

根据穷举分析，发现如下规律：

对于$j < i$并且$nums[j] < nums[i]$，有$dp[i] = max(dp[j]) + 1$

$max(dp[j])$就是最优子结构。

第四步：写出状态转移方程

通过前面三个步骤，得到状态转移方程如下：

$$dp[i] = \begin{cases} 1 & i = 0 \\ max(dp[j]) + 1 & 0 \le j < i \text{ 且 } nums[j] < nums[i] \end{cases}$$

因此数组nums的最长严格递增子序列的长度为 max(dp[i])。

求解最长严格递增子序列长度的代码如下：

```
1   # include<iostream>
2   using namespace std;
3
4   int lengthOfLIS(int array[], int length){
5           if(length == 0) return 0;
6           int dp[length];
7           int ans = 1;
8           for(int i = 0; i < length; ++i){
9               dp[i] = 1;
10              for(int j = 0; j < i; ++j)
11                  if(array[j] < array[i])
12                      dp[i] = max(dp[i], dp[j] + 1);
13              ans = max(ans, dp[i]);
14          }
15          return ans;
16  }
17
18  int main(){
19      int length;
20      cin>>length;
21      int nums[length];
22      for(int i = 0; i < length; ++i)
23          cin>>nums[i];
24      cout<<lengthOfLIS(nums, length);
25      return 0;
26  }
```

- 第4行，数组作为函数的参数时，必须同时传递数组地址和数组中元素的数量，否则无法知道数组的有效范围。详细内容见本章1.5小节。
- 第11-12行，只有 array[i] 大于 array[j]，才能将 array[i] 放在 array[j] 后面以形成更长的严格递增子序列。
- 第12行用于计算 max(dp[j]+1)，从而确定最终的dp[i]，这与 max(dp[j])+1 是等价的。

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

int lengthOfLIS(int array[], int length){
        if(length == 0) return 0;
        int dp[length];
        int ans = 1;
        for(int i = 0; i < length; ++i){
            dp[i] = 1;
            for(int j = 0; j < i; ++j)
                if(array[j] < array[i])
                    dp[i] = max(dp[i], dp[j] + 1);
            ans = max(ans, dp[i]);
        }
        return ans;
}

int main(){
    int length;
    cin>>length;
    int nums[length];
    for(int i = 0; i < length; ++i)
        cin>>nums[i];
    cout<<lengthOfLIS(nums, length);
    return 0;
}
```

- 第13行用于计算所有dp[i]中的最大值。

- 分析lengthOfLIS函数可知，该函数先解决子问题再递推到大问题，具体过程为**使用多个for循环填写数组，根据已计算的结果，逐步递推出大问题的解决方案**。这便是动态规划的解题思路。

知识点

索引	要点	正链	反链
T528	动态规划将大问题分解为子问题，求解时从子问题递推到大问题	T513, T525	

03 二维数组

多维数组声明的形式

C++ 支持多维数组。多维数组声明的一般形式如下：

type arrayName[size1][size2]...[sizeN];

type 可以是任意有效的 C++ 数据类型，arrayName 是一个有效的 C++ 标识符。

e.g. 创建一个三维 5*10*4 整型数组

int threedim[5][10][4];

一个二维数组可以被认为是一个带有 x 行和 y 列的表格。下面是一个二维数组，包含 3 行和 4 列：

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

多维数组的初始化

多维数组可以通过在括号内为每行指定值来进行初始化。

```
int a[3][4] = {{0, 1, 2, 3} ,   /*  初始化索引号为 0 的行 */
               {4, 5, 6, 7} ,   /*  初始化索引号为 1 的行 */
               {8, 9, 10, 11}   /*  初始化索引号为 2 的行 */ };
```

内部嵌套的括号是可选的，下面的初始化与上面是等同的：

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

多维数组如果被进行初始化，或作为函数参数时，紧挨变量名的第一个维度的长度可以省略，其余的元素数量必须明确定义。否则编译器无法获知每个维度的长度。

二维数组遍历

编程时，无论是输入输出还是遍历，多维数组通常是和嵌套循环搭配使用的。

样例输出

```
行\列   0   1   2   3   4
0       3   5   7   6   8
1       1   8   2   5   3
```

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

int main ()
{
    int a[2][5] = { {3,5,7,6,8}, {1,8,2,5,3}};
    cout<<"行\\列";
    for ( int j = 0; j < 5; j++ )
        cout<<'\t'<<j;
    cout<<endl;
    for ( int i = 0; i < 2; i++ ){
        cout<<i;
        for ( int j = 0; j < 5; j++ ){
            cout <<'\t'<< a[i][j];
        }
        cout<<endl;
    }
    return 0;
}
```

例题5.12

旋转是图像处理的基本操作，在这个问题中，你需要将一个图像逆时针旋转90°。计算机中的图像可以用一个矩阵来表示。为了旋转一个图像，只需要旋转对应的矩阵即可。

【输入】

输入的第一行包含两个整数 n、m,分别表示图像矩阵的行数和列数。

接下来n行，每行包含m个整数，表示输入的图像。

1≤n,m≤1000， 矩阵中的数都是不超过1000的非负整数。

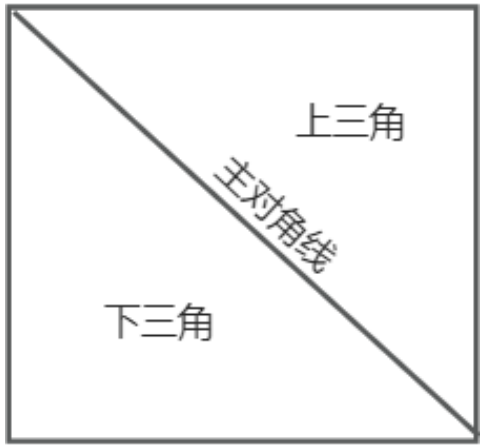
【输出】

输出m行，每行包含n个整数，表示原始矩阵逆时针旋转90° 后的矩阵。

样例输入	样例输出
2 3 1 5 3 3 2 4	3 4 5 2 1 3

```cpp
#include <iostream>
using namespace std;
int a[1005][1005];
int main ()
{
    int n,m;
    cin>>n>>m;
    for ( int i = 0; i < n; i++ ){
        for ( int j = 0; j < m; j++ ){
            cin>>a[i][j];
        }
    }
    for ( int j = m-1; j >=0; j-- ){
        for ( int i = 0; i < n; i++ ){
            cout<<a[i][j]<<' ';
        }
        cout<<endl;
    }
    return 0;
}
```

在二维数组中，最重要的是对角线、上三角和下三角的概念。对角线上行列坐标相等，**上三角中的行坐标小于列坐标，下三角中列坐标小于行坐标**。




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

int main ()
{
    int n;
    cin>>n;
    int a[n][n];
    for ( int i = 0; i < n; i++ ){
        for ( int j = 0; j < n; j++ ){
            a[i][j]=abs(i-j);
            cout<<a[i][j]<<' ';
        }
        cout<<endl;
    }
    return 0;
}
```

样例输入	样例输出
5	0 1 2 3 4 1 0 1 2 3 2 1 0 1 2 3 2 1 0 1 4 3 2 1 0

知识点

索引	要点	正链	反链
T531	掌握二维数组的基本使用方法，掌握主对角线、上三角、下三角的概念。	T511	T623

04 C++的字符串

4.1 字符串的基本操作

字符串是常见的基本类型，在C++中提供了string类型进行字符串处理。可以将字符串理解成一个数组，其中每个元素都是一个字符，**可以按照数组的形式进行访问**。

样例输出

```
1 2 3 4 5 6 7 8 9 0
12345a7890
```

```cpp
#include <iostream>
#include <string>
using namespace std;
int main(){
    string s = "1234567890";
    for(int i=0,len=s.length(); i<len; i++){//采用数组形式进行遍历
        cout<<s[i]<<" ";
    }
    cout<<endl;
    s[5] = 'a';          //修改字符串中某个字符
    cout<<s<<endl;
    return 0;
}
```

字符串在C语言中用'\0'结尾的字符数组进行表达，操作和理解上都比较复杂，按照数组的基本规定，不能进行整体赋值，不能进行整体比较。

C++将字符数组封装成了一个string类，通过成员函数和操作符重载等一系列面向对象的处理，string类在处理上变得非常简单。例如将两个字符串进行拼接时，使用+即可。此外string还提供了增删改查等基本操作。

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5       cout<<"字符串拼接："<<endl;
6       string s1 = "first ";
7       string s2 = "second ";
8       cout<<s1+s2<<endl<<s1+"third"<<endl;
9       cout<<"插入子字符串："<<endl;
10      s1.insert(3,"aaa");
11      cout<<s1<<endl;
12      cout<<"删除子字符串："<<endl;
13      s1.erase(3);
14      s2.erase(3,2);
15      cout<<s1<<endl<<s2<<endl;
16      cout<<"抽取子字符串："<<endl;
17      s1 = "first third second";
18      s2 = s1.substr(6, 5);
19      cout<<s2<<endl;
```

- insert() 函数**在 string字符串中指定的位置插入另一个字符串**，第一个参数表示要插入的位置，也就是下标；第二个参数表示要插入的字符串。第10行在下标为3的位置插入了子串"aaa"。
- erase() 函数**删除 string中的一个子字符串**。第一个参数表示要删除子字符串的起始下标，第二个参数表示要删除子字符串的长度。如果没有第二个参数，直接删除到字符串结束处的所有字符。
- substr() 函数**从 string字符串中提取子字符串**。第一个参数为要提取的子字符串的起始下标，第二个参数为要提取的子字符串的长度。

```
20      cout<<"字符串查找："<<endl;
21      s1 = "first second third second";
22      s2 = "second";
23      size_t index = s1.find(s2,7);
24      if(index < s1.length())
25          cout<<"Found at index : "<< index <<endl;
26      else
27          cout<<"Not found"<<endl;
28      index = s1.rfind(s2,7);
29      if(index < s1.length())
30          cout<<"Found at index : "<< index <<endl;
31      else
32          cout<<"Not found"<<endl;
33      cout<<"查找子字符串任意字符在字符串中首次出现的位置："<<endl;
34      index = s1.find_first_of(s2);
35      if(index < s1.length())
36          cout<<"Found at index : "<< index <<endl;
37      else
38          cout<<"Not found"<<endl;
39      return 0;
40  }
```

- find() 函数**在 string字符串中查找子字符串出现的位置**。第一个参数为待查找的子字符串，第二个参数为开始查找的位置（下标）；如果不指明，则从第0个字符开始查找。

- find()函数**返回子字符串第一次出现在字符串中的起始下标**。如果没有查找到子字符串，返回一个无穷大值。第23行的查找从下标7开始，因此找到的是第二个second出现的位置19。

- 在string的众多函数中，涉及位置、长度信息时，使用的数据类型都是size_t，在64位系统中size_t定义为long long unsigned int，在32位系统中定义为unsigned int。

```
20      cout<<"字符串查找："<<endl;
21      s1 = "first second third second";
22      s2 = "second";
23      size_t index = s1.find(s2,7);
24      if(index < s1.length())
25          cout<<"Found at index : "<< index <<endl;
26      else
27          cout<<"Not found"<<endl;
28      index = s1.rfind(s2,7);
29      if(index < s1.length())
30          cout<<"Found at index : "<< index <<endl;
31      else
32          cout<<"Not found"<<endl;
33      cout<<"查找子字符串任意字符在字符串中首次出现的位置："<<endl;
34      index = s1.find_first_of(s2);
35      if(index < s1.length())
36          cout<<"Found at index : "<< index <<endl;
37      else
38          cout<<"Not found"<<endl;
39      return 0;
40  }
```

- rfind() 函数与find()类似，但是它**最多查找到第二个参数处**，如果到了第二个参数所指定的下标还没有找到子字符串，则返回一个无穷大值。

- find_first_of()函数不把子串作为一个整体，而是**查找子串中任意字符在字符串中第一次出现的位置**。第34行进行查找时，'s'出现在second中，并且它在s1中的下标为3，因此返回结果为3，并不是second的6。

上述代码的输出如下：

样例输出

```
字符串拼接：
first second
first third
插入子字符串：
firaaast
删除子字符串：
fir
secd
抽取子字符串：
third
字符串查找：
Found at index : 19
Found at index : 6
查找子字符串任意字符在字符串中首次出现的位置：
Found at index : 3
```

知识点

索引	要点	正链	反链
T541	掌握string字符串增删改查的基本操作和对应的函数。	T243,T511	T832,T842
	洛谷：U271216(LX513), U271219(LX514)		

4.2 字符串的容量和长度*

字符串的长度和存储容量是两个概念，二者并不一致。**长度指字符串中实际有效的字符数量，而存储容量是指对该字符串分配的存储空间大小。**长度可以用size()或length()函数获取，两个函数是完全等价的；存储容量用capacity()获取。

容量和长度不相等的原因

字符串本质上是一个字符数组，数组中所有元素在内存中必须连续分配。当字符串长度发生变化，已有空间不能满足需求时，需要重新分配空间，而为了保证空间的连续性，只能进行全部存储空间的重新分配。

字符串作为常用数据类型，其基本操作增删改都会涉及长度变化，如果每次都全部重新分配内存空间，运行效率将非常低。因此**通常字符串的容量都要大于有效字符的长度，当字符串长度进行小范围变化时，不需要重新分配空间**。而**当出现空间不足的情况时，重新分配的新空间还是会有一定的冗余度**。至于新空间具体是多少，是由操作系统提供的策略进行保障的，用户不需要过多了解。

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

int main(){
    string s;
    cout<<s.empty()<<endl;
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s="abcdefghijklmnopqrstuvwxyz";
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s="abcdefghijklmnopqrstuvwxyzabcdefg";
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.resize(20);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.resize(70);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.reserve(80);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.reserve(50);//或s.reserve()
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    return 0;
}
```

- empty**判断一个字符串是否为空**，其效率比size()函数高。

- 第7行，空字符串的长度为0，其容量并非为0。当字符数量小于15时，不需要为该字符串重新分配空间。

样例输出

```
1
0    15
26    30
33    40
20    60
70    120
70    80
70    70
```

- 通过**resize()函数调整字符串的长度(size)**，通过**reserve()函数调整字符串的容量(capacity)**。size调整时会导致capacity跟随发生变化，但是capacity调整时，size不会发生变化。

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

int main(){
    string s;
    cout<<s.empty()<<endl;
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s="abcdefghijklmnopqrstuvwxyz";
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s="abcdefghijklmnopqrstuvwxyzabcdefg";
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.resize(20);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.resize(70);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.reserve(80);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.reserve(50);//或s.reserve()
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    return 0;
}
```

样例输出

```
1
0    15
26   30
33   60
20   60
70   120
70   80
70   70
```

- 第8行因为新字符串长度超过15，size变为实际长度26，但是capacity=30>26，第10行再次打破capacity的限制变成33后，capacity被调整为60。第12行通过resize函数将size变小后，capacity不会发生变化，但第14行将size变大后超过了当前的capacity，capacity被再次扩容。可以确定，**capacity一直大于等于size，但具体的值是根据操作系统的策略进行动态调整的。**

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

int main(){
    string s;
    cout<<s.empty()<<endl;
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s="abcdefghijklmnopqrstuvwxyz";
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s="abcdefghijklmnopqrstuvwxyzabcdefg";
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.resize(20);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.resize(70);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.reserve(80);
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    s.reserve(50);//或s.reserve()
    cout<<s.size()<<'\t'<<s.capacity()<<endl;
    return 0;
}
```

样例输出

```
1
0    15
26    30
33    40
20    60
70    120
70    80
70    70
```

- string可以调用reserve()缩减实际容量。但**用一个“小于现有容量”的参数调用reserve()，是一种非强制性请求**。也就是说可能想要缩减容量至某个目标，但不保证一定达成。string的reserve()参数默认值为0，所以调用reserve()并且不给参数，就是一种“非强制性适度缩减请求”。第18行缩放目标小于size的值，但新的capacity变为70，并没有按照目标指示变成50或0。

例题5.13

完成函数string str_remove(string s, char ch)，从s中删除指定的字符ch，并将剩余字符串作为返回值。

样例输入	样例输出
s="abbce", ch='b'	ace

```
1  string str_remove(string s, char ch)
2  {
3      size_t j=0;
4      for(size_t i=0; i<s.size(); i++)
5          if(s[i]!=ch)
6              s[j++]=s[i];
7      s.resize(j);
8      return s;
9  }
```

string在本质上是由字符组成的数组。不能做物理删除，只能形成逻辑删除。上方代码借助了**复制形成新字符串**的想法，将留存的字符形成新的字符串。因为删除后的字符串长度一定小于或等于原字符串，即i>=j，因此可以在原字符串的空间上完成复制操作。j只有在找到符合要求的字符后，才执行加1操作。这种想法的算法复杂度为$O(n)$，并且不需要开辟新的空间。

例题5.13

完成函数string str_remove(string s, char ch)，从s中删除指定的字符ch，并将剩余字符串作为返回值。

```
1  string str_remove(string s, char ch)
2  {
3      size_t j=0;
4      for(size_t i=0; i<s.size(); i++)
5          if(s[i]!=ch)
6              s[j++]=s[i];
7      s.resize(j);
8      return s;
9  }
```

样例输入	样例输出
s="abbce", ch='b'	ace

- 第3-4行中i和j的数据类型定义为size_t，**无论是size()还是length()返回的数据类型都是size_t**，保证数据类型的兼容性。size_t依赖于编译器，在32位编译器下等同于unsigned int，在64位编译器下等同于unsigned long long，其值永远非负。

- 第7行中用resize()重新调整s的长度，否则删除处理后s的长度保持不变。

知识点

索引	要点	正链	反链
T542	明确区分字符串的物理空间和逻辑上的有效空间，新建立的字符串一定要重新resize，否则逻辑空间长度不正确。	T513	T825

4.3 字符串与整型的相互转换

4.3.1 字符串转数值

cin可以从标准输入流中读取整型或其他类型的数据，因此可以将字符串首先转换为流数据，然后从流中读取相应类型的数据，这是转换的第一种方法；此外，C语言的库函数中存在字符串转换其他标准数据类型的函数，例如：字符串转整型atoi，字符串转浮点数atof等，但C语言中不存在string类型，需要先调用string的c_str()函数将string转换为C风格字符串，才能得到正确的结果。

```cpp
#include<iostream>
#include<cstdlib>      //atoi,atof
#include<sstream>       //istringstream
using namespace std;

int main()
{
    string a="1234";
    //使用字符串流将字符串转换为整型
    istringstream is(a); //构造输入字符串流，流的内容初始化为“12”的字符串
    int i;
    is >> i; //从is流中读入一个int整数存入i中
    cout<<i+1<<endl;  //i已经是整型，可以进行数学运算
    //atoi和atof的使用方式
    cout<<atoi(a.c_str())-1<<endl;     //注意一定要使用c_str函数将string转换为C风格字符串
    cout<<atof("1212.34")+1<<endl;     //双引号构成的字符串是C风格字符串
    return 0;
}
```

样例输出

```
1235
1233
1213.34
```

atoi和atof是C语言函数，只支持C语言风格字符串，即以\0结尾的字符数组。**如果用到string类型，必须通过c_str函数转换为C风格字符串。**

知识点

索引	要点	正链	反链
T543	掌握字符串转数值的常用方法。istringstream虽然使用上比较繁琐，但是好用。	T484	
T544	了解string与C风格字符串不同，以及通过c_str完成转换。		

4.3.2 数值转字符串

C++中的std提供了标准的函数**to_string()可以将基础数据类型转换为字符串**，不需要任何头文件。函数to_string()可以满足绝大多数情况下的转换需求。

如果有精度和宽度限制，或其他特殊需求时，处理比较繁琐。这时可以**使用sprintf函数将需要的内容转换为C风格字符串，然后再将C风格字符串转换为string类型**。sprintf函数与函数printf的使用方法几乎完全类似，只是printf将结果打印到标准输出中，而sprintf将结果打印到一个C风格的字符串中。

```cpp
#include<iostream>
#include<sstream>       //ostringstream的头文件
using namespace std;

int main()
{
    //to_string方式
    cout<<to_string(1234)<<endl;
    cout<<to_string(1234.56)<<endl;
    //sprintf方式转换为C风格字符串
    char str[1000];     //C风格的字符数组用来存储C风格的字符串
    sprintf(str,"%.2f",1.2345);
    //C风格字符串转换为string类型
    string s=str;       //初始化
    string s1(str);     //构造函数
    string s2 {str};    //初始化
    cout<<s<<endl;
    cout<<s1<<endl;
    cout<<s2<<endl;
    ostringstream oss;
    oss<<3.14;
    oss<<" ";
    oss<<55555555;
    cout<<oss.str();
    return 0;
}
```

样例输出

```
1234
1234.560000
1.23
1.23
1.23
3.14
55555555
```

知识点

索引	要点	正链	反链
T545	掌握数值转字符串的基本方法，最重要的方法是to_string。 ostringstream是C++中拼接字符串的重要方法。		T546

4.4 字符串的分割

字符串分割是常见操作，在Python，Java等语言中，可以通过简单的调用split完成分割，但是C/C++中不存在这样的函数。C语言中可以采用“循环+strtok”的方式完成分割，但代码比较繁琐。本节提供两种基于字符串数据流的方法，简单有效地完成分割操作。具体的代码如下：

```cpp
#include <iostream>
#include <sstream>
using namespace std;
int main()
{
    string str = "good good study day day up";
    istringstream in(str);                          //将字符串转换为数据流
// 方法1：借助数据流以空白符分割的特性，形成字符串的分割
    string s;
    while (in >> s){
        cout<<s<<' ';
    }
    cout<<endl;

    str = "good,good,study,day,day,up";
    istringstream in2(str);                         //将字符串转换为数据流
// 方法2：利用自定义分割符的getline函数，达到采用任意字符分割的效果
    while (getline(in2, s, ',')){    //这里单引号要注意，第3个参数可以是任意字符
        cout<<s<<' ';
    }
}
```

- 如果分割符为空白符，采用方法1更加简单；如果分割符为任意字符，只能采用方法2。

- 采用ostringstream对字符串进行拼接，采用istringstream对字符串进行分割。

知识点

索引	要点	正链	反链
T546	掌握用数据流对字符串进行分割的方法。	T271,T484, T545	T547
	洛谷：U271219(LX514), U271222(LX515)		

4.5 子串问题

例题5.14

昕哥有一串很长的英文字母，可能包含大写和小写。在这串字母中，有很多连续的是重复的。昕哥想了一个办法将这串字母表达得更短：将连续的几个相同字母写成字母+出现次数的形式。例如，连续的5个a，即 aaaaa，简写成 a5。对于这个例子：aaaaaCCeeelHH，昕哥可以简写成 a5C2e3lH2。为了方便表达，昕哥不会将连续的超过9个相同的字符写成简写的形式。请帮助昕哥完成简写。

【输入】输入一行为一个由大写字母和小写字符构成的字符串，长度不超过100000。

【输出】输出为一行字符串，表示简写后的字符串。

样例输入	样例输出
aaaaaCCeeelHH	a5C2e3lH2

```
1  #include <iostream>
2  using namespace std;
3  void print(const string& s){
4      if(s.size()==1||s.size()>9)
5          cout<<s;
6      else
7          cout<<s[0]<<s.size();
8  }
9  int main() {
10     string a;
11     cin>>a;
12     size_t left=0;
13     a += '$';
14     for(size_t j=1;j<a.size();j++){
15         if(a[j]^a[left]){                    //或写成if(a[j]!=a[i])
16             print(a.substr(left,j-left));
17             left = j;
18         }
19     }
20     return 0;
21 }
```

- 对于每个子串的缩写，是一个相对比较独立的过程。将其独立成一个函数print，能够有效降低程序的复杂度，强烈建议使用。
- 第15行用异或操作判断两个元素不相等，它与直接书写不等于是等价的。
- 用变量left记录每个子串的起点，然后用substr截取每个子串，分别送到print中进行缩写。这是子串拆分的经典方法。其本质上是采用尺取法的同向扫描，利用快慢指针形成字符完全相同的“移动窗口”，然后对移动窗口做出相应的简写处理。尺取法是处理子串分隔的一种基本方法。

4.5 子串问题

例题5.14

昕哥有一串很长的英文字母，可能包含大写和小写。在这串字母中，有很多连续的是重复的。昕哥想了一个办法将这串字母表达得更短：将连续的几个相同字母写成字母+出现次数的形式。例如，连续的5个a，即 aaaaa，简写成 a5。对于这个例子：aaaaaCCeeelHH，昕哥可以简写成 a5C2e3lH2。为了方便表达，昕哥不会将连续的超过9个相同的字符写成简写的形式。请帮助昕哥完成简写。

【输入】输入一行为一个由大写字母和小写字符构成的字符串，长度不超过100000。

【输出】输出为一行字符串，表示简写后的字符串。

样例输入	样例输出
aaaaaCCeeelHH	a5C2e3lH2

```
1  #include <iostream>
2  using namespace std;
3  void print(const string& s){
4      if(s.size()==1||s.size()>9)
5          cout<<s;
6      else
7          cout<<s[0]<<s.size();
8  }
9  int main() {
10     string a;
11     cin>>a;
12     size_t left=0;
13     a += '$';
14     for(size_t j=1;j<a.size();j++){
15         if(a[j]^a[left]){                    //或写成if(a[j]!=a[i])
16             print(a.substr(left,j-left));
17             left = j;
18         }
19     }
20     return 0;
21 }
```

- 特别注意第13行的操作。因为循环是对有效部分进行操作，当第14行循环结束时，最后一个子串并未得到相应的处理。因此在第13行中**插入一个原字符串中不可能出现的字符，保证原字符串中的最后一个子串被处理**，又对题目要求的结果没有任何影响。这种方法称为边界填充法。如果没有这行处理，必须在循环结束时对最后一个子串进行处理。

知识点

索引	要点	正链	反链
T547	掌握使用尺取法进行子串分隔，特别注意最后一个子串的处理，掌握边界填充法。	T524,T546	
	洛谷：U271223(LX516)		
T548	将相对独立的功能抽取为函数，将极大减轻程序书写逻辑。	T331	

4.6 高精度计算

例题5.15

输入两个不超过100位的位数相等的正整数，求其和。

样例输入	样例输出
6868979797088045643286344758256878457535883226687	116272366755455815265132l

【题目分析】C++中提供的最大整数类型为long long，为64位，大概表示10^{18}量级，显然无法表示题目所述的整数。因此需要**用字符串表达这两个100位的整数**，通过小学所学的逐位计算的方法计算加法。

```
1   #include <iostream>
2   using namespace std;
3   string plus(const string& s1,const string& s2){
4       string r=s1;
5       int carry = 0;
6       for(size_t i=r.size()-1;i<r.size();i--){
7           int temp = s1[i]-'0'+s2[i]-'0'+carry;
8           r[i] = temp%10+'0';
9           carry = temp/10;
10      }
11      return carry?"1"+r:r;
12  }
13  int main() {
14      string s1,s2;
15      cin>>s1>>s2;
16      cout<<plus(s1,s2)<<endl;
17      return 0;
18  }
```

- 模拟加法计算过程。第4行让结果r有一个初始长度，然后第6行的循环逐位执行加法操作。**注意**循环条件有点不合常规，这是因为size_t是unsigned类型，-1是该类型的最大值，一定大于字符串长度。
- 按照加法计算规则，每位的结果都是对应两位相加，并且加上进位。设置初始进位为0。
- 第11行是为了防止最高位有进位，根据加法规则，进位只能是1。

随堂练习5.4

如果两个大数相加，且位数不相等，应该如何处理？

知识点

索引	要点	正链	反链
T549	掌握大数计算的基本方法。	T242	
	洛谷：U271006(LX517)		

05 C风格的字符串

5.1 C风格字符串的定义和初始化

C语言并没有提供“字符串”这种复杂数据类型，它**借助字符数组来存储一个串的内容，以特殊字符'\0'作为串的结束标志**。存储结构上，C语言的字符串就是“以'\0'结尾的字符数组”，长度指串中位于'\0'之前的字符个数。

字符串一定是一个字符数组，但字符数组未必是字符串，关键看是否包含'\0'。一个字符数组可以存储多个'\0'，但它存储的字符串内容到第一个'\0'出现的位置就结束了。字符串的查找、求长度、复制、比较等常见算法都是紧紧围绕“以'\0'结尾”这一特性，对字符数组进行操作。

定义字符串类型的变量其实就是定义字符数组类型的变量，**必须保证数组大小足够容纳末尾的'\0'。**

```
1   #include <iostream>
2   using namespace std;
3   int main ()
4   {
5       char s1[10]; //数组c是一维数组，它可以存放10个字符，或者一个长度不大于9的字符串
6       char s2[6]="China";                                    //用字符串常量赋值
7       char s3[6]= {'C', 'h', 'i', 'n', 'a', '\0'};           //用字符常量赋值
8       char name[3][10];        //数组name是二维数组，存放3个长度不大于9的字符串
9       char w_day[ ][10]={"Sunday","Monday", "Tuesday",
10              "Wednesday","Thursday","Friday", "Saturday" };  //二维字符串初始化
11      int num;
12      cin>>num;
13      cout<<s2<<' '<<s3<<endl;
14      s2[3]='\0';
15      cout<<s2<<' '<<w_day[num]<<endl;
16  }
```

- 第6行和第9行用字符串常量进行初始化，会自动在尾部添加一个'\0'，表示字符串的结束。

- 第8-9行定义了两个二维字符串，用于存储多个字符串，每个字符串都要以'\0'结束。

- 第14行将字符串s2的第3个字符设置为'\0'，因此其逻辑长度修改为3，第15行只输出Chi三个字符。

样例输入	样例输出
3	China China Chi Wednesday

知识点

索引	要点	正链	反链
T551	掌握C风格字符串的定义和初始化，理解物理长度和逻辑长度。		

5.2 C风格字符串的基本操作

C风格字符串本质上是字符数组，因此不能整体赋值和整体比较。**对于字符串的常见操作，都存放在头文件<cstring>中**，求长度函数strlen，赋值函数strcpy，比较函数strcmp，连接函数为strcat。

```
1   #include<iostream>
2   #include<cstring>
3   using namespace std;
4   int main ()
5   {
6       char source[30] = "Why always me?",target[20];
7       cout<<strlen(source)<<' '<<sizeof(source)<<endl;
8       strcpy(target, source);
9       cout<<strlen(target)<<' '<<target<<endl;
10      char other[] = "Why not he?";
11      cout<<strcmp(source,other)<<endl;
12      cout<<strcat(source,other)<<endl;
13  }
```

样例输出

```
14 30
14 Why always me?
-1
Why always me?Why not he?
```

```
1   #include<iostream>
2   #include<cstring>
3   using namespace std;
4   int main ()
5   {
6       char source[30] = "Why always me?",target[20];
7       cout<<strlen(source)<<' '<<sizeof(source)<<endl;
8       strcpy(target, source);
9       cout<<strlen(target)<<' '<<target<<endl;
10      char other[] = "Why not he?";
11      cout<<strcmp(source,other)<<endl;
12      cout<<strcat(source,other)<<endl;
13  }
```

样例输出

```
14 30
14 Why always me?
-1
Why always me?Why not he?
```

- 第7行输出结果显示，source的逻辑长度为14，物理长度为30，strlen返回字符串的逻辑长度。

- 第8行将source的内容赋值给target，特别注意C风格字符串是数组，不能进行整体赋值，因此绝对不能写成target=source，只能采用strcpy对字符串进行赋值。

```cpp
#include<iostream>
#include<cstring>
using namespace std;
int main ()
{
    char source[30] = "Why always me?",target[20];
    cout<<strlen(source)<<' '<<sizeof(source)<<endl;
    strcpy(target, source);
    cout<<strlen(target)<<' '<<target<<endl;
    char other[] = "Why not he?";
    cout<<strcmp(source,other)<<endl;
    cout<<strcat(source,other)<<endl;
}
```

样例输出

```
14 30
14 Why always me?
-1
Why always me?Why not he?
```

- 第11行进行字符串的大小比较，采用结果为：0（相等），正数（第一个字符串大），负数（第二个字符串大）。**这里的大小指字典序，即单词在字典中的排列顺序**。从左到右依次遍历字符，当遇到第一个不相同的字符时，该字符的ASCII码比较结果就是两个字符串的大小关系。例如：strcmp("abcd","abck")<0，strcmp("abc","ab")>0，strcmp("abc","b")<0。

- 第12行对两个字符串进行了连接，并将连接的结果放在第一个字符串source中，函数的返回值也是连接的结果。

以下内容将C风格字符串中的常见操作进行了具体的代码实现。使用时可以采用库函数，以下实现是为了更好地从基础操作中理解C风格字符串的使用。

```
1  int my_strlen(char s[])
2  {
3      int len=0; //len计算串长
4      while (s[len]!='\0') len++; //统计'\0'前的字符数
5      return len;
6  }
```

- 根据C风格字符串的特点，求串长就是统计'\0'前的字符数

。

- 第4行用while循环实现，这使得代码的可读性更强；在熟练使用for循环后也可以写成“for (len = 0; s[len]!='\0' ;len++) ;”，注意不要遗漏了末尾的分号，这表示循环条件成立时执行空语句，仅改变计数变量len的值。

字符串复制

```
1   void my_strcpy(char target[], char source[])
2   {
3       int i=0; //两个字符串从首位开始复制
4       while((target[i++]=source[i])); //将source字符串中'\0'及之前的字符复制给target
5   }
```

- 根据C风格字符串的特点，字符串复制就是将source字符串中'\0'及之前的字符复制给target。常规的写法有很多种，这里采用了尽量简洁的写法。

- 该函数使用时注意**一定要保证target数组的大小足够保存source字符串**，必要的话可以在上面函数中添加一段长度验证程序（略）。

```
1   void my_strcpy(char target[], char source[])
2   {
3       int i=0; //两个字符串从首位开始复制
4       while((target[i++]=source[i])); //将source字符串中'\0'及之前的字符复制给target
5   }
```

- 第4行用到了几个知识点：①C语言用0代表“false”，非0代表“true”；②赋值表达式的终值是赋值符左边的数值；③变量i的自加运算符'++'在后，先取i的值进行运算，然后自增1。因此整个执行过程为：先将source[i]赋值给target[i]，然后i自增1，然后判断条件表达式结果是否为“true”，当source[i]为'\0'时结束循环。

- 特别注意，source字符串末尾的'\0'一定也要复制过来，或者手动添加。

字符串比较

```
1  int my_strcmp(char str1[], char str2[])
2  {
3      int res,i=0 ; //res保存第一个出现的不同字符的ASCII码差值。
4      while((res = str1[i] - str2[i]) == 0 && str1[i]) i++; //字符相同且str1未到末尾时i下移。
5      return res;
6  }
```

- 根据C风格字符串的特点，字符串比较就是依次对比str1和str2的同位置字符的ASCII码数值，直到不同为止，结果为：0（相等），正数（str1大），负数（str2大）。常规的写法有很多种，这里采用了尽量简洁的写法。

- 第4行的条件逻辑是str1未到末尾（不是'\0'），且同位置字符不相同时，位移变量i下移一位；不用判断str2是否到末尾，因为如果str1长度大于str2，str2先到末尾，那么此处字符是'\0'，必然小于str1同位置字符。

```
1  int my_strcmp(char str1[], char str2[])
2  {
3      int res,i=0 ; //res保存第一个出现的不同字符的ASCII码差值。
4      while((res = str1[i] - str2[i]) == 0 && str1[i]) i++; //字符相同且str1未到末尾时i下移。
5      return res;
6  }
```

- res返回的结果是str1和str2首次出现不同字符时，两个字符的ASCII码差值。

- 后面的练习题期望结果为：0（相等），1（str1大），-1（str2大），即：将任意正值变成1，任意负值变成-1。这里提供一个解法：将上述代码第5行改为return (res>0)-((-res)>0)。

随堂练习5.5

如何让my_strcmp的返回值为-1,0,1，即正数返回1，负数返回-1。

提示：知识点T267

字符串串联

```
1  void my_strcat(char target[], char source[])
2  {
3      int lt=strlen(target),i=0; //lt保存target串长，i标记source串下标
4      while((target[lt++]=source[i++])); //将source字符串中'\0'及之前的字符串联到target末尾
5  }
```

- 该函数实现思想类似于字符串复制，只是运算位置从target末尾的'\0'处开始。

- 该函数使用时注意一定要保证target数组的大小足够保存串联后的字符串，必要的话可以在上面函数中添加一段长度验证程序（略）。

```c
1  void my_strcat(char target[], char source[])
2  {
3      int lt=strlen(target),i=0; //lt保存target串长，i标记source串下标
4      while((target[lt++]=source[i++])); //将source字符串中'\0'及之前的字符串联到target末尾
5  }
```

- 第3行求target串长时用到了strlen函数，也可以自己写个循环代替，参考求串长代码。上述算法考虑到运算效率，不希望每次调用strlen函数重新计算target串长，因此用了临时变量lt记录target初始长度，这也是空间换时间的思想体现。

- 其他知识点参考my_strcpy的实现，第4行代码在复制source的'\0'后结束循环。

- 特别注意，source字符串末尾的'\0'一定也要复制过来，或者手动添加。

知识点

索引	要点	正链	反链
T552	掌握C风格字符串的基本操作以及实现方式，通过代码示例掌握C风格字符串的基本操作，掌握字典序		

5.3 C风格字符串的应用

例题5.16

从字符串str中查找某个字符ch第一次出现的位置。

【输入】输入一个字符串str和一个字符ch，以空格分隔。

【输出】输出ch在str中第一次出现的位置。

样例输入	样例输出
asdsf1234567 6	at position: 10

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char str[100],ch;
6      int i=0;
7      cin.getline(str,100); //读入字符串str
8      cin.get(ch); //读入要查找的字符ch
9      while(str[i]!='\0' && str[i]!=ch) i++; //未到串尾且未找到ch时，下标下移
10     //找到则输出位置，否则提示未找到
11     str[i] == ch ? cout<<"at position: "<<i<<endl : cout<<"not found!"<<endl;
12     return 0;
13 }
```

- 第9行将找到/找不到的条件合并在一起。
- 第11行用了条件运算符输出结果，注意不要遗漏了找不到的情况

例题5.17

从字符串str中查找第一个只出现一次的字符，如果没有找到，输出"-1"。

【输入】输入一个字符串str，可能包含空格。

【输出】如果找到符合条件的字符则输出，否则输出"-1"。

样例输入	样例输出
asdf fdsa 123	1

```
1  #include<iostream>
2  #include<cstring>
3  using namespace std;
4  int main()
5  {
6      char str[100];
7      int a[128] = {0};//记录字符串str中每个字符的出现次数
8      cin.getline(str,100); //读入字符串str
9      int len = strlen(str); //保存字符串长度
10     int i;
11     for(i=0; i<len; i++) //统计字符串str中每个字符的出现次数
12         a[str[i]] ++;
13     for(i=0; i<len; i++) //按出现顺序查找只出现一次的字符
14         if (1 == a[str[i]]) break;
15     i<len ? cout<<str[i]<<endl : cout<<"-1"<<endl;
16     return 0;
17 }
```

- 该程序也是一个比较经典的以空间换时间的操作，比较自然的思路是对每个字符统计出现次数，这需要用到嵌套循环，算法复杂度比本程序高。

- 第7行定义的数组a记录字符串str中每个字符的出现次数，因为ASCII表有128个字符，因此数组大小至少定义为128。

例题5.17

从字符串str中查找第一个只出现一次的字符，如果没有找到，输出"-1"。

【输入】输入一个字符串str，可能包含空格。

【输出】如果找到符合条件的字符则输出，否则输出"-1"。

样例输入	样例输出
asdf fdsa 123	1

```
1  #include<iostream>
2  #include<cstring>
3  using namespace std;
4  int main()
5  {
6      char str[100];
7      int a[128] = {0};//记录字符串str中每个字符的出现次数
8      cin.getline(str,100); //读入字符串str
9      int len = strlen(str); //保存字符串长度
10     int i;
11     for(i=0; i<len; i++) //统计字符串str中每个字符的出现次数
12         a[str[i]] ++;
13     for(i=0; i<len; i++) //按出现顺序查找只出现一次的字符
14         if (1 == a[str[i]]) break;
15     i<len ? cout<<str[i]<<endl : cout<<"-1"<<endl;
16     return 0;
17 }
```

- 第9行变量len保存字符串str的长度，因为要多次使用，所以用变量保存下来，节约调用strlen函数的次数。
- 第14行**将常量1放在'=='左边，这是一个良好的习惯**，当粗心将'=='误写为'='时编译器会报错。
- 如果查找到，str字符串中i的位置就是符合条件的字符；查找不到则最终i的值等于len。

例题5.18

输入若干单词，将它们从小到大排列后输出。

【输入】第一个输入数字n，然后n行，每行输入一个单词，每个单词无空格。

【输出】输出排序后的单词，每行一个单词。

样例输入	样例输出
5 word work homework job good	good homework job word work

```
1  #include<iostream>
2  #include<cstring>
3  using namespace std;
4  int main()
5  {
6      int n;
7      cin>>n;//首先读入第一行的整数，即单词数
8      cin.ignore();//清除残留的回车符
9      char str[n][100], t[100];//str保存读取的n个单词
10     for (int i=0; i<n; i++) cin.getline(str[i],100);
11     for(int i=0; i<n-1; i++)//交换法排序
12         for (int j=i+1; j<n; j++)
13             if (strcmp( str[i], str[j]) > 0 )
14             {
15                 strcpy ( t, str[i]);
16                 strcpy (str[i], str[j]);
17                 strcpy(str[j], t);
18             }
19     for (int i=0; i<n; i++)  cout<<str[i]<<endl;
20     return 0;
21 }
```

- 程序用一个字符串数组保存单词，在C语言中存储结构用二维字符数组描述。

- 第8行不要忘了**用cin.ignore()清除读走整数后缓冲区中残留的回车符**，否则接下来读到的第一个单词是空串。当然也有其他解决方法，例如用cin.get()读走单个字符、用getline读走空行等等。

例题5.18

输入若干单词，将它们从小到大排列后输出。

【输入】第一个输入数字n，然后n行，每行输入一个单词，每个单词无空格。

【输出】输出排序后的单词，每行一个单词。

样例输入	样例输出
5 word work homework job good	good homework job word work

```
1   #include<iostream>
2   #include<cstring>
3   using namespace std;
4   int main()
5   {
6       int n;
7       cin>>n;//首先读入第一行的整数，即单词数
8       cin.ignore();//清除残留的回车符
9       char str[n][100], t[100];//str保存读取的n个单词
10      for (int i=0; i<n; i++) cin.getline(str[i],100);
11      for(int i=0; i<n-1; i++)//交换法排序
12          for (int j=i+1; j<n; j++)
13              if (strcmp( str[i], str[j]) > 0 )
14              {
15                  strcpy ( t, str[i]);
16                  strcpy (str[i], str[j]);
17                  strcpy(str[j], t);
18              }
19      for (int i=0; i<n; i++)  cout<<str[i]<<endl;
20      return 0;
21  }
```

- 第11-18行是交换法排序，需要注意的是，**字符串的比较、赋值要用专用的处理函数strcmp、strcpy**，而不是用于简单变量的运算符。

题单

序号	洛谷	题目名称	知识点	序号	洛谷	题目名称	知识点
LX501	U270927	逆序输出数组	T511	LX502	U270929	数组求和	T514
LX503	U271197	昨天	T512	LX504	U271201	求最值	T514
LX505	U271200	平均之上	T514	LX506	U271202	编程团体赛	T516
LX507	U271204	光棍串	T522	LX508	U271209	小写数转大写	T522
LX509	U271211	最小正整数	T526	LX510	U271212	建国的难题	T526
LX511	U271214	排序	T527	LX512	U271215	矩阵运算	T515
LX513	U271216	字符串测试	T541	LX514	U271219	字符串跨距	T541,T546
LX515	U271222	首字母大写	T546	LX516	U271223	交替01串	T547
LX517	U271006	大数加1	T549				

THANKS