第一节 线性表-顺序存储-数组

数据结构的一些基本概念

时间复杂度

概念

可以理解为某算法代码执行次数`k`与数据规模`n`在**渐进**意义上成什么数学关系。用大O记号表示。

在没有特殊说明时，均默认时间复杂度为最坏情况下的时间复杂度。

- 常数时间$O(1)$：通过下标访问数组
- 线性时间$O(n)$：遍历一维数组
- 平方$O(n^2)$
- $O(\sqrt{n})$（根号n）
- 对数$O(\log n)$
- $O(n*\log n)$
- 对于带对数的时间复杂度，通常使用了“分而治之”算法思想，即每次将问题规模复杂度减半。

例题

求下列函数的时间复杂度（）

```
int func(int n) {
    int i = 0, sum = 0;
    while (sum < n) sum += ++i;
    return i;
}
```

A. $O(\log n)$ B. $O(\sqrt{n})$ C. $O(n)$ D. $O(n^2)$

解析：

先理解代码的执行过程，可将代码转化成自己便于理解的等价形式。

```
int func(int n) {
    int sum = 0;
    for (int i = 1; ; i++) {
        if (sum >= n) {
            return i;
        }
        sum += i;
    }
}
```

设for循环语句总共执行了k次

$$1+2+3+.....+k >= n$$
$$\frac{(1+k)*k}{2} >= n$$
$$0.5k^2+0.5k >= n$$
$$k \propto \sqrt{n}\text{（当}n \to \infty\text{）}$$

所以该函数的时间复杂度为$O(\sqrt{n})$

数据的逻辑结构

- 线性结构（一对一）
- 半线性结构（树）（一对多）
- 非线性结构（图）（多对多）

数组

数组的性质

`int A[n] = {a0, a1, a2, ..., a(n-1) }`

- 前驱、后继
- `A[i]`的在内存中物理地址：`A + i * sizeof(int)`
- 可通过下标来访问数组中的每一个元素，时间复杂度O(1)。随机访问

对数组进行常用的操作

- 查找：找到值等于e的元素在数组中的位置（下标）p
- 插入：在数组指定位置（下标）p插入一个元素e，原来的元素以及其后面的元素都依次往后挪一个位置。
- 删除：将数组指定下标p的元素删除，其后面的元素依次往前挪一个位置

顺序存储（数组）

定义

```
#define MAX_SIZE 1000

struct SqlList {
    int data[MAX_SIZE]; //数组
    int size;   //数组的长度
};
```

访问

时间复杂度O(1)

```
int get(SqlList &L, int p) {    //这里可以不加引用&，但是程序运行效率较低
    if (p < 0 || p >= L.size) return -1;    //访问下标不合法
    return L.data[p]
}
```

查找

时间复杂度O(n)

```
int find(SqlList &L, int x) {    //这里可以不加引用&，但是程序运行效率较低
    //在数组当中查找指定元素，如果找到了返回其数组下标，没找到返回-1
    for (int i = 0; i < L.size; i++) {
        if (L.data[i] == x) return i;
    }
    return -1;  // -1 表示没找到
}
```

插入

时间复杂度O(n)

```
void insert(SqlList &L, int pos, int e) {
    if (pos < 0 || pos > L.size) return;    //插入的下标不合法
    if (L.size == MAX_SIZE) return;         //表满了e
    for (int i = L.size; i > pos; i--) {    //往后腾出一个位置
        L.data[i] = L.data[i - 1];
    }
    L.data[pos] = e;                        //放进去
    L.size++;                               //表的长度+1
}
```

在表末尾插入一个元素

```
void push_back(SqlList &L, int e) {
    //insert(L, L.size, e);
    if (L.size == MAX_SIZE) return;         //表满了
    L.data[L.size++] = e;
}
```

删除

删除指定下标的单个元素：时间复杂度O(n)

```
void deleteElem(SqlList &L, int pos) {
    if (pos < 0 || pos >= L.size) return;   //删除的下标不合法
    for (int i = pos; i < L.size - 1; i++) {
        L.data[i] = L.data[i + 1];
    }
    L.size--;
}
```

删除表末尾元素，并将其返回

```
int pop_back(SqlList &L) {  //删除最后一个元素，并将其返回
    if (L.size < 1) return -1;  //数组为空，返回出错标记
    return L.data[--L.size];
}
```

批量删除下标 lo~hi 的元素：

1. 反复调用删除单个元素的函数？时间复杂度O(n^2)

```
void deleteElem_batch(SqlList &L, int lo, int hi) {
    for (int i = lo; i <= hi; i++) {
        deleteElem(L, i);
    }
}
```

2. 正确做法，时间复杂度O(n)

```
void deleteElem_batch(SqlList &L, int lo, int hi) {
    if (lo < 0 || hi >= L.size || lo > hi) return;  //删除的下标不合法
    int t = hi - lo + 1;
    for (int i = hi + 1; i < L.size; i++) {
        L.data[i - t] = L.data[i];
    }
    L.size -= t;
}
```

翻转

将数组翻转一遍，也就是数组第一个元素与最后一个元素交换，第二个元素与倒数第二个元素交换.....

```
void reverse_1(SqlList &L, int lo, int hi) {    //对数组下标lo到hi（不包括hi）的元素反转，左闭右开[lo, hi)
    for (int i = lo; i < (lo + hi) / 2; i++) {
```

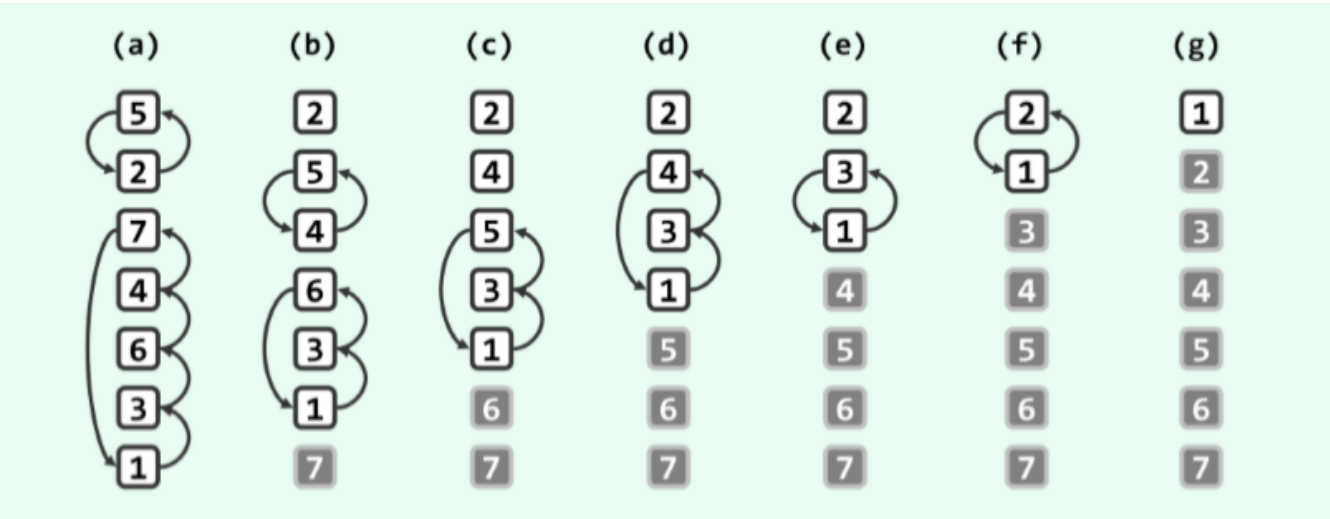
```
        // swap(L.data[i], L.data[hi - i - 1]);
        int temp = L.data[i];
        L.data[i] = L.data[hi - i - 1];
        L.data[hi - i - 1] = temp;
    }
}

void reverse_2(SqlList &L, int lo, int hi) {    //对数组下标lo到hi（不包括hi）的元素反转，左闭右开[lo, hi)
    for (int i = lo, j = hi - 1; i < j; i++, j--) {
        // swap(L.data[i], L.data[j]);
        int temp = L.data[i];
        L.data[i] = L.data[j];
        L.data[j] = temp;
    }
}
```

冒泡排序

时间复杂度$O(n^2)$

逆序对




```
void bubbleSort(int A[], int n) {
    for (int t = n; t > 0; t--) {
        bool sorted = true;
        for (int i = 1; i < t; i++) {
            if (A[i - 1] > A[i]) {
                swap(A[i - 1], A[i]);
                sorted = false;
            }
        }
        if (sorted) break;
    }
}
```

上机完整代码

大部分数据结构教材采用的是这种写法

```cpp
#include <stdio.h>
#include <algorithm>
using namespace std;

#define MAX_SIZE 1000

struct SqlList {
    int data[MAX_SIZE]; //数组
    int size;   //数组的长度
};

void initSqlList(SqlList &L) {
    L.size = 0;
}

int get(SqlList &L, int p) {    //这里可以不加引用&，但是程序运行效率较低
    if (p < 0 || p >= L.size) return -1;    //访问下标不合法
    return L.data[p]
}

int find(SqlList &L, int x) {   //这里可以不加引用&，但是程序运行效率较低
    for (int i = 0; i < L.size; i++) {
        if (L.data[i] == x) return i;
    }
    return -1;  // -1 表示没找到
}

void insert(SqlList &L, int pos, int e) {
    if (pos < 0 || pos > L.size) return;    //插入的下标不合法
    if (L.size == MAX_SIZE) return;         //表满了
    for (int i = L.size; i > pos; i--) {    //往后腾出一个位置
        L.data[i] = L.data[i - 1];
    }
    L.data[pos] = e;                        //放进去
    L.size++;                               //表的长度+1
}

void push_back(SqlList &L, int e) {
    //insert(L, L.size, e)
    if (L.size == MAX_SIZE) return;         //表满了
    L.data[L.size++] = e;
}

void deleteElem(SqlList &L, int pos) {
    if (pos < 0 || pos > L.size - 1) return;    //删除的下标不合法
    for (int i = pos; i < L.size - 1; i++) {
        L.data[i] = L.data[i + 1];
    }
    L.size--;
}

```

```cpp
void deleteElem_batch(SqlList &L, int lo, int hi) {     //批量删除下标lo到下标hi的元素[lo, hi]
    if (lo < 0 || hi >= L.size || lo > hi) return;  //删除的下标不合法
    int t = hi - lo + 1;
    for (int i = hi + 1; i < L.size; i++) {
        L.data[i - t] = L.data[i];
    }
    L.size -= t;
}

int pop_back(SqlList &L) {  //删除最后一个元素，并将其返回
    if (L.size < 1) return -1;  //数组为空，返回出错标记
    return L.data[--L.size];
}

void printArray(SqlList &L) {   //输出整个数组。可以不加&，但运行效率降低
    for (int i = 0; i < L.size; i++) {
        printf("%d ", L.data[i]);
    }
    printf("\n");
}

void reverse_1(SqlList &L, int lo, int hi) {    //对数组下标lo到hi（不包括hi）的元素反转，左闭右开[lo, hi)
    for (int i = lo; i < (lo + hi) / 2; i++) {
        //swap(L.data[i], L.data[hi - i - 1]);
        int temp = L.data[i];
        L.data[i] = L.data[hi - i - 1];
        L.data[hi - i - 1] = temp;
    }
}

void reverse_2(SqlList &L, int lo, int hi) {    //对数组下标lo到hi（不包括hi）的元素反转，左闭右开[lo, hi)
    for (int i = lo, j = hi - 1; i < j; i++, j--) {
        //swap(L.data[i], L.data[j]);
        int temp = L.data[i];
        L.data[i] = L.data[j];
        L.data[j] = temp;
    }
}

void bubbleSort(SqlList &L) {
    for (int t = L.size; t > 0; t--) {
        bool sorted = true;
        for (int i = 1; i < t; i++) {
            if (L.data[i - 1] > L.data[i]) {
                swap(L.data[i - 1], L.data[i]);
                sorted = false;
            }
        }
        if (sorted) break;
    }
```

```
}

int main() {
    SqlList L;
    initSqlList(L);

    for (int i = 1; i <= 6; i++) {
        push_back(L, i);
    }
    pop_back(L);
    printArray(L);

    insert(L, 2, 10);
    printArray(L);

    insert(L, 1, 20);
    printArray(L);

    deleteElem(L, 3);
    printArray(L);

    reverse_1(L, 0, L.size);
    printArray(L);

    reverse_2(L, 0, L.size);
    printArray(L);

    bubbleSort(L);
    printArray(L);

    deleteElem_batch(L, 1, 3);
    printArray(L);

    return 0;
}
```

完全不封装的代码示例

考试的时候（初试、机试）一般使用这种写法

```
#include <stdio.h>
#define MAX_SIZE 1000

int _data[MAX_SIZE];    //数组
int _size;  //数组的长度

void push_back(int e) {
    if (_size == MAX_SIZE) return;          //表满了
    _data[_size++] = e;
}

void swap(int &a, int &b) {
    int temp = a;
```

```
    a = b;
    b = temp;
}

void bubbleSort(int A[], int n) {
    for (int t = n; t > 0; t--) {
        bool sorted = true;
        for (int i = 1; i < t; i++) {
            if (A[i - 1] > A[i]) {
                swap(A[i - 1], A[i]);
                sorted = false;
            }
        }
        if (sorted) break;
    }
}

void printArray() {
    for (int i = 0; i < _size; i++) {
        printf("%d ", _data[i]);
    }
    printf("\n");
}

int main() {
    for (int i = 1, sign = 1; i <= 5; i++, sign = -sign) {
        push_back(sign * i);
    }
    printArray();

    bubbleSort(_data, _size);
    printArray();

    return 0;
}
```

使用面向对象来封装代码

体验一下使用C++面向对象的方式来描述数据结构，应该更能加深你对数据结构的理解，尤其是抽象数据类型ADT概念的理解。

```
#include <cstdio>
#include <algorithm>
using namespace std;

class Vector {
private:
    int* _data; //动态分配数组
    int _size;  //当前数组实际使用的长度
    int _capacity;  //当前数组的最大容量

public:
```

```
    Vector(int capacity = 1) {  //构造函数
        _data = new int[capacity];  //动态分配数组
        _capacity = capacity;
        _size = 0;
    }

    ~Vector() { //析构函数
        delete[] _data;
        printf("数组所占内存已释放\n");
    }

    void expandArray() {    //实现一个自动扩容机制
        int newCapacity = _capacity * 2;    //简单的扩充为原来的2倍
        int* newArray = new int[newCapacity];
        if (newArray == NULL) throw exception("内存分配失败");
        for (int i = 0; i < _size; i++) {
            newArray[i] = _data[i];
        }
        _capacity = newCapacity;
        delete[] _data;
        _data = newArray;
    }

    int& get(int p) {
        if (p < 0 || p >= _size) throw exception("访问下标不合法");
        return _data[p];
    }

    int find(int x) {
        for (int i = 0; i < _size; i++) {
            if (_data[i] == x) return i;
        }
        return -1;  // -1 表示没找到
    }

    void insert(int pos, int e) {
        if (pos < 0 || pos > _size) throw exception("插入的下标不合法");
        if (_size == _capacity) expandArray();
        for (int i = _size; i > pos; i--) {     //往后腾出一个位置
            _data[i] = _data[i - 1];
        }
        _data[pos] = e;                     //放进去
        _size++;                            //表的长度+1
    }

    void push_back(int e) {
        if (_size == _capacity) expandArray();
        _data[_size++] = e;
    }

    void deleteElem(int pos) {
        if (pos < 0 || pos > _size - 1) throw exception("删除的下标不合法");
        for (int i = pos; i < _size - 1; i++) {
```

```
            _data[i] = _data[i + 1];
        }
        _size--;
    }

    int pop_back() {    //删除数组最后一个元素，并将其返回
        if (_size < 1) throw exception("数组为空，不能pop");
        return _data[--_size];
    }

    bool empty() {
        return _size == 0;
    }

    void clear() {
        _size = 0;
    }

    int size() {
        return _size;
    }

    int capacity() {
        return _capacity;
    }

    void printAll() {   //输出整个数组
        for (int i = 0; i < _size; i++) {
            printf("%d ", _data[i]);
        }
        printf("\n");
    }

    //对数组下标lo到hi（不包括hi）的元素反转，左闭右开[lo, hi)
    void reverse(int lo, int hi) {
        for (int i = lo, j = hi - 1; i < j; i++, j--) {
            swap(_data[i], _data[j]);
        }
    }

    //对整个数组进行反转。该函数与上面的函数同名，但参数列表不同，使用了函数重载特性
    void reverse() {
        for (int i = 0, j = _size - 1; i < j; i++, j--) {
            swap(_data[i], _data[j]);
        }
    }

    void bubbleSort() {
        for (int t = _size; t > 0; t--) {
            bool sorted = true;
            for (int i = 1; i < t; i++) {
                if (_data[i - 1] > _data[i]) {
                    swap(_data[i - 1], _data[i]);
```

```
                    sorted = false;
                }
            }
            if (sorted) break;
        }
    }

    //体验一下运算符[]重载，可以像使用数组一样方便的使用我们自己写的Vector对象
    int& operator[] (int p) {
        //return get(p);
        if (p < 0 || p >= _size) throw exception("访问下标不合法");
        return _data[p];
    }
};

int main() {
    Vector L;

    //L.pop_back();
    try {
        L.pop_back();           //如果这句不放在try-catch语句中执行，代码运行时会报错
    } catch (exception e) {     //体验一下异常处理（仅作了解）
        printf("%s\n", e.what());   //打印异常e的信息
    }

    for (int i = 1; i <= 6; i++) {
        L.push_back(i);
    }
    L.pop_back();
    L.printAll();

    L.insert(2, 10);
    L.printAll();

    L.insert(1, 20);
    L.printAll();

    L.deleteElem(3);
    L.printAll();

    L.reverse();
    L.printAll();

    L.bubbleSort();
    L.printAll();

    //体验一下运算符[]重载，可以像使用数组一样方便的使用我们自己写的Vector
    printf("L[2] = %d\n", L[2]);
    L[1] = 100;
    L.printAll();
    return 0;
}
```

体验C++ STL：Vector

vector是C++标准库中自带的一个封装好了的可扩容数组。

在创建vector容器时，可以使用构造函数可以为容器直接分配足够大的内存空间，也可以使用`reserve()`函数设置vector的容量。

```
vector<int> v1;     //v1的初始容量为默认值0
vector<int> v2(100);    //v2的初始容量为100
v2.reserve(1000);       //这时v2的容量变成了1000
```

如果想像C语言数组那样对一个vector容器赋初始值，可以这么做

```
int A[] = {1, 2, 3, 4};
vector<int> V = {1, 2, 3, 4};
```

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

int main() {
    vector<int> V = { 10,20,30 };
    printf("capacity: %d\n", V.capacity());
    printf("size: %d\n", V.size());

    V.reserve(100);
    printf("capacity: %d\n", V.capacity());
    printf("size: %d\n", V.size()); //注意区分一下vector的capacity和size

    for (int i = 1; i <= 5; i++) {
        V.push_back(i);
    }
    V.pop_back();

    for (int i = 0; i < V.size(); i++) {
        printf("%d ", V[i]);
    }
    printf("\n");

    printf("This vector is empty? %s\n", V.empty() ? "Yes" : "No");

    for (int x : V) {   //遍历一个容器（只读）还可以这样操作
        //增强for循环，C++11语法
        printf("%d ", x);
    }
    printf("\n");

    for (int &x : V) {  //如果想修改的话还得这样操作，需要加个引用&
        x = 100;
    }
```

```
35
36      for (int x : V) {   //打印试试
37          printf("%d ", x);
38      }
39      printf("\n");
40      return 0;
41  }
```

对数组求和

```
1  //样例输入，第一行表示有多少个数
2  5
3  2 4 6 3 1
4  //样例输出
5  16
```

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main() {
6       int N;
7       scanf("%d", &N);
8       vector<int> A(N);
9       for (int i = 0; i < N; i++) {
10          scanf("%d", &A[i]);
11      }
12      int sum = 0;
13      for (int x : A) {
14          sum += x;
15      }
16      printf("%d\n", sum);
17      return 0;
18  }
```

抽象数据类型

各种数据结构都可看作是由若干数据项组成的集合，同时对数据项定义一组标准的操作。现代数据结构普遍遵从“信息隐藏”的理念，通过统一接口和内部封装，分层次从整体上加以设计、 实现与使用。

所谓封装，就是将数据项与相关的操作结合为一个整体，并将其从外部的可见性划分为若干级别，从而将数据结构的外部特性与其内部实现相分离，提供一致且标准的对外接口，隐藏内部的实现细节。于是，数据集合及其对应的操作可超脱于具体的程序设计语言、具体的实现方式， 即构成所谓的抽象数据类型（abstract data type, ADT）。抽象数据类型的理论催生了现代面向对象的程序设计语言，而支持封装也是此类语言的基本特征。

考试题目

插入排序思想

在数组A[]中，表中元素存放在数组下标`0 ~ m+n-1`中，且数组前m个元素递增有序，后n个元素递增有序。设计一个算法，使得数组整体有序。

时间复杂度$O(n^2 + m * n)$

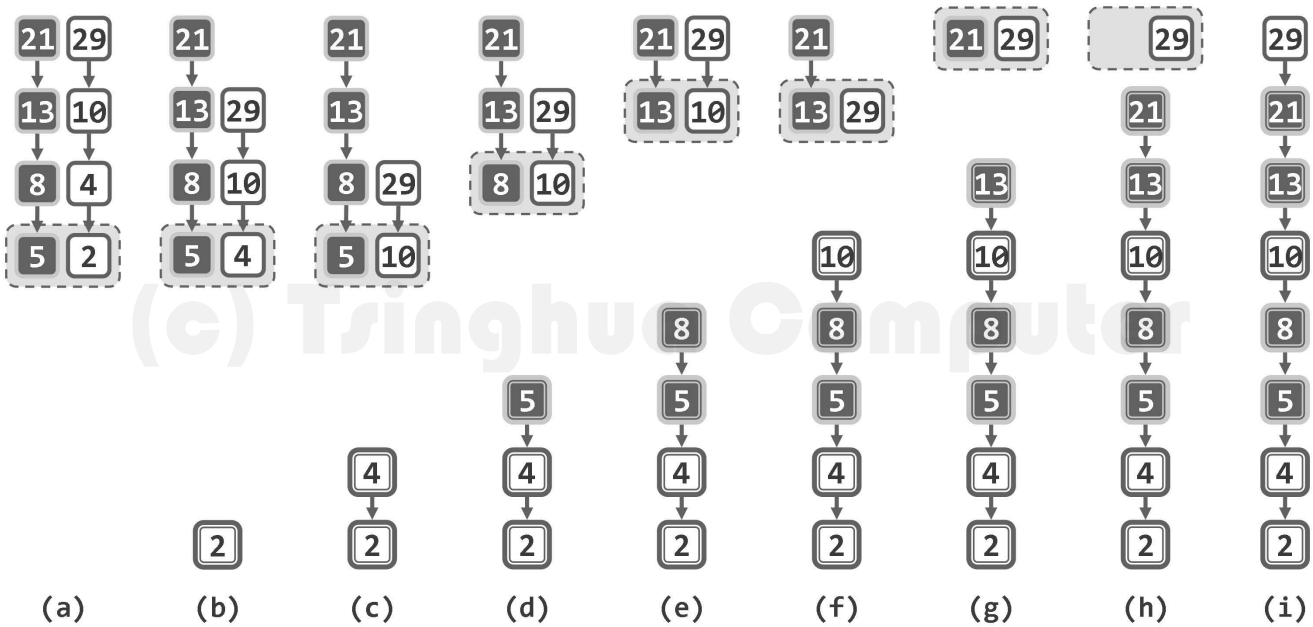
空间复杂度$O(1)$

```
#include <stdio.h>

void AdjustArray(int A[], int M, int N) {
    for (int k = M; k < M + N; k++) {
        int temp = A[k];
        int i;
        for (i = k - 1; i >= 0 && A[i] > temp; i--) {
            A[i + 1] = A[i];
        }
        A[i + 1] = temp;
    }
}

int main() {
    int A[] = { 2,5,10,12,18,19,1,2,3,6,9,11,20 };
    int M = 6, N = 7;
    AdjustArray(A, M, N);
    for (int i = 0; i < M + N; i++) {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```

归并排序思想：合并两个有序数组



时间复杂度$O(m + n)$

```cpp
#include <cstdio>

int* mergeSortedArray(int A[], int m, int B[], int n) {
    int* C = new int[m + n];
    int i = 0, j = 0;
    int k = 0;
    while (i < m && j < n) {
        if (A[i] < B[j]) {
            C[k++] = A[i++];
        } else {
            C[k++] = B[j++];
        }
    }
    while (i < m) {
            C[k++] = A[i++];
    }
    while (j < n) {
            C[k++] = B[j++];
    }
    return C;
}

int main() {
    int A[] = { 2,5,10,12,18,19 };
    int B[] = { 1,2,3,6,9,11,20 };
    int M = sizeof(A) / sizeof(int);
    int N = sizeof(B) / sizeof(int);
    int* C = mergeSortedArray(A, M, B, N);
    for (int i = 0; i < M + N; i++) {
        printf("%d ", C[i]);
    }
    printf("\n");
    return 0;
}
```

删除无序数组中所有值为x的元素

时间复杂度$O(n)$

```cpp
#include <stdio.h>

int remove_x_1(int A[], int &N, int x) {
    //在长度为N的数组A中删除所有值为x的元素，并返回删除元素的个数
    int cnt = 0;
    for (int i = 0; i < N; i++) {
        if (A[i] == x) {
            cnt++;
            continue;
        }
        A[i - cnt] = A[i];  //当前元素应该往前挪cnt个位置
    }
    N -= cnt;
    return cnt;
```

```
}

int remove_x_2(int A[], int &N, int x) {
    //在长度为N的数组A中删除所有值为x的元素，并返回删除元素的个数
    int index = 0;
    for (int i = 0; i < N; i++) {
        if (A[i] != x) {
            A[index++] = A[i];  //当前元素应当放在数组index下标处
        }
    }
    int ret = N - index;
    N = index;
    return ret;
}

int main() {
    int A[] = { 2,3,3,5,-1,3,2,10 };
    int N = sizeof(A) / sizeof(int);
    //remove_x_1(A, N, 3);
    remove_x_2(A, N, 3);
    printf("N = %d\n", N);
    for (int i = 0; i < N; i++) {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```

有序数组去重

时间复杂度$O(n)$

```
#include <stdio.h>

int unique_1(int A[], int &N) {
    //对长度为N的数组A进行去重，函数返回删除元素的个数
    if (N == 1) return 0;
    int cnt = 0;
    for (int i = 1; i < N; i++) {
        if (A[i] == A[i - 1]) {
            cnt++;
            continue;
        }
        A[i - cnt] = A[i];  //当前元素应该往前挪cnt个位置
    }
    N -= cnt;
    return cnt;
}

int unique_2(int A[], int &N) {
    //对长度为N的数组A进行去重，函数返回删除元素的个数
    if (N == 1) return 0;
    int index = 1;
```

```
    for (int i = 1; i < N; i++) {
        if (A[i] != A[index - 1]) {
            A[index++] = A[i];  //当前元素应当放在数组index下标处
        }
    }
    int ret = N - index;
    N = index;
    return ret;
}

int main() {
    int A[] = { 1,1,2,3,3,3,4,4,7,7,7,9,9,9,10 };
    int N = sizeof(A) / sizeof(int);
    printf("before unique, N = %d\n", N);
    //int cnt = unique_1(A, N);
    int cnt = unique_2(A, N);
    printf("after unique: cnt = %d, N = %d\n", cnt, N);
    for (int i = 0; i < N; i++) {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```

奇偶分离

设计一个算法，使得数组当中所有的奇数都位于偶数的前面。

若能保证所有奇数与奇数之间、偶数与偶数之间的相对次序不变，则称该算法是稳定算法

方法1：

时间$O(n)$，空间$O(n)$，不稳定

```
void adjustArray_1(int A[], int N) {
    int* arr = new int[N];
    int p = 0, q = N - 1;
    for (int i = 0; i < N; i++) {
        if (isOddNum(A[i])) {
            arr[p++] = A[i];
        } else {
            arr[q--] = A[i];
        }
    }
    for (int i = 0; i < N; i++) {
        A[i] = arr[i];
    }
    delete[] arr;
}
```

方法2：

时间$O(n)$，空间$O(n)$，稳定

```
void adjustArray_2(int A[], int N) {
    int oddNumCnt = 0;
    for (int i = 0; i < N; i++) {
        if (isOddNum(A[i])) oddNumCnt++;
    }
    int* arr = new int[N];
    int p = 0, q = oddNumCnt;
    for (int i = 0; i < N; i++) {
        if (A[i] % 2 != 0) {
            arr[p++] = A[i];
        } else {
            arr[q++] = A[i];
        }
    }
    for (int i = 0; i < N; i++) {
        A[i] = arr[i];
    }
    delete[] arr;
}
```

方法3：

时间$O(n)$，空间$O(1)$，不稳定

```
void adjustArray_3(int A[], int N) {
    int i = 0, j = N - 1;
    while (i < j) {
        while (i < j && isOddNum(A[i])) i++;
        while (i < j && isEvenNum(A[i])) j--;
        /*
        当算法执行到此时有两种情况：
        情况1：i指向偶数，j指向奇数，此时需交换；
        情况2：i，j均指向同一个元素，此时应结束算法
            但对同一个元素执行swap操作也没问题，
            因此没有对该情况做专门判断
        */
        swap(A[i], A[j]);
    }
}
```

方法4：

时间$O(n^2)$，空间$O(1)$，稳定

```
void adjustArray_4(int A[], int N) {
    int firstEvenPos = -1;
    for (int i = 0; i < N; i++) {
        if (isEvenNum(A[i])) {
            firstEvenPos = i;
            break;
        }
    }
    if (firstEvenPos == -1) return; //说明数组全是奇数

    for (int i = firstEvenPos + 1; i < N; i++) {
        if (isOddNum(A[i])) {
            //说明当前从下标firstEvenPos到i-1全是偶数，下标i为奇数
            //需要将这些偶数全部往后面挪一个位置，将奇数放进去
            //并更新firstEvenPos+=1
            int temp = A[i];
            for (int j = i; j > firstEvenPos; j--) {
                A[j] = A[j - 1];
            }
            A[firstEvenPos] = temp;
            firstEvenPos++;
        }
    }
}
```

完整代码：

```
#include <stdio.h>
#include <algorithm>
using namespace std;

bool isOddNum(int n) { return n % 2 == 1; }
bool isEvenNum(int n) { return n % 2 == 0; }

void adjustArray_1(int A[], int N) {
    int* arr = new int[N];
    int p = 0, q = N - 1;
    for (int i = 0; i < N; i++) {
        if (isOddNum(A[i])) {
            arr[p++] = A[i];
        } else {
            arr[q--] = A[i];
        }
    }
    for (int i = 0; i < N; i++) {
        A[i] = arr[i];
    }
    delete[] arr;
}

void adjustArray_2(int A[], int N) {
```

```
    int oddNumCnt = 0;
    for (int i = 0; i < N; i++) {
        if (isOddNum(A[i])) oddNumCnt++;
    }
    int* arr = new int[N];
    int p = 0, q = oddNumCnt;
    for (int i = 0; i < N; i++) {
        if (A[i] % 2 != 0) {
            arr[p++] = A[i];
        } else {
            arr[q++] = A[i];
        }
    }
    for (int i = 0; i < N; i++) {
        A[i] = arr[i];
    }
    delete[] arr;
}

void adjustArray_3(int A[], int N) {
    int i = 0, j = N - 1;
    while (i < j) {
        while (i < j && isOddNum(A[i])) i++;
        while (i < j && isEvenNum(A[i])) j--;
        /*
        当算法执行到此时有两种情况：
        情况1：i指向偶数，j指向奇数，此时需交换；
        情况2：i，j均指向同一个元素，此时应结束算法
            但对同一个元素执行swap操作也没问题，
            因此没有对该情况做专门判断
        */
        swap(A[i], A[j]);
    }
}

void adjustArray_4(int A[], int N) {
    int firstEvenPos = -1;
    for (int i = 0; i < N; i++) {
        if (isEvenNum(A[i])) {
            firstEvenPos = i;
            break;
        }
    }
    if (firstEvenPos == -1) return; //说明数组全是奇数

    for (int i = firstEvenPos + 1; i < N; i++) {
        if (isOddNum(A[i])) {
            //说明当前从下标firstEvenPos到i-1全是偶数，下标i为奇数
            //需要将这些偶数全部往后面挪一个位置，将奇数放进去
            //并更新firstEvenPos+=1
            int temp = A[i];
            for (int j = i; j > firstEvenPos; j--) {
                A[j] = A[j - 1];
```

```
            }
            A[firstEvenPos] = temp;
            firstEvenPos++;
        }
    }
}

int main() {
    int A[] = { 1,2,2,2,1,1,2,2 };
    int N = sizeof(A) / sizeof(int);

    //adjustArray_1(A, N);
    //adjustArray_2(A, N);
    //adjustArray_3(A, N);
    adjustArray_4(A, N);

    for (int i = 0; i < N; i++) {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```