

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（ ）

```
1 int func(int n) {  
2     int i = 0, sum = 0;  
3     while (sum < n) sum += ++i;  
4     return i;  
5 }
```

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。

```
1 int func(int n) {  
2     int sum = 0;  
3     for (int i = 1; ; i++) {  
4         if (sum >= n) {  
5             return i;  
6         }  
7         sum += i;  
8     }  
9 }
```

设for循环语句总共执行了 k 次

$$1 + 2 + 3 + \dots + k \geq n$$

$$\frac{(1+k) * k}{2} \geq n$$

$$0.5k^2 + 0.5k \geq n$$

$$k \propto \sqrt{n} \text{ (当 } n \rightarrow \infty \text{)}$$

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

```
1 #define MAX_SIZE 1000
2
3 struct SqlList {
4     int data[MAX_SIZE]; //数组
5     int size; //数组的长度
6 };
```

访问

时间复杂度 $O(1)$

```

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
2     if (p < 0 || p >= L.size) return -1;    //访问下标不合法
3     return L.data[p]
4 }

```

查找

时间复杂度 $O(n)$

```

1 int find(SqlList &L, int x) {    //这里可以不加引用&, 但是程序运行效率较低
2     //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
3     for (int i = 0; i < L.size; i++) {
4         if (L.data[i] == x) return i;
5     }
6     return -1;    // -1 表示没找到
7 }

```

插入



时间复杂度 $O(n)$

```

1 void insert(SqlList &L, int pos, int e) {
2     if (pos < 0 || pos > L.size) return;    //插入的下标不合法
3     if (L.size == MAX_SIZE) return;    //表满了e
4     for (int i = L.size; i > pos; i--) {    //往后腾出一个位置
5         L.data[i] = L.data[i - 1];
6     }
7     L.data[pos] = e;    //放进去
8     L.size++;    //表的长度+1
9 }

```

在表末尾插入一个元素

```

1 void push_back(SqlList &L, int e) {
2     //insert(L, L.size, e);
3     if (L.size == MAX_SIZE) return;    //表满了
4     L.data[L.size++] = e;
5 }

```

删除



删除指定下标的单个元素：时间复杂度 $O(n)$

```
1 void deleteElem(SqlList &L, int pos) {
2     if (pos < 0 || pos >= L.size) return; //删除的下标不合法
3     for (int i = pos; i < L.size - 1; i++) {
4         L.data[i] = L.data[i + 1];
5     }
6     L.size--;
7 }
```

删除表末尾元素，并将其返回

```
1 int pop_back(SqlList &L) { //删除最后一个元素，并将其返回
2     if (L.size < 1) return -1; //数组为空，返回出错标记
3     return L.data[--L.size];
4 }
```

批量删除下标 $lo \sim hi$ 的元素：

1. 反复调用删除单个元素的函数？时间复杂度 $O(n^2)$

```
1 void deleteElem_batch(SqlList &L, int lo, int hi) {
2     for (int i = lo; i <= hi; i++) {
3         deleteElem(L, i);
4     }
5 }
```

2. 正确做法，时间复杂度 $O(n)$

```
1 void deleteElem_batch(SqlList &L, int lo, int hi) {
2     if (lo < 0 || hi >= L.size || lo > hi) return; //删除的下标不合法
3     int t = hi - lo + 1;
4     for (int i = hi + 1; i < L.size; i++) {
5         L.data[i - t] = L.data[i];
6     }
7     L.size -= t;
8 }
```

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.....

```
1 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)
2     for (int i = lo; i < (lo + hi) / 2; i++) {
```

```

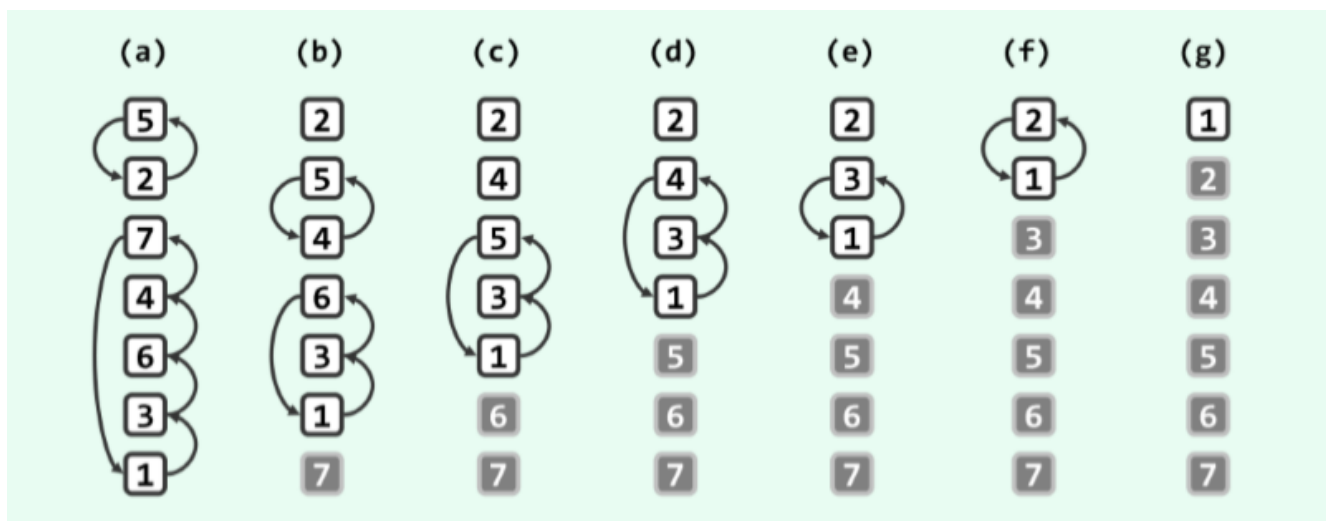
3      // swap(L.data[i], L.data[hi - i - 1]);
4      int temp = L.data[i];
5      L.data[i] = L.data[hi - i - 1];
6      L.data[hi - i - 1] = temp;
7  }
8  }
9
10 void reverse_2(SqlList &L, int lo, int hi) {    //对数组下标lo到hi (不包括hi) 的元素反转, 左
    闭右开[lo, hi)
11     for (int i = lo, j = hi - 1; i < j; i++, j--) {
12         // swap(L.data[i], L.data[j]);
13         int temp = L.data[i];
14         L.data[i] = L.data[j];
15         L.data[j] = temp;
16     }
17 }

```

冒泡排序

时间复杂度 $O(n^2)$

逆序对



```

1  void bubbleSort(int A[], int n) {
2      for (int t = n; t > 0; t--) {
3          bool sorted = true;
4          for (int i = 1; i < t; i++) {
5              if (A[i - 1] > A[i]) {
6                  swap(A[i - 1], A[i]);
7                  sorted = false;
8              }
9          }
10         if (sorted) break;
11     }
12 }

```

上机完整代码

大部分数据结构教材采用的是这种写法

```
1  #include <stdio.h>
2  #include <algorithm>
3  using namespace std;
4
5  #define MAX_SIZE 1000
6
7  struct SqlList {
8      int data[MAX_SIZE]; //数组
9      int size;    //数组的长度
10 };
11
12 void initSqlList(SqlList &L) {
13     L.size = 0;
14 }
15
16 int get(SqlList &L, int p) {    //这里可以不加引用&, 但是程序运行效率较低
17     if (p < 0 || p >= L.size) return -1;    //访问下标不合法
18     return L.data[p]
19 }
20
21 int find(SqlList &L, int x) {    //这里可以不加引用&, 但是程序运行效率较低
22     for (int i = 0; i < L.size; i++) {
23         if (L.data[i] == x) return i;
24     }
25     return -1;    // -1 表示没找到
26 }
27
28 void insert(SqlList &L, int pos, int e) {
29     if (pos < 0 || pos > L.size) return;    //插入的下标不合法
30     if (L.size == MAX_SIZE) return;    //表满了
31     for (int i = L.size; i > pos; i--) {    //往后腾出一个位置
32         L.data[i] = L.data[i - 1];
33     }
34     L.data[pos] = e;    //放进去
35     L.size++;    //表的长度+1
36 }
37
38 void push_back(SqlList &L, int e) {
39     //insert(L, L.size, e)
40     if (L.size == MAX_SIZE) return;    //表满了
41     L.data[L.size++] = e;
42 }
43
44 void deleteElem(SqlList &L, int pos) {
45     if (pos < 0 || pos > L.size - 1) return;    //删除的下标不合法
46     for (int i = pos; i < L.size - 1; i++) {
47         L.data[i] = L.data[i + 1];
48     }
49     L.size--;
50 }
51
```

```

52 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]
53     if (lo < 0 || hi >= L.size || lo > hi) return; //删除的下标不合法
54     int t = hi - lo + 1;
55     for (int i = hi + 1; i < L.size; i++) {
56         L.data[i - t] = L.data[i];
57     }
58     L.size -= t;
59 }
60
61 int pop_back(SqlList &L) { //删除最后一个元素，并将其返回
62     if (L.size < 1) return -1; //数组为空，返回出错标记
63     return L.data[--L.size];
64 }
65
66 void printArray(SqlList &L) { //输出整个数组。可以不加&，但运行效率降低
67     for (int i = 0; i < L.size; i++) {
68         printf("%d ", L.data[i]);
69     }
70     printf("\n");
71 }
72
73 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)
74     for (int i = lo; i < (lo + hi) / 2; i++) {
75         //swap(L.data[i], L.data[hi - i - 1]);
76         int temp = L.data[i];
77         L.data[i] = L.data[hi - i - 1];
78         L.data[hi - i - 1] = temp;
79     }
80 }
81
82 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)
83     for (int i = lo, j = hi - 1; i < j; i++, j--) {
84         //swap(L.data[i], L.data[j]);
85         int temp = L.data[i];
86         L.data[i] = L.data[j];
87         L.data[j] = temp;
88     }
89 }
90
91 void bubbleSort(SqlList &L) {
92     for (int t = L.size; t > 0; t--) {
93         bool sorted = true;
94         for (int i = 1; i < t; i++) {
95             if (L.data[i - 1] > L.data[i]) {
96                 swap(L.data[i - 1], L.data[i]);
97                 sorted = false;
98             }
99         }
100         if (sorted) break;
101     }

```

```

102 }
103
104 int main() {
105     SqlList L;
106     initSqlList(L);
107
108     for (int i = 1; i <= 6; i++) {
109         push_back(L, i);
110     }
111     pop_back(L);
112     printArray(L);
113
114     insert(L, 2, 10);
115     printArray(L);
116
117     insert(L, 1, 20);
118     printArray(L);
119
120     deleteElem(L, 3);
121     printArray(L);
122
123     reverse_1(L, 0, L.size);
124     printArray(L);
125
126     reverse_2(L, 0, L.size);
127     printArray(L);
128
129     bubbleSort(L);
130     printArray(L);
131
132     deleteElem_batch(L, 1, 3);
133     printArray(L);
134
135     return 0;
136 }

```

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

```

1  #include <stdio.h>
2  #define MAX_SIZE 1000
3
4  int _data[MAX_SIZE];    //数组
5  int _size;    //数组的长度
6
7  void push_back(int e) {
8      if (_size == MAX_SIZE) return;    //表满了
9      _data[_size++] = e;
10 }
11
12 void swap(int &a, int &b) {
13     int temp = a;

```



```

14     a = b;
15     b = temp;
16 }
17
18 void bubbleSort(int A[], int n) {
19     for (int t = n; t > 0; t--) {
20         bool sorted = true;
21         for (int i = 1; i < t; i++) {
22             if (A[i - 1] > A[i]) {
23                 swap(A[i - 1], A[i]);
24                 sorted = false;
25             }
26         }
27         if (sorted) break;
28     }
29 }
30
31 void printArray() {
32     for (int i = 0; i < _size; i++) {
33         printf("%d ", _data[i]);
34     }
35     printf("\n");
36 }
37
38 int main() {
39     for (int i = 1, sign = 1; i <= 5; i++, sign = -sign) {
40         push_back(sign * i);
41     }
42     printArray();
43
44     bubbleSort(_data, _size);
45     printArray();
46
47     return 0;
48 }

```

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。

```

1  #include <cstdio>
2  #include <algorithm>
3  using namespace std;
4
5  class Vector {
6  private:
7      int* _data; //动态分配数组
8      int _size;  //当前数组实际使用的长度
9      int _capacity; //当前数组的最大容量
10
11 public:

```

```

12 Vector(int capacity = 1) { //构造函数
13     _data = new int[capacity]; //动态分配数组
14     _capacity = capacity;
15     _size = 0;
16 }
17
18 ~Vector() { //析构函数
19     delete[] _data;
20     printf("数组所占内存已释放\n");
21 }
22
23 void expandArray() { //实现一个自动扩容机制
24     int newCapacity = _capacity * 2; //简单的扩充为原来的2倍
25     int* newArray = new int[newCapacity];
26     if (newArray == NULL) throw exception("内存分配失败");
27     for (int i = 0; i < _size; i++) {
28         newArray[i] = _data[i];
29     }
30     _capacity = newCapacity;
31     delete[] _data;
32     _data = newArray;
33 }
34
35 int& get(int p) {
36     if (p < 0 || p >= _size) throw exception("访问下标不合法");
37     return _data[p];
38 }
39
40 int find(int x) {
41     for (int i = 0; i < _size; i++) {
42         if (_data[i] == x) return i;
43     }
44     return -1; // -1 表示没找到
45 }
46
47 void insert(int pos, int e) {
48     if (pos < 0 || pos > _size) throw exception("插入的下标不合法");
49     if (_size == _capacity) expandArray();
50     for (int i = _size; i > pos; i--) { //往后腾出一个位置
51         _data[i] = _data[i - 1];
52     }
53     _data[pos] = e; //放进去
54     _size++; //表的长度+1
55 }
56
57 void push_back(int e) {
58     if (_size == _capacity) expandArray();
59     _data[_size++] = e;
60 }
61
62 void deleteElem(int pos) {
63     if (pos < 0 || pos > _size - 1) throw exception("删除的下标不合法");
64     for (int i = pos; i < _size - 1; i++) {

```

```

65         _data[i] = _data[i + 1];
66     }
67     _size--;
68 }
69
70 int pop_back() {    //删除数组最后一个元素，并将其返回
71     if (_size < 1) throw exception("数组为空，不能pop");
72     return _data[--_size];
73 }
74
75 bool empty() {
76     return _size == 0;
77 }
78
79 void clear() {
80     _size = 0;
81 }
82
83 int size() {
84     return _size;
85 }
86
87 int capacity() {
88     return _capacity;
89 }
90
91 void printAll() {    //输出整个数组
92     for (int i = 0; i < _size; i++) {
93         printf("%d ", _data[i]);
94     }
95     printf("\n");
96 }
97
98 //对数组下标lo到hi（不包括hi）的元素反转，左闭右开[lo, hi)
99 void reverse(int lo, int hi) {
100     for (int i = lo, j = hi - 1; i < j; i++, j--) {
101         swap(_data[i], _data[j]);
102     }
103 }
104
105 //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
106 void reverse() {
107     for (int i = 0, j = _size - 1; i < j; i++, j--) {
108         swap(_data[i], _data[j]);
109     }
110 }
111
112 void bubbleSort() {
113     for (int t = _size; t > 0; t--) {
114         bool sorted = true;
115         for (int i = 1; i < t; i++) {
116             if (_data[i - 1] > _data[i]) {
117                 swap(_data[i - 1], _data[i]);

```

```

118         sorted = false;
119     }
120 }
121     if (sorted) break;
122 }
123 }
124
125 //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
126 int& operator[] (int p) {
127     //return get(p);
128     if (p < 0 || p >= _size) throw exception("访问下标不合法");
129     return _data[p];
130 }
131 };
132
133 int main() {
134     Vector L;
135
136     //L.pop_back();
137     try {
138         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
139     } catch (exception e) {    //体验一下异常处理 (仅作了解)
140         printf("%s\n", e.what()); //打印异常e的信息
141     }
142
143     for (int i = 1; i <= 6; i++) {
144         L.push_back(i);
145     }
146     L.pop_back();
147     L.printAll();
148
149     L.insert(2, 10);
150     L.printAll();
151
152     L.insert(1, 20);
153     L.printAll();
154
155     L.deleteElem(3);
156     L.printAll();
157
158     L.reverse();
159     L.printAll();
160
161     L.bubbleSort();
162     L.printAll();
163
164     //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
165     printf("L[2] = %d\n", L[2]);
166     L[1] = 100;
167     L.printAll();
168     return 0;
169 }

```

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。

```
1 vector<int> v1;           //v1的初始容量为默认值0
2 vector<int> v2(100);      //v2的初始容量为100
3 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

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

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     vector<int> v = { 10,20,30 };
7     printf("capacity: %d\n", v.capacity());
8     printf("size: %d\n", v.size());
9
10    v.reserve(100);
11    printf("capacity: %d\n", v.capacity());
12    printf("size: %d\n", v.size()); //注意区分一下vector的capacity和size
13
14    for (int i = 1; i <= 5; i++) {
15        v.push_back(i);
16    }
17    v.pop_back();
18
19    for (int i = 0; i < v.size(); i++) {
20        printf("%d ", v[i]);
21    }
22    printf("\n");
23
24    printf("This vector is empty? %s\n", v.empty() ? "Yes" : "No");
25
26    for (int x : v) { //遍历一个容器（只读）还可以这样操作
27        //增强for循环, C++11语法
28        printf("%d ", x);
29    }
30    printf("\n");
31
32    for (int &x : v) { //如果想修改的话还得这样操作，需要加个引用&
33        x = 100;
34    }
```

```

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

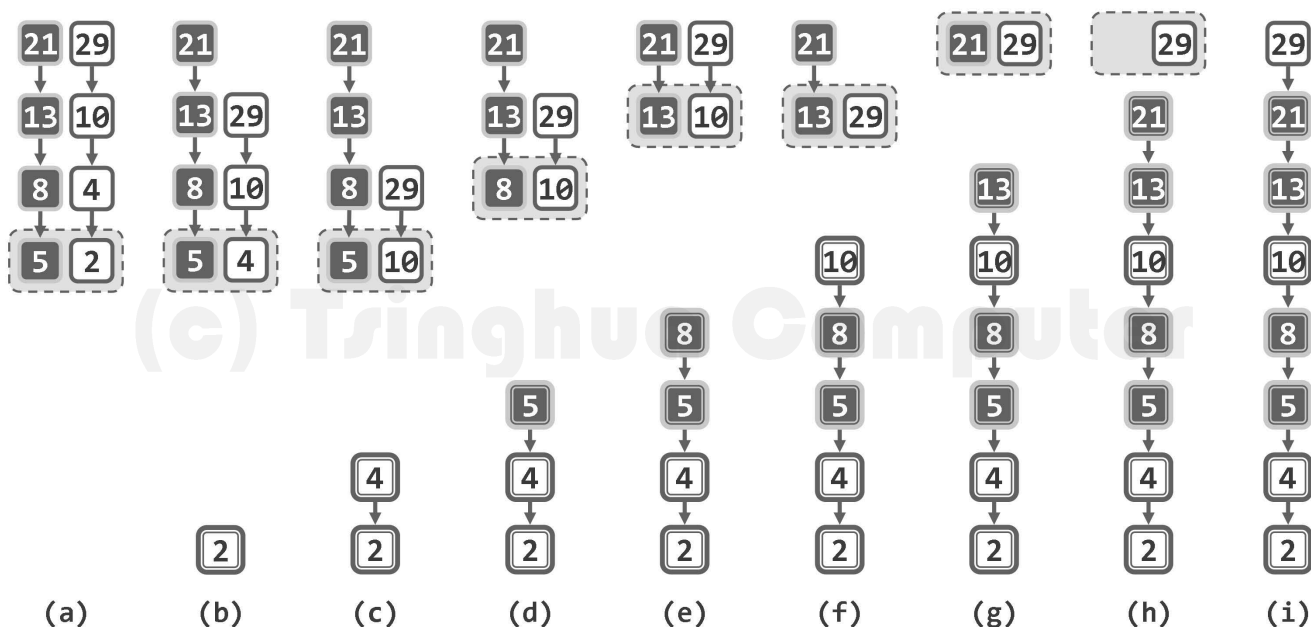
在数组A[]中，表中元素存放在数组下标 $0 \sim m+n-1$ 中，且数组前m个元素递增有序，后n个元素递增有序。设计一个算法，使得数组整体有序。

时间复杂度 $O(n^2 + m * n)$

空间复杂度 $O(1)$

```
1  #include <stdio.h>
2
3  void AdjustArray(int A[], int M, int N) {
4      for (int k = M; k < M + N; k++) {
5          int temp = A[k];
6          int i;
7          for (i = k - 1; i >= 0 && A[i] > temp; i--) {
8              A[i + 1] = A[i];
9          }
10         A[i + 1] = temp;
11     }
12 }
13
14 int main() {
15     int A[] = { 2,5,10,12,18,19,1,2,3,6,9,11,20 };
16     int M = 6, N = 7;
17     AdjustArray(A, M, N);
18     for (int i = 0; i < M + N; i++) {
19         printf("%d ", A[i]);
20     }
21     printf("\n");
22     return 0;
23 }
```

归并排序思想：合并两个有序数组



时间复杂度 $O(m + n)$

```

1  #include <stdio>
2
3  int* mergeSortedArray(int A[], int m, int B[], int n) {
4      int* C = new int[m + n];
5      int i = 0, j = 0;
6      int k = 0;
7      while (i < m && j < n) {
8          if (A[i] < B[j]) {
9              C[k++] = A[i++];
10         } else {
11             C[k++] = B[j++];
12         }
13     }
14     while (i < m) {
15         C[k++] = A[i++];
16     }
17     while (j < n) {
18         C[k++] = B[j++];
19     }
20     return C;
21 }
22
23 int main() {
24     int A[] = { 2,5,10,12,18,19 };
25     int B[] = { 1,2,3,6,9,11,20 };
26     int M = sizeof(A) / sizeof(int);
27     int N = sizeof(B) / sizeof(int);
28     int* C = mergeSortedArray(A, M, B, N);
29     for (int i = 0; i < M + N; i++) {
30         printf("%d ", C[i]);
31     }
32     printf("\n");
33     return 0;
34 }

```

删除无序数组中所有值为x的元素

时间复杂度 $O(n)$

```

1  #include <stdio.h>
2
3  int remove_x_1(int A[], int &N, int x) {
4      //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
5      int cnt = 0;
6      for (int i = 0; i < N; i++) {
7          if (A[i] == x) {
8              cnt++;
9              continue;
10         }
11         A[i - cnt] = A[i]; //当前元素应该往前挪cnt个位置
12     }
13     N -= cnt;
14     return cnt;

```



```

15 }
16
17 int remove_x_2(int A[], int &N, int x) {
18     //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
19     int index = 0;
20     for (int i = 0; i < N; i++) {
21         if (A[i] != x) {
22             A[index++] = A[i]; //当前元素应当放在数组index下标处
23         }
24     }
25     int ret = N - index;
26     N = index;
27     return ret;
28 }
29
30 int main() {
31     int A[] = { 2,3,3,5,-1,3,2,10 };
32     int N = sizeof(A) / sizeof(int);
33     //remove_x_1(A, N, 3);
34     remove_x_2(A, N, 3);
35     printf("N = %d\n", N);
36     for (int i = 0; i < N; i++) {
37         printf("%d ", A[i]);
38     }
39     printf("\n");
40     return 0;
41 }

```

有序数组去重

时间复杂度 $O(n)$

```

1  #include <stdio.h>
2
3  int unique_1(int A[], int &N) {
4      //对长度为N的数组A进行去重，函数返回删除元素的个数
5      if (N == 1) return 0;
6      int cnt = 0;
7      for (int i = 1; i < N; i++) {
8          if (A[i] == A[i - 1]) {
9              cnt++;
10             continue;
11         }
12         A[i - cnt] = A[i]; //当前元素应该往前挪cnt个位置
13     }
14     N -= cnt;
15     return cnt;
16 }
17
18 int unique_2(int A[], int &N) {
19     //对长度为N的数组A进行去重，函数返回删除元素的个数
20     if (N == 1) return 0;
21     int index = 1;

```

```

22     for (int i = 1; i < N; i++) {
23         if (A[i] != A[index - 1]) {
24             A[index++] = A[i]; //当前元素应当放在数组index下标处
25         }
26     }
27     int ret = N - index;
28     N = index;
29     return ret;
30 }
31
32 int main() {
33     int A[] = { 1,1,2,3,3,3,4,4,7,7,7,9,9,9,10 };
34     int N = sizeof(A) / sizeof(int);
35     printf("before unique, N = %d\n", N);
36     //int cnt = unique_1(A, N);
37     int cnt = unique_2(A, N);
38     printf("after unique: cnt = %d, N = %d\n", cnt, N);
39     for (int i = 0; i < N; i++) {
40         printf("%d ", A[i]);
41     }
42     printf("\n");
43     return 0;
44 }

```

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

```

1 void adjustArray_1(int A[], int N) {
2     int* arr = new int[N];
3     int p = 0, q = N - 1;
4     for (int i = 0; i < N; i++) {
5         if (isOddNum(A[i])) {
6             arr[p++] = A[i];
7         } else {
8             arr[q--] = A[i];
9         }
10    }
11    for (int i = 0; i < N; i++) {
12        A[i] = arr[i];
13    }
14    delete[] arr;
15 }

```

方法2:

时间 $O(n)$, 空间 $O(n)$, 稳定

```
1 void adjustArray_2(int A[], int N) {
2     int oddNumCnt = 0;
3     for (int i = 0; i < N; i++) {
4         if (isOddNum(A[i])) oddNumCnt++;
5     }
6     int* arr = new int[N];
7     int p = 0, q = oddNumCnt;
8     for (int i = 0; i < N; i++) {
9         if (A[i] % 2 != 0) {
10             arr[p++] = A[i];
11         } else {
12             arr[q++] = A[i];
13         }
14     }
15     for (int i = 0; i < N; i++) {
16         A[i] = arr[i];
17     }
18     delete[] arr;
19 }
```

方法3:

时间 $O(n)$, 空间 $O(1)$, 不稳定

```
1 void adjustArray_3(int A[], int N) {
2     int i = 0, j = N - 1;
3     while (i < j) {
4         while (i < j && isOddNum(A[i])) i++;
5         while (i < j && isEvenNum(A[j])) j--;
6         /*
7          当算法执行到此时有两种情况:
8          情况1: i指向偶数, j指向奇数, 此时需交换;
9          情况2: i, j均指向同一个元素, 此时应结束算法
10             但对同一个元素执行swap操作也没问题,
11             因此没有对该情况做专门判断
12         */
13         swap(A[i], A[j]);
14     }
15 }
```

方法4:

时间 $O(n^2)$, 空间 $O(1)$, 稳定

```

1 void adjustArray_4(int A[], int N) {
2     int firstEvenPos = -1;
3     for (int i = 0; i < N; i++) {
4         if (isEvenNum(A[i])) {
5             firstEvenPos = i;
6             break;
7         }
8     }
9     if (firstEvenPos == -1) return; //说明数组全是奇数
10
11    for (int i = firstEvenPos + 1; i < N; i++) {
12        if (isOddNum(A[i])) {
13            //说明当前从下标firstEvenPos到i-1全是偶数，下标i为奇数
14            //需要将这些偶数全部往后面挪一个位置，将奇数放进去
15            //并更新firstEvenPos+=1
16            int temp = A[i];
17            for (int j = i; j > firstEvenPos; j--) {
18                A[j] = A[j - 1];
19            }
20            A[firstEvenPos] = temp;
21            firstEvenPos++;
22        }
23    }
24 }

```

完整代码：

```

1 #include <stdio.h>
2 #include <algorithm>
3 using namespace std;
4
5 bool isOddNum(int n) { return n % 2 == 1; }
6 bool isEvenNum(int n) { return n % 2 == 0; }
7
8 void adjustArray_1(int A[], int N) {
9     int* arr = new int[N];
10    int p = 0, q = N - 1;
11    for (int i = 0; i < N; i++) {
12        if (isOddNum(A[i])) {
13            arr[p++] = A[i];
14        } else {
15            arr[q--] = A[i];
16        }
17    }
18    for (int i = 0; i < N; i++) {
19        A[i] = arr[i];
20    }
21    delete[] arr;
22 }
23
24 void adjustArray_2(int A[], int N) {

```

```

25     int oddNumCnt = 0;
26     for (int i = 0; i < N; i++) {
27         if (isOddNum(A[i])) oddNumCnt++;
28     }
29     int* arr = new int[N];
30     int p = 0, q = oddNumCnt;
31     for (int i = 0; i < N; i++) {
32         if (A[i] % 2 != 0) {
33             arr[p++] = A[i];
34         } else {
35             arr[q++] = A[i];
36         }
37     }
38     for (int i = 0; i < N; i++) {
39         A[i] = arr[i];
40     }
41     delete[] arr;
42 }
43
44 void adjustArray_3(int A[], int N) {
45     int i = 0, j = N - 1;
46     while (i < j) {
47         while (i < j && isOddNum(A[i])) i++;
48         while (i < j && isEvenNum(A[j])) j--;
49         /*
50         当算法执行到此时有两种情况:
51         情况1: i指向偶数, j指向奇数, 此时需交换;
52         情况2: i, j均指向同一个元素, 此时应结束算法
53             但对同一个元素执行swap操作也没问题,
54             因此没有对该情况做专门判断
55         */
56         swap(A[i], A[j]);
57     }
58 }
59
60 void adjustArray_4(int A[], int N) {
61     int firstEvenPos = -1;
62     for (int i = 0; i < N; i++) {
63         if (isEvenNum(A[i])) {
64             firstEvenPos = i;
65             break;
66         }
67     }
68     if (firstEvenPos == -1) return; //说明数组全是奇数
69
70     for (int i = firstEvenPos + 1; i < N; i++) {
71         if (isOddNum(A[i])) {
72             //说明当前从下标firstEvenPos到i-1全是偶数, 下标i为奇数
73             //需要将这些偶数全部往后挪一个位置, 将奇数放进去
74             //并更新firstEvenPos+=1
75             int temp = A[i];
76             for (int j = i; j > firstEvenPos; j--) {
77                 A[j] = A[j - 1];

```

```
78         }
79         A[firstEvenPos] = temp;
80         firstEvenPos++;
81     }
82 }
83
84
85 int main() {
86     int A[] = { 1,2,2,2,1,1,2,2 };
87     int N = sizeof(A) / sizeof(int);
88
89     //adjustArray_1(A, N);
90     //adjustArray_2(A, N);
91     //adjustArray_3(A, N);
92     adjustArray_4(A, N);
93
94     for (int i = 0; i < N; i++) {
95         printf("%d ", A[i]);
96     }
97     printf("\n");
98     return 0;
99 }
```