W11、12笔记

搜索 Search

- 在一个数组中找到某个数的位置（或确认是否存在）
- 基本方法：遍历

线性搜索 Linear Search

The search starts at the beginning of the array and goes straight down the line of elements until it finds a match or reaches the end of the array.

```
int search(int key,int a[],int len){
    int ret=-1;
    for(int i=0;i<len;i++){
        if(a[i]==key){
            ret=i;
            break;
        }
    }
    return ret;
}
int main(void){
    int a[]={1,3,5,7,9,11,13,15,19,21};
    int loc=search(15,a,sizeof(a)/sizeof(a[0]));
    printf("%d\n",loc);
    loc=search(14,a,sizeof(a)/sizeof(a[0]));
    printf("%d\n",loc);
}
```

缺点

- 普通的线性搜索算法只能够返回匹配的第一个位置，当列表中有多个数据匹配时就需要对代码进行更改。
- 由于需要逐个进行匹配，所以效率比较低。
- 算法效率不稳定。比如列表第一个数据就是匹配项，和列表无匹配项时所需要的运行时间相差较大。

算法复杂度

- 最优情况下的时间复杂度

  当列表中的第一个数据就是待查找数据时，程序只需要进行2次比较，此时时间复杂度为Θ (1)。

- 最坏情况下的时间复杂度

当数据不在列表中时，就需要对所有的数据进行比较，需要2N+1次比较(N为数组的长度)，此时时间复杂度为Θ (N)。

- 平均情形下的时间复杂度

当列表中的第一个数据就是待查找数据时，程序只需要进行2次比较，而第二个数据是待查找数据时，程序又需要进行2次比较，因此程序平均需要进行(2+4+6+8+...+2*N)/N = N+1次比较。即Θ(N).

二分搜索 Binary Search

二分查找又称折半查找，它是一种效率较高的查找方法。

二分查找要求：线性表是有序表，即表中结点按关键字有序，并且要用向量作为表的存储结构。不妨设有序表是递增有序的。

二分查找只适用顺序存储结构。

基本思想（这里假设数组元素呈升序排列）

将n个元素分成个数大致相同的两半，取a[n/2]与欲查找的x作比较，如果x=a[n/2]则找到x，算法终止；如 果x<a[n/2]，则我们只要在数组a的左半部继续搜索x；如果x>a[n/2]，则我们只要在数组a的右半部继续搜索x。

Steps

- test the value in the middle
- search the lower part if middle > target
- search the upper part if middle < target

```
int search(int key,int array[],int left,int right){
    int ret =-1;
    int mid=(left+right)/2;
    if(key>array[mid]){
        ret=search(key,array,mid+1,right);
    }else if(key<array[mid]){
        ret=search(key,array,left,mid-1);
    }else{
        ret =mid;
    }
    return ret;
}
```

```
int search(int key,int array[],int left,int right){
    int ret=1;
    while(left<=right){
        int mid=(left+right)/2;
        if(key>array[mid]){
            left=mid+1;
        }else if(key<array[mid]){
            right=mid-1;
        }else{
            ret=mid;
            break;
        }
    }
```

```
    return ret;
}
```

算法复杂度

To search an array of N elements requires N comparisons if you use linear search and log2N comparisons if you use binary search. 时间复杂度可以表示O(h)=O(log2n)

排序 Sort

选择排序

选择排序是一种简单和直观的排序算法

工作原理

第一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零。

- Test bed

```
#define SIZE 100

void sort(int a[], int begin, int end);
int main(){
    int a[SIZE];
    srand(0);
    for ( int i=0; i<SIZE; i++ ) {
        a[i] = rand()%SIZE;
        printf("%d\n", a[i]);
    }
    sort(a, 0, SIZE-1);
    for ( int i=0; i<SIZE; i++ ) {
        printf("%d\n", a[i]);
    }
}
```

- the sort

```
void sort(int a[], int begin, int end){
    if ( end > begin ) {
        int loc = findmin(a, begin, end);
        int t = a[begin];
        a[begin] = a[loc];
        a[loc] = t;
        sort(a, begin+1, end);
    }
}
```

- find the min

```
int findmin(int a[], int begin, int end){
    int loc = begin;
    for ( int i=begin+1; i<=end; i++ ) {
        if ( a[i] < a[loc] ) {
            loc = i;
        }
    }
    return loc;
}
```

- Tail Recursive

```
void sort(int a[], int begin, int end){
    if ( end > begin ) {
        int loc = findmin(a, begin, end);
        int t = a[begin];
        a[begin] = a[loc];
        a[loc] = t;
        sort(a, begin+1, end);
    }
}

void sort_it(int a[], int begin, int end){
    while ( end > begin ) {
        int loc = findmin(a, begin, end);
        int t = a[begin];
        a[begin] = a[loc];
        a[loc] = t;
        begin++;
    }
}
```

- Put Them Together

```
void sort_common(int a[], int size){
    for ( int i=0; i<size; i++ ) {
        int minloc = i;
        for ( int j=i+1; j<size; j++ ) {
            if ( a[j] < a[minloc] ) {
            minloc = j;
            }
        }
        int t= a[i];
        a[i] = a[minloc];
        a[minloc] = t;
    }
}
```

时间复杂度

选择排序的交换操作介于 0 和 (n - 1） 次之间。选择排序的比较操作为 n (n - 1） / 2 次之间。选择排序的赋值操作介于 0 和 3 (n - 1） 次之间。比较次数O(n^2），比较次数与关键字的初始状态无关，总的比较次数N=(n-1）+(n-2）+...+1=n*(n-1）/2。交换次数O(n），最好情况是，已经有序，交换0次；最坏情况交换n-1次，逆序交换n/2次。交换次数比冒泡排序少多了，由于交换所需CPU时间比比较所需的CPU时间多，n值较小时，选择排序比冒泡排序快。其中直接选择排序的时间复杂度为O（n*n），空间复杂度为O（1）。树形选择排序的时间复杂度为O（nlog2n），空间复杂度为O（n）。堆排序的平均时间复杂度为O（nlog2n），效率高，但是实现相对复杂，空间代价为O（1）。

稳定性

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第n-1个元素，第n个元素不用选择了，因为只剩下它一个最大的元素了。那么，在一趟选择，如果一个元素比当前元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么交换后稳定性就被破坏了。举个例子，序列5 8 5 2 9，我们知道第一遍选择第1个元素5会和2交换，那么原序列中两个5的相对前后顺序就被破坏了，所以选择排序是一个**不稳定**的排序算法 。

优缺点

优点：一轮只换一次位置

缺点：效率低，不稳定

冒泡排序 Bubble Sort

重复地走访过要排序的元素列，依次比较两个相邻的元素，如果顺序（如从大到小、首字母从Z到A）错误就把他们交换过来。走访元素的工作是重复地进行直到没有相邻元素需要交换，也就是说该元素列已经排序完成。

工作原理

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
void sort_bubble(int a[], int size){
    for ( int i=size-1; i>0; i-- ) {
        for ( int j=0; j<i; j++ ) {
            if ( a[j] > a[j+1] ) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
    }
}
```

- Optimize

```
void sort_bubble(int a[], int size){
    for ( int i=size-1; i>0; i-- ) {
        int loc = -1;
        for ( int j=0; j<i; j++ ) {
            if ( a[j] > a[j+1] ) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                loc = j;
            }
        }
        i = loc+1;
    }
```

```
}
```

时间复杂度

若文件的初始状态是正序的，一趟扫描即可完成排序。所需的关键字比较次数C和记录移动次数M均达到最小值：$C_{\min}=n-1$，$M_{\min}=0$。[1]

所以，冒泡排序最好的时间复杂度为$O(n)$。

若初始文件是反序的，需要进行$n-1$趟排序。每趟排序要进行$n-i$次关键字的比较(1≤i≤n-1)，且每次比较都必须移动记录三次来达到交换记录位置。在这种情况下，比较和移动次数均达到最大值：[1]

$$C_{\max}=\frac{n(n-1)}{2}=O(n^2)$$

$$M_{\max}=\frac{3n(n-1)}{2}=O(n^2)$$

冒泡排序的最坏时间复杂度为$O(n^2)$。[1]

综上，因此冒泡排序总的平均时间复杂度为$O(n^2)$。[1]

稳定性

冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，是不会再交换的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种**稳定**排序算法。

优缺点

优点：比较简单、空间复杂度低、稳定

缺点：时间复杂度高、效率低