排序算法代码总结

选择排序

从0-n-1遍历，每次选择一个最小的元素并交换。

```
void select_sort(int* array, int len) {
    int index, i, j;
    for (i = 0; i < len - 1; ++i) {
        index = i;
        for (j=i+1;j < len; ++j) {
            if (array[j] < array[index]) {
                index = j;
            }
        }
        if (index != i) {
            int temp = array[index];
            array[index] = array[i];
            array[i] = temp;
        }
    }
}
```

插入排序

从i=1-n-1遍历，每次向前找插入点，并向后移动元素；

```
void insertion_sort(int* array, int len) {
    for (int i = 1; i < len; ++i) {
        int j = i - 1; //j每次都要重新赋值
        int temp = array[i];
        while ((j >= 0) && (array[j] > temp)) {
            array[j+1] = array[j];
            j--;
        }
        if (j != (i-1))
            array[j+1] = temp;
    }
}
```

冒泡排序

相邻元素两两比较交换；
优化点：

1. 如果某次遍历没有发生交换，那么已经排序好；
2. 每次并不需要遍历到n-1，只需要遍历到上次发生交换的位置-1即可。

```
void buddle_sort(int* array, int len) {
    int temp, max_index, j;
    int flag = len;
    while (flag > 0) {
            max_index = flag;
            flag = 0;
            for (j = 1; j < max_index; ++j) {
                if (array[j-1] > array[j]) {
                    temp = array[j-1];
                    array[j-1] = array[j];
                    array[j] = temp;
                    flag = j;
                }
            }
    }
}
```

合并排序/归并排序

采用分而治之的思想，先排序好左右两边，然后合并两个已排序好的数组

```
void merge_sort(int* array, int begin, int end) {  //0~n-1
    if (begin >= end) return;
    int middle = (int)((end+begin)/2.0);
    merge_sort(array, begin, middle);
    merge_sort(array, middle+1, end);
    merge_no_guard(array, begin, middle, end);
}

void merge_no_guard(int* array, int begin, int middle, int end) {
    if (begin >= end) return;
    int left_len = middle - begin + 1;
    int right_len = end - middle;
    int* left = new int[left_len];
    int* right = new int[right_len];
    int i = 0;
    for (i = 0; i < left_len; ++i) {
        left[i] = array[begin+i];
    }
```

```
    for (i = 0; i < right_len; ++i) {
        right[i] = array[middle+1+i];
    }
    int k = 0;
    int l = 0;
    i = begin;
    while (k < left_len && l < right_len) {
        if (left[k]<right[l]) {
            array[i] = left[k];
            k++;
        } else {
            array[i] = right[l];
            l++;
        }
        i++;
    }
    if (k != left_len) {
        for (l = k; l < left_len; ++l) {
            array[begin+i] = left[l];
            i++;
        }
    }
    if (l != right_len) {
        for (k = l; k < right_len; ++k) {
            array[begin+i] = right[k];
            i++;
        }
    }
    delete left;
    delete right;
}
```

快速排序

```
void quick_sort(int* array, int begin, int end) {  //0~n-1
    if (begin >= end) return;
    int i = quick_base(array, begin, end);
    quick_sort(array, begin, i-1);
    quick_sort(array, i+1, end);
}

int quick_base(int* array, int begin, int end) {
    int i = begin;
    int j = end;
    int base = array[i];
    while (i < j) {
        /*从右向左查找比基准小的数,并交换到i处*/
        while (i < j && array[j] >= base) j--;
```

```
        if (i < j) {
            array[i] = array[j];
            i++;
        }

        /*从左向右查找比基准大的数,并交换到j处*/
        while (i < j && array[i] < base) i++;
        if (i < j) {
            array[j] = array[i];
            j--;
        }
    }

    array[i] = base;

    return i;
}
```

shell排序

先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前两种方法有较大提高

```
void shell_sort(int* array, int len) {
    int gap;
    for (gap = len / 2; gap >0; gap /= 2) {
        for (int i = gap; i < len; ++i) {
            if (array[i] < array[i-gap]) {
                int temp = array[i];
                int j = i - gap;
                while ((j >= 0) && (array[j] > temp)) {
                    array[j+gap] = array[j];
                    j -= gap;
                }
                array[j+gap] = temp;
            }
        }
    }
}
```

计数排序

使用counts数组保存每个元素出现的次数，然后counts中计算小于等于某个元素的数据量，最后根据counts数组，将排序结果保存到temp中；

```
void count_sort(int* array, int len, int range) {
    int* temp_array = new int[len];
    int* counts = new int[range];
    for (int i = 0; i < range; ++i) {
        counts[i] = 0;
    }
    for (int i = 0; i < len; ++i) {
        counts[array[i]]++;
    }
    for (int i = 1; i < range; ++i) {
        counts[i] = counts[i] + counts[i-1];
    }
    for (int i = len - 1; i >=0 ; --i) {
        temp_array[counts[array[i]] - 1] = array[i];
        counts[array[i]]--;
    }
    for (int i = 0; i < len; ++i) {
        array[i] = temp_array[i];
    }
    if (temp_array) delete temp_array;
    if (counts) delete counts;
}
```

堆排序

```
/*最大堆的下降函数:在左右子数已经为最大堆时,调整根节点,使其形成新的最大堆*/
void maxHeapify(int* array, int len, int i) {
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int largest = i;
    if (left < len && array[left] > array[largest]) {
        largest = left;
    }
    if (right < len && array[right] > array[largest]) {
        largest = right;
    }
    if (largest != i) {
        int temp = array[largest];
        array[largest] = array[i];
        array[i] = temp;
        maxHeapify(array, len, largest);
    }
}

/*建立最大堆*/
void buildMaxHeap(int* array, int len) {
    for (int i = (len/2 - 1); i >= 0; --i) {
        maxHeapify(array, len, i);
```

```
    }
}

/*最大堆排序算法*/
void maxHeapSort(int* array, int len) {
    buildMaxHeap(array, len);
    cout << "maxHeap:";
    printArray(array, len);
    int temp;
    int heapsize = len;
    for (int i = len - 1; i >=0; --i) {
        temp = array[0];
        array[0] = array[i];
        array[i] = temp;
        heapsize--;
        maxHeapify(array, heapsize, 0);
    }
}
```