

各种排序算法总结和比较

排序算法可以说是一项基本功，解决实际问题中经常遇到，针对实际数据的特点选择合适的排序算法可以使程序获得更高的效率，有时候排序的稳定性还是实际问题中必须考虑的，这篇博客对常见的排序算法进行整理，包括：插入排序、选择排序、冒泡排序、快速排序、堆排序、归并排序、希尔排序、二叉树排序、计数排序、桶排序、基数排序。

代码都经过了CodeBlocks的调试，但是很可能有没注意到的BUG，欢迎指出。

比较排序和非比较排序

常见的排序算法都是比较排序，非比较排序包括计数排序、桶排序和基数排序，非比较排序对数据有要求，因为数据本身包含了定位特征，所有才能不通过比较来确定元素的位置。

比较排序的时间复杂度通常为 $O(n^2)$ 或者 $O(n\log n)$ ，比较排序的时间复杂度下界就是 $O(n\log n)$ ，而非比较排序的时间复杂度可以达到 $O(n)$ ，但是都需要额外的空间开销。

比较排序时间复杂度为 $O(n\log n)$ 的证明：

$a_1, a_2, a_3, \dots, a_n$ 序列的所有排序有 $n!$ 种，所以满足要求的排序 $a_1', a_2', a_3', \dots, a_n'$ (其中 $a_1' \leq a_2' \leq a_3' \leq \dots \leq a_n'$) 的概率为 $1/n!$ 。基于输入元素的比较排序，每一次比较的返回不是0就是1，这恰好可以作为决策树的一个决策将一个事件分成两个分支。比如冒泡排序时通过比较 a_1 和 a_2 两个数的大小可以把序列分成 a_1, a_2, \dots, a_n 与 a_2, a_1, \dots, a_n (气泡 a_2 上升一个身位) 两种不同的结果，因此比较排序也可以构造决策树。根节点代表原始序列 $a_1, a_2, a_3, \dots, a_n$ ，所有叶子节点都是这个序列的重排 (共有 $n!$ 个，其中有一个就是我们排序的结果 $a_1', a_2', a_3', \dots, a_n'$)。如果每次比较的结果都是等概率的话 (恰好划分为概率空间相等的两个事件)，那么二叉树就是高度平衡的，深度至少是 $\log(n!)$ 。

又因为 $1 \cdot n! < n^n$ ，两边取对数就得到 $\log(n!) < n\log(n)$ ，所以 $\log(n!) = O(n\log n)$ 。

$2 \cdot n! = n(n-1)(n-2)(n-3)\dots 1 > (n/2)^{(n/2)}$ 两边取对数得到 $\log(n!) > (n/2)\log(n/2) = \Omega(n\log n)$ ，所以 $\log(n!) = \Omega(n\log n)$ 。

因此 $\log(n!)$ 的增长速度与 $n\log n$ 相同，即 $\log(n!) = \Theta(n\log n)$ ，这就是通用排序算法的最低时间复杂度 $O(n\log n)$ 的依据。

排序的稳定性和复杂度

不稳定：

选择排序 (selection sort) — $O(n^2)$

快速排序 (quicksort) — $O(n\log n)$ 平均时间, $O(n^2)$ 最坏情况; 对于大的、乱序序列一般认为是最快的已知排序

堆排序 (heapsort) — $O(n\log n)$

希尔排序 (shell sort) — $O(n\log n)$

基数排序 (radix sort) — $O(n \cdot k)$; 需要 $O(n)$ 额外存储空间 (K为特征个数)

稳定：

插入排序 (insertion sort) — $O(n^2)$

冒泡排序 (bubble sort) — $O(n^2)$

归并排序 (merge sort) — $O(n \log n)$; 需要 $O(n)$ 额外存储空间

二叉树排序 (Binary tree sort) — $O(n\log n)$; 需要 $O(n)$ 额外存储空间

计数排序 (counting sort) — $O(n+k)$; 需要 $O(n+k)$ 额外存储空间, k 为序列中 $\text{Max}-\text{Min}+1$

桶排序 (bucket sort) — $O(n)$; 需要 $O(k)$ 额外存储空间

每种排序的原理和实现

插入排序

遍历数组, 遍历到 i 时, $a_0, a_1 \dots a_{i-1}$ 是已经排好序的, 取出 a_i , 从 a_{i-1} 开始向前和每个比较大小, 如果小于, 则将此位置元素向后移动, 继续先前比较, 如果不小于, 则放到正在比较的元素之后。可见相等元素比较是, 原来靠后的还是拍在后边, 所以插入排序是稳定的。

当待排序的数据基本有序时, 插入排序的效率比较高, 只需要进行很少的数据移动。



```
void insertion_sort (int a[], int n) {
    int i, j, v;
    for (i=1; i<n; i++) {
        //如果第i个元素小于第j个, 则第j个向后移动
        for (v=a[i], j=i-1; j>=0&&v<a[j]; j--)
            a[j+1]=a[j];
        a[j+1]=v;
    }
}
```



选择排序

遍历数组, 遍历到 i 时, $a_0, a_1 \dots a_{i-1}$ 是已经排好序的, 然后从 i 到 n 选择出最小的, 记录下位置, 如果不是第 i 个, 则和第 i 个元素交换。此时第 i 个元素可能会排到相等元素之后, 造成排序的不稳定。



```
void selection_sort (int a[], int n) {
    int i, j, pos, tmp;
    for (i=0; i<n-1; i++) {
        //寻找最小值的下标
        for (pos=i, j=i+1; j<n; j++)
            if (a[pos]>a[j])
                pos=j;
        if (pos != i) {
            tmp=a[i];
            a[i]=a[pos];
            a[pos]=tmp;
        }
    }
}
```




冒泡排序


冒泡排序的名字很形象, 实际实现是相邻两节点进行比较, 大的向后移一个, 经过第一轮两两比较和移动, 最大的元素移动到了最后, 第二轮次大的位于倒数第二个, 依次进行。这是最基本的冒泡排序, 还可以进行一些优化。

优化一: 如果某一轮两两比较中没有任何元素交换, 这说明已经都排好序了, 算法结束, 可以使用一个Flag做标记, 默认为false, 如果发生交互则置为true, 每轮结束时检测Flag, 如果为true则继续, 如果为false则返回。

优化二：某一轮结束位置为j，但是这一轮的最后一次交换发生在lastSwap的位置，则lastSwap到j之间是排好序的，下一轮的结束点就不必是j--了，而直接到lastSwap即可，代码如下：




```
void bubble_sort (int a[], int n) {
    int i, j, lastSwap, tmp;
    for (j=n-1; j>0; j=lastSwap) {
        lastSwap=0;        //每一轮要初始化为0，防止某一轮未发生交换，lastSwap保留上一轮的值进入死循环
        for (i=0; i<j; i++) {
            if (a[i] > a[i+1]) {
                tmp=a[i];
                a[i]=a[i+1];
                a[i+1]=tmp;
                //最后一次交换位置的坐标
                lastSwap = i;
            }
        }
    }
}
```



快速排序

快速排序首先找到一个基准，下面程序以第一个元素作为基准（pivot），然后先从右向左搜索，如果发现比pivot小，则和pivot交换，然后从左向右搜索，如果发现比pivot大，则和pivot交换，一直到左边大于右边，此时pivot左边的都比它小，而右边的都比它大，此时pivot的位置就是排好序后应该在的位置，此时pivot将数组划分为左右两部分，可以递归采用该方法进行。快排的交流使排序成为不稳定的。




```
int mpartition(int a[], int l, int r) {
    int pivot = a[l];

    while (l<r) {
        while (l<r && pivot<=a[r]) r--;
        if (l<r) a[l++]=a[r];
        while (l<r && pivot>a[l]) l++;
        if (l<r) a[r--]=a[l];
    }
    a[l]=pivot;
    return l;
}

void quick_sort (int a[], int l, int r) {

    if (l < r) {
        int q = mpartition(a, l, r);
        msort(a, l, q-1);
        msort(a, q+1, r);
    }
}
```



堆排序

堆排序是把数组看作堆，第 i 个结点的孩子结点为第 $2*i+1$ 和 $2*i+2$ 个结点（不超出数组长度前提下），堆排序的第一步是建堆，然后是取堆顶元素然后调整堆。建堆的过程是自底向上不断调整达成的，这样当调整某个结点时，其左节点和右结点已经是满足条件的，此时如果两个子结点不需要动，则整个子树不需要动，如果调整，则父结点交换到子结点位置，再以此结点继续调整。

下述代码使用的大顶堆，建立好堆后堆顶元素为最大值，此时取堆顶元素即使堆顶元素和最后一个元素交换，最大的元素处于数组最后，此时调整小了一个长度的堆，然后再取堆顶和倒数第二个元素交换，依次类推，完成数据的非递减排序。

堆排序的主要时间花在初始建堆期间，建好堆后，堆这种数据结构以及它奇妙的特征，使得找到数列中最大的数字这样的操作只需要 $O(1)$ 的时间复杂度，维护需要 $\log n$ 的时间复杂度。堆排序不适宜于记录数较少的文件



```
void heapAdjust(int a[], int i, int nLength)
{
    int nChild;
    int nTemp;
    for (nTemp = a[i]; 2 * i + 1 < nLength; i = nChild)
    {
        // 子结点的位置=2*(父结点位置)+1
        nChild = 2 * i + 1;
        // 得到子结点中较大的结点
        if (nChild < nLength-1 && a[nChild+1] > a[nChild])
            ++nChild;
        // 如果较大的子结点大于父结点那么把较大的子结点往上移动，替换它的父结点
        if (nTemp < a[nChild])
        {
            a[i] = a[nChild];
            a[nChild] = nTemp;
        }
        else
            // 否则退出循环
            break;
    }
}

// 堆排序算法
void heap_sort(int a[], int length)
{
    int tmp;
    // 调整序列的前半部分元素，调整完之后第一个元素是序列的最大的元素
    // length/2-1是第一个非叶节点，此处"/"为整除
    for (int i = length / 2 - 1; i >= 0; --i)
        heapAdjust(a, i, length);
    // 从最后一个元素开始对序列进行调整，不断的缩小调整的范围直到第一个元素
    for (int i = length - 1; i > 0; --i)
    {
        // 把第一个元素和当前的最后一个元素交换，
        // 保证当前的最后一个位置的元素都是在现在的这个序列之中最大的
        /// Swap(&a[0], &a[i]);
        tmp = a[i];
        a[i] = a[0];
        a[0] = tmp;
        // 不断缩小调整heap的范围，每一次调整完毕保证第一个元素是当前序列的最大值
        heapAdjust(a, 0, i);
    }
}
```



归并排序

归并排序是采用分治法 (Divide and Conquer) 的一个非常典型的应用。首先考虑下如何将二个有序数列合并。这个非常简单, 只要从比较二个数列的第一个数, 谁小就先取谁, 取了后就在对应数列中删除这个数。然后再进行比较, 如果有数列为空, 那直接将另一个数列的数据依次取出即可。这需要将待排序序列中的所有记录扫描一遍, 因此耗费 $O(n)$ 时间, 而由完全二叉树的深度可知, 整个归并排序需要进行 $\log n$ 次, 因此, 总的时间复杂度为 $O(n \log n)$ 。

归并排序在归并过程中需 要与原始记录序列同样数量的存储空间存放归并结果, 因此空间复杂度为 $O(n)$ 。

归并算法需要两两比较, 不存在跳跃, 因此归并排序是一种稳定的排序算法。



```
void mergearray(int a[], int first, int mid, int last, int temp[])
{
    int i = first, j = mid + 1;
    int m = mid, n = last;
    int k = 0;

    while (i <= m && j <= n)
    {
        if (a[i] <= a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }

    while (i <= m)
        temp[k++] = a[i++];

    while (j <= n)
        temp[k++] = a[j++];

    for (i = 0; i < k; i++)
        a[first + i] = temp[i];
}

void merge_sort(int a[], int first, int last, int temp[])
{
    if (first < last)
    {
        int mid = (first + last) / 2;
        merge_sort(a, first, mid, temp);    //左边有序
        merge_sort(a, mid + 1, last, temp); //右边有序
        mergearray(a, first, mid, last, temp); //再将二个有序数列合并
    }
}
```



有的地方看到在mergearray()合并有序数列时分配临时数组, 即每一步mergearray的结果存放的一个新的临时数组里, 这样会在递归中消耗大量的空间。因此做出小小的变化。只需要new一个临时数组。后面的操作都共用这一个临时数组。合并完后将临时数组中排好序的部分写回原数组。

归并排序计算时间复杂度时可以很容易的列出递归方程, 也是计算时间复杂度的一种方法。

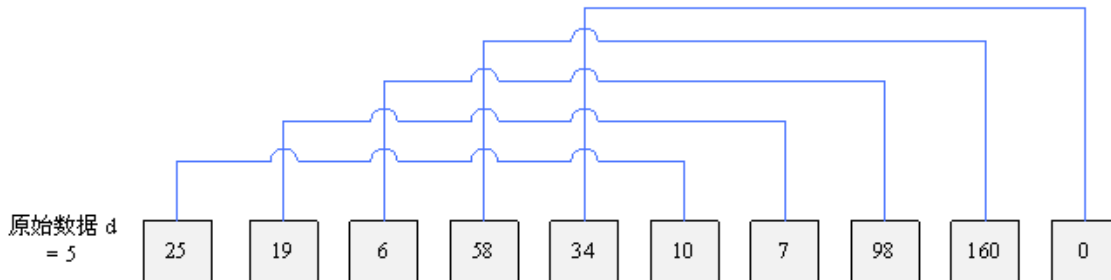
希尔排序

希尔排序是对插入排序的优化，基于以下两个认识：1. 数据量较小时插入排序速度较快，因为 n 和 n^2 差距很小；2. 数据基本有序时插入排序效率很高，因为比较和移动的数据量少。

因此，希尔排序的基本思想是将需要排序的序列划分成为若干个较小的子序列，对子序列进行插入排序，通过则插入排序能够使得原来序列成为基本有序。这样通过对较小的序列进行插入排序，然后对基本有序的数列进行插入排序，能够提高插入排序算法的效率。

希尔排序的划分子序列不是像归并排序那种的二分，而是采用的叫做增量的技术，例如有十个元素的数组进行希尔排序，首先选择增量为 $10/2=5$ ，此时第1个元素和第 $(1+5)$ 个元素配对成子序列使用插入排序进行排序，第2和 $(2+5)$ 个元素组成子序列，完成后增量继续减半为2，此时第1个元素、第 $(1+2)$ 、第 $(1+4)$ 、第 $(1+6)$ 、第 $(1+8)$ 个元素组成子序列进行插入排序。这种增量选择方法的好处是可以使数组整体均匀有序，尽可能的减少比较和移动的次數，二分法中即使前一半数据有序，后一半中如果有比较小的数据，还是会造成大量的比较和移动，因此这种增量的方法和插入排序的配合更佳。

希尔排序的时间复杂度和增量的选择策略有关，上述增量方法造成希尔排序的不稳定性。



```
void shell_sort(int a[], int n)
{
    int d, i, j, temp; //d为增量
    for(d = n/2; d >= 1; d = d/2) //增量递减到1使完成排序
    {
        for(i = d; i < n; i++) //插入排序的一轮
        {
            temp = a[i];
            for(j = i - d; (j >= 0) && (a[j] > temp); j = j - d)
            {
                a[j + d] = a[j];
            }
            a[j + d] = temp;
        }
    }
}
```

二叉树排序

二叉树排序法借助了数据结构二叉排序树，二叉排序数满足三个条件：（1）若左子树不空，则左子树上所有结点的值均小于它的根结点的值；（2）若右子树不空，则右子树上所有结点的值均大于它的根结点的值；（3）左、右子树也分别为二叉排序树。根据这三个特点，用中序遍历二叉树得到的结果就是排序的结果。

二叉树排序法需要首先根据数据构建二叉排序树，然后中序遍历，排序时间复杂度为 $O(n\log n)$ ，构建二叉树需要额外的 $O(n)$ 的存储空间，有相同的元素是可以设置排在后边的放在右子树，在中序变量的时候也会在后边，所以二叉树排序是稳定的。

在实现此算法的时候遇到不小的困难，指针参数在函数中无法通过new赋值，后来采用取指针地址，然后函数设置BST** tree的方式解决。



```
int arr[] = {7, 8, 8, 9, 5, 16, 5, 3, 56, 21, 34, 15, 42};

struct BST{
    int number; //保存数组元素的值
    struct BST* left;
    struct BST* right;
};

void insertBST(BST** tree, int v) {
    if (*tree == NULL) {
        *tree = new BST;
        (*tree)->left=(*tree)->right=NULL;
        (*tree)->number=v;
        return;
    }
    if (v < (*tree)->number)
        insertBST(&((*tree)->left), v);
    else
        insertBST(&((*tree)->right), v);
}

void printResult(BST* tree) {
    if (tree == NULL)
        return;
    if (tree->left != NULL)
        printResult(tree->left);
    cout << tree->number << " ";
    if (tree->right != NULL)
        printResult(tree->right);
}

void createBST(BST** tree, int a[], int n) {
    *tree = NULL;
    for (int i=0; i<n; i++)
        insertBST(tree, a[i]);
}

int main()
{
    int n = sizeof(arr)/sizeof(int);

    BST* root;
    createBST(&root, arr, n);
    printResult(root);
}
```



计数排序

如果通过比较进行排序，那么复杂度的下界是 $O(n\log n)$ ，但是如果数据本身有可以利用的特征，可以不通过比较进行排序，就能使时间复杂度降低到 $O(n)$ 。

计数排序要求待排序的数组元素都是 整数，有很多地方都要去是0-K的正整数，其实负整数也可以通过都加一个偏移量解决的。

计数排序的思想是，考虑待排序数组中的某一个元素a，如果数组中比a小的元素有s个，那么a在最终排好序的数组中的位置将会是s+1，如何知道比a小的元素有多少个，肯定不是通过比较去觉得，而是通过数字本身的属性，即累加数组中最小值到a之间的每个数字出现的次数（未出现则为0），而每个数字出现的次数可以通过扫描一遍数组获得。

计数排序的步骤：

1. 找出待排序的数组中最大和最小的元素（计数数组C的长度为max-min+1，其中位置0存放min，依次填充到最后位置存放max）
2. 统计数组中每个值为 的元素出现的次数，存入数组C的第 项
3. 对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）
4. 反向填充目标数组：将每个元素放在新数组的第C[i]项，每放一个元素就将C[i]减去1（反向填充是为了保证稳定性）

以下代码中寻找最大和最小元素参考编程之美，比较次数为1.5n次。

计数排序适合数据分布集中的排序，如果数据太分散，会造成空间的大量浪费，假设数据为（1,2,3,1000000），这就需要1000000的额外空间，并且有大量的空间浪费和时间浪费。



```
void findArrMaxMin(int a[], int size, int *min, int *max)
{
    if(size == 0) {
        return;
    }
    if(size == 1) {
        *min = *max = a[0];
        return;
    }

    *min = a[0] > a[1] ? a[1] : a[0];
    *max = a[0] <= a[1] ? a[1] : a[0];

    int i, j;
    for(i = 2, j = 3; i < size, j < size; i += 2, j += 2) {
        int tempmax = a[i] >= a[j] ? a[i] : a[j];
        int tempmin = a[i] < a[j] ? a[i] : a[j];

        if(tempmax > *max)
            *max = tempmax;
        if(tempmin < *min)
            *min = tempmin;
    }

    //如果数组元素是奇数个，那么最后一个元素在分组的过程中没有包含其中，
    //这里单独比较
    if(size % 2 != 0) {
        if(a[size - 1] > *max)
            *max = a[size - 1];
        else if(a[size - 1] < *min)
            *min = a[size - 1];
    }
}
```



```
void count_sort(int a[], int b[], int n) {
    int max, min;
    findArrMaxMin(a, n, &min, &max);
    int numRange = max-min+1;
    int* counter = new int[numRange];

    int i, j, k;
    for (k=0; k<numRange; k++)
        counter[k]=0;

    for (i=0; i<n; i++)
        counter[a[i]-min]++;

    for (k=1; k<numRange; k++)
        counter[k] += counter[k-1];

    for (j=n-1; j>=0; j--) {
        int v = a[j];
        int index = counter[v-min]-1;
        b[index]=v;
        counter[v-min]--;
    }
}
```



桶排序

假设有一组长度为N的待排关键字序列K[1.....n]。首先将这个序列划分成M个子区间(桶)。然后基于某种映射函数，将待排序列的关键字k映射到第i个桶中(即桶数组B的下标 i)，那么该关键字k就作为B[i]中的元素(每个桶B[i]都是一组大小为N/M的序列)。接着对每个桶B[i]中的所有元素进行比较排序(可以使用快排)。然后依次枚举输出B[0].....B[M]中的全部内容即是一个有序序列。

桶排序利用函数的映射关系，减少了计划所有的比较操作，是一种Hash的思想，可以用在海量数据处理中。

我觉得计数排序也可以看作是桶排序的特例，数组关键字范围为N，划分为N个桶。

基数排序

基数排序也可以看作一种桶排序，不断的使用不同的标准对数据划分到桶中，最终实现有序。基数排序的思想是对数据选择多种基数，对每一种基数依次使用桶排序。

基数排序的步骤：以整数为例，将整数按十进制位划分，从低位到高位执行以下过程。

1. 从个位开始，根据0~9的值将数据分到10个桶桶，例如12会划分到2号桶中。
2. 将0~9的10个桶中的数据顺序放回到数组中。

重复上述过程，一直到最高位。

上述方法称为LSD (Least significant digital)，还可以从高位到低位，称为MSD。



```
int getNumInPos(int num,int pos) //获得某个数字的第pos位的值
{
    int temp = 1;
    for (int i = 0; i < pos - 1; i++)
        temp *= 10;

    return (num / temp) % 10;
}
```

```
}

#define RADIX_10 10    //十个桶，表示每一位的十个数字
#define KEYNUM 5       //整数位数
void radix_sort(int* pDataArray, int iDataNum)
{
    int *radixArrays[RADIX_10];    //分别为0~9的序列空间
    for (int i = 0; i < RADIX_10; i++)
    {
        radixArrays[i] = new int[iDataNum];
        radixArrays[i][0] = 0;    //index为0处记录这组数据的个数
    }

    for (int pos = 1; pos <= KEYNUM; pos++)    //从个位开始到31位
    {
        for (int i = 0; i < iDataNum; i++)    //分配过程
        {
            int num = getNumInPos(pDataArray[i], pos);
            int index = ++radixArrays[num][0];
            radixArrays[num][index] = pDataArray[i];
        }

        for (int i = 0, j = 0; i < RADIX_10; i++) //写回到原数组中，复位radixArrays
        {
            for (int k = 1; k <= radixArrays[i][0]; k++)
                pDataArray[j++] = radixArrays[i][k];
            radixArrays[i][0] = 0;
        }
    }
}
```