数据结构与算法

Data Structure and Algorithm

极夜酱

目录

1 排序算法 **1**

1.1 希尔排序 1

1.2 归并排序 7

1.3 快速排序 11

1.4 堆排序 18

Chapter 1 排序算法

1.1 希尔排序

1.1.1 希尔排序（Shell Sort）

希尔排序本质上是直接插入排序的升级版。对于插入排序而言，在大多数元素已经有序的情况下，工作量会比较小。这个结论很明显，如果一个数组大部分元素都有序，那么数组中的元素自然不需要频繁地进行比较和交换。

如何能够让待排序的数组中大部分元素有序呢？需要对原始数组进行预处理，使得原始数组的大部分元素变得有序。采用分组的方法，可以将数组进行一定程度地粗略调整。

例如一个有 8 个数字组成的无序序列 $\{5, 8, 6, 3, 9, 2, 1, 7\}$，进行升序排序。让元素两两一组，同组两个元素之间的跨度为数组总长度的一半。

接着让每组元素进行独立排序，排序方式使用直接插入排序即可。由于每一组的元素数量很少，所以插入排序的工作量很少。这样一来，仅仅经过几次简单的交换，数组整体的有序程序得到了显著提高，使得后续再进行直接插入排序的工作量大大减少。

但是这样还不算完，还可以进一步缩小分组跨度，重复上述工作。

例如一个有 8 个数字组成的无序序列 $\{5, 8, 6, 3, 9, 2, 1, 7\}$，进行升序排序。

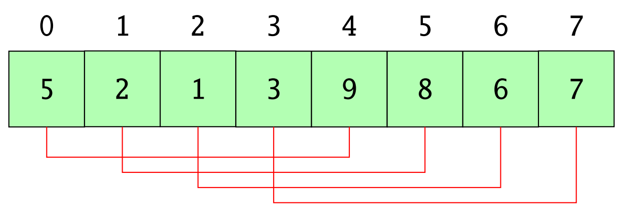

图 1.1: 跨度为 4 分组交换

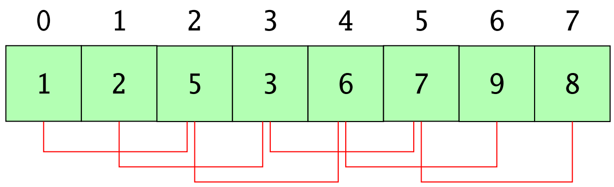

图 1.2: 跨度为 2 分组交换

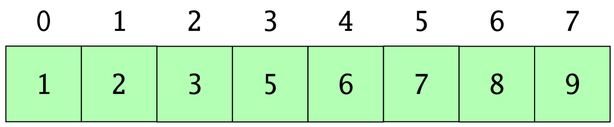

图 1.3: 跨度为 1 分组交换

希尔排序的发明者是计算机科学家 Donald Shell。希尔排序中说使用的分组跨度被称为希尔排序的增量。增量的选择可以有很多种，最朴素的就是 Donald Shell 在发明希尔排序时所提出的逐步折半的方法。

希尔排序

```
void shellSort(int *arr, int n) {
    int gap = n;
    while(gap > 1) {
        gap /= 2;
        for(int i = 0; i < gap; i++) {
            for(int j = i+gap; j < n; j += gap) {
                int temp = arr[j];
                int k = j - gap;
                while(k >= 0 && arr[k] > temp) {
                    arr[k+gap] = arr[k];
                    k -= gap;
                }
                arr[k+gap] = temp;
            }
        }
    }
}
```

1.1.2 算法分析

希尔排序利用分组粗略调整的方式减少了直接插入排序的工作量，使得算法的平均时间复杂度低于 $O(n^2)$。但是在某些极端情况下，希尔排序的最坏时间复杂度仍然是 $O(n^2)$，甚至比插入排序更慢。

例如 $\{2, 1, 5, 3, 7, 6, 9, 8\}$，无论是以 4 为增量，还是以 2 为增量，每组内部的元素都没有任何交换。直到增量缩减为 1，数组才会按照直接插入排序的方式进行调整。

对于这样的数组，希尔排序不但没有减少直接插入排序的工作量，反而白白增加了分组操作的成本。

这是因为每一轮希尔增量之间都是等比的，这就导致了希尔增量存在盲区。为了避免这样的极端情况，科学家发明了许多更为严谨的增量方式。其中最具有代表性的是 Hibbard 增量和 Sedgewick 增量。

Hibbard 增量序列

Hibbard 增量序列为 $1, 3, 7, 15, \ldots$，通项公式为 $2^i - 1$。

利用这种增量方式的希尔排序，最坏时间复杂度是 $O(n^{3/2})$。

Hibbard 增量序列

```python
def get_hibbard_sequence(n):
    """
        生成Hibbard序列
        1, 3, 7, 15, 31, 63, ...
    """
    sequence = []
    i = 1
    while i <= n:
        sequence.append(i)
```

```python
        i = (i << 1) + 1
    sequence.reverse()
    return sequence

def shell_sort_hibbard(lst):
    """
        希尔排序（Hibbard增量序列）
    """
    n = len(lst)
    hibbard = get_hibbard_sequence(n)
    for gap in hibbard:
        for i in range(gap, n):
            j = i
            temp = lst[j]
            while j >= gap:
                if temp < lst[j-gap]:
                    lst[j] = lst[j-gap]
                    j -= gap
                else:
                    break
            lst[j] = temp
```

Sedgewick 增量序列

Sedgewick 增量序列为 $1, 5, 19, 41, 109, \ldots$，通项公式为 $9 \times 4^i - 9 \times 2^i + 1$ 和 $4^{i+2} - 3 \times 2^{i+2} + 1$。

利用这种增量方式的希尔排序，最坏时间复杂度是 $O(n^{4/3})$。

Sedgewick 增量序列

```python
def get_sedgewick_sequence(n):
    """
        生成Sedgewick序列
        1, 5, 19, 41, 109, ...
    """
```

```
    sequence = []
    i = 0
    while True:
        # 9 * 4^i - 9 * 2^i + 1
        # ==> 9 * (2^(2*i) - 2^i) + 1
        item = 9 * ((1 << (2 * i)) - (1 << i)) + 1
        if item <= n:
            sequence.append(item)
        else:
            break

        # 4^(i+2) - 3 * 2^(i+2) + 1
        # ==> 2^(2i+4) - 3 * 2^(i+2) + 1
        item = (1 << (2 * i + 4)) - 3 * (1 << (i + 2)) + 1
        if item <= n:
            sequence.append(item)
        else:
            break

        i += 1
    return sequence

def shell_sort_sedgewick(lst):
    """
        希尔排序（Sedgewick增量序列）
    """
    n = len(lst)
    sedgewick = get_sedgewick_sequence(n)
    for gap in sedgewick:
        for i in range(gap, n):
            j = i
            temp = lst[j]
            while j >= gap:
                if temp < lst[j-gap]:
                    lst[j] = lst[j-gap]
                    j -= gap
                else:
```

```
                break
        lst[j] = temp
```

这两种增量方式的时间复杂度需要很复杂的数学证明，有些是人们的大致猜想。

时间复杂度	**空间复杂度**	**稳定性**
$O(n^{1.3\sim 2})$	$O(1)$	不稳定

表 1.1: 希尔排序算法分析

1.2 归并排序

1.2.1 归并排序优化

简单的归并排序利用分治法，递归地将对小规模子数组进行处理。但是递归会使小规模问题中方法调用太过频繁，因此对于规模较小的子数组可以采用插入排序。一般来说插入排序在小数组中比归并更快，这种优化可以使归并排序的运行时间缩短 $10\% \sim 15\%$。

另一个可以优化的地方是对于单次合并的过程，例如将子数组 $arr[start..mid]$ 和 $arr[mid+1..end]$ 进行合并，如果 $arr[mid] \leq arr[mid+1]$ 的话，说明 $arr[start..end]$ 已经为有序状态，无序再进行不必要的合并。

归并排序优化

```
void mergeSortWorker(int *arr, int start, int end, int *temp) {
    // 列表长度小于10时，采用二分插入排序
    if(end - start <= 10) {
        binaryInsertionSort(arr, start, end);
        return;
    }
    if(start < end) {
        int mid = start + (end - start) / 2;
        mergeSortWorker(arr, start, mid, temp);
        mergeSortWorker(arr, mid+1, end, temp);
        // 避免不必要的合并
        if(arr[mid] <= arr[mid+1]) {
            return;
        }
        merge(arr, start, mid, end, temp);
    }
}
```

1.2.2 归并排序迭代实现

递归实现的归并排序是自顶向下的过程，基于循环的归并排序是自底向上进行的。非递归的归并排序避免了递归时深度为 $logn$ 的栈空间，空间上只用到了长度为 n 的临时空间。

归并排序（迭代）

```java
public static void mergeSort(int[] arr) {
    int n = arr.length;
    int[] temp = new int[n];
    int pos = 0;            // 临时数组的下表
    int left1, left2;       // 左子数组边界
    int right1, right2;     // 右子数组边界

    for (int i = 1; i < n; i *= 2) {
        for (left1 = 0; left1 < n - i; left1 = right2) {
            // 设置子数组边界
            right1 = left2 = left1 + i;
            right2 = left2 + i;

            // 防止右边界越界
            right2 = right2 > n ? n : right2;

            pos = 0;
            while (left1 < left2 && right1 < right2) {
                if (arr[left1] < arr[right1]) {
                    temp[pos++] = arr[left1++];
                } else {
                    temp[pos++] = arr[right1++];
                }
            }

            while (left1 < left2) {
                arr[--right1] = arr[--left2];
            }
```

```

            // 将排好序的部分保存回数组
            while (pos > 0) {
                arr[--right1] = temp[--pos];
            }
        }
    }
}
```

1.2.3 外部排序

在内存中进行的排序称为内部排序，而在许多实际应用中，经常需要对大文件进行排序。因为文件中的信息量庞大，无法将整个文件拷贝进内存进行排序。因此需要将待排序的记录存储在外存上，排序时再把数据一部分一部分调入内存进行排序，再讲排好序的记录写回文件中。

因为磁盘读写的时间远超过内存计算的时间，因此外部排序过程中的时间代价主要是磁盘 I/O 次数。

假如需要对一个包含 40 亿个 int 类型整数的文件进行排序，而计算机的内存只有 2GB。一个 int 占 4 个字节，40 亿个需要 160 亿字节，大概占用 8GB 的内存。因此可以把 8GB 分割成 4 份 2GB 的数据进行排序，然后再把它们凑回去。

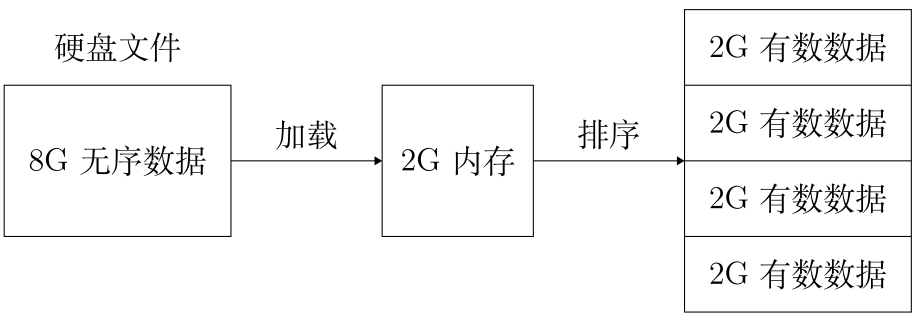

图 1.4: 分割数据

排序的时候可以采用归并排序，每次将两个有序子串合并层一个大的有序子串。

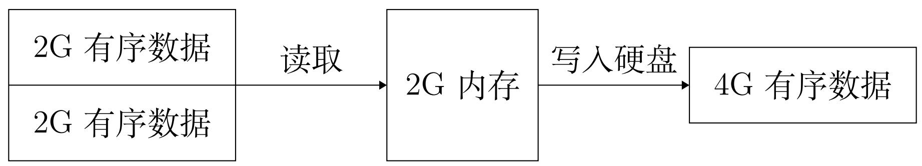

图 1.5: 二路归并

不过硬盘的读写速度比内存要慢得多，通过优化可以降低数据从硬盘读写的次数。

在进行有序数据合并的时候，不采取两两合并的方法，而是可以三组或四组数据一起合并。n 个有序数据的合并被称为 n 路归并。

1.3 快速排序

1.3.1 随机选择基准值

快速排序利用分治法，通过一趟排序将数组分为两部分，其中一部分小于等于基准值，另一部分大于等于基准值，然后再递归对两个子问题排序。

基本的快速排序采用序列的第一个元素作为基准值，但是这不是一种好方法。当数组已经有序时，这样的分割效率非常糟糕。为了缓解这种极端情况，可以在待排序数组中随机选择一个元素作为基准值。

随机选取基准值

```
int selectRandomPivot(int *arr, int start, int end) {
    srand(time(NULL));
    int pos = rand() % (end - start) + start;
    swap(&arr[pos], &arr[start]);
    return arr[start];
}
```

1.3.2 三数取中

虽然随机选取基准值可以减少出现分割不好的几率，但是最坏情况下还是 $O(n)$。另一种选取基准值的方法就是三数取中，也就是取序列中 start、mid、end 三个元素的中间值作为基准值。

三数取中

```
int selectMedianPivot(int *arr, int start, int end) {
    int mid = start + (end - start) / 2;
    if(arr[mid] > arr[end]) {
        swap(&arr[mid], &arr[end]);
    }
```

```c
    if(arr[start] > arr[end]) {
        swap(&arr[start], &arr[end]);
    }
    if(arr[mid] > arr[start]) {
        swap(&arr[mid], &arr[start]);
    }
    // 此时arr[mid] <= arr[start] <= arr[end]
    return arr[start];
}
```

1.3.3 三数取中 + 插入排序

对于很小和部分有序的数组，快速排序的效率不如插入排序。因此当待排序数组被分割到一定大小后，可直接采用插入排序。

三数取中 + 插入排序

```c
void quickSort(int *arr, int start, int end) {
    if(end - start <= 10) {
        binaryInsertionSort(arr, start, end);
        return;
    }

    if(start < end) {
        int i = start;
        int j = end;
        int pivot = selectMedianPivot(arr, start, end);

        while(i < j) {
            while(i < j && arr[j] > pivot) {
                j--;
            }
            if(i < j) {
                arr[i] = arr[j];
                i++;
            }
```

```
            while(i < j && arr[i] < pivot) {
                i++;
            }
            if(i < j) {
                arr[j] = arr[i];
                j--;
            }
        }
        arr[i] = pivot;
        quickSort(arr, start, i-1);
        quickSort(arr, i+1, end);
    }
}
```

1.3.4 聚集相等基准值

在一次分割结束后，可以把所有与基准值相等的元素聚集在一起，这样在下次分割时，就不用对这些值再分割了。

例如待排序序列为 $\{1, 4, 6, 7, 6, 6, 7, 6, 8, 6\}$，选择 6（下标为 4）作为基准值。在进行一次分割后，得到两个子序列 $\{1, 4, 6\}$ 和 $\{7, 6, 7, 6, 8, 6\}$。将所有与基准值相等的元素聚集后，可得到 $\{1, 4, 6, 6, 6, 6, 6, 7, 8, 7\}$。这样下一次分割的子序列可以减少为 $\{1, 4\}$ 和 $\{7, 8, 7\}$。

聚集相等基准值

```
/**
 * @brief  聚集相等基准值
 * @param  arr: 待排序数组
 * @param  start: 数组开始位置
 * @param  end: 数组结束位置
 * @param  pivotPos: 基准值下标
 * @param  left: 相等基准值左边界
 * @param  right: 相等基准值右边界
 */
```

```
void gather(int *arr, int start, int end,
            int pivotPos, int *left, int *right) {
    if(start >= end) {
        return;
    }

    int cnt = pivotPos - 1;
    for(int i = pivotPos - 1; i >= start; i--) {
        if(arr[i] == arr[pivotPos]) {
            swap(&arr[i], &arr[cnt]);
            cnt--;
        }
    }
    *left = cnt;

    cnt = pivotPos + 1;
    for(int i = pivotPos + 1; i <= end; i++) {
        if(arr[i] == arr[pivotPos]) {
            swap(&arr[i], &arr[cnt]);
            cnt++;
        }
    }
    *right = cnt;
}
```

1.3.5 尾递归优化

快速排序在函数尾部有 2 次递归操作，可以对其中的尾递归进行优化。因为在第一次递归后，start 就没用了，第二次递归可以用循环代替。

尾递归优化

```
void quickSort(int *arr, int start, int end) {
    if(end - start <= 10) {
        binaryInsertionSort(arr, start, end);
        return;
```

```
    }

    while(start < end) {
        int i = start;
        int j = end;
        int pivot = selectMedianPivot(arr, start, end);

        while(i < j) {
            while(i < j && arr[j] > pivot) {
                j--;
            }
            if(i < j) {
                arr[i] = arr[j];
                i++;
            }
            while(i < j && arr[i] < pivot) {
                i++;
            }
            if(i < j) {
                arr[j] = arr[i];
                j--;
            }
        }
        arr[i] = pivot;

        // 聚集与基准值相等元素
        int left, right;
        gather(arr, start, end, i, &left, &right);

        quickSort(arr, start, left);
        // quickSort(arr, right, end);  // 消除尾递归
        start = right;
    }
}
```

其实这种优化编译器会自己进行优化，因此相比不使用优化的方法，运行时间几乎无异。

1.3.6 快速排序迭代实现

递归实现主要是在划分子区间，因此可以通过利用栈的特性来保存区间即可，因为递归本身就是压栈的过程。

快速排序（迭代）

```c
int partition(int *arr, int start, int end) {
    int i = start - 1;
    int pivot = arr[end];

    for(int j = start; j < end; j++) {
        if(arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i+1], &arr[end]);
    return i + 1;
}

void quickSort(int *arr, int start, int end) {
    Stack *s = initStack(end - start + 1);
    push(s, start);
    push(s, end);

    while(!isEmptyStack(s)) {
        int right = pop(s);
        int left = pop(s);

        int index = partition(arr, left, right);
        if(index - 1 > left) {
            push(s, left);
            push(s, index - 1);
        }
```

```
        if(index + 1 < right) {
            push(s, index + 1);
            push(s, right);
        }
    }
}
```

1.4 堆排序

1.4.1 堆（Heap）

二叉堆本质上是一种完全二叉树，分为最大堆和最小堆两个类型。在最大堆中，任何一个父结点的值都大于等于它左右孩子结点的值。在最小堆中，任何一个父结点的值都小于等于它左右孩子结点的值。