

02-F1 归并排序

#数据结构邓神

归并排序：构思

- C.B.A(基于比较多排序算法) 都存在一个时间下界 $\Omega(n \log n)$

是否有一个算法在最坏的情况下也只需要 $n \log n$ 时间呢？

归并排序



//分治策略

//向量与列表通用

//J. von Neumann, 1945

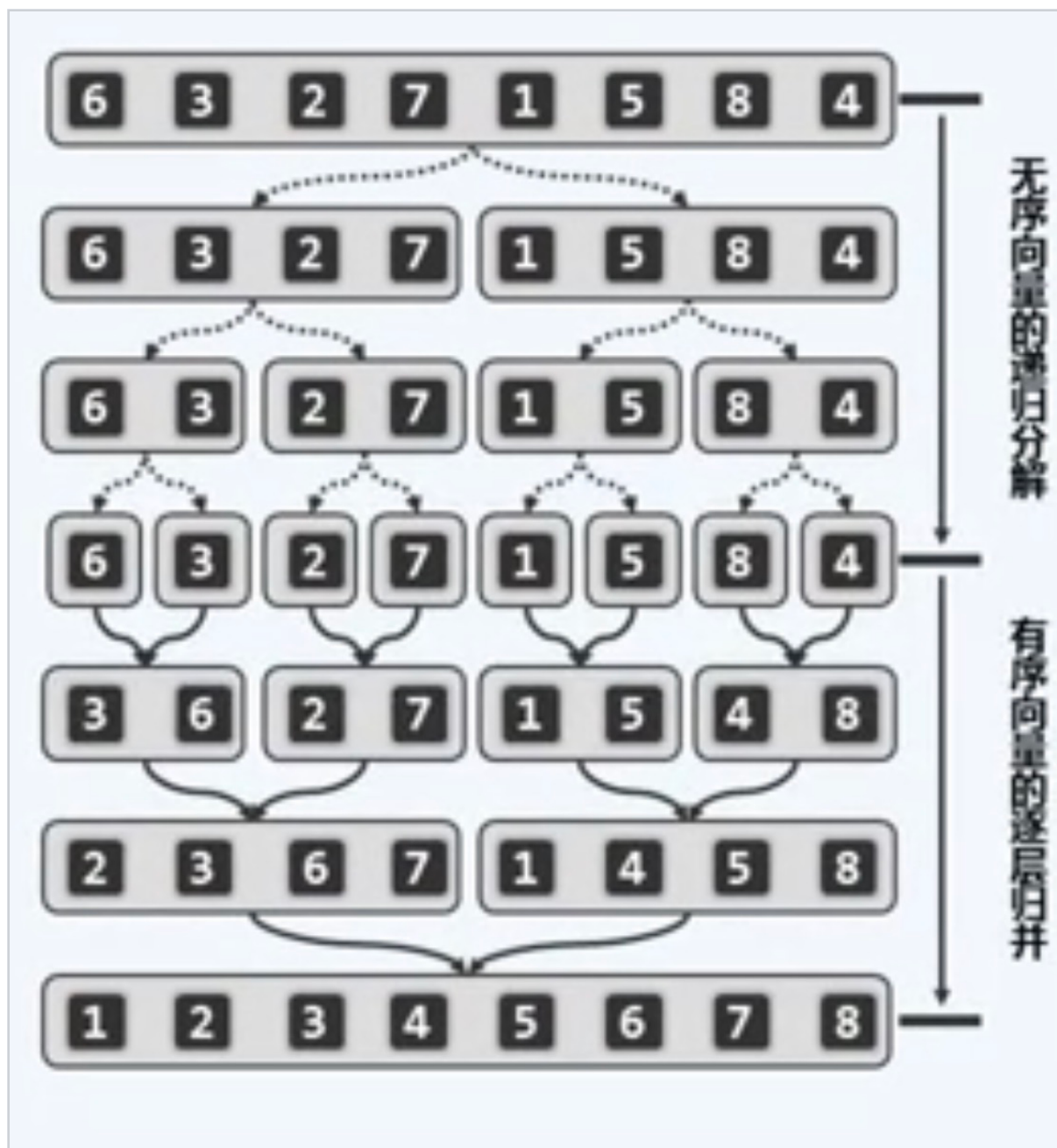
序列一分为二 // $O(1)$

子序列递归排序 // $2 \times T(n/2)$

合并有序子序列 // $O(n)$



归并排序图解



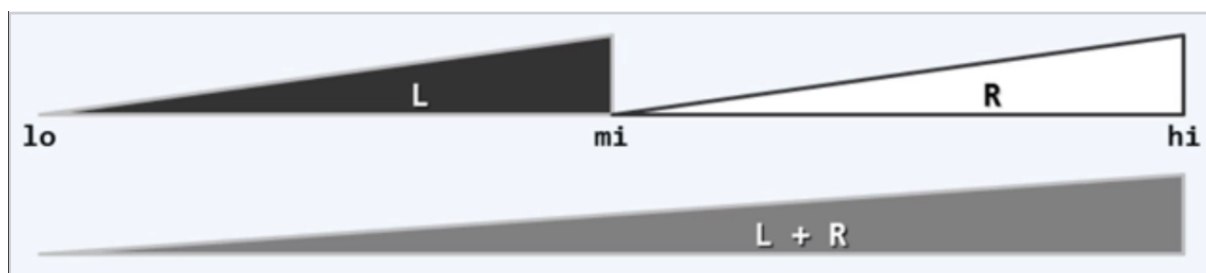
如果真能如此，时间复杂度就应该是 $O(n \log n)$

那么如何进行呢

我们首先来看分

这个应该是递归的问题，应该不难

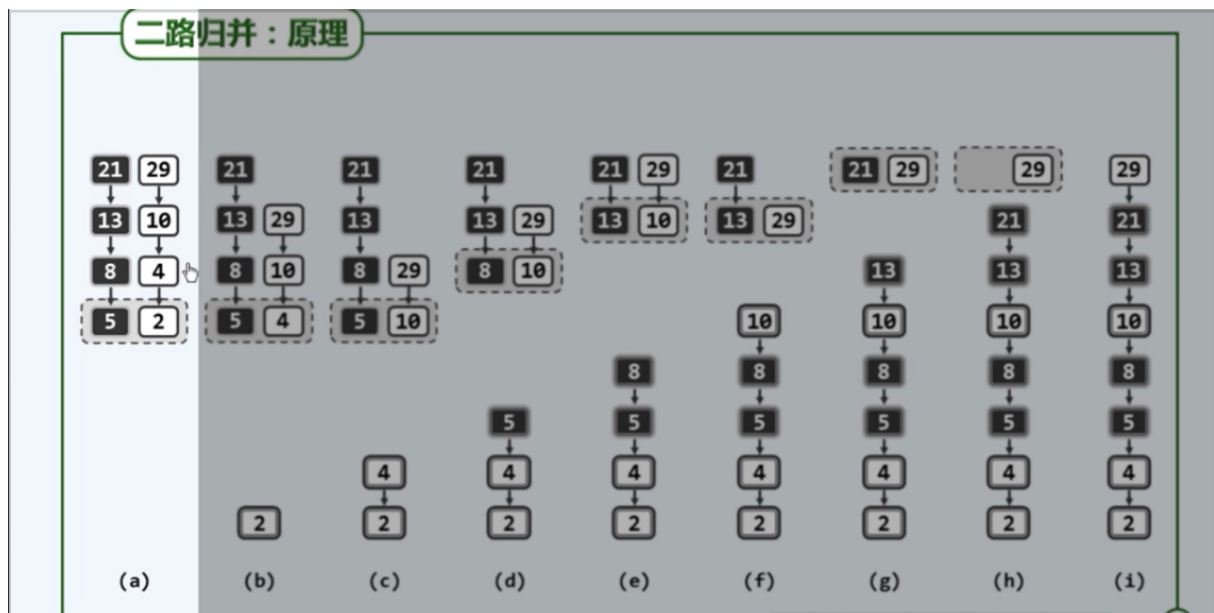
所以主要问题在于如何把两个有序的序列归并成一个有序的序列



归并排序: 主要算法

```
template <typename T> void Vector<T>::mergeSort(Rank lo, Rank hi){
    // 递归基
    if(lo - hi < 2) {
        return; // 一个元素必然有序
    }
    int mi = (lo + hi) >> 1; // 中点为界限
    mergeSort(lo, mi); // 对前半段进行归并排序
    mergeSort(mi, hi); // 对后半段进行归并排序
    merge(lo, mi, hi); // 归并
}
```

归并算法：实例



两列代表两组有序的序列

我们每次从灰色框框中取出更小的元素

所以一共只需要比较 \minLen 次

二路归并算法：基本实现

二路归并：基本实现

```
template <typename T> void Vector<T>::merge(Rank lo, Rank mi, Rank hi) {
    T* A = _elem + lo; //合并后的向量A[0, hi - lo) = _elem[lo, hi)
    int lb = mi - lo; T* B = new T[lb]; //前子向量B[0, lb) = _elem[lo, mi)
    for (Rank i = 0; i < lb; B[i] = A[i++]); //复制前子向量B
    int lc = hi - mi; T* C = _elem + mi; //后子向量C[0, lc) = _elem[mi, hi)
    for (Rank i = 0, j = 0, k = 0; (j < lb) || (k < lc); ) { //B[j]和C[k]中小者转至A的末尾
        if ( (j < lb) && (lc <= k || (B[j] <= C[k])) ) A[i++] = B[j++]; //C[k]已无或不小
        if ( (k < lc) && (lb <= j || (C[k] < B[j])) ) A[i++] = C[k++]; //B[j]已无或更大
    } //该循环实现紧凑；但就效率而言，不如拆分处理
    delete [] B; //释放临时空间B
}
```

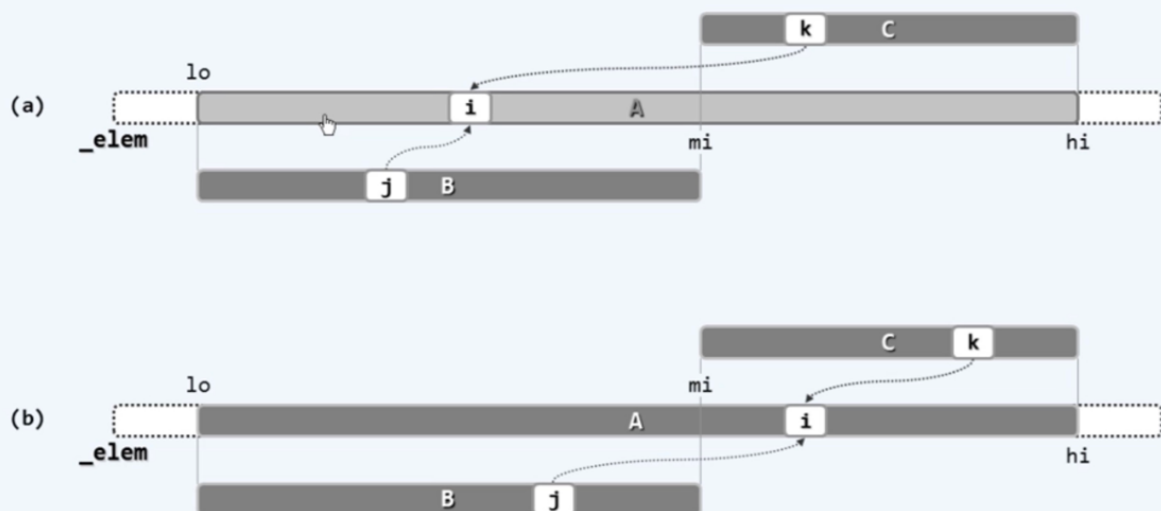
Data Structures & Algorithms (Fall 2013), Tsinghua University

4

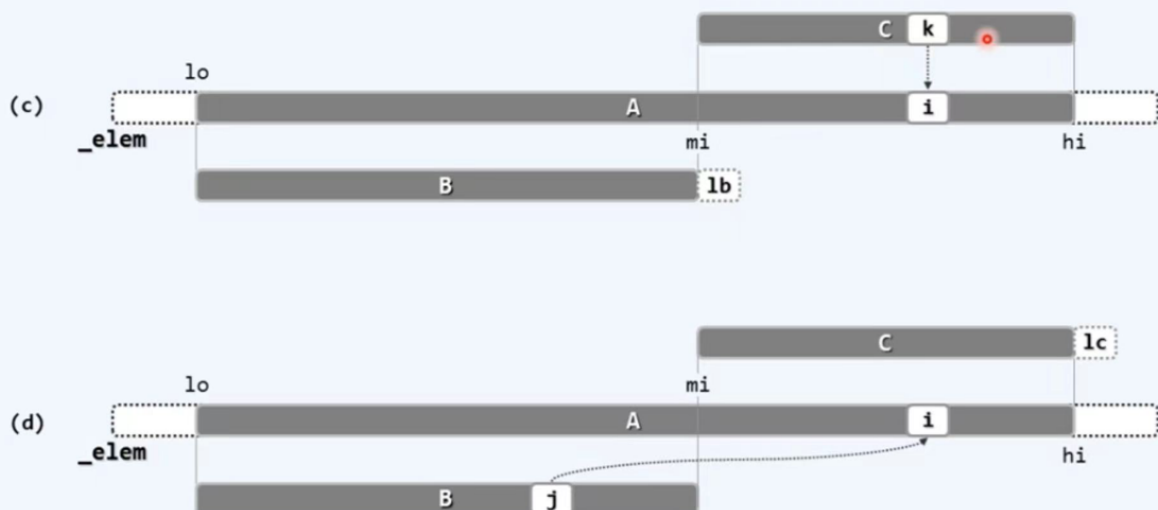
```
template <typename T> void Vector<T>::merge(Rank lo, Rank hi){
    T* A = _elem + lo; // 合并后的向量为 A[0,hi-lo) = _elem[lo,hi)
    Rank lb = mi - lo;
    T* B = new T[lb]; // 前子向量B[0,lb) = _elem[lo,mi)
    for (Rank i = 0; i < lb; ++i) { // 复制前子向量B
        B[i] = A[i];
    }
    Rank lc = hi-mi;
    T* C = _elem + mi; // 后子向量C[0,lc) = _elem[mi,hi]
    for (Rank i = 0, j = 0, k = 0; (j < lb) || (k < lc); ) { // B[j],C[k]中的小的
        // 部分转换为A的末尾
        if(j < lb && (lc < k || (B[j] <= c[k]))){ // c[k]空 或者 不小
            // j < lb j没有到底，就是没有出完
            A[i++] = B[j++];
        }
        if(k < lc && (lb <= j || (c[k] < B[j]))){ // B[j]空 或者更大
            A[i++] = C[k++];
        }
    }
    delete [] B;
}
```

二路归并：正确性

二路归并：正确性



二路归并：正确性



C：情况其实可以提前退出，因为C其实是本身引用

但是为了与D相统一，所以还是继续保持移动，其实是自己朝自己移动

但是D的情况是必须移动！

基于上述理论可以进行一定的简化

二路归并简化版本

```
template <typename T> void Vector<T>::merge(Rank lo, Rank hi){
    T* A = _elem + lo; // 合并后的向量为 A[0, hi-lo) = _elem[lo, hi)
    Rank lb = mi - lo;
    T* B = new T[lb]; // 前子向量B[0, lb) = _elem[lo, mi)
    for (Rank i = 0; i < lo; ++i) { // 复制前子向量B
```

```

        B[i] = A[i];
    }
    Rank lc = hi-mi;
    T* C = _elem + mi; // 后子向量C[0,lc) = _elem[mi,hi]
    for (Rank i = 0, j = 0, k = 0; j < lb;) { // B[j], C[k]中的小的部分转换为A的末尾
        if (lc < k || (B[j] <= c[k])) { // c[k]空 或者 不小
            // j < lb j没有到底, 就是没有出完
            A[i++] = B[j++];
        }
        if (k < lc && (c[k] < B[j])) { // B[j]空 或者更大
            A[i++] = C[k++];
        }
    }
    delete [] B;
}

```

二路归并：复杂度版本

主要复杂度在 `for` 循环

```

for(...., (j < lb) || (k < lc)) {
    ...
}

```

循环的最大次数为 $O(lb + lc = n) = O(n)$

就是每一次迭代 `j, k` 至少执行一句, 也就是 `j` 或者 `k` 至少有一个 `++`

所以 `merge()` 函数最多执行 `N` 次 也只需要线性次的时间 $O(n)$

注意: 子序列不一定需要等长, 也就是说 `lb != lc`

```
mi := (lo + hi)/2
```