Data Structures

Ch8

排序
Sort

2024 年 12 月 20 日

Chapter 8 排序

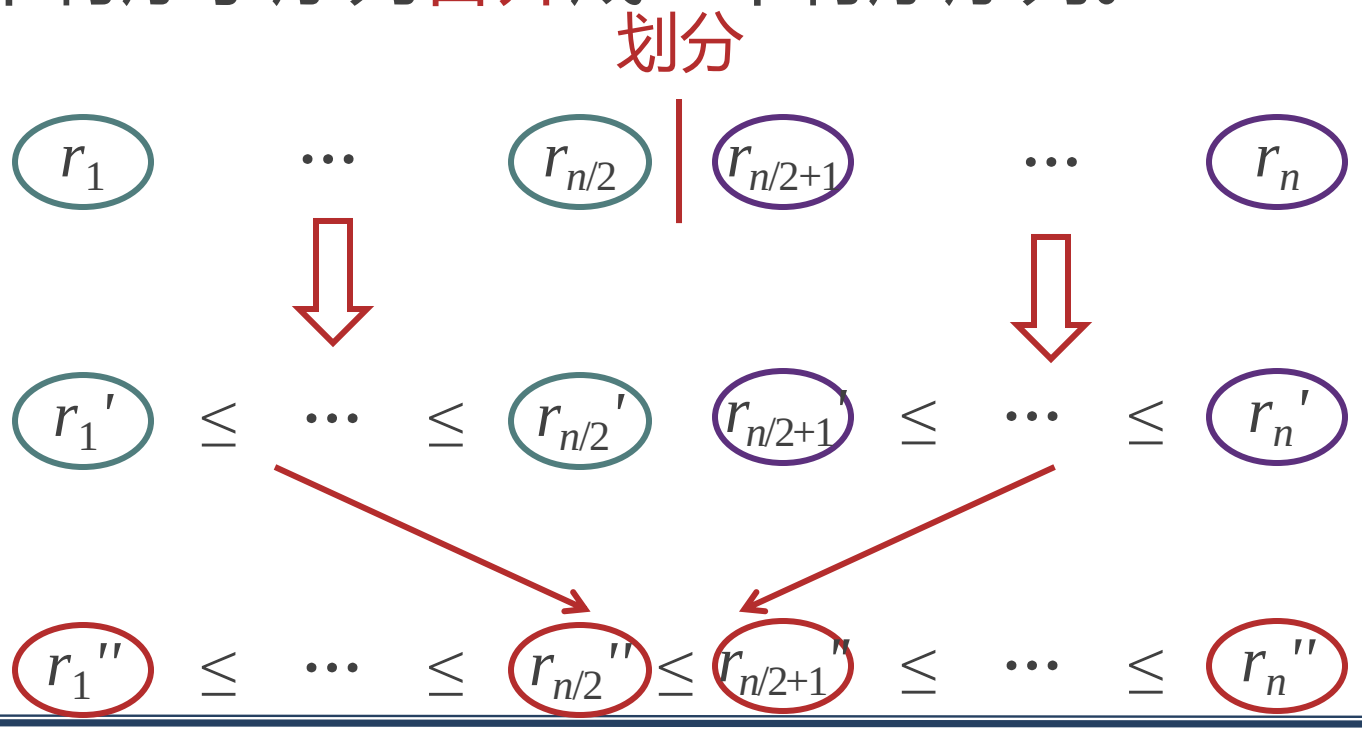
- 8.1 概述
- 8.2 插入排序：直接插入排序、希尔排序
- 8.3 交换排序：起泡、快速排序
- 8.4 选择排序：简单选择、堆排序
- 8.5 归并排序：二路归并排序
- 8.6 各种排序方法比较
- 8.7 扩展与提高

8.5 归并排序

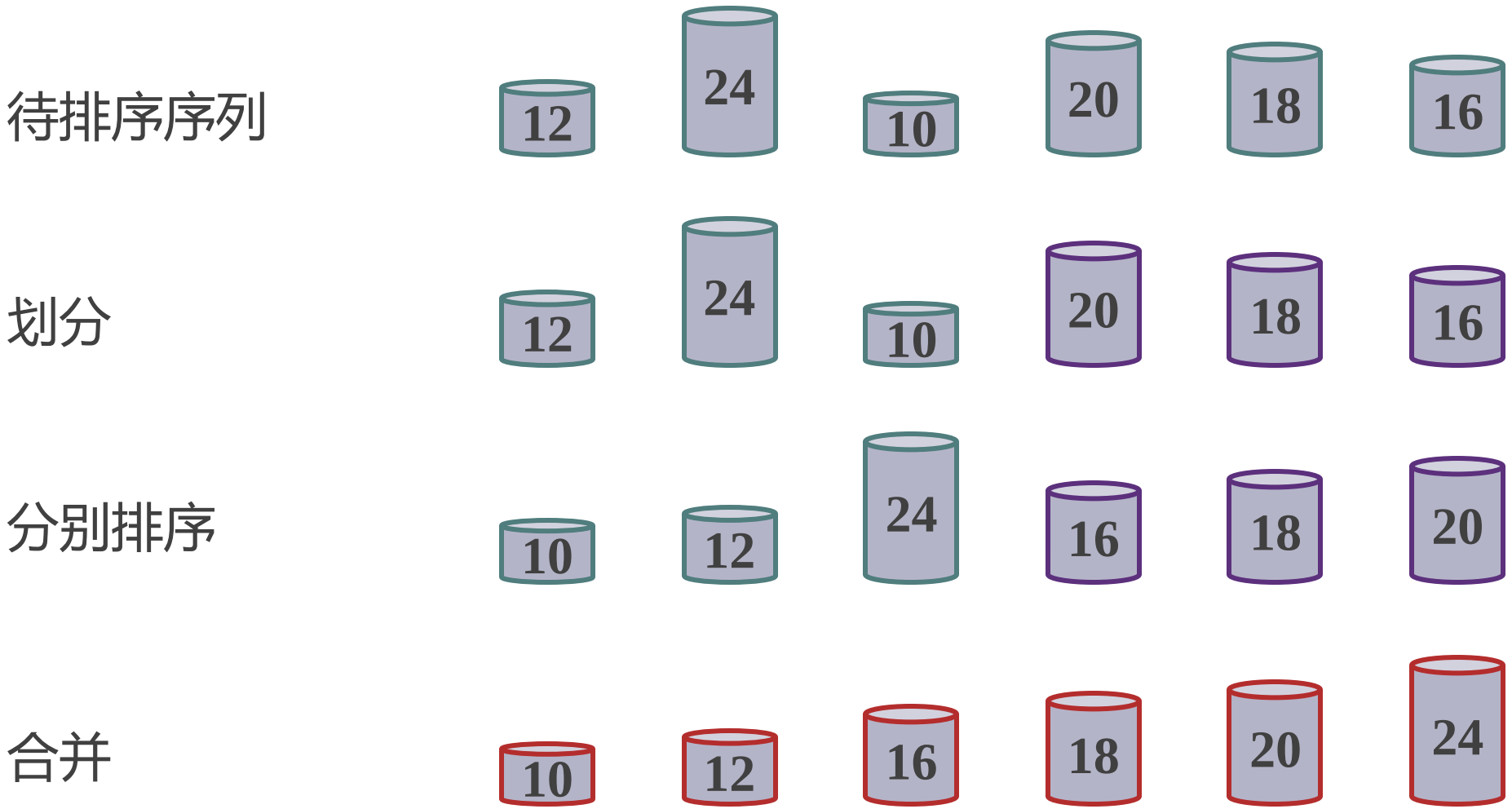
8-5-1二路归并排序的递归实现

1. 二路归并排序

二路归并排序的基本思想：将待排序序列划分为两个长度相等的子序列，分别对这两个子序列进行排序，得到两个有序子序列，再将这两个有序子序列合并成一个有序序列。



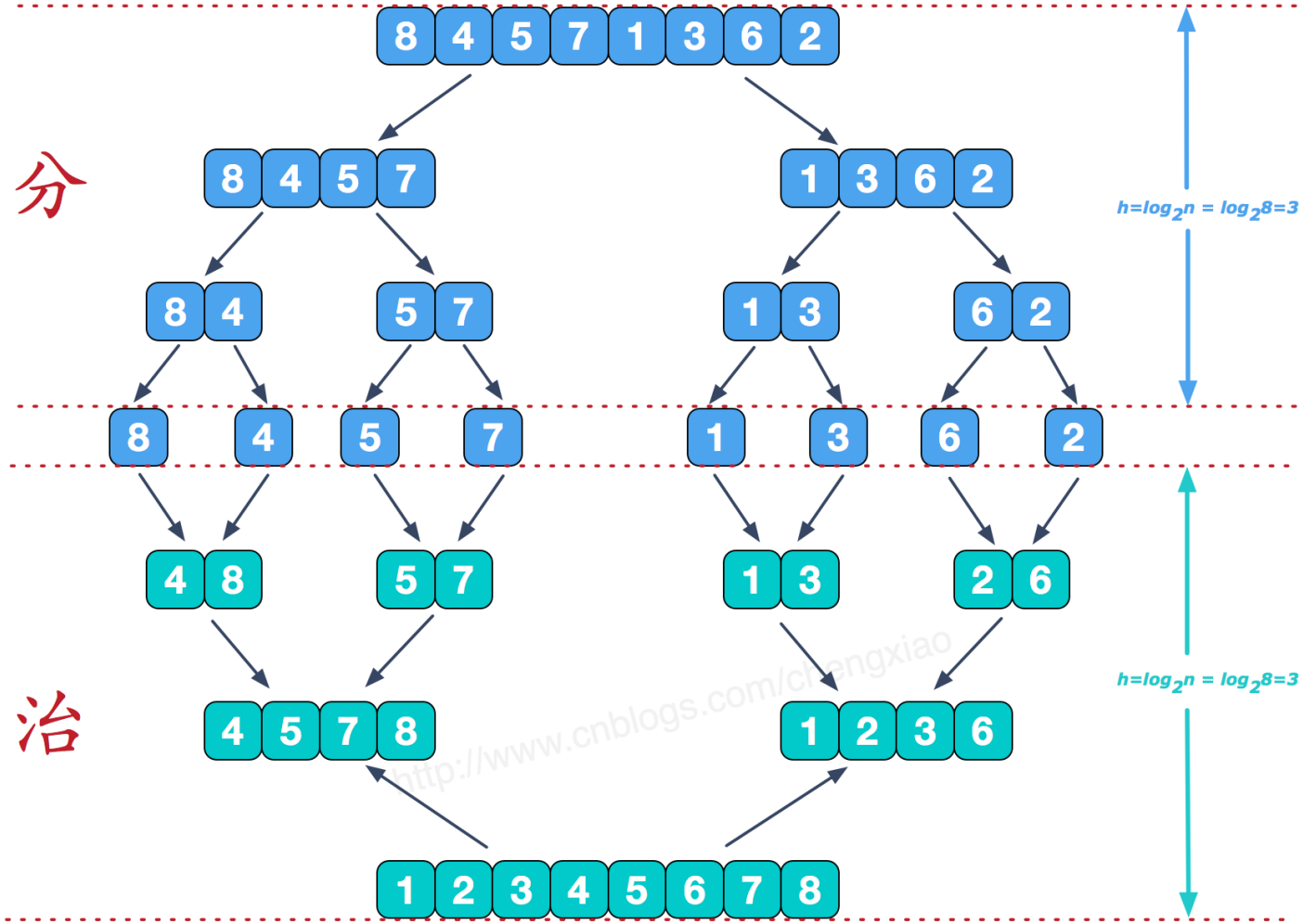
1. 二路归并排序


待排序序列
12
24
10
20
18
16
划分
12
24
10
20
18
16
分别排序
10
12
24
16
18
20
合并
10
12
16
18
20
24

8.5 归并排序

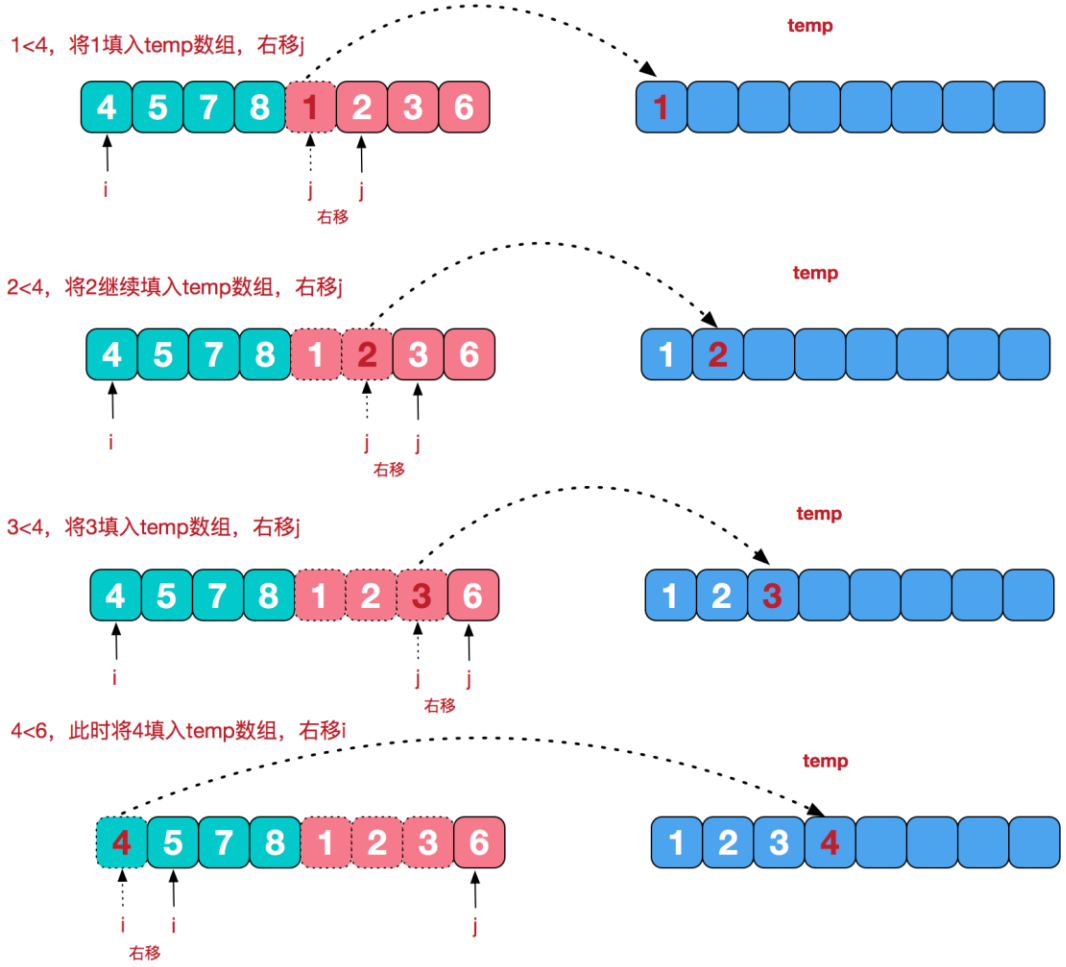
1. 二路归并排序

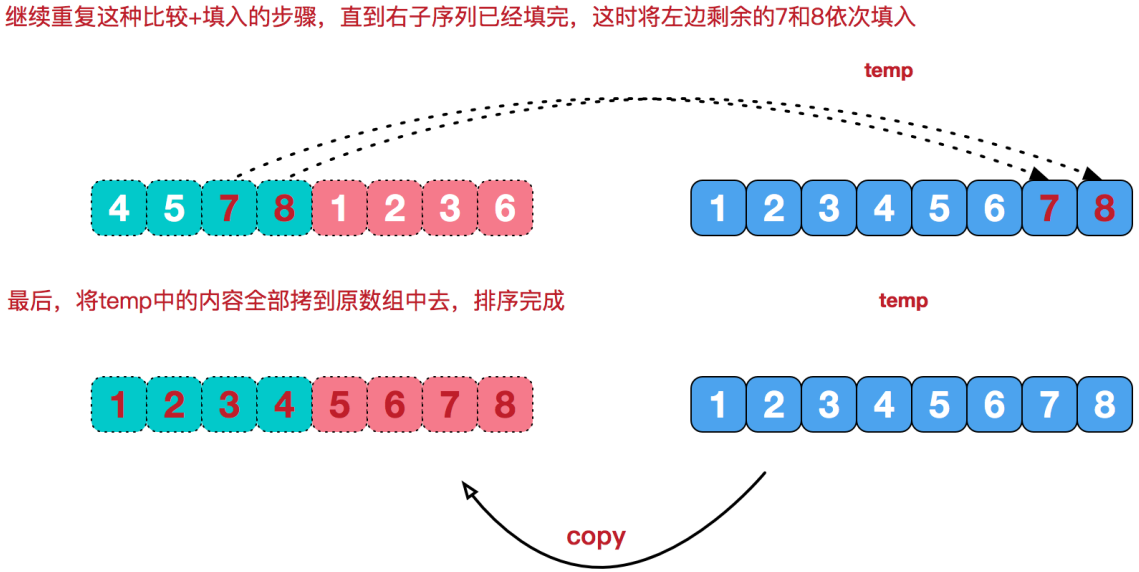
归并排序（MERGE-SORT）是利用**归并**的思想实现的排序方法，该算法采用经典的**分治**（divide-and-conquer）策略（分治法将问题**分**(divide)成一些小的问题然后递归求解，而**治(conquer)**的阶段则将分的阶段得到的各答案"修补"在一起，即分而治之)。



8.5 归并排序

2. 二路归并过程



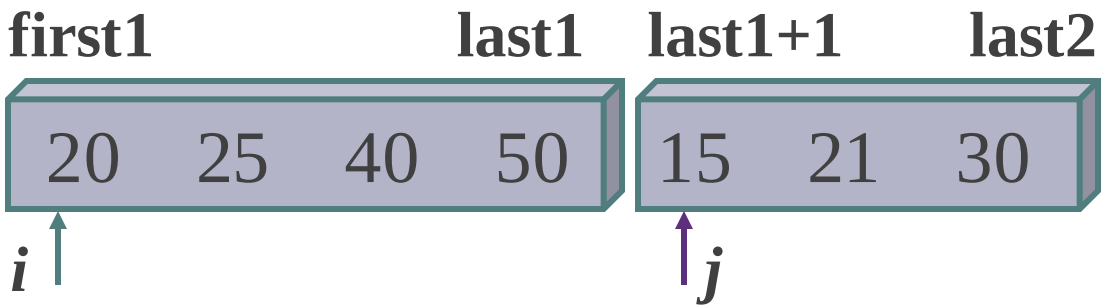


- 如何表示两个相邻的子序列?
- 合并可以就地进行吗?

2. 二路归并过程

一次合并：合并两个相邻的有序子序列

data[ ]

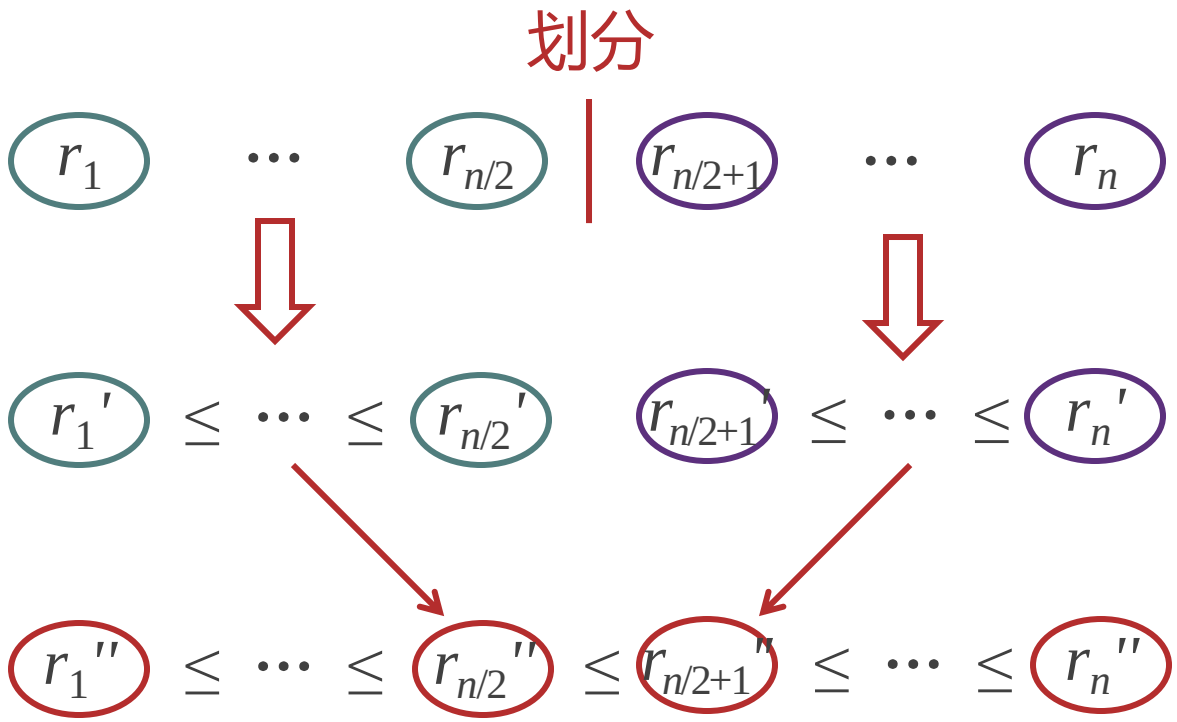


temp[ ]


```
void Sort :: Merge(int first1, int last1, int last2)
{
    int *temp = new int[length];
    int i = first1, j = last1 + 1, k = first1;
    while (i <= last1 && j <= last2)
    {
        if (data[i] <= data[j]) temp[k++] = data[i++];
        else temp[k++] = data[j++];
    }
    while (i <= last1)
        temp[k++] = data[i++];
    while (j <= last2)
        temp[k++] = data[j++];
    for (i = first1; i <= last2; i++)
        data[i] = temp[i];
    delete[ ] temp;
}
```

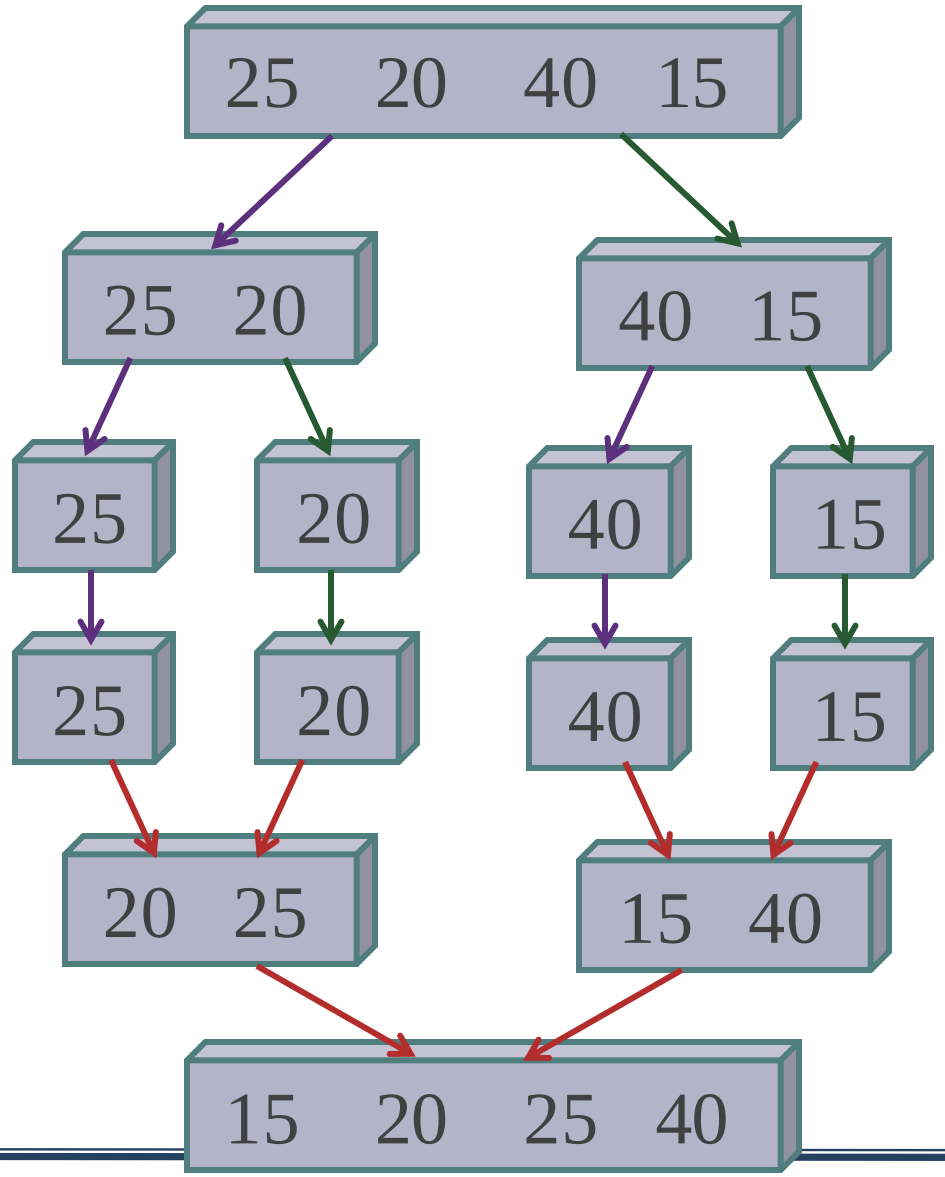
3. 二路归并排序的递归实现

```
void Sort :: MergeSort1(int first, int last)
{
    if (first == last) return;
    else {
        int mid = (first + last)/2;
        MergeSort1(first, mid);
        MergeSort1(mid+1, last);
        Merge(first, mid, last);
    }
}
```



3. 二路归并排序的递归实现

```
void Sort :: MergeSort1(int first, int last)
{
    if (first == last) return;
    else {
        int mid = (first + last)/2;
        MergeSort1(first, mid);
        MergeSort1(mid+1, last);
        Merge(first, mid, last);
    }
}
```



3. 二路归并排序的递归实现

```
void Sort :: MergeSort1(int first, int last)
{
    if (first == last) return;
    else {
        int mid = (first + last)/2;
        MergeSort1(first, mid);
        MergeSort1(mid+1, last);
        Merge(first, mid, last);
    }
}
```

```
void Sort :: Merge(int first1, int last1, int last2)
{
    int *temp = new int[length];
    int i = first1, j = last1 + 1, k = first1;
    while (i <= last1 && j <= last2)
    {
        if (data[i] <= data[j]) temp[k++] = data[i++];
        else temp[k++] = data[j++];
    }
    while (i <= last1)
        temp[k++] = data[i++];
    while (j <= last2)
        temp[k++] = data[j++];
    for (i = first1; i <= last2; i++)
        data[i] = temp[i];
    delete[ ] temp;
}
```

8.5 归并排序

8-5-2 二路归并排序的非递归实现

1. 非递归算法

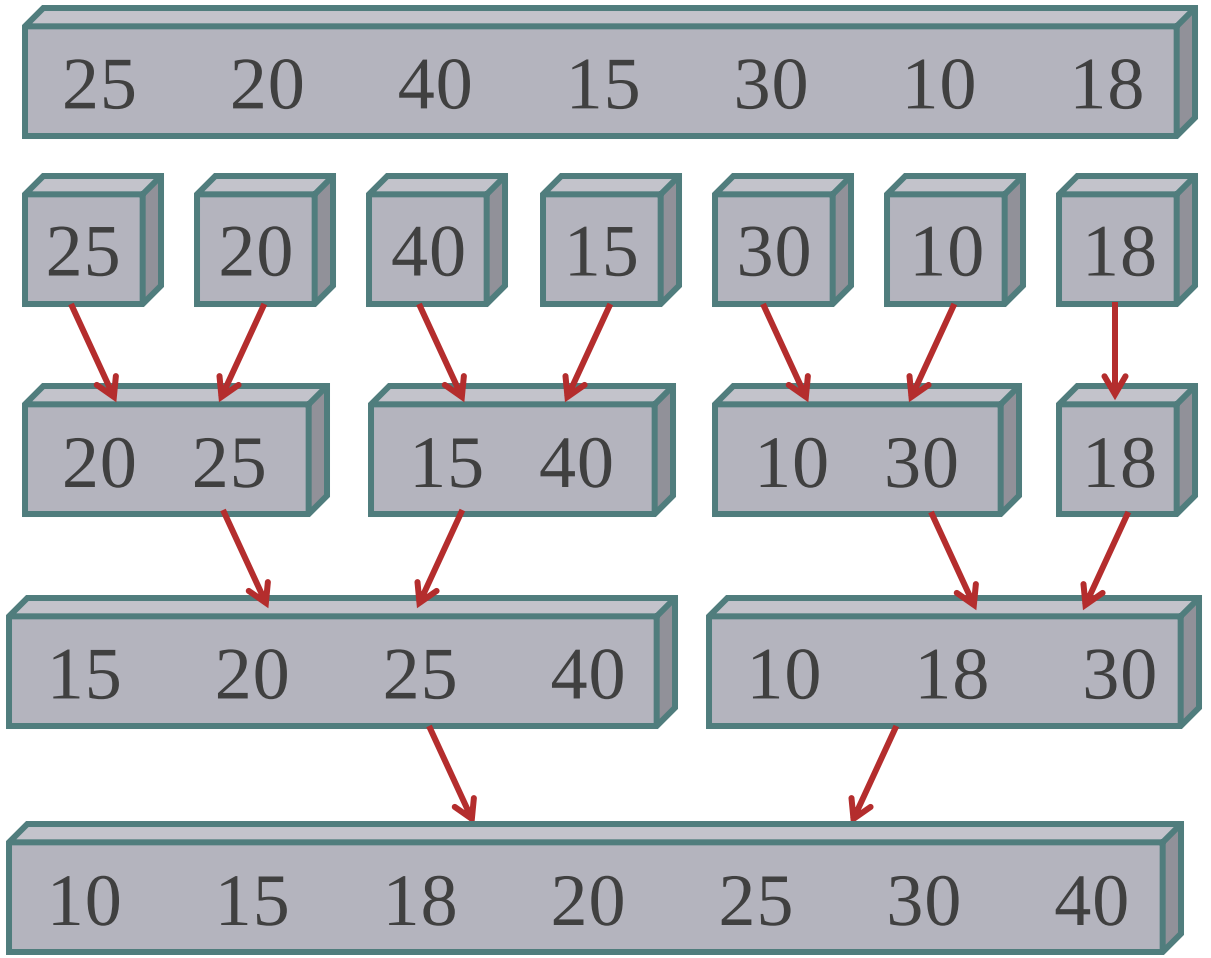
待排序序列

构造初始有序子序列

第一趟归并结果

第二趟归并结果

第三趟归并结果



子序列长度有什么规律？在一趟归并中有几种情况？

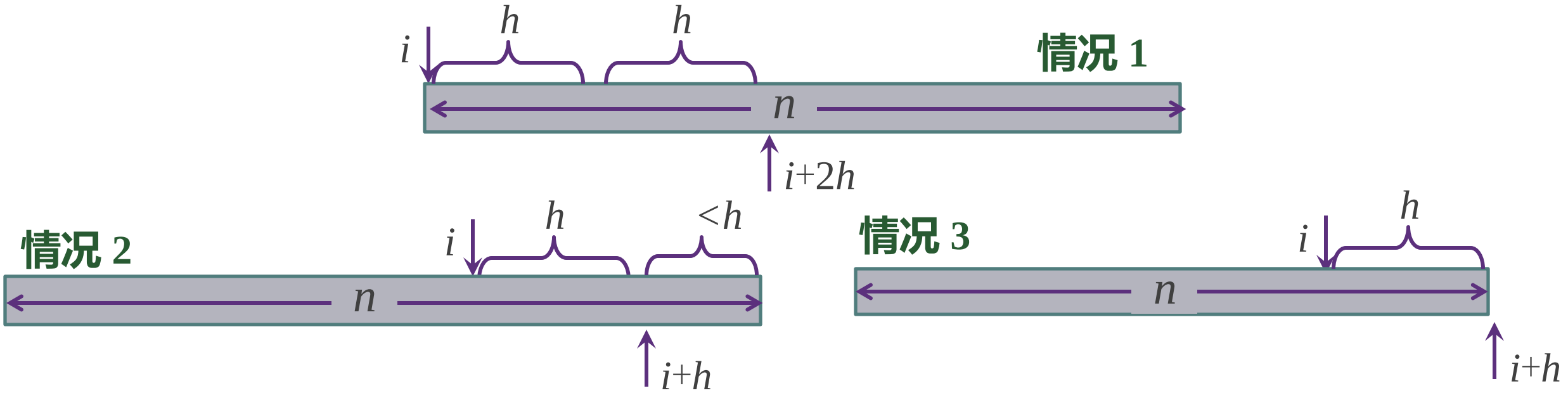
2. 一趟归并

设 i 指向待归并序列的第一个记录，归并的步长是$2h$

情况 1：$i+2h \le n$，则相邻两个有序子序列的长度均为 h

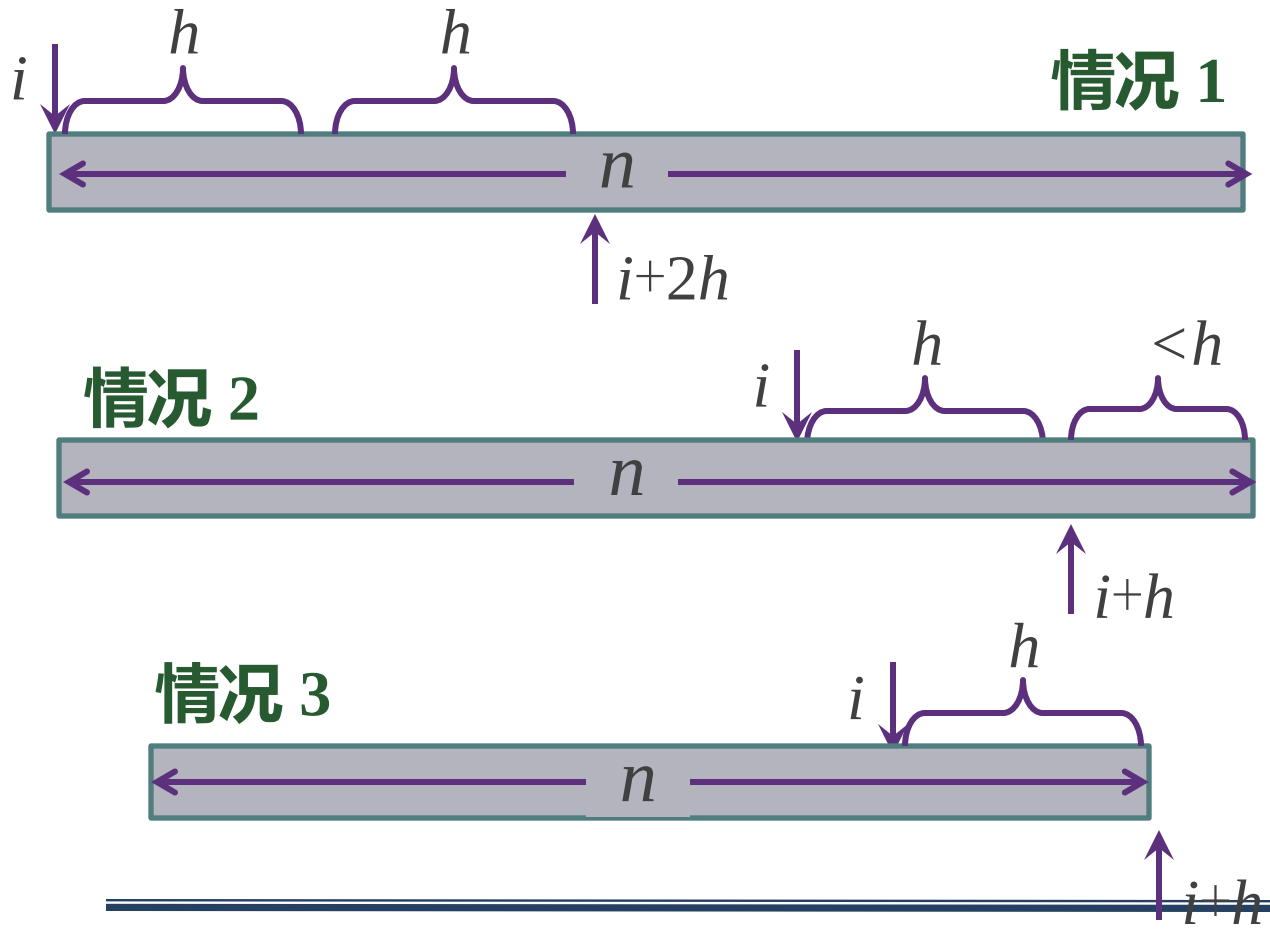
情况 2：$i+h < n$，则相邻有序子序列一个长度为 h，另一个长度小于 h

情况 3：$i+h \ge n$，则表明只剩下一个有序子序列



2. 一趟归并

设 i 指向待归并序列的第一个记录，归并的步长是$2h$




```
void Sort :: MergePass(int h) //一趟归并排序
{
    int i = 0;
     //有两个长度为h的子序列
    while (i + 2 * h <= length)
    {
        Merge(i, i+h-1, i+2*h-1);
        i = i + 2 * h;
    }
   //两个子序列一个长度小于h
   if (i + h < length) Merge(i, i+h-1, length-1);
}
```

3. 二路归并排序的非递归算法

```
void Sort :: MergeSort2( )
{
    int h = 1;
    while (h < length)
    {
        MergePass(h);  //一趟归并排序
        h = 2*h;
    }
}
```

```
void Sort :: MergePass(int h) //一趟归并排序
{
    int i = 0;
    //有两个长度为h的子序列
    while (i + 2 * h <= length)
    {
        Merge(i, i+h-1, i+2*h-1);
        i = i + 2 * h;
    }
   //两个子序列一个长度小于h
   if (i + h < length) Merge(i, i+h-1, length-1);
}
```

4. 时间复杂度分析

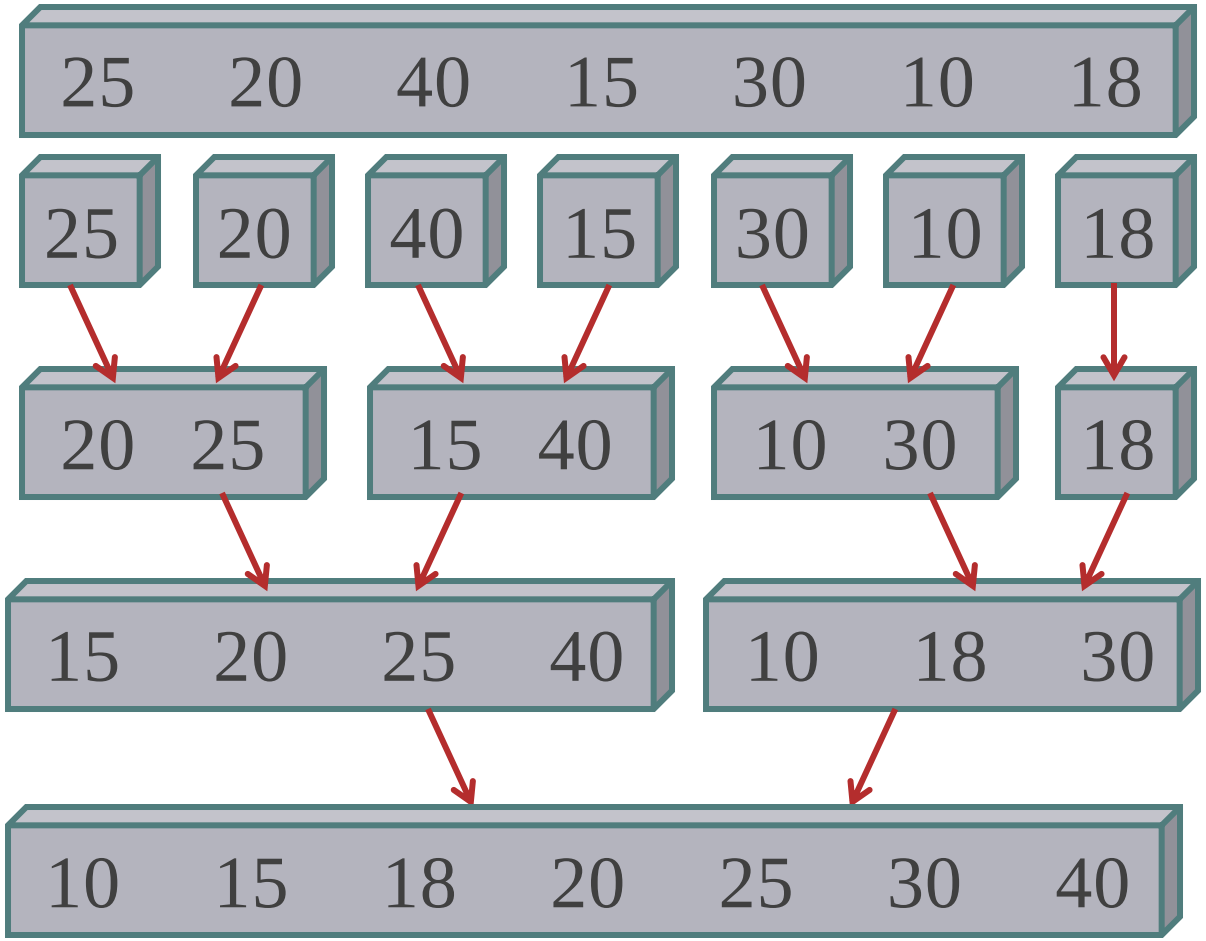
待排序序列

构造初始有序子序列

第一趟归并结果

第二趟归并结果

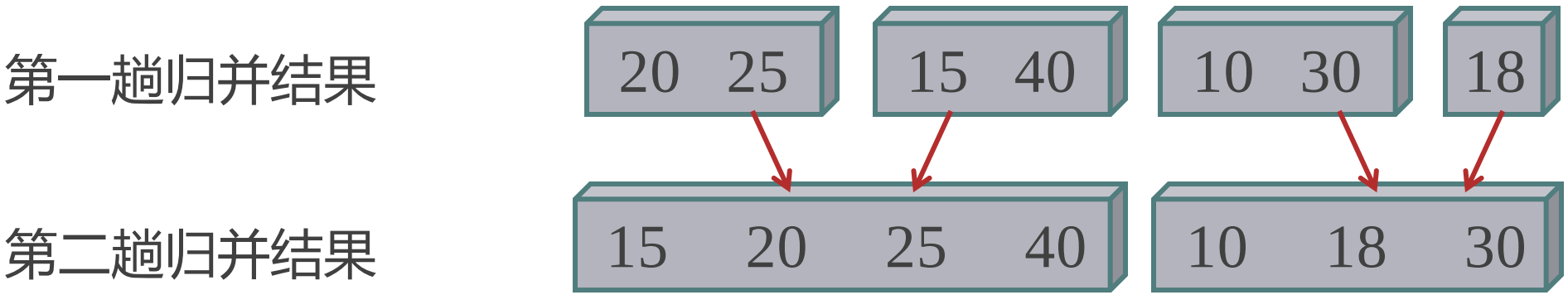
第三归并结果



二路归并执行多少趟？每一趟的时间性能是多少？

4. 时间复杂度分析

- 执行趟数：$\log_2 n$
- 每一趟：将记录扫描一遍，$O(n)$
- 最好、最坏、平均情况：$O(n\log_2 n)$

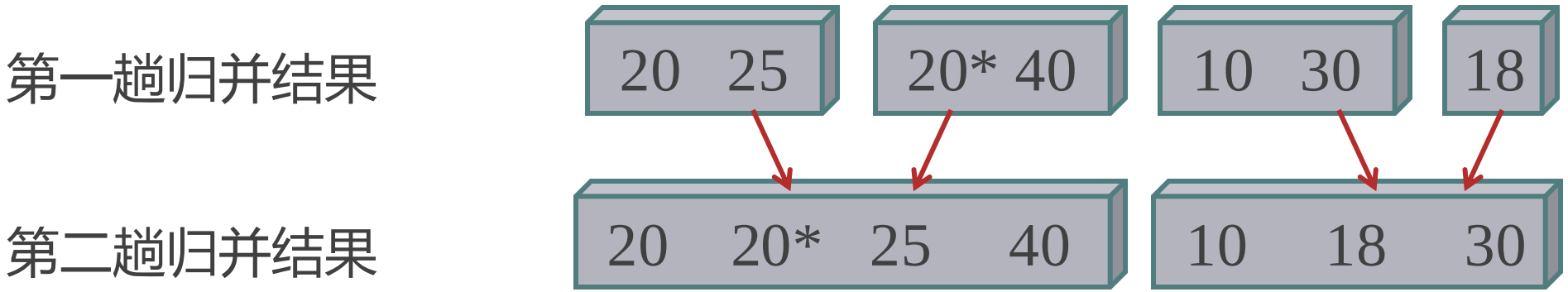


5. 空间复杂度分析

- 空间性能：合并不能就地进行，$O(n)$
- 稳定性：稳定

```
while (i <= m && j <= t)
    if (r[i] <= r[j])  r1[k++] = r[i++];
    else  r1[k++] = r[j++];
```

- 如果将判断条件改为(r[i] < r[j])，仍然是稳定的吗？



8.6 各种排序方法比较

8.6 各种排序方法比较

1. 时间性能

排序方法	平均情况	最好情况	最坏情况
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$
希尔排序	$O(n\log_2 n)$~$O(n^2)$	$O(n^{1.3})$	$O(n^2)$
起泡排序	$O(n^2)$	$O(n)$	$O(n^2)$
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$

8.6 各种排序方法比较

1. 时间性能

利用随机数，随机生成区间0 ~ K之间的序列，共计N个数字，利用各种算法进行排序，记录排序所需时间

算法\输入数据	N=50 K=50	N=200 K=100	N=500 K=500	N=2000 K=2000	N=5000 K=8000	N=10000 K=20000	N=20000 K=20000	N=20000 K=200000
冒泡排序	0ms	15ms	89ms	1493ms	9363ms	36951ms	147817ms	143457ms
插入排序	1ms	13ms	82ms	1402ms	8698ms	34731ms	134817ms	134836ms
希尔排序	0ms	1ms	6ms	30ms	110ms	257ms	599ms	606ms
选择排序	0ms	5ms	31ms	461ms	2888ms	11736ms	45308ms	44838ms
堆排序	0ms	3ms	9ms	40ms	124ms	247ms	525ms	527ms
归并排序	2ms	6ms	18ms	75ms	199ms	392ms	778ms	793ms
快速排序	0ms	1ms	2ms	14ms	36ms	84ms	196ms	163ms
计数排序	0ms	1ms	1ms	5ms	15ms	32ms	51ms	62ms
基数排序	0ms	1ms	4ms	19ms	47ms	114ms	237ms	226ms
桶排序	0ms	2ms	6ms	25ms	68ms	126ms	254ms	251ms

8.6 各种排序方法比较

1. 时间性能

排序方法	平均情况
直接插入排序	$O(n^2)$
希尔排序	$O(n\log_2 n)$~$O(n^2)$
起泡排序	$O(n^2)$
快速排序	$O(n\log_2 n)$
简单选择排序	$O(n^2)$
堆排序	$O(n\log_2 n)$
归并排序	$O(n\log_2 n)$

从平均情况看

（1）直接插入排序、简单选择排序和起泡排序属于一类，时间复杂度为$O(n^2)$；

（2）堆排序、快速排序和归并排序属于一类，时间复杂度为$O(n\log_2 n)$；

（3）希尔排序的时间性能取决于增量序列，介于$O(n^2)$和$O(n\log_2 n)$之间。

快速排序是目前最快的一种排序方法

在待排序记录个数较多的情况下，归并排序比堆排序更快

8.6 各种排序方法比较

1. 时间性能

排序方法	最好情况
直接插入排序	$O(n)$
希尔排序	$O(n^{1.3})$
起泡排序	$O(n)$
快速排序	$O(n\log_2 n)$
简单选择排序	$O(n^2)$
堆排序	$O(n\log_2 n)$
归并排序	$O(n\log_2 n)$

从最好情况看

（1）直接插入排序和起泡排序最好，$O(n)$；

（2）其他排序算法的最好情况与平均情况相同。

如果待排序序列接近正序，首选起泡排序和直接插入排序

8.6 各种排序方法比较

1. 时间性能

排序方法	最坏情况
直接插入排序	$O(n^2)$
希尔排序	$O(n^2)$
起泡排序	$O(n^2)$
快速排序	$O(n^2)$
简单选择排序	$O(n^2)$
堆排序	$O(n\log_2 n)$
归并排序	$O(n\log_2 n)$

从最坏情况看

（1）快速排序的时间复杂度为$O(n^2)$；

（2）直接插入排序和起泡排序虽然与平均情况相同，但系数大约增加一倍，所以运行速度将降低一半；

（3）最坏情况对直接选择排序、堆排序和归并排序影响不大。

如果待排序序列接近正序或逆序，不使用快速排序

8.6 各种排序方法比较

2. 空间性能

排序方法	辅助空间
直接插入排序	$O(1)$
希尔排序	$O(1)$
起泡排序	$O(1)$
快速排序	$O(\log_2 n)$ ~$O(n)$
简单选择排序	$O(1)$
堆排序	$O(1)$
归并排序	$O(n)$

从空间性能看

(1) 归并排序的空间复杂度为$O(n)$;

(2) 快速排序的空间复杂度为$O(\log_2 n)$~$O(n)$;

(3) 其它排序方法的空间复杂度为$O(1)$。

3. 稳定性与简单性

从稳定性看

（1）稳定：包括直接插入排序、起泡排序和归并排序；

（2）不稳定：包括希尔排序、简单选择排序、快速排序和堆排序。

从算法简单性看

（1）简单算法：包括直接插入排序、简单选择排序和起泡排序，

（2）改进算法，较复杂：包括希尔排序、堆排序、快速排序和归并排序。

8.6 各种排序方法比较

4. 记录本身信息量

从记录本身信息量的大小看

记录本身信息量越大，占用的存储空间就越多，移动记录所花费的时间就越多，所以对记录的移动次数较多的算法不利。

排序方法	最好情况	最坏情况	平均情况
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$
起泡排序	0	$O(n^2)$	$O(n^2)$
简单选择排序	0	$O(n)$	$O(n)$

记录本身信息量的大小对改进算法的影响不大。

记录个数不多且记录本身的信息量较大时，首选简单选择排序算法

5. 关键码分布

从关键码的分布看

（1）当待排序记录按关键码有序时，插入排序和起泡排序能达到$O(n)$的时间复杂度；对于快速排序而言，这是最坏情况，时间性能蜕化为$O(n^2)$；

（2）简单选择排序、堆排序和归并排序的时间性能不随记录序列中关键码的分布而改变。

各种排序算法各有优缺点，
应该根据实际情况选择合适的排序算法

本章小结

1. 掌握**直接插入排序**算法、实现及改进
2. 掌握**希尔排序算法**原理与性能分析方法
3. 理解**交换排序**的基本思想
4. 掌握**起泡排序**的思想和实现方法
5. 掌握**快速排序**的基本思想、**一次划分**方法和**递归实现**
6. 掌握**简单选择排序**算法及实现
7. 掌握**堆排序**算法及实现
8. 掌握**二路归并排序**方法及实现

作业

1．已知关键字序列(3, 26, 38, 5, 47, 15, 36, 26*, 2, 4, 19, 50)，使用直接插入排序、希尔排序、起泡排序、快速排序、简单选择排序、堆排序、二路归并排序七种排序算法进行排序，请分别给出七种排序算法每一趟排序的结果，并给出时间和空间复杂度量级。

实验安排

实验九 排序算法的实现与应用

一、实验目的

1. 复习线性表顺序存储结构的实现方法
2. 掌握七种排序算法的基本原理、实现方法
3. 用C++语言实现相关算法，并上机调试。

插入排序：直接插入排序、希尔排序
交换排序：起泡排序、快速排序
选择排序：简单选择排序、堆排序
归并排序：二路归并排序算法的递归与非递归实现

二、实验内容

1. 实现至少三种排序算法，并使用相同数据测试，统计比较次数和移动次数。
2. 实现至少三种排序算法的执行时间统计（调用clock或其他函数记录系统时间，计算时间差）。
3. 给出测试过程和测试结果。

测试数据：{3,44,38,5,47,15,36,26,27,2,46,4,19,50,48}。

12月24日前提交预习报告，2024年12月30日前提交正式报告。

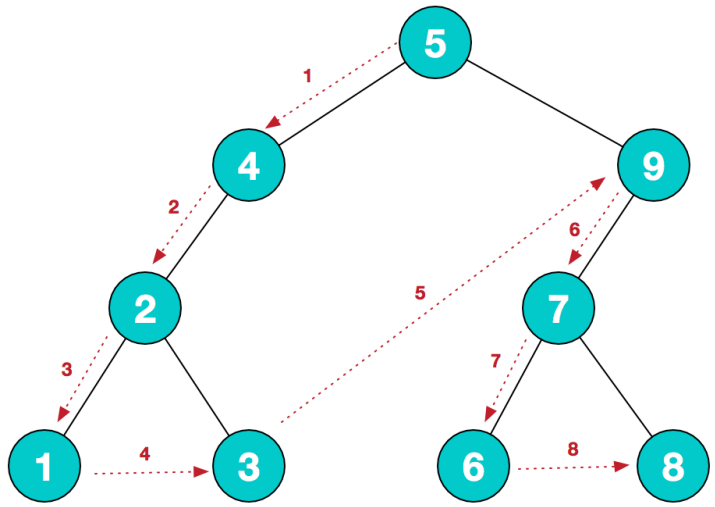
实验时间：　第18周周四晚 19:00-21:00
实验地点：　格物楼A216

Thank You !

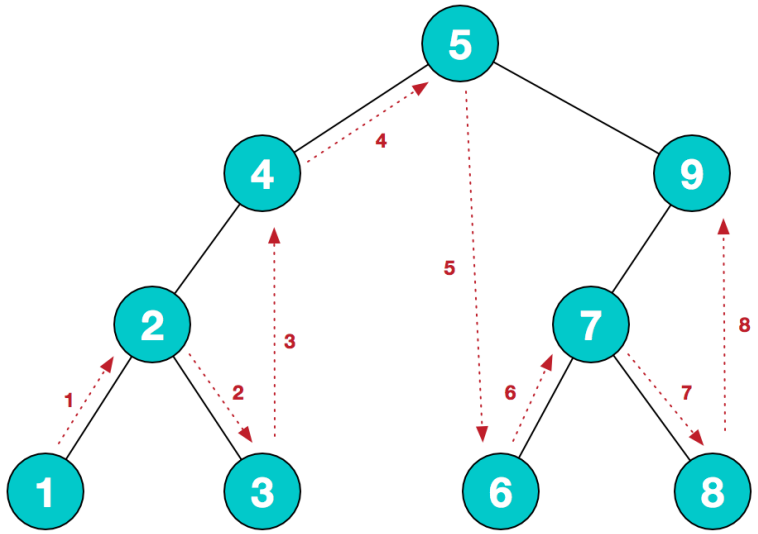
Q & A

```
#include <chrono>
auto startTime = std::chrono::high_resolution_clock::now();
//To Do List
auto endTime = std::chrono::high_resolution_clock::now();
std::chrono::duration<double, std::milli> fp_ms = endTime - startTime;
std::cout << "Runing time is: "<<fp_ms.count()/1000 <<" s"<< std::endl;
```

前序遍历顺序：5-4-2-1-3-9-7-6-8



中序遍历顺序：1-2-3-4-5-6-7-8-9



后序遍历顺序：1-3-2-4-6-8-7-9-5