数据结构与算法分析

华中科技大学软件学院

2017年秋

大纲

课程计划

- 已经学习了
  - 散列
  - 冲突的处理
  - 再散列
  - 优先队列和堆

课程计划

- 已经学习了
  - 散列
  - 冲突的处理
  - 再散列
  - 优先队列和堆
- 即将学习
  - 排序算法
  - 排序算法的分析
  - 外部排序

Roadmap

为什么需要排序

- 计算机上运行的最重要的工作之一
- 有许多重要的应用
  - 先排序，再查找
  - Google搜索，PageRank
- 研究的比较全面
- 关键算法在许多语言中已经实现，例如qsort

排序为何重要

- 非常常见，很多计算机花大量时间排序
- 收件箱中的邮件可以按照日期，主题，发件人排序
- 可以使得其他工作变得容易一些
  - 数组排序后可以使用折半查找 ，时间复杂度降为$\log n$
  - 寻找中位数
  - 寻找重复值
  - 寻找重数

排序的算法

- 简单算法，运算较慢
  - 插入排序
  - 冒泡排序
- 次二次型排序算法：希尔排序
- 最优排序，复杂度O(n log n)
  - 归并排序
  - 快速排序
  - 堆排序
- 特殊情况：桶式排序、基数排序

洗牌与排序

- 洗牌：随机选取一种排列方式，如果恰好排好，停止
- 否则重新洗牌
  - 最好情况： $O(1)$
  - 最坏情况： ∞
  - 平均情况：$n * n!$，n次调用随机数发生器，n!种排列

回顾插入排序

```
void InsertionSort (ElementType A[], int N)
{
    int j, p;
    ElementType tmp;

    for (p = 1; p < N; p++)
    {
        tmp = A[p];
        for (j = p; j > 0 && A[j-1] > tmp; j--)
            A[j] = A[j-1];
        A[j] = tmp;
    }
}
```

递归求解

- 平凡情况，若N为1，已经排序，返回
- 将长度为n的数组划分为2部分： 长为n−1的数组，和最后一个元素
  - 解决次问题
  - 将最后一个元素插入到n−1的已排序数组中
- 最后一步可以更有效率些吗？ 二分查找怎么样?

冒泡排序

- 想法：通过交换把大数移到顶部

```
for i = n-1 down to 1
    for j = 1 to i
        if A[j ] < A[j-1]
            swap A[j] and A[j-1]
```

- 每趟排序将最大数移到合适的位置
  - 只有n趟排序
  - 每趟排序的交换少于n次
  - $O(n^2)$

Simple slow algorithms

- “简单”的慢速算法有什么坏处?
- 只比较相邻项
- 冒泡排序显然是这样的
- 插入排序可认为是这样的

复杂度分析

Theorem

任何相邻项比较的算法复杂度都是$\Theta(n^2)$的

证明

证：设序偶(i, j)，不妨 i < j，有 a[i] > a[j]，称此为逆序(inversion).
有多少种可能的(ordered)有序偶？答案是 n*(n-1)/2. 考虑一些排序0和反转的0^{-1}，(i, j)会在0 或 0^{-1} 若0是随机的，则可期望半数的序偶被反转. 故可期望的反转数量为：$\frac{n*(n-1)/2}{2} = \Theta(n^2)$
为什么这很重要？任何相邻项的交换都可以精确地进行一次反转. 平均地，只交换相邻项的算法复杂度是$\Theta(n^2)$

Roadmap

ShellSort

- 如何更快地排序？ 必须交换相距更远的项
- 希尔排序： Donald Shell, 1959. 亚二次，但仅比插入排序难一点点
  - 具体想法： 交换相距很远的项
  - 用若干“增量(increments)”做插入排序
  - 最后，按距离排序
  - Eg: 8, 4, 2, 1, 是递减的 尾数为 1
- 结果呈现： 后做的排序不会影响先前的

举例

- 考虑一个队列:
  4 63 20 3 61 40 1 65 10 5 64 30 2 62 50
- 用 3-sort 排序:
  1 61 10 2 62 20 3 63 30 4 64 40 5 65 50
- 现在用 2-sort 排序:
  1 2 3 4 5 20 10 40 30 61 62 63 50 65 64
- 用 1-sort 排序:
  1 2 3 4 5 10 20 30 40 50 61 62 63 64 65
- 注意 81 94 11 96 12 35 17 95 28 58 41 75 15
- 按 5, 3, 1 做排序

ShellSort代码

```
void shellSort (int a[], int n)
{
   int i, j, gap, cur;

   for (gap = n/2; gap > 0; gap = gap/2)
       for (i = gap; i < n; i++)
       {
            cur = a[i];
            for (j = i; j >= gap &&
                     cur < a[j - gap]; j -= gap)
                a[j] = a[j - gap];
            a[j] = cur;
       }
}
```

ShellSort的增量

- 第一批增量来自Shell： 1 2 4 8 16 n/2 （按逆序） $\rightarrow \Theta(n^2)$
- 但结果表明:据数论，平均时间为 $\Theta(n^{1.5})$
- $\Omega(n^2)$： second-to-last is 2-sort
  - 将大值置于偶数位
  - 1-sort 会反转 $\Theta(n^2)$
- $0(n^2)$： h-sort 基于 n/h 元素

Shell排序的下界

- 须说明复杂度也是 $\Omega(n^2)$
- 足以发现复杂度为$\Theta(n^2)$的一种序列类型，但不仅仅是一个序列！
- 在偶数位选n个小数，奇数位选n个大数：
  10 0 11 1 12 2 13 3
- 1-sort之前的所有排序，都在偶数位留下偶数，奇数位留下奇数
- 在1-sort，最小数(从0开始)排在位置2i+1，而不在位置i
  - 每次必须移动i+1的距离
  - n/2中的每一个都必须大致移动 $\sum_{i=0}^{n/2} i + 1 \to n^2$

Shell排序的分析

一趟排序由 h 个子集的 n/h个元素组成
令 $n = 2^k$，最坏情况

$$n^2+2*(n/2)^2+4*(n/4)^2+8*(n/8)^2+...+n/2*(n/(n/2))^2$$

$$= (1+1/2+1/4+...+1/2^{k-1})*n^2$$

$$\leq 2n^2 = \Theta(n^2)$$

最好情况，对一个已排好序的数组执行Shell Sort

$$n+2*(n/2)+4*(n/4)+8*(n/8)+...+n/2*(n/2)$$

$$= (1+1+1+...+1)*n$$

$$= (k+1)n = \Theta(n \log n)$$

改进增量

- 如果增量之间不互质，总在重复同样的工作。引入新的增量序列
- Hibbard(一种增量)： $1, 3, 7, \ldots, 2^k - 1$，连续增量是相关的素数
- 结果表明足以有效
  - 可表明： Hibbard = $O(n^{3/2})$ 为最坏情况
  - 未表明但可信： $O(n^{5/4})$ 为平均情况
- Sedgewick(一种增量)： $O(n^{4/3})$ 为最坏情况，可能： $O(n^{7/6})$ 为平均情况，序列： $9 * 4^i - 9 * 2^i + 1$ or $4^i - 3 * 2^i + 1$

Hibbard增量的复杂度分析

Theorem

使用Hibbard增量的希尔排序最坏情况运行时间为$\Theta(n^{\frac{3}{2}})$

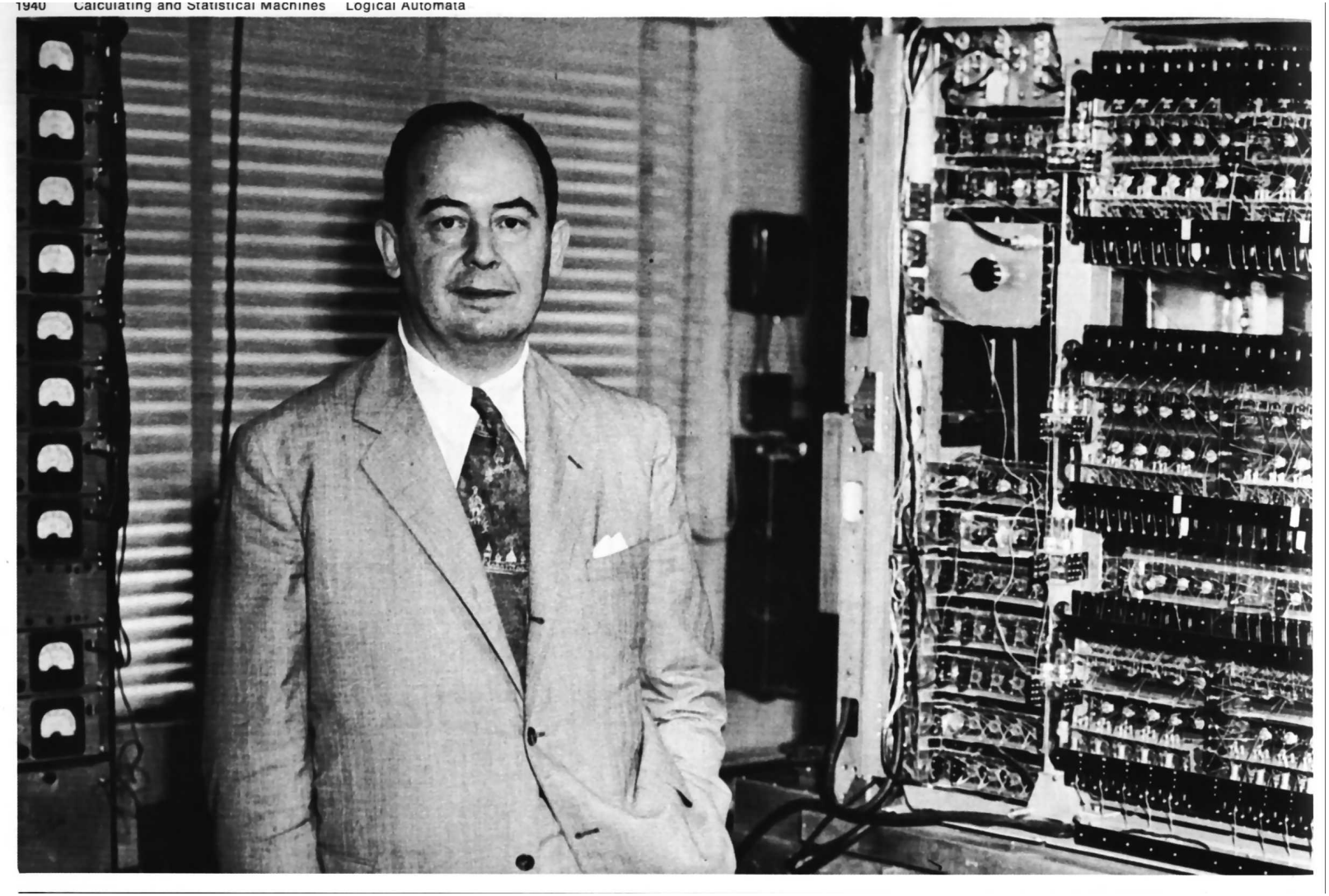
证法分析： 在h_k-sort之前， 数组是h_{k+1}-sorted的. 对于两个元素，这些元素是h_{k+1}和h_{k+2}分开的正线性组合， 他们处于正确的顺序. 在每趟排序中，只有不可表示为先前间隔的线性组合的位置，才须被排序。 那需要 $h_k N$次操作. 然后我们可以分离h_k's通过与 by comparing with $\sqrt{N}$的比较，并对每趟排序的花费求和

Roadmap

归并排序

- 基本想法： 用递归步骤分治(divide and conquer)
  - MergeSort (left half);
  - MergeSort (right half);
  - 合并2半;
  - 平凡情况：若长度为1，则返回
- 很简单的递归算法，于 1945 由 John von Neumann 提出 ，他发明了 博弈论(Game Theory)，致力于量子力学(QM)，还发明了 “冯·诺依曼体系结构(von Neumann architecture)”，用程序和数据单一存储的“存储程序”计算机

冯诺依曼和计算机


1940 Calculating and Statistical Machines Logical Automata

MergeSort

- 重复对半划分，直到平凡情况
- 合并2个长为m的序列：time 2m
- 但结果放在哪里呢?
  - 保留在同一数组，如insert-sort m^2
  - 为了得到线性时间，需放在一个新数组
- MergeSort 用时 nlogn 但使用 O(n) 的空间
  - 大缺点："记忆墙"
  - 以及：获取 n 的空间费时
  - 来回复制很慢

合并过程

98	89	56	87	34	21	43	65

合并过程

98	89	56	87	34	21	43	65

89	98	56	87	21	34	43	65

合并过程

98	89	56	87	34	21	43	65
89	98	56	87	21	34	43	65
56	87	89	98	21	34	43	65

合并过程

98	89	56	87	34	21	43	65
89	98	56	87	21	34	43	65
56	87	89	98	21	34	43	65
56	87	89	98	21	34	43	65

合并过程

98	89	56	87	34	21	43	65

89	98	56	87	21	34	43	65

56	87	89	98	21	34	43	65

56	87	89	98	21	34	43	65

21	34	43	56	65	87	89	98

合并过程

98	89	56	87	34	21	43	65

89	98	56	87	21	34	43	65

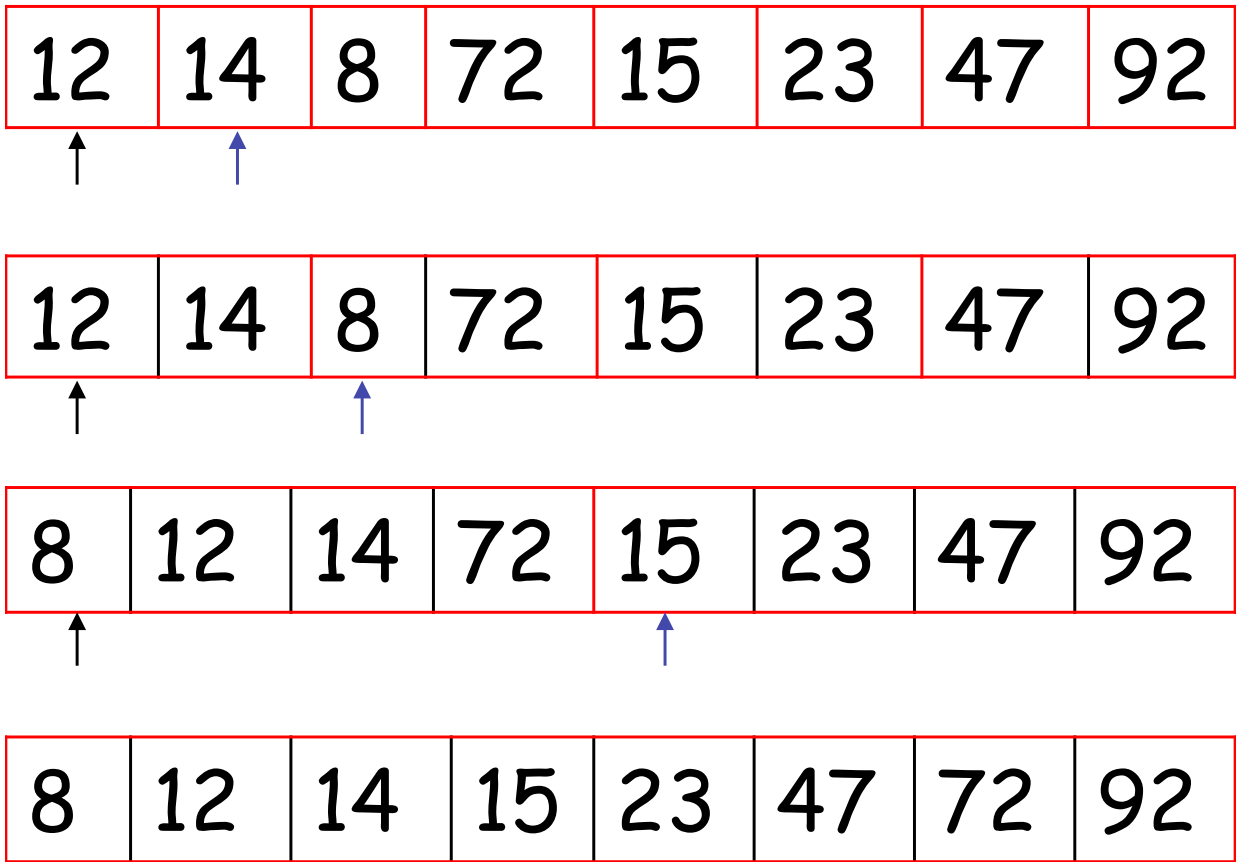
56	87	89	98	21	34	43	65

56	87	89	98	21	34	43	65

21	34	43	56	65	87	89	98

Sort 12, 14, 8, 72, 15, 23, 47, 92

合并


12 14 8 72 15 23 47 92
12 14 8 72 15 23 47 92
8 12 14 72 15 23 47 92
8 12 14 15 23 47 72 92

代码

```
void MS (ElementType A[], ElementType Tmp[], int left, int right)
{
    int center;

    if (left < right)
    {
        center = (left + right)/2;
        MS (A, Tmp, left, center);
        MS (A, Tmp, center + 1 , right);
        merger (A, Tmp, center + 1, right);
    }
}

void MergeSort (ElementType A[], int N)
 {
    ElementType *Tmp;

    Tmp = malloc (N*sizeof (ElementType);
    if (Tmp != NULL)
    {
        MS (A, Tmp, 0, N-1);
        free (Tmp);
    }
}
```

合并

合并 A[left..right-1], A[right..rightEnd] 到 Tmp[left..rightEnd] 然后复制回 A[left..rightEnd]

复杂度分析

$$\begin{cases} T(1) = 1 & \text{trivial case} \\ T(n) = 2T(n/2) + n & \text{recursive step} \end{cases}$$

除以 n , T(n)/n = 2T(n/2)/n + n/n

T(n)/n = T(n/2)/(n/2) + 1

然后代入较小的n值:

T(n/2)/(n/2) = T(n/4)/(n/4) + 1

T(n/4)/(n/4) = T(n/8)/(n/8) + 1

...

T(2)/2 = T(1)/1 + 1

现在添加所有的等式——大部分的项都被抵消了

T(n)/n = logn + T(1)/1

Thus T(n) = O(nlogn)

快速排序

- 由Tony Hoare在1960年发明
- 最坏情形： $O(n^2)$
- 平均情况:O(nlogn)，且常系数小
- 通常被认为是最好的排序算法

QuickSort

```
QuickSort(int A[], int n)
{
    if (1 == n)
        return;

    ;
     listA              listlist;

    return {QuickSort(leftside), pivot,
                QuickSort(rightside)};
}
```

快速排序举例

- Quicksort: 13 22 79 18 2 8 42 50 6
- 选取枢纽元，比如，22
- 返回 (quicksort(13,18,2,8,6), 22, quicksort(79,42,50))
  - Quicksort: 13 18 2 8 6
  - 选取枢纽元 8
  - 返回 quicksort(2,6), 8, quicksort(13,18)
- Quicksort: 79 42 50, etc.
- 我们希望的是：给定枢纽元，左右两边list大小相等

选择枢纽元

- 总是选择第一个元素。 如果数组是完全随机的，这就是可行的
- 但若情况不理想，一侧将填满剩下的所有元素
- 规模折半 $\rightarrow$ 删除第一个元素
- $\Theta(n)$ 递归调用 $\rightarrow$ $\Theta(n^2)$ 时间，一个坏的主意
- 在列表中随机选择枢纽元。 在选择枢纽元的随机性方面堪称完美。 但随机选取过程的代价是昂贵的。
- 选择中值。 可以在线性时间内完成，但消耗仍然较大。

Median of 3

- 选择 "median of 3"
  - 几乎和中位数一样好，但是很简单
  - 查看首元素，中间元素和尾元素，选择大小居中的那个

划分过程

```
int comp_median3 (int a[], int left, int right)
{
    int ctr = (left+right)/2;

    if (a[ctr] < a[left]) swap (a,left,ctr);
    if (a[right] < a[left]) swap (a,right,left);
    if (a[right] < a[ctr]) swap (a,ctr,right);

    //      right-1
    //
    swap (a,ctr,right-1);

    return (a[right-1]);
}
```

划分过程

1. 用数组中最右边的元素交换枢纽元
2. 左指针从第一个元素开始，一直向前移动，直到找到比枢纽元更大的元素为止
3. 右指针从第二个位置开始，向后移动直到找到小于枢纽元的元素
4. 如果左指针过右，则与最后一个元素交换并停止；否则，交换指针指向的元素，并重复移动

划分过程

```
QuickSort (A)
{
    i = left; j = right - 1;
    while (1)
    {
        while a[++i] < pivot
            ;
        while a[--j] > pivot
            ;
        if (i < j)
            swap a[i] and a[j];
        else
            break;
    }
    swap a[i] and pivot;
}
```

QuickSort举例

18	8	14	6	35	2	7	0	91	78	80

QuickSort举例

18	8	14	6	35	2	7	0	91	78	80

80	8	14	6	35	2	7	0	91	78	18

QuickSort举例

18	8	14	6	35	2	7	0	91	78	80

80	8	14	6	35	2	7	0	91	78	18

0	8	14	6	35	2	7	80	91	78	18

QuickSort举例

18	8	14	6	35	2	7	0	91	78	80

80	8	14	6	35	2	7	0	91	78	18

0	8	14	6	35	2	7	80	91	78	18

0	8	14	6	7	2	35	80	91	78	18

QuickSort举例

18	8	14	6	35	2	7	0	91	78	80

80	8	14	6	35	2	7	0	91	78	18

0	8	14	6	35	2	7	80	91	78	18

0	8	14	6	7	2	35	80	91	78	18

0	8	14	6	7	2	18	80	91	78	35

QuickSort举例

18	8	14	6	35	2	7	0	91	78	80

80	8	14	6	35	2	7	0	91	78	18

0	8	14	6	35	2	7	80	91	78	18

0	8	14	6	7	2	35	80	91	78	18

0	8	14	6	7	2	18	80	91	78	35

0	8	14	2	7	6	18	35	91	78	80

QuickSort举例

18	8	14	6	35	2	7	0	91	78	80
80	8	14	6	35	2	7	0	91	78	18
0	8	14	6	35	2	7	80	91	78	18
0	8	14	6	7	2	35	80	91	78	18
0	8	14	6	7	2	18	80	91	78	35
0	8	14	2	7	6	18	35	91	78	80
0	2	14	8	7	6	18	35	78	91	80

QuickSort举例

18	8	14	6	35	2	7	0	91	78	80

80	8	14	6	35	2	7	0	91	78	18

0	8	14	6	35	2	7	80	91	78	18

0	8	14	6	7	2	35	80	91	78	18

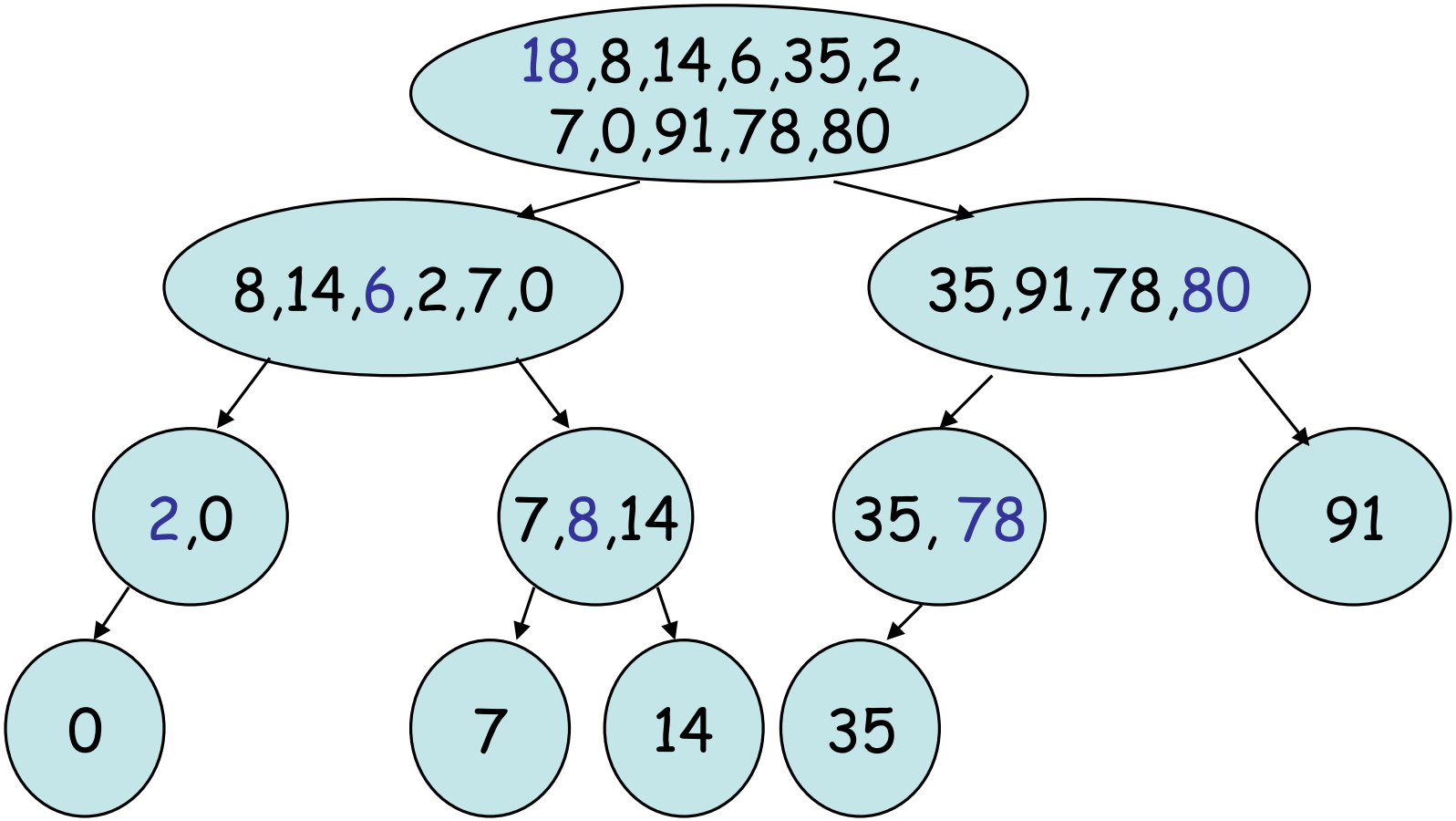
0	8	14	6	7	2	18	80	91	78	35

0	8	14	2	7	6	18	35	91	78	80

0	2	14	8	7	6	18	35	78	91	80

0	2	6	8	7	14	18	35	78	80	91

枢纽元与划分

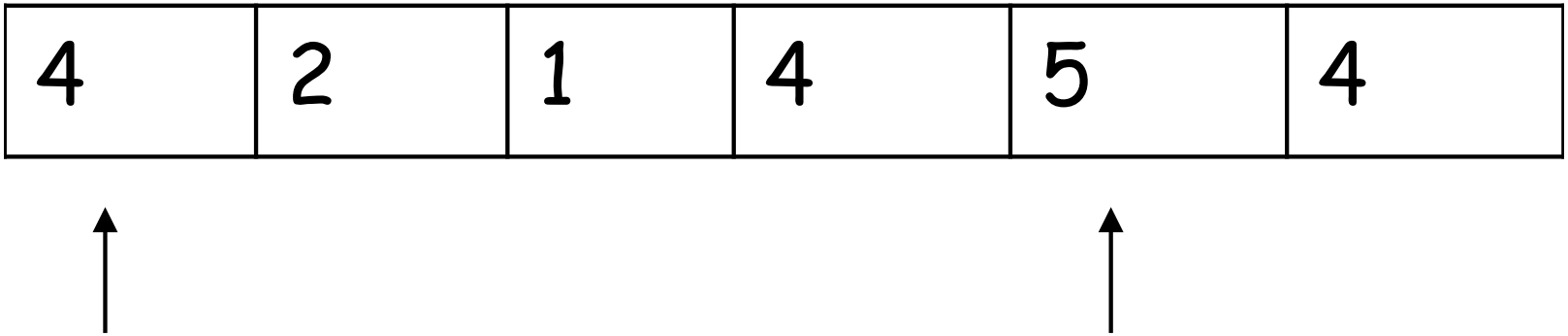


0	2	6	7	8	14	18	35	78	80	91

编程中的细节问题

- 为什么不像下面那样做?
  `while (a[i] < piv) i++;` `while (a[j] > piv) j--;`
- 考虑这个情况: 4 2 1 4 5 4, 三值均为 4
- 然后i, j不动, 4和4交换, 如此往复, 形成一个无限循环
- 为什么不这样? `while (a[i] <= piv) i++;` `while (a[j] => piv) j--;`
- 考虑 444444, i 总是向右移动

枢纽元举例



枢纽元编程

- 保留主体代码，但在选择中值时，排序左、右、中部分
- 好处：
  - 没有无限循环
  - i, j总是向前移动
  - 对同值的初始输入保持平衡

快速排序时间复杂度分析

令 i 为右侧部分大小

然后 T(n) = 1 + n + T(i) + T(n-i-1)

1 为选择枢纽元时间，n 为分区时间

右边是 n - left size - pivot的大小

最坏情况是什么？ i always = 0

$$T(n) = 1 + n + 1 + T(n-1)$$

$$= 1 + n + 1 + (1 + n - 1 + 1 + T(n-2)$$

$$= n + n - 1 + (n - 2 + T(n-3)) = n^2$$

最好情况？ i 总是为 n/2

$$T(n) = 1 + n + 2 * T(n/2) = 1 + n + 2 * (1 + n/2 + 2 * T(n/4))$$

$$= n \log n$$

快速排序最坏情况

另一种写法：

$$T(n) = T(n-1) + n$$

$$T(n-1) = T(n-2) + n - 1$$

$$T(n-2) = T(n-3) + n - 2$$

$$\dots$$

$$T(2) = T(1) + 2$$

累加所有等式，逐项相消：
右侧为：

$$T(n) = n + n - 1 + n - 2 + \dots + 2 + T(1) = \Theta(n^2)$$

快速排序最好情况

$$T(n) = 2T(n/2) + n$$

$$T(n)/n = T(n/2)/(n/2) + 1$$

$$T(n/2)/(n/2) = T(n/4)/(n/4) + 1$$

$$\dots$$

$$T(2)/1 = T(1)/1 + 1$$

$$T(n)/n = T(1)/1 + \log n$$

$$T(n) = n \log n$$

快速排序平均情况

- 随机的主元将列表平均分为一半
- 期望为 $\rightarrow n \log n$
- 平均情况还不够
- 假设枢纽元算法总是选择最小或最大的
- 左侧总是为空或者包含全部元素 $\rightarrow$ 平均来说是一半!

  $T(n) = T(L) + T(R) + n$

  需要估计 $T(L)$, $T(R)$

  大小从 0 到 n-1

  平均: $T(L) = T(R) = \frac{T(0)+\ldots+T(n-1)}{n}$

快速排序平均情况

- 对于给定的枢纽元选法，我们有average
  $T(i) = \frac{\sum_{j=0}^{n-1} T(j)}{n}$
- 所以 $T(n) = \frac{2}{n} * (\sum_{j=0}^{n-1} T(j)) + n$
- 乘以 n:
  $nT(n) = 2 * (\sum_{j=0}^{n-1} T(j)) + n^2 \quad (1)$
- 代入 n-1 到 n:
  $(n-1)T(n-1) = 2 * (\sum_{j=0}^{n-2} T(j)) + (n-1)^2 \quad (2)$
- 记住 $(n-1)^2 = n^2 - 2n + 1$
  (1) - (2) = nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n-1. 减 1，两边加 (n-1)T(n-1) :
  nT(n) = (n+1)T(n-1) + 2n, 除以 n(n+1):
  T(n)/(n+1) = T(n-1)/n + 2/(n+1)

快速排序平均情况

- 已经得到： $T(n)/(n+1) = T(n-1)/n + 2/(n+1)$
- 现在：
  - $T(n-1)/(n) = T(n-2)/(n-1) + 2/(n)$
  - $T(n-2)/(n-1) = T(n-3)/(n-2) + 2/(n-1)$
  - $T(n-3)/(n-2) = T(n-4)/(n-3) + 2/(n-2)$
  - ...
  - $T(2)/3 = T(1)/2 + 2/3$
- 多数项抵消：

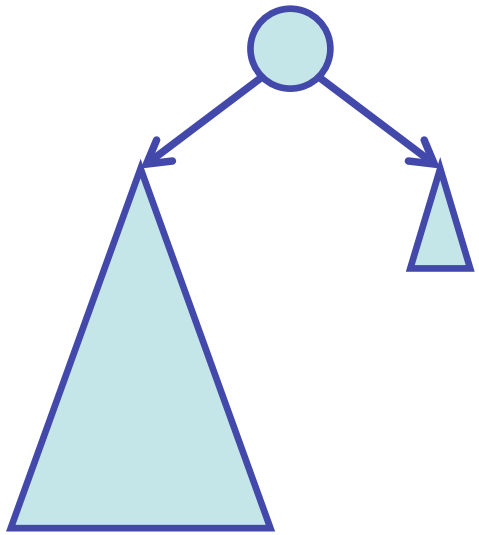
  $T(n)/(n+1) = 2(1/(n+1) + 1/n + \cdots + 1/3)$

  $=2H_{n+1} = \log n$, 调和级数
- 即 $T(n)/(n+1) \rightarrow T(n) = n\log n$

快速排序小结

- 快速排序平均情况 O(n log n)
- 可以表明n^2情况可能性极小
- 举个例子，如果非常不幸运，总是分成 9/10*n 和 1/10*n 两部分，结果仍然是 O(n log n)

$$T(n) = T(0.9n) + T(0.1n) + n$$



qsort() 函数

- C语言库中的函数 void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*)) 给一个数组排序
- 参数
  - base - 指向要排序的数组的第一个元素的指针
  - nitems - base指向的数组的元素个数
  - size - 数组中每个元素的字节大小
  - compar - 比较两个元素的函数
- 返回值
  - 这个函数不返回任何值

举例

```c
#include <stdio.h>
#include <stdlib.h>

int values[] = { 88, 56, 100, 2, 25 };

int cmpfunc (const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}

int main()
{
   int n;

   printf("Before sorting the list is: \n");
   for( n = 0 ; n < 5; n++ )
   {
      printf("%d ", values[n]);
   }

   qsort(values, 5, sizeof(int), cmpfunc);

   printf("\nAfter sorting the list is: \n");
   for( n = 0 ; n < 5; n++ )
   {
      printf("%d ", values[n]);
   }

  return(0);
}
```

混合排序算法

- 元素较少的数组上插入排序非常快
- 快速排序/归并排序一般在长度>20的数组上使用
  - 元素较少使用快速排序 过多的开销导致速度不会很快
  - 不会消耗很多时间，但是相比插入排序，计算量较大
- 对于元素较少的数组，调用插入排序即可
  - sort(int[]) and sort(Object[]) in Arrays and Collections all do this
  - Both quicksort and mergesort

选择所需的排序算法

- 最常用：
  - 插入排序 Insertion
  - 希尔排序 Shell
  - 归并排序 Merge
  - 快速排序 quick
- 插入排序在数组元素<20或者数组大部分已经排序时非常快
- 希尔排序易于代码编写而且运行很快
- 在外部排序，而且需要稳定排序时，归并排序理论上很快
- 电子邮件先按姓名排序，然后按日期排序
- 快速排序一般都很快

Roadmap

基于比较的排序算法分析

- 假设我们有 a, b, c 三个需要排序的元素
- 怎么做？
- 自然想到：比较 再比较
- 在每一点上做一些测试（比较）
  - 二叉树 视结果而定
  - 叶子=解决方案
- 决策树
  - 画一个决策树
  - 每个节点 = 一个状态
  - 每个状态 = 一种可能性
- 这样一棵树的最小深度是多少？

基于比较的排序算法分析

Lemma

深度为d的二叉树有 $\leq 2^d$ 个叶子

Lemma

有l个叶子的二叉树深度一定 $\geq$ (log l)向下取整

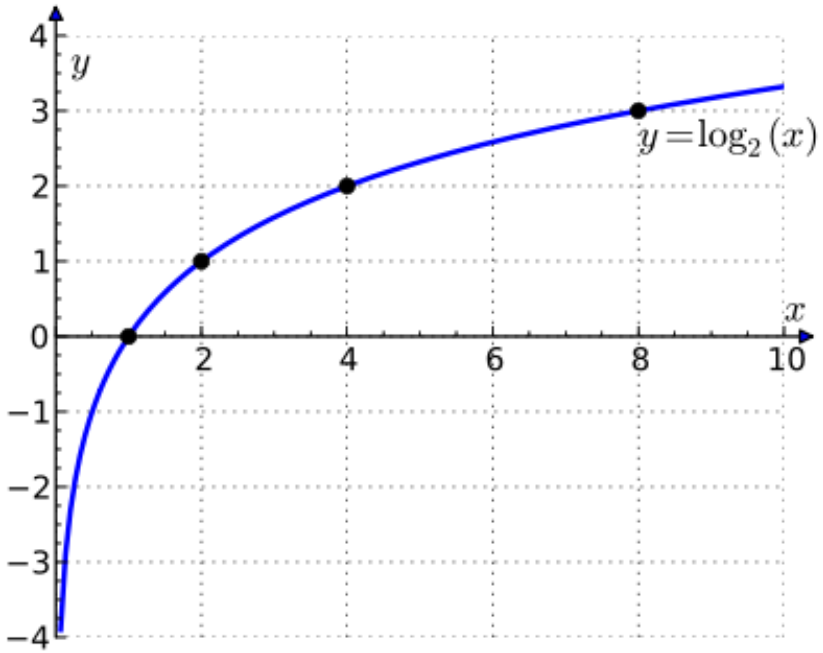
Lemma

只使用比较的排序算法在最坏情形下需要 $\geq$ log(n!)次比较

n! 次可能的输入 $\rightarrow$ n! 次可能的输出 $\rightarrow$ n! 叶子 $\rightarrow$ 深度 $\geq$ log(n!)

Log函数

- 凸性： $\forall x_1, x_2 \in X, \forall t \in [0,1]$, $f(tx_1 + (1-x)t_2) \leq tf(x_1) + (1-t)f(x_2)$
- 对数函数单调凸



分析

Lemma

任何基于比较的排序算法都是 $\Omega(n \log n)$

$$\log(n!) = \log(n * (n-1)(n-2) * 1) = \sum_{i=1}^{n} \log i$$
$$= \sum_{i=1}^{n/2} \log i + \sum_{i=n/2+1}^{n} \log i \geq \sum_{i=n/2+1}^{n} \log i$$
$$\geq \frac{n}{2} * \log \frac{n}{2}$$
$$= n/2 * (\log n - 1) \rightarrow \log(n!) = \Omega(n \log n)$$

Theorem

任何基于比较的排序算法都是 $\Theta(n \log n)$

$$\log(n!) \leq \log n^n = n \log n \rightarrow \log(n!) = 0(n \log n)$$

结论

- 决策树具有最小深度 $n \log n$
- 另一种理解方式：最初有 $\Pi_1 = n!$ 可能的输入序列，each comparison at best sets $\Pi_{i+1} = \Pi_i/2$
- 在 $\Pi_k = 1$ 之前，需要做多少次？
  $\log \Pi_1 = \log n! = n \log n$ 次
- 另一种理解方法：每个比较给出1个信息位
- 有 $n!$ 种输入可能
- 为了查找条目索引，需要 $\log n!$ bits $= n \log n$

其它排序模型

- 我们可以比 nlogn 做得更好吗？ 如果我们有额外的信息，我们就能做得更好。
- 以前的模型假定2个项目之间的每一个比较都需要一定的时间。
- 桶排序呢？ 额外信息=值的上界
- 桶排序采用M型比较，类似于哈希
  - 从列表中遍历并找到最大值M
  - 对M个桶进行桶排序
- 对某些输入，桶排序可以很快
  - 整数范围小
  - 但要获得 $\Theta(M)$ 内容空间

Roadmap

External Sort

- 问题：用1MB的RAM数据排序1GB数据
- 我们在哪里需要：
  - Data requested in sorted order (ORDER BY)
  - 分组操作所需
  - 排序合并连接算法的第一步
  - 去除重复
  - b树索引的批量加载

External Sort Idea

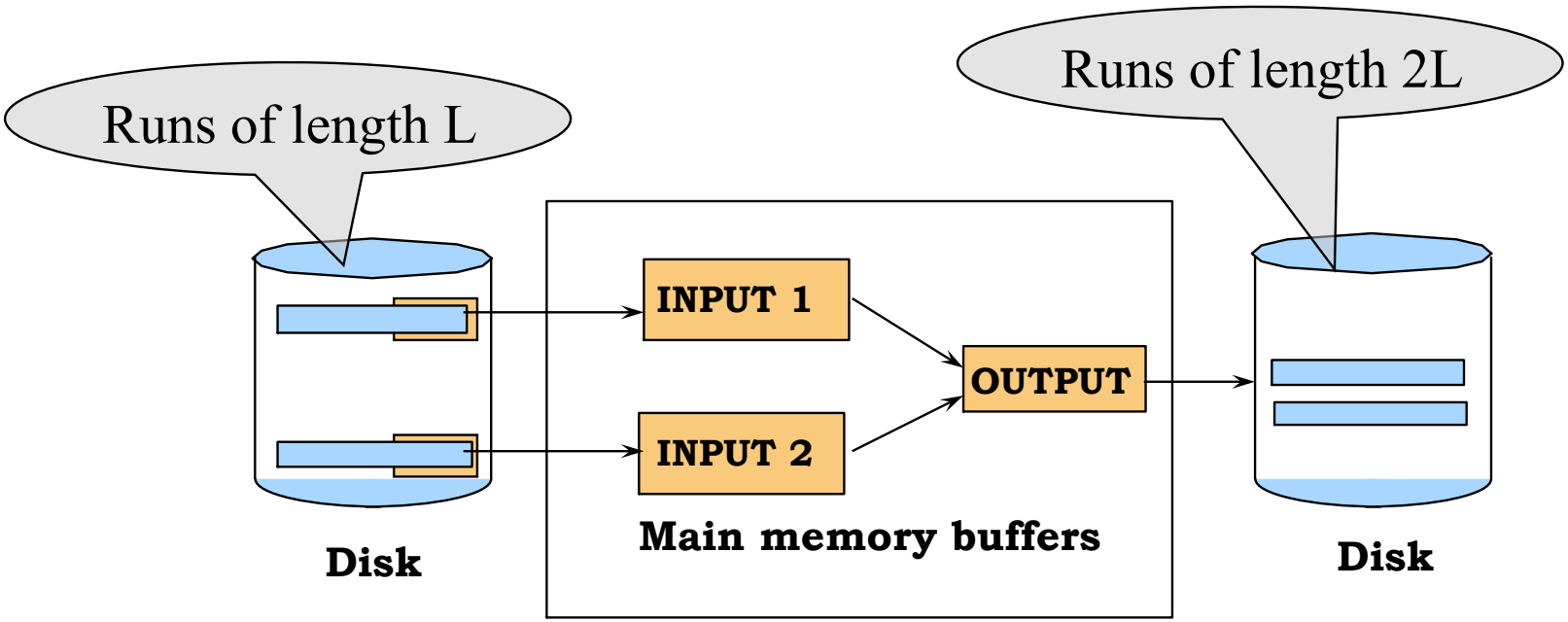
- 文件只能逐块读取，没有可用的随机访问
- 想法：使用小内存来缓冲文件数据，反复排序合并直到形成一个独立文件
  - 从文件读取一些数据到缓冲区
  - 对缓冲区中的数据进行排序
  - 将数据写入临时文件
  - 将一些从临时文件中排序的数据读到缓冲区中
  - 归并读入的数据
  - 将合并后的数据写入临时文件（run）
- 文件I/O比内存操作慢得多

Example

- 6个数字:8, 1, 6, 3, 4, 5, small memory for only 3 numbers
- 在内存的帮助下将数字排序
  - 读 8, 1, 6 ， 排序并写 1, 6, 8 到外部存储
  - 读 3, 4, 5 排序， 写 3, 4, 5
- 归并排序数据
  - 读 1 和 3，比较选择较小的一个， 写到 1 外部存储
  - 读 6 与 3进行比较， 写 3
  - 读 4 与 6进行比较， 写 4；重复直到所有均已归并

Two Way Merge Sort

- 在RAM中需要3个缓冲区
- 传递1：读一个块，整理，写
- 传递2，3，…，etc.：归并，写入 不需要读完整的子列表再合并



二路外部归并排序

假设块大小是 B = 4Kb
Step 1, 运行长度 L = 4Kb
Step 2, 运行长度 L = 8Kb
Step 3, 运行长度 L = 16Kb = 2^{3-1} * 4Kb
...
Step 9, 运行长度 L = 1MB = 2^{9-1} * 4Kb
...
Step 19, 运行长度 L = 1GB = 2^{19-1} * 4Kb
需要对磁盘数据进行19次迭代以排序 1GB的数据

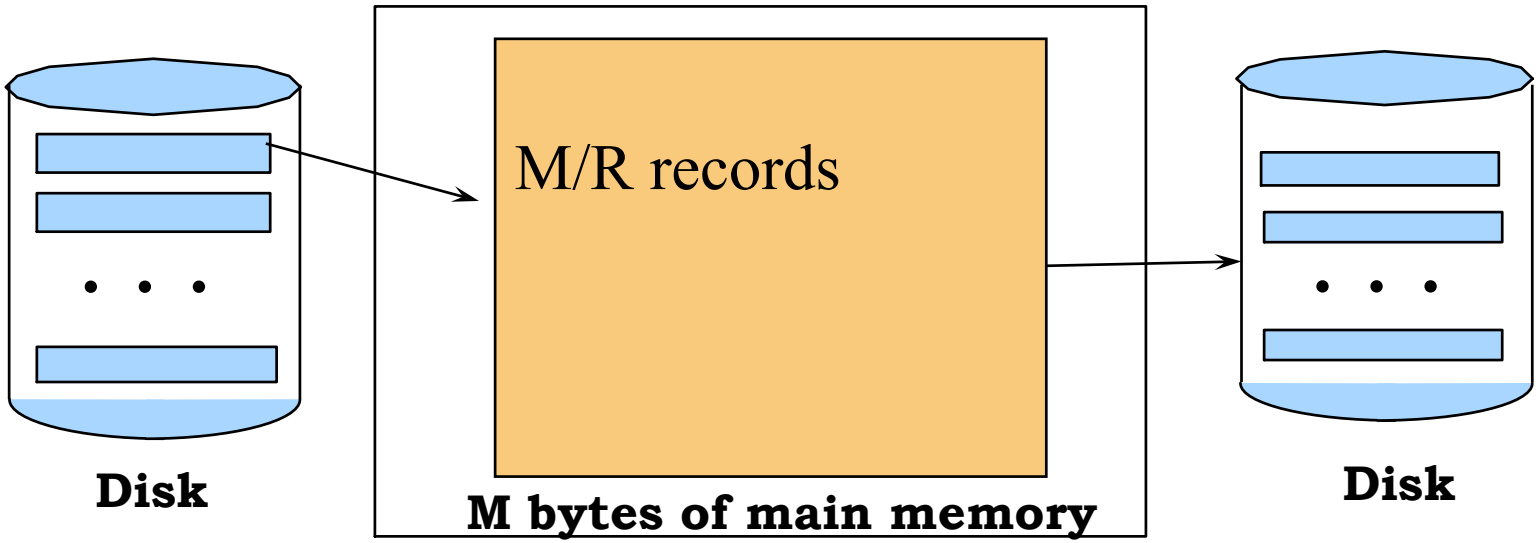
成本模型

我们能做得更好吗？

- B：块的大小（= 4KB）
- M：主存的大小（= 1MB）
- N：文件中的记录数
- R：一个记录的大小
- L：当前被排序的子列表的大小

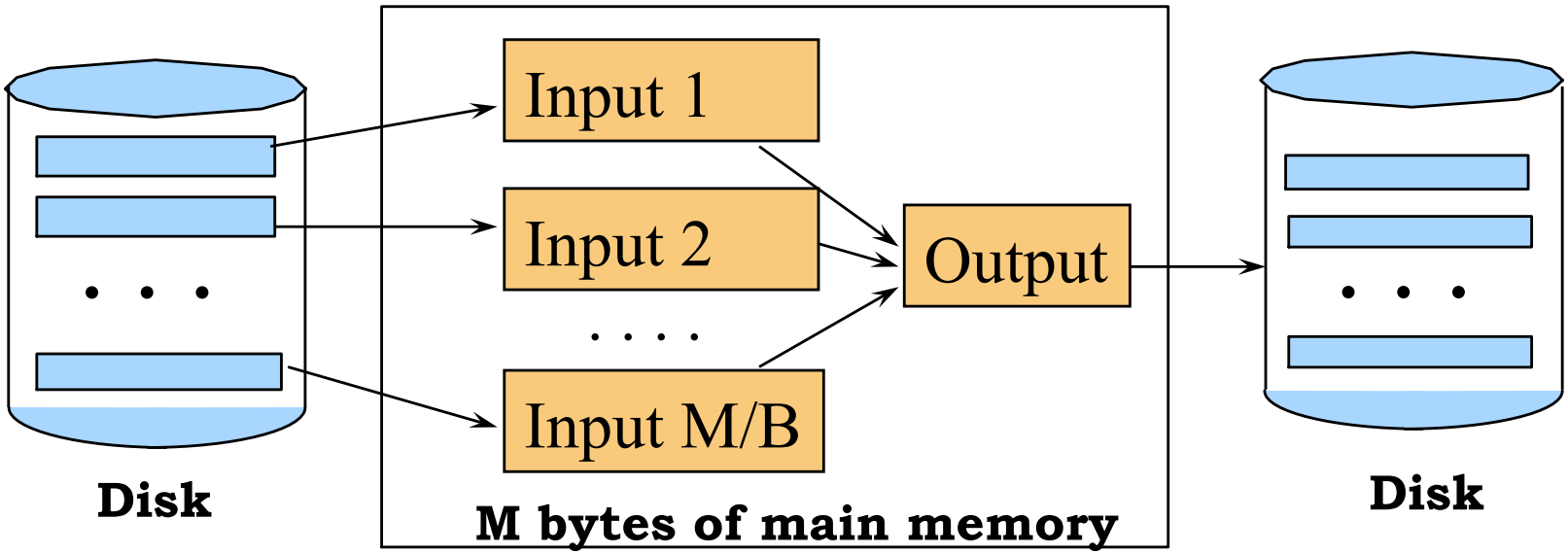
多路外部归并排序

Phase one: 在内存中加载m字节，排序
结果: N*R/M lists of length M bytes (1MB)
L = M



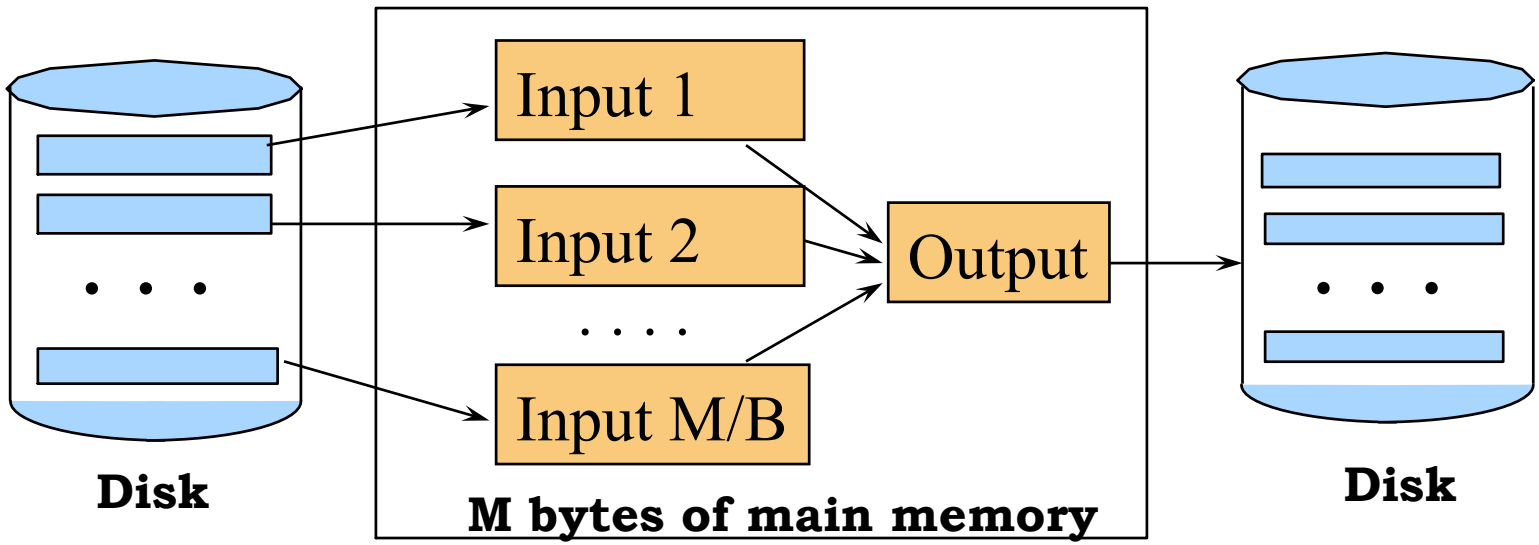
Phase 2

- 归并 M/B - 1 lists into a new list
  M/B-1 = 1MB / 4kb ≈ 249
- 结果: lists of size M *(M/B - 1) bytes
  L = 249 * 1MB ≈ 250 MB



Phase 3

- 归并 M/B - 1 lists into a new list
- 结果: lists of size $M * (M/B - 1)^2$ bytes
- L = 249 * 250 MB $\approx$ 62,500 MB = 625 GB



Cost of External Merge Sort

- Amount sorted in P passes: $M * (M/B)^{P-1}$
- 排序 大小为N records为R 的次数: $\log_{M/B}(N * R/M) + 1$
- 我们能用 1MB RAM 排序多大的数据? 1 次: 1MB, 2 次: 250MB (M/B = 250), 3 次: 625GB
- 时间:
  假设读、写块 ~ 10 ms = .01 s
  每次: 读写所有数据
  每次: 2*625GB/4kb*.01s = 2*1562500s = 2*26041m = 2*434 = 2*18 days = 36 days

Cost of External Merge Sort

- 传递次数： $\log_{M/B}(N*R/M)+1$
- 我们能用 10MB RAM 排序多大的数据 (M/B = 2500)？
  - 1次： 10MB
  - 2次： 10MB * 2500 = 25,000MB = 25GB
  - 3次： 2500 * 25GB = 62,500GB

Cost of External Merge Sort

- 传递次数： $\log_{M/B}(N*R/M)+1$
- 我们能用 100MB RAM 排序多大的数据 (M/B = 25000)？
  - 1次： 100MB
  - 2次： 100MB * 25,000 = 2,500,000MB = 2,500GB = 2.5TB
  - 3次： 25,000 * 2.5TB = 62,500TB = 62.5PB

小结

- 简单排序算法比较和交换相邻的元素，一次只能修正一对逆序，因而运行缓慢
- 基于比较的排序算法使用决策树决定元素排列关系，二叉树的叶子结点有n!个，树的高度为0(n log n)
- 最优排序的时间复杂度为0(n log n)，如归并排序、快速排序和堆排序
- 外部排序利用较小的内存缓存解决大量外部数据的排序问题，可以使用排序—归并的算法

实验8

- 7.35，实现希尔排序。使用不同的增量序列，度量在不同序列的增量下，排序算法的时间性能。
- 如何找到10000个整数中第5大的数？假定内存只能一次容纳1000个数。编程实现你的算法，并分析时间复杂度。如果内存只能存放10个数呢？你的算法是否仍然适用？