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

- Trivial case, if N is 1, already sorted, return
- Divide an array of n into two: an array of n-1, and the last element
  - Solve the sub-problems
  - Insert the last element into the sorted array of n-1
- Can we do the last step more efficiently? What about binary search?

冒泡排序

- Idea: move up the big numbers to the top by swapping

```
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]
```

- Each pass moves the biggest number to its right place
  - Only n passes
  - Fewer than n swaps each time
  - $O(n^2)$

Simple slow algorithms

- What's bad about "simple" slow algorithms?
- All compare only adjacent items
- Bubble sort is obviously like this
- Insertion sort can be viewed like this

复杂度分析

Theorem

Any adjacent-comparing algorithm is $\Theta(n^2)$

证明

Proof: if for pair (i, j), such that i < j, have a[i] > a[j], call this an inversion.
How many possible (ordered) pairs? The answer is n*(n-1)/2. Consider some ordering O and reverse O^{-1}, (i, j) will be inverted in either O or O^{-1}
If O is random, then expect half of pairs to be inverted. So the expected number of inversions: $\frac{n*(n-1)/2}{2} = \Theta(n^2)$
Why does this matter? Any swap of adjacent items fixes exactly one inversion. Algorithm that just swaps adjacent items is $\Theta(n^2)$ average time

Roadmap

ShellSort

- How to sort faster? Must swap farther-apart-items
- Shellsort: Donald Shell, 1959. Sub-quadratic, but only a little harder than insertion sort
  - Explicit idea: swap items that are far apart
  - Do InsertionSort with several "increments"
  - Lastly, sort on distance one
  - Eg: 8, 4, 2, 1, with the decreasing sizes and the last is 1
- Turns out: later sorts don't undo earlier ones

举例

- Consider a list:
  4 63 20 3 61 40 1 65 10 5 64 30 2 62 50
- Let's 3-sort it:
  1 61 10 2 62 20 3 63 30 4 64 40 5 65 50
- Now 2-sort it:
  1 2 3 4 5 20 10 40 30 61 62 63 50 65 64
- Now 1-sort it:
  1 2 3 4 5 10 20 30 40 50 61 62 63 64 65
- Look at 81 94 11 96 12 35 17 95 28 58 41 75 15
- Do 5, 3, 1 sorts

ShellSort代码

```
Void ShellSort (ElementType A[], int N)
{
    int i, j, gap;
    ElementType curr;

    for (gap = N/2; gap > 0; gap /= 2)
        for (i = gap; i < N; i++)
        {
            curr = A[i];
            for (j = i; j >= gap &&
                 curr < A[j-gap]; j -= gap)
                A[j] = curr;
        }
}
```

ShellSort的增量

- First increments from Shell: 1 2 4 8 16 n/2 (in reverse order) $\rightarrow \Theta(n^2)$
- But turns out: $\Theta(n^{1.5})$ average time by number theory
- $\Omega(n^2)$: second-to-last is 2-sort
  - Put big values in even slots,
  - 1-sort will invert $\Theta(n^2)$
- $O(n^2)$: h-sort is on n/h elements

Shell排序的下界

- Must also show it's $\Omega(n^2)$
- Enough to find a sequence type that's $\Theta(n^2)$, but not just one sequence!
- Choose n small numbers in even slots, n larges in odd slots:
  10 0 11 1 12 2 13 3
- All sorts before 1-sort leave evens in evens, odds in odds
- In 1-sort, ith smallest (from 0) in position 2i+1, not in position i
  - Each must move distance i+1
  - Each of n/2 must move roughly $\sum_{i=0}^{n/2} i + 1 \rightarrow n^2$

Shell排序的分析

A pass consists of h subsets of n/h elements
let $n = 2^k$, 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: $1, 3, 7, \ldots, 2^k - 1$, consecutive increments are relatively prime
- Turns out to be enough to help
  - Can show: Hibbard = $O(n^{3/2})$ for worst case
  - Can't show but believe: $O(n^{5/4})$ for average case
- Sedgewick: $O(n^{4/3})$ in worst case, maybe: $O(n^{7/6})$ on average, sequence: $9 * 4^i - 9 * 2^i + 1$ or $4^i - 3 * 2^i + 1$

Hibbard增量的复杂度分析

Theorem

ShellSort with Hibbard increments has the worst case complexity of $\Theta(n^{\frac{3}{2}})$

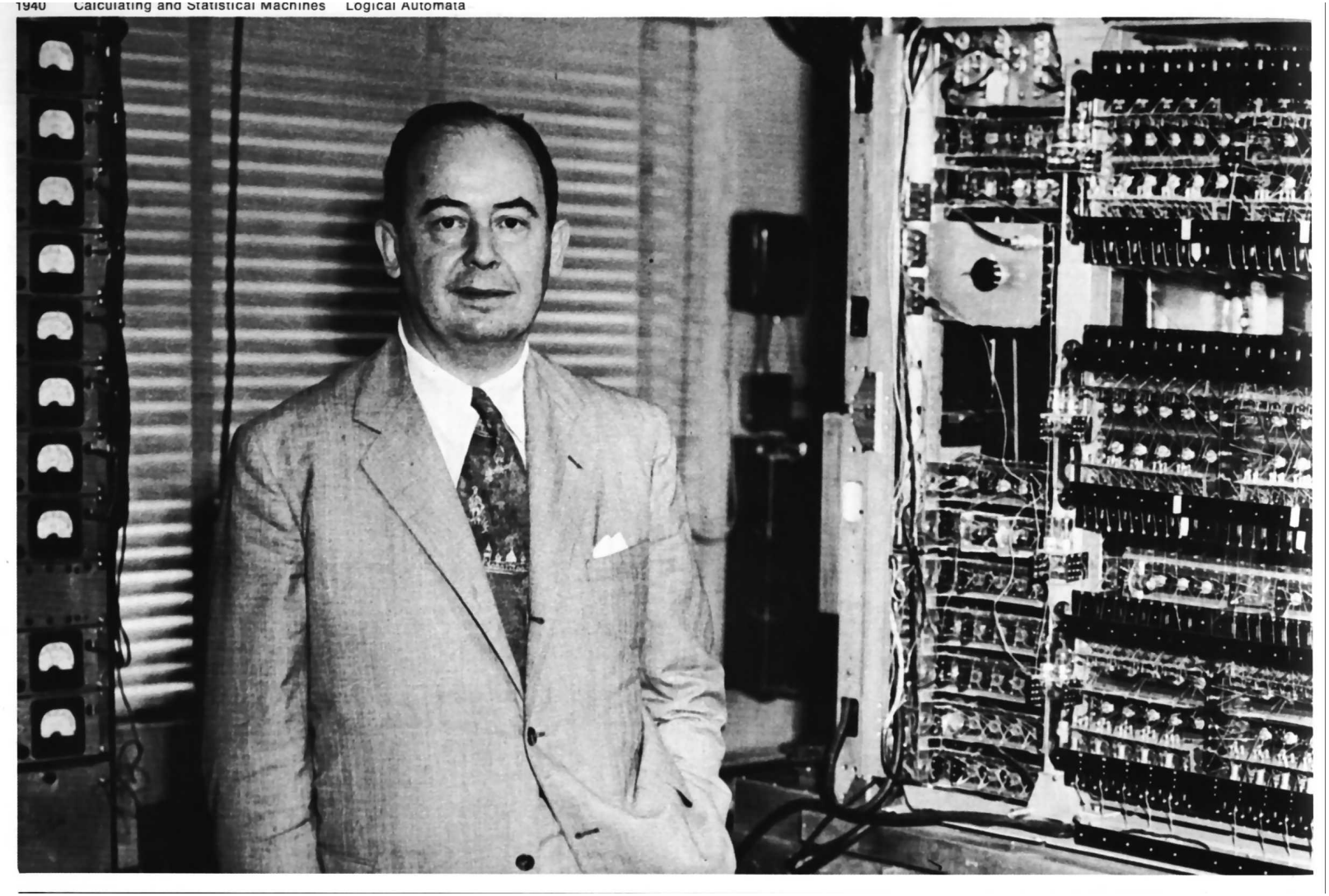
Proof Idea: Before h_k-sort, the array is h_{k+1}-sorted. For two elements that are a positive linear combination of h_{k+1} and h_{k+2} apart, they are in correct order. In each pass, only positions that cannot be expressed as the linear combinations of the previous gaps have to be sorted. That needs $h_k N$ operations. Then we may separate h_k's by comparing with $\sqrt{N}$, and sum up the cost of each pass.

Roadmap

归并排序

- Basic idea: divide and conquer, using recursive steps
  - MergeSort (left half);
  - MergeSort (right half);
  - merge 2 halves;
  - Trivial case: if length 1, just return
- Very simple recursive algorithm introduced by John von Neumann in 1945, who invented Game Theory, worked on QM, and also invented "von Neumann architecture", the "Stored-program" computers using single storage for both program and data

冯诺依曼和计算机


1940 Calculating and Statistical Machines Logical Automata

MergeSort

- Divide in half, repeat until trivial case reached
- Merge 2 m-length lists: time 2m
- But where to put the results?
  - Keep in same array, like insert-sort m^2
  - To get linear time, need to put in a new array
- MergeSort is nlogn time but uses O(n) space
  - Big downside: "memory wall"
  - Also: obtaining n space takes time
  - Copying back and forth is slow

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

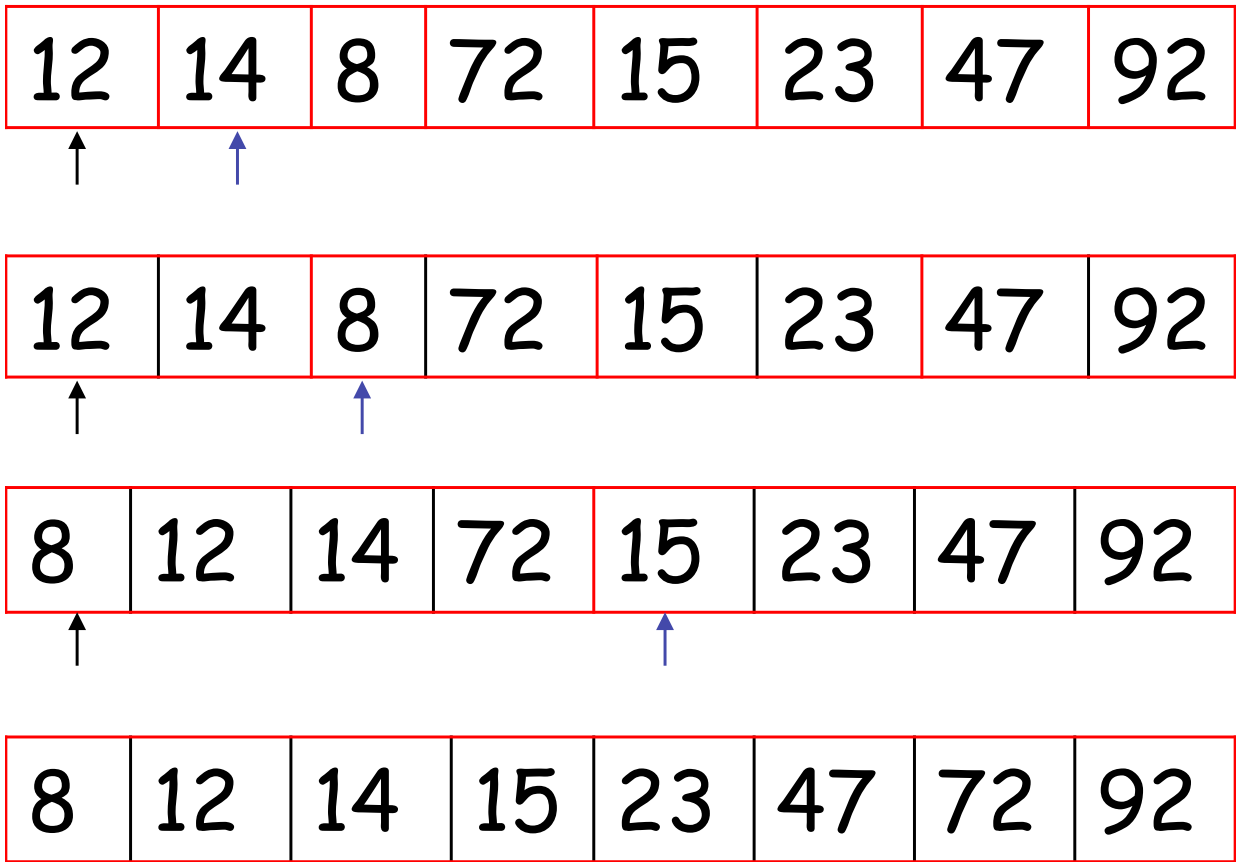
98	89	56	87	34	21	43	65
89	98	56	87	21	34	43	65
56	87	89	98	21	34	43	65
21	34	43	56	65	87	89	98

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
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

Merges A[left..right-1], A[right..rightEnd] to Tmp[left..rightEnd] and then copies back to A[left..rightEnd]

复杂度分析

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

Divide by n, T(n)/n = 2T(n/2)/n + n/n

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

Then plug in smaller values for n:

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

Now add all equations - most terms cancelled

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

Thus T(n) = O(nlogn)

快速排序

- Invented by Tony Hoare in 1960
- Worst case: $O(n^2)$
- Average case: O(nlogn) with small coefficient
- Generally considered best

QuickSort

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

    pick one element as pivot;
    partition list A into left side, pivot,
            rightside;

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

快速排序举例

- Quicksort: 13 22 79 18 2 8 42 50 6
- Pick pivot, say, 22
- Then return (quicksort(13,18,2,8,6), 22, quicksort(79,42,50)
  - Quicksort: 13 18 2 8 6
  - Pick pivot 8
  - Return quicksort(2,6), 8, quicksort13,18)
- Quicksort: 79 42 50, etc.
- Hopefully: given pivot, left side and right side about equal size

选择枢纽元

- Always pick first element. If array is totally random, this is fine
- But whenever (partially) not, right side gets whole rest of list
- Each divide in half $\rightarrow$ remove first element
- $\Theta(n)$ recursive calls $\rightarrow$ $\Theta(n^2)$ time, very bad idea
- Choose pivot randomly in list. In terms of pivot selection, perfect. But rand() calls are expensive
- Choose median value. Can be done in linear time, but somewhat hard/expensive.

Median of 3

- Choose "median of 3"
  - Almost as good as real median, but easy
  - Look at first, last middle, pick median

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);

    // put pivot in position right-1
    // this will be the starting point
    swap (a,ctr,right-1);

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

划分过程

1. Swap the pivot with the rightmost element in the array
2. The left pointer starts from the first element, moving forward until an element bigger than the pivot is found
3. The right pointer starts from the second last position, moving backward until an element smaller than the pivot is found
4. If left pointer overruns the right, swap with the last element and stop; else swap the elements pointed by the pointers, repeat moving

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

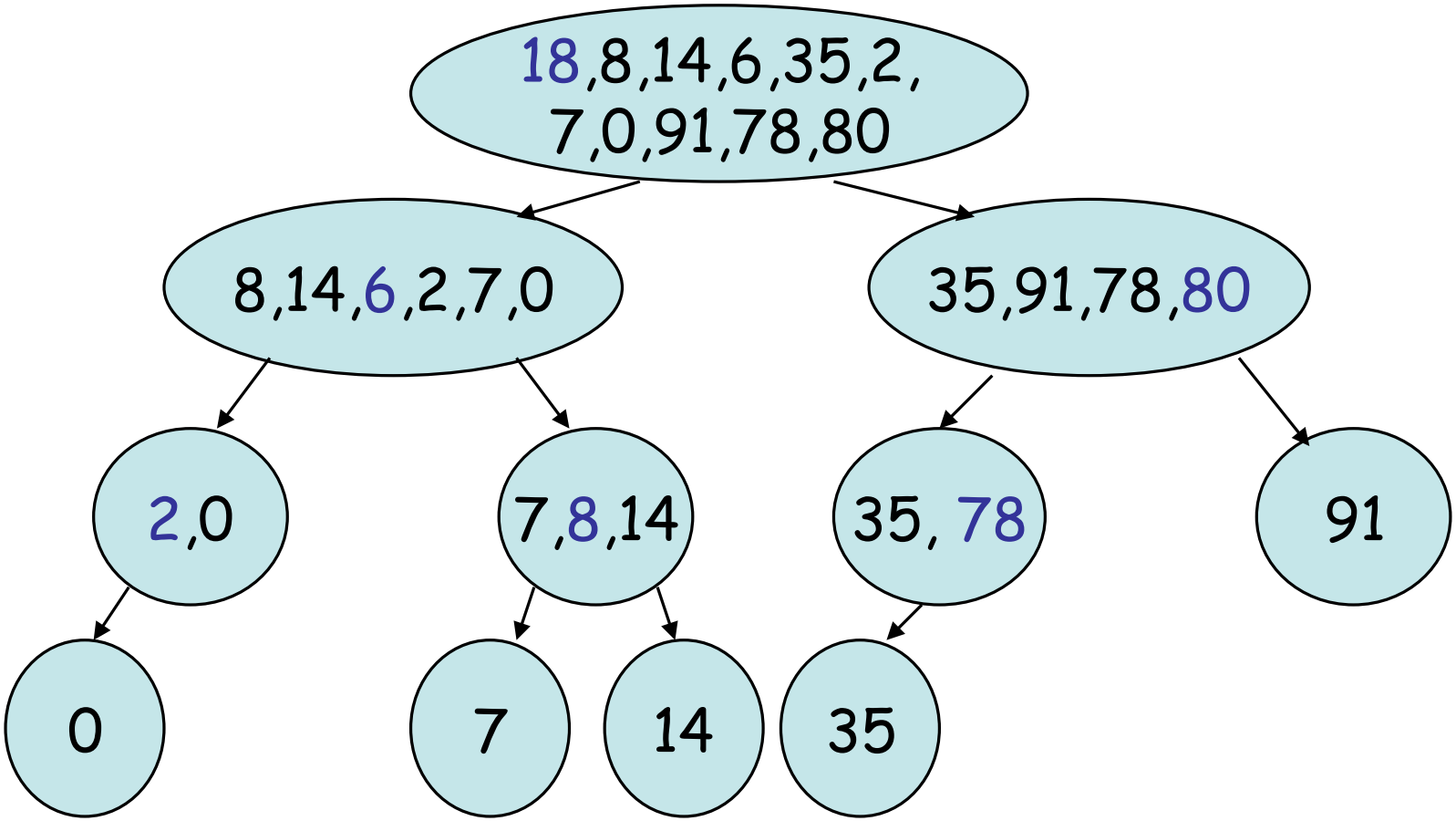
0	8	14	6	7	2	18	80	91	78	35

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

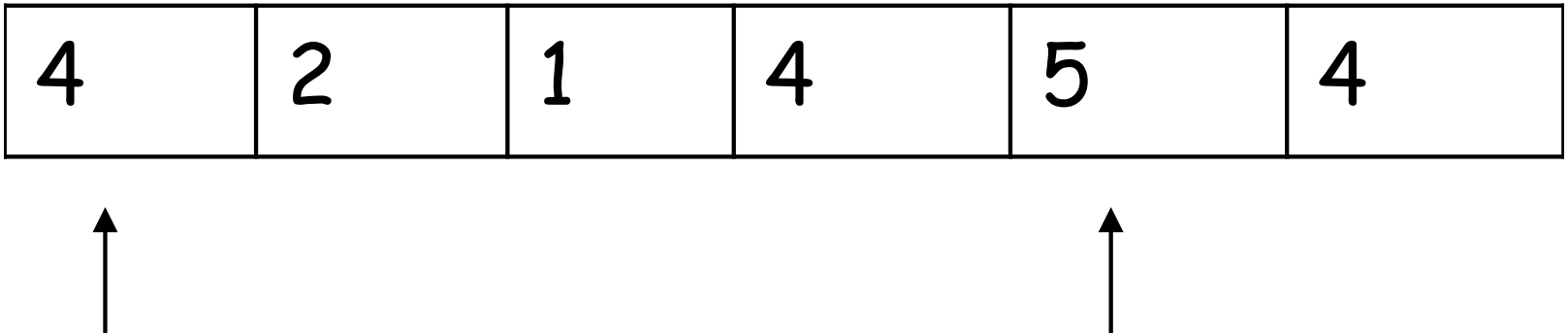


0	2	6	7	8	14	18	35	78	80	91

编程中的细节问题

- Why not say following?
  while (a[i] < piv) i++; while (a[j] > piv) j--;
- Consider case: 4 2 1 4 5 4, all three are 4
- Then i, j don't move, 4 and 4 are switched, and continue, forming an infinite loop
- Then why not this? while (a[i] <= piv) i++; while (a[j] => piv) j--;
- Consider 444444, i goes all way to right, uneven

枢纽元举例



枢纽元编程

- Leave main code as written, but sort left, right, mid at start, when choosing median
- Benefits:
  - No infinite loop
  - Always move i, j forward
  - Stays balanced on all-same-value input

快速排序时间复杂度分析

Let i = size of left side

Then T(n) = 1 + n + T(i) + T(n-i-1)

1 to select pivot, n to partition

Right side is size n - left size - pivot

What's worst case? i always = 0

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

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

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

Best case? i always 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

Another way, write:

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

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

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

$$\ldots$$

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

Now sum all equations, first T(n-1) cancels with second T(n-1), etc.
Left with:

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

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

- Random pivot divides list in half on average
- Hopefully this $\rightarrow$ n log n
- Average isn't enough
- Suppose pivot algorithm always picks either smallest or biggest
- Left side always none or all $\rightarrow$ half on average!

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

  Need to estimate T(L), T(R)

  Size varies from 0 to n-1

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

快速排序平均情况

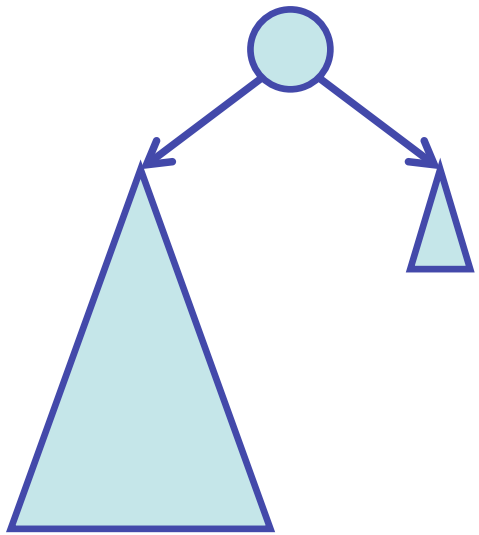
- Given our pivot strategy, we have average value of $T(i) = \frac{\sum_{j=0}^{n-1} T(j)}{n}$
- So $T(n) = \frac{2}{n} * (\sum_{j=0}^{n-1} T(j)) + n$
- Multiply by n:
  $nT(n) = 2 * (\sum_{j=0}^{n-1} T(j)) + n^2 \quad (1)$
- Plug in n-1 for n:
  $(n-1)T(n-1) = 2 * (\sum_{j=0}^{n-2} T(j)) + (n-1)^2 \quad (2)$
- Remember $(n-1)^2 = n^2 - 2n + 1$
  (1) - (2) = nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n-1. Drop 1, add (n-1)T(n-1) to both side: nT(n) = (n+1)T(n-1) + 2n, divide by n(n+1): T(n)/(n+1) = T(n-1)/n + 2/(n+1)

快速排序平均情况

- We have got: T(n)/(n+1) = T(n-1)/n + 2/(n+1)
- Now:
  - 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
- Most terms cancel:
  T(n)/(n+1) = 2(1/(n+1) + 1/n + ⋯ + 1/3)
  $=2H_{n+1} = \log n$, Harmonic
- This is just T(n)/(n+1) $\rightarrow$ T(n) = nlogn

快速排序小结

- So QuickSort average is O(n logn)
- Can show that n2 is very unlikely
- For example, if very unlucky and always divide into 9/10*n and 1/10*n, will still be O(nlogn)
  T(n) = T(0.9n) + T (0.1n) + n



qsort() 函数

- The C library function void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*)) sorts an array
- Parameters
  - base - This is the pointer to the first element of the array to be sorted.
  - nitems - This is the number of elements in the array pointed by base.
  - size - This is the size in bytes of each element in the array.
  - compar - This is the function that compares two elements
- Return Value
  - This function does not return any value.

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

- Insertion sort very fast on small arrays
- Often: quicksort/mergesort only go down to, say, lists of > 20
  - Too much overhead to be fast
  - Won't take long absolutely, but done many times
- For that small, just calls insertion sort
  - 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

- Most often used:
  - Insertion
  - Shell
  - Merge
  - quick
- Insertion very fast for very small array $<$ 20, or if array is already mostly sorted
- Shell pretty fast and easy to code
- Merge fast in theory, used in external sort, and when need stable sort
- Sort email by name, then by date
- Quicksort fastest in general

Roadmap

基于比较的排序算法分析

- Suppose have 3 items a,b,c want to sort
- What to do?
- Natural idea: compare, compare
- At each point, do some test (compare)
  - Branch 2 ways, depending on result
  - Leaves = solutions
- Remember decision trees
  - Draw a decision tree
  - Each node = state
  - Each state = possibility
- What is the min depth of such a tree?

基于比较的排序算法分析

Lemma

a binary tree of depth d has $\leq 2^d$ leaves

Lemma

a binary tree with l leaves must have depth $\geq$ flooring (log l)

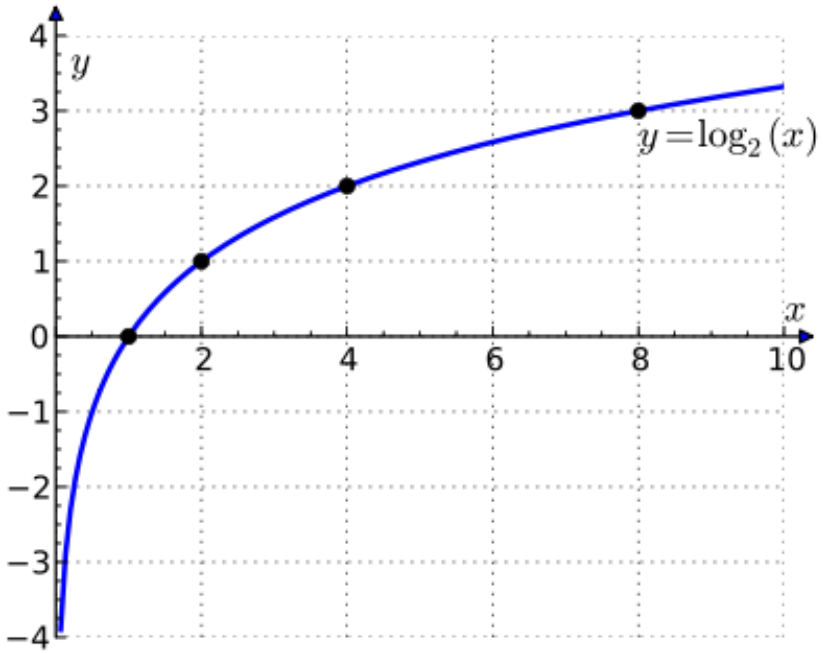
Lemma

any sort algorithm that uses only comparisons requires $\geq$ log(n!) comparisons in worst case

n! possible inputs $\rightarrow$ n! possible outcomes $\rightarrow$ n! leaves $\rightarrow$ depth $\geq$ log(n!)

Log函数

- Convexity: $\forall x_1, x_2 \in X, \forall t \in [0,1]$, $f(tx_1 + (1-x)t_2) \leq tf(x_1) + (1-t)f(x_2)$
- Logarithm function is monotonic convex



分析

Lemma

any comparison-based sort algorithm is $\Omega(n \log n)$

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

Theorem

any comparison-based sort algorithm is $\Theta(n \log n)$

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

结论

- The decision tree has the minimum depth $n \log n$
- Another way to understand: at start, there are $\Pi_1 = n!$ possible input sequences, each comparison at best sets $\Pi_{i+1} = \Pi_i/2$
- How many times must this be done until $\Pi_k = 1$? $\log \Pi_1 = \log n! = n \log n$ times
- Another way to understand: each comparison gives 1 bit of information
- There are n! possible inputs
- To find out index of entry, need $\log n!$ bits $= n \log n$

其它排序模型

- Can we ever do better than nlogn? If we have extra information, then we can do better
- Previous model assumes each comparison between 2 items takes constant time
- What about bucketsort? extra information = upper bound on values
- Bucket sort uses M-way comparison, similar to hashing
  - Walk through list and find max M
  - Apply bucketsort to M buckets
- Bucketsort can be very fast for certain kinds of inputs
  - Small range of integers
  - But must obtain $\Theta(M)$ memory

Roadmap

External Sort

- Problem: sort 1 GB of data with 1MB of RAM
- Where do we need this:
  - Data requested in sorted order (ORDER BY)
  - Needed for grouping operations
  - First step in sort-merge join algorithm
  - Duplicate removal
  - Bulk loading of B+-tree indexes.

External Sort Idea

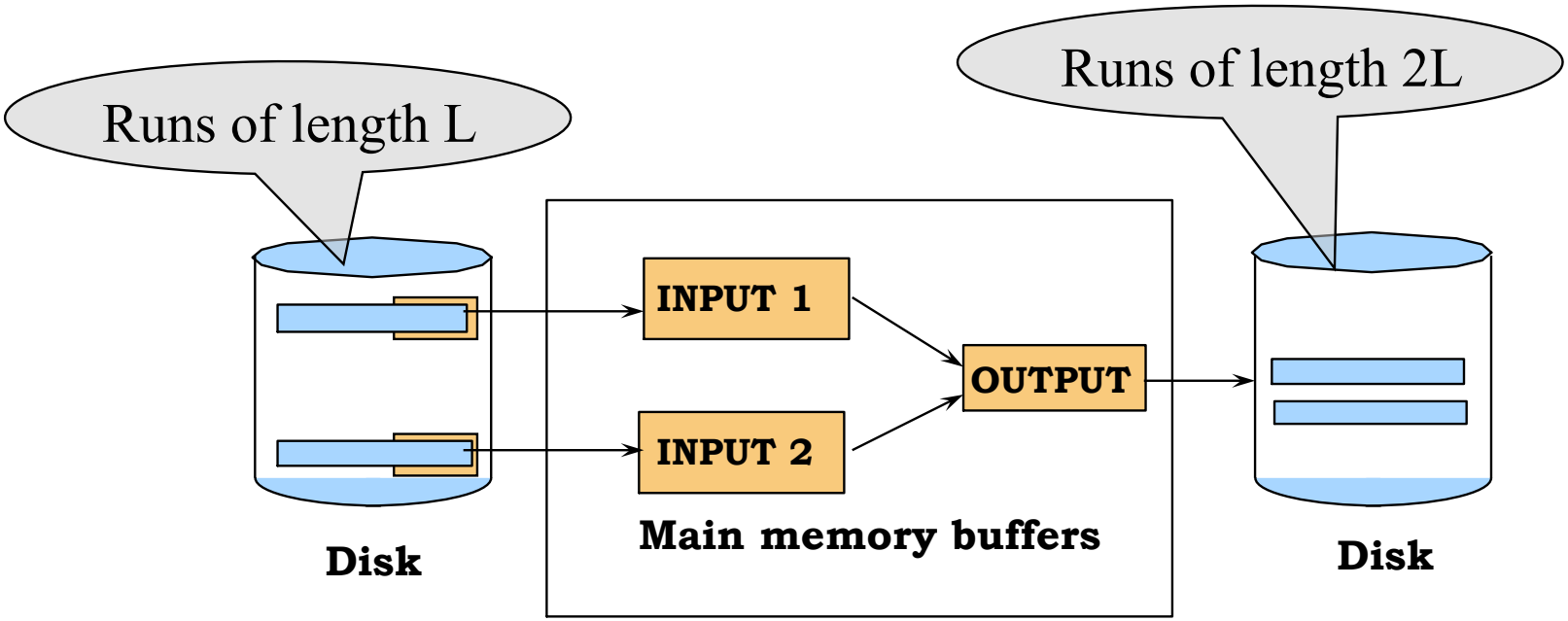
- Files can only be read block by block, no random access is available
- Idea: use small memory to buffer amount of file data, repeatedly sort-merge until a single file is formed
  - read some data from a file into buffer
  - sort the data in the buffer
  - write data back to a temporary file
  - read some of the sorted data from temporary files into the buffer
  - merge the read data
  - write the merged data back to a temporary file (run)
- File I/O is much slower than memory operations

Example

- Six numbers:8,1,6,3,4,5, small memory for only 3 numbers
- Sort numbers into runs with the help of the memory
  - Read 8,1,6 into memory, sort them and write 1,6,8 back to external storage
  - Read 3,4,5 into memory and sort, write 3,4,5 back
- Merge the sorted data
  - Read 1 and 3 into memory, compare and select the smallest one, write 1 back to external storage
  - Read 6 into memory and compare with 3, write 3 back
  - Read 4 into memory and compare with 6, write 4 back; repeat until all merged

Two Way Merge Sort

- Requires 3 buffers in RAM
- Pass 1: read a block, sort it, write it
- Pass 2, 3, …, etc.: merge two runs, write them, don't need to read in full sublists to merge



二路外部归并排序

Assume block size is B = 4Kb
Step 1, runs of length L = 4Kb
Step 2, runs of length L = 8Kb
Step 3, runs of length L = 16Kb = 2^{3-1} * 4Kb
...
Step 9, runs of length L = 1MB = 2^{9-1} * 4Kb
...
Step 19, runs of length L = 1GB = 2^{19-1} * 4Kb
Need 19 iterations over the disk data to sort 1GB

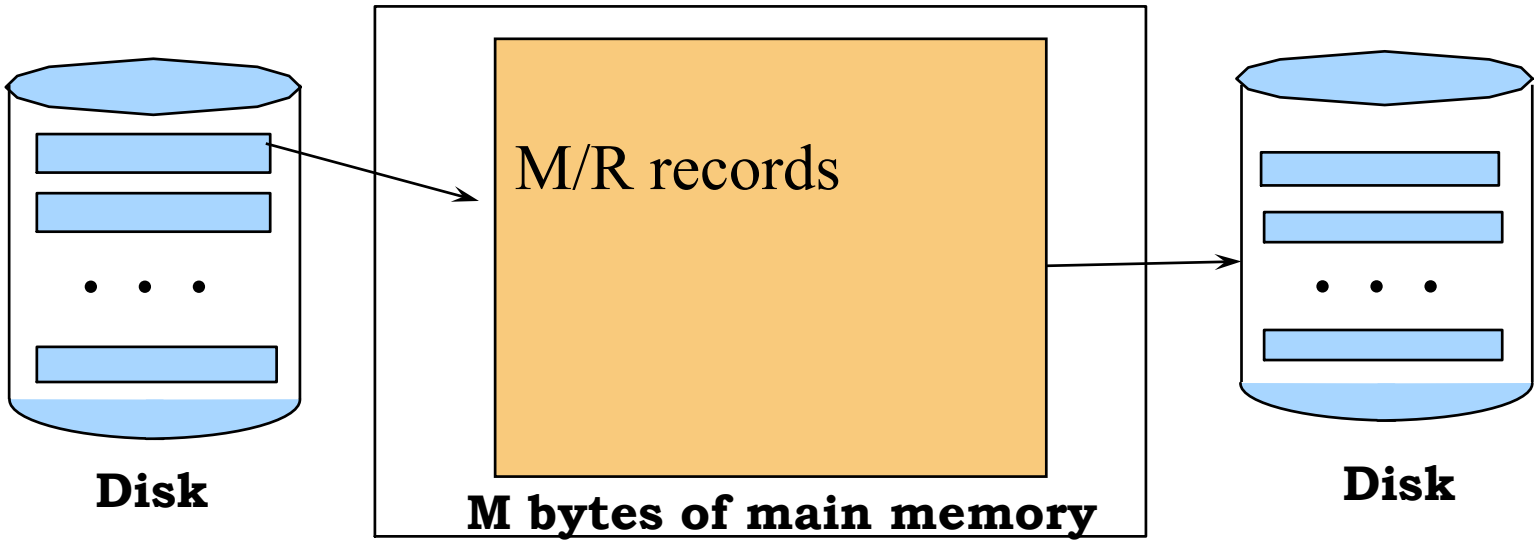
成本模型

Can we do better? Denote

- B: Block size (= 4KB)
- M: Size of main memory (= 1MB)
- N: Number of records in the file
- R: Size of one record
- L: current size of sorted sublists

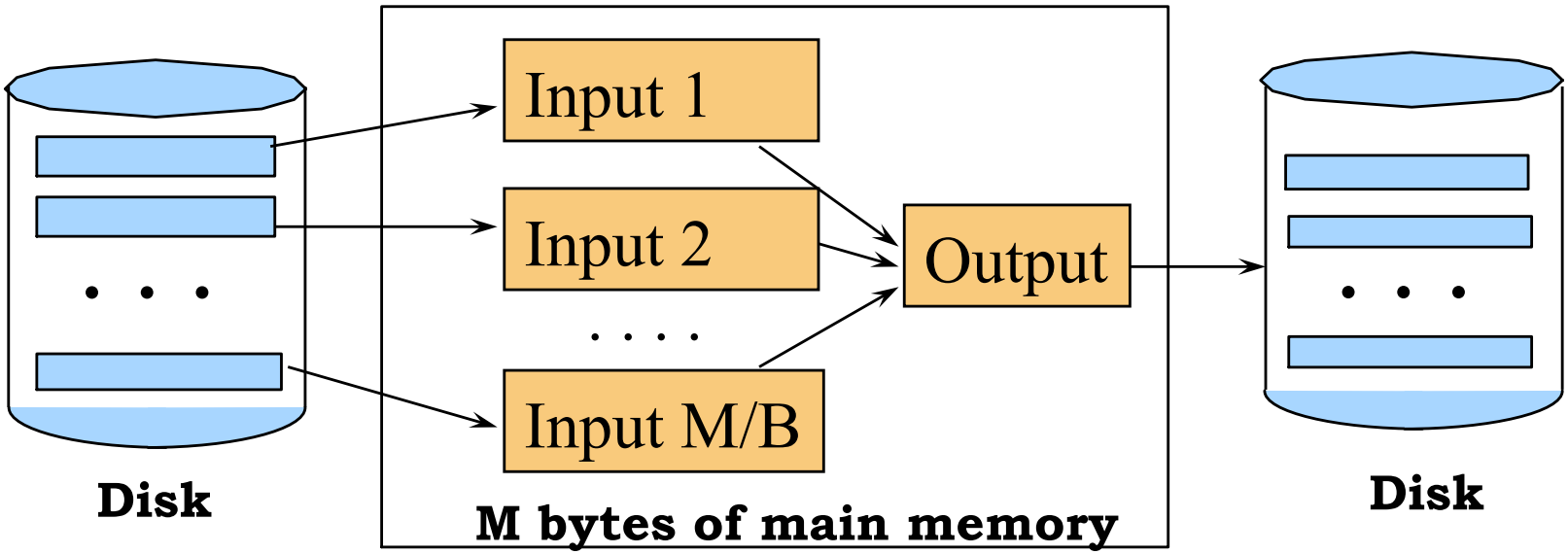
多路外部归并排序

Phase one: load M bytes in memory, sort
Result: N*R/M lists of length M bytes (1MB)
L = M



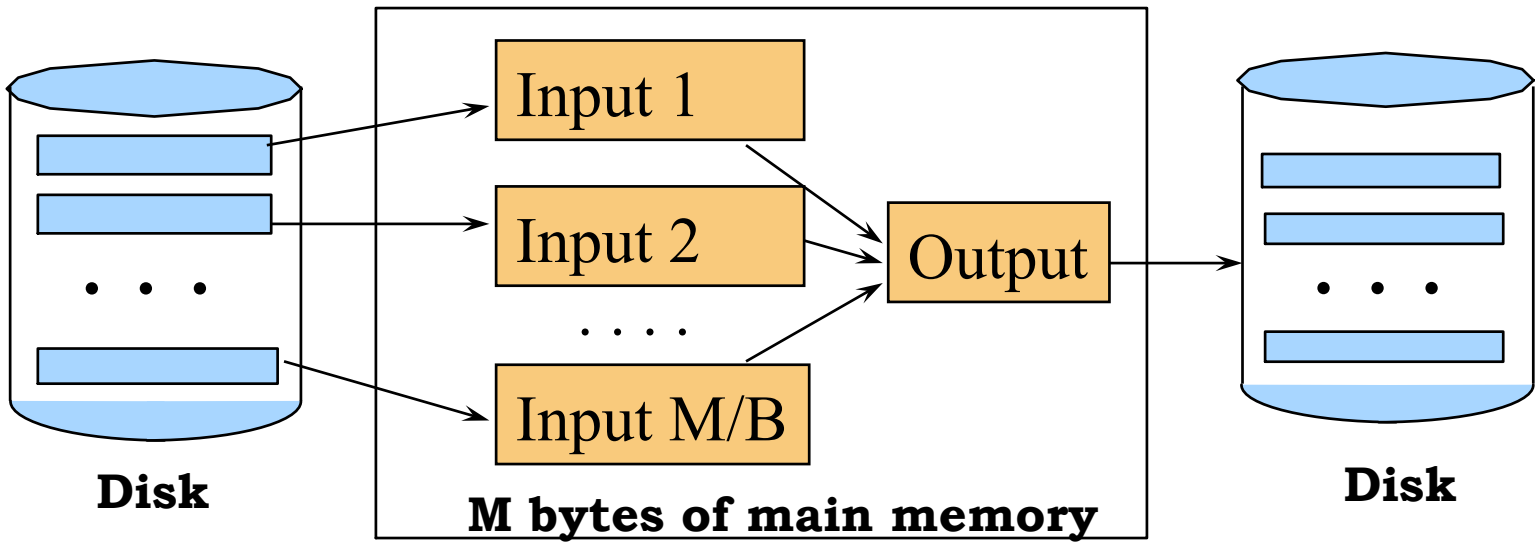
Phase 2

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



Phase 3

- Merge M/B - 1 lists into a new list
- Result: 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: $M * (M/B)^{P-1}$
- Number of passes to sort N size-R records: $\log_{M/B}(N * R/M) + 1$
- How much data can we sort with 1MB RAM? 1 pass: 1MB, 2 passes : 250MB (M/B = 250), 3 passes: 625GB
- Time:
  assume read/write block $\sim$ 10 ms = .01 s
  each pass: read, write all data
  each pass: 2*625GB/4kb*.01s = 2*1562500s = 2*26041m = 2*434 = 2*18 days = 36 days

Cost of External Merge Sort

- Number of passes: $\log_{M/B}(N * R/M) + 1$
- How much data can we sort with 10MB RAM (M/B = 2500)?
  - 1 pass: 10MB
  - 2 passes: 10MB * 2500 = 25,000MB = 25GB
  - 3 passes: 2500 * 25GB = 62,500GB

Cost of External Merge Sort

- Number of passes: $\log_{M/B}(N * R/M) + 1$
- How much data can we sort with 100MB RAM (M/B = 25000)?
  - 1 pass: 100MB
  - 2 passes: 100MB * 25,000 = 2,500,000MB = 2,500GB = 2.5TB
  - 3 passes: 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？