

数据结构与算法

DATA STRUCTURE

第六讲 数组排序

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

- 上节思考题：检测单链表是否有回路？有的话入口点在哪里？
- 数组排序
 - 在IntArray基础上，加入SelectionSort, MergeSort, QuickSort
- 二分法搜索
 - 继续完善IntArray，添加BinarySearch成员函数

自定义数组

- 之前我们封装了动态数组IntArray，支持动态分配内存，支持 **insert/delete/search**
- 现在我们要加入排序功能，即

```
class SortedIntArray : public IntArray
```

继承IntArray

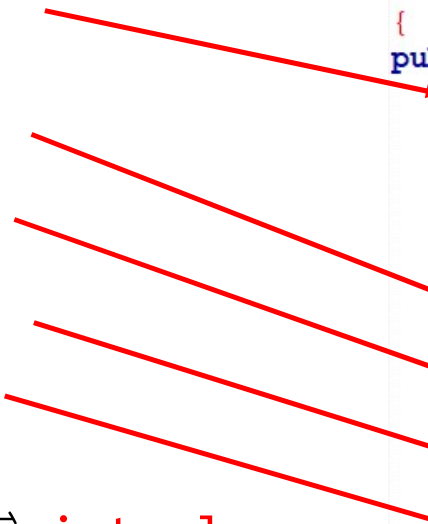
- 定义constructor
- 支持SelectionSort
- 支持MergeSort
- 支持QuickSort
- 支持BinarySearch

注意IntArray析构函数变成virtual

```
class SortedIntArray : public IntArray
{
public:
    SortedIntArray(int size)
        : IntArray(size)
    {
    }

    void SelctionSort();
    void MergeSort(int start, int end);
    void QuickSort(int start, int end);
    bool BinarySearch(int val);

private:
```



数组排序算法

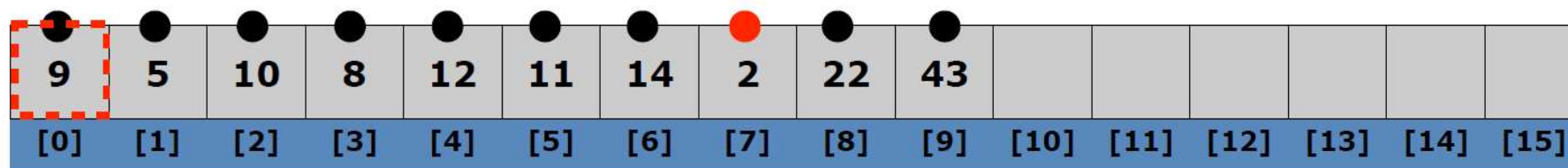
- **SelectionSort**
- **MergeSort**
- **QuickSort**

SelectionSort

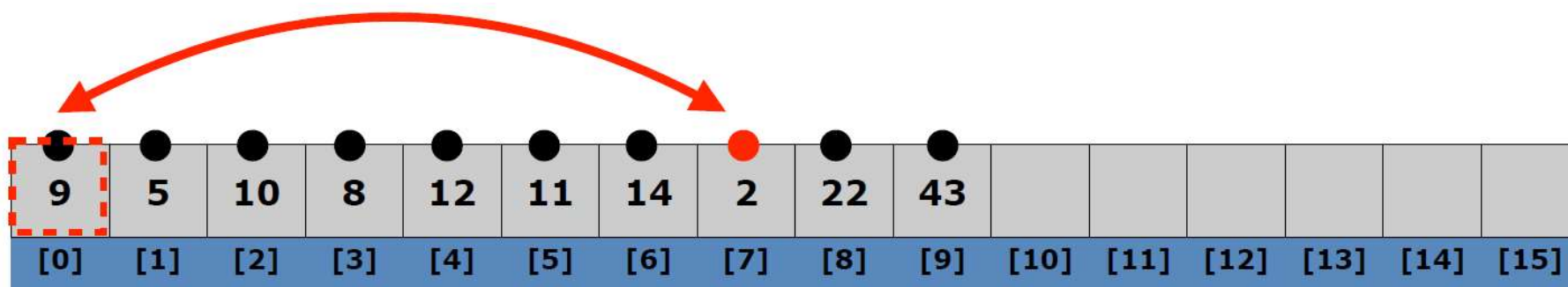
9	5	10	8	12	11	14	2	22	43						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- 算法：
 - 数组看成左右两区，左区**L**排好序，右区**R**没排好；
 - 初始是**L**为空，**R**为整个数组
 - 每次从**R**找一个最小元素，然后和**R**最左边的元素交换，然后并入**L**
 - 循环直到**R**区域为空

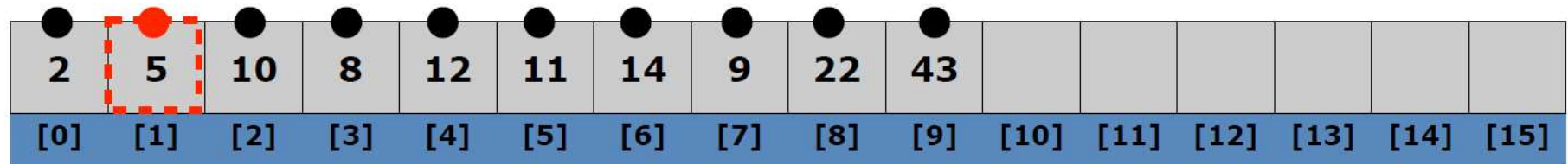
1 找到最小元素2，需要经过10次查找



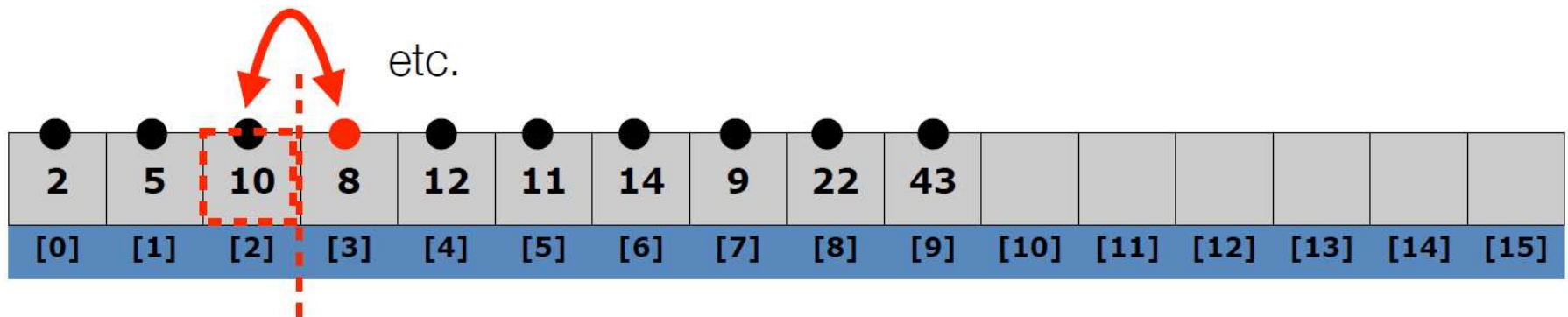
2 然后和最左边未排好序的元素**9**交换，
最小元素**2**就到了数组头部



3 然后从下一个未排好序的元素**5**继续，经过
9次查找，发现**5**是最小的，正好不需要交换



4 循环直到最后一个未排序元素



代码

```
void SortedIntArray::SelctionSort()
{
    for (int i = 0; i < this->Length() - 1; i++)
    {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < this->Length(); j++)
        {
            if ((*this)[j] < (*this)[min])
            {
                min = j;
            }
        }
        // swap smallest value to proper place, (*this)[i]
        if (i != min)
        {
            int temp = (*this)[i];
            (*this)[i] = (*this)[min];
            (*this)[min] = temp;
        }
    }
}
```



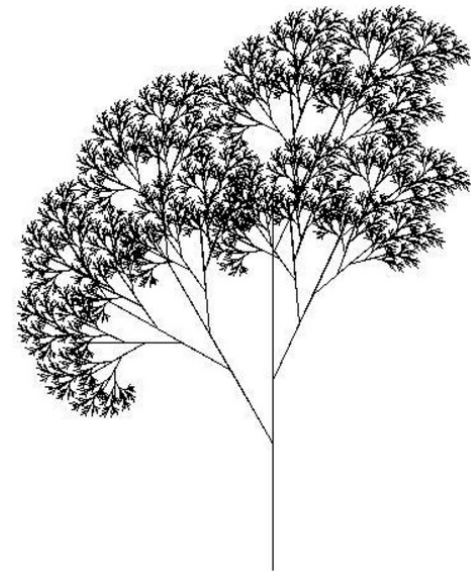
```
if (i != min)
{
    std::swap((*this)[i], (*this)[min]);
}
```

复杂度

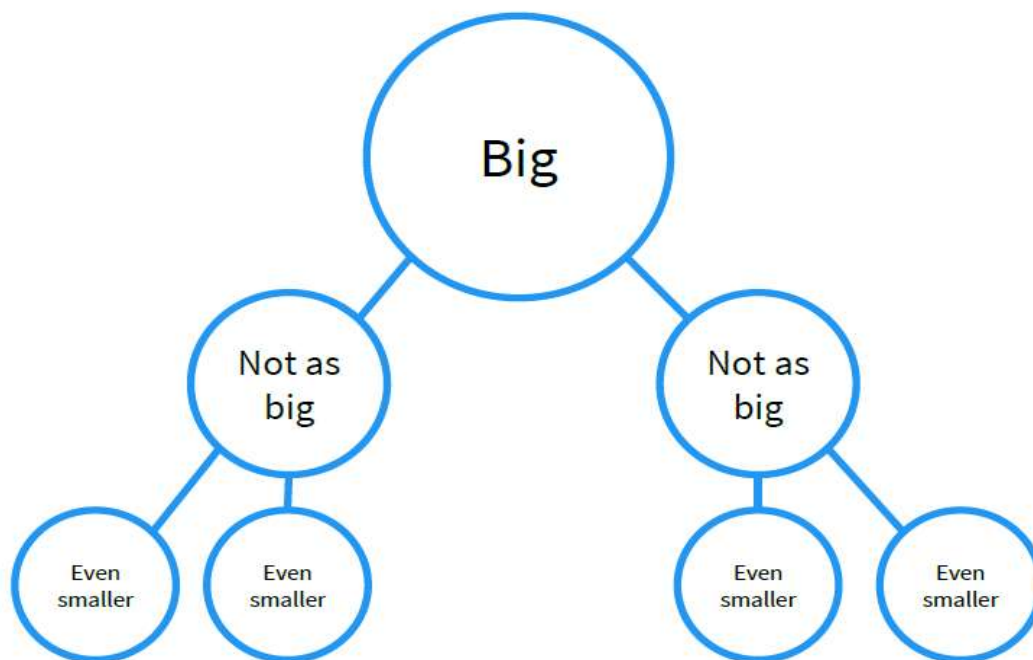
- 时间复杂度：
 - $O(n^2)$
- 空间复杂度：
 - $O(1)$ ，不需要额外空间，直接在原有数组内交换完成。

归并排序MergeSort

- 比SelectionSort好很多
- 用了分治法的思想(Divide & Conquer)
- 可用递归方法(recursion)
- 注意如果递归层次太深不能用



分治法 - 从下往上的方法



改进的方法：**MergeSort**采用了一种分治的策略。

归并排序 (MergeSort)

对长度为8的数组归并排序

- 分别对 $A[0,3]$ 和 $A[4,7]$ 排序
- 然后把两个排好序的半段归并

4	8	1	5	3	2	6	7
---	---	---	---	---	---	---	---

1	4	5	8	2	3	6	7
---	---	---	---	---	---	---	---

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

归并算法

```
algorithm mergesort(list A):  
  if length(A) ≤ 1:  
    return A  
  let left = first half of A  
  let right = second half of A  
  return merge(  
    mergesort(left),  
    mergesort(right)  
  )
```


归并算法 (cont)

```
algorithm merge(list A, list B):  
  let result = []  
  while both A and B are nonempty:  
    if head(A) < head(B):  
      append head(A) to result  
      pop head(A) from A  
    else:  
      append head(B) to result  
      pop head(B) from B  
  append remaining elements in A to result  
  append remaining elements in B to result  
  return result
```

Total work: $O(a+b)$, where a and b are the lengths of lists A and B .

Mergesort复杂度分析

- $T(n)$ 是用对长度为 n 的数组归并排序

- 分别对 $A[0,3]$ 和 $A[4,7]$ 排序

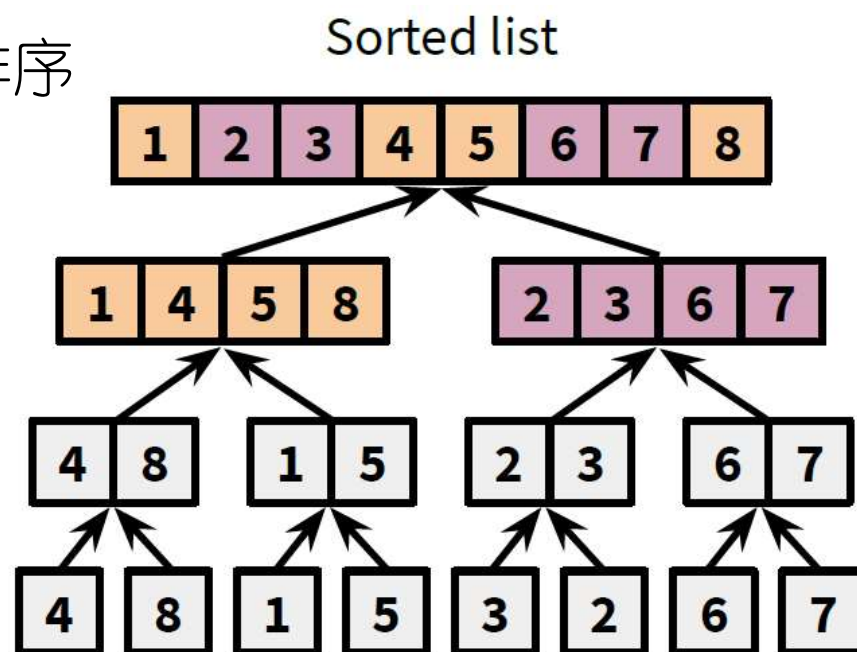
- 复杂度 $2T(n/2)$

- 然后把两个排好序的半段归并

- 复杂度 $\Theta(n)$

- $T(0) = \Theta(1)$

- $T(n) = 2T(n/2) + \Theta(n)$



复杂度

- 时间复杂度
 - $O(n \log n)$
- 空间复杂度
 - $O(n)$, 需要额外数组来做合并的事情, 每一次迭代都要额外数组
 - $O(1)$? 要in-place合并, 会比较复杂
 - Don Knuth在他的《编程艺术》里留作习题
 - “[In-place sorting with fewer moves](#)” Jyrki Katajainen, Tomi A. Pasanen
- 思考: mergesort链表 (best choice)

链表排序

- Mergesort比Quicksort更好
- Heapsort甚至不可行

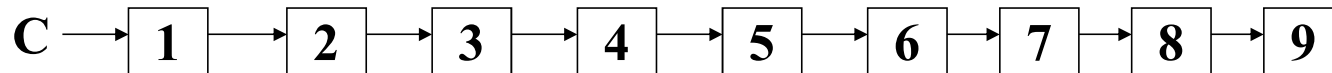
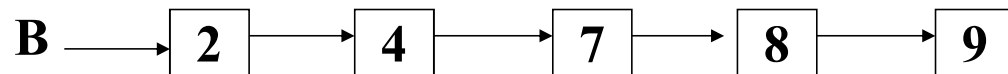
```
Node * MergeSort (Node * head)  
    if head == nullptr  
        return nullptr  
    SplitHalf(head, left, right)  
    left = MergeSort(left)  
    right = MergeSort(right)  
    return MergeList(left, right)
```

MergeList

合并二个有序链表



+



MergeList实现

- 思路和数组Merge一样
- 容易改编成in-place, 即不占用额外内存

```
ListNode* MergeList(ListNode *lNode, ListNode *rNode)
{
    // Assume each input list has at least one node with value INT_MAX
    assert(lNode && rNode);

    ListNode *head = nullptr;
    ListNode *tail = nullptr;

    while (lNode->next || rNode->next)
    {
        int key = 0;
        // Pick node with smaller value and it is safe to proceed to next
        // since this node has not reach last MAX node yet
        if (lNode->val < rNode->val)
        {
            key = lNode->val;
            lNode = lNode->next;
        }
        else
        {
            key = rNode->val;
            rNode = rNode->next;
        }

        tail = Insert(tail, key);
        if (head == nullptr)
        {
            head = tail;
        }
    }

    // 额外加一个虚拟节点, 使得
    // 后要删除的结点不是尾节点
    Insert(tail, INT_MAX);
    return head;
}
```

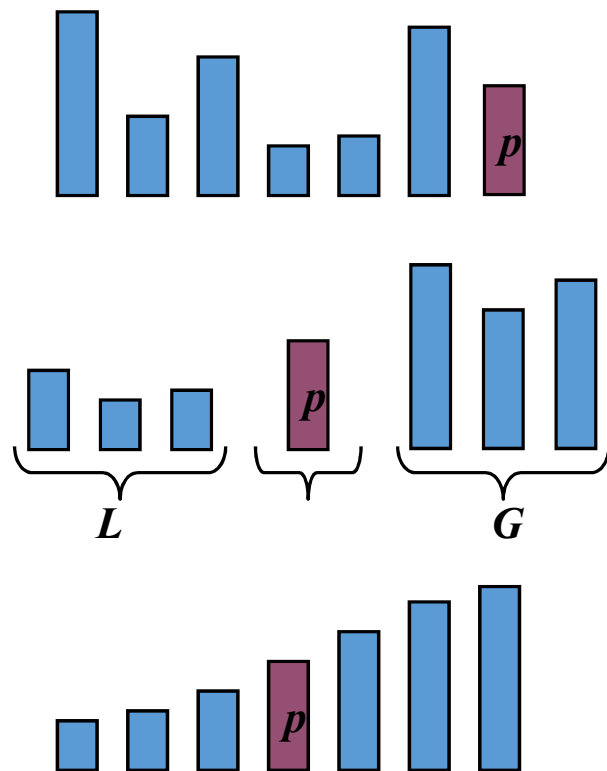
快速排序quicksort

快速排序 QuickSort

- **Tony Hoare**在1962年发明，被誉为“20世纪十大经典算法之一”
- 用的也是分治法的思想
- 把大的问题分成2个小的子问题，然后把解决子问题合并


Quicksort思想

1. 取一个元素作为基轴 (比如最后一个元素)
2. 按轴划分元素:
大的到右边; 小的到左边
3. 递归解决子问题



1 取基轴56

0	1	2	3	4	5	6	7
56	25	37	58	95	19	73	30

 pivot (56)

2 两个指针分别从第二个元素，最后一个元素开始

pivot (56)

0	1	2	3	4	5	6	7
56	25	37	58	95	19	73	30

lh rh

3 右边的指针停在第一个小于基轴的位置

pivot (56)

0	1	2	3	4	5	6	7
56	25	37	58	95	19	73	30

lh rh

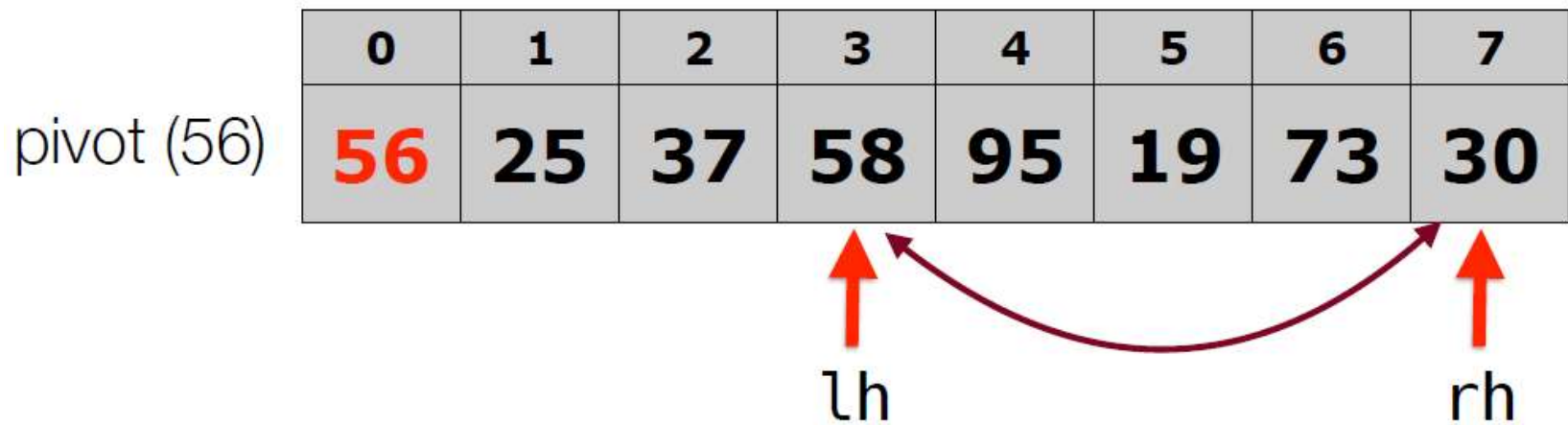
4 左边的指针停在第一个大于基轴的位置

pivot (56)

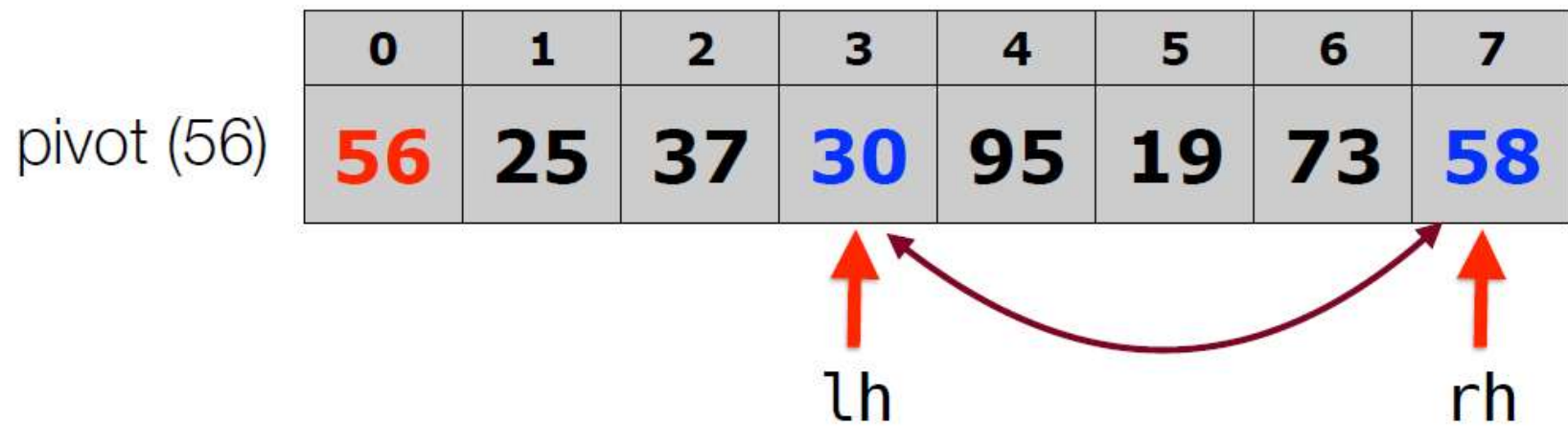
0	1	2	3	4	5	6	7
56	25	37	58	95	19	73	30

lh rh

5 交换左右指针指向的数据



6 交换后结果



7 检查左指针是否等于右指针

pivot (56)

0	1	2	3	4	5	6	7
56	25	37	30	95	19	73	58

lh rh

8 右指针继续往左边走，直到小于基轴的位置

pivot (56)

0	1	2	3	4	5	6	7
56	25	37	30	95	19	73	58

lh rh

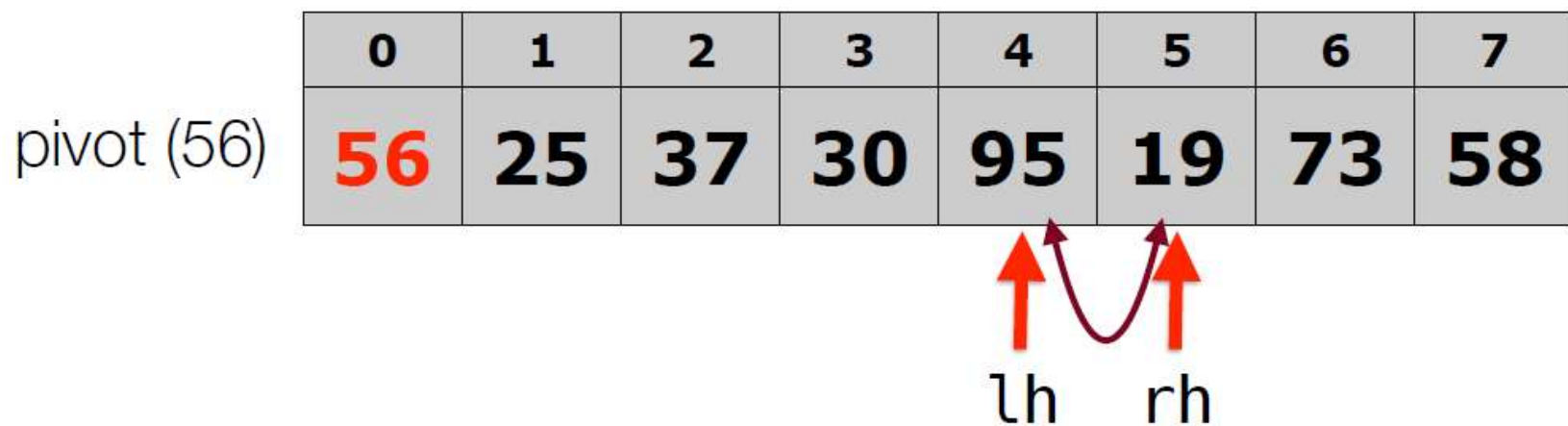
9 左指针继续往右边走，直到大于基轴的位置

pivot (56)

0	1	2	3	4	5	6	7
56	25	37	30	95	19	73	58

lh rh

10 交换左右指针指向的数据



11 交换后结果

pivot (56)


0	1	2	3	4	5	6	7
56	25	37	30	19	95	73	58

lh rh

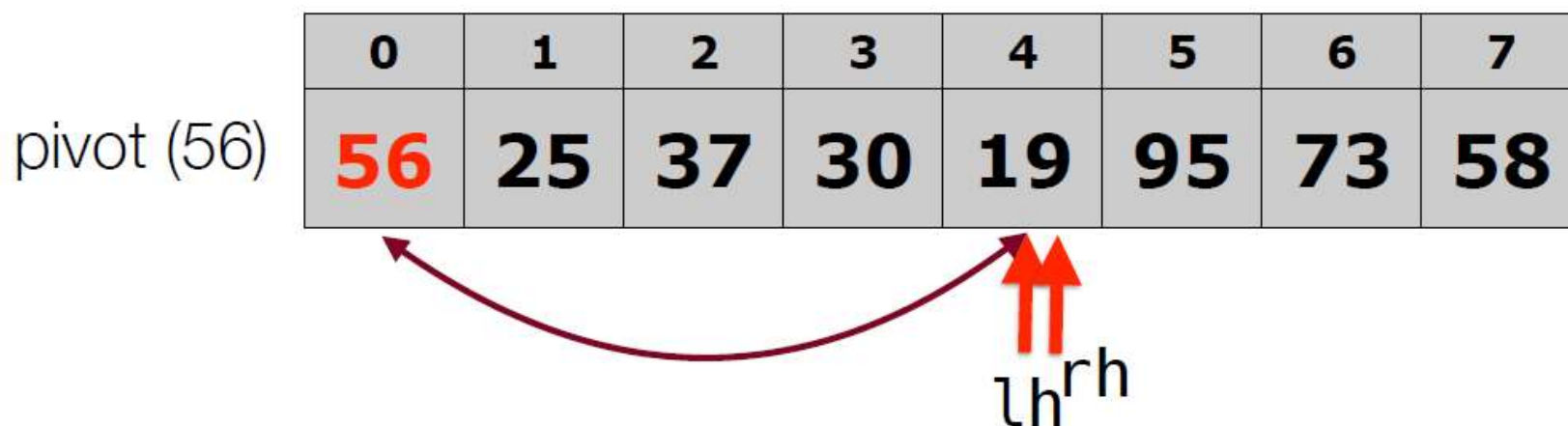
12 这时候，会发现左右指针碰头了

pivot (56)

0	1	2	3	4	5	6	7
56	25	37	30	19	95	73	58

 lh rh

13 最后把基轴和左指针的数据交换



14 完成了一次以基轴为中心的排序

QuickSort(0, 3);

QuickSort(4, 7);

pivot (56)

0	1	2	3	4	5	6	7
19	25	37	30	56	95	73	58


 l^h r^h

In-place递归算法

- 实现和前面例子示范有点差别（代码更简洁），但思路一样

```
void SortedIntArray::QuickSort(int start, int end)
{
    if (start >= end)
    {
        return;
    }
    int boundary = PartitionEnd(start, end);
    QuickSort(start, boundary - 1);
    QuickSort(boundary + 1, end);
}

int SortedIntArray::PartitionEnd(int start, int end)
{
    int pivot = (*this)[end];

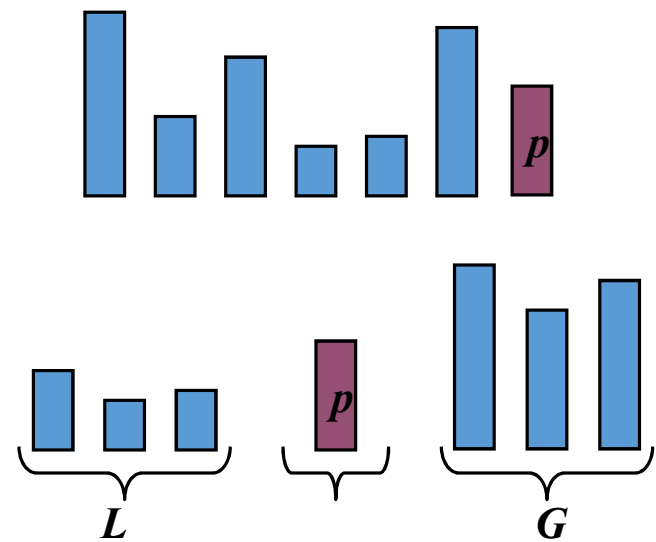
    int l = start - 1;
    for(int j = start; j < end; j++)
    {
        if ((*this)[j] <= pivot)
        {
            l++;
            swap((*this)[l], (*this)[j]);
        }
    }
    swap((*this)[l+1], (*this)[end]);

    return l+1;
}
```


QuickSort平均复杂度

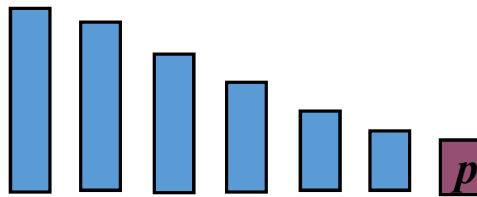
$$\begin{aligned} E(n) &= 2E(n/2) + n \\ &= 2^2 E(n/2^2) + 2(n/2) + n \\ &= \dots \\ &= 2^{\log n} (n / 2^{\log n}) + \dots + 2(n/2) + n \\ &= n \cdot (\log n + 1) \end{aligned}$$

- 注意以上是平均意义下的复杂度



QuickSort最差情况

- 如果每次取得最后一个元素都是最小元素
- 那么复杂度就变成了 $O(n^2)$
- 类似于SelectionSort



复杂度

- 平均时间复杂度
 - $O(n \log n)$
- 最差时间复杂度
 - $O(n^2)$
- 空间复杂度
 - $O(1)$ 不需要额外空间
 - 真的吗？（递归章节中还会进一步分析）

改进方法

1. 取一个元素作为基轴(**随机**元素)
2. 按轴划分元素:
大的到右边；小的到左边
3. 递归解决子问题

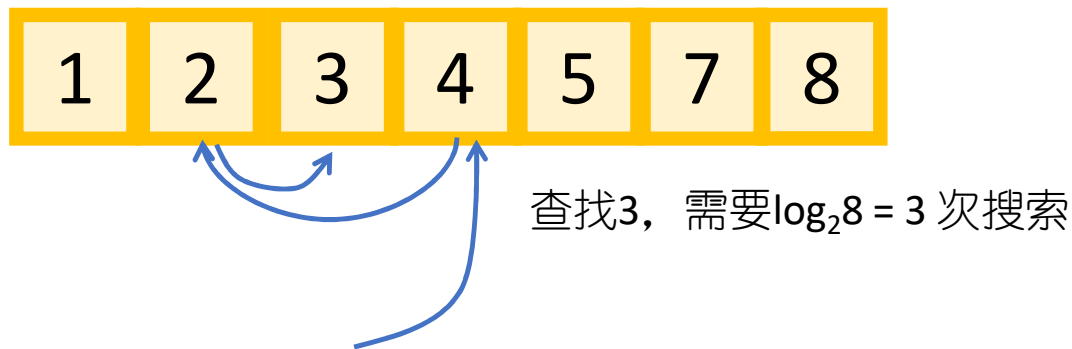


小结QuickSort

- 平均复杂度 $O(n \log n)$
- 最差情况下复杂度可以达到 $O(n^2)$. 注意随机算法也没法改变最差复杂度，但是可以让最差情况能出现的概率非常小
- 空间复杂度低
- 实际中效果最好

二分法查找SortedIntArray

- $O(\log(n))$ search, $O(1)$ select:



- 前提是数据元素排好序, 并且连续存放

二分法查找（递归实现）

- 注意函数声明
- 死循环
- 溢出（Java库存在十年的bug）
- 边界问题

```
int SortedIntArray::BinarySearch(int left, int right, int x)
{
    if (right < left)
    {
        return -1;
    }

    int mid = left + (right - left)/2;

    // If the element is at the middle, return itself
    if ((*this)[mid] == x)
    {
        return mid;
    }

    // If element is smaller than mid, then
    // it can only be in left sub-array
    if ((*this)[mid] > x)
    {
        return BinarySearchRecursive(left, mid-1, x);
    }

    // Else the element can only be present
    // in right sub-array
    return BinarySearchRecursive(mid+1, right, x);
}
```

二分法查找（迭代实现）

- 每次循环都可以根据中间元素确定要查找的数在左半部分还是右半部分
- 从而可以使查找次数指数次递减
- 面试手写代码

```
bool SortedIntArray::BinarySearch(int x)
{
    int low = 0;
    int high = this->Length() - 1;
    int count = 0;
    bool found = false;

    while (low <= high)
    {
        int mid = low + (high - low) / 2; // to avoid overflow
        if ((*this)[mid] == x)
        {
            found = true;
            break;
        }

        if ((*this)[mid] > x)
        {
            high = mid - 1;
        }
        else
        {
            low = mid + 1;
        }
        count++;
    }

    if (found)
    {
        cout << "\nFind number [" << x << "] in \"" << count << "\" compares." << endl;
    }

    return found;
}
```


二分法查找

- 如何在circular sorted array查找 **x** ?

[**7** **8** **9** **1** **2** **3** **4** **5** **6**]

- 进一步，如何查找环形有序数组的最小元素？

总结

Sorting Big-O Cheat Sheet			
Sort	Worst Case	Best Case	Average Case
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

作业

- 实现 `SortedIntArray`（继承 `IntArray`），包括成员函数 `SelectionSort`，`MergeSort` 和 `QuickSort`。其中 `QuickSort` 中取轴是随机的一个，避免最差情况。
- 测试 `SelectionSort`，`MergeSort`，`QuickSort`，至少要十组不同大小的数据，数组最大可以到上百万元素。
- 输出测试数据，把三种排序结果画一图对比显示，纵坐标是排序时间，横坐标为数组大小（matlab, python, R, etc.）

测时间（一）

```
#include <ctime>
void TestMergeSort(int length)
{
    SortedIntArray myarray(length);

    clock_t tstart = clock();
    myarray.MergeSort(0, myarray.Length() - 1);
    clock_t tend = clock();
    cout << "\nMergeSort on " << length << " elements. Elapse " << tend - tstart << " milliseconds" << endl;

    // Verify sort results.
    for (int i = 1; i < length; i++)
    {
        assert(myarray[i-1] <= myarray[i]);
    }
}
```

测时间（二）

```
void TestMergeSort(int length)
{
    SortedIntArray myarray(length);
    {
        AutoTimer timer;
        myarray.MergeSort(0, myarray.Length() - 1);
    }

    for (int i = 1; i < length; i++)
    {
        assert(myarray[i-1] <= myarray[i]);
    }
}
```

利用destructor自动输出耗费时间
重复使用封装的AutoTimer

```
class AutoTimer
{
public:
    AutoTimer() { m_startTime = clock(); }
    ~AutoTimer()
    {
        cout << "Elapsed time: " << GetElapsedTimeInSeconds() << " seconds" << endl;
    }

    double GetElapsedTimeInSeconds() const
    {
        clock_t endTime = clock();
        return (double)(endTime - m_startTime) / CLOCKS_PER_SEC;
    }

    long GetElapsedTimeInMilliseconds() const
    {
        clock_t endTime = clock();
        return endTime - m_startTime;
    }

    void Reset()
    {
        m_startTime = clock();
    }

private:
    clock_t m_startTime;
};
```

随机数

- 现实中有些随机数例子，比如扔硬币，掷骰子
- 计算机里我们用随机数来决定宝箱里的宝物，设计迷宫道路
- 问题是计算机里都是确定性的数字，没有现实中的复杂的物理变量



模拟随机数

- 数学计算来模拟随机数的产生，也叫做伪随机数
 - 取一个种子数`seed`
 - 根据种子数`seed`，用数学公式计算一个看上去和`seed`没多大关系的数字
 - 然后再根据上一次算出来的数字，用同样的数学公式计算下一个看上去“随机”的数字
 - 循环以上过程

简单的伪随机数生成器

只用简单的编程语言和数学计算

```
int main()
{
    static unsigned int seed = 23564;

    int count;
    cout << "Please enter count of random numbers: ";
    cin >> count;

    unsigned int random = seed;
    while (count-- > 0)
    {
        random = (8253729 * random + 2396403) % 32768;
        cout << random << " ";
    }
    return 0;
}
```


C++内置了随机数生成器

- `#include <cstdlib>`
- `srand()`, 开始前先调用一次来设置seed
- `rand()`, 产生下一个随机数, `[0, RAND_MAX]`

例子

使用srand()和rand()产生了一批随机数
测试了奇数个数和偶数个数

但是如果你运行这个程序几次后，会发现同样的结果

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    srand(23564);

    int count;
    cout << "Please enter count of random numbers: ";
    cin >> count;

    int oddNum = 0;
    int evenNum = 0;
    while (count-- > 0)
    {
        int randNum = rand();
        cout << rand() << " ";

        if (randNum % 2 == 0)
        {
            evenNum++;
        }
        else
        {
            oddNum++;
        }
    }

    cout << endl;
    cout << "Odd numbers have " << oddNum << endl;
    cout << "Even numbers have " << evenNum << endl;
    return 0;
}
```

改进

- 引入一个每次运行都不一样的数作为seed
- `time()` 返回自1970年1月1日以来的秒数
 - `#include <ctime>`
 - `srand(time(0));`
- 也可以返回`[min, max]`之间的随机数

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    srand((unsigned int)time(0));

    int min, max;
    cout << "Please enter range of random numbers: ";
    cin >> min >> max;

    if (min > max)
    {
        return 1;
    }

    double fraction = 1.0 / (RAND_MAX + 1.0);
    int randNum = rand() * fraction * (max - min + 1) + min;
    cout << "Generate random number: " << randNum << endl;
    return 0;
}
```

什么是好的随机数

1. 给定范围内的数字都有相同的概率生成
2. 下一个随机数不能之前一个容易推断出来，比如 $\text{num} = \text{num} + 1$
3. 随机数在整体上也是随机的，不能一段时间都是比较小的数，然后下一段时间都是比较大的数
4. 伪随机数都是有周期的，好的随机数就是对任何seed都有很长的周期

Q&A

Thanks!