Merge 将元素合并到一个 temp array，然后复制回去

Merge sort 有许多递归调用，只用创建一个 temp array，传递给递归方法。

```
public static <T extends Comparable<? super T>> void mergeSort(T[] a, int n)
{
    mergeSort(a, 0, n - 1);
}

public static <T extends Comparable<? super T>> void mergeSort(T[] a, int first, int last)
{
    T[] tempArray = (T[])new Comparable<?>[a.length]; //创建 temp array，长度与 a 相同.
    mergeSort(a, tempArray, first, last);
}

private static <T extends Comparable<? super T>>
    void mergeSort(T[] a, T[] tempArray, int first, int last)
{
    if (first < last)
    {
        int mid = (first + last)/2;
        mergeSort(a, tempArray, first, mid);
        mergeSort(a, tempArray, mid + 1, last);

        if (a[mid].compareTo(a[mid + 1]) > 0)
            merge(a, tempArray, first, mid, last);
    }
}
```

元素数

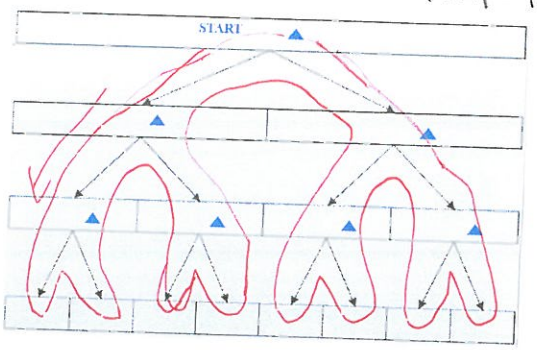
//将 each half 排序.

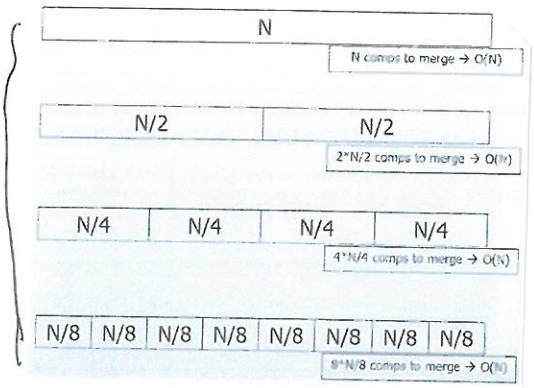
//？左大于右，不需排序合并

Lecture 19

Mergesort 时间复杂度 — $O(\log_2 N)$

Total = $\log N * O(N)$





lgN

$total = N + 2(N/2) + 4(N/4) + \cdots$

若 $N = 2^k$.

$total = (k+1)2^k = 2^0 2^k + 2^1 2^{k-1} + \cdots + 2^k 2^0$

↓ 问题数目　↘ 问题大小.

$k = \log_2 N \Rightarrow O(N \log_2 N)$

MergeSort 需要 temp array 以 copy、store 信息

这导致容易外 memory

另法：Quicksort

1) 问题分割

给一个特殊值 pivot 来分割

分成三组：小于 pivot；pivot；大于 pivot

0	1	2	3	4	5	6	7
50	80	60	20	30	10	70	40

↘ 选定 [7] 为 pivot

↓ dividing partition

0	1	2
10	30	20
<= pivot		

3
40
pivot

4	5	6	7
80	50	70	60
>= pivot			

数组此时未排序，但走心的 pivot 位置确定.

```
QuickSort(A)
  if (size of A > 1)
    Choose a pivot value
    Partition A into left and right sides
    based on the pivot
        Recursively sort left side
        Recursively sort right side
```

QuickSort的伪代码流程

】递归用于sort. → 该两步排序.

2) 用子答案解决原问题
无需

重点在于partition分区.

{ left data < pivot
  right data > pivot } 不用做 ; 如果get stuck, 说明data放错了位置

例: 初始状态.

INITIALLY:

index { left = 0
right = 7 }
pivot = 40
pivotIndex = 7

b = 0
(indexFromLeft)

c = 6
(indexFromRight)

①

0	1	2	3	4	5	6	7
50	80	60	20	30	10	70	40
b					c	c	

A[b] > pivot, 在左边; A[c] < pivot, 在右边
swap

②

0	1	2	3	4	5	6	7
10	80	60	20	30	50	70	40
	b			c			

依然 b > , c < , 需要再次swap

③

0	1	2	3	4	5	6	7
10	30	60	20	80	50	70	40
		b	c				

b >, c <, swap.
b与c越发接近

④

0	1	2	3	4	5	6	7
10	30	20	60	80	50	70	40
		c	b				

此时还是 b >, c <, 但是 b >= c 了!
下面开始移动pivot
把pivot的index换到indexb.
即 pivotIndex = indexFromLeft,

0	1	2	3	4	5	6	7
10	30	20	40	80	50	70	60

分完区

0	1	2	3	4	5	6	7
10	30	20	40	80	50	70	60

此时pivot位于绝对正确的位置, 不会被移动
而两侧会有较大变动

```
private static <T extends Comparable<? super T>> int partition(T[] a, int first, int last)
{
    int pivotIndex = last;            // pivot定在数组尾
    T pivot = a[pivotIndex];          // 把pivot变量index定给数组.
    int indexFromLeft = first;
    int indexFromRight = last - 1;    // 分区步骤.

    boolean done = false;
    while (!done)
    {
        while (a[indexFromLeft].compareTo(pivot) < 0)   // 左index < pivot, 左进一格
            indexFromLeft++;

        while (a[indexFromRight].compareTo(pivot) > 0 && indexFromRight > first)   // 右index > pivot且右 > 初, 右退一格
            indexFromRight--;

        if (indexFromLeft < indexFromRight)   // precon
        {
            swap(a, indexFromLeft, indexFromRight);   // 数组内左右交换.
            indexFromLeft++;
            indexFromRight--;
        }
        else
            done = true;
    }
    swap(a, pivotIndex, indexFromLeft);   // pivot 与左换(对应④).
    pivotIndex = indexFromLeft;
    return pivotIndex;
} //
```

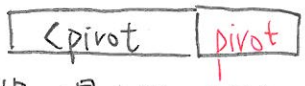
```
public static <T extends Comparable<? super T>>
                        void quickSort(T[] array, int n)
{
        quickSort(array, 0, n-1);//index从头到尾
}

public static <T extends Comparable<? super T>>
                        void quickSort(T[] array, int first, int last)
{
        if (first < last)
        {
                int pivotIndex = partition(array, first, last);

                quickSort(array, first, pivotIndex-1);
                quickSort(array, pivotIndex+1, last);   //使用方法
        }
}
```

基时间复杂度(QuickSort)——取决于分区质量

1) pivot处于分区最中间

此时执行轨迹与MergeSort相似(但比它快),时间为 $O(N \log_2 N)$

2) pivot处于分区边缘



(并非终点,这是分区完成后pivot的位置)

- 此时pivot最大,位于边缘(最小时也适用)
- 且每次递归调用都如此发生

分析:-首次分区调用,有 $N-1$ 个元素与pivot比较 $\Rightarrow$ 产生一个大小为 $N-1$ 的子问题

-二次 ～ $N-2$ ～ $\Rightarrow$ ～ $N-2$ ～

⋮

总大小为 $N-1+N-2+\cdots+1 = \frac{(N-1)N}{2} \Rightarrow O(N^2)$,

why? 此时递归节约效率极低(分而治之几乎无用),

Lecture 20

QuickSort的时间复杂度取决于pivot的选择

以下为已排序数组(以A[last]为pivot)

Call	0	1	2	3	4	5	6	7
1	10	20	30	40	50	60	70	80
2	10	20	30	40	50	60	70	80
3	10	20	30	40	50	60	70	80
4	10	20	30	40	50	60	70	80

此时时间复杂度为 $O(N \lg N)$

红色为当前pivot,绿色为之前pivot,橙色为无序数据

以更智能的方式选择pivot：Median of Three — pivot不为任何index

每次分区时考虑3种可能：A[first]；A[mid]；A[last]

⇓ 对该3元素排序，大小对应

使A[first] < A[mid] < A[last]

⇓ 作为pivot

first			mid			last
50			20			40

first			mid			last
20			40			50

Red = pivot, green = previous pivots, orange = partitioned data, blue = base cases

0	1	2	3	4	5	6
10	20	30	40	50	60	70
10	20	30	40	50	60	70
10	20	30	40	50	60	70

```
private static <T extends Comparable<? super T>>
               int partition(T[] a, int first, int last)
{
    int mid = (first + last)/2;
    sortFirstMiddleLast(a, first, mid, last); //获取mid并排序
    swap(a, mid, last - 1); //把pivot换到倒数第二个位置

    int pivotIndex = last - 1; //获取pivot index
    T pivot = a[pivotIndex]; //完全正式定义pivot
    int indexFromLeft = first + 1;
    int indexFromRight = last - 2; //开始

    boolean done = false;
    while (!done)
    {
       while (a[indexFromLeft].compareTo(pivot) < 0)
          indexFromLeft++;
       while (a[indexFromRight].compareTo(pivot) > 0)
          indexFromRight--;
    //其余代码相同
    } //结束分区
```

但是，即便使用Median of Three，最坏情况仍可能发生。

综上，QuickSort时间复杂度为

- Except case — $O(N\log_2 N)$
- Worst case — $O(N^2)$

?如果随机选择pivot ⇒ 并非坏事

并不总是bad case，很完美____。

random pivot出现坏情况的可能数为 $\frac{2^N}{N!}$

?多个pivot　Dual Pivot Quicksort

P_1 & P_2 分出3个区：(<P_1) (>=P_1&<=P_2) (>P_2) ⇒ 产生3个必须递归排序的子数组

?停止递归　logical size为1?

随着问题变小，递归的耗费大于D&C的节省

因此在合适时切换算法为InsertionSort

```
public static <T extends Comparable<? super T>>
             void quickSort(T[] a, int first, int last)
{
     if (last - first + 1 < MIN_SIZE)
     {
        insertionSort(a, first, last);   //剩余元素<MIN,换插入
     }
     else
     {
        int pivotIndex = partition(a, first, last);

        quickSort(a, first, pivotIndex - 1);
        quickSort(a, pivotIndex + 1, last);
     } //end if
}
```

/或在base case > 1时停止但不排序

所有递归完成后，InsertionSort整个数组

⇙ 此时其使用率不高

MergeSort与QuickSort的选择

MergeSort有更稳定的运行时间；而QuickSort性能更优

⇓ 需要额外数组与数据复制

QuickSort并非稳定排序

两个相等元素 X_1 与 X_2，原始数据中 X_1 比 X_2 出现得早
MergeSort后 X_1 依旧比 X_2 早，但 QuickSort不保证

对于复杂类型，最好使用 MergeSort { QuickSort for 基本数据类型
MergeSort for 对象

```
for (; (beginHalf1 <= endHalf1) && (beginHalf2 <= endHalf2); index++)
{
    if (a[beginHalf1].compareTo(a[beginHalf2]) <= 0)
    {
        tempArray[index] = a[beginHalf1];
        beginHalf1++;
    }
    else
    {
        tempArray[index] = a[beginHalf2];
        beginHalf2++;
    }
}
```

稳定的原因：元素之间的比较
运算符若次为 <，相同元素将从右侧出取
这会使相等项易位

QuickSort不稳定的原因

```
while (a[indexFromLeft].compareTo(pivot) < 0)
      indexFromLeft++;
while (a[indexFromRight].compareTo(pivot) > 0)
      indexFromRight--;
if (indexFromLeft < indexFromRight)
{
      swap(a, indexFromLeft, indexFromRight);
      indexFromLeft++;
      indexFromRight--;
}
else
      done = true;
```

问题

QuickSort例

0	1	2	3	4	5	6	7
60_1	40_1	20	10	70	60_2	40_2	50
b						c	

分区执行时，60_1 与 40_2 在 wrong side
从而交换易位也

0	1	2	3	4	5	6	7
40_2	40_1	20	10	70	60_2	60_1	50

如何排序链表？
1) MergeSort可以但开销很大 { 一分半 $\Rightarrow O(N)$
一合并 $\Rightarrow O(N)$
2) QuickSort也行但得双链表

Lecture 2

基于比较的排序：将元素与其它比较并在必要时移动，以排序数据
下限为 $N \lg N$

考虑 array of string (string is an array of characters)
思考每个字符串从最右到最左的不同，及位置的字符值。
并非字符比较，使用单个字符作为 'bin' 的索引而比较。

例、

0	1	2	3	4
H	E	L	L	O
C	E	L	L	O
C	E	L	L	S
H	E	L	P	S
S	H	E	L	F
B	E	L	L	S
C	H	E	S	S

首先考虑 position 4
=> bin "O" "S" "F"

'A'
...
'F' SHELF
...
'O' HELLO, CELLO
...
'S' CELLS, HELPS, BELLS, CHESS
...

0	1	2	3	4
S	H	E	L	F
H	E	L	L	O
C	E	L	L	O
C	E	L	L	S
H	E	L	P	S
B	E	L	L	S
C	H	E	S	S

按顺序将数据从bin复制回数组

(此时已根据position4重排序)

接着考虑position3

bin "L" "P" "S"

'A'	
...	
'L'	SHELF, HELLO, CELLO, CELLS, BELLS
...	
'P'	HELPS
...	
'S'	CHESS
...	

0	1	2	3	4
S	H	E	L	F
C	H	E	S	S
H	E	L	L	O
C	E	L	L	O
C	E	L	L	S
B	E	L	L	S
H	E	L	P	S

再次复制数据(根据position3)

考虑position 2

bin "E" "L"

0	1	2	3	4
S	H	E	L	F
H	E	L	L	O
C	E	L	L	O
C	E	L	L	S
B	E	L	L	S
H	E	L	P	S
C	H	E	S	S

'A'	
...	
'E'	SHELF, CHESS
...	
'L'	HELLO, CELLO, CELLS, BELLS, HELPS
...	

根据position2把数据复制回数组

考虑position1 bin "H" "E".

0	1	2	3	4
H	E	L	L	O
C	E	L	L	O
C	E	L	L	S
B	E	L	L	S
H	E	L	P	S
S	H	E	L	F
C	H	E	S	S

'A'	
...	
'E'	HELLO, CELLO, CELLS, BELLS, HELPS
...	
'H'	SHELF, CHESS
...	

数据复制回数组.

考虑position 0

bin "B" "C" "H" "S"

'A'	
'B'	BELLS
'C'	CELLO, CELLS, CHESS
...	
'H'	HELLO, HELPS
...	
'S'	SHELF
...	

最终结果

0	1	2	3	4
B	E	L	L	S
C	E	L	L	O
C	E	L	L	S
C	H	E	S	S
H	E	L	L	O
H	E	L	P	S
S	H	E	L	F

每次把数据放进bin都根据字母排序(由最低有效字符到最高有效字符)

相同字符由之后判定

Note: 与一般string比较不同, Radix Sort的比较由右到左

? 如果字符串长度不同

从right index of string (position length()-1)开始

例. A[0] = HELP
A[1] = HELPS
A[2] = HELPED

填充

A[0] = HELP@@
A[1] = HELPS@
A[2] = HELPED

```java
public class RadixDemo
{
    public static int numValues = 27;
    public static void RadixSort(String [] data)
    {
        ArrayList<Queue<String>> bins;
        if (data.length > 0)
        {
            int max = data[0].length();          // 找到最长的字符串长度
            data[0] = data[0].toUpperCase();
            for (int i = 1; i < data.length; i++)
            {
                if (data[i].length() > max) max = data[i].length();
                data[i] = data[i].toUpperCase();   // 大写
            }

            bins = new ArrayList<Queue<String>>();   // 创建队列
            for (int i = 0; i < numValues; i++)
                bins.add(new LinkedList<String>());

        char curr;
        for (int k = max-1; k >= 0; k--)   // 从后面起经历所有位置
        {
            for (int i = 0; i < data.length; i++)   // 处理所有字符串
            {
                if (data[i].length()-1 < k)
                    curr = '@';                  // 需要填充
                else
                    curr = data[i].charAt(k);

                int loc = curr - '@';
                bins.get(loc).offer(data[i]);   // 把字母指向index
                                                // 把字符串放进队列
            }
            int count = 0;
            for (int j = 0; j < bins.size(); j++)   // 把数据放回数组
            {
                while (!bins.get(j).isEmpty())
                {
                    data[count] = bins.get(j).poll();
                    count++;
                }
            }
        }
    }
}
```

List Interface (& List)

是一组方法的抽象表示类的行为

实现List的类如何自身实现并未规定

(数据能以多种方式存储)

Q?: List类的用户如何"顺序访问"所有数据?

即我们希望一次一个获取列表中的每个数据,直至所有

一种方法:用ListInterface中的toArray()将数据复制到一个数组并返回该数组

能否不造新数组?

getEntry(i)

访问

使用一个循环与getEntry()

列表里的每一个list都会调用

一对于LinkedList而言操作量非常大.

why?

getEntry()会在每次开头重启

因无法记得上次停留,所以需要重访

用iterator实现remember

运行时间

遍历所有字符串的每个位置,并将每个字符串放进bucket

然后从bucket中删除,放回数组

=> 最大字符串长度为K,数组长度为N => O(kN) => O(N)

对于中小型数组,它的开销很大

Radix并非普遍算法

iterator迭代器

一一个程序组件,使按序遍历列表.

一保持迭代的状态,一次前进一位

(不需每次从头开始,可以remember)

Lecture 22

Iterator (功能&实现)

```
public interface Iterator<T>
{
  public boolean hasNext();

  public T next();

  public void remove();

}
```

// 若集合中被删除的项目尚未在迭代中访问，返回true
// 返回集合中的下一个项目；如无项目，抛出异常
// 删除集合中返回的最后一个项目，每个项目只能被调用一次，且可选

例：求列表数据集合的mode → (集合中最常出现的项目)

一个一个值找它出现的次数

30	60	20	30	20	80	60	20	30	60	80	20	60	70	20

30三次，60四次，20五次…… 有两个单独的迭代 { 一个遍历列表标记每个项目
另一个计算项目出现频率

也对于List (linked list不行)，用嵌套的for循环和getEntry()来实现

也因此使用两个iterator { 外迭代器考虑列表中的下一项
内迭代器计算该项目的出现次数 } 相互独立，次数不同，位置不同

```
public static void getModeIterator1(List<Integer> L)
{
        Iterator<Integer> outer = L.iterator();//外迭代
        Iterator<Integer> inner;                //内
        Integer theMode = null, currOuter = null, currInner = null;//初始化
        int modeCount = 0, currCount = 0;
        while (outer.hasNext())
        {
                currOuter = outer.next();       //外循环不查找下项
                currCount = 0;
                inner = L.iterator();
                while (inner.hasNext())
                {
                        currInner = inner.next();
                        if (currInner.equals(currOuter))
                                currCount++;
                }
                if (currCount > modeCount)
                {
                        theMode = currOuter;
                        modeCount = currCount;
                }
        }
        System.out.println("The mode is " + theMode + " with " +
                        modeCount + " occurrences ");//作输出
}
```

//内循环 ↓ 计算频率.

? List本身就是接口，如何实现该接口，让其成为List的一部分？
能否使迭代器接口成list interface的一部分？
一只需加以3个方法，允许接口方法访问基础列表.

如果迭代器是列表的一部分，同一数据无法创建&使用不同迭代

30	60	20	30	20	80	60	20	30	60	80	20	60	70	20

一current一改，全局更改，无法有第2个迭代器

! 将迭代器与List分开，但迭代器仍能访问List内的数据

Solution：在list的外部顶端实现迭代器

使每个迭代器成为新对象，但给它访问实现基础列表的权限（使迭代器不成为列表的一部分）.

[inner curr]　[outer curr]：outer与inner访问相同的list of data →有分离的state virable 指定其在迭代的位置

30	60	20	30	20	80	60	20	30	60	80	20	60	70	20

→这是List的instance variable

⇓？修其应用于ListInterface

有一个修改接口：ListWithIteratorInterface<T> →作者修改

⇓它具有所有ListInterface方法，并能在list上生成新迭代器

public Iterator<T> getIterator() 对应 solution

⇓

该方法返回一个在当前list顶部构建的新迭代器 ⇒有自己的状态，因此可以在一个list上多个迭代器

正常List<T>已经内置该iterator方法

list类中有getIterator()方法 ⇒允许类生成iterator对象 →独立的

—能在不违反data abstraction的情况下按序访问元素

→实现

```
public class LinkedListWithIterator<T>
             implements ListWithIteratorInterface<T>
{

   private Node firstNode;
   private int  numberOfEntries;   // List66 数据、方法与链表相同

   public Iterator<T> getIterator()
   {
        return iterator();
   } //结束 getIterator

   public Iterator<T> iterator()
   {
        return new IteratorForLinkedList();
   } //结束Iterator
   ...
```

getIterator()返回一个新的IteratorForLinkedList对象

↓对象内为具体实现

?该对象如何创建（这是不同的类）

—将IteratorForLinkedList设为内部私有类

⇒直接访问链表的实例变量.

（从而，这个类构建在当前列表中，以高效方式遍历列表中所有数据）

链表中的节点

```
      private class IteratorForLinkedList implements Iterator<T>
      {
          private Node nextNode;
          private IteratorForLinkedList()
          {
              nextNode = firstNode;   //初始节点的设置
          }

          public boolean hasNext()
          {
             return nextNode != null;
          }

          public void remove()
          {
             throw new UnsupportedOperationException("Not supported"); //结束remove
          }
```

—(private) inner class可以访问outer class的变量

这个类完全在LinkedListWithIterator类中

```
    public T next()
    {
        if (hasNext())
        {
            Node returnNode = nextNode;   //保存当前节点
            nextNode = nextNode.getNextNode();   //高级迭代器
            return returnNode.getData();   //返回下一个条目
        }
        else
            throw new NoSuchElementException("Bad call to next()");
    }
} // End 类 IteratorForLinkedList
```

逻辑：尚有元素可以访问时，存储现在节点，获得下一节点，返回其数据

注：remove()未实现

/迭代器的实现方式取决于List由Array还是linked list实现

⇓

1) 在Linked List中，Node reference是迭代器的唯一实例变量。

节点引用变量—node reference在创建迭代器时初始化为firstNode

每次调用next()时，它都在列表中向下推进。

2) Array List

仅需一个整数存储迭代中当前值的index

⇓ 每次调用next()时都增加

remove()可实现，通过shift来填补gap

Note：对array list使用iterator不会增加访问时间

迭代允许一致访问—我们不必知道LL还是AL

Iterator接口适用于任何Java集合，不仅包括List<T>接口，也包括如Queue<T>, Deque<T>, Set<T>...

对于List，Iterator可以添加更多功能

其要求双向移动—单链表不支持 ListIterator

```
public interface ListIterator<T> extends Iterator<T>   // 综合迭代器
{
   boolean hasNext();       //正序遍历列表，若LL有元素便返回true
   T next();                //返回list下一个元素，前进一位
   boolean hasPrevious();   //逆序遍历列表，若LL有元素便返回true
   T previous();            //返回list上一个元素，后退一位
   int nextIndex();         //获得将由下一个next()调用返回的元素的index
   int previousIndex();     //  〃  〃  〃  previous()  〃  〃  〃
   void remove();           //删除返回的最后一个元素
   void set(T o);           //将next()或previous()返回的最后一个元素用元素o替代
   void add(T o);           //将元素o插入ListIterator<T>
}
```

ListIterator的功能实现

—双向，允许添加/删除对象

实现Listiterator的最佳方法 {
外部（方法不属于迭代类的一部分）
—列表顶部构建Listiterator对象，同时执行多个迭代
实现Listiterator的类为内部类—访问列表信息。
}