② 每次把尺寸翻倍

1° add为单一操作 blue

2° double size需要分配新数组并复制 red

$N = 2^k + 1$

频率会很低

Insert #	# of assigs	End array size
1	1	1
2	$2 = 1 + 2^0$	2
3	$3 = 1 + 2^1$	4
4	1	4
5	$5 = 1 + 2^2$	8
…	1	8
9	$9 = 1 + 2^3$	16
…	1	16
17	$17 = 1 + 2^4$	32
…	1	32
32	1	32

综上,对于N次add(),需 { N次实际添加; $2^0 + 2^1 + \dots + 2^x$ 复制 }

↓ $x = \log_2 N - 1$

$O(N) = N - 1$

⇓ N次add()操作为 $(N+N-1)/N = O(N)/N = O(1)$ - constant

换个简单思路, double操作的频率非常低,

等于每次直接插入.

因此,对于LinkedBag, add()摊销时间为O(1).

Lecture 11

从getEntry(int i)找List与Bag运行复杂度的区别:访问list任意位置

- 对于AList<T>,只需index数组,所以O(1)
- 对于LList<T>,需遍历i个Node才能得到Node i, O(N)是最坏情况

? 考虑一下average case

选到给定index的概率 $P(i) = \frac{1}{N}$, 查看该node需要i次操作 → Ops(i).

⇒ $P(i) \times Ops(i)$ [Idea: 选择位置i (这位置数), 需要Ops(i)次访问, i被选中概率为P(i)]

$$\text{Average Ops} = \text{Sum\_over\_i}[Ops(i) \times P(i)] = \frac{1}{N} \times \frac{N(N+1)}{2} = (N+1)/2 \Rightarrow O(N)$$

综上, { getEntry() for array O(1); getEntry() for linked list O(N) }

⊗ 但这不意味AList<T>优于LList<T>

⇓ add()与remove()的average case对 { AList为O(N); LList为O(1) }.

关于队列的实现

1) 把front固定在最前面(index 0)，从前面出去，整体向前移
2) 把back固定在前面(index 0)，从后面出去，整体向后移。

以上两者需shift

3) 前后循环移动

圆形的array(闭环，add和remove()不需shift)

假设把所有东西放进去再放出来

则开销时间为N个enqueue()，N个dequeue() => 共需2N个操作

resizing is O(1)

[先考虑front于0的队列

{ N个enqueue()：每一次为O(1)
{ N个dequeue()：需要 - 分配返回值
  \ 用shift以填补去除的gap

=> 1^{st} dequeue()：N-1 shift + 1 return
2^{nd} dequeue()：N-2 shift + 1 return.
N^{st} dequeue()：1 return.
⇓ total：N(N+1)/2 => $O(N^2)$.

每一次为O(N) ⇐

[考虑 circular queue

{ N个enqueue()：每一次为O(1).
{ N个dequeue()：由于不需要shift，每一次也为O(1) $\frac{O(N)+O(N)}{2N}$

⇓ amortized per operation 为 O(1)

Recursion 递归

Idea：某些问题P由更多的问题P'定义，P'本质与P类似

Req：{ - base case：不产生递归调用
{ - recursize case：算法根据自身定义，最终引出 base case

例，阶乘N!

- 具迭代 N! = N*(N-1)*(N-2)* ... *1
- 递归定义 N! = N * (N-1)! when N > 0
  N! = 1 when N = 0

分析：{ N=0 为 base case
{ N>0 为 recursize case
{ 递归调用有参数(N-1)，所以最终引发 base case

另例. 整数幂 X^N

- 迭代 $X^N = X * X * X * X \cdots X * X$ （N次）
- 递归 $X^N = X * X^{(N-1)}$ when N > 0 ; $X^N = 1$ when N = 0 ⇒分析 { N=0 为 base case; N>0 为 recursize case; 递归调用能最终引出 base case

阶乘的代码实现

```
public long factorial (int N)
{
   if (N < 0)
       throw new IllegalArgumentException();
   if (N <= 1)  // Isn't N = 0 the base case?
       return 1;
   return N * factorial(N-1);
}
```

// N<0时无法递归，条件不存在.

→N=1时结果相同，这是 base case

// recursize case

函数又使用 return 句中的表达式调用自身.

递归工作的2个思想

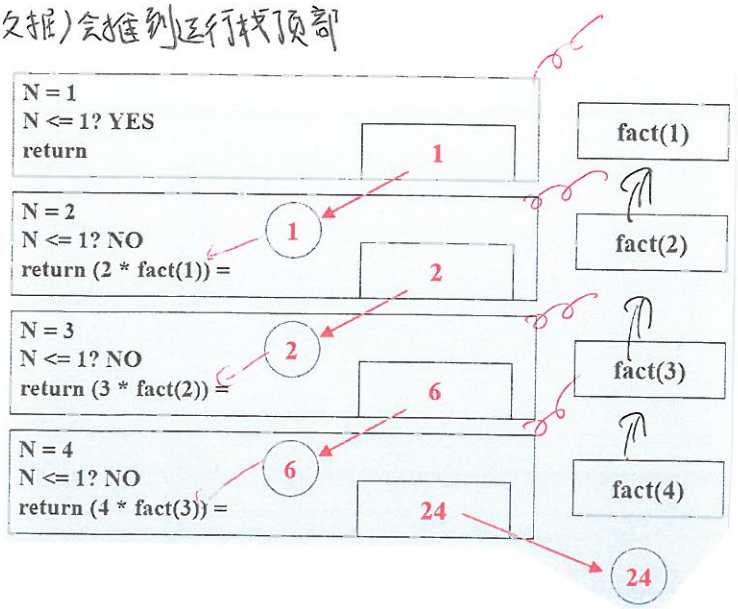
1) Activation Record (AR) 活动记录
   - 在函数调用期间，用于存储形参、本地变量、返回值(地址)的内存块
   - 每调用一次函数，便创建一次活动记录

2) Run-Time Stack (RTS) 运行栈
   以 Last In First Out (后进先出) 顺序维护 AR 的区域 图.

当一个方法被调用时，一个包含形参、返回地址、形参的 AR 被推到运行栈顶部
如果方法随后调自己，一个新的不同的 AR (包含新参数) 会推到运行栈顶部

运行栈顶部的 AR 代表正在执行的递归调用
下面的 AR 表示等待返回的先前调用

当顶部的调用终止，控制从顶部 AR 返回地址，随后顶部 AR 从运行栈弹出



先果再因

逐步判断，先给出情况...

Lecture12

另例. 顺序搜寻：通过检验每个元素来找到数组中的目标.

迭代：- 用 loop 遍历每个元素

- 用 contains() 方法，当动态 array bag 或 linked bag

其递归实现

先check length
- length == 0, 完成 (base case not found)
  ? 调用间length怎样变化
  答：通过更改beginning & ending index values来更新logical length
  由递归实现
- else, check first element
  > first element == key, (base case found)
  > 否则继续搜索剩下N-1个元素 (recursize case)

```
public static <T extends Comparable<? super T>>
                int recSeqSearch(T [] a, T key, int first)
{
        if (first >= a.length)
                return -1;          //数组logical size为0, no data found
        else if (a[first].compareTo(key) == 0)
                return first;       //找到, 返回该index
        else return recSeqSearch(a, key, first+1);
                                    //递归搜索(移向下一个位置, logical size-1)
}
```

链表的递归顺序搜索

```
public static <T extends Comparable<? super T>>
                int recLinkedSearch(Node<T> list, T key, int loc)
{
        if (list == null)
                return -1;          //此时logical size为0, no data found
        else if (list.getData().compareTo(key) == 0)
                return loc;         //第一项与key匹配了, 找到=>返回index
        else
                return (recLinkedSearch(list.getNextNode(), key, loc+1));
                                    //递归搜索下一个node, loc+1
}
```
logical size-1.

Divide & Conquer：将一个问题划分成多个小问题

子问题{是原问题的一部分(与递归不同)
本质与原问题相同

区别在于大小。

1) 如何将问题划分成子问题？
即递归前如何处理数据

2) 如何用子问题的解来解决原问题？
即递归后如何处理结果

⇒ 将以上应用于幂函数.

1) 分解 → cutting N in half
递归解决 $X^{N/2}$

对此，复杂度降为 $O(\log_2 N)$
(思想与二叉搜索相同)

2) 生成解
(仍需考虑奇偶问题)

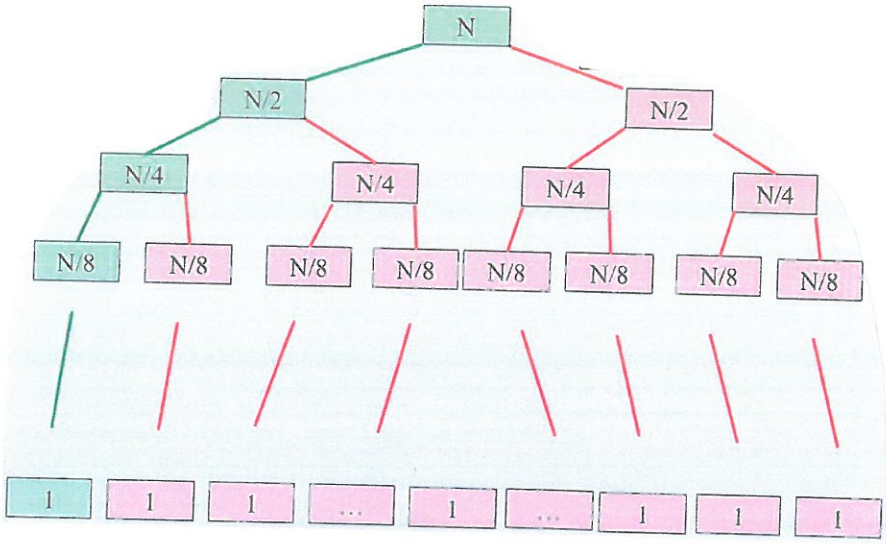
$X^N =$	$(X^{N/2})^2$	N even, greater than 0
$X^N =$	$(X^{N/2})^2 * X$	N odd, greater than 0
$X^N =$	1	N = 0

Lecture 14

注意：递归引发其它递归，分析其使效率！

$X^N = (X^{N/2})^2$　　N even, > 0
$X^N = X*(X^{N/2})^2$　N odd, > 0
$X^N = 1$　　　　　N = 0

绿色部分为初始版本
红色部分为多余操作



分析：－类似于二叉搜索
　　－运行栈高度趋于 $\log_2 N$

－每层1~2操作 ＜ square (always)
　　　　　　　　 odd N (sometimes)

另法：低效

$X^N = (X^{N/2})(X^{N/2})$　　N even, > 0
$X^N = X*(X^{N/2})(X^{N/2})$　N odd, > 0
$X^N = 1$　　　　　　N = 0

分析：－每次调用引发2个额外调用
　　－ O(N)次操作，相当于没有优化

⇒两者的相对高都为 $\log_2 N$，但前者是线性调用链，后者是完整二叉树。前者 $\log_2 N$ 次调用，后者 N 次调用

递归与二叉搜索（数据以顺序排列）
- divide：二分
- 子问题结果解决原问题

base case：两种 ＜ not found ⇒ 数组 logical size 降至 0
　　　　　　　　 found ⇒ key 匹配

recursize case：(中间元素为 M) ＜ M=S，完成，lead to base case
　　　　　　　　　　　　　 M<S，返回数组右侧的二叉搜索
　　　　　　　　　　　　　 M>S，返回数组左侧的二叉搜索

当调用返回时，只是"传递结果"
－返回左&右的二叉搜索

与迭代类似，将问题砍半

```
public <T extends Comparable<? super T>>
       int binarySearchr (T [] a, T obj, int low, int high)
{
     int ans;
     if (low <= high) //当low>high时，base case找不到了，因此 low<=high为递归的precon
     {
          int mid = (low + high)/2;
          T midItem = a[mid];
          int res = midItem.compareTo(obj);
          if (res < 0)
               return (binarySearchr(a, obj, mid + 1, high)); //中间值<key，右边递归
          else if (res > 0)
               return (binarySearchr(a, obj, low, mid - 1)); //中间值>key，左边递归
          else
               return mid; //中间值=key，找到
     }
     return -1;
}
```

tail recursive 可以使用迭代来实现，

即一个递归算法，其递归部分位于方法调用的 LAST 语句

而递归比等效的迭代更慢（Space：AR，RTS，Time：造成两者的时间）

递归的好处

1) 更自然，易于理解

[尾递归可以转换成迭代

2) 某些问题无法迭代解决，需要多个递归

[如回溯问题

回溯：持续寻找解决方法，直至实在无解

如穿越迷宫，遇到死胡同退回路口重走

目标是单用找到解决方案前进，前进的每一步都在递归

陷入困境时，回到上次调用，再次前进

最终结果：找到 solution / 尝试所有方案，确定 solution 不存在。

Lecture 15

回溯实例：8 queens problem（8X8的棋盘上放8个queen，任意两者不能处于同一行，列，斜线），

如何求解？

每行每列必须只有一个 queen

以递归角度思考，先放一个queen，再摆其它7个

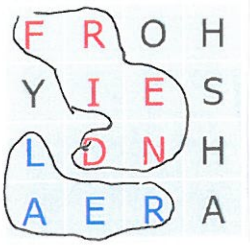
回溯出现：我们得撤销选择，换一个

迭代为何实现不了？：需存储 state information（冲突与否）

思想：有冲突解决冲突，
没有冲突往前走，
无路可走往回退，
走到最后是答案。

/另一个回溯问题：二维字母网格中找单词

上下左右找，一字只用一次



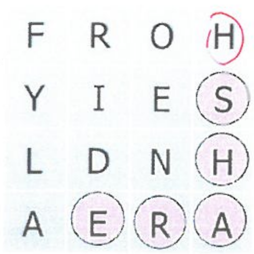
递归解决：每次递归调用考虑一个位置
递归 stack up，尝试匹配
一一对应即完成 => 一个单词k个字母，运行找k个调用
无法匹配即回溯

以寻找 SHARE 为例

row 1 找到 S，向上 H，
失败，回溯，向下 H

......

最终完成。



x => 回溯

r、c 当前行、列

word 试图匹配的单词

loc 在单词的当前位置

bo 字母网格

```
public boolean findWord(int r, int c, String word, int loc, char [][] bo)
```

递归操作

```
findWord(r, c+1, word, loc+1, bo) //右
findWord(r-1, c, word, loc+1, bo) //上
findWord(r, c-1, word, loc+1, bo) //左
findWord(r+1, c, word, loc+1, bo) //下
```

Lecture 16

另一个递归算法：Towers of Hanoi Problem 汉诺塔递归算法

问题描述：有3个塔，第一座有大小逐减的盘子，其它两塔目前是空的。目的是把盘子移至最后一塔。
操作时一次只能动一个盘子，且大盘子不能放到小盘子上。

如何递归实现？

有N个盘子，3个塔（start, mid, end），要把N个盘子从start移到end

{把N-1个盘子从start移到mid →
把最后一个盘子从start移到end
把N-1个盘子从mid移到end →

不能一次移N-1个？方法是递归的，解决N-1即可

```
public void solveHanoi(int sz, int strt, int mid, int end)
{                              //盘子数
    if (sz == 1)
        System.out.println("Move from " + strt + " to " + end); //只有一个盘子，直接移过去
    else
    {
        solveHanoi(sz-1, strt, end, mid);
        System.out.println("Move from " + strt + " to " + end); //自己输入输出移动方法
        solveHanoi(sz-1, mid, strt, end);
    }
}
```

```
sz=2 strt=1 mid=0 end=2
solveHanoi(1, 1, 2, 0);
(6) Move from 1 to 2
solveHanoi(1, 0, 1, 2);
```

为何难以迭代？单一递归的算法 线性调用

当又又递归调用时变二叉树

汉诺塔很短，在用重复调用，其运行时间！ O(2^N) 很大

Lecture 17

Sorting 排序

一般升序，即 i, j, A[i] <= A[j]

1) Insertion Sort 插入排序法

Idea：从原Array中删除元素，以正确顺序插入新数组（sort in place）

将数组想成sorted与unsorted

每次外循环把一个unsorted排入sorted正确位置。

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

```
public static <T extends Comparable<? super T>> void insertionSort(T[] a, int n)
{   insertionSort(a, 0, n - 1);          }//结束排序插入方法.
```

```
public static <T extends Comparable<? super T>>
                void insertionSort(T[] a, int first, int last)
{
        int unsorted, index;
        for (unsorted = first + 1; unsorted <= last; unsorted++)
        {       // Assertion: a[first] <= a[first + 1] <= ... <= a[unsorted - 1]
                T firstUnsorted = a[unsorted];
                insertInOrder(firstUnsorted, a, first, unsorted - 1); //将元素插入正确位置，
        }
}
```

重要的for loop

```java
private static <T extends Comparable<? super T>>
                void insertInOrder(T element, T[] a, int begin, int end)
{
        int index;
        for (index = end; (index >= begin) && (element.compareTo(a[index]) < 0); index--)
        {
                a[index + 1] = a[index]; //找到插入项目的合适位置, 从后往前
        }
        a[index + 1] = element;
}
```

以上代码的思想：一初始方法的参数只有数组和长度

一重载版本将begin index和end index作为参数；从而只对部分数组排序

< 每次迭代将unsorted的一个元素带入sorted中

调用另一个方法将该元素放到正确位置　　从左到右

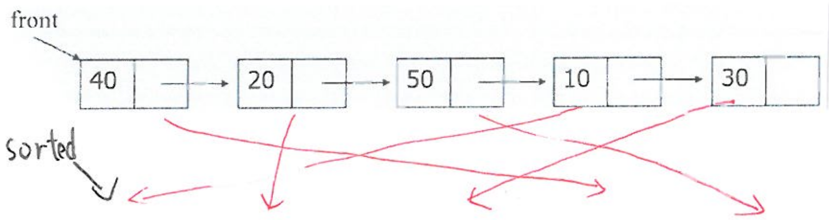
?插入排序的运行时间(在于数组元素的比较)

最坏情况?完全为反序的数据.　$1+2+\cdots+N-1 = \frac{N(N-1)}{2} \Rightarrow O(N^2)$

average 无数量级变化, 仍为$O(N^2)$

插入排序在链表的实现

Idea: 每次迭代, 从老列表中删除节点引用, 把其插入到新列表. [不需创造新节点, 只用移动]



拆散了, 重新匹配

SelectionSort 选择排序

Idea: 通过第i次迭代的外循环, 找到第i小的元素, 将其置换到位置i　　共需N-1次迭代

```java
public static <T extends Comparable<? super T>>
       void selectionSort(T[] a, int n) //(数组, 长度)
{
    for (int index = 0; index < n - 1; index++) //循环条件从index 0开始, 递增
    {
        int indexOfNextSmallest = getIndexOfSmallest(a,index,n-1); → 获得最小值的index
        swap(a, index, indexOfNextSmallest); //替换操作(数组 a, ..., ...) 将后两者交换
    }                                        将其swap到, 进入循环
}

private static <T extends Comparable<? super T>>
       int getIndexOfSmallest(T[] a, int first, int last) //数组, 头, 尾
{
    T min = a[first]; //假设a[first]是最小的
    int indexOfMin = first; //获取index
    for (int index = first + 1; index <= last; index++) //从头到尾++的loop
    {
        if (a[index].compareTo(min) < 0) //与min即a[first]相比
        {
            min = a[index];  //与first替换
            indexOfMin = index;
        }
    }
    return indexOfMin; //返回索引
}
```

选择排序的运行时间(关键指令为比较)

插入排序与选择排序的比较：它俩都用了for循环

项目插对就停止　　循环中不比较，循环的迭代基于index　外loop，i 从 0 ~ N-2；内loop，j 从 i+1 ~ N-1 → 比较

从而计算选择排序的复杂度

外 i = 0，内进行 N-1次比较(1 ~ N-1)

⋮

i = N-2，　1次比较

$\Rightarrow \frac{N(N-1)}{2} \Rightarrow O(N^2)$

Bubblesort 冒泡排序

元素 j 与 j+1 相比，j<j+1 便更换位置

例：1st 外循环，将80放对。

2nd　，将70放入index5

⋮

其时间复杂度为 $O(N^2)$

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

插入排序与选择排序都只有一个递归调用，没有分而治之。

Lecture 18

Shellsort 希尔排序(插入排序的优化版)

Idea：不比较相邻，比较距离较远(距离为k)的项目

⇓ 我们对原数组的子数组插入排序，子数组相距k

k逐渐减小，以1结尾(1即直接插入排序)

其复杂度：$O(N^{\frac{3}{2}})$

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

```
public static <T extends Comparable<? super T>>
       void shellSort(T[] a, int first, int last)
{
   int n = last - first + 1;   //数组元素个数
   int space = n / 2;   //initial gap
   while (space > 0)   //持续直到gap为0
   {
         for (int begin = first; begin < first + space; begin++)
         {
                 incrementalInsertionSort(a, begin, last, space);
                                     //插入排序由现有gap决定
         }
         space = space / 2;   //缩减gap
   }
}
```

```
private static <T extends Comparable<? super T>>
   void incrementalInsertionSort(T[] a, int first,
                                 int last, int space)  //引入gap变量.
{
   int unsorted, index;
   for (unsorted = first+space; unsorted<=last;
                          unsorted=unsorted+space)
   {
      T nextToInsert = a[unsorted];
      index = unsorted - space;
      while ((index >= first) &&
                mextToInsert.compareTo(a[index] < 0))
      {
           a[index + space] = a[index];
           index = index - space;
      }

      a[index + space] = nextToInsert;
   }
} /
```

(外: outer for loop; 内: inner while loop)

更好的方法? '分而治之

排序更小的数组来排序原数组

1) 如何切割问题? 减半

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

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

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

持续至size1
子数组由索引限制确定

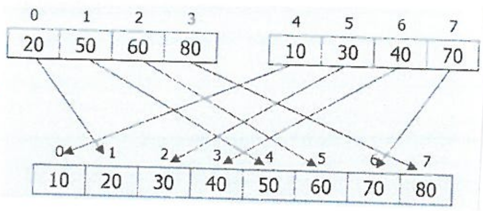
0	1	2	3	4	5	6	7
50	80	60	20	30	10	70	40

2) 用子解解决总问题?

此时我们有两个已排序的子数组,

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

=> Merge!!
compare items & move them



Mergesort

```
private static <T extends Comparable<? super T>>
        void merge(T[] a, T[] tempArray, int first, int mid, int last) //前后两部分的子数组
{                                              都为index
     int beginHalf1 = first;      int endHalf1 = mid;
     int beginHalf2 = mid + 1;    int endHalf2 = last;

     int index = beginHalf1; //从bh1为头index
     for (; (beginHalf1 <= endHalf1) && (beginHalf2 <= endHalf2); index++) //确保还能滴出的precon
     {
          if (a[beginHalf1].compareTo(a[beginHalf2]) <= 0)
          {
               tempArray[index] = a[beginHalf1];
               beginHalf1++;
          }
          else
          {
               tempArray[index] = a[beginHalf2];
               beginHalf2++;
          }
     }

     for (; beginHalf1 <= endHalf1; beginHalf1++, index++)
        tempArray[index] = a[beginHalf1];

     for (; beginHalf2 <= endHalf2; beginHalf2++, index++)
        tempArray[index] = a[beginHalf2];
                                               //copy进temp array
     for (index = first; index <= last; index++)
        a[index] = tempArray[index];  //复制回原array
} //结束分出合
```

0	1	2	3		4	5	6	7
20	50	60	80		10	30	40	70
bh1			eh1		bh2			eh2

tempArray

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

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

a

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