

列表

插入排序



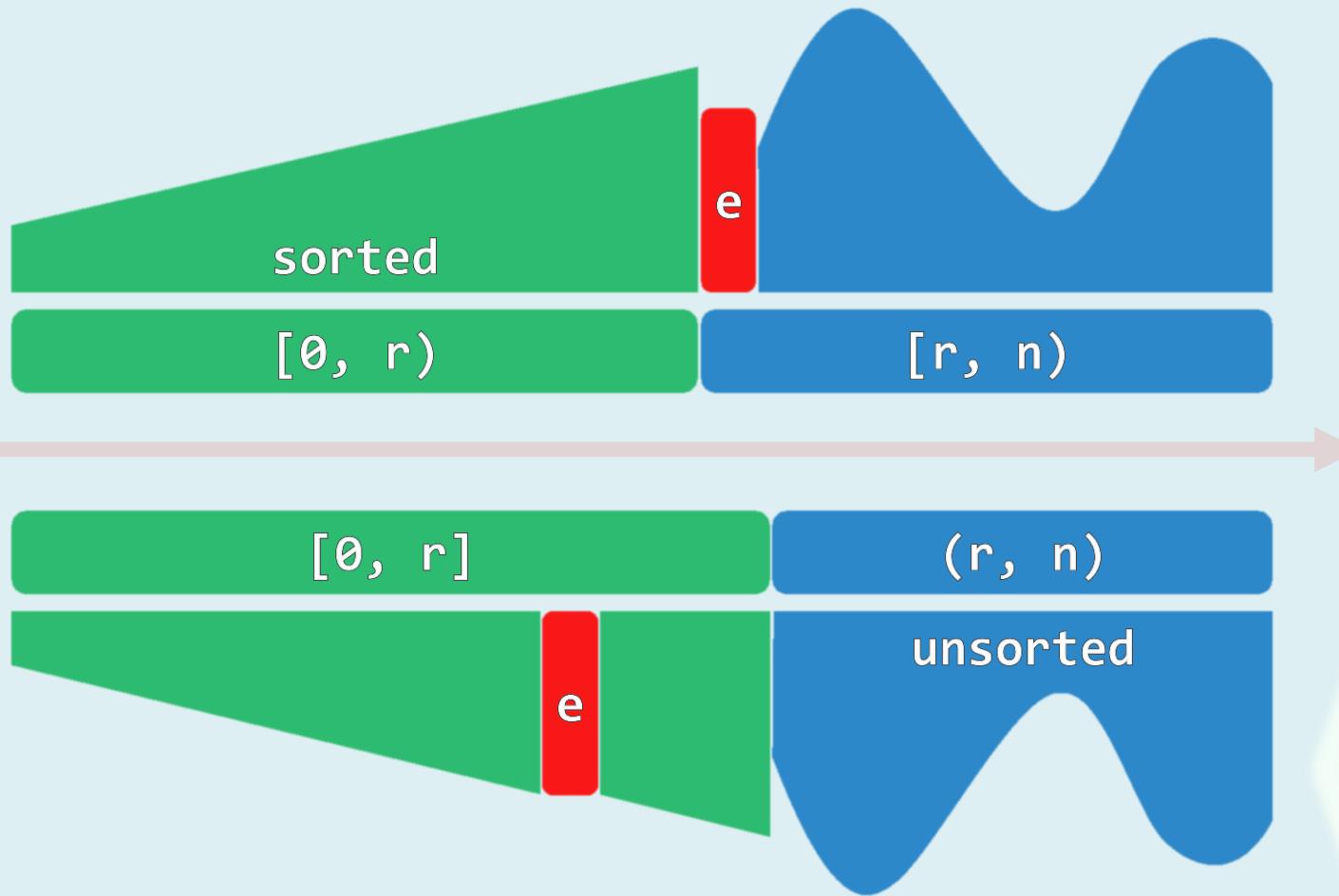
- G

邓俊辉

deng@tsinghua.edu.cn

一语未了，只见宝玉笑嘻嘻的捐了一枝红梅进来，众丫鬟忙已  
接过，插入瓶内。

# 减而治之



❖ 不变性

序列总能视作两部分：

$$S[0, r) + U[r, n)$$

❖ 初始化

$$|S| = r = 0$$

❖ 反复地，针对  $e = A[r]$

在  $S$  中 **查找适当位置，以插入  $e$**

❖ 二分查找？顺序查找？

# 实例

迭代轮次	前缀有序子序列	当前元素	后缀无序子序列
-1	^	^	5 2 7 4 6 3 1
0	^	5	2 7 4 6 3 1
1	(5)	2	7 4 6 3 1
2	(2) 5	7	4 6 3 1
3	2 5 (7)	4	6 3 1
4	2 (4) 5 7	6	3 1
5	2 4 5 (6) 7	3	1
6	2 (3) 4 5 6 7	1	^
7	(1) 2 3 4 5 6 7	^	^

# 实现

```
//对列表中起始于位置p、宽度为n的区间做插入排序，valid(p) && rank(p) + n <= size
template <typename T> void List<T>::insertionSort( Posi(T) p, int n ) {
    for ( int r = 0; r < n; r++ ) { //逐一引入各节点，由Sr得到Sr+1
        insertA( search( p->data, r, p ), p->data ); //查找 + 插入
        p = p->succ; remove( p->pred ); //转向下一节点
    } //n次迭代，每次O(r + 1)
} //仅使用O(1)辅助空间，属于就地算法
```

❖ 紧邻于search()接口返回的位置之后插入当前节点，总是保持有序

❖ 验证各种情况下的正确性，体会哨兵节点的作用：

S<sub>r</sub>中含有/不含与p相等的元素；S<sub>r</sub>中的元素均严格小于/大于p

# 性能分析

- ❖ 属于就地算法 ( in-place )
- ❖ 属于在线算法 ( online )
- ❖ 具有输入敏感性 ( input sensitivity )
  - 后面将会看到 : Shellsort 之类算法的高效性 , 完全依赖于 insertionsort 的这一特性
- ❖ 最好情况 : 完全 ( 或几乎 ) 有序
  - 每次迭代 , 只需 1 次比较 , 0 次交换 : 累计  $\mathcal{O}(n)$  时间 !
- ❖ 最坏情况 : 完全 ( 或几乎 ) 逆序
  - 第 k 次迭代 , 需  $\mathcal{O}(k)$  次比较 , 1 次交换 : 累计  $\mathcal{O}(n^2)$  时间 !
- ❖ “优化” 的可能 : 在有序前缀中的查找定位 , 为何采用了顺序查找 , 而不是二分查找 ?

## 平均性能：后向分析

- 若各元素的取值系**独立均匀分布**，**平均要做多少次元素比较**？
- 考查 $e=[r]$ 刚插入完成的那一时刻...此时的有序前缀 $[0, r]$ 中，谁是 $e$ ？
- 观察：其中的 $r+1$ 个元素均有可能，且概率均为 $1/(r+1)$

因此，在刚完成的这次迭代中

为引入 $s[r]$ 所花费时间的数学期望为

$$1 + \sum_{k=0}^r k/r + 1 = 1 + r/2$$

于是，总体时间的数学期望为  $\sum_{r=0}^{n-1} (1 + r/2) = \mathcal{O}(n^2)$

再问：在 $n$ 次迭代中，平均有多少次**无需交换**呢？ //习题[3-10]

