

2. 向量

抽象数据类型

接口与实现

邓俊辉

deng@tsinghua.edu.cn

Abstract Data Type vs. Data Structure

❖ **抽象数据类型** = 数据模型 + 定义在该模型上的一组操作

抽象定义

外部的逻辑特性

操作&语义

一种定义

不考虑时间复杂度

不涉及数据的存储方式

数据结构 = 基于某种特定语言，实现ADT的一整套算法

具体实现

内部的表示与实现

完整的算法

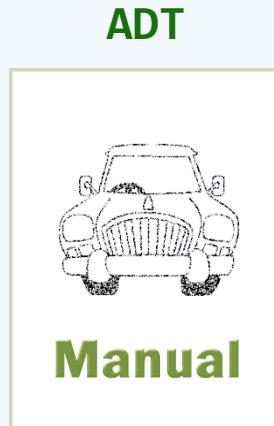
多种实现

与复杂度密切相关

要考虑数据的具体存储机制



Application



Interface



Implementation

Application = Interface × Implementation

❖ 在数据结构的具体实现与实际应用之间

ADT就分工与接口制定了统一的规范

- 实现：高效兑现数据结构的ADT接口操作

//做冰箱、造汽车

- 应用：便捷地通过操作接口使用数据结构

//用冰箱、开汽车

❖ 按照ADT规范

- 高层算法设计者与底层数据结构实现者

可高效地分工协作

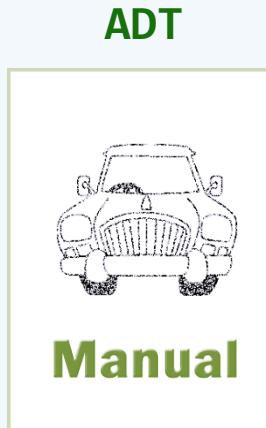
- 不同的算法与数据结构可以便捷组合

- 每种操作接口只需统一地实现一次

代码篇幅缩短，软件复用度提高



Application



Interface



Implementation

2. 向量

抽象数据类型 从数组到向量

阿圆眼快，把手一点说：“到了，就是这里。妈妈，你只管找号头，311，就是爸爸的号。”

邓俊辉

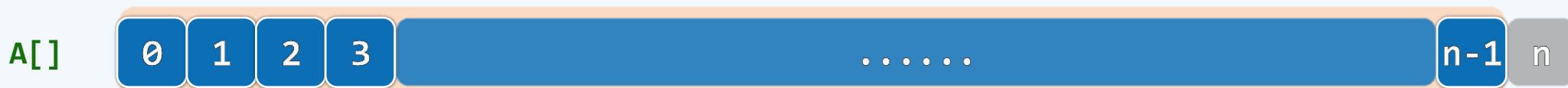
deng@tsinghua.edu.cn

贵贱长少，秩秩焉，莫不从桓公而贵敬之，是天下之大节也。

循秩访问

- ❖ C/C++语言中，数组元素与编号一一对应： $A[0], A[1], A[2], \dots, A[n-1]$
- ❖ 反之，元素各由编号唯一指代，并可直接访问 //故亦称作线性数组 (linear array)

$A[i]$ 的物理地址 = $A + i \times s$, s 为单个元素占用的空间量



- ❖ 向量是数组的抽象与泛化，由一组元素按线性次序**封装**而成

- 各元素与 $[0, n)$ 内的**秩** (rank) 一一对应

```
typedef int Rank; //循秩访问 ( call-by-rank )
```

- 操作、管理维护更加简化、统一与安全

- 元素类型可灵活选取，便于**定制**复杂数据结构 //Vector< PFCTree* > pfcForest;

向量ADT接口

操作	功能	适用对象
<code>size()</code>	报告向量当前的规模（元素总数）	向量
<code>get(r)</code>	获取秩为r的元素	向量
<code>put(r, e)</code>	用e替换秩为r元素的数值	向量
<code>insert(r, e)</code>	e作为秩为r元素插入，原后继依次后移	向量
<code>remove(r)</code>	删除秩为r的元素，返回该元素原值	向量
<code>disordered()</code>	判断所有元素是否已按非降序排列	向量
<code>sort()</code>	调整各元素的位置，使之按非降序排列	向量
<code>find(e)</code>	查找目标元素e	向量
<code>search(e)</code>	查找e，返回不大于e且秩最大的元素	有序向量
<code>deduplicate(), uniquify()</code>	剔除重复元素	向量/有序向量
<code>traverse()</code>	遍历向量并统一处理所有元素	向量

ADT操作实例

操作	输出	向量组成 (自左向右)	操作	输出	向量组成 (自左向右)
初始化			disordered()	3	4 3 7 4 9 6
insert(0, 9)	9	9	find(9)	4	4 3 7 4 9 6
insert(0, 4)	4 9	4 9	find(5)	-1	4 3 7 4 9 6
insert(1, 5)	4 5 9	4 5 9	sort()		3 4 4 6 7 9
put(1, 2)	4 2 9	4 2 9	disordered()	0	3 4 4 6 7 9
get(2)	9	4 2 9	search(1)	-1	3 4 4 6 7 9
insert(3, 6)	4 2 9 6	4 2 9 6	search(4)	2	3 4 4 6 7 9
insert(1, 7)	4 7 2 9 6	4 7 2 9 6	search(8)	4	3 4 4 6 7 9
remove(2)	2	4 7 9 6	search(9)	5	3 4 4 6 7 9
insert(1, 3)	4 3 7 9 6	4 3 7 9 6	search(10)	5	3 4 4 6 7 9
insert(3, 4)	4 3 7 4 9 6	4 3 7 4 9 6	uniquify()		3 4 6 7 9
size()	6	4 3 7 4 9 6	search(9)	4	3 4 6 7 9

STL Vector

```
❖ #include <iostream>

#include <vector>

using namespace std;

❖ vector<int> v; // an empty vector of integers

vector<int> s( 32, 63 ); // { 63, 63, 63, ..., 63 }, sum = 2016

s.insert( s.begin + 2, 2017 ); // { 63, 63, 2017, 63, ..., 63 }

s.erase( s.end - 30, s.end ); // { 63, 63, 2017 }

for ( i = 0; i < s.size(); i++ )

    cout << s[i] << endl;
```

2. 向量

抽象数据类型

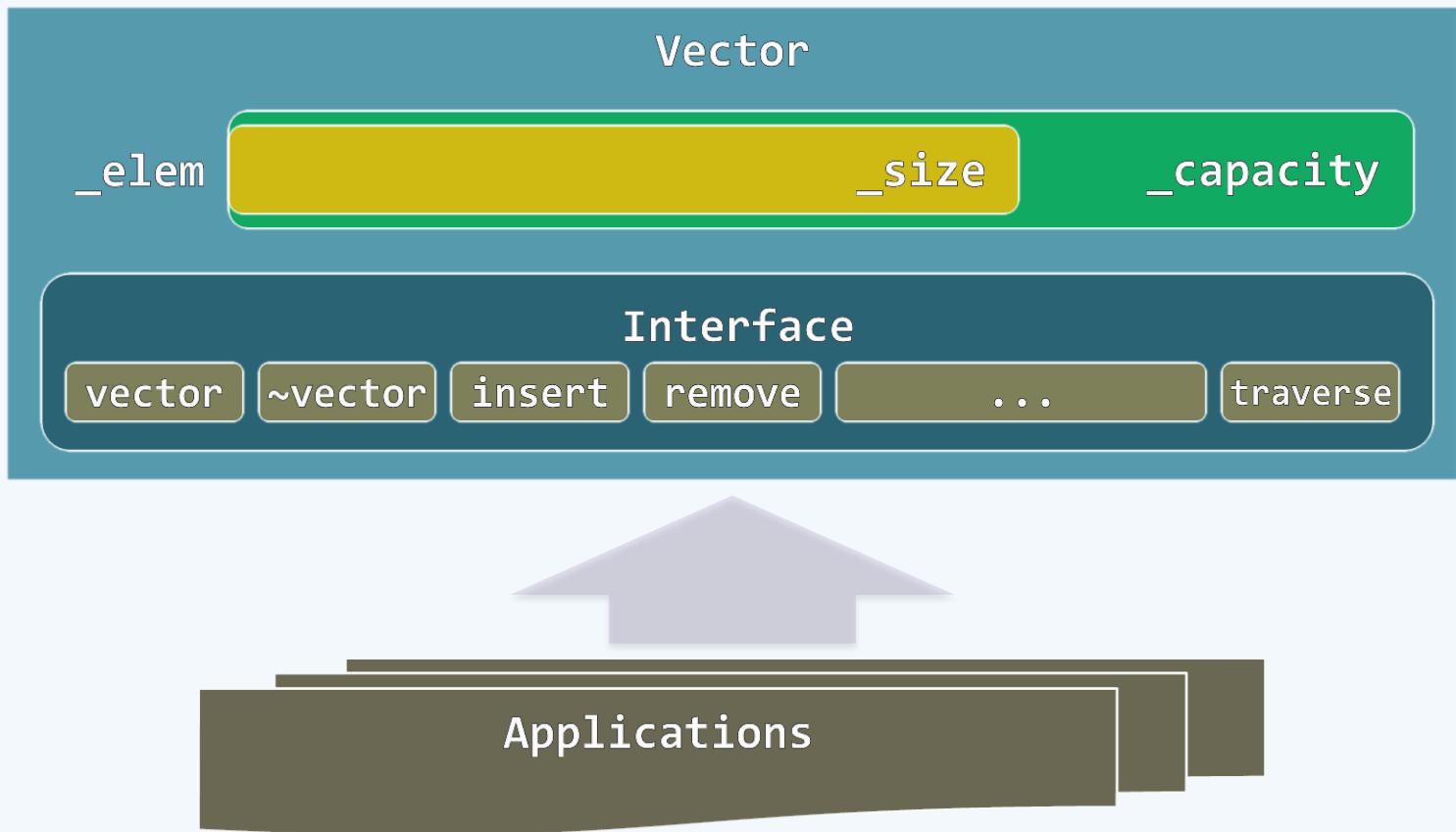
模板类

官职须由生处有，文章不管用时无
堪笑翰林陶学士，年年依样画葫芦

邓俊辉

deng@tsinghua.edu.cn

```
template <typename T> class Vector { //向量模板类  
private: Rank _size; int _capacity; T* _elem; //规模、容量、数据区  
  
protected:  
    /* ... 内部函数 */  
  
public:  
    /* ... 构造函数 */  
    /* ... 析构函数 */  
    /* ... 只读接口 */  
    /* ... 可写接口 */  
    /* ... 遍历接口 */  
};
```



构造 + 析构

```
❖ #define DEFAULT_CAPACITY 3 //默认初始容量(实际应用中可设置为更大)

❖ Vector( int c = DEFAULT_CAPACITY )
    { _elem = new T[ _capacity = c ]; _size = 0; } //默认

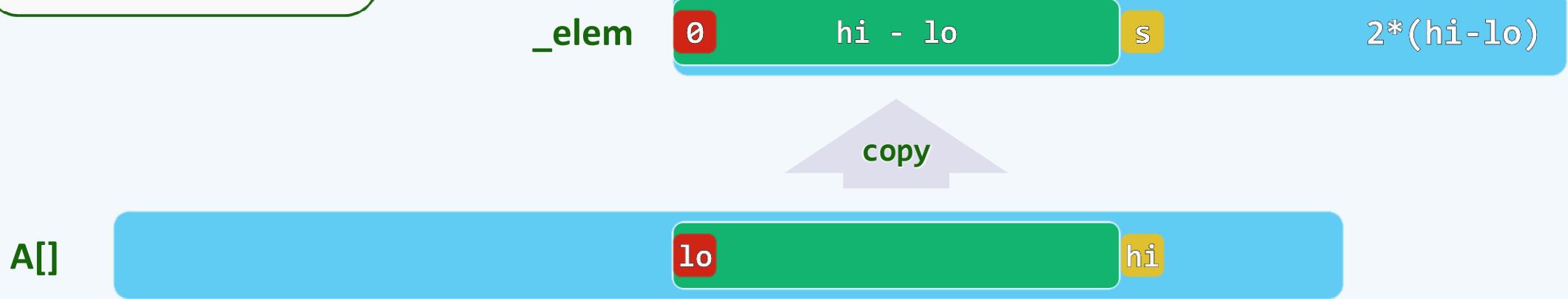
❖ Vector( T const * A, Rank lo, Rank hi ) //数组区间复制
    { copyFrom( A, lo, hi ); }

Vector( Vector<T> const& V, Rank lo, Rank hi ) //向量区间复制
    { copyFrom( V._elem, lo, hi ); }

Vector( Vector<T> const& V ) //向量整体复制
    { copyFrom( V._elem, 0, V._size ); }

❖ ~Vector() { delete [] _elem; } //释放内部空间
```

基于复制的构造



❖ template <typename T> //T为基本类型，或已重载赋值操作符'='

```
void Vector<T>::copyFrom( T const * A, Rank lo, Rank hi ) {  
    _elem = new T[ _capacity = 2 * (hi - lo) ]; //分配空间  
    _size = 0; //规模清零  
    while ( lo < hi ) //A[lo, hi)内的元素逐一  
        _elem[ _size++ ] = A[ lo++ ]; //复制至_elem[0, hi - lo)  
} //O(hi - lo) = O(n)
```

向量

可扩充向量：算法

一个人办一县事，要有一省的眼光；
办一省事，要有一国之眼光；
办一国事，要有世界的眼光。

其实“我”不需扩大，宇宙只是一个“我”，只有在
我们精神往下陷落时，宇宙与我才分开

邓俊辉

deng@tsinghua.edu.cn

静态空间管理

- ❖ 开辟内部数组 `_elem[]` 并使用一段地址连续的物理空间

`_capacity` : 总容量

`_size` : 当前的实际规模n

`_elem`

`_size`

`_capacity`

- ❖ 若采用静态空间管理策略，容量 `_capacity` 固定，则有明显的不足...

- ❖ - 上溢/overflow : `_elem[]` 不足以存放所有元素，尽管此时系统往往仍有足够的空间

- 下溢/underflow : `_elem[]` 中的元素寥寥无几

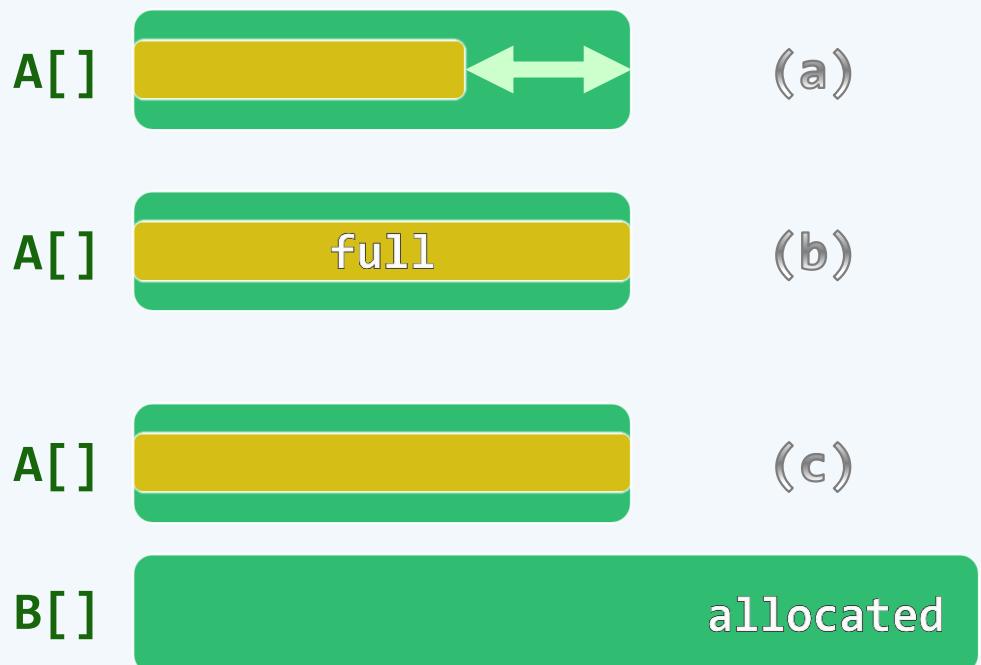
- 装填因子/load factor : $\lambda = \text{_size}/\text{_capacity} \ll 50\%$

- ❖ 更糟糕的是，一般的应用环境中难以准确预测空间的需求量

- ❖ 可否使得向量可随实际需求动态调整容量，并同时保证高效率？

动态空间管理

- ❖ 蝉的哲学：身体经过一段时间的生长，会蜕去原先的外壳，代之以**更大的新外壳**
- ❖ 向量：在即将上溢时，适当扩大内部数组的容量



扩容算法

```
❖ template <typename T> void Vector<T>::expand() { //向量空间不足时扩容  
    if ( _size < _capacity ) return; //尚未满员时，不必扩容  
    _capacity = max( _capacity, DEFAULT_CAPACITY ); //不低于最小容量  
    T* oldElem = _elem; _elem = new T[ _capacity <<= 1 ]; //容量加倍  
    for ( int i = 0; i < _size; i++ ) //复制原向量内容  
        _elem[i] = oldElem[i]; //T为基本类型，或已重载赋值操作符'= '  
    delete [] oldElem; //释放原空间  
} //得益于向量的封装，尽管扩容之后数据区的物理地址有所改变，却不易出现野指针
```

❖ 为何必须采用**容量加倍**策略呢？其它策略是否可行？

向量

可扩充向量：分摊

...在他的心理上，他总以为北平是天底下最可靠的大城，不管有什么灾难，到三个月必定灾消难满，而后诸事大吉。北平的灾难恰似一个人免不了有些头疼脑热，过几天自然会好了的。

邓俊辉

deng@tsinghua.edu.cn

容量递增策略

- ❖ `T* oldElem = _elem; _elem = new T[_capacity += INCREMENT]; //追加固定增量`
- ❖ 最坏情况：在初始容量 θ 的空向量中，连续插入 $n = m*I \gg 2$ 个元素...
- ❖ 于是，在第1、 $I + 1$ 、 $2I + 1$ 、 $3I + 1$ 、...次插入时，都需扩容
- ❖ 即便不计申请空间操作，各次扩容过程中复制原向量的时间成本依次为

$0, I, 2I, \dots, (m-1)*I$ //算术级数

总体耗时 = $I * (m-1) * m/2 = \Theta(n^2)$ ，每次扩容的分摊成本为 $\Theta(n)$

$(m-1)*I$

$m * I = \Theta(n)$

...

.....

...

.....

$2*I$

$3 * I$

I

$2 * I$

θ

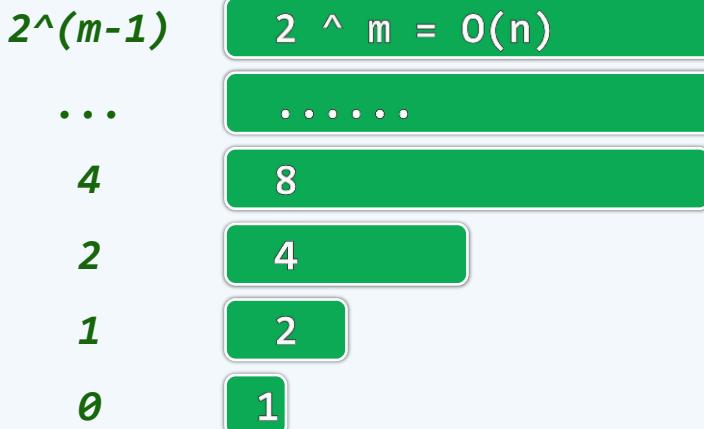
Increment

容量加倍策略

- ❖ `T* oldElem = _elem; _elem = new T[_capacity <<= 1];` //容量加倍
- ❖ 最坏情况：在初始容量 1 的满向量中，连续插入 $n = 2^m \gg 2$ 个元素...
- ❖ 于是，在第 $1, 2, 4, 8, 16, \dots$ 次插入时都需扩容
- ❖ 各次扩容过程中复制原向量的时间成本依次为

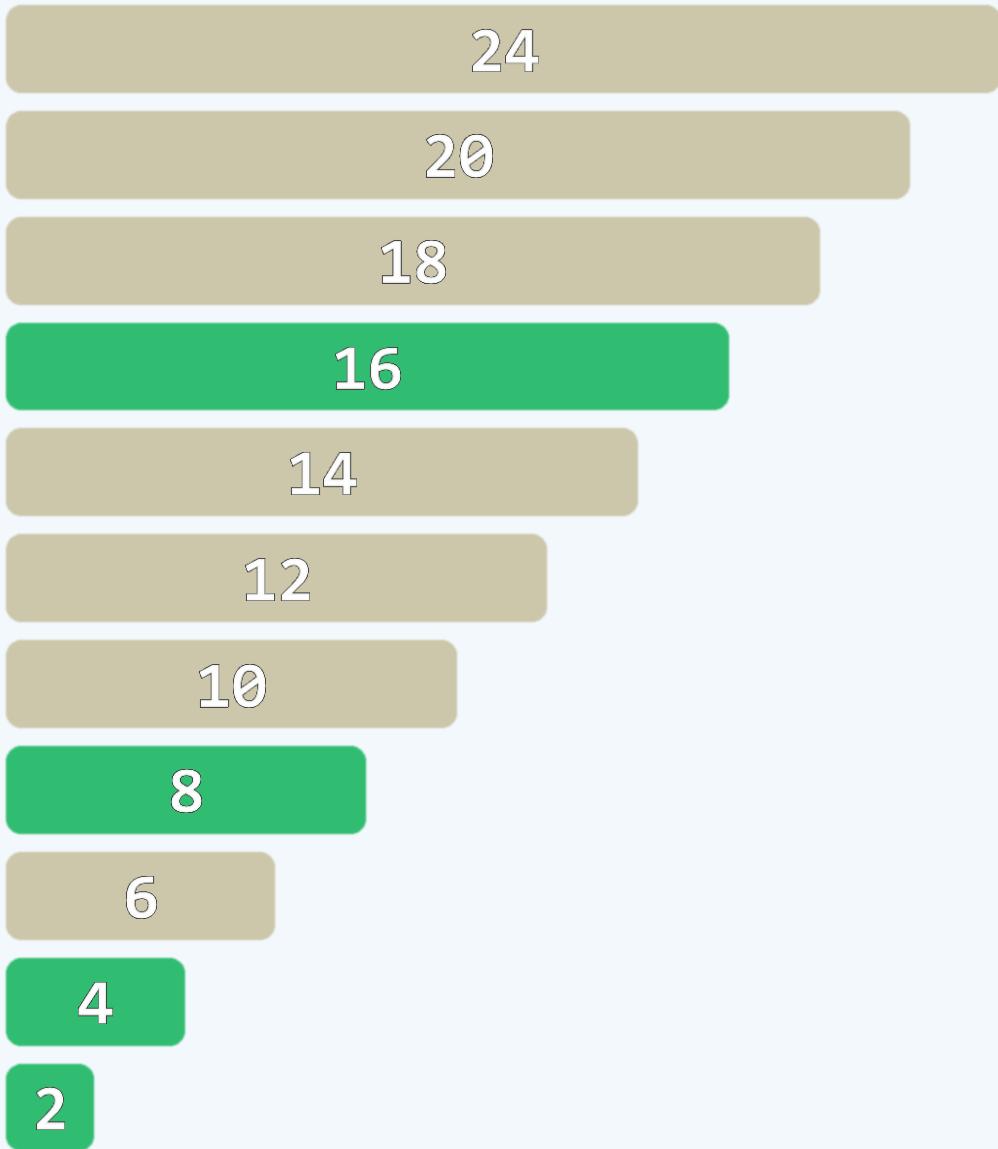
$1, 2, 4, 8, \dots, 2^m = n$ //几何级数

总体耗时 = $\mathcal{O}(n)$ ，每次扩容的分摊成本为 $\mathcal{O}(1)$



对比

	递增策略	倍增策略
累计增容时间	$\Theta(n^2)$	$\Theta(n)$
分摊增容时间	$\Theta(n)$	$\Theta(1)$
装填因子	$\approx 100\%$	$> 50\%$



平均分析 vs. 分摊分析

❖ 平均复杂度 (average complexity)

根据数据结构各种操作出现概率的分布，将对应的成本加权平均

- 各种可能的操作，作为**独立事件**分别考查
- 割裂了操作之间的**相关性和连贯性**
- 往往**不能准确地评判**数据结构和算法的**真实性能**

❖ 分摊复杂度 (amortized complexity)

对数据结构**连续地实施足够多次操作**，所需**总体成本**分摊至**单次操作**

- 从实际可行的角度，对一系列操作做整体的考量
- 更加**忠实地刻画了**可能出现的操作序列
- 更为**精准地评判**数据结构和算法的**真实性能**

❖ 后面将看到更多、更复杂的例子

向量

无序向量：基本操作

邓俊辉

deng@tsinghua.edu.cn

它污浊，它美丽，它衰老，它活泼，它杂乱，它安闲，它可爱，它是伟大的夏初的北平。

元素访问

❖ `v.get(r)` 和 `v.put(r,e)` 不够便捷、直观，可否沿用数组的访问方式 `v[r]` ?

可以！比如，通过重载下标操作符“`[]`”

❖ `template <typename T> //可作为左值 : v[r] = (T) (2*x + 3)`

```
T & vector<T>::operator[]( Rank r ) { return _elem[ r ]; }
```

❖ `template <typename T> //仅限于右值 : T x = v[r] + u[s] * w[t]`

```
const T & Vector<T>::operator[]( Rank r ) const { return _elem[ r ]; }
```

❖ 这里采用了简易的方式处理意外和错误（比如，入口参数约定：`0 <= r < _size`）

实际应用中，应采用更为严格的方式

插入

❖ template <typename T> //e作为秩为r元素插入， $0 \leq r \leq \text{size}$

```
Rank Vector<T>::insert( Rank r, T const & e ) { //O(n-r)
    expand(); //若有必要，扩容
    for ( int i = _size; i > r; i-- ) //自后向前
        _elem[i] = _elem[i - 1]; //后继元素顺次后移一个单元
    _elem[r] = e; _size++; return r; //置入新元素，更新容量，返回秩
}
```

(a) [0, n) : may be full

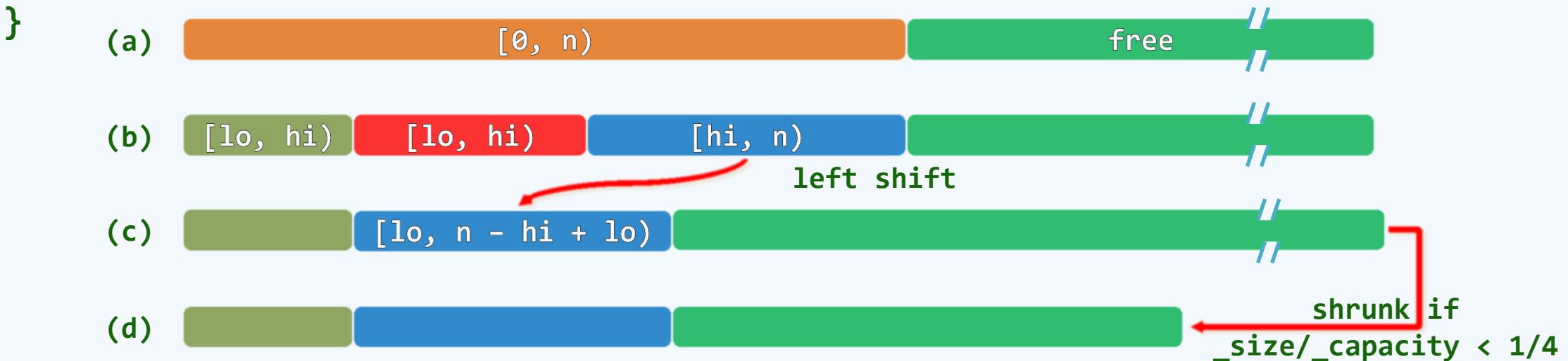
(b) [0, r) [r, n) expanded if necessary
right shift

(c) . (r, n]

(d) e

区间删除

```
❖ template <typename T> //删除区间[lo, hi), 0 <= lo <= hi <= size  
int Vector<T>::remove( Rank lo, Rank hi ) { //O(n-hi)  
    if ( lo == hi ) return 0; //出于效率考虑，单独处理退化情况  
    while ( hi < _size ) _elem[ lo ++ ] = _elem[ hi ++ ]; // [hi, _size)顺次前移  
    _size = lo; shrink(); //更新规模，若有必要则缩容  
    return hi - lo; //返回被删除元素的数目  
}
```



单元素删除

❖ template <typename T>

```
T Vector<T>::remove( Rank r ) {  
    T e = _elem[r]; //备份  
    remove( r, r+1 ); // “区间” 删除  
    return e; //返回被删除元素  
} // $\theta(n-r)$ ,  $0 \leq r < size$ 
```

❖ 也就是将单元素视作区间的特例：

$[r] = [r, r + 1)$

❖ 反过来，通过反复调用remove(r)接口实现

remove(lo, hi)呢？

❖ 每次循环耗时，正比于删除区间的后缀长度

$$n - hi = \mathcal{O}(n)$$

而循环次数等于区间宽度

$$hi - lo = \mathcal{O}(n)$$

如此，将导致总体 $\mathcal{O}(n^2)$ 的复杂度

向量

无序向量：查找

他便站将起来，背着手踱来踱去，侧眼把那些人逐个个觑将去，内中一个果然衣领上挂着一寸来长短彩线头。

邓俊辉

deng@tsinghua.edu.cn

无序向量：判等器

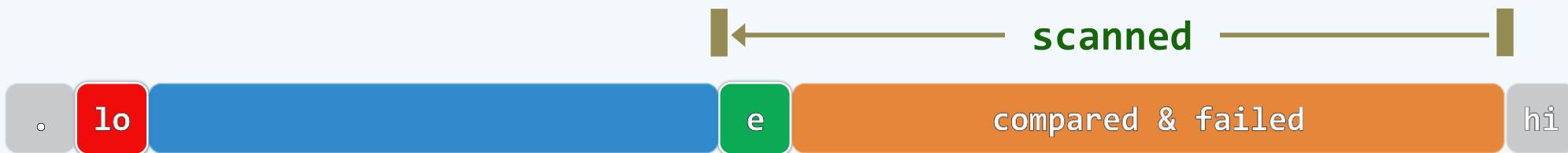
```
❖ template <typename K, typename V> struct Entry { //词条模板类  
    K key; V value; //关键码、数值  
  
    Entry ( K k = K(), V v = V() ) : key ( k ), value ( v ) {}; //默认构造函数  
  
    Entry ( Entry<K, V> const& e ) : key ( e.key ), value ( e.value ) {}; //克隆  
  
    bool operator== ( Entry<K, V> const& e ) { return key == e.key; } //等于  
    bool operator!= ( Entry<K, V> const& e ) { return key != e.key; } //不等  
  
    /* ... */  
  
    /* ... */  
  
};
```

有序向量：比较器

```
❖ template <typename K, typename V> struct Entry { //词条模板类  
    K key; V value; //关键码、数值  
  
    Entry ( K k = K(), V v = V() ) : key ( k ), value ( v ) {}; //默认构造函数  
  
    Entry ( Entry<K, V> const& e ) : key ( e.key ), value ( e.value ) {}; //克隆  
  
    bool operator== ( Entry<K, V> const& e ) { return key == e.key; } //等于  
    bool operator!= ( Entry<K, V> const& e ) { return key != e.key; } //不等于  
    bool operator< ( Entry<K, V> const& e ) { return key < e.key; } //小于  
    bool operator> ( Entry<K, V> const& e ) { return key > e.key; } //大于  
}; //得益于比较器和判等器，从此往后，不必严格区分词条及其对应的关键码
```

顺序查找

```
❖ template <typename T> Rank Vector<T>:: // $\theta(hi - lo) = \theta(n)$ 
    find( T const & e, Rank lo, Rank hi ) const { // $0 \leq lo < hi \leq _size$ 
        while ( (lo < hi--) && (e != _elem[hi]) ); //逆向查找
        return hi; //hi < lo意味着失败；否则hi即命中者的秩（多个命中时，返回最大的秩）
    } //Excel::match(e, range, type)
```



❖ 输入敏感 (input-sensitive) : 最好 $\theta(1)$, 最差 $\theta(n)$

向量

无序向量：去重

鳳兮鳳兮，故是一鳳

你去问问你琏二婶子，正月里请吃年酒的日子拟了没有。若拟定了，叫书房里明白开了单子来，咱们再请时，就不能重犯了。旧年不留心重了几家，不说咱们不留神，倒象两宅商议定了送虚情怕费事一样。

邓俊辉

deng@tsinghua.edu.cn

Vector::deduplicate()

❖ template <typename T> int Vector<T>::deduplicate() {

```
int oldSize = _size;
```

```
Rank i = 1;
```

```
while ( i < _size )
```

```
    find( _elem[i], 0, i ) < 0 ? (a)
```

```
        i++
```

```
    : remove(i);
```

```
return oldSize - _size;
```

```
}
```

(b)



(a)



(c)



向量

无序向量：遍历

让他们每个人轮流到你的宝座下，同样诚恳地坦白他们的内心，
然后再看有没有一个人敢向你说：“我比这个人好。”

邓俊辉

deng@tsinghua.edu.cn

遍历

❖ 对向量中的每一元素，统一实施visit()操作 //如何指定visit()？如何将其传递到向量内部？

❖ template <typename T> //函数指针，只读或局部性修改

```
void Vector<T>::traverse( void ( * visit )( T & ) )  
{ for ( int i = 0; i < _size; i++ ) visit( _elem[i] ); }
```

❖ template <typename T> template <typename VST> //函数对象，全局性修改更便捷

```
void Vector<T>::traverse( VST & visit )  
{ for ( int i = 0; i < _size; i++ ) visit( _elem[i] ); }
```

实例

◆ 比如，为统一地将向量中所有元素分别加一，只需

- 实现一个可使单个T类型元素加一的类（结构）

```
template <typename T> //假设T可直接递增或已重载操作符“++”
```

```
struct Increase //函数对象：通过重载操作符“()”实现
```

```
{   virtual void operator()( T & e ) { e++; } }; //加一
```

- 将其作为参数传递给遍历算法

```
template <typename T> void increase( Vector<T> & V )
```

```
{   V.traverse( Increase<T>() ); } //即可以之作为基本操作，遍历向量
```

◆ 作为练习，可模仿此例，实现统一的减一、加倍、求和等遍历功能

向量

有序向量：唯一化

贾政道：“我要你另换个主意，不许雷同了前人，只做个破题也使得。”宝玉只得答应着，低头搜索枯肠。

邓俊辉

deng@tsinghua.edu.cn

有序性及其甄别

❖ 还记得起泡排序的原理？有序/无序序列中，任何/总有一对相邻元素顺序/逆序

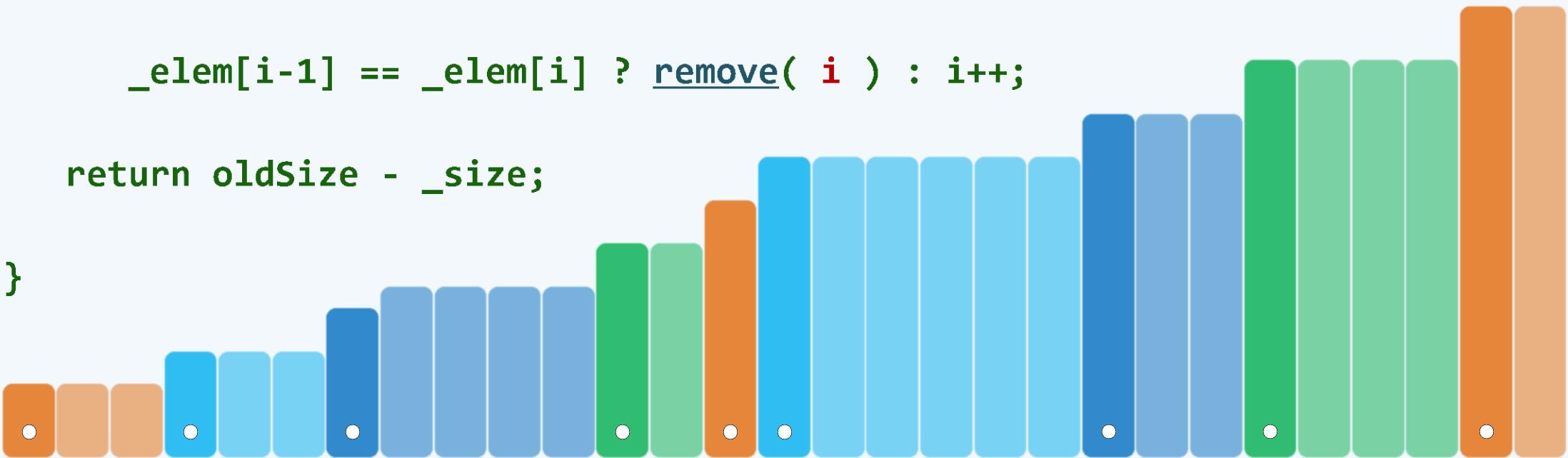
❖ 因此，相邻逆序对的数目，可在一定程度上度量向量的紊乱程度

```
template <typename T> int Vector<T>::disordered() const {  
    int n = 0; //相邻逆序对的计数器  
  
    for ( int i = 1; i < _size; i++ ) //逐一检查各对相邻元素  
        n += ( _elem[i-1] > _elem[i] ); //逆序则计数  
  
    return n; //向量有序，当且仅当n = 0  
}
```

❖ 无序向量经预处理转换为有序向量之后，相关算法多可优化

低效算法

```
❖ template <typename T> int Vector<T>::uniquify() {  
  
    int oldSize = _size; int i = 1;  
  
    while ( i < _size )  
  
        _elem[i-1] == _elem[i] ? remove( i ) : i++;  
  
    return oldSize - _size;  
}
```

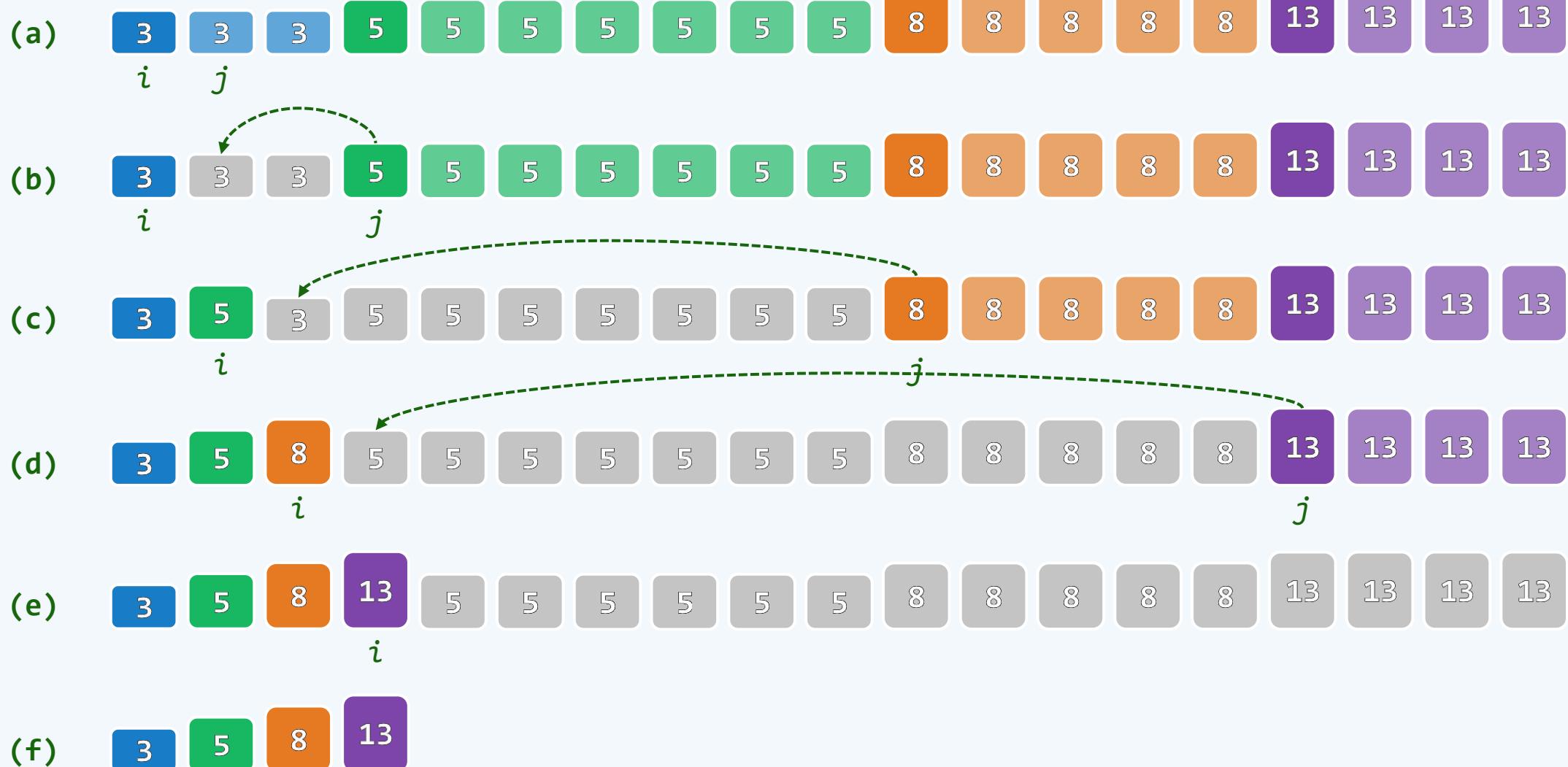


高效算法

```
❖ template <typename T> int Vector<T>::uniquify() {  
    Rank i = 0, j = 0;  
  
    while ( ++j < _size )  
  
        if ( _elem[ i ] != _elem[ j ] ) _elem[ ++i ] = _elem[ j ];  
  
    _size = ++i; shrink();  
  
    return j - i;  
}
```



实例



向量

有序向量：二分查找（版本A）

自从爷爷去后，这山被二郎菩萨点上火，烧杀了大半。我们蹲在井里，钻在涧内，藏于铁板桥下，得了性命。及至火灭烟消，出来时，又没花果养赡，难以存活，别处又去了一半。我们这一半，捱苦的住在山中，这两年，又被些打猎的抢了一半去也。

邓俊辉

deng@tsinghua.edu.cn

统一接口

❖ template <typename T> //查找算法统一接口, $0 \leq lo < hi \leq _size$

Rank Vector<T>::search(T const & e, Rank lo, Rank hi) const {

return (rand() % 2) ? //按各50%的概率随机选用

binSearch(_elem, e, lo, hi) //二分查找算法, 或者

: fibSearch(_elem, e, lo, hi); //Fibonacci查找算法

}



0

lo

hi

$_size$

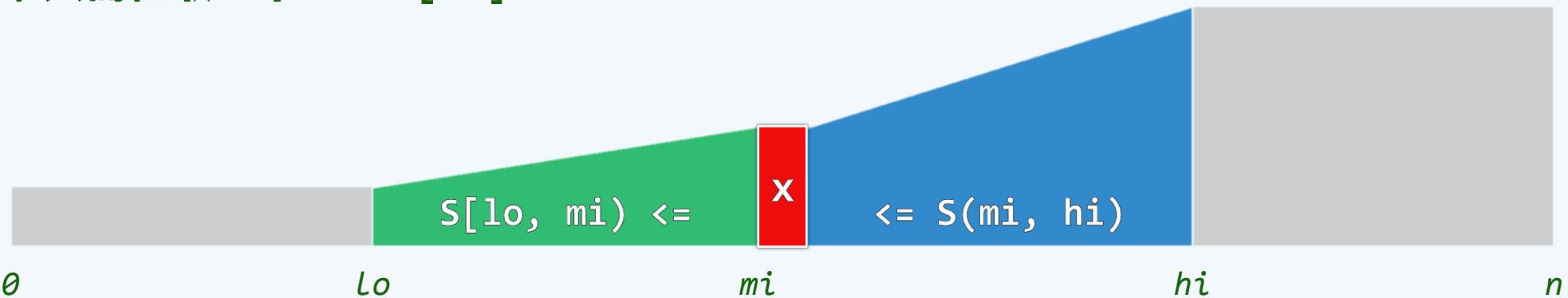
轴点

❖ 有序向量中，处处皆为轴点 //说人话！

❖ 以任一元素 $s[mi]$ 为界

- 都可将待查找区间 $[lo, hi)$ 分为三部分，且
- $s[lo, mi] \leq s[mi] \leq s(mi, hi)$

❖ 因此，只需任取一个 $x = s[mi]$



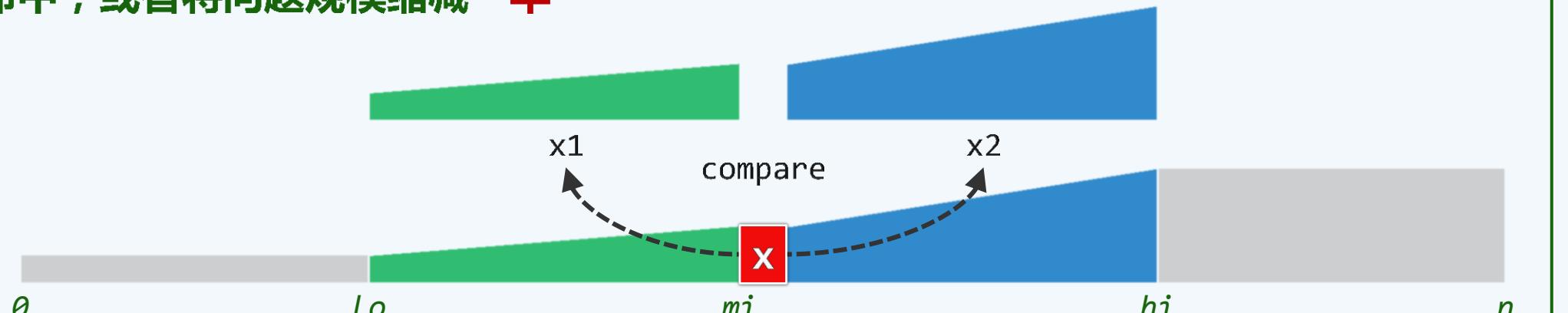
减而治之

❖ 只需将目标元素e与x做一比较，即可分三种情况进一步处理：

- $e < x$ ：则e若存在必属于**左侧子区间**，故可（减除 $s[mi, hi]$ 并）递归深入 $s[lo, mi]$
- $x < e$ ：则e若存在必属于**右侧子区间**，亦可（减除 $s[lo, mi]$ 并）递归深入 $s(mi, hi)$
- $e = x$ ：已在此处命中，可随即返回 //若有多处，返回何者？

❖ 若轴点mi取作中点，则每经过**至多两次比较**

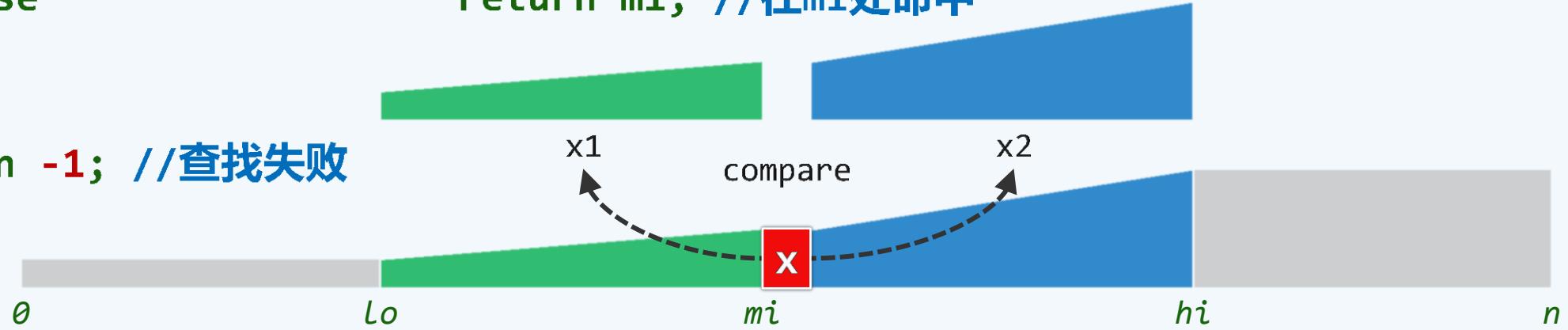
或者能够命中，或者将问题规模缩减一半



实现

❖ template <typename T> //在有序向量区间[lo, hi)内查找元素e

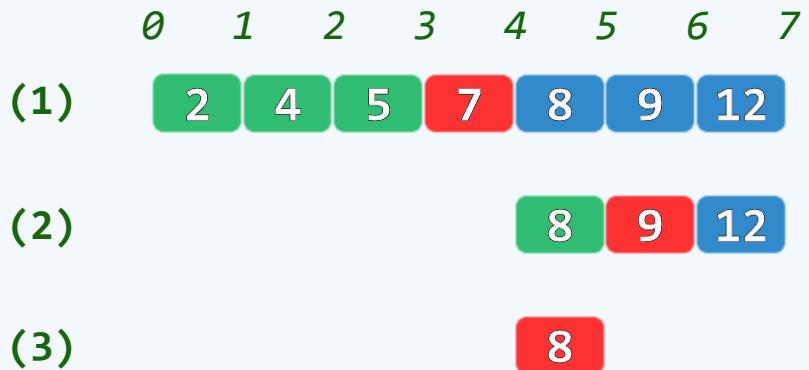
```
static Rank binSearch( T * S, T const & e, Rank lo, Rank hi ) {  
    while ( lo < hi ) { //每步迭代可能要做两次比较判断，有三个分支  
        Rank mi = ( lo + hi ) >> 1; //以中点为轴点（区间宽度的折半，等效于其数值表示的右端点减去左端点的一半）  
        if ( e < S[mi] ) hi = mi; //深入前半段[lo, mi)继续查找  
        else if ( S[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)  
        else return mi; //在mi处命中  
    }  
    return -1; //查找失败  
}
```



实例 + 复杂度

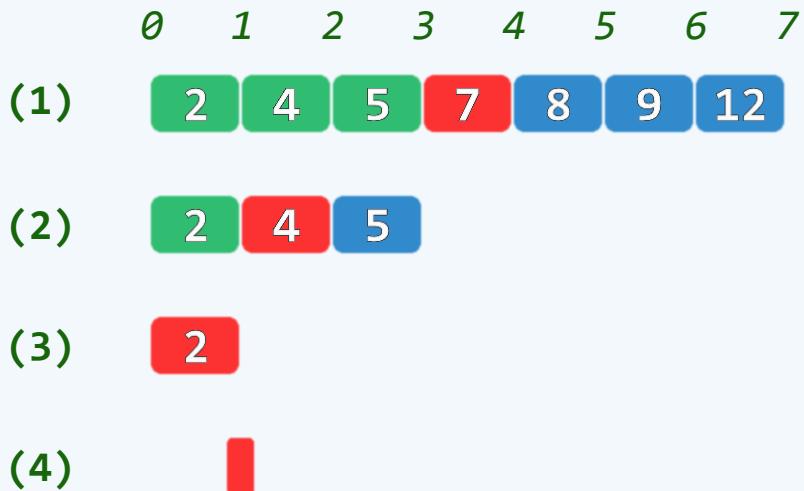
❖ `S.search(8, 0, 7)` :

经 $2 + 1 + 2 = 5$ 次比较，在 $s[4]$ 命中



`S.search(3, 0, 7)` :

经 $1 + 1 + 2 = 4$ 次比较，在 $s[1]$ 失败

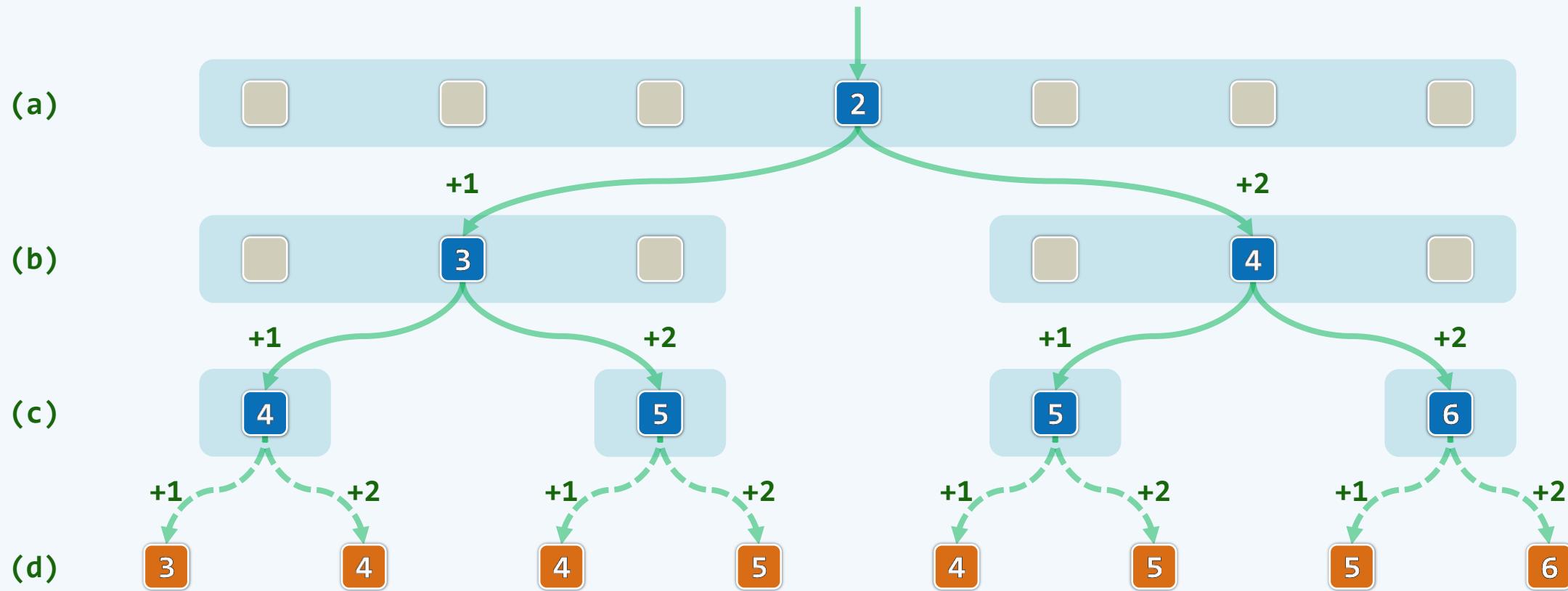


❖ 线性递归： $T(n) = T(n/2) + \mathcal{O}(1) = \mathcal{O}(\log n)$ ，大大优于顺序查找

递归跟踪：轴点总能取到中点，递归深度 $\mathcal{O}(\log n)$ ；各递归实例仅耗时 $\mathcal{O}(1)$

查找长度

- ❖ 如何更精细地评估查找算法的性能？考查关键码的比较次数（search length）
- ❖ 通常，需分别针对成功与失败查找，从最好、最坏、平均等角度评估
- ❖ 比如，成功、失败时的平均查找长度均大致为 $\mathcal{O}(1.50 \cdot \log n)$ //详见教材、习题解析



查找长度

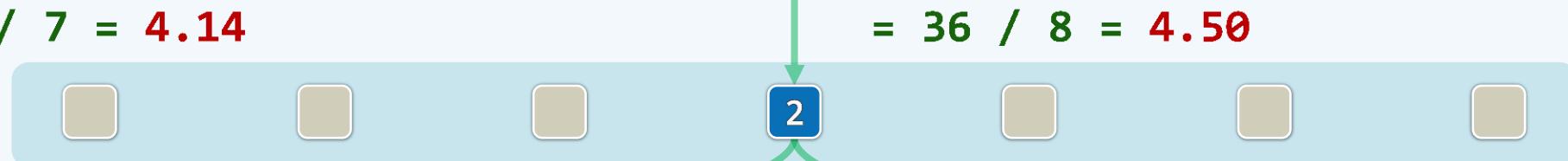
❖ $n = 7$ 时，各元素对应的成功查找长度为

{ 4, 3, 5, 2, 5, 4, 6 }

在等概率情况下，平均成功查找长度

$$= 29 / 7 = 4.14$$

(a)



(b)



(c)



(d)

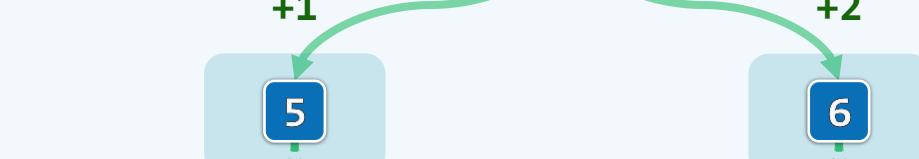


❖ 共8种失败情况，查找长度分别为

{ 3, 4, 4, 5, 4, 5, 5, 6 }

在等概率情况下，平均失败查找长度

$$= 36 / 8 = 4.50$$



向量

有序向量：Fibonacci查找

常伟思微微一笑说：这个比例很奇怪，是吗？

他又想来想去，又想不出好地方，于是终于决心，假定这“幸福的家庭”所在的地方叫做A。

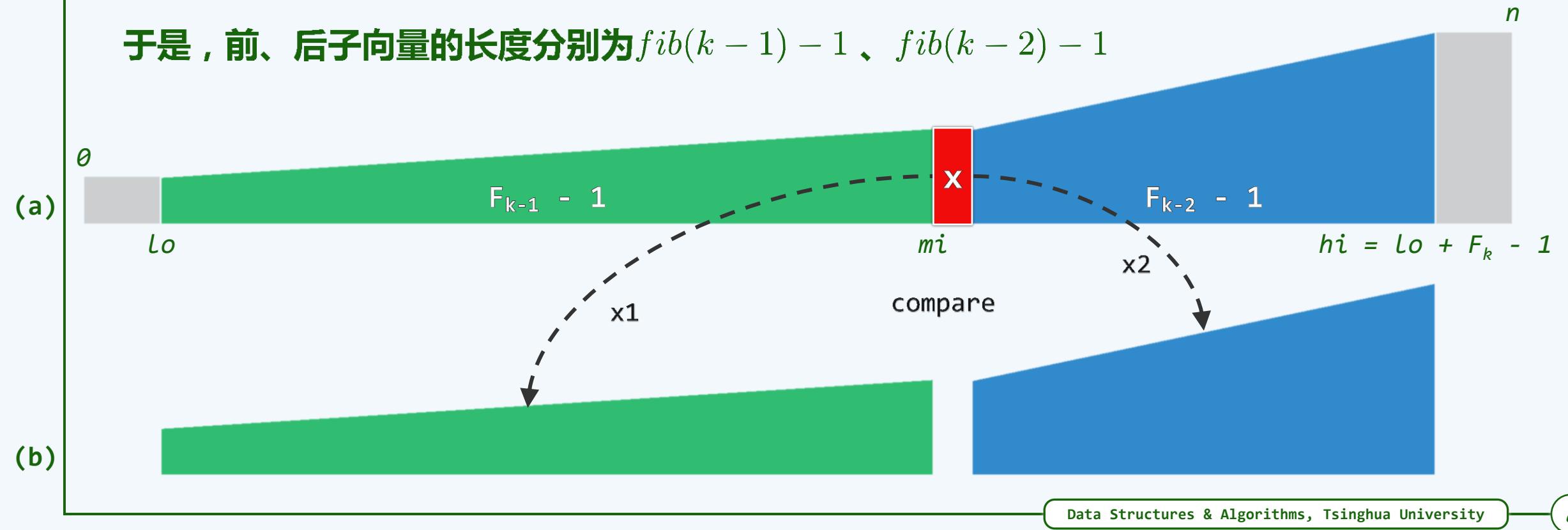
邓俊辉

deng@tsinghua.edu.cn

思路及原理

- ❖ 版本A：转向左、右分支前的关键码**比较次数不等**，而**递归深度却相同**
- ❖ 通过**递归深度的不均衡对转向成本的不均衡做补偿**，平均查找长度应能进一步缩短！
- ❖ 比如，若有 $n = fib(k) - 1$ ，则可取 $mi = fib(k - 1) - 1$

于是，前、后子向量的长度分别为 $fib(k - 1) - 1$ 、 $fib(k - 2) - 1$

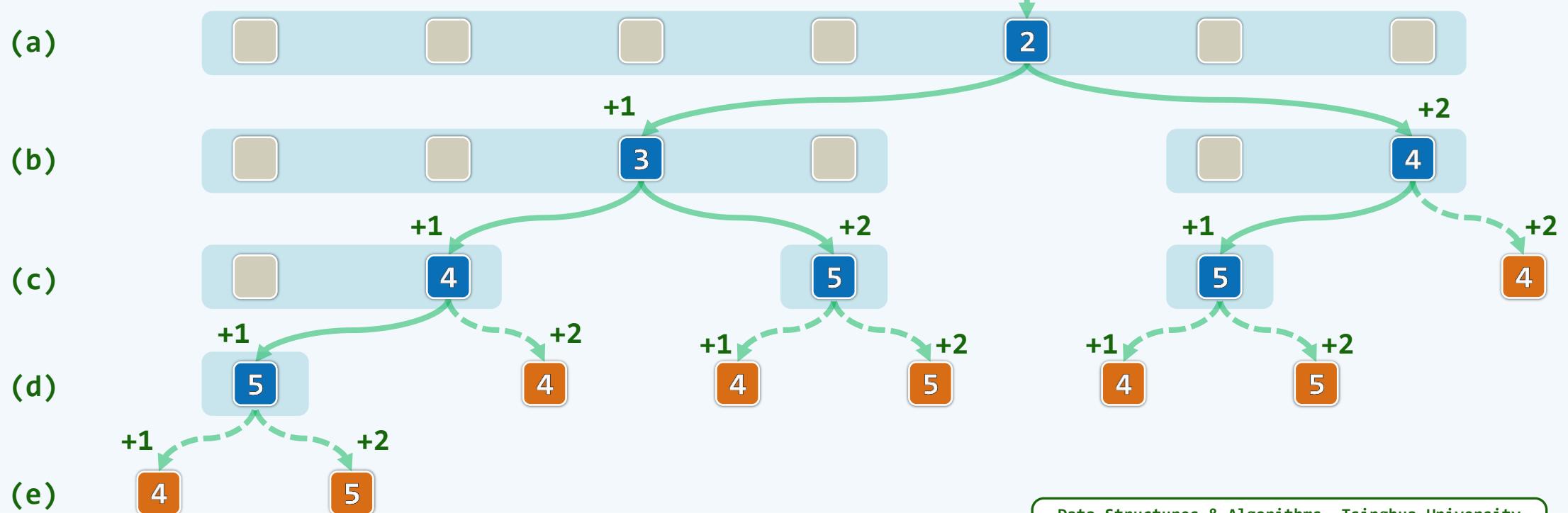


实现

```
❖ template <typename T> //0 <= lo <= hi <= _size  
static Rank fibSearch( T * S, T const & e, Rank lo, Rank hi ) {  
    for ( Fib fib(hi - lo); lo < hi; ) { //Fib数列制表备查  
        while ( hi - lo < fib.get() ) fib.prev(); //自后向前顺序查找轴点(分摊O(1))  
        Rank mi = lo + fib.get() - 1; //确定形如Fib(k) - 1的轴点  
        if ( e < S[mi] ) hi = mi; //深入前半段[lo, mi)继续查找  
        else if ( S[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)  
        else return mi; //在mi处命中  
    }  
    return -1; //查找失败  
}
```

查找长度

- ❖ Fibonacci 查找的ASL , (在常系数的意义上) 优于二分查找 //详见教材、习题解析
- ❖ 仍以 $n = \text{fib}(6) - 1 = 7$ 为例，在等概率情况下
 - 平均成功查找长度 = $(5 + 4 + 3 + 5 + 2 + 5 + 4) / 7 = 28/7 = 4.00$
 - 平均失败查找长度 = $(4 + 5 + 4 + 4 + 5 + 4 + 5 + 4) / 8 = 35 / 8 = 4.38$



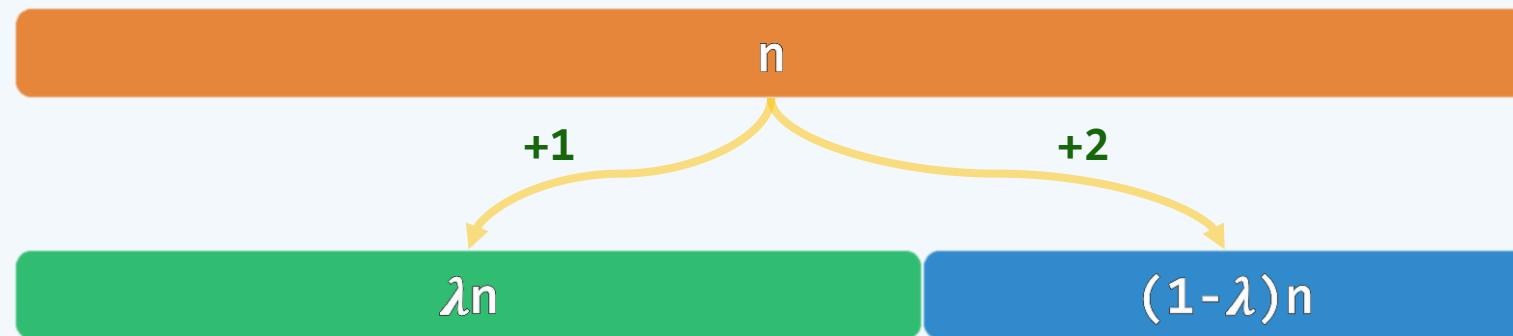
通用策略

❖ 在任何区间 $[0, n)$ 内，总是选取 $[\lambda \cdot n]$ 作为轴点， $0 \leq \lambda < 1$

比如：二分查找对应于 $\lambda = 0.5$ ，Fibonacci 查找对应于 $\lambda = \phi = 0.6180339\dots$

❖ 这类查找算法的渐进复杂度为 $\alpha(\lambda) \cdot \log_2 n = \mathcal{O}(\log n)$

❖ (在常系数意义上的) 性能最优，即意味着 $\alpha(\lambda)$ 达到最小

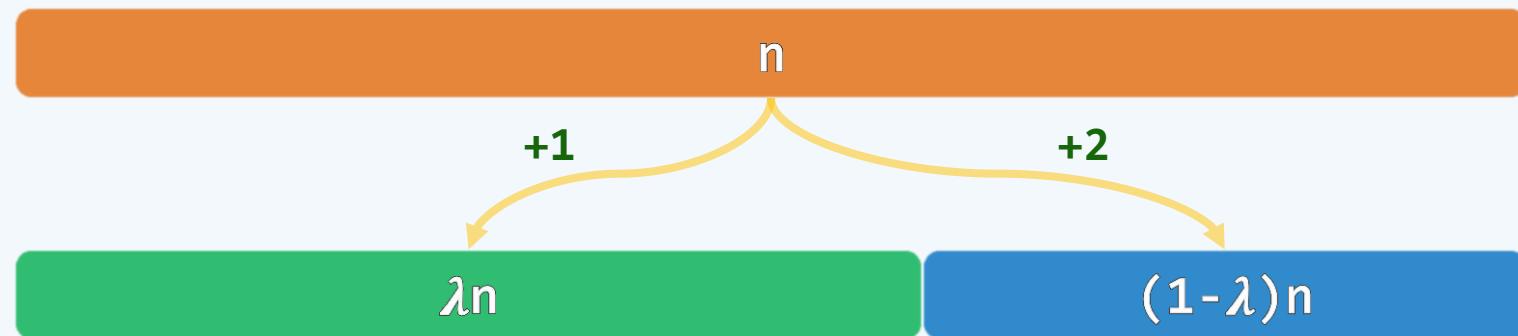


$$\phi = 0.6180339\dots$$

❖ 递推式 : $\alpha(\lambda) \cdot \log_2 n = \lambda \cdot [1 + \alpha(\lambda) \cdot \log_2 (\lambda n)] + (1 - \lambda) \cdot [2 + \alpha(\lambda) \cdot \log_2 ((1 - \lambda)n)]$

❖ 整理后 :
$$\frac{-\ln 2}{\alpha(\lambda)} = \frac{\lambda \cdot \ln \lambda + (1 - \lambda) \cdot \ln(1 - \lambda)}{2 - \lambda}$$

❖ 当 $\lambda = \phi = (\sqrt{5} - 1)/2$ 时 , $\alpha(\lambda) = 1.440420\dots$ 达到最小



向量

有序向量：二分查找（版本B）

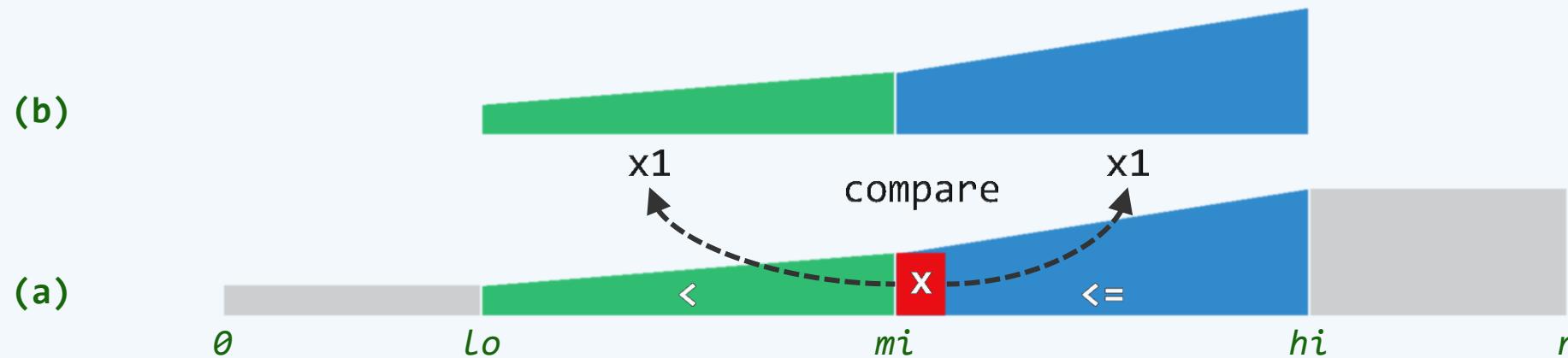
和微风匀到一起的光，象冰凉的刀刃儿似的，把宽静的大街切成两半，一半儿黑，一半儿亮。那黑的一半，使人感到阴森，亮的一半使人感到凄凉。

邓俊辉

deng@tsinghua.edu.cn

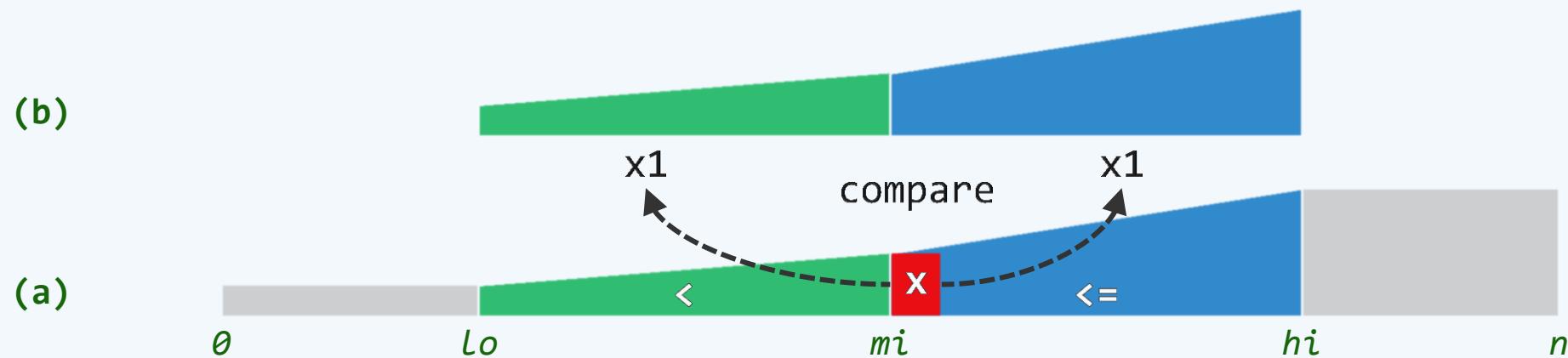
改进思路

- ❖ 二分查找中左、右分支转向代价不平衡的问题，也可直接解决
 - ❖ 比如，每次迭代仅做1次关键码比较；如此，所有分支只有2个方向，而不再是3个
 - ❖ 同样地，轴点 mi 取作中点，则查找每深入一层，问题规模也缩减一半
 - $e < x$ ：则 e 若存在必属于左侧子区间 $S[lo, mi)$ ，故可递归深入
 - $x \leq e$ ：则 e 若存在必属于右侧子区间 $S[mi, hi)$ ，亦可递归深入
- 直到 $hi - lo = 1$ ，才判断是否命中



实现

```
❖ template <typename T> static Rank binSearch( T * S, T const & e, Rank lo, Rank hi ) {  
    while ( 1 < hi - lo ) { //有效查找区间的宽度缩短至1时，算法才终止  
        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入  
        e < S[mi] ? hi = mi : lo = mi; // [lo, mi) 或 [mi, hi)  
    } //出口时 hi = lo + 1，查找区间仅含一个元素 A[lo]  
    return e == S[lo] ? lo : -1; //返回命中元素的秩，或者（失败时）非法的秩  
} //相对于版本A，最好（坏）情况下更坏（好）；各种情况下的SL更加接近，整体性能更趋稳定
```



语义约定

❖ 各种特殊情况，如何统一地处置？比如

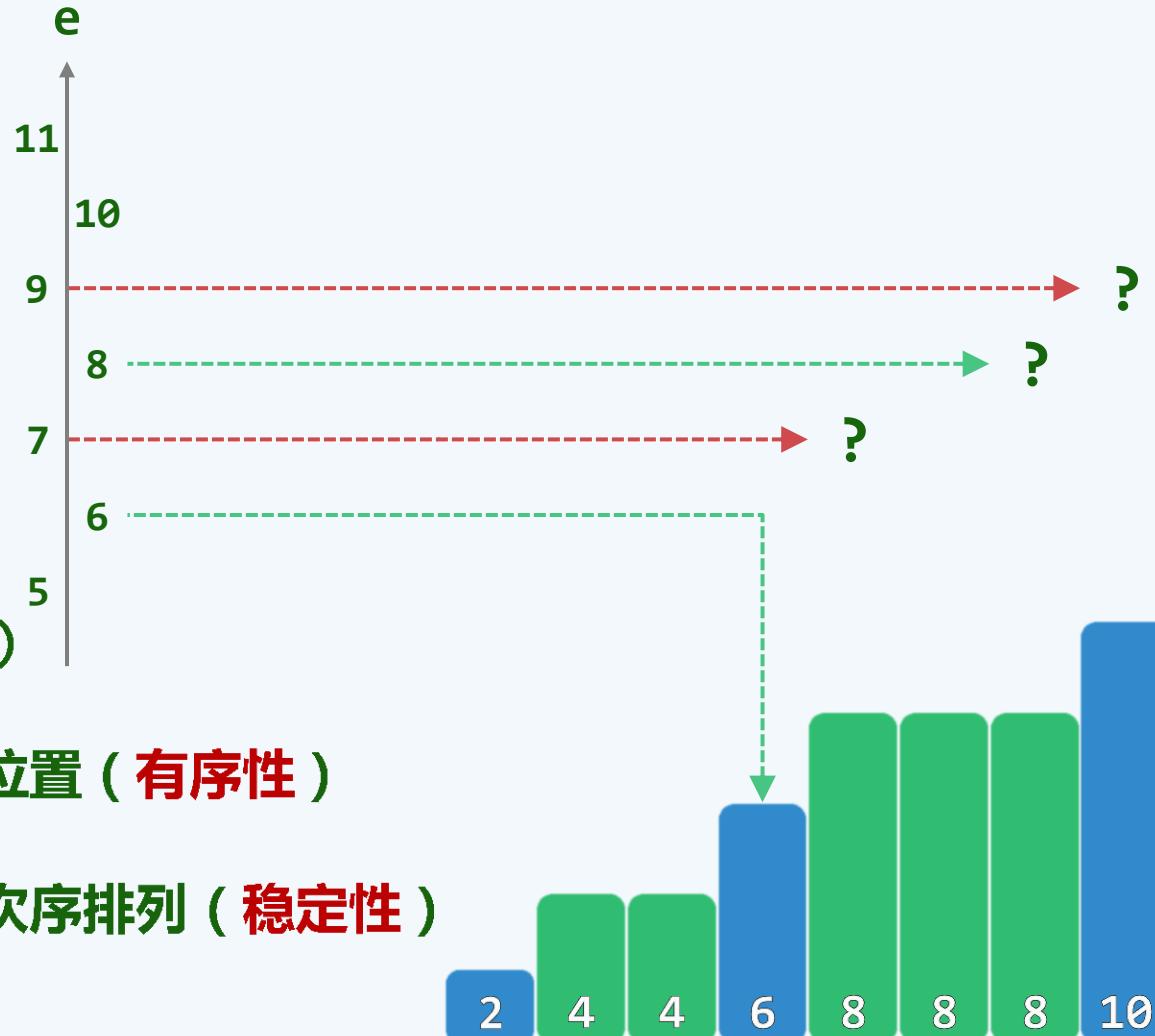
- 目标元素不存在；或反过来
- 目标元素同时存在多个

❖ 有序向量自身，如何便捷地维护？

比如：`V.insert(1 + V.search(e), e)`

- 即便失败，也应给出新元素适当的插入位置（有序性）
- 若有重复元素，每一组也需按其插入的次序排列（稳定性）

❖ 为此，需要更为精细、明确、简捷地定义search()的返回值



语义约定

◆ 约定：search()总是返回

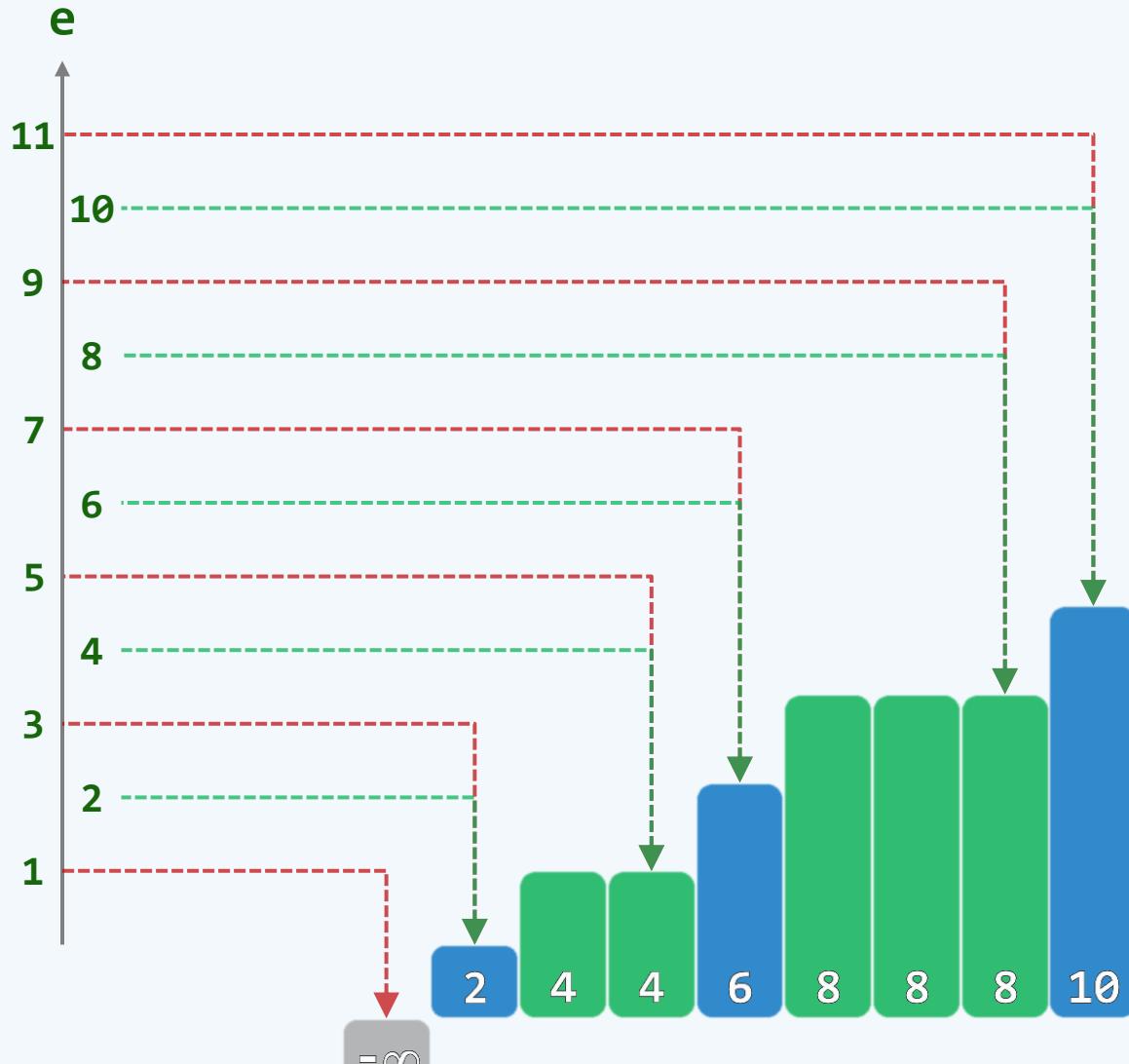
- $m = \text{不大于} e \text{的最后一个元素}$
- (其后继 $M = \text{大于} e \text{的第一个元素})$

◆ 改进版本B：

```
return e == S[lo] ? lo : -1 ;  
  
return e < S[lo] ? lo-1 : lo ;
```

◆ 课后：相应地改进fibSearch()

◆ 有没有更为简明的实现方式？



向量

有序向量：二分查找（版本C）

邓俊辉

deng@tsinghua.edu.cn

Outward failure may be a manifested variant of inward success.

实现

```
❖ template <typename T> static Rank binSearch( T * S, T const & e, Rank lo, Rank hi ) {  
    while ( lo < hi ) { //不变性 : A[0, lo) <= e < A[hi, n)  
        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入  
        e < S[mi] ? hi = mi : lo = mi + 1; // [lo, mi) 或 (mi, hi)  
    } //出口时，必有 S[lo = hi] = m  
    return lo - 1; //故，S[lo-1] = m  
} //留意与版本B的差异...
```

❖ 待查找区间宽度缩短至0而非1时，算法才结束 //lo == hi

转入右侧子向量时，左边界取作mi+1而非mi //A[mi]会被遗漏？

无论成功与否，返回的秩严格符合接口的语义约定... //如何证明其正确性？

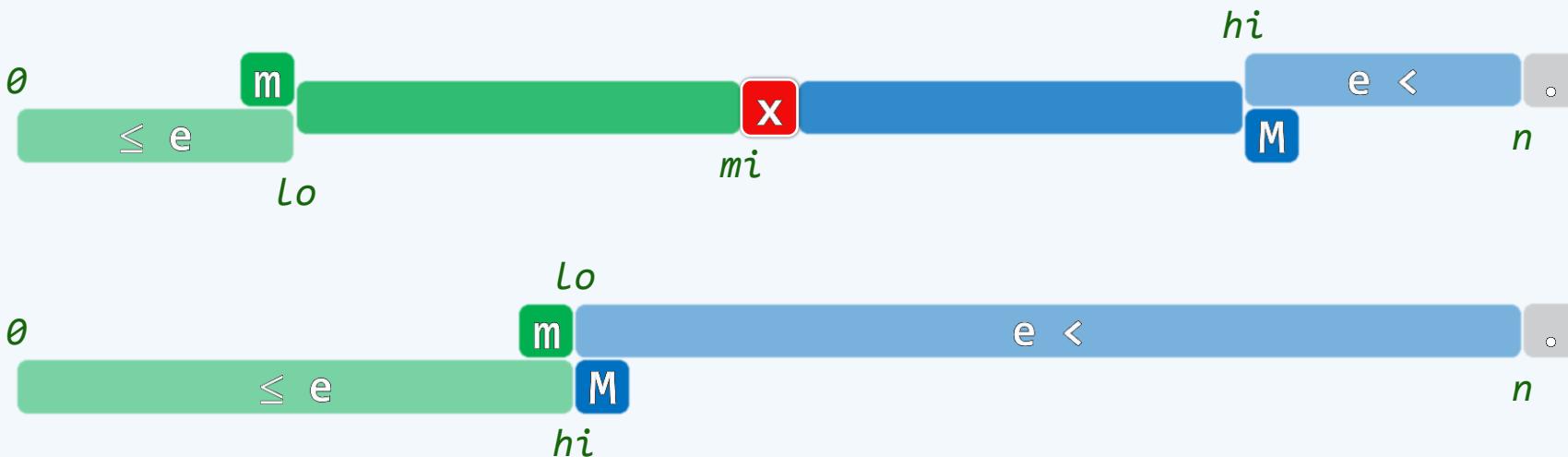
不变性 : $A[0, lo) \leq e < A[hi, n)$

❖ 在算法执行过程中的任意时刻

- $A[lo-1]$ 总是 (截至当前已确认的) 不大于 e 的最大者 (m)
- $A[hi]$ 总是 (截至当前已确认的) 大于 e 的最小者 (M)

❖ 当算法终止时

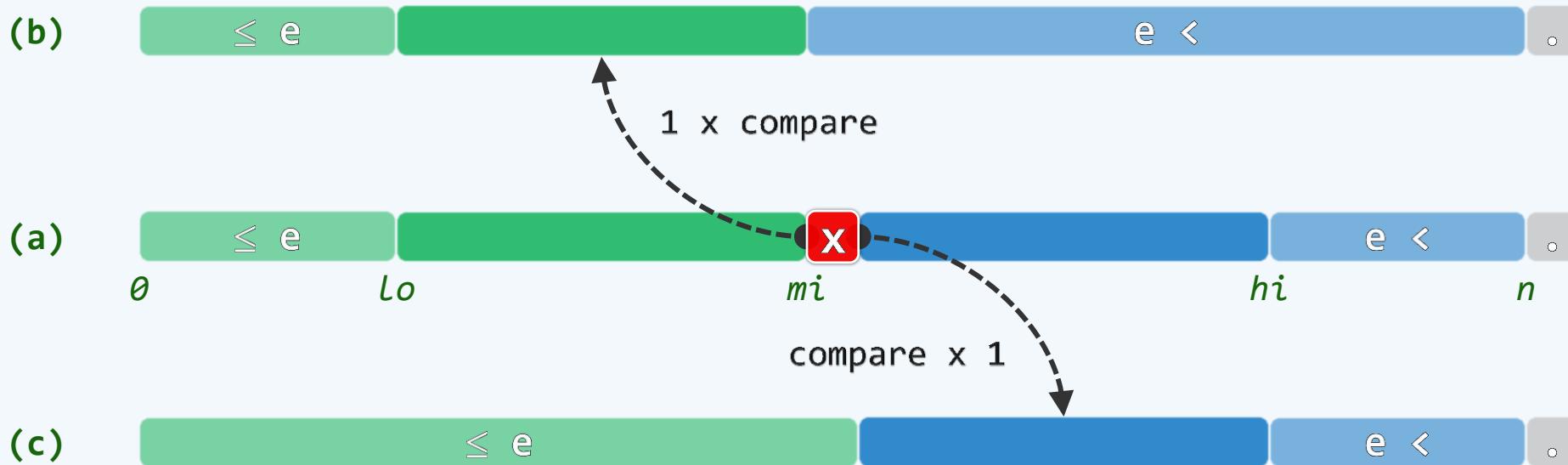
- $A[lo-1] = A[hi-1]$ 即是 (全局) 不大于 e 的最大者



不变性 : $A[0, lo] \leq e < A[hi, n]$

♦ 初始时， $lo = 0$ 且 $hi = n$, $A[0, lo] = A[hi, n] = \emptyset$, 自然成立

♦ 数学归纳：假设不变性一直保持至(a), 以下无非两种情况...



向量

有序向量：插值查找

邓俊辉

deng@tsinghua.edu.cn

...那么在最后剩下的一万个猎手中，肯定有人会做出这样的选择：向那个位置开一枪试试...

原理与算法

❖ 假设：已知有序向量中各元素随机分布的规律

比如：独立且均匀的随机分布

❖ 于是： $[lo, hi]$ 内各元素应大致呈线性趋势增长

$$\frac{mi - lo}{hi - lo} \approx \frac{e - A[lo]}{A[hi] - A[lo]}$$

❖ 因此：通过猜测轴点 mi ，可以极大地提高收敛速度

$$mi \approx lo + (hi - lo) \cdot \frac{e - A[lo]}{A[hi] - A[lo]}$$

❖ 以英文词典为例：**binary**大致位于 $2/26$ 处

search大致位于 $19/26$ 处

[lo]	0	A	1	[1, 53)
1	B		74	[53, 104)
2	C		158	[104, 156)
3	D		292	[156, 208)
4	E		368	[208, 259)
5	F		409	[259, 311)
6	G		473	[311, 363)
7	H		516	[363, 414)
8	I		562	[414, 466)
9	J		607	[466, 518)
10	K		617	[518, 569)
11	L		628	[569, 621)
12	M		681	[621, 673)
13	N		748	[673, 724)
14	O		771	[724, 776)
15	P		806	[776, 827)
16	Q		915	[827, 879)
17	R		922	[879, 931)
18	S		1002	[931, 982)
19	T		1176	[982, 1034)
20	U		1253	[1034, 1086)
21	V		1271	[1086, 1137)
22	W		1289	[1137, 1189)
23	X		1337	[1189, 1241)
24	Y		1338	[1241, 1292)
25	Z		1341	[1292, 1344)
[hi]	26		1344	

实例

❖ 查找目标 : $e = 50$

❖ $lo = 0, hi = 18$



$$\text{插值: } mi = 0 + (18 - 0) * (50 - 5) / (92 - 5) = 9$$

比较: $A[9] = 41 < e$

❖ $lo = 10, hi = 18$



$$\text{插值: } mi = 10 + (18 - 10) * (50 - 44) / (92 - 44) = 11$$

比较: $A[11] = 51 > e$

❖ $lo = hi = 10$



插值: $mi = 10$

比较: $A[10] = 49 < e$, 故返回: NOT_FOUND

性能

◆ 最坏： $hi - lo = \mathcal{O}(n)$

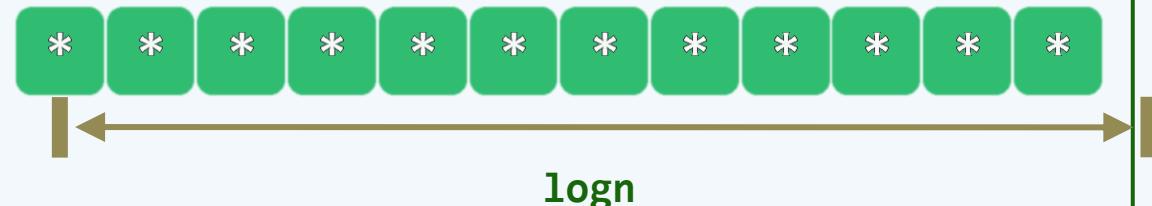
//具体实例？

◆ 平均：每经一次比较，待查找区间宽度由 n 缩至 \sqrt{n} // [Yao76, PIA78]，习题解析[2-24]

$$n \rightarrow \sqrt{n} \rightarrow \sqrt{\sqrt{n}} \rightarrow \sqrt{\sqrt{\sqrt{n}}} \rightarrow \dots \rightarrow 2$$

$$n \rightarrow n^{1/2^1} \rightarrow n^{1/2^2} \rightarrow n^{1/2^3} \rightarrow \dots \rightarrow 2$$

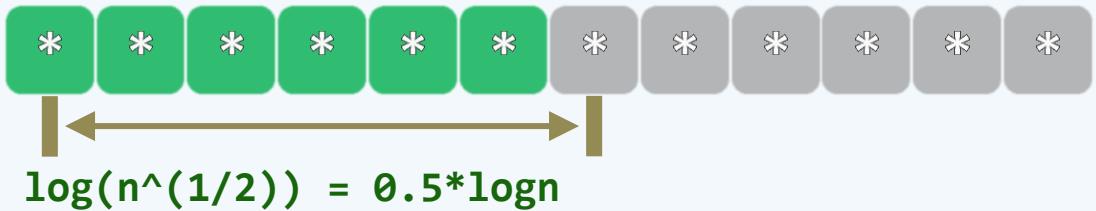
$\mathcal{O}(\log \log n)$



◆ 每经一次比较

待查找区间宽度的数值 n 开方，有效字长 $\log n$ 减半

- 差值查找 = 在字长意义上的折半查找
- 二分查找 = 在字长意义上的顺序查找



综合评价

❖ 从 $\Theta(\log n)$ 到 $\Theta(\log \log n)$ ，优势并不明显

(除非查找表极长，或比较操作成本极高)

比如， $n = 2^{(2^5)} = 2^{32} = 4G$ 时

- $\log_2(n) = 32$

- $\log_2(\log_2(n)) = 5$

❖ 须引入乘法、除法运算

❖ 易受小扰动的干扰和“蒙骗”

❖ 实际可行的方法

- 首先通过插值查找

迅速将查找范围缩小到一定的尺度

- 然后再进行二分查找

进一步缩小范围

- 最后（当数据项只有200~300时）

使用顺序查找

向量

起泡排序

邓俊辉

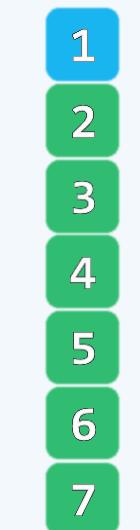
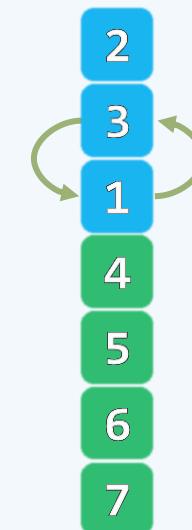
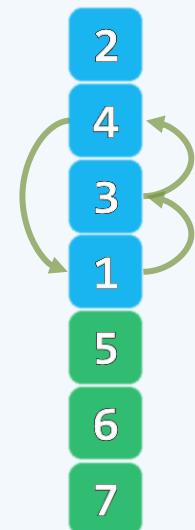
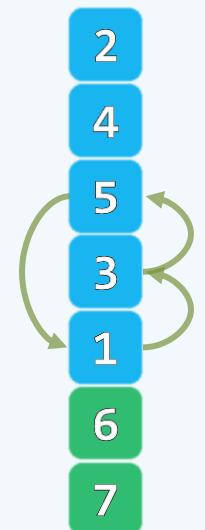
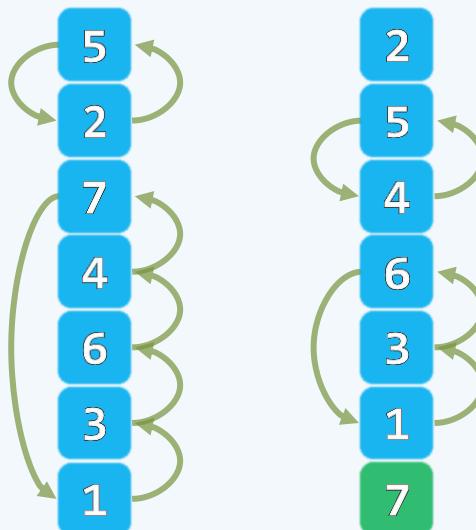
deng@tsinghua.edu.cn

排序器：统一入口

```
❖ template <typename T> void Vector<T>::sort( Rank lo, Rank hi ) {  
    switch ( rand() % 6 ) {  
        case 1: bubbleSort( lo, hi ); break; //起泡排序  
        case 2: selectionSort( lo, hi ); break; //选择排序(习题)  
        case 3: mergeSort( lo, hi ); break; //归并排序  
        case 4: heapSort( lo, hi ); break; //堆排序(第12章)  
        case 5: quickSort( lo, hi ); break; //快速排序(第14章)  
        default: shellSort( lo, hi ); break; //希尔排序(第14章)  
    } //随机选择算法以充分测试。实用时可视具体问题的特点灵活确定或扩充  
} //向量区间[lo, hi)排序
```

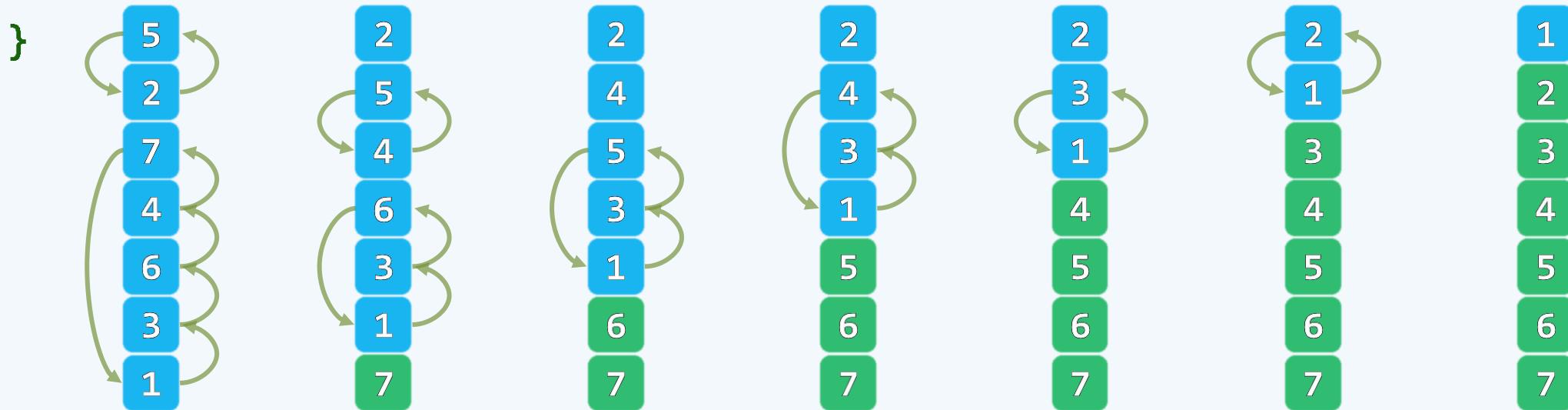
构思

- ◆ 问题：给定 n 个可比较的元素，将它们按（非降）序排列
- ◆ 观察：有序/无序序列中，任何/总有一对相邻元素顺序/逆序
- ◆ 扫描交换：依次比较每一对相邻元素；如有必要，交换之
- ◆ 若整趟扫描都没有进行交换，则排序完成；否则，再做一趟扫描交换



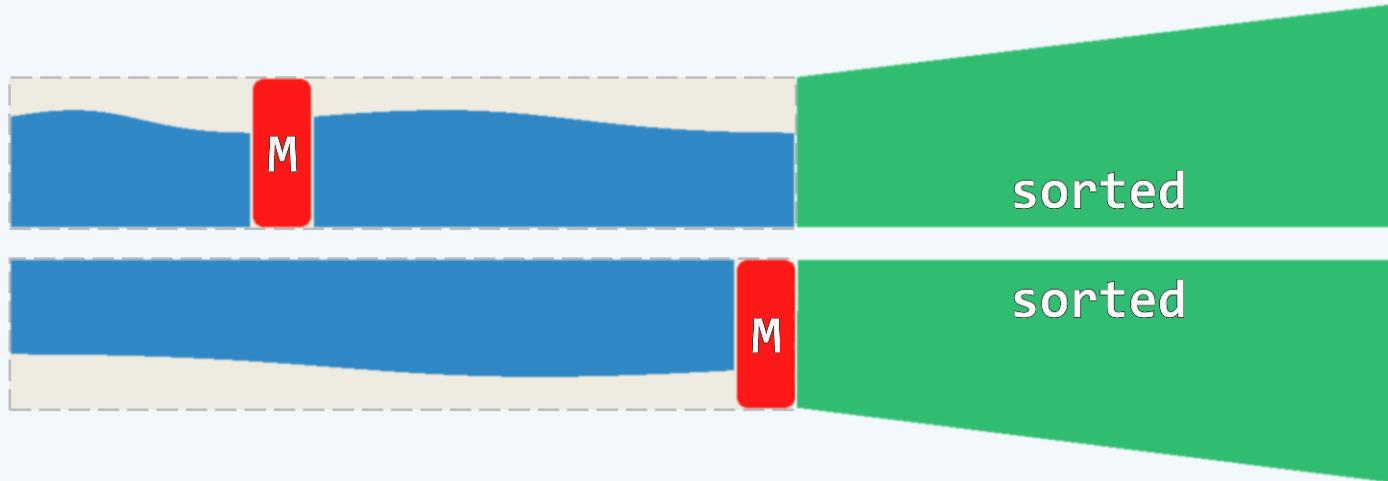
基本版

```
❖ template <typename T> Vector<T>::bubbleSort( Rank lo, Rank hi ) {  
    while( lo < --hi ) //逐趟起泡扫描 ( 输入保证 : 0 <= lo < hi <= size )  
        for( Rank i = lo; i < hi; i++ ) //若相邻元素  
            if( _elem[i] > _elem[i + 1] ) //逆序  
                swap( _elem[i], _elem[i + 1] ); //则交换
```

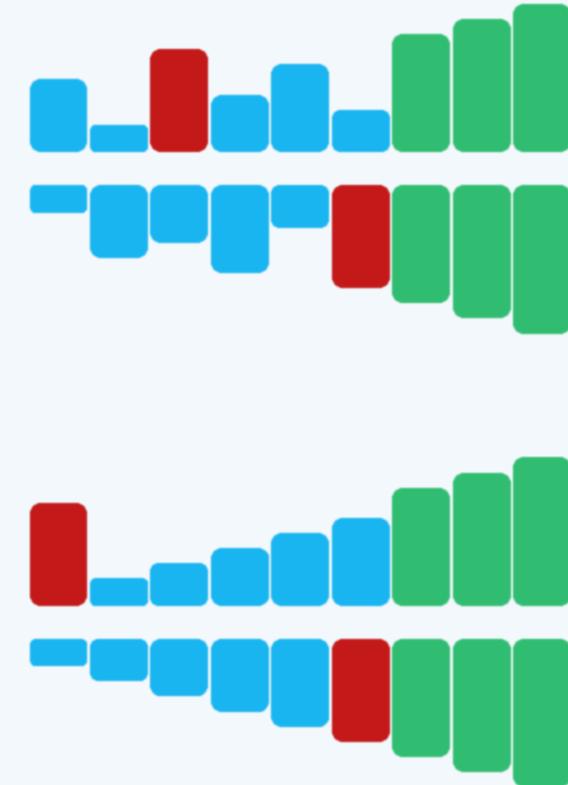


正确性

- ❖ 不变性：经 k 趟扫描交换后，最大的 k 个元素必然就位
- ❖ 单调性：经 k 趟扫描交换后，问题规模缩减至 $n-k$
- ❖ 正确性：经至多 n 趟扫描后，算法必然终止，且能给出正确解答

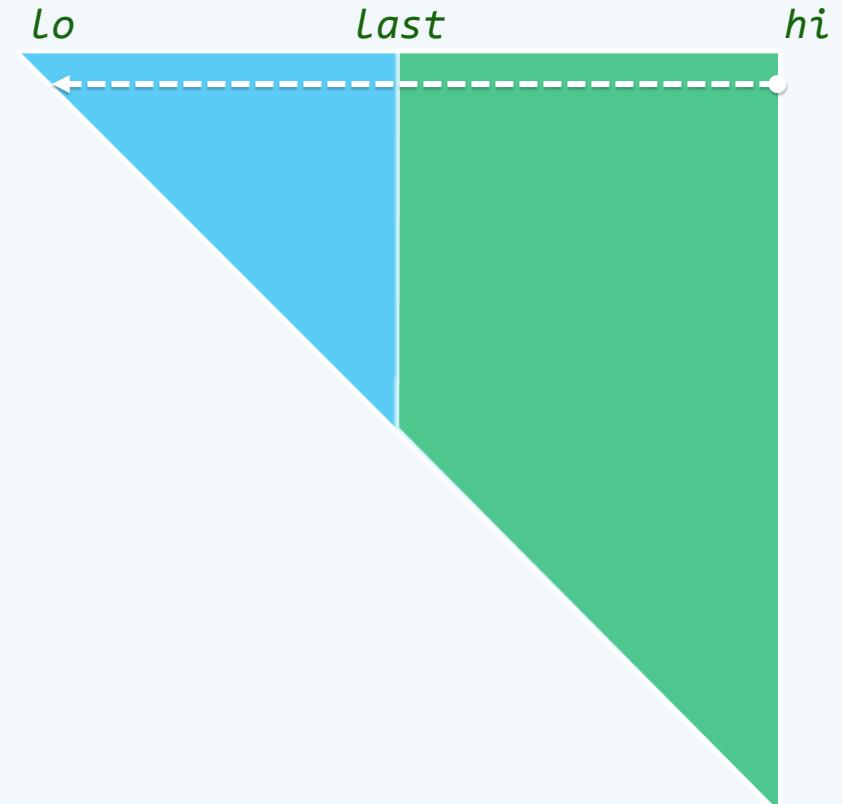


- ❖ $n-1$ 趟起泡扫描一定足够，但往往不必，比如...
- ❖ $[hi]$ 就位后， $[lo, hi]$ 可能已经有序（sorted）——此时，应该可以...



提前终止版

```
❖ template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi ) {  
    for( bool sorted = false; sorted = !sorted; )  
        for( Rank i = lo; i < hi - 1; i++ )  
            if( _elem[i] > _elem[i + 1] ) {  
                swap( _elem[i], _elem[i+1] );  
                sorted = false; //仍未完全有序  
            } //else ... 提前终止  
    }
```



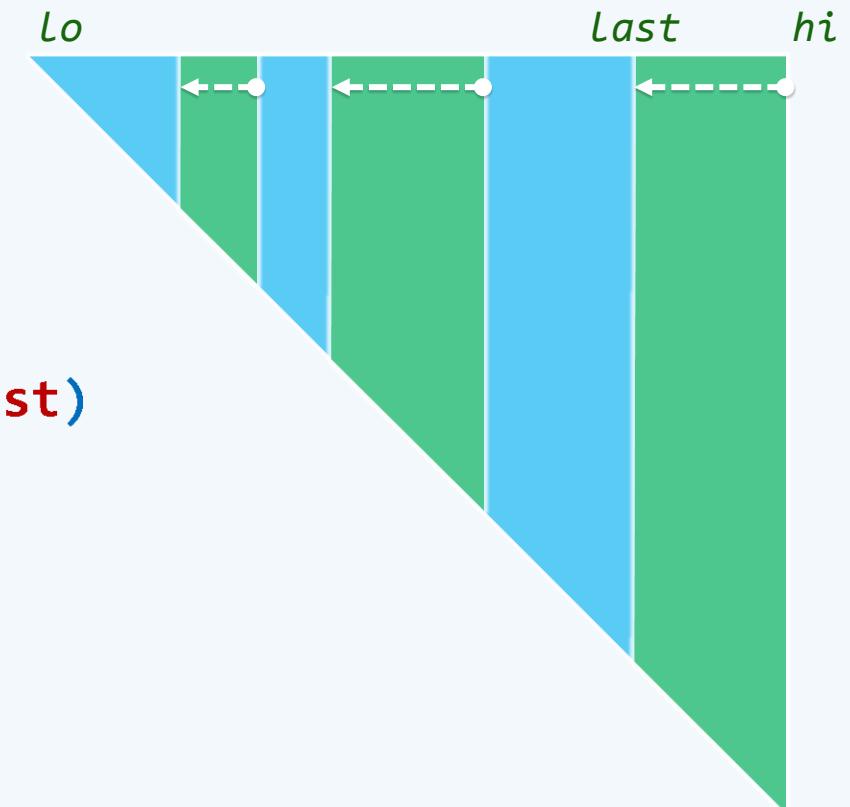
❖ 有改进，但仍有继续改进的余地，比如...

❖ [hi]就位后，尽管[lo,hi)未必有序，但某后缀[last,hi)可能有序——此时，应该可以...

跳跃版

```
❖ template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi ) {  
    for( Rank last = --hi; lo < hi; hi = last )  
        for( Rank i = last = lo; i < hi; i++ )  
            if( _elem[i] > _elem[i + 1] ) {  
                swap( _elem[i], _elem[i+1] );  
                last = i; //逆序对只可能残留于[lo, last)  
            }  
}
```

```
❖ A[lo, last) <= A[last, hi)  
A[lo, last] < A(last, hi)
```



综合评价

- ❖ 时间效率：最好 $\theta(n)$ ，最坏 $\theta(n^2)$
- ❖ 输入含重复元素时，算法的**稳定性** (stability) 是更为细致的要求
重复元素在输入、输出序列中的相对次序，是否保持不变？
 - 输入： 6, **7a**, 3, 2, **7b**, 1, 5, 8, **7c**, 4
 - 输出：
 - 1, 2, 3, 4, 5, 6, **7a**, **7b**, **7c**, 8 //stable
 - 1, 2, 3, 4, 5, 6, **7a**, **7c**, **7b**, 8 //unstable
- ❖ 以上起泡排序算法是稳定的吗？是的！
- ❖ 在起泡排序中，元素a和b的相对位置发生变化，只有一种可能：
 - 经分别与其它元素的交换，二者相互接近直至相邻
 - 在接下来一轮扫描交换中，二者因**逆序**而交换位置
- ❖ 在if一句的判断条件中，若把"**>**"换成"**>=**"，将有何变化？

向量

归并排序：分而治之

几曾随逝水，岂必委芳尘
万缕千丝终不改，任他随聚随分

邓俊辉

deng@tsinghua.edu.cn

原理

❖ //分治策略

//向量与列表通用

//J. von Neumann, 1945

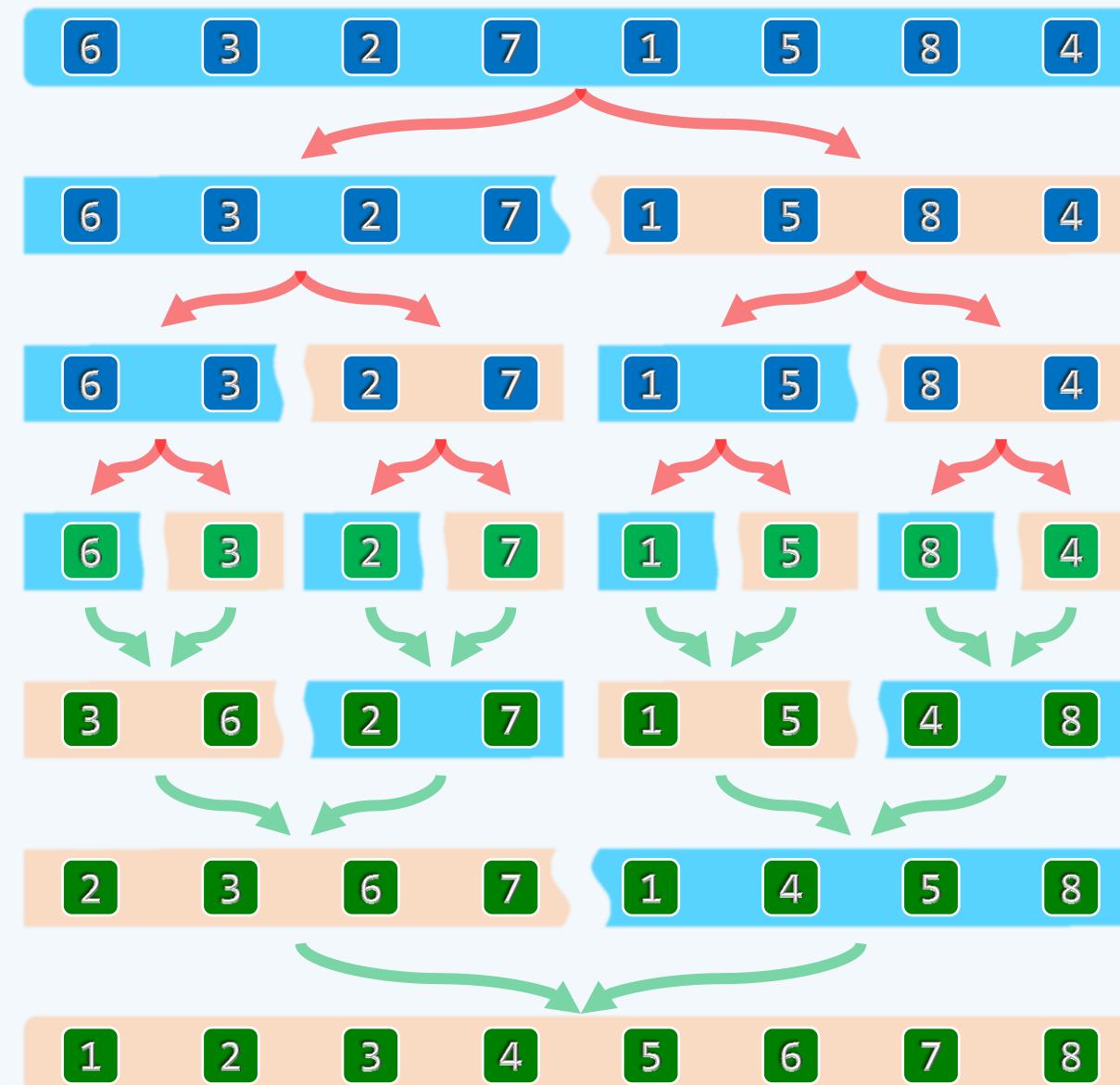
序列一分为二 // $\Theta(1)$

子序列递归排序 // $2 \times T(\frac{n}{2})$

合并有序子序列 // $\Theta(n)$

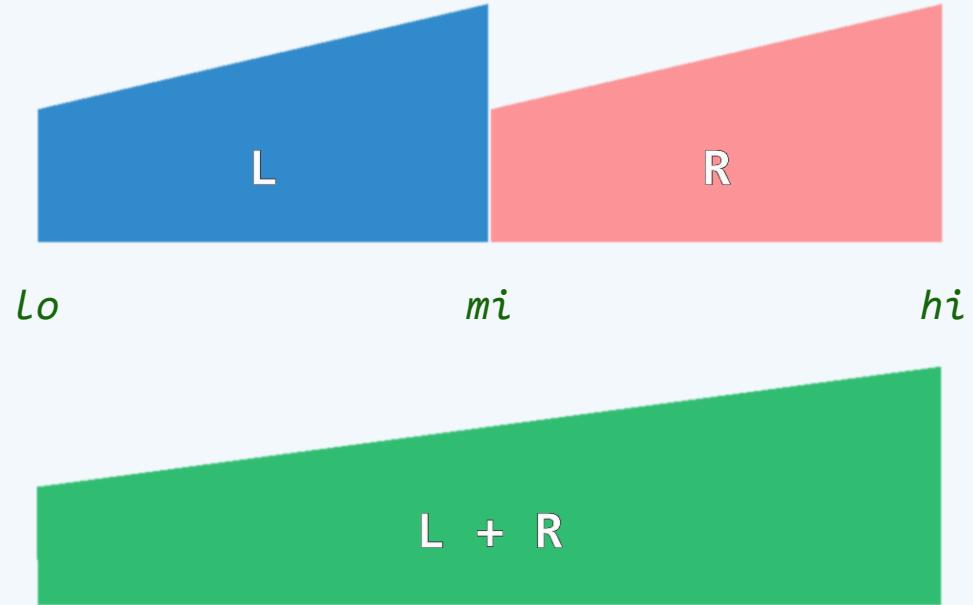
❖ 若真能如此，整体的运行成本

应是 $\mathcal{O}(n \cdot \log n)$



分而治之

```
❖ template <typename T> void Vector<T>::mergeSort( Rank lo, Rank hi ) {  
    if ( hi - lo < 2 ) return; //单元素区间自然有序，否则...  
  
    int mi = (lo + hi) >> 1; //以中点为界  
    mergeSort( lo, mi ); //对前半段排序  
    mergeSort( mi, hi ); //对后半段排序  
    merge( lo, mi, hi ); //归并  
}
```



向量

归并排序：二路归并

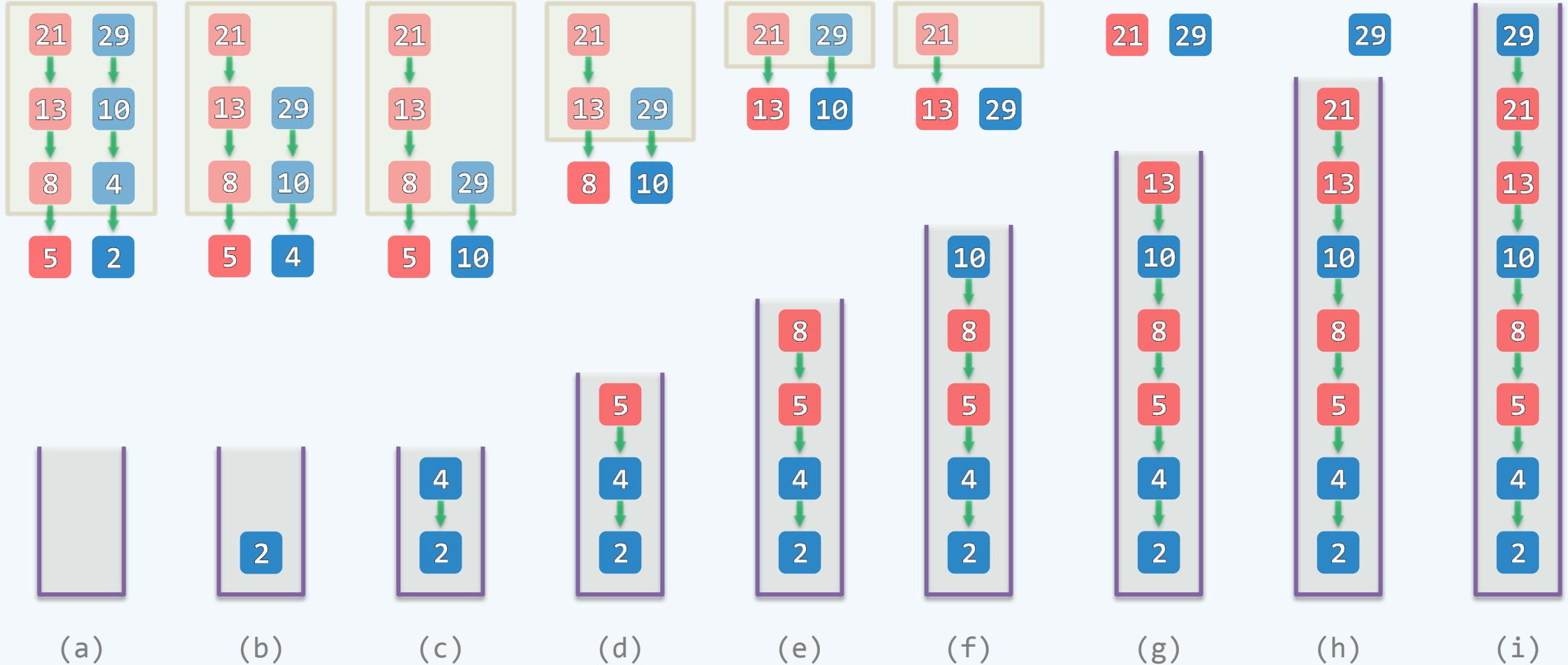
邓俊辉

deng@tsinghua.edu.cn

天下大势，分久必合，合久必分

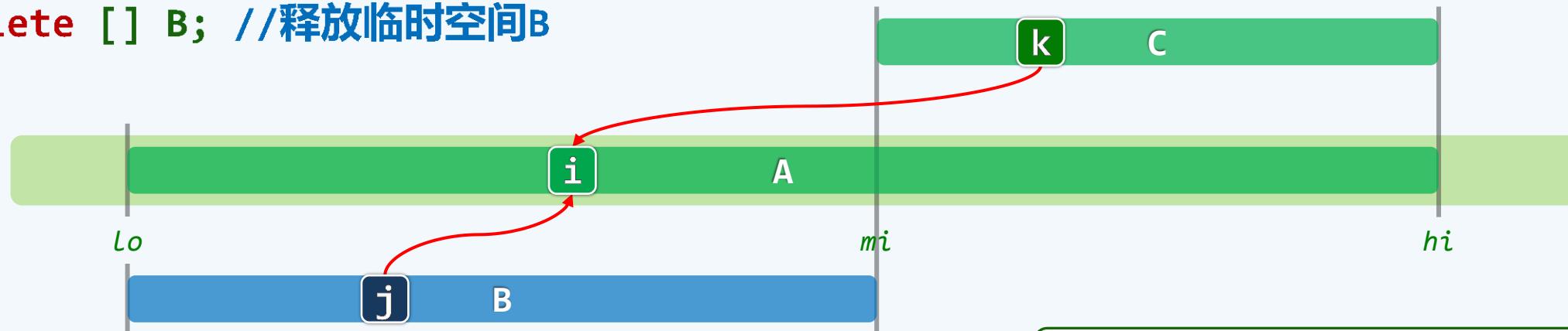
二路归并

❖ 2-way merge : 有序序列，合二为一： $S[lo, hi] = S[lo, mi] + S[mi, hi]$



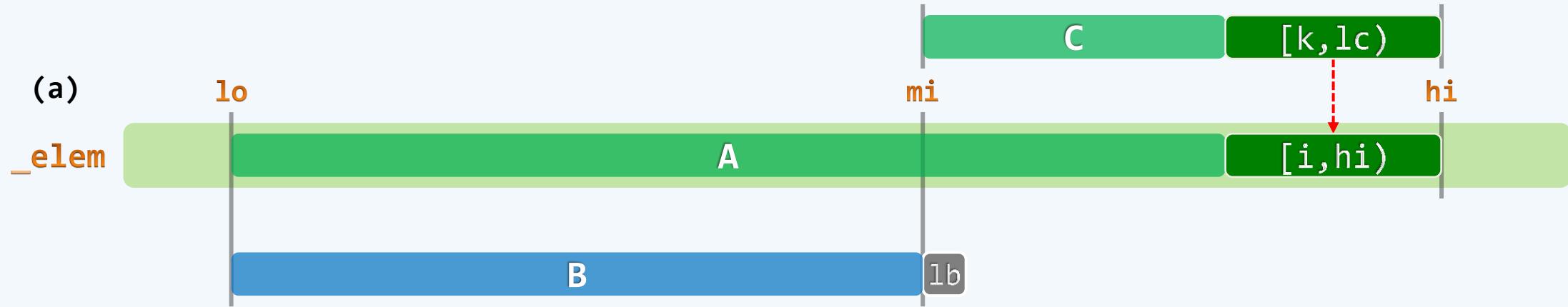
实现

```
template <typename T> void Vector<T>::merge( Rank lo, Rank mi, Rank hi ) {  
    T* A = _elem + lo; int lb = mi - lo; T* B = new T[lb]; //A[0, hi - lo) = _elem[lo, mi)  
    for ( Rank i = 0; i < lb; i++ ) B[i] = A[i]; //复制前子向量B[0, lb) = _elem[lo, mi)  
    int lc = hi - mi; T* C = _elem + mi; //后子向量C[0, lc) = _elem[mi, hi)  
    for ( Rank i = 0, j = 0, k = 0; j < lb; ) //归并：反复从B和C首元素中取出更小者  
        A[i++] = ( lc <= k || B[j] <= C[k] ) ? B[j++] : C[k++]; //将其归入A中  
    delete [] B; //释放临时空间B  
}
```

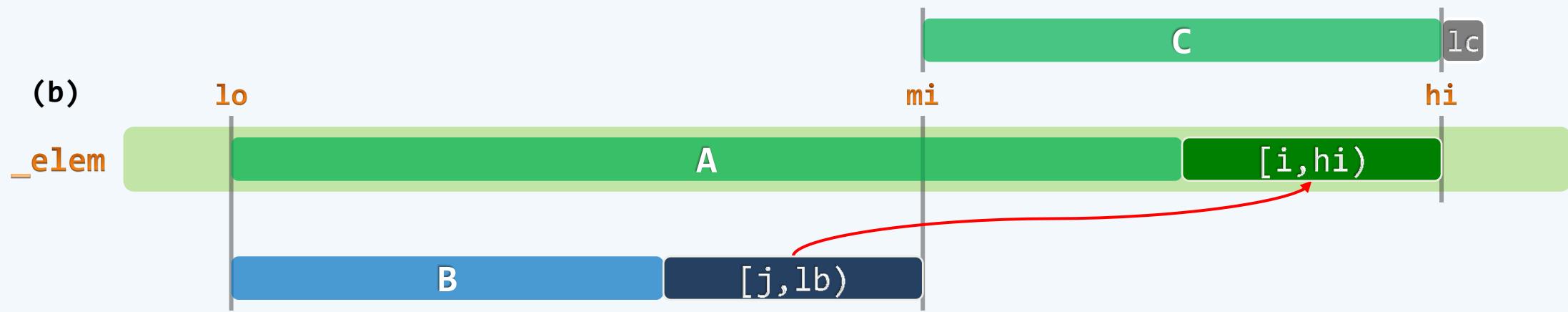


正确性

(a)



(b)



向量

归并排序：复杂度

I think there is a world market for about five computers.

- T. J. Watson, 1943

邓俊辉

deng@tsinghua.edu.cn

❖ 算法的运行时间主要消耗于for循环，共有两个控制变量

- 初始： $j = 0, k = 0$
- 最终： $j = 1b, k = 1c$
- 亦即： $j + k = 1b + 1c = hi - lo = n$

❖ 观察：每经过一次迭代，j和k中至少有一个会加一（ $j+k$ 也必至少加一）

❖ 故知：merge()总体迭代不过 $\mathcal{O}(n)$ 次，累计只需线性时间！

❖ 这一结论与排序算法的 $\Omega(n \log n)$ 下界并不矛盾——毕竟这里的B和C均已各自有序

❖ 注意：待归并子序列不必等长，允许： $1b \neq 1c, mi \neq (lo + hi)/2$

❖ 实际上，这一算法及结论也适用于另一类序列——列表（下一章）

综合评价

◆ 优点

- 实现最坏情况下最优 $\mathcal{O}(n \log n)$ 性能的第一个排序算法
- 不需随机读写，完全顺序访问——尤其适用于列表之类的序列、磁带之类的设备
- 只要实现恰当，可保证稳定——出现雷同元素时，左侧子向量优先
- 可扩展性极佳，十分适宜于外部排序——海量网页搜索结果的归并
- 易于并行化

◆ 缺点

- 非就地，需要对等规模的辅助空间——可否更加节省？
- 即便输入完全（或接近）有序，仍需 $\Omega(n \log n)$ 时间——如何改进？

向量

位图：数据结构

这样做能保存的信息量就小多了，不到原来的万分之一，但他们也只能接受这个结果。

邓俊辉

deng@tsinghua.edu.cn

整数集

$k \in S ?$ **bool test(int k);**

$S \cup \{k\}$ **void set(int k);**

$S \setminus \{k\}$ **void clear(int k);**



结构

```
❖ class Bitmap {
```

```
private:
```

```
int N;
```

```
char * M;
```

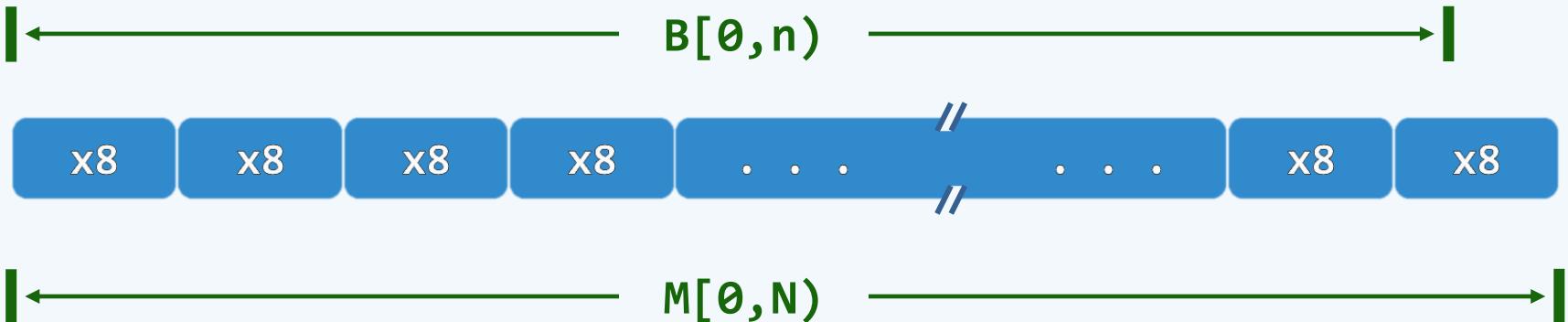
```
public:
```

```
    Bitmap( int n = 8 ) { M = new char[ N = (n+7)/8 ]; memset( M, 0, N ); }
```

```
    ~Bitmap() { delete [] M; M = NULL; }
```

```
    void set( int k ); void clear( int k ); bool test( int k );
```

```
};
```

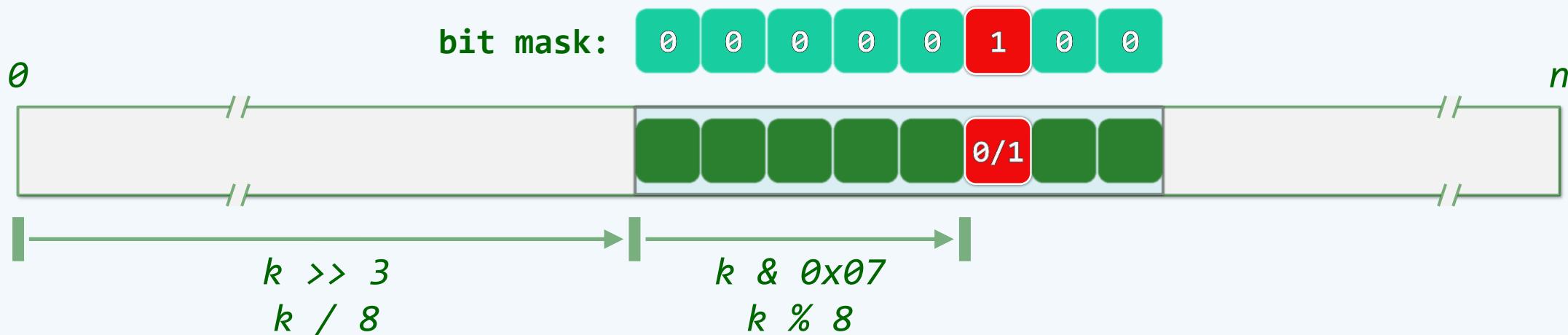


实现

```
❖ bool test( int k ) { return M[ k >> 3 ] & ( 0x80 >> (k & 0x07) ); }

❖ void set( int k ) { expand( k ); M[ k >> 3 ] |= ( 0x80 >> (k & 0x07) ); }

❖ void clear( int k ) { expand( k ); M[ k >> 3 ] &= ~(0x80 >> (k & 0x07)); }
```



向量

位图：典型应用

邓俊辉

deng@tsinghua.edu.cn

Those too big to pass through are our friends.

❖ 老问题

- `int A[n]` 的元素均取自 $[0, m)$
- 如何剔除其中的重复者？

❖ 比如

$$2^{24} = m \quad \ll \quad n = 10^{10}$$

亦即， $10,000,000,000$ 个 24 位无符号整数

❖ 仿照 `vector::deduplicate()` 改进版

先排序，再扫描

—— $\mathcal{O}(n \log n + n)$ —— 毫无压力

❖ 如果采用内部排序算法

至少需要 $4 * n = 40\text{GB}$ 内存

—— 否则，频繁的 I/O 将导致整体效率的低下

❖ 新特点：数据量虽大，但重复度极高

- 想想我们电脑里的 mp3、mp4
- 还有，朋友圈 ...

❖ 那么

$m \ll n$ 的条件，又应如何加以利用？

小集合 + 大数据

❖ Bitmap B(m); // $\mathcal{O}(m)$

```
for (int i = 0; i < n; i++)  
    B.set( A[i] ); //  $\mathcal{O}(n)$   
  
for (int k = 0; k < m; k++)  
    if ( B.test( k ) )  
        /* ... */; //  $\mathcal{O}(m)$ 
```

❖ 拓展：搜索引擎的使用规律亦是如此
词表规模不大，但重复度极高
——如何从中剔除重复的索引词？

❖ 总体运行时间 = $\mathcal{O}(n+m) = \mathcal{O}(n)$

❖ 空间 = $\mathcal{O}(m)$

- 就上例而言，降至：

$$m/8 = 2^{21} = 2\text{MB} \ll 40\text{GB}$$

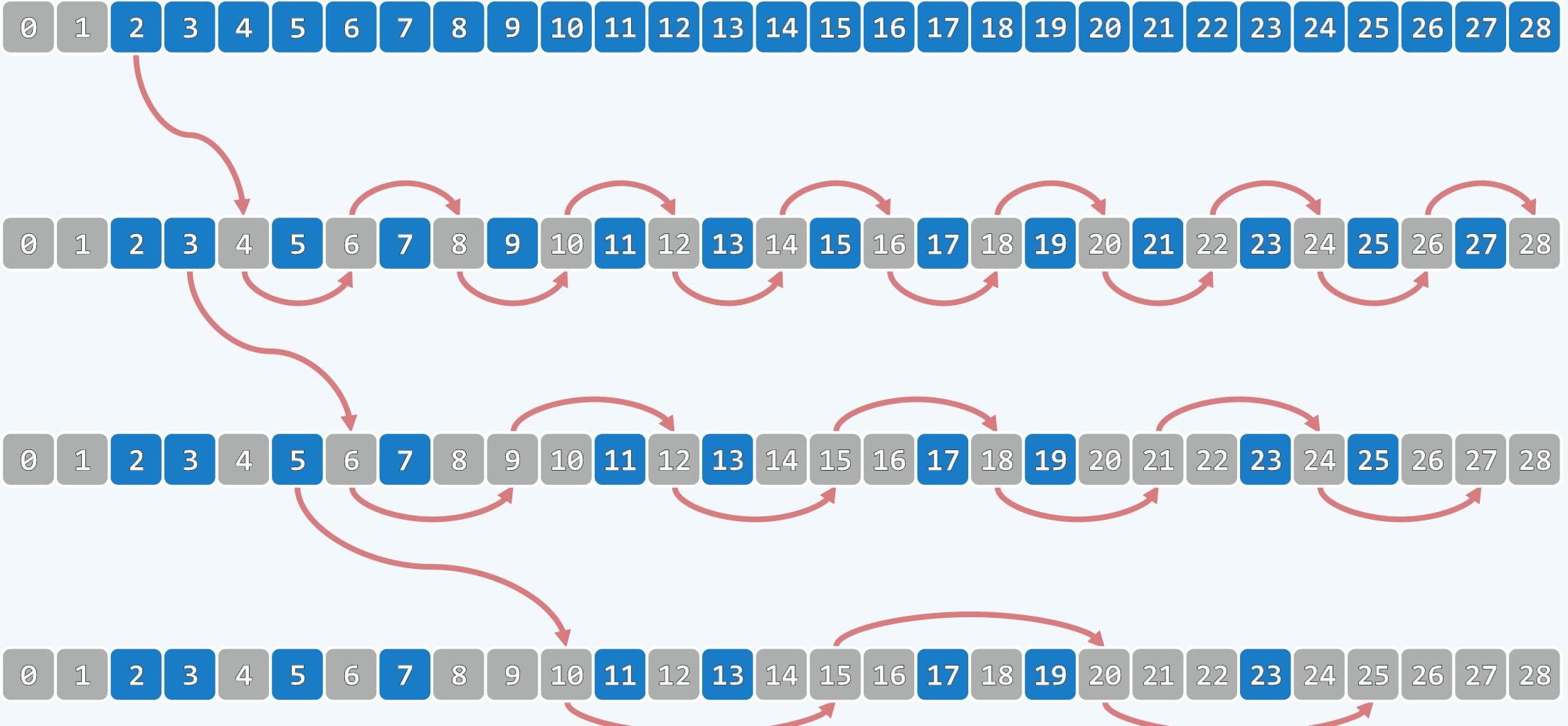
- 即便 $m = 2^{32}$ ，也不过：

$$2^{29} = 0.5\text{GB}$$

❖ 关键在于

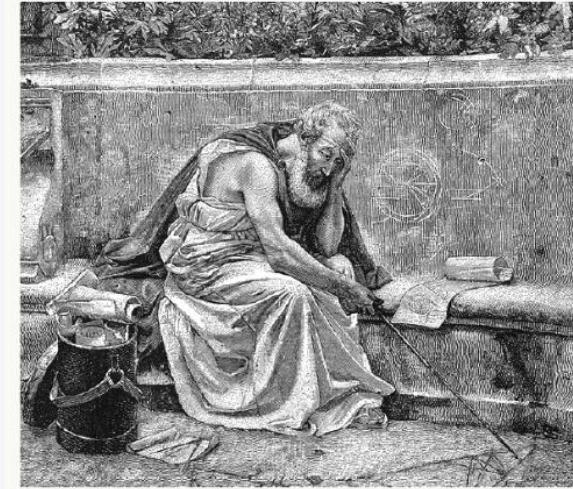
如何将查询词表转换为某一集合
——留作习题

篩法：思路



筛法：实现

```
❖ void Eratosthenes( int n, char * file ) {  
    Bitmap B( n );  
    B.set( 0 ); B.set( 1 );  
    for ( int i = 2; i < n; i++ )  
        if ( ! B.test( i ) )  
            for ( int j = 2*i; j < n; j += i )  
                B.set( j );  
    B.dump( file );  
}
```



Eratosthenes
(276 ~ 194 B.C.)

篩法：效果

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
17 18
19 20
21 22
23 24
25 26
27 28
29 30

1 2 3
4 5 6
7 8 9
10 11 12
13 14 15
16 17 18
19 20 21
22 23 24
25 26 27
28 29 30

1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
26 27 28 29 30

1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

效率与改进

❖ 不计内循环，外循环自身每次仅一次加法、两次判断，累计 $\mathcal{O}(n)$

❖ 内循环每趟迭代 $\mathcal{O}(n/i)$ 步，由素数定理至多 $n/\ln n$ 趟，累计耗时不过

$$n/2 + n/3 + n/5 + n/7 + n/11 + \dots$$

$$< n/2 + n/3 + n/4 + n/5 + n/6 + \dots + n/(n/\ln n)$$

$$= \mathcal{O}(n \cdot (\ln(n/\ln n) - 1)) = \mathcal{O}(n \cdot \ln n - n \cdot \ln(\ln(n))) = \mathcal{O}(n \cdot \log n)$$

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30
31	32	33	34	35

❖ 循环起点 “`i+i`” 可改作 “`i*i`” //为什么？

❖ 如此，内循环的长度将由 $\mathcal{O}(n/i)$ 降至 $\mathcal{O}(\max(1, n/i - i))$ //从渐进的角度看，是否实质的改进？

❖ 基于以上，如何实现primeNLT(int low)？

向量

从那天看见他我心里就放不下呀
因此上我偷偷地就爱上他呀
但愿这个年轻的人哪他也把我爱呀
过了门，他劳动，我生产，又织布，纺棉花
我们学文化，他帮助我，我帮助他
争一对模范夫妻立业成家呀

位图：快速初始化

我从那无比圣洁的河水那里
走了回来，仿佛再生了一般
正如新的树用新的枝叶更新
一身洁净，准备就绪，就飞往星辰

邓俊辉

deng@tsinghua.edu.cn

$\mathcal{O}(n) \sim \mathcal{O}(1)$

❖ Bitmap的构造函数中，通过`memset(M, 0, N)`统一清零

这一步只需 $\mathcal{O}(1)$ 时间？不，实际上仍等效于诸位清零， $\mathcal{O}(N) = \mathcal{O}(n)$ ！

❖ 尽管这并不会影响上例的渐进复杂度，但并非所有问题都是如此

❖ 有时，对于大规模的散列表，初始化的效率直接影响到实际性能

例如：第11章中`bc[]`表的构造算法，需要 $\mathcal{O}(|\Sigma|+m) = \mathcal{O}(s+m)$ 时间

若能省去`bc[]`表各项的初始化，则可严格地保证是 $\mathcal{O}(m)$

❖ 有时，甚至会影响到算法的整体渐进复杂度

例如，为从 $n=10^8$ 个32位整数中找出重复者，可仿造剔除算法... //但这里无需回收

因此，若能省去Bitmap的初始化，则只需 $\mathcal{O}(n)$ 时间

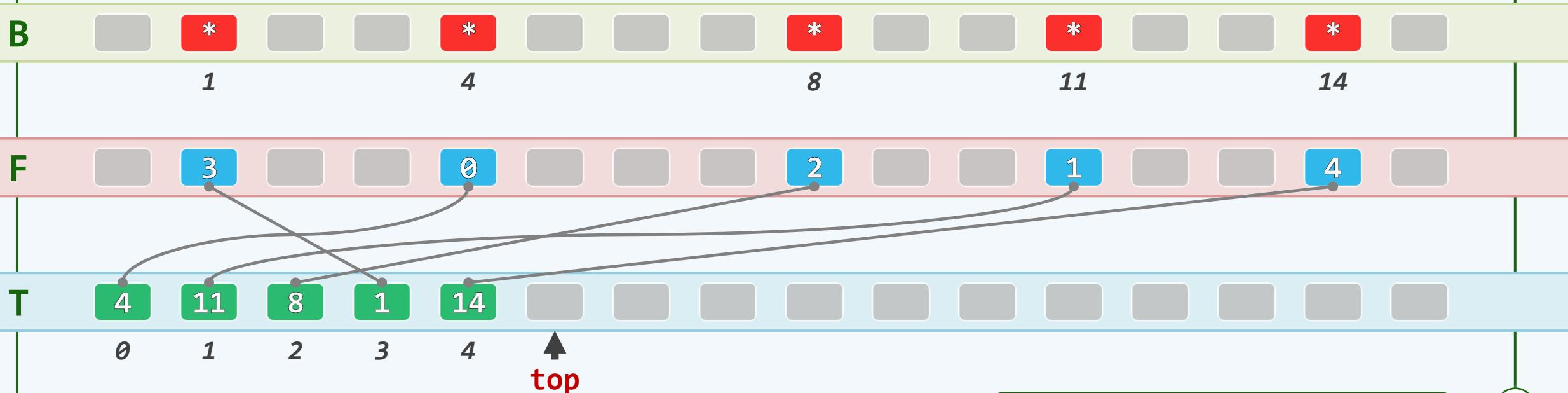
结构：校验环

❖ // [J. Hopcroft, 1974] 将B[m]拆分为一对等长的Rank型向量

//有效位须满足: T[F[k]] = k, F[T[k]] = k

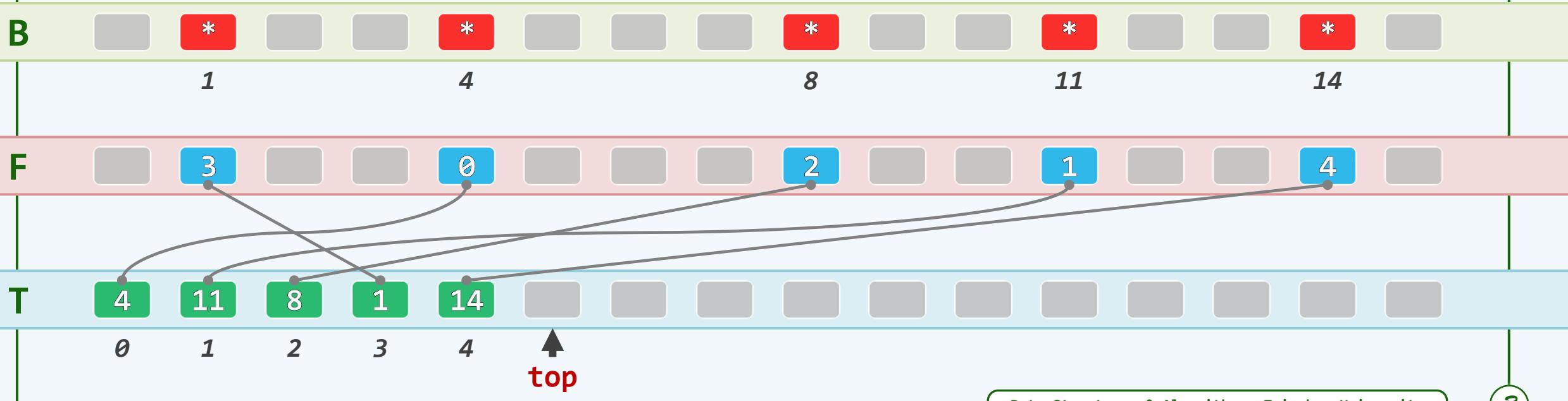
Rank F[m]; //From

Rank T[m]; Rank top = 0; //To及其栈顶指示



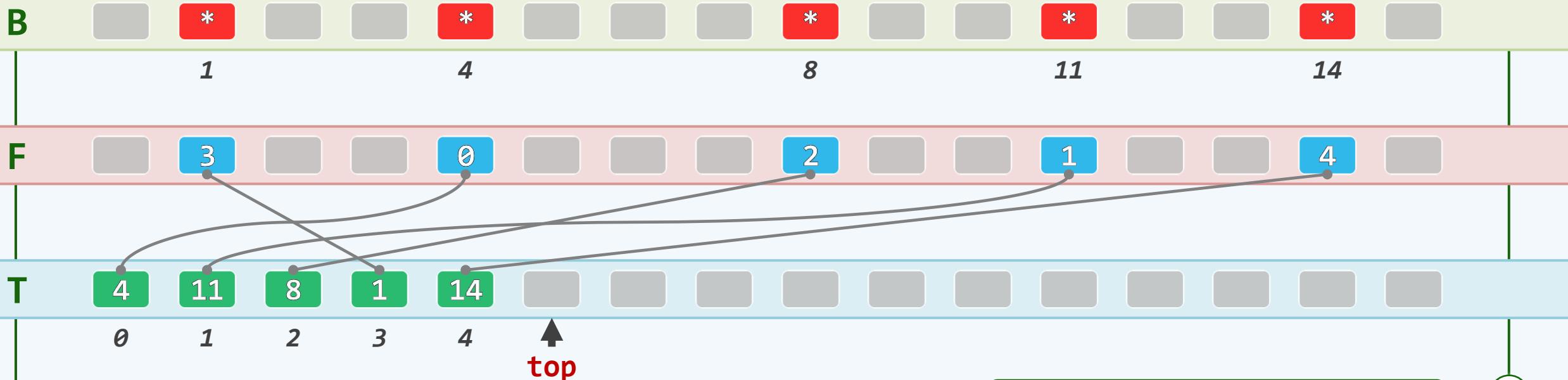
判断

```
❖ bool Bitmap::test( Rank k ) {  
    return ( 0 <= F[ k ] ) && ( F[ k ] < top )  
        && ( k == T[ F[ k ] ] );  
}
```



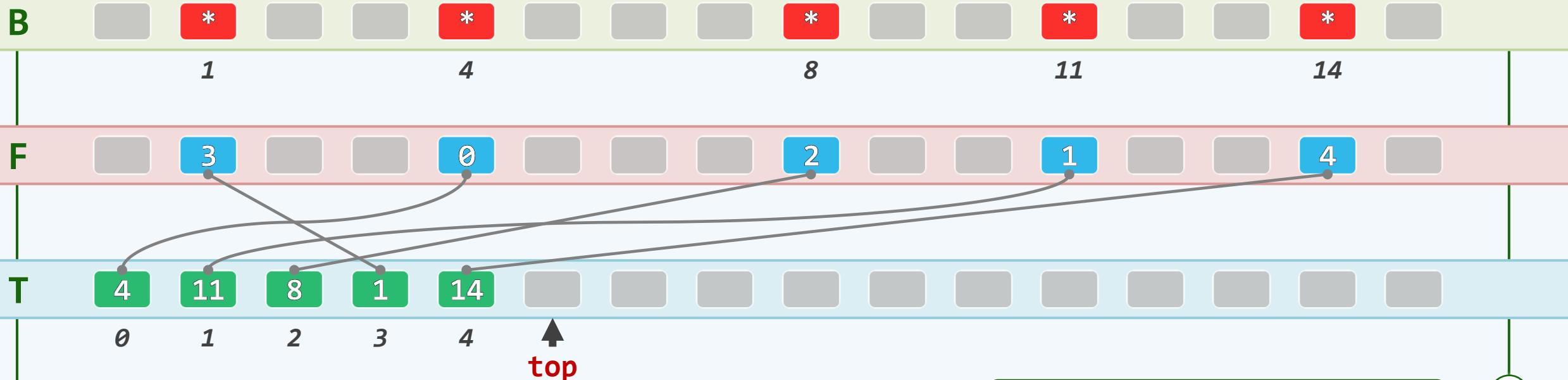
复位

❖ void Bitmap::reset() { top = 0; }



插入

```
❖ void Bitmap::set( Rank k ) {  
    if ( ! test ( k ) ) { T[ top ] = k; F[ k ] = top++; }  
}
```



删除

```
❖ void Bitmap::clear( Rank k ) {  
    if ( test ( k ) && ( --top ) )  
    { F[ T[ top ] ] = F[ k ]; T[ F[ k ] ] = T[ top ]; }  
}
```

