Algorithm Design and Analysis

David N. JANSEN, Bohua ZHAN
名 姓

算法设计与分析

詹博华，杨大卫

This week's content

- Today Wednesday:
  - Chapter 11: Hashing (small addition)
  - Chapter 17: Amortized Analysis
  - Exercises
- Tomorrow Thursday:
  - Exercise solutions
  - Chapter 22: Elementary Graph Algorithms

这周的内容

- 今天周三：
  - 第11章：散列表（小附录）
  - 第17章：摊还分析
  - 练习
- 明天周四：
  - 练习题解答
  - 第22章：基本的图算法

Algorithm Design and Analysis

算法设计与分析

Hash Tables

散列表

David N. JANSEN
名 姓

杨大卫

Ch. 17 17章

How to find a perfect hash table

如何找到完全的散列表

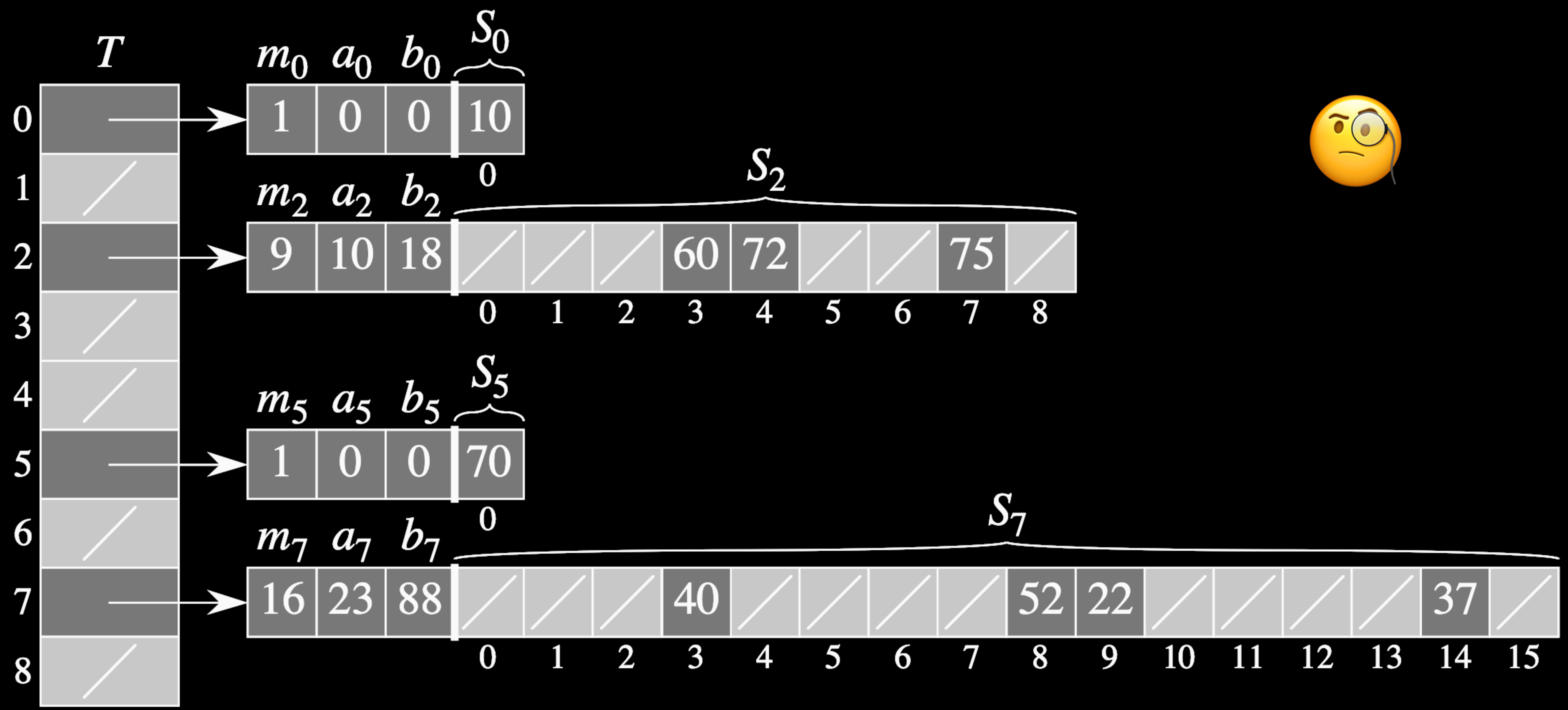


图 11-6　利用完全散列技术来存储关键字集合 $K=\{10, 22, 37, 40, 52, 60, 70, 72, 75\}$。外层的散列函数为 $h(k)=((ak+b) \bmod p) \bmod m$，这里 $a=3$，$b=42$，$p=101$，$m=9$。例如，$h(75)=2$，因此，关键字 75 散列到表 T 的槽 2 中。一个二级散列表 S_j 中存储了所有散列到槽 j 中的关键字。散列表 S_j 的大小为 $m_j=n_j^2$，并且相关的散列函数为 $h_j(k)=((a_j k+b_j) \bmod p) \bmod m_j$。因为 $h_2(75)=7$，故关键字 75 被存储在二级散列表 S_2 的槽 7 中。二级散列表没有冲突，因而查找操作在最坏情况下所需的时间为常数

How to find a perfect hash table

如何找到完全的散列表

1. Decide which data to store ➨ set K
   Primary hash table has size $n = m = |K|$.
   Choose a prime $p > \max K$.

2. Guess a, b for the outer hash function.
   Repeat until there are few collisions.
   (Corollary 11.12)

3. Guess a_i, b_i for every inner hash function.
   Repeat until there are no collisions.
   (Theorem 11.9)

1. 决定要存储的数据 ➨ 集合 K
   一次散列表的大小 $n = m = |K|$。
   选择一个素数 $p > \max K$。

2. 猜外层的散列函数的参数a、b。
   重复，直到很少发生冲突。
   （推论 11.12）

3. 猜每个内层的散列函数的参数a_i、b_i。
   重复，直到没有冲突。
   （定理 11.9）

How to find a perfect hash table

1. K = {22, 37, 40, 52, 60, 70, 72, 75, 90}.
   Primary hash table has size $m = |K| = 9$.
   Choose a prime $p = 101 > \max K = 90$.

2. Guess a, b for the outer hash function.
   Repeat until there are few collisions.
   (Corollary 11.12)
   Try $a = 22$, $b = 29$.

The probability that one needs > 4m memory cells for secondary hash tables is ≤ ½.

如何找到完全的散列表

1. K = {22, 37, 40, 52, 60, 70, 72, 75, 90}.
   一次散列表的大小 $m = |K| = 9$。
   选择一个素数 $p = 101 > \max K = 90$。

2. 猜外层的散列表函数的参数a、b。
   重复，直到很少发生冲突。
   （推论 11.12）
   试 $a = 22$、$b = 29$。

二次散列表需要 > 4m内存单元的概率 ≤ ½。

How to find a perfect hash table

如何找到完全的散列表

Check for collisions (Corollary 11.12):
Accept a, b if the memory use for secondary hash tables ≤ $4m = 36$.

检查冲突次数（推论 11.12）：
如果内存使用量 ≤ $4m = 36$,
则接受a、b。

$((ak + b) \bmod 101) \bmod 9$

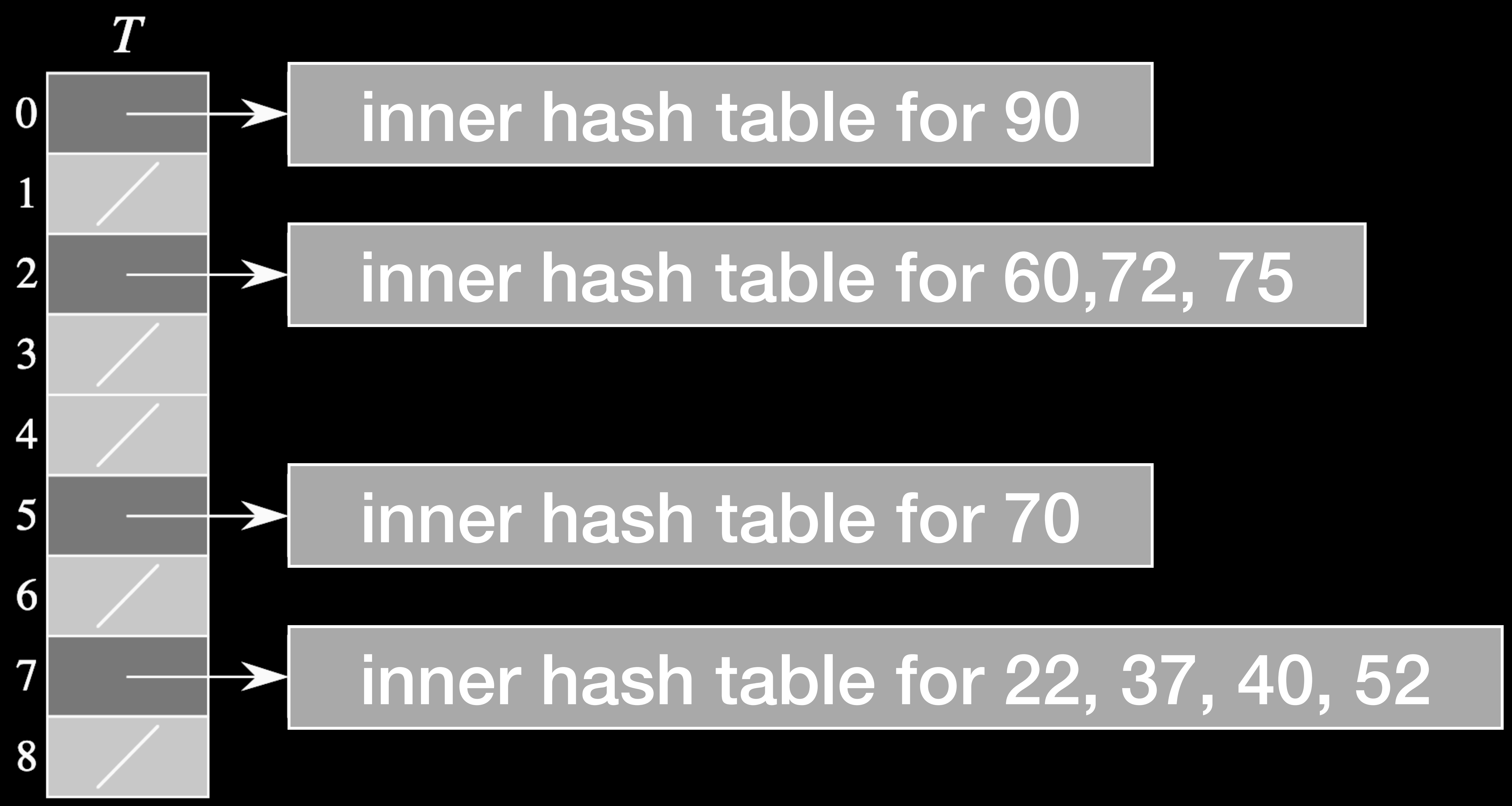
	22	37	40	52	60	70	72	75	90	memory use for secondary hash tables
$a = 22$ $b = 29$	8	0	8	0	8	0	0	8	0	$4^2 + 5^2 = 41$ entries
$a = 3$ $b = 42$	7	7	7	7	2	5	2	2	0	$4^2 + 3^2 + 1^2 + 1^2 = 27$ entries

How to find a perfect hash table

如何找到完全的散列表

Outer hash table found:

找到了外层的散列表：

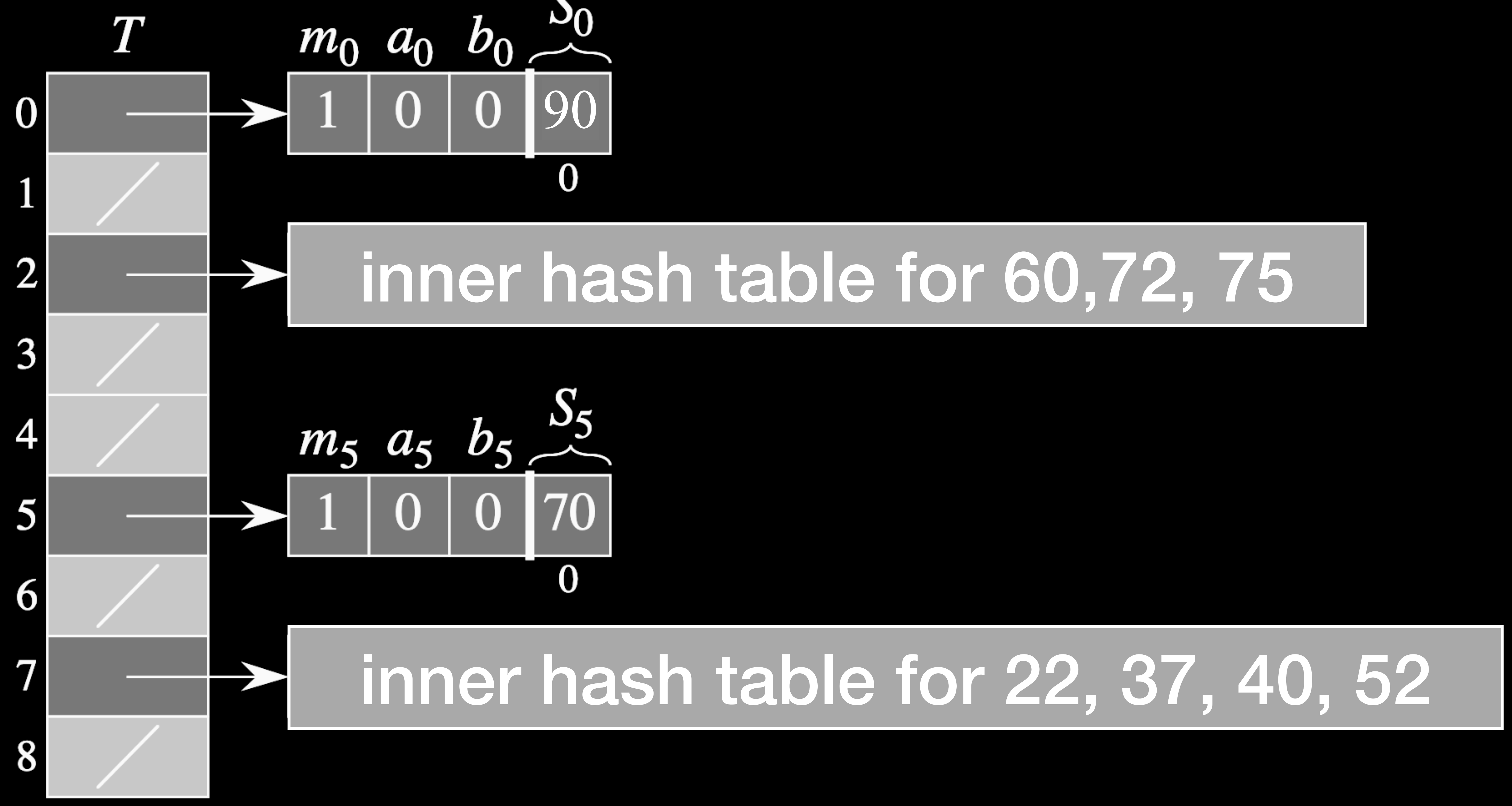


How to find a perfect hash table

如何找到完全的散列表

Inner hash tables without collisions are trivial:

没有冲突的内层的散列表很简单：



How to find a perfect hash table

如何找到完全的散列表

Guess a_2, b_2 for inner hash function for keys 60, 72, 75.
Repeat until there are no collisions.
(Theorem 11.9)

猜内层的散列函数的参数a_2、b_2
为键字60，72，75。
重复，直到没有冲突。（定理 11.9）

If m keys are stored in a hash table of size m^2, then the probability of collision is ≤ ½.

如果m个键字存储在大小为m^2的散列表中，则冲突的概率 ≤ ½。

	60	72	75
$a_2 = 6$, $b_2 = 18$	3	1	1
$a_2 = 10$, $b_2 = 18$	3	4	7

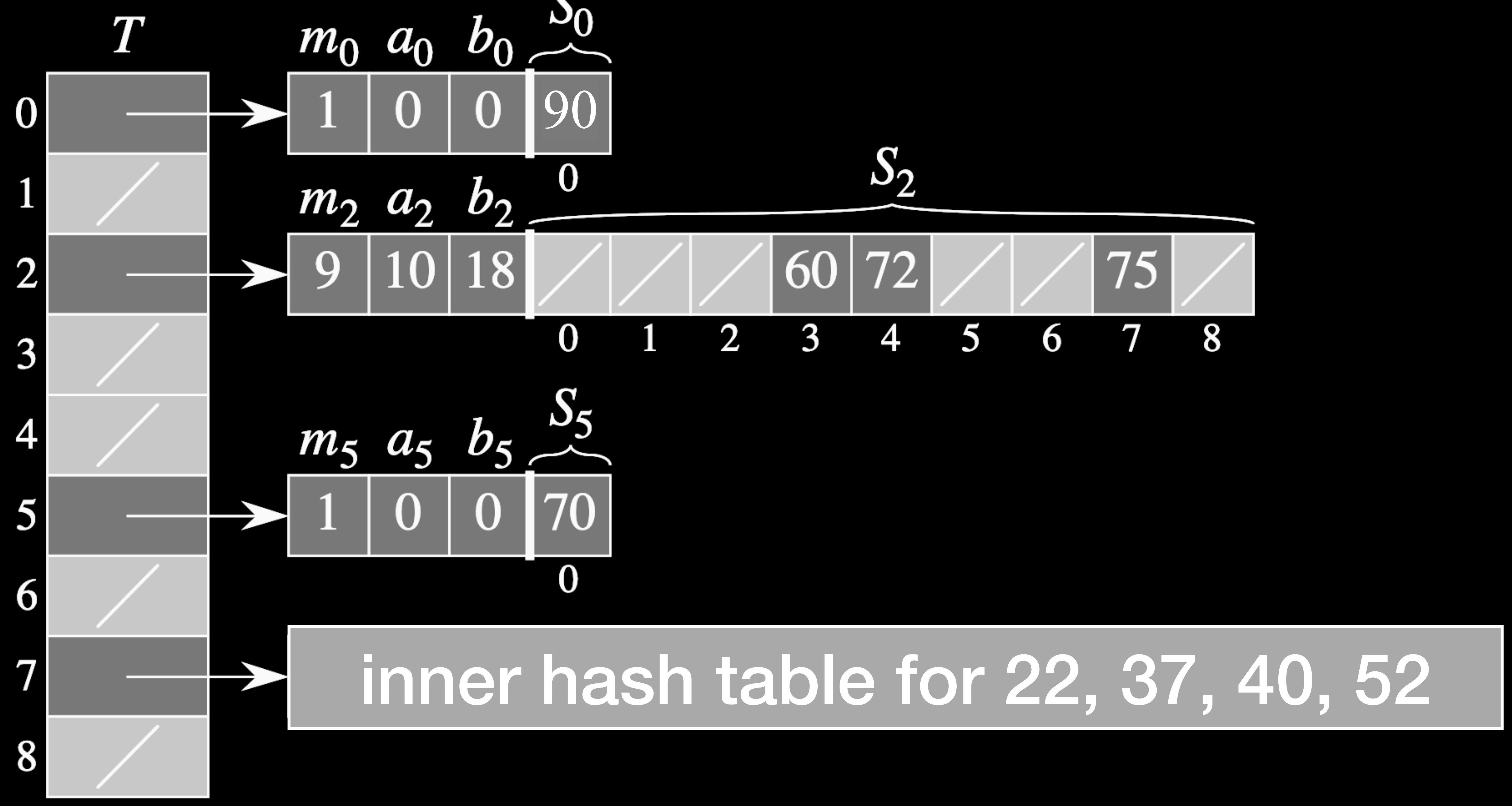
$((a_2k + b_2) \bmod 101) \bmod 9$

How to find a perfect hash table

如何找到完全的散列表

Inner hash table S_2 found:

找到了内层的散列表 S_2：



How to find a perfect hash table

如何找到完全的散列表

Guess a_7, b_7 for inner hash function for keys 22, 37, 40, 52.
Repeat until there are no collisions.
(Theorem 11.9)

猜内层的散列函数的参数a_7、b_7
为键字22，37，40，52。
重复，直到没有冲突。（定理 11.9）

	22	37	40	52
$a_7 = 66$ $b_7 = 12$	2	14	10	10
$a_7 = 84$ $b_7 = 88$	1	1	14	12
$a_7 = 23$ $b_7 = 88$	9	14	3	8

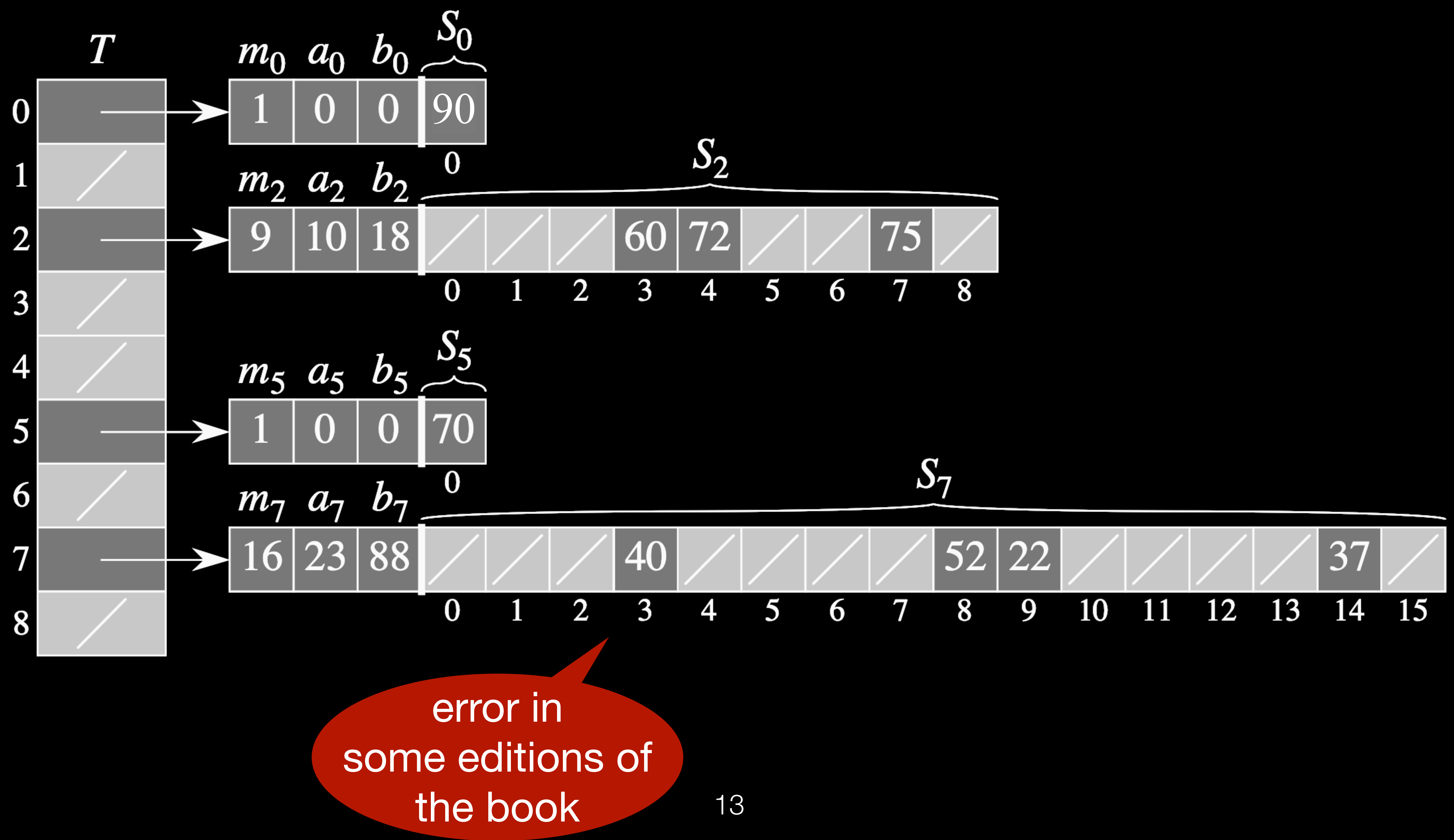
$((a_7k + b_7) \bmod 101) \bmod 16$

How to find a perfect hash table

如何找到完全的散列表

Inner hash table S_7 found:

找到了内层的散列表 S_7：



Algorithm Design and Analysis

算法设计与分析

Amortized Analysis

摊还分析

David N. JANSEN

名 姓

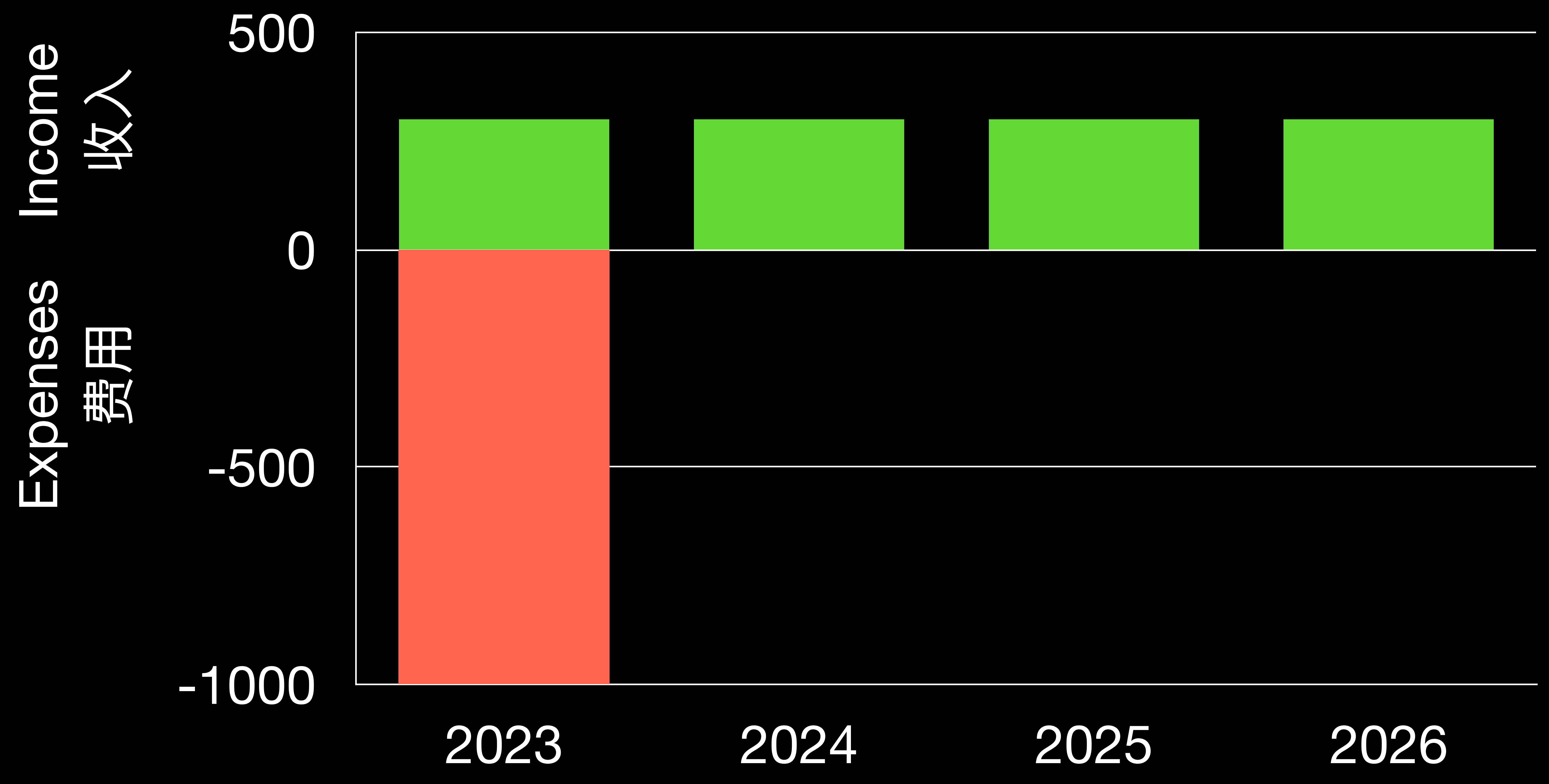
杨大卫

Ch. 17 17章

Amortize? 摊还有什么意思?

bookkeeping:
an expensive item is bought
and *amortized* over several years.

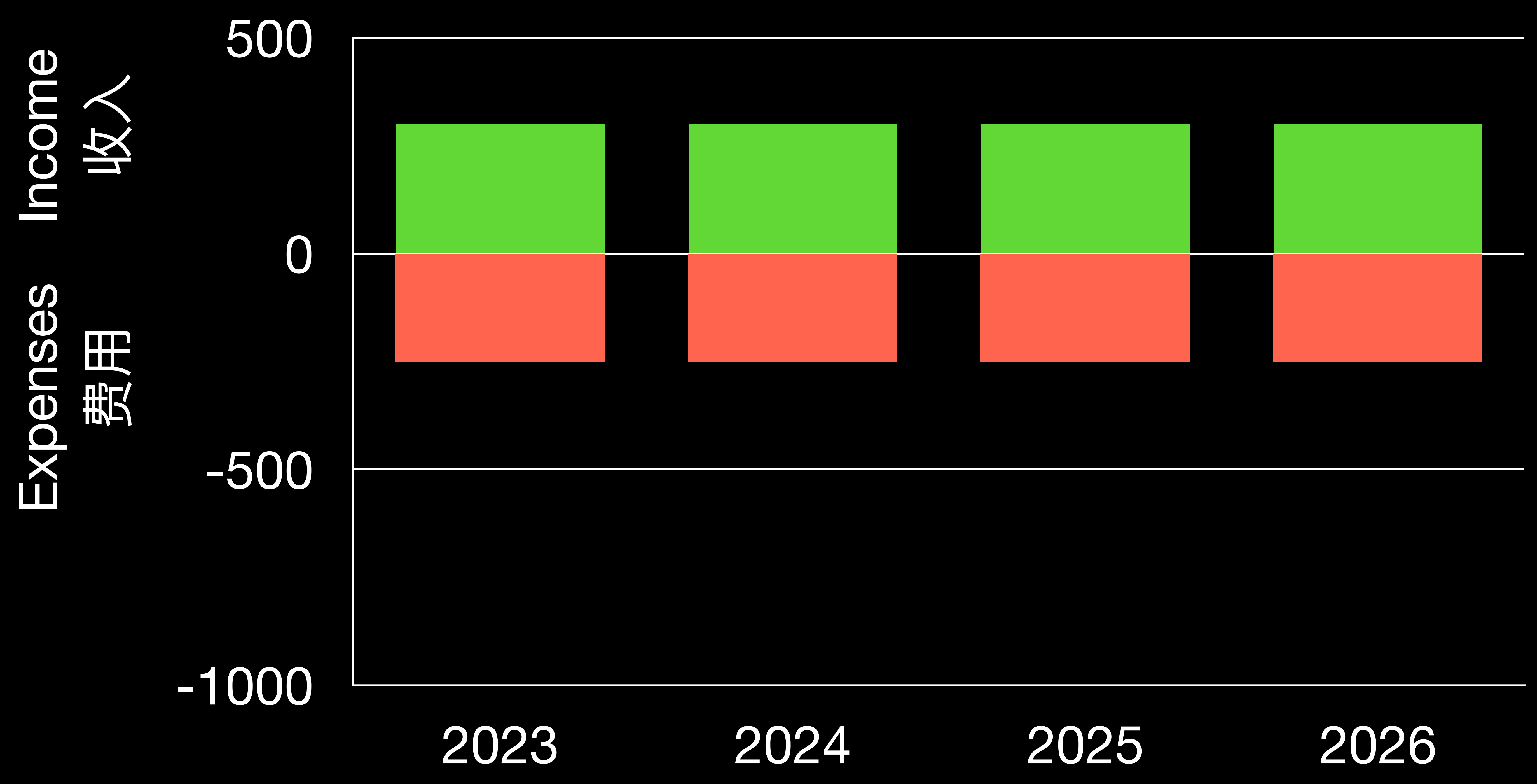
簿记:
买了一件昂贵的东西
并在几年内摊还。



Amortize? 摊还有什么意思?

bookkeeping:
an expensive item is bought
and *amortized* over several years.

簿记:
买了一件昂贵的东西
并在几年内摊还。



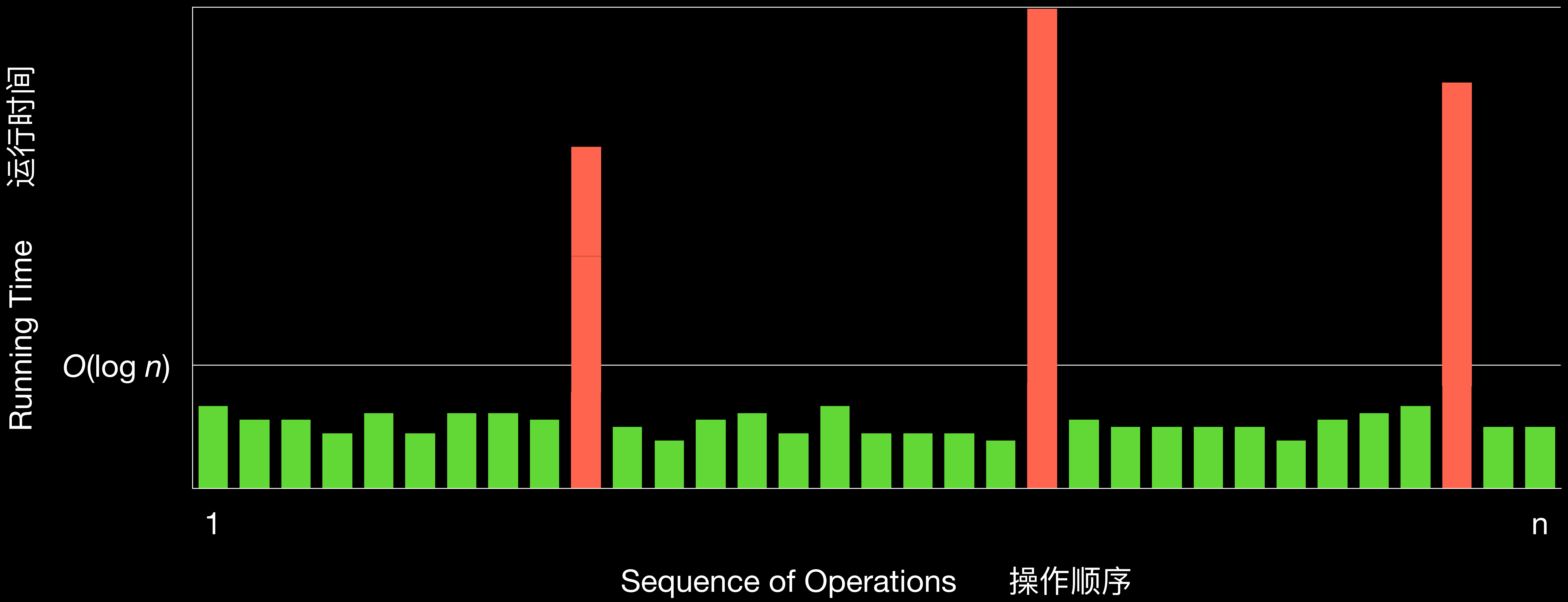
Scapegoat Tree

替罪羊树

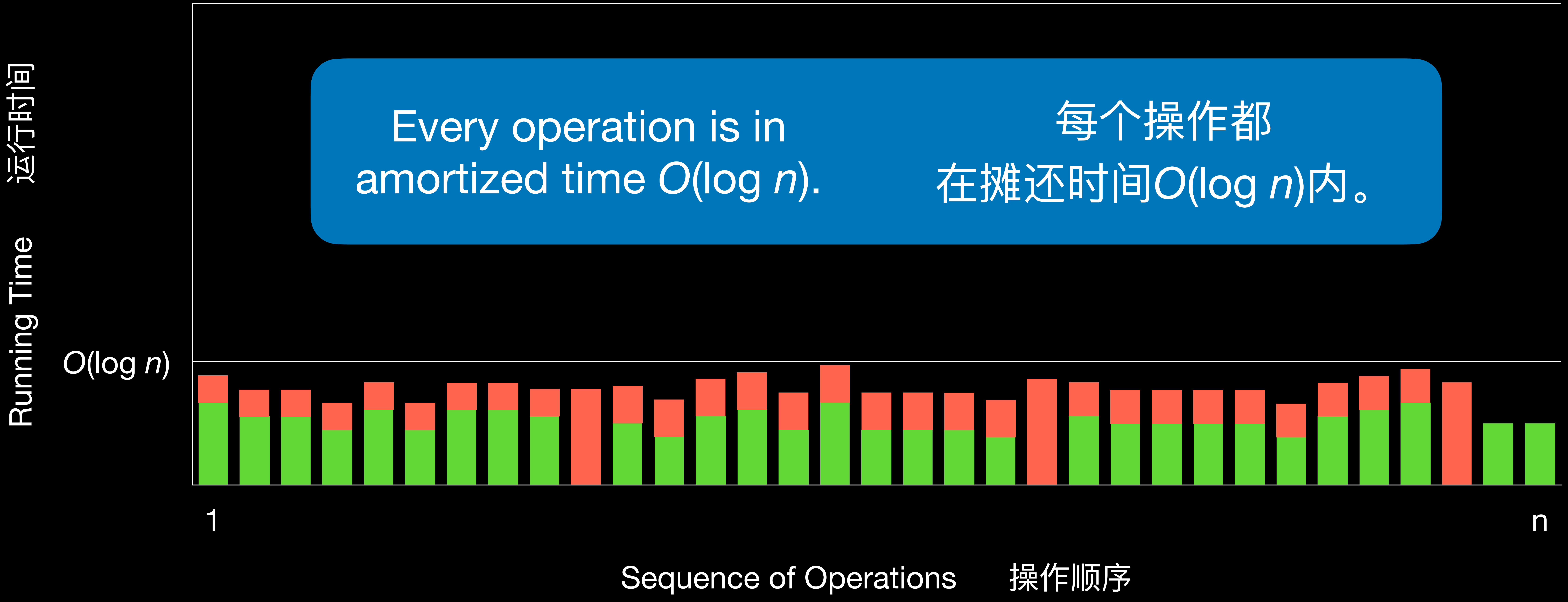
- balanced binary search tree
- Most changes
  do not destroy the balance.
- Sometimes, a sequence of changes
  destroys the balance.
  ➥ occasional rebalancing

- 平衡二叉搜索树
- 大多数变化
  不能破坏平衡。
- 有时，一条操作的序列破坏平衡。
  ➥ 偶尔的再平衡

Scapegoat Tree 替罪羊树



Scapegoat Tree 替罪羊树



Multipop Stack

多弹出堆栈

- stack with PUSH and POP operations, running time $O(1)$
- additional operation:

- 有PUSH与POP操作的堆栈，运行时间在O(1)内
- 增加操作：

```
MULTIPOP(S, k)
while S is not empty ∧ k > 0
        POP(S)
        k = k – 1
```

- MULTIPOP takes time in $O(\min\{|S|, k\})$

- MULTIPOP的运行时间在$O(\min\{|S|, k\})$内

Simple Analysis of Running Time

运行时间浅析

- Assume given a sequence of n PUSH/POP/MULTIPOP operations on a stack that starts empty.
- The maximum stack size is in $O(n)$.
- A MULTIPOP operation may need up to $O(n)$ time.
- So the total running time is in $O(n^2)$.

- 假设在一个开始为空的堆栈上，给定一个n个PUSH/POP/MULTIPOP 操作序列。
- 最大堆栈大小以$O(n)$为单位。
- MULTIPOP操作最多需要 $O(n)$时间。
- 因此，总运行时间以 $O(n^2)$内。

Simple Analysis of Running Time

运行时间浅析

- Repeated MULTIPOP operations:
  After a slow MULTIPOP operation,
  the stack is (almost) empty.
  Another MULTIPOP operation will be fast.

- We use this example to illustrate
  three amortized analysis methods.

- 重复MULTIPOP操作：
  一个慢的MULTIPOP操作以后，
  堆栈大概空的。
  第二次MULTIPOP操作快。

- 使用这个例子介绍三个摊还分析技术。

Three Methods

三个技术

- Aggregate Analysis:
  total running time of a (worst-case) sequence of operations / number of operations

- Accounting Method:
  every object can store time credit

- Potential Method:
  the whole data structure stores time credit

- 聚合分析：
  （最坏情况）操作序列的总运行时间/操作次数

- 核算法：
  每个对象都可以存储时间信用

- 势能法：
  整个数据结构存储时间信用

Aggregate Analysis

- “Aggregate” = massed together

- Idea: divide total running time of a worst-case sequence of operations by the number of operations

- upper bound on the average running time per operation

- not an average-case running time!
  not a probabilistic analysis!

聚合分析

- “聚合” = 聚集在一起

- 注意： $\dfrac{\text{最坏情况操作序列的运行时间}}{\text{操作的次数}}$

- 每次操作的平均运行时间上限

- 不是平均情况的运行时间！
  不是概率分析！

Aggregate Analysis of the Multipop Stack

聚合分析Multipop栈的构造

- Assume given a sequence of n PUSH/POP/MULTIPOP operations on a stack that starts empty.
- The maximum stack size is in $O(n)$.
- All MULTIPOPs together call POP at most $O(n)$ times and use time in $O(n)$.
- The other operations together are in $O(n)$.
- The running time of the sequence is in $O(n)$. The amortized time per operation is in $O(n)/n = O(1)$.

- 假设在一个开始为空的堆栈上，给定一个n个PUSH/POP/MULTIPOP 操作序列。
- 最大堆栈大小以$O(n)$为单位。
- 所有MULTIPOP一起调用POP最多 $O(n)$ 次，使用时间为$O(n)$。
- 其他操作一起使用时间为$O(n)$。
- 序列的运行时间为$O(n)$。每次操作的摊销时间为$O(n)/n = O(1)$。

How to use Aggregate Analysis

如何使用聚合分析

1. Assume a worst-case sequence of *n* operations.
2. Find an upper bound on the execution time of all the operations together.
3. The amortized cost of every operation is the upper bound / *n*.

1. 假设给定最坏情况 *n* 的操作序列。
2. 找到操作序列的总结运行时间的上界。
3. 所有的操作的摊还代价为上界 / *n*。

Accounting Method 核算法

- Every operation is priced individually its “amortized cost”.
- We assign the difference between amortized cost and actual running time to specific objects as credit.
- Credit can pay for later operations.
- (Credit is not actually stored as data, but only serves to calculate the time.)

- 每项业务都单独定价其“摊还代价”。
- 我们将摊还代价和实际运行时间之间的差额分配给特定对象作为信用。
- 信用可以支付以后的操作费用。
- (信用不是作为数据存储的，但只用于计算运行时间。）

Accounting Analysis of the Multipop Stack

核算法

- Prices for operations:
- 运营价格：

Operation	PUSH	POP	MULTIPOP	操作
Amortized cost	2	0	0	摊还代价

- PUSH actually costs 1 unit,
  so 1 unit is credited
  to the item pushed on the stack.
- POP actually costs 1 unit.
  This is paid by the credit of the item.

- PUSH实际成本为1个单位，
  因此，1个单位记入贷方
  到堆栈上推送的项目。
- POP实际上要花1个单位。
  这是用该物品的信用证支付的。

Accounting Analysis of the Multipop Stack

核算法

- Prices for operations:
- 运营价格：

Operation	PUSH	POP	MULTIPOP	操作
Amortized cost	2	0	0	摊还代价

- MULTIPOP: when k items are popped from the stack, it actually costs k units. This is also paid by the credit of the items.
- MULTIPOP：当从堆栈中弹出k元素时，实际上需要花费k个单位。这也是用该物品的信用证支付的。

How to use the Accounting Method

如何使用核算法

1. Choose an amortized cost for every operation

2. Define which objects store credit

3. Prove for every operation that running time + credit difference ≤ amortized cost.

1. 选择所有的操作的摊还代价

2. 定义什么对象存储信用

3. 把所有的操作证明
运行时间 + 信用减额 ≤ 摊还代价

Potential Method

势能法

- Every state of the data structure contains some “potential energy”.
- The amortized cost of an operation is its running time + the change in potential.
- Need to define a potential function Φ: states → potential energy = $\mathbb{N}$.

- 数据结构的每个状态都包含一些“势能”。
- 一项业务的摊余成本是其运行时间+潜在的变化。
- 需要定义势函数 Φ：状态 → 是能 = $\mathbb{N}$。

Potential Analysis of the Multipop Stack

- potential function:
  $\Phi(S)$ = number of items on stack S

- amortized cost of PUSH:

running time	1
+ new potential	$\lvert S\rvert + 1$
− old potential	$-\lvert S\rvert$
amortized cost	2

$|S|$ = stack size before the operation

势能法

- 势函数：
  $\Phi(S)$ = S栈内的对象数量

- PUSH的摊还代价：

运行时间	1
+ 操作后的势能	$\lvert S\rvert + 1$
− 操作前的势能	$-\lvert S\rvert$
摊还代价	2

$|S|$ = 操作前的对栈大小

Potential Analysis of the Multipop Stack

势能法

- potential function:
  $\Phi(S)$ = number of items on stack S

- amortized cost of POP:

running time	1
+ new potential	$\lvert S\rvert - 1$
– old potential	$-\lvert S\rvert$
amortized cost	0

$\lvert S\rvert$ = stack size before the operation

- 势函数：
  $\Phi(S)$ = S栈内的对象数量

- POP的摊还代价：

运行时间	1
+ 操作后的势能	$\lvert S\rvert - 1$
– 操作前的势能	$-\lvert S\rvert$
摊还代价	0

$\lvert S\rvert$ = 操作前的对栈大小

Potential Analysis of the Multipop Stack

势能法

- potential function:
  $\Phi(S)$ = number of items on stack S

- amortized cost of MULTIPOP:

running time	$\min\{k,\lvert S\rvert\}$
+ new potential	$\max\{\lvert S\rvert - k, 0\}$
– old potential	$-\lvert S\rvert$
amortized cost	0

$|S|$ = stack size before the operation

- 势函数：
  $\Phi(S)$ = S栈内的对象数量

- MULTIPOP的摊还代价：

运行时间	$\min\{k,\lvert S\rvert\}$
+ 操作后的势能	$\max\{\lvert S\rvert - k, 0\}$
– 操作前的势能	$-\lvert S\rvert$
摊还代价	0

$\max\{|S| - k, 0\} =$
$|S| + \max\{-k, -|S|\} =$
$|S| - \min\{k, |S|\}$

$|S|$ = 操作前的对栈大小

How to use Potential Analysis

如何使用势能法

1. Define a potential function
   Φ: states → potential energy = ℕ.

2. Calculate the amortized cost
   of every operation.

1. 定义势函数
   Φ：状态 → 是能 = ℕ。

2. 计算所有的操作的摊还代价。

Comparison

比较

Aggregate analysis

- simple
- Every operation is assigned the same complexity.

Accounting method / Potential method

- allow operations of varying complexity
- lead to the same result in the example, but in complex data structures one or the other may appear easier to define

聚合分析

- 易于理解的
- 每个操作都分配了相同的复杂性。

核算法 / 势能法

- 允许不同复杂度的操作
- 在示例中导致相同的结果，
但在复杂的数据结构中，
一种或另一种似乎更方便定义

Dynamic Tables

- C++ **class std::vector<T>**
  Java **class java.util.ArrayList<E>**

- a table that can store elements
  and adapts its size at runtime

- When we find out that the size of the table is not enough, we reallocate a larger table and move all objects.

- We shall see:
  The amortized cost of every operation is still $O(1)$.

动态表

- C++ **class std::vector<T>** 或者
  Java **class java.util.ArrayList<E>**

- 可以存储元素的表
  并在运行时调整其大小

- 当我们发现表的大小不够时，
  我们会重新分配一个较大的表
  并移动所有对象。

- 我们将看到：
  每一个操作的摊还代价还是在$O(1)$内。

Data and Operations 数据和操作

- *T.table* = memory to store items
  T.num = number of items stored
  T.size = capacity of the memory
- TABLE-INSERT(*T*,*x*)
  adds a new item *x* to table *T*
  (may also expand *T.table*)
- TABLE-DELETE(*T*,*x*)
  deletes item *x* from table *T*
  (may also shrink *T.table*)

- *T.table* = 存储项目的内存
  T.num = 存储的项目数
  T.size = 内存的容量
- TABLE-INSERT(*T*,*x*)
  将新项目x添加到表T中
  （也可以扩展*T.table*）
- TABLE-DELETE(*T*,*x*)
  从表*T*中删除项目*x*
  （也可收缩*T.table*）

Load Factor

- $\alpha(T) = \dfrac{T.num}{T.size}$
- α(table without memory) = 1
- We want to keep $\alpha(T) \geq \frac{1}{2}$.

装载因子

- $\alpha(T) = \dfrac{T.num}{T.size}$
- α(无内存的表) = 1
- 我们想要保持 $\alpha(T) \geq \frac{1}{2}$.

Table Expansion 表扩张

$T.size/2 \le T.num \le T.size$

empty table: allocate *T.table* for the first time

空表：第一次分配*T.table*

TABLE-INSERT(T,x)
if $T.size == 0$
    allocate *T.table* with 1 slot
    $T.size = 1$
else if $T.num == T.size$
    allocate *new-table* with $2 \times T.size$ slots
    move items from *T.table* to *new-table*
    free *T.table*
    T.table = *new-table*
    $T.size = 2 \times T.size$
insert x into *T.table*
$T.num = T.num + 1$

table is full and needs to be expanded

表格充满需要扩展

$T.size/2 \le T.num \lneqq T.size$

(or $T.num = 0$ and $T.size = 1$)

$T.size/2 \le T.num \le T.size$

Table Expansion 表扩张

$T.size/2 \le T.num \le T.size$

TABLE-INSERT(T,x)
if $T.size == 0$
    allocate $T.table$ with 1 slot
    $T.size = 1$
else if $T.num == T.size$
    allocate *new-table* with $2{\times}T.size$ slots
    move items from $T.table$ to *new-table*
    free $T.table$
    $T.table$ = *new-table*
    $T.size = 2{\times}T.size$
insert x into $T.table$
$T.num = T.num + 1$

$O(T.size)$

$T.size/2 \le T.num \le T.size$

Potential Analysis

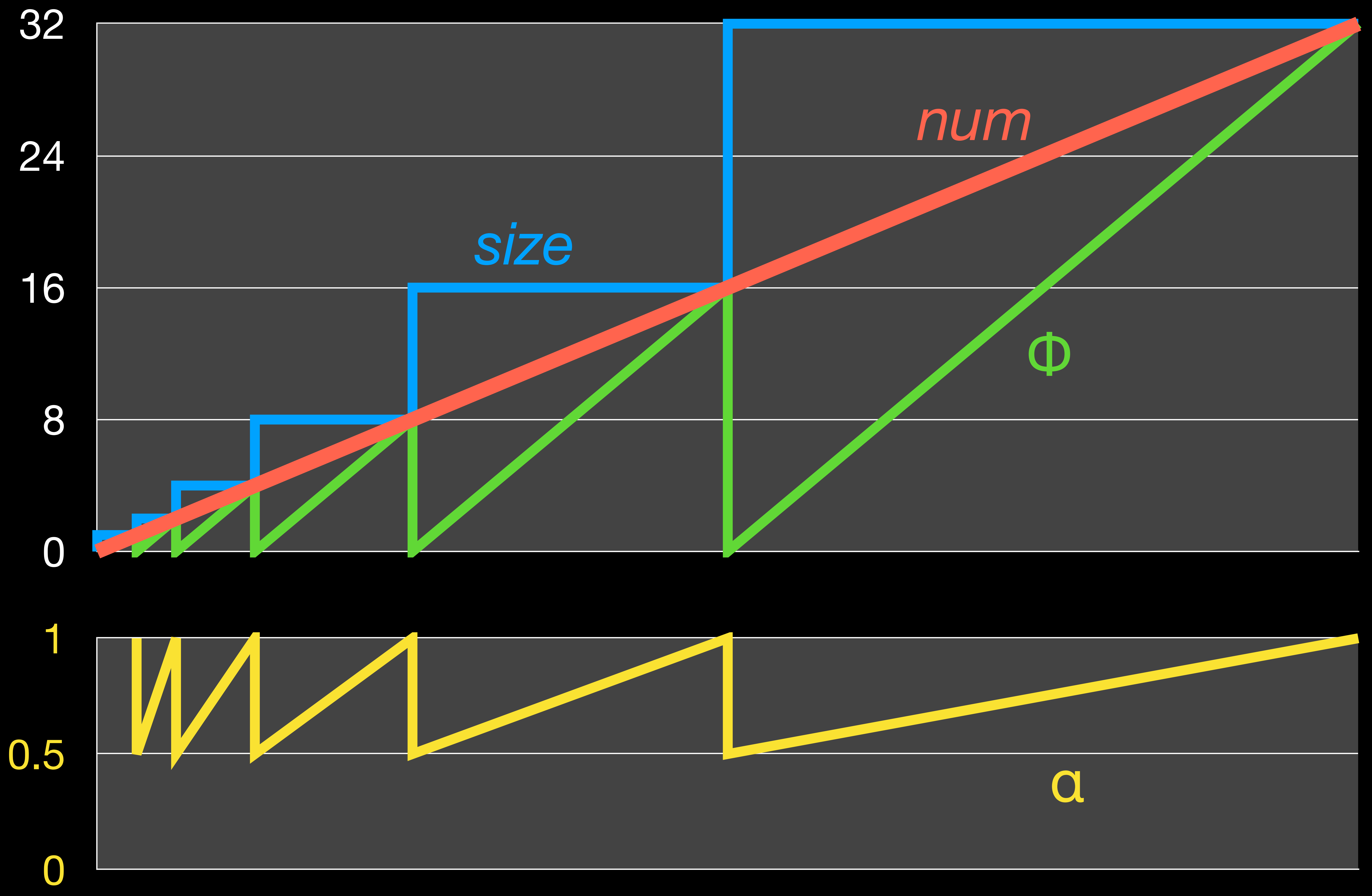
- Define the potential of a table as
  $\Phi(T) = 2\times T.num - T.size$
- never negative, as $T.size/2 \leq T.num$
- low when $\alpha(T) \approx ½$,
  high when the table is full
  (i.e. when we need to do extra work!)

势能分析

- 将表的潜力定义为
  $\Phi(T) = 2\times T.num - T.size$
- 决不为负数，如$T.size/2 \leq T.num$
- $\alpha(T) \approx ½$时为低,
  当表满了的时为高
  （即，当我们需要做额外的工作时！）

Table Expansion and Potential

表扩张和势能



Sequence of Insertions　　插入的序列

Table Expansion and Potential

表扩张和势能

Expansion can use time $O(size - 0)$.

扩展可以用 $O(size - 0)$的时间。

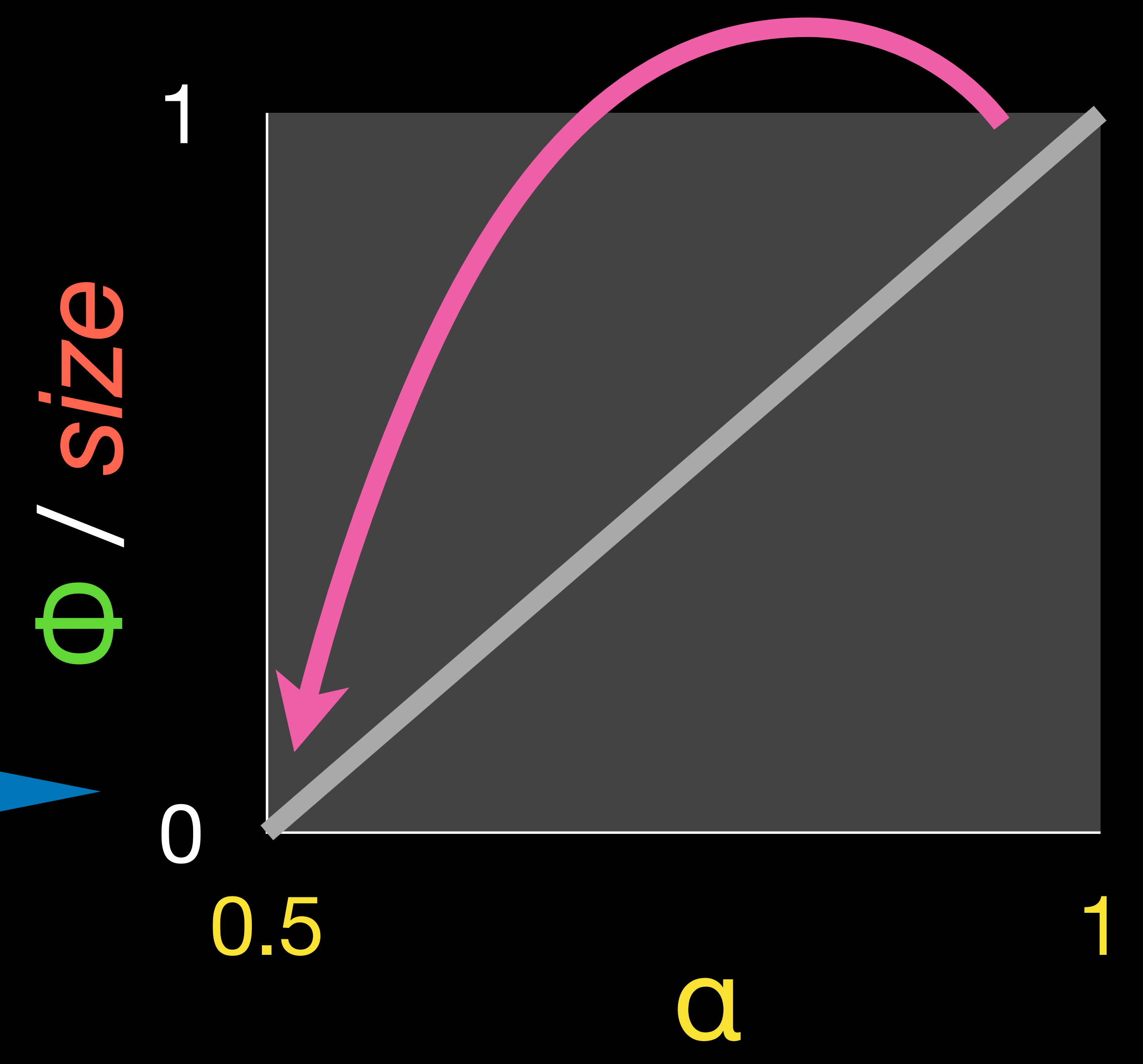


Table full: $\Phi = size$

表格充满： $\Phi = size$

Table ½-full: $\Phi = 0$

表格半满： $\Phi = 0$

Table Shrinking and Potential

表收缩和势能

Shrinking can use time $O(0 - size)$?

收缩可以用 $O(0 - size)$的时间吗?

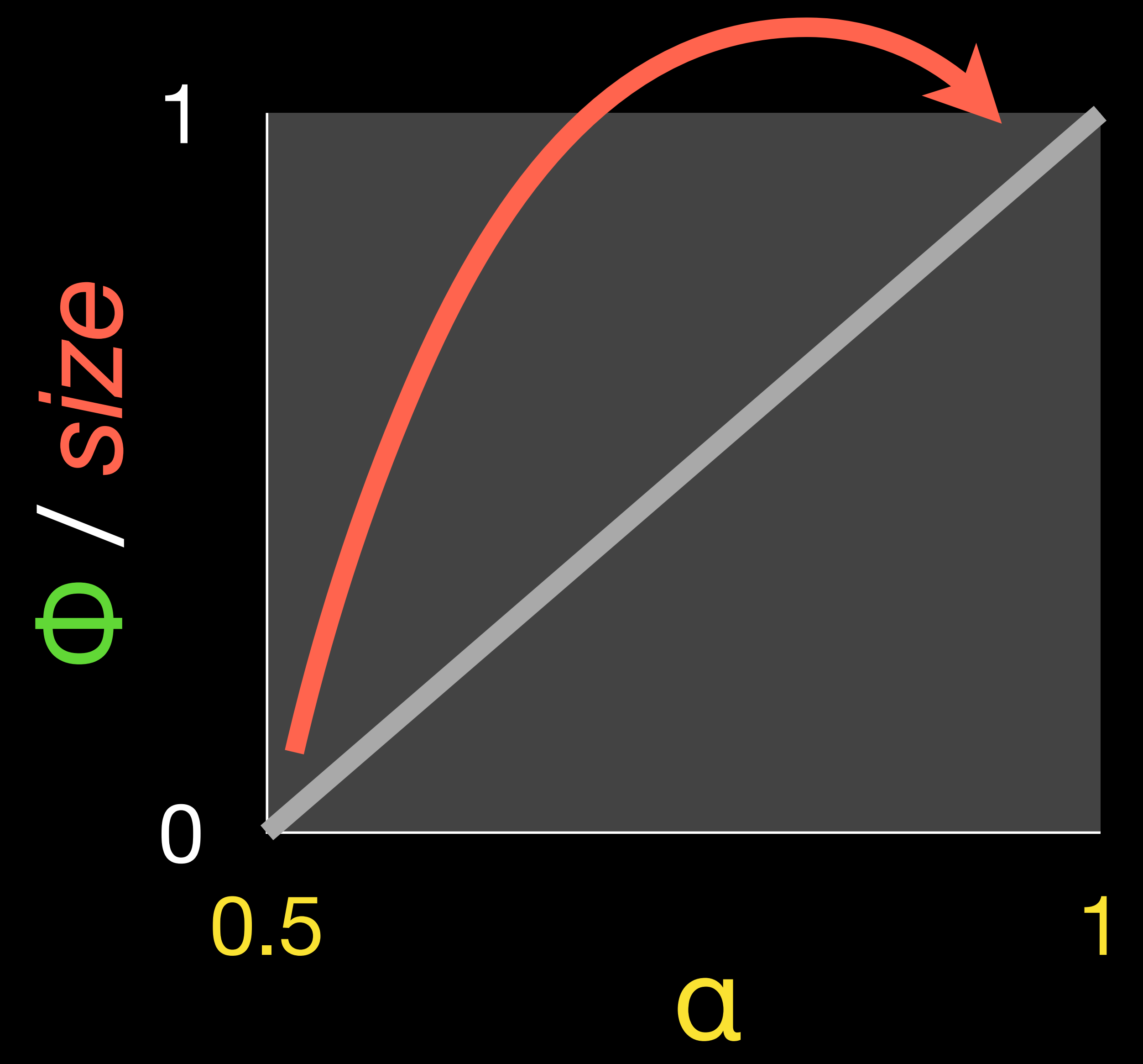


Table Shrinking and Potential

表收缩和势能

Shrinking can use time $O(size/4 - 0) = O(num)$.

收缩可以用$O(size/4 - 0) = O(num)$的时间。

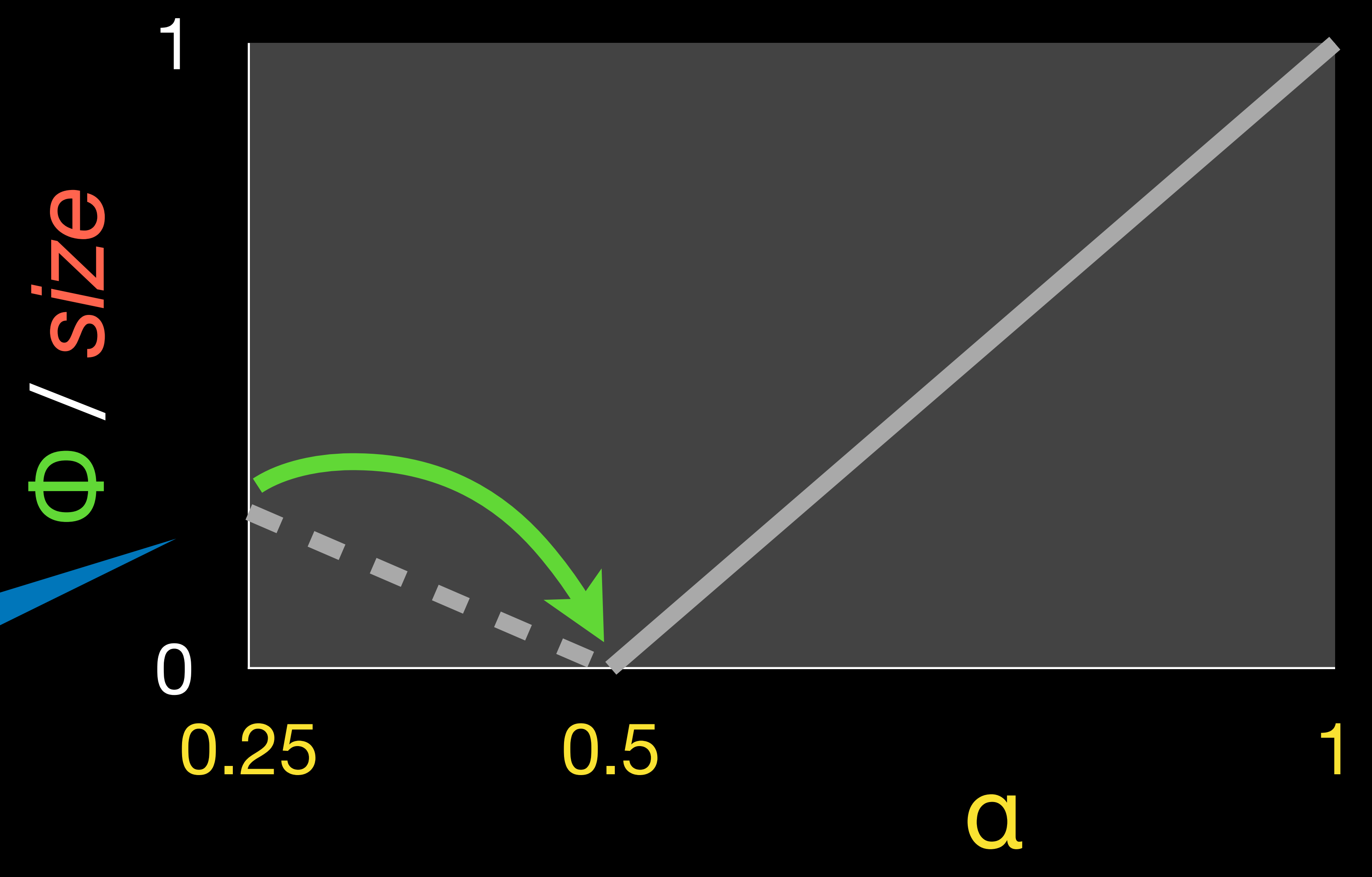
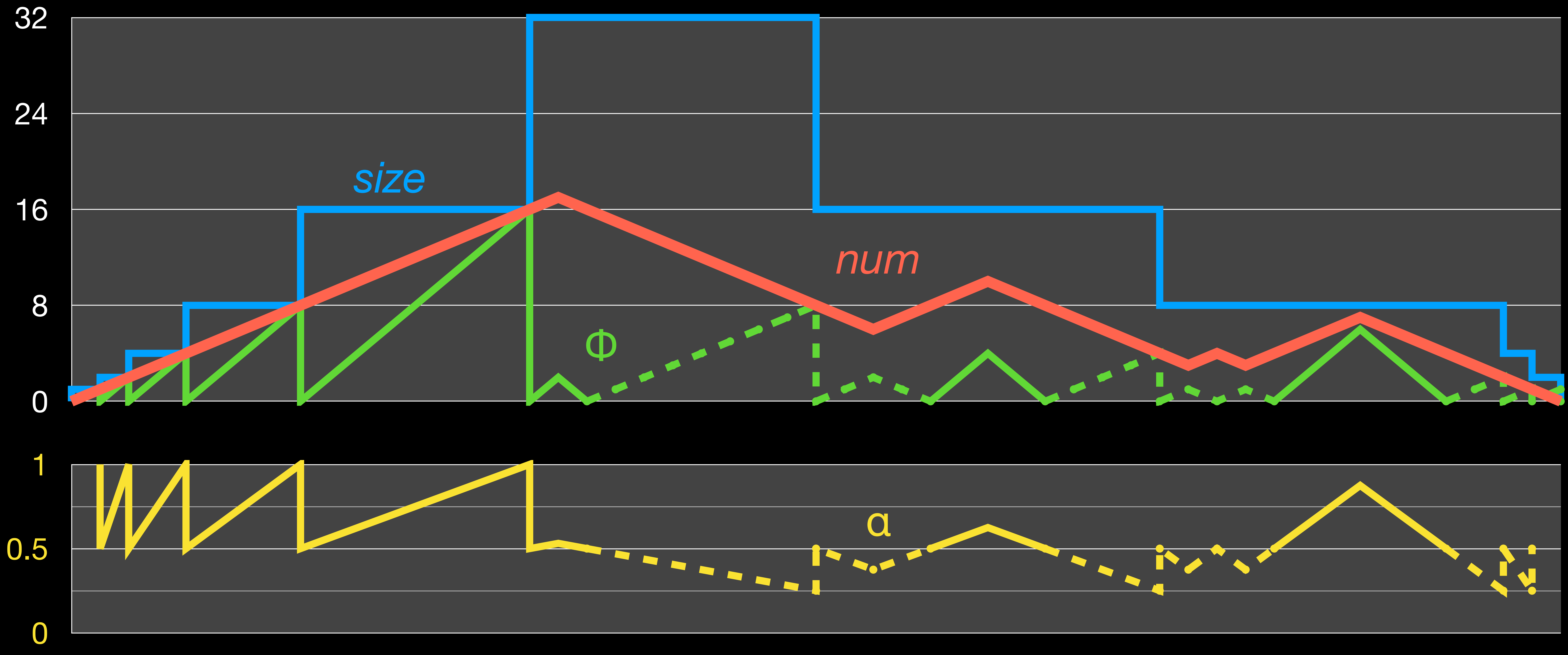


Table ¼-full:
表格满四分之一：
$\Phi = size/4 = num$

Table Expansion and Shrinking

表扩张和收缩



Sequence of Insertions and Deletions 插入、删除的序列

How to use Potential Analysis

如何使用势能法

1. Define a potential function
   Φ: states → potential energy = ℕ.

2. Calculate the amortized cost of every operation.

Slow operations should go from high-potential to low-potential states.

1. 定义势函数
   Φ：状态 → 是能 = ℕ。

2. 计算所有的操作的摊还代价。

慢速操作应
从高势能状态
变为低势能状态。

Exercise 16.2-5

练习 16.2-5

- Describe an efficient algorithm that, given a set $\{x_1, x_2, ..., x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.
- **Note:** it should not be difficult to come up with the algorithm. Try to write out the detailed correctness proof!

- 设计一个高效算法，对实数线上给定的一个点集 $\{x_1, x_2, ..., x_n\}$，求一个单位长度闭区间的集合，包含所有给定的点，并要求此机和最小。证明你的算法是正确的。
- **注意：** 想出这个算法应该不难。试着写出详细的正确性证明！

Exercise 17.1-1+

- If the set of stack operations included a MULTIPUSH(S, *k*, *i*) operation, which pushes *k* copies of *i* onto stack *S*, would the *O*(1) bound on the amortized cost of stack operations continue to hold?

- Give an analysis that shows that even in the presence of MULTIPUSH with a suitable amortized cost, PUSH/POP/MULTIPOP have *O*(1) amortized cost.

练习 17.1-1+

- 如果栈操作包括 MULTIPUSH(S, *k*, *i*) 操作，它将 *k* 个数据项 *i* 压入栈 *S* 中，那么栈操作摊还代价的界还是 *O*(1) 吗?

- 分析表明，即使存在具有适当摊还代价的MULTIPUSH， 那么PUSH/POP/MULTIPOP也具有 *O*(1)摊还代价。

Exercise 17.3-7

Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

- INSERT(S, x) inserts x into S.
- DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

练习 17.3-7

微动态整数多重集 S （允许包含重复制）设计一种数据结构，支持如下两个操作：

- INSERT(S, x) 将 x 插入 S 中。
- DELETE-LARGER-HALF(S) 将最大的 $\lceil |S|/2 \rceil$ 个元素从 S 中删除。

解释如何实现这种数据结构，使得任意 m 个 INSERT 和 DELETE-LARGER-HALF 操作的序列能在 $O(m)$ 时间内完成。还要实现一个能在 $O(|S|)$ 时间内输出所有元素的操作。

Exercise

- Write pseudocode for TABLE-DELETE (e.g. adapt from TABLE-INSERT)

练习

- 请写TABLE-DELETE的伪代码（可以适应TABLE-INSERT的）。