Algorithm Design and Analysis

算法设计与分析

David N. JANSEN, Bohua ZHAN
名 姓

詹博华，杨大卫

This week's content

这周的内容

- Today Wednesday:
  - Chapter 9: Selection
  - Chapter 11: Hashing (start)
    - 11.1–11.2
  - Exercises
- Tomorrow Thursday:
  - Exercise solutions
  - Chapter 11: Hashing (end)
    - 11.3–11.5

- 今天周三：
  - 第9章：
  - 第11章：
    - 11.1–11.2
  - 练习
- 明天周四：
  - 练习题解答
  - 第11章：
    - 11.3–11.5

Recap

- What is a heap?
  - Heap data structure: a binary tree that satisfies the max-heap property
- How does heapsort work?
  - Build a max-heap, and then extract the largest element repeatedly.
- How does quicksort work?
  - Partition the array into “small” and “large” elements, and then recurse.

- 什么是堆?
  - 堆的数据结构：满足最大堆性质的二叉树
- 堆排序是如何工作的?
  - 构建一个最大堆，然后重复提取最大元素。
- 快速排序是如何工作的?
  - 把数组划分为“小”和“大”元素，然后递归地排序这两个部分。

Selection

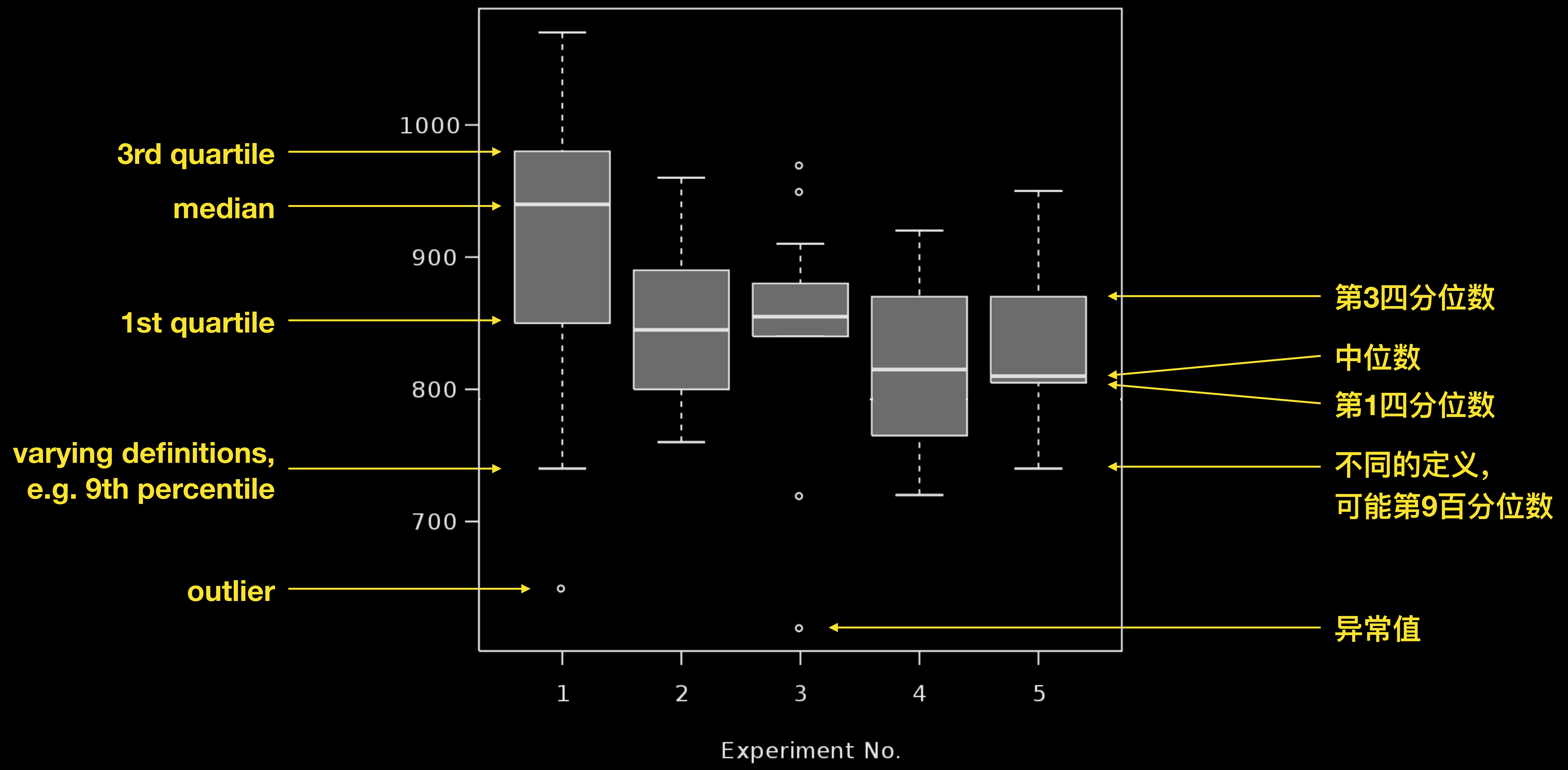
Order Statistic

顺序统计量

- ith order statistic of a data set = ith-smallest element of the set
  - 1st order statistic = minimum
  - nth order statistic = maximum (size of set = n)
  - $n/2$th order statistic = median
  - $n/4$th order statistic = 1st quartile = 25th percentile
  - $3n/4$th order statistic = 3rd quartile = 75th percentile

- 数据集的第 i 个顺序统计量 = 数据集的第 i 个最小元素。
  - 第一个顺序统计量 = 最小值。
  - 第 n 个顺序统计量 = 最大值（集合大小 = n）
  - 第 $n/2$ 个顺序统计量 = 中位数
  - 第 $n/4$ 个顺序统计量 = 第1四分位数 = 第25百分位数
  - 第 $3n/4$ 个顺序统计量 = 第3四分位数 = 第75百分位数

Example: Box Plot 例子：箱式图



By User:Schutz - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1501411

How to find an order statistic

如何找到顺序统计量

- simple method:
  1. sort data in array *A*[1] ... *A*[*n*]
  2. *A*[*i*] = ith order statistic

- requires time $\Omega(n \log n)$

- inefficient
  if only a few order statistics are needed

- e.g. minimum can be found in $O(n)$

- 简单的方式：
  1. 排序数组 *A*[1] ... *A*[*n*] 的数据
  2. *A*[*i*] = 第 *i* 个顺序统计量

- 运行时间 $\Omega(n \log n)$

- 效率低下的
  如果只需要一些订单统计信息

- 例如，最小值可以被找到在 $O(n)$ 中

How to find the minimum 如何找到最小值

MINIMUM(array A)
min = A[1]
for j := 2 **to** A.*length*
    if $A[j] < min$
        $min = A[j]$
return *min*

- requires A.*length*–1 comparisons
- impossible to use fewer comparisons
- finding any order statistic is in $\Omega(n)$

- 需要 A.*length*–1 比较
- 不可能使用更小的比较
- 找到顺序统计量在 $\Omega(n)$ 中

How to find minimum and maximum optimally

- Idea: form pairs
  - If $A[i] < A[j]$,
    then $A[i]$ is not the maximum
    and $A[j]$ is not the minimum

如何以最佳方式找到最小值和最大值

- 主意：创建对
  - 如果 $A[i] < A[j]$,
    那么 $A[i]$ 不可能最大值
    与 $A[j]$ 不可能最小值

How to find minimum and maximum optimally

如何以最佳方式找到最小值和最大值

```
MINIMUM-AND-MAXIMUM(array A)
if A.length is even
        if A[1] < A[2] then min = A[1]; max = A[2]; j = 3
        else  min = A[2]; max = A[1]; j = 3
else  min = A[1]; max = A[1]; j = 2
while j < A.length
        if A[j] < A[j+1] then
                if A[j] < min then min = A[j]
                if A[j+1] > max then max = A[j+1]
        else
                if A[j+1] < min then min = A[j+1]
                if A[j] > max then max = A[j]
        j = j + 2
return (min, max)
```

How to find minimum and maximum optimally

- Idea: form pairs
  - If $A[i] < A[j]$,
    then $A[i]$ is not the maximum
    and $A[j]$ is not the minimum
- 3 comparisons per 2 elements,
  apart from initialisation
  overall $\leq 3(n-1)/2$ comparisons
- Would it be even better to compare triples or quadruples?

如何以最佳方式 找到最小值和最大值

- 主意：创建对
  - 如果 $A[i] < A[j]$,
    那么 $A[i]$ 不可能最大值
    与 $A[j]$ 不可能最小值
- 每2个元素进行3次比较，
  除了初始化
  总计 $\leq 3(n-1)/2$ 比较
- 比较三胞胎还是四胞胎更好吗?

How to find any order statistic optimally

- Idea: Check quicksort again...
- To find the *i*th order statistic, we only need to find *A*[*i*]
- After PARTITION, we know whether the pivot is before, at or after position *i*
- Only recurse into part that contains *A*[*i*]

如何以最佳方式找到任何顺序统计量

- 注意：在看快速排序
- 为找到第 i 个双虚统计量仅需要找到 *A*[*i*]
- PARTITION 以后知道主元是在位置*i*之前、位置*i*处还是位置*i*之后
- 只要递归到包含 *A*[*i*] 的部分

Quickselect example 例子

2	9	12	1	4	11	8	6

- Find the 5th order statistic in this array
- 找到这个数组的第 5 个顺序统计量

Choose pivot 选主元

2	9	12	1	4	11	8	6

pivot/主元

$\underbrace{}_{\le \text{pivot}}$ $\underbrace{}_{\ge \text{pivot}}$

Find small element　　找到小的元素

2		9	12	1	4	11	8	6

pivot/主元

≤ pivot

≥ pivot

Find large element 找到大的元素

2	9	12	1	4	11	8	6
$\underbrace{\quad}_{\le \text{pivot}}$	$\underbrace{\quad}_{\ge \text{pivot}}$						pivot/主元

Find large element　　找到大的元素

Find small element 找到小的元素

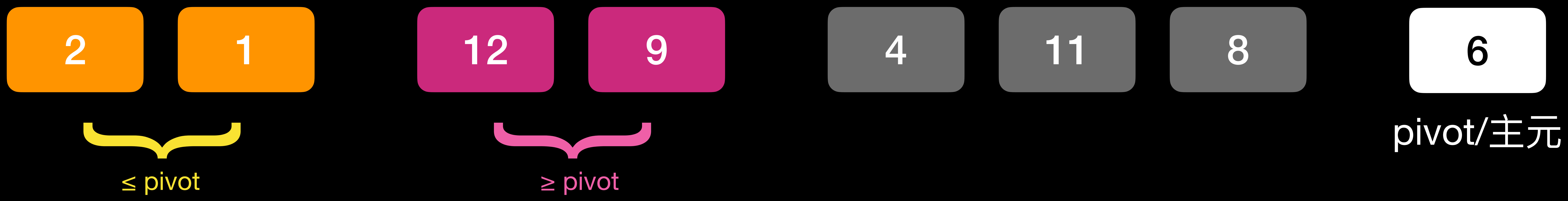
2 9 12 1 4 11 8 6

≤ pivot

≥ pivot

pivot/主元

Find small element 找到小的元素



Find small element 找到小的元素

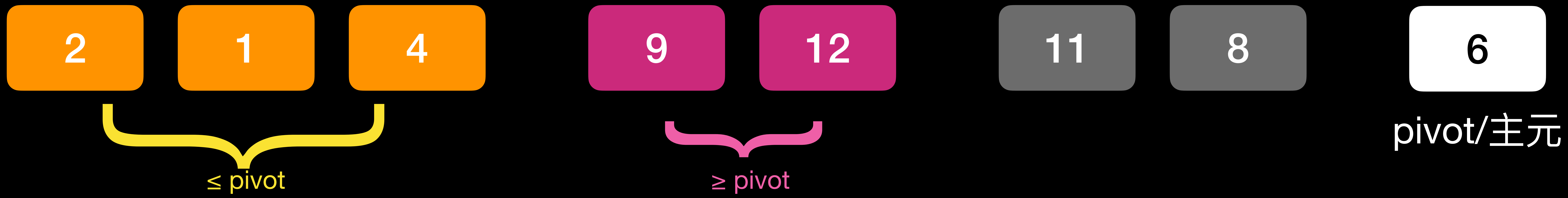
2 1 12 9 4 11 8 6

≤ pivot

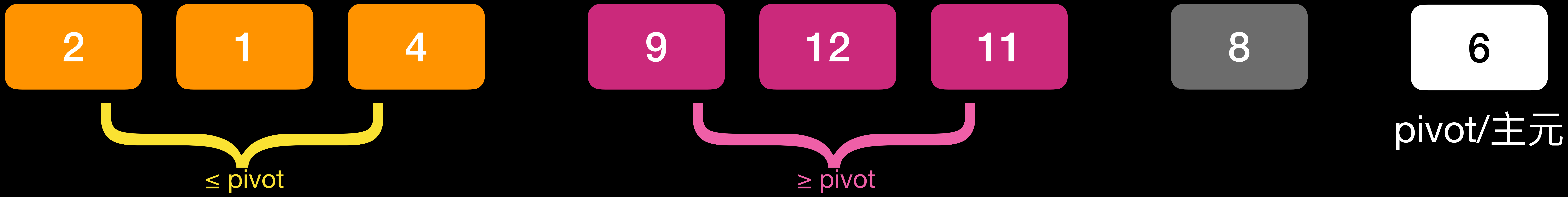
≥ pivot

pivot/主元

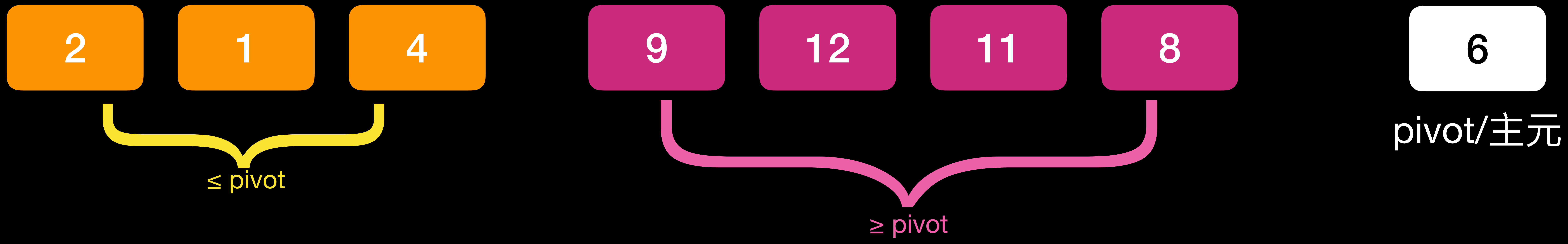
Find large element 找到大的元素



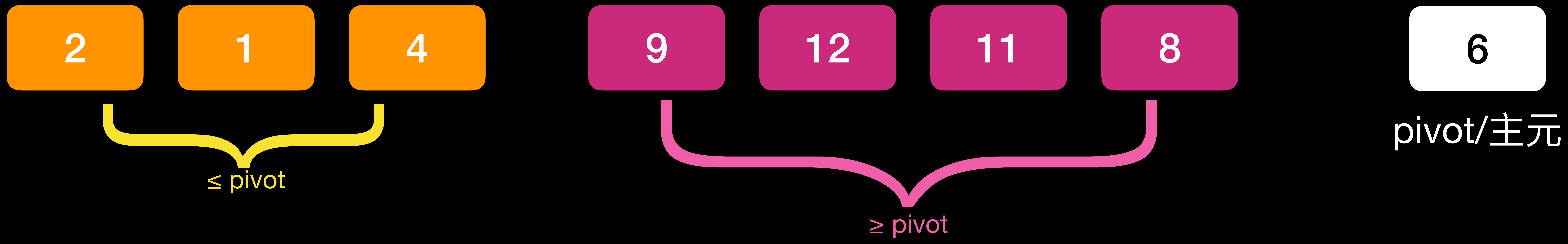
Find large element 找到大的元素



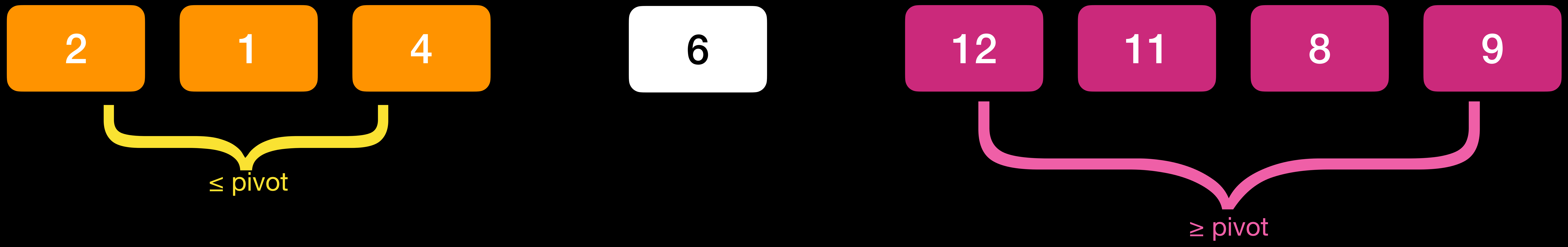
Find large element　　找到大的元素



Place pivot correctly 校正主元的位置



Partition finished 划分结束



These elements must be in positions 1–3

Pivot must be in position 4

These elements must be in positions 5–8

Partition finished 划分结束

2	1	4		6		12	11	8	9

These elements are 1st–3rd order statistics

Pivot is 4th order statistic

These elements are 5th–8th order statistic

$\textsc{Quickselect}(A, 5, 8, 1)$

Partition right part 划分结束

Partition right part 划分结束

2	1	4	6	8	9	12	11

This element is 5th order statistic

Pivot is 6th order statistic

These elements are 7th–8th order statistic

QUICKSELECT(A, 5, 5, 1)

Quickselect

QUICKSELECT(array A, p, r, i)
// Specification: output is the ith-smallest element in $A[p]$ ... $A[r]$
// Precondition: $1 \leq i \leq r-p+1$
if $p == r$ **then** **return** $A[p]$
q = PARTITION(A, p, r)
$k = q - p + 1$ // pivot is the kth-smallest element in $A[p]$ ... $A[r]$
if $i == k$ **then** **return** $A[q]$
else if $i < k$ **then** **return** QUICKSELECT(A, p, $q-1$, i)
else **return** QUICKSELECT(A, $q+1$, r, $i-k$)

Quickselect: Timing

Quickselect: 运行时间

- Sometimes the input is already sorted, then this version of QUICKSELECT is slow $O((n-i)^2)$
- We want to change the algorithm so every input order gives the same (expected) runtime
- Use randomization!

- 有时输入已经排序，则此版本的 QUICKSELECT速度慢 $O((n-i)^2)$
- 我们想更改算法因此每个输入顺序都提供相同的（期望的）运行时间
- 使用随机算法！

Average-case time　平均情况运行时间

- Average-case analysis of runtime: makes assumption about probability distribution of inputs
  - e.g. 7.2: average-case analysis of quicksort assumed every sequence has the same probability (but in reality, sorted input is more likely)
- Bad inputs lead to long runtime

- 平均情况运行时间：对输入的概率分布进行了假设
  - 例如 7.2 节：QUICKSORT的平均情况运行时间假设所有的循序有一样的概率（但实际上，排序输入更有可能）
- 坏的输入导致运行时间过长

Randomized Algorithms

随机算法

- Randomization := make some random choices as part of the algorithm
- enforces probability distribution over inputs
- e.g. choose pivot in Quicksort randomly (instead of always the last element)
- advantage: there is no “bad” input
- disadvantage: every input could occasionally lead to a slow execution

- 随机化：作在算法中
进行一些随机选择
- 强制输入的概率分布
- 例如，在Quicksort中随机选择主元
（而不是总是最后一个元素）
- 优点：没有“坏”输入
- 缺点：每次输入偶尔都会导致执行缓慢

Average / Expected Time 平均/期望运行时间

- Average Runtime: runtime depends on probability distribution of inputs
- Expected Runtime: runtime depends on internal random choices (but not on input)

- 平均运行时间：
运行时间取决于输入的概率分布
- 期望运行时间：
运行时间取决于内部随机选择
（但未于输入）

Randomized Quickselect 随机化的Quickselect

RANDOMIZED-QUICKSELECT(array A, p, r, i)
// Specification: output is the ith-smallest element in $A[p]$ ... $A[r]$
// Precondition: $1 \le i \le r\text{–}p+1$
if $p == r$ **then** **return** $A[p]$
Exchange $A[r]$ with $A[\text{RANDOM}(p \ldots r)]$
q = PARTITION(A, p, r)
$k = q - p + 1$ // pivot is the kth-smallest element in $A[p]$ ... $A[r]$
if $i == k$ **then** **return** $A[q]$
else if $i < k$ **then return** RANDOMIZED-QUICKSELECT(A, p, q–1, i)
else **return** RANDOMIZED-QUICKSELECT(A, q+1, r, i–k)

Randomized Quickselect: Runtime

随机化的Quickselect：运行时间

- Assume that RANDOM(p ... r) produces random number in $\{p, p+1, ..., r\}$ every number with the same probability $1/(r-p+1)$
- same effect as: every input permutation has the same probability
- In the analysis, take care to depend only on RANDOM(p ... r), not on other properties of the input

Randomized Quickselect: Runtime

- RANDOMIZED-QUICKSELECT(A, p, r, i)

- Indicator random variable
  X_k := 1 if pivot is the kth order statistic
  X_k := 0 otherwise

- Expected value: $E[X_k] = 1/n$
  (because any element of $A[p]$ ... $A[r]$ can be the pivot with the same probability)

Randomized Quickselect: Runtime

- $T(n) \leq \sum_{k=1}^{n} X_k \cdot \max\{T(k-1), T(n-k)\} + O(n)$

  $= \sum_{k=1}^{n} X_k \cdot T(\max\{k-1, n-k\}) + O(n)$

- Careful: we could think that $T(k-1)$ is more likely if k is large, but that assumes a probability distribution over the input i.

Randomized Quickselect: Runtime

- $T(n) \leq \sum_{k=1}^{n} X_k \cdot \max\{T(k-1), T(n-k)\} + O(n)$

  $= \sum_{k=1}^{n} X_k \cdot T(\max\{k-1, n-k\}) + O(n)$

- $E[T(n)] \leq E[\sum_{k=1}^{n} X_k \cdot T(\max\{k-1, n-k\}) + O(n)]$

  $= \sum_{k=1}^{n} E[X_k \cdot T(\max\{k-1, n-k\})] + O(n)$

  $= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max\{k-1, n-k\})] + O(n)$

$\max\{k-1, n-k\} = k-1$ if $k \geq (n+1)/2$
$\max\{k-1, n-k\} = n-k$ if $k \leq (n+1)/2$

Randomized Quickselect: Runtime

- $E[T(n)] \leq \sum_{k=1}^{n} E[X_k] \cdot E[T(\max\{k-1, n-k\})] + O(n)$

  $\leq 2 \sum_{k=\lceil (n+1)/2 \rceil}^{n} 1/n \cdot E[T(k-1)] + O(n)$

- Now assume $E[T(n)] = O(n)$
  and prove by substitution:
  Let $E[T(n)] \leq cn$ for some constant c.
  Further assume $O(n) \leq an$ (for large n).

- Also assume $E[T(n)] = O(1)$ for small n.

- 假设$E[T(n)] = O(n)$
  使用代入法证明：
  假设存在c这样$E[T(n)] \leq cn$。

- 以外假设$E[T(n)] = O(1)$如果n为小。

Randomized Quickselect: Runtime

- $E[T(n)] \le 2 \sum_{k=\lceil (n+1)/2 \rceil}^{n} 1/n \cdot E[T(k-1)] + an$

$$\le an + 2c/n \sum_{k=\lceil (n+1)/2 \rceil}^{n} k-1 = an + 2c/n \sum_{k=\lceil (n+1)/2 \rceil - 1}^{n-1} k = an + 2c/n \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil (n+1)/2 \rceil - 2} k \right)$$

$$= an + 2c/n \left(n(n-1)/2 - (\lceil (n+1)/2 \rceil - 1)(\lceil (n+1)/2 \rceil - 2)/2 \right)$$

$$\le an + c/n \left(n(n-1) - (n-1)(n-3)/4 \right)$$
$$= an + c/n \left(n^2 - n - (n^2 - 4n + 3)/4 \right)$$
$$= an + c \left(3n/4 - 3/n \right)$$
$$\le cn + \underbrace{an - cn/4}$$

needs to be ≤ 0

Randomized Quickselect: Runtime

- It remains to be proven: $an - cn/4 \leq 0$.
- This is the case if $a - c/4 \leq 0$, i.e. $4a < c$.
- Overall, we have proven that expected runtime of Randomized-Quickselect
  $= E[T(n)] \leq cn$ for some constant c, for large n,
  so $E[T(n)] = O(n)$.

Worst-case $O(n)$?

- Is it possible to have a selection algorithm in worst-case $O(n)$?
- main trick:
  find an approximation of the median
  and use it as pivot

Worst-case *O*(*n*)

- Detailed idea:
  1. Divide array into groups of 5 elements
  2. Find median of every group (using INSERTION-SORT or similar)
  3. Select median of medians recursively
  4. Use the found median-of-medians as pivot in PARTITION
  5. Recurse further (like QUICKSELECT)

Worst-case $O(n)$

16 13 12 18 2 3 10 17 15 8 5 4 9 14 11 1 6 7

Form groups of five elements each:

16 13 12 18 2 | 3 10 17 15 8 | 5 4 9 14 11 | 1 6 7

Sort every group:

2 12 13 16 18 | 3 8 10 15 17 | 4 5 9 11 14 | 1 6 7

Take the median of every group, and select their (lower) median:

6 | 9 | 10 | 13

Using this median-of-medians, we already know that at least approx. 30% are smaller and at least approx. 30% are larger:

2 12 13 16 18 | 3 8 10 15 17 | 4 5 9 11 14 | 1 6 7

Worst-case *O*(*n*)

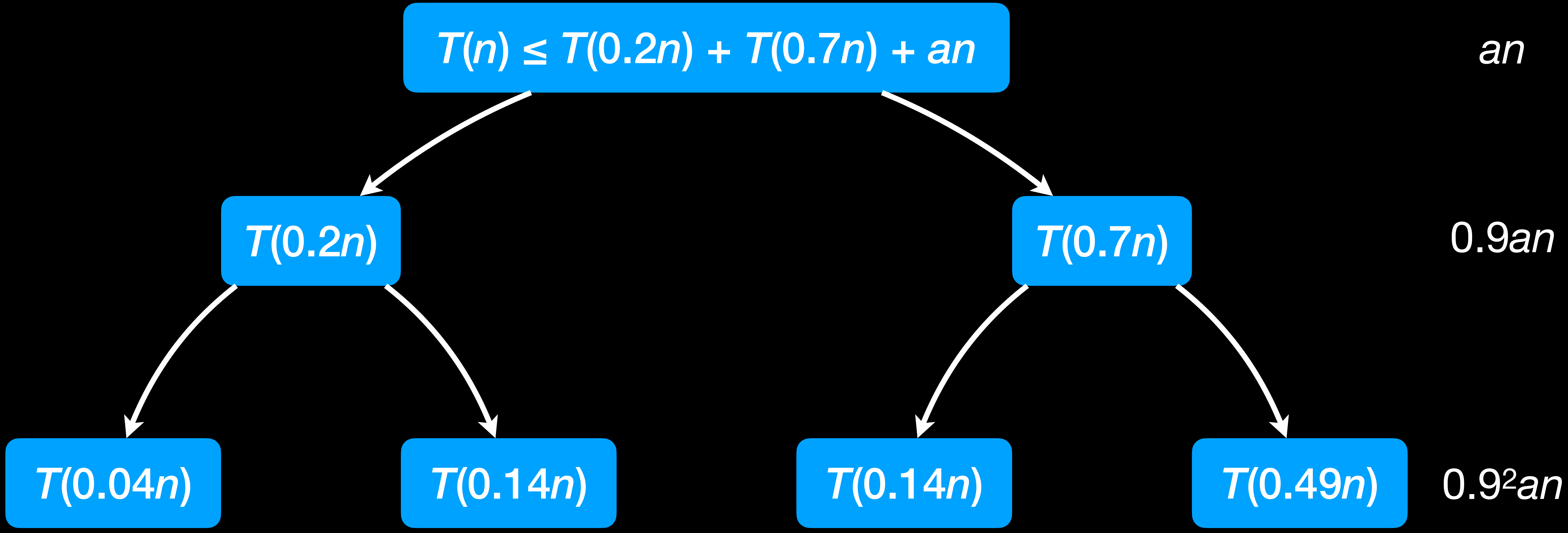
- For the median-of-medians, we have:
  - The group in which the median-of-median is, contains 2 larger elements
  - Every 5-element group with a larger median contains 3 larger elements
  - The last group contains 1–3 larger elements.
- Detailed computation shows that at least $0.3n - 6$ elements are larger
  ⤷ partition cannot be very bad

Worst-case $O(n)$

- The recurrence is something like
  $T(n) \leq T(0.2n) + T(0.7n) + O(n)$

- Prove $T(n) \leq T(0.2n) + T(0.7n) + an$
  for some constant a
  using the recursion tree method.

Worst-case $O(n)$



Worst-case $O(n)$

- The recurrence is something like
  $T(n) \leq T(0.2n) + T(0.7n) + O(n)$

- Prove $T(n) \leq T(0.2n) + T(0.7n) + an$
  for some constant a
  using the recursion tree method.

- So,
  $T(n) \leq an \sum_{i=0}^{\infty} 0.9^i = 10an = O(n)$.

- This is only a proof sketch!

Hashing 散列表

Dictionary

- (dynamic) dictionary. Operations:
  - insert
  - search
  - delete
- Example: identifiers (variable names) in a program:
  declare = insert,
  access = search,
  end of scope = delete.

- （动态的）字典，需要的操作：
  - insert
  - search
  - delete
- 例子：程序中的标识符（变量名）：
  声明 = insert,
  访问 = search,
  作用域的末尾 = delete.

How to implement a dictionary?

如何实现字典?

- Direct access:
  identifier = array index
  - advantage: fast
  - disadvantage: uses much memory
- feasible if there are few possible identifiers
  - BASIC: variable name = 1 or 2 letters
    $26 + 26^2 = 702$ possible variable names
  - C99: at least 31 characters are relevant
    $> 5 \cdot 10^{55}$ possible variable names

- 直接访问：
  标识符 = 数组索引
  - 优点：速度快
  - 缺点：占用大量内存
- 如果可能的标识符很少，则可行
  - BASIC：变量名 = 1或2字母
  - C99: 最少31字母

How to implement a dictionary?

如何实现字典?

- Hash access:
  h(identifier) = array index
  for a suitable function *h*: {identifiers} –> {array indices}

  - advantage: fast (if *h* is simple)
    uses moderate memory

  - disadvantage:
    if |{identifiers}| > |{array indices}|,
    h cannot be injective.
    ➥ need to resolve conflicts

Example

- Hash the values AN, AZ, V, and AK with:
  $h(k)$ = ((23k + 88) mod 101) mod 16
-

- AN --> 1*26 + 14 = 40
  $h(40)$ = (1008 mod 101) mod 16 = 3
- AZ --> 1*26 + 26 = 52
  $h(52)$ = (1284 mod 101) mod 16 = 8
- V --> 22
  h(22) = (594 mod 101) mod 16 = 9
- AK --> 1*26 + 11 = 37
  h(37) = (939 mod 101) mod 16 = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			AN					AZ	V					AK	

How to handle collisions?

- Chaining: every array entry stores a linked list of identifiers
- Reduce probability of collisions
  ➥ select h wisely
- Open addressing: if an array entry is occupied, calculate alternative index
- Perfect hashing: For a fixed set of identifiers, select h so that there are no collisions at all

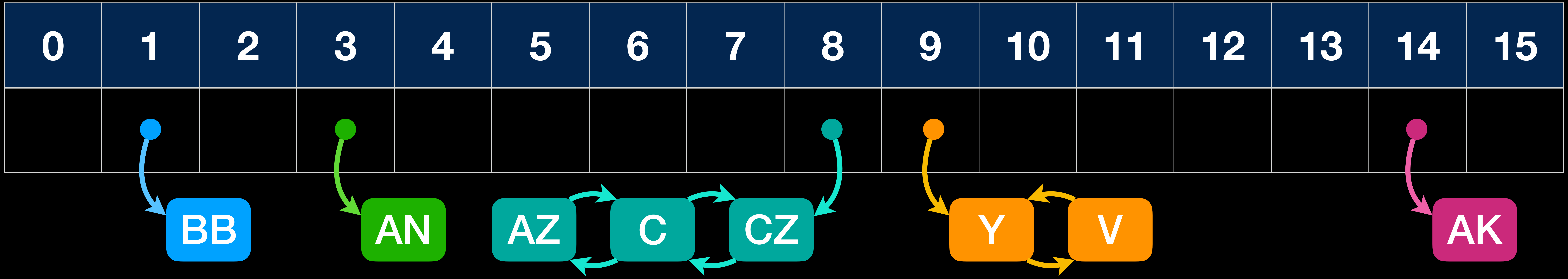
Chaining

- A hash table entry contains not one element, but (a pointer to) a list of elements

- INSERT(T, x)
  Insert x at the head of list $T[h(x.key)]$
- SEARCH(T, k)
  Linearly search for key k in list $T[h(k)]$
- DELETE(T, x)
  Delete x from its list (namely $T[h(x.key)]$)

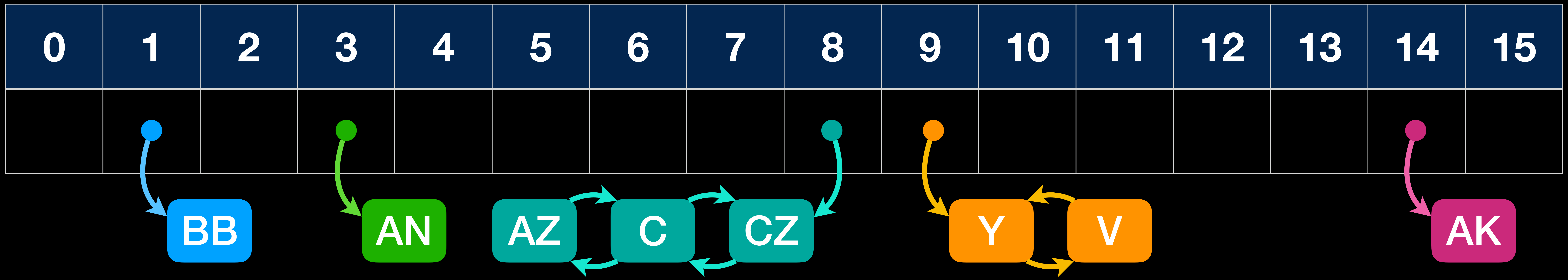
Example

- Add further values to the hash table:
  - BB --> 2*26 + 2 = 54, $h(54) = 1$
  - C --> 3, $h(3) = 8$
  - CZ --> 3*26 + 26 = 104, $h(104) = 8$
  - Y --> 25, $h(25) = 9$



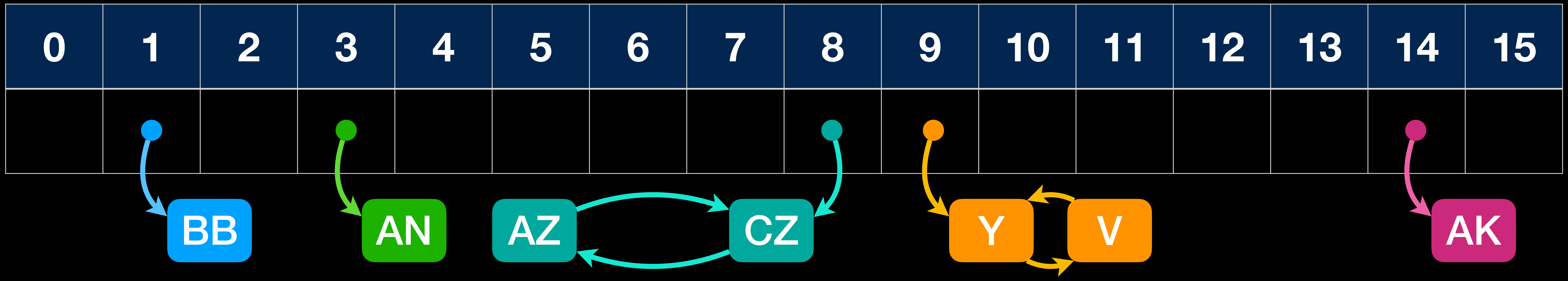
Example

- SEARCH(*T*, “AN”) --> search through *T*[3]
- SEARCH(T, “V”) --> search through *T*[9]
- DELETE(T, “C”) --> see next slide



Example

- SEARCH(T, “AN”) --> search through $T[3]$
- SEARCH(T, “V”) --> search through $T[9]$
- DELETE(T, “C”) --> see this slide



Timing Analysis

- Assumption: simple uniform hashing: Input of keys k is such that $h(k)$ is uniformly distributed over all array indices.
- Allows to calculate average-case time
- Load factor $\alpha = n/m$ = elements stored / number of places in array
- INSERT and DELETE need constant time (DELETE assumes doubly-linked lists)

Timing Analysis

- Unsuccessful search for key *k*:
  searches to the end of list *T*[*h*(*k*)].

- Simple uniform hashing
  ==> any *h*(*k*) is equally likely
  ==> average length of list is α
  ==> average running time = O(1 + α).

Timing Analysis

- Successful search for key *k*:
  searches (non-empty) list $T[h(k)]$.

- How many elements need to be searched?
  elements inserted after the one w/key *k*.

Timing Analysis

- Successful search for key k:
  searches (non-empty) list $T[h(k)]$.

- average number of elements searched

$$= \sum_{i=1}^{n} \text{Prob}(k \text{ was inserted as } i\text{th element}) \cdot (\text{number of elements in this case})$$

$$= \sum_{i=1}^{n} 1/n \left[1 + \sum_{j=i+1}^{n} \text{Prob}(\text{element } j \text{ has hash } k)\right]$$

$$= \sum_{i=1}^{n} 1/n \left[1 + \sum_{j=i+1}^{n} 1/m\right] = 1 + (n-1)/2m \leq 1 + n/2m = 1 + \alpha/2 = O(1 + \alpha).$$

Exercises 练习

7.3-1

- Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

为什么我们分析随机化算法的期望运行时间，而不是其最坏运行时间呢?

Problem 7-4

(Stack depth for Quicksort)
The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure.
This technique, called tail recursion, is provided automatically by good compilers.

（快速排序的栈深度） 7.1 节中的 QUICKSORT 算法包含了两个对其自身的递归调用。在调用 PARTITION 后，QUICKSORT 分别递归调用了左边的子数组和右边的子数组。QUICKSORT 中的第二个递归调用并不是必须的。我们可以用一个循环控制结构来代替它。这一技术称为**尾递归**，好的编译器都提供这一功能。考虑下面这个版本的快速排序，它模拟了尾递归情况：

Problem 7-4

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
while p < r
    // Partition and sort left subarray.
    q = PARTITION(A, p, r)
    TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
    p = q + 1
```

Problem 7-4

a. Argue that TAIL-RECURSIVE-QUICKSORT(A, 1, A.length) correctly sorts array A.

a. 证明：TAIL-RECURSIVE-QUICKSORT(A，1，$A.length$)能正确地对数组 A 进行排序。

Compilers usually execute recursive procedures by using a stack that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is pushed onto the stack; when it terminates, its information is popped. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires O(1) stack space. The stack depth is the maximum amount of stack space used at any time during a computation.

编译器通常使用**栈**来存储递归执行过程中的相关信息，包括每一次递归调用的参数等。最新调用的信息存在栈的顶部，而第一次调用的信息存在栈的底部。当一个过程被调用时，其相关信息被**压入**栈中；当它结束时，其信息则被**弹出**。因为我们假设数组参数是用指针来指示的，所以每次过程调用只需要 $O(1)$ 的栈空间。**栈深度**是在一次计算中会用到的栈空间的最大值。

Problem 7-4

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is Θ(n) on an n-element input array.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is O(log n). Maintain the O(n log n) expected running time of the algorithm.

b. 请描述一种场景，使得针对一个包含 n 个元素数组的 TAIL-RECURSIVE-QUICKSORT 的栈深度是 $\Theta(n)$。

c. 修改 TAIL-RECURSIVE-QUICKSORT 的代码，使其最坏情况下栈深度是 $\Theta(\lg n)$，并且能够保持 $O(n \lg n)$的期望时间复杂度。

8.2-2

- Prove that COUNTING-SORT is stable.

试证明 COUNTING-SORT 是稳定的。

8.3-1

- Using Figure 8.3 as a model, illustrate the operation of radix sort on the following list of English words:

参照图 8-3 的方法，说明 RADIX-SORT 在下列英文单词上的操作过程：

COW, DOG, SEA, RUG, ROW,
MOB, BOX, TAB, BAR, EAR,
TAR, DIG, BIG, TEA, NOW, FOX.

9.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

证明：在最坏情况下，找到 n 个元素中第二小的元素需要 $n+\lceil \lg n \rceil-2$ 次比较。（提示：可以同时找最小元素。）

9.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

在算法 SELECT 中，输入元素被分为每组 5 个元素。如果它们被分为每组 7 个元素，该算法仍然会是线性时间吗？证明：如果分成每组 3 个元素，SELECT 的运行时间不是线性的。