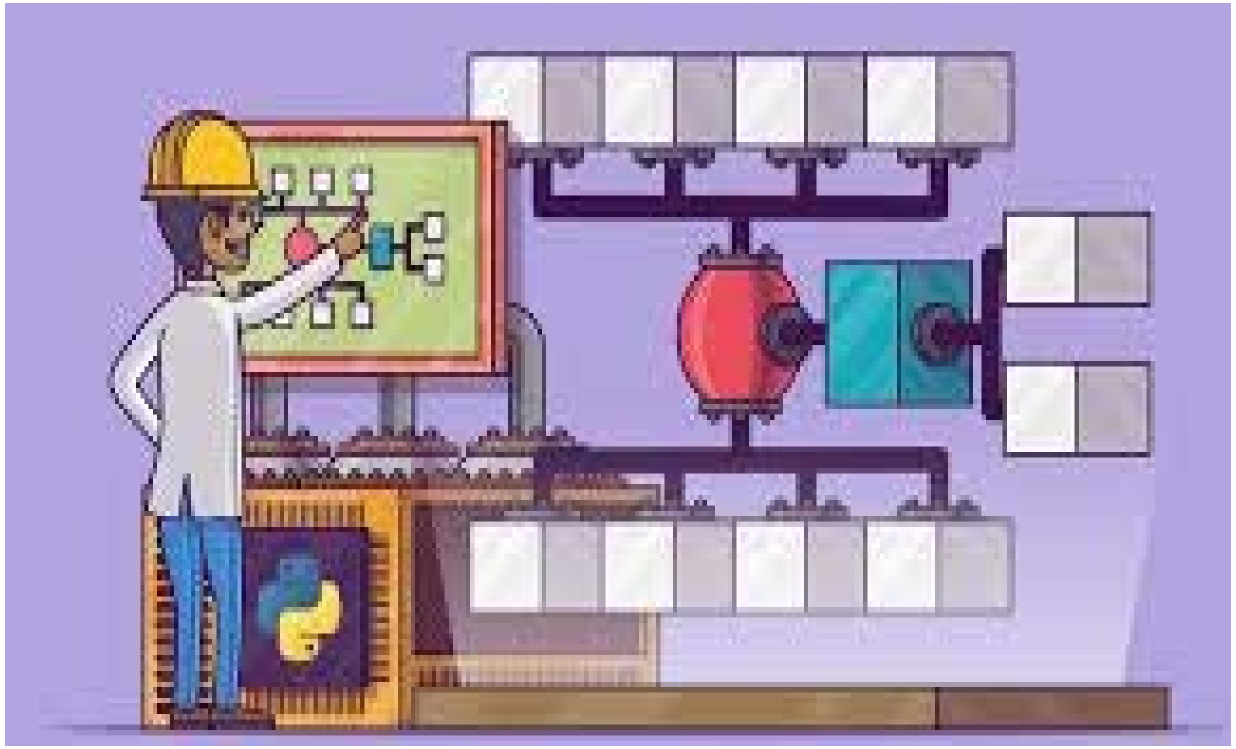
ساختمان داده ها



مدرس:
سمانه حسینی سمنانی

دانشگاه صنعتی اصفهان- دانشکده برق و کامپیوتر

مرتب سازی – sorting

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- Insertion sort:
  - takes $\theta(n^2)$ time in the worst case.
  - fast in-place sorting algorithm for small input sizes.
- Merge sort:
  - has a better asymptotic running time: $\theta(\text{nlogn})$
  - MERGE procedure it uses does not operate in place.

مرتب سازی – sorting

- Heapsort sorts:
  - sorts n numbers in place
  - in O(nlogn)
- Quicksort sorts:
  - n numbers in place,
  - but its worst-case running time is $\theta(n^2)$

مرتب سازی – sorting

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

مرتب سازی – sorting

- انواع الگوریتم های مرتب سازی
  - مقایسه ای و غیر مقایسه ای
  - پایدار و غیر پایدار

- Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts
- We will prove that a lower bound of $\theta(\mathrm{nlogn})$ on the worst-case running time of any comparison sort on n inputs,
- heapsort and merge sort are asymptotically optimal comparison sorts.

Quicksort

Divide: Partition (rearrange) the array $A[p \mathinner{..} r]$ into two (possibly empty) subarrays $A[p \mathinner{..} q-1]$ and $A[q+1 \mathinner{..} r]$ such that each element of $A[p \mathinner{..} q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \mathinner{..} r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p \mathinner{..} q-1]$ and $A[q+1 \mathinner{..} r]$ by recursive calls to quicksort.

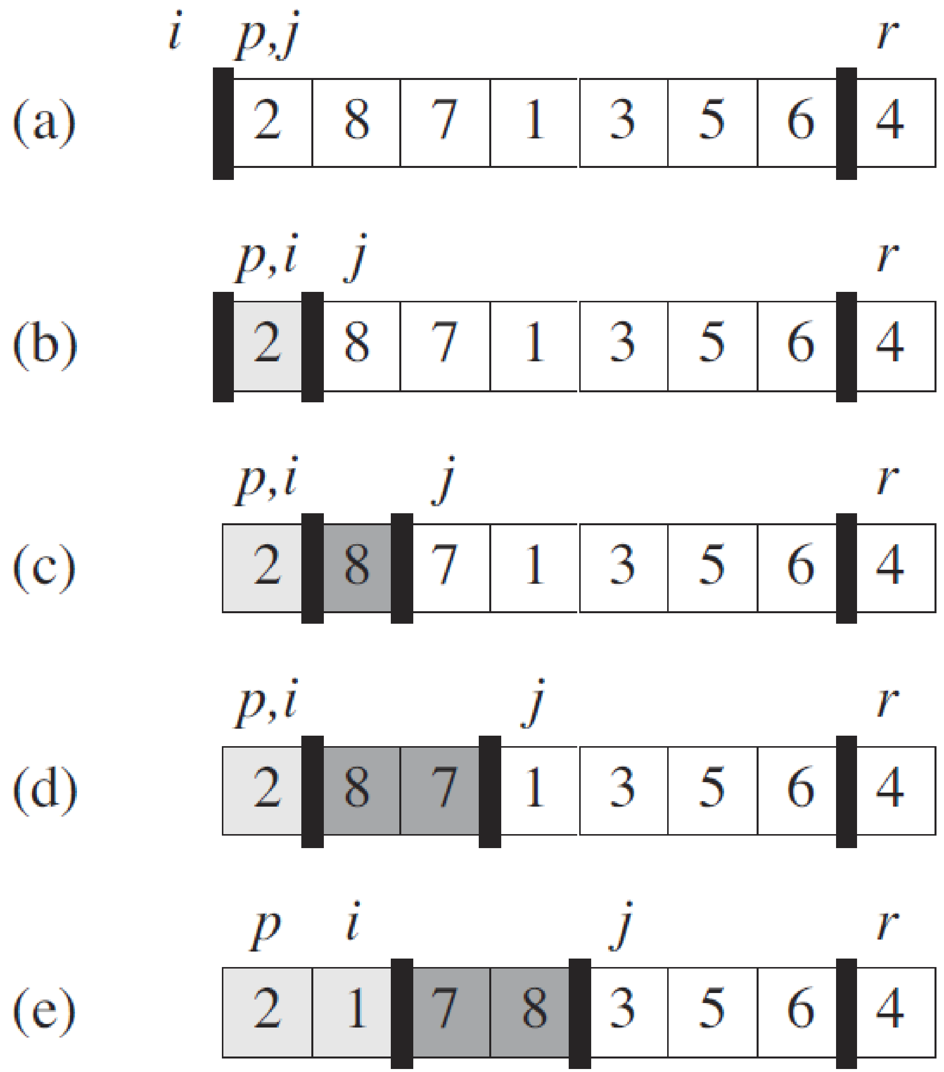
Quicksort

QUICKSORT(A, p, r)
1 **if** $p < r$
2 $\quad q = \text{PARTITION}(A, p, r)$
3 $\quad \text{QUICKSORT}(A, p, q - 1)$
4 $\quad \text{QUICKSORT}(A, q + 1, r)$

Quicksort

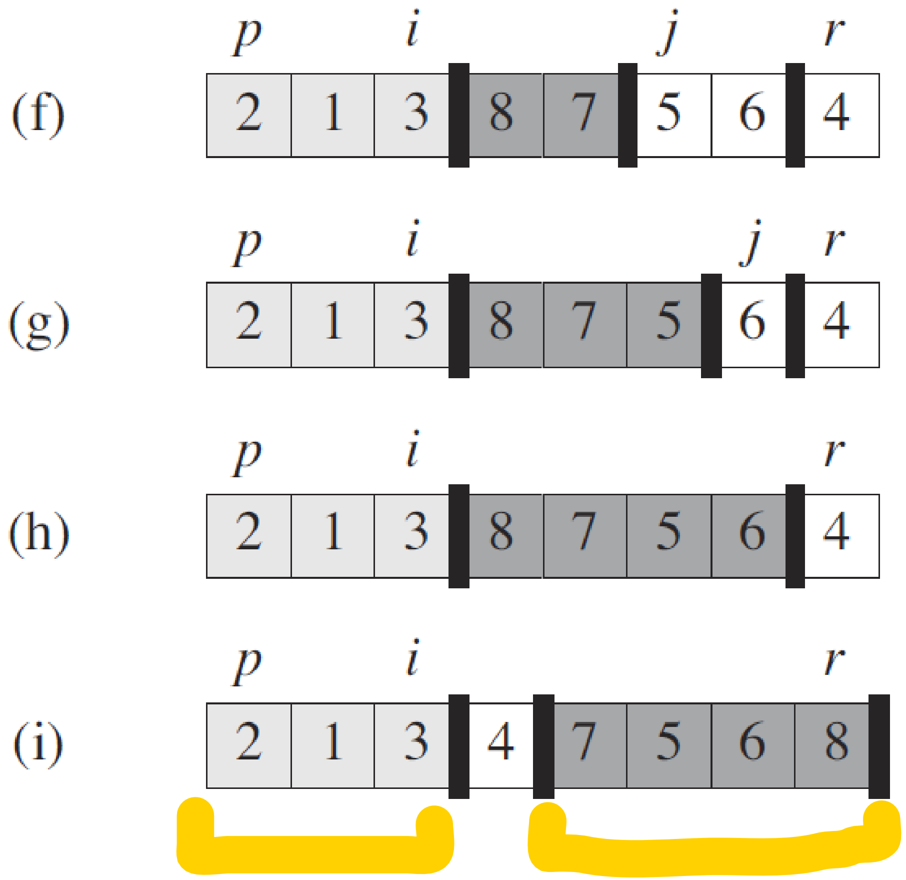
PARTITION(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



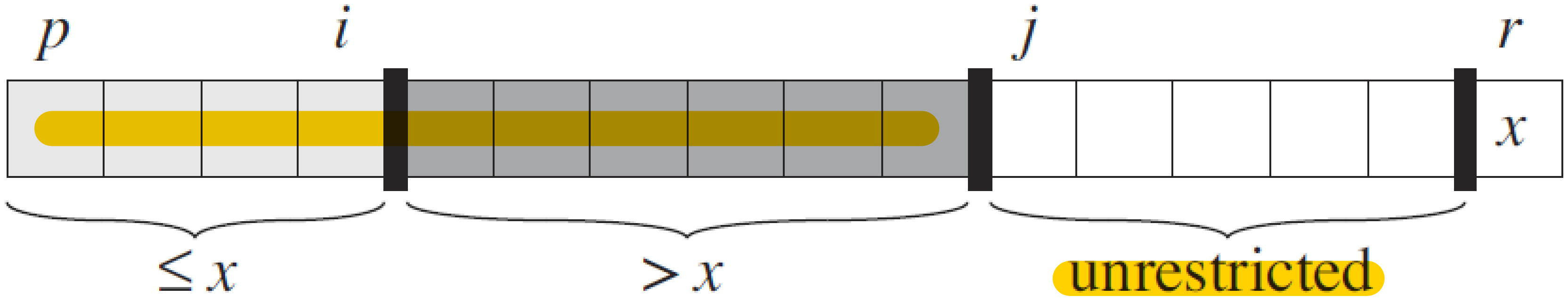
Quicksort

```
PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```



Quicksort

partitions the array into four (possibly empty) regions.



The running time of PARTITION on the subarray $A[p..r]$ is $\Theta(n)$, where $n = r - p + 1$.

Performance of quicksort

Worst-case partitioning:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

$$\Theta(n^2).$$

Performance of quicksort

Best-case partitioning:

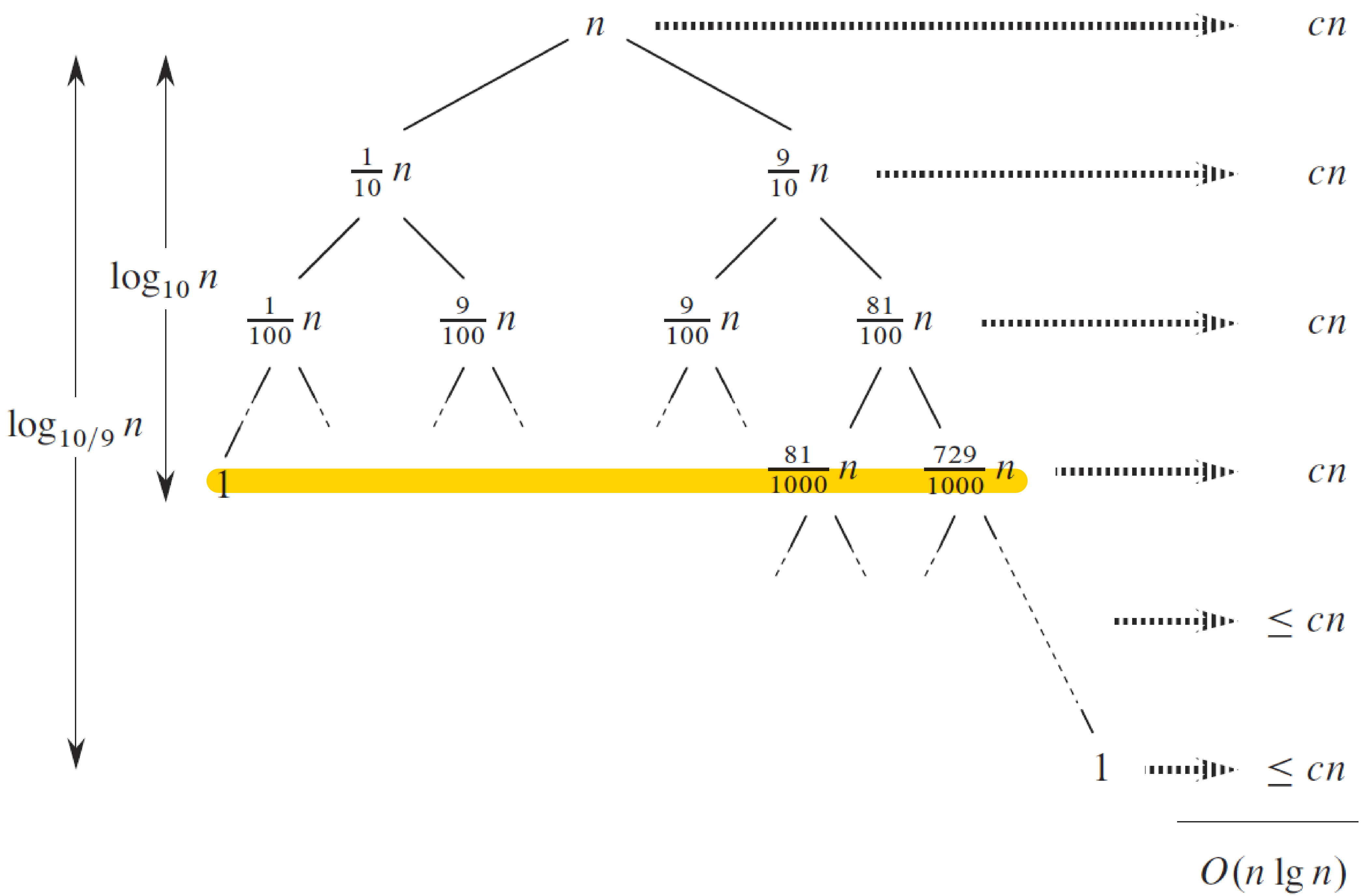
$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \Theta(n \lg n).$$

Performance of quicksort

- Balanced partitioning:
  - The average-case running time of quicksort is much closer to the best case than to the worst case
  - Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split,

$$T(n) = T(9n/10) + T(n/10) + cn$$



n
cn
$\frac{1}{10}n$
$\frac{9}{10}n$
cn
$\log_{10} n$
$\frac{1}{100}n$
$\frac{9}{100}n$
$\frac{9}{100}n$
$\frac{81}{100}n$
cn
$\log_{10/9} n$
1
$\frac{81}{1000}n$
$\frac{729}{1000}n$
cn
$\leq cn$
1
$\leq cn$
$O(n \lg n)$

A randomized version of quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

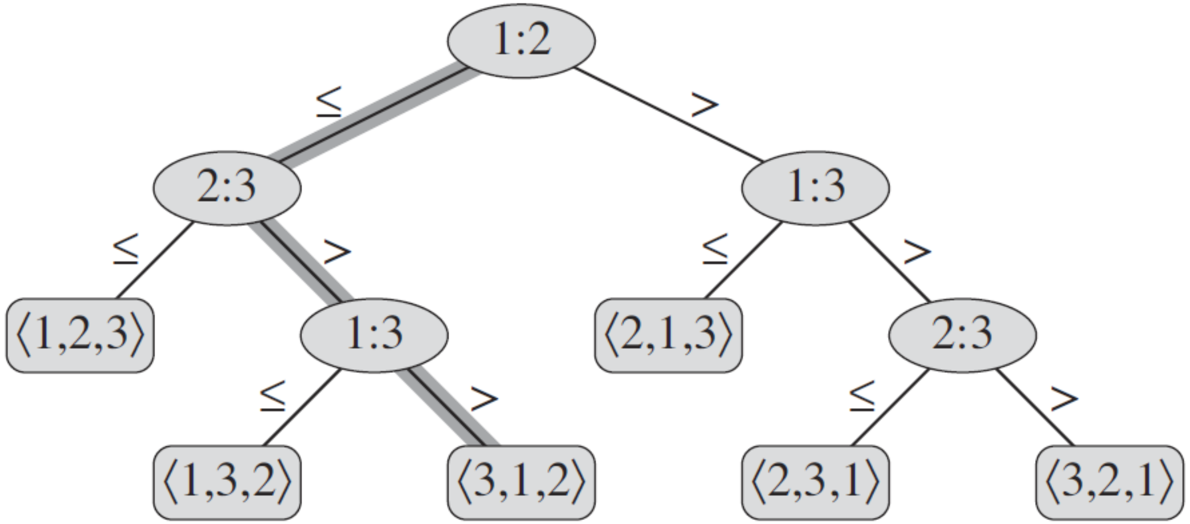
```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Lower bounds for sorting

Theorem 8.1

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Decision tree for insertion sort



An internal node annotated by $i : j$ indicates a comparison between a_i and a_j

$\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

The shaded path: $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$;

Decision tree for insertion sort

- The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf
- Each internal node indicates a comparison $a_i \leq a_j$
- When we come to a leaf, the sorting algorithm has established the ordering

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$$

- Any correct sorting algorithm must be able to produce each permutation of its input
- each of the *n!* permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct.

Decision tree for insertion sort

- each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort.
- l :reachable leaves corresponding to a comparison sort on n elements
- h: height of the tree

$$n! \leq l \leq 2^h$$

$$\begin{aligned} h &\geq \lg(n!) \\ &= \Omega(n \lg n) \end{aligned}$$

Sorting in Linear Time

- In all the previous algorithms (Merge sort, heapsort, Quicksort, Insertion sort):

the sorted order they determine is based only on comparisons between the input elements.

- We call such sorting algorithms ***comparison sorts***.
- $O(nlogn)$

Sorting in Linear Time

- Counting sort, radix sort, and bucket sort—that run in linear time.
- These algorithms use operations other than comparisons to determine the sorted order.

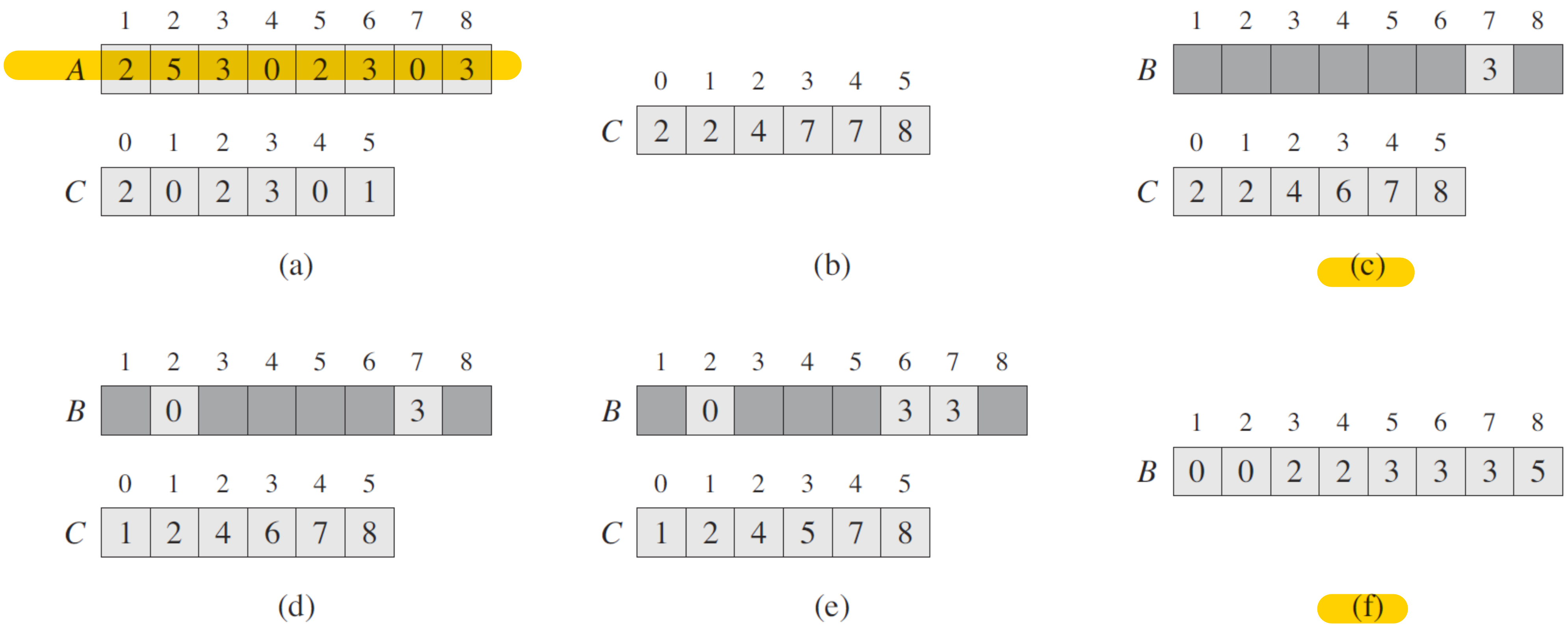
Counting sort

- assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.

Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Counting sort



Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
        C[i] = 0
for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
        C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] − 1
```

$\Theta(k + n)$.

$k = O(n)$,

$\Theta(n)$.

stable: numbers with the same value appear in the output array in the same order as they do in the input array.

Counting sort

- The property of stability is important:

1. when satellite data are carried around with the element being sorted.
2. Counting sort is often used as a subroutine in radix sort.

- In order for radix sort to work correctly, counting sort must be stable.