CS590/CPE590

Partitioning | Quickselect | Advanced Sorting

Kazi Lutful Kabir

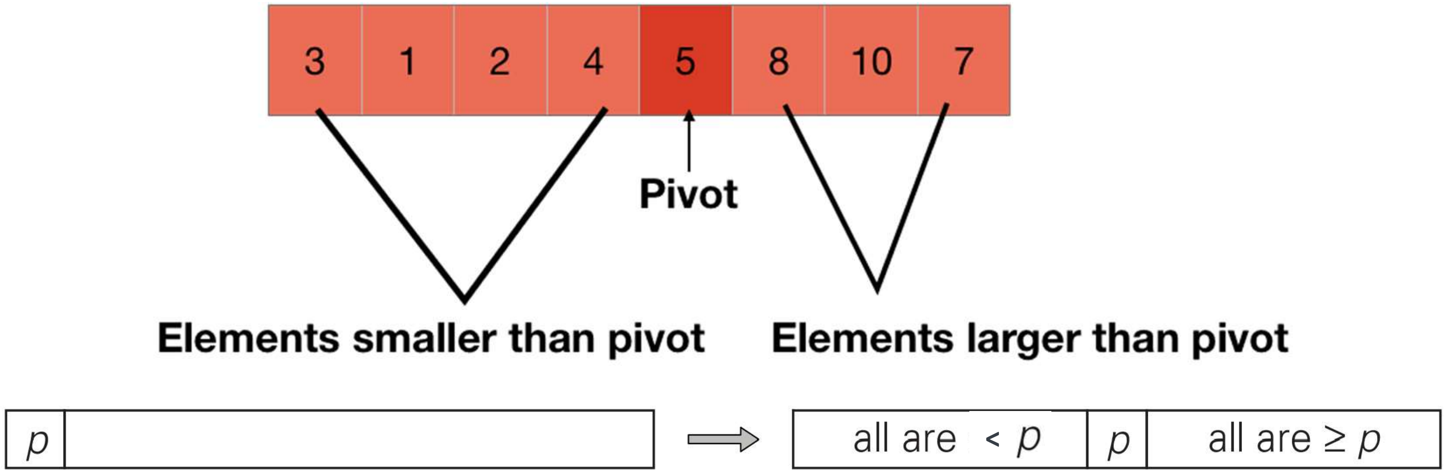
Spring 2023

Array Partitioning

- Moving all the elements that satisfies some criterion to one side of the array
- Most common practice: partitioning a numeric array based on a reference number (*pivot*) such that all the elements that are greater than or equal to that are on the right side and all the elements that are less than that are on the left side of it
- Partitioning of an array can be done **in-place** (by the rearrangement of the elements within the same given array without copying it to another array)
- Implementation and demonstration of a partitioning algorithm that takes linear time

Partitioning an Array : Key Idea

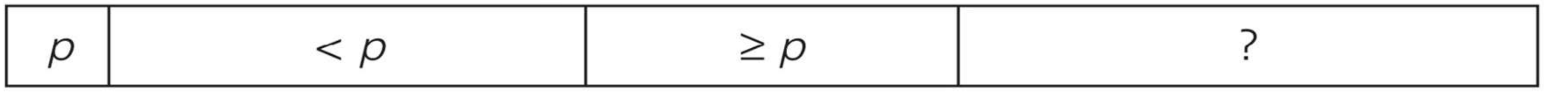
- Given an unordered Array, A [0……n-1]; re-arrange the elements of it around some value ***p*** such that:



- **Left part** contains all the elements smaller than ***p***, followed by the pivot ***p*** itself
- **Right part** having all the elements greater than or equal to ***p***

Lomuto Partitioning : Insights

- Let’s think of a subarray *$A[l..r]$ $(0 \le l \le r \le n - 1)$*
- Composed of 3 contiguous segments (listed in the order they follow pivot ***p***):
  -- A segment with elements known to be smaller than ***p***
  -- The segment of elements known to be greater than or equal to ***p***
  -- The segment of elements yet to be compared to ***p***

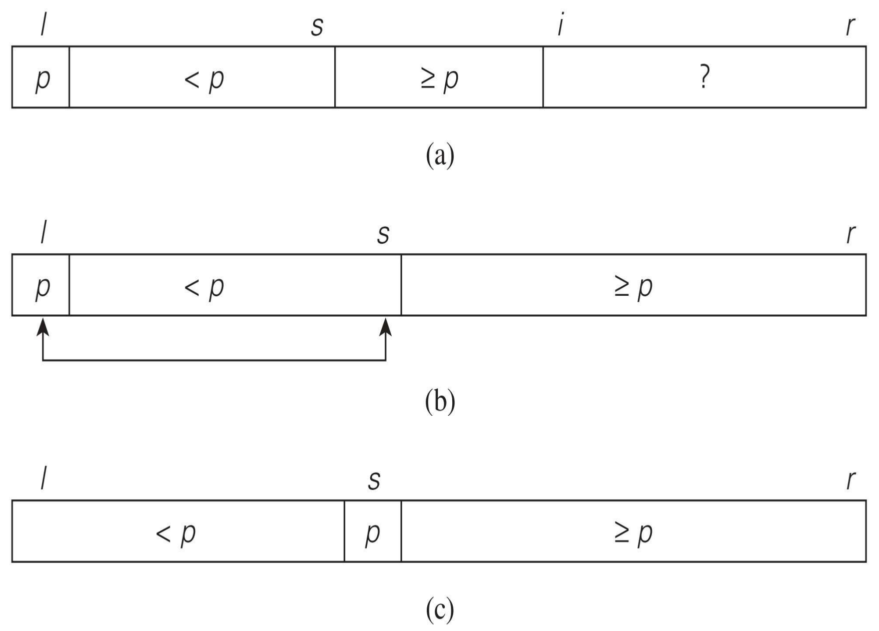


Lomuto Partitioning : Insights …

- Starting with *i = l+1* scan the subarray from left to right
- On each iteration, compares the first element in the unknown segment with ***p***:
  -- *A[i]* ≥ ***p***; *i* is incremented to expand the segment of the elements greater than or equal to ***p*** while shrinking the unprocessed

  -- *A[i]* < ***p***; segment of the elements smaller than ***p*** is expanded by incrementing *s*, *swap A[i] and A[s]*, increment *i* to point to the new first element of the shrunk unprocessed segment

  -- No unprocessed elements remain; swap the pivot with *A[s]* to achieve a desired partition



Lomuto Partitioning : Algorithm

```
LomutoPartition(A, l, r)
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
// indices l and r (l ≤ r)
//Output: Partition of A[l..r] and the new position s of the pivot
1.  p ← A[l]
2.  s ← l
3.  for i ← l + 1 to r do
4.      if A[i] < p
5.          s ← s + 1
6.          swap(A[s], A[i])
7.  swap(A[l], A[s])
8.  return s
```

Running time:
Partitioning an n-element array requires (n − 1) key comparisons => $\Theta(n)$

Lomuto Partitioning : Demonstration

```
5   7   2   6   4   9

5   7   2   6   4   9

5   2   7   6   4   9

5   2   7   6   4   9

5   2   4   6   7   9

5   2   4   6   7   9
```

```
1. p ← A[l]
2. s ← l
3. for i ← l + 1 to r do
4.     if A[i] < p
5.         s ← s + 1
6.         swap(A[s], A[i])
7. swap(A[l], A[s])
8. return s
```

```
4  2  5  6  7  9
```

Class Task 1

- Apply **Lomuto-partitioning** to the following list of numbers

  5, 24, 1, 13, 16, 12, 5

```
1. p ← A[l]
2. s ← l
3. for i ← l + 1 to r do
4.     if A[i] < p
5.         s ← s + 1
6.         swap(A[s], A[i])
7. swap(A[l], A[s])
8. return s
```

- Any relation between sorting and partitioning?

Class Task 1: Solution

- Apply **Lomuto-partitioning** to the following list of numbers

  5, 24, 1, 13, 16, 12, 5

```
1. p ← A[l]
2. s ← l
3. for i ← l + 1 to r do
4.     if A[i] < p
5.         s ← s + 1
6.         swap(A[s], A[i])
7. swap(A[l], A[s])
8. return s
```

- **i=1:**
  5, 24, 1, 13, 16, 12, 5
- **i=2,….,6:**
  5, 1, 24, 13, 16, 12, 5
- **1, 5, 24, 13, 16, 12, 5**
  s=1

- The pivot element is in the position where it would be if the entire array were sorted!

Class Task 2

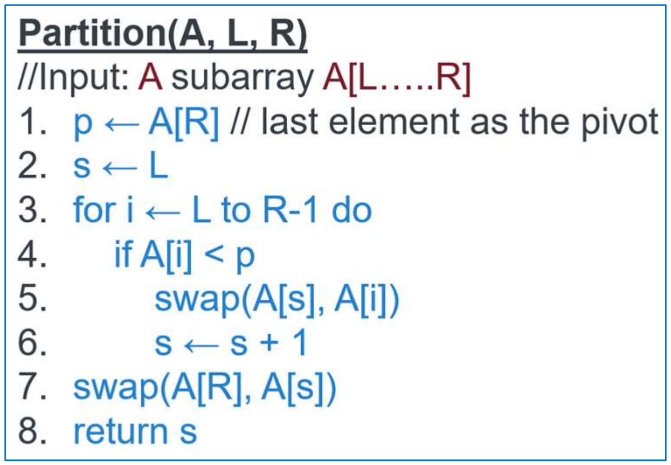
- What modifications do we need if we want to consider the last element as the pivot in the **Lomuto-partitioning** algorithm

```
1. p ← A[l]
2. s ← l
3. for i ← l + 1 to r do
4.     if A[i] < p
5.         s ← s + 1
6.         swap(A[s], A[i])
7. swap(A[l], A[s])
8. return s
```

- Any idea to generalize the procedure if we want to consider any element as the pivot?

Class Task 2: Solution

- What modifications do we need if we want to consider the last element as the pivot in the **Lomuto-partitioning** algorithm



```
Partition(A, L, R)
//Input: A subarray A[L.....R]
1.  p ← A[R] // last element as the pivot
2.  s ← L
3.  for i ← L to R-1 do
4.      if A[i] < p
5.          swap(A[s], A[i])
6.          s ← s + 1
7.  swap(A[R], A[s])
8.  return s
```

```
1.  p ← A[l]
2.  s ← l
3.  for i ← l + 1 to r do
4.      if A[i] < p
5.          s ← s + 1
6.          swap(A[s], A[i])
7.  swap(A[l], A[s])
8.  return s
```

- At first, swap the element with the first/last element (generalization)

Quickselect : Key Idea

- How can we take advantage of array partitioning to find the **k**-th smallest element in it?

- Let **s** be the partition's split position, i.e., the index of the array's element occupied by the pivot after partitioning.
  -- If **s = k − 1**, pivot p itself is obviously the **k**th smallest element, which solves the problem

  -- If **s > k − 1**, the kth smallest element in the entire array can be found as the **k**th smallest element in the left part of the partitioned array

  -- if **s < k − 1**, it can be found as the **(k − s)**th smallest element in its right part

Quickselect : Algorithm

```
**ALGORITHM** $Quickselect(A[l..r], k)$
    //Solves the selection problem by recursive partition-based algorithm
    //Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
    //          integer $k$ $(1 \le k \le r - l + 1)$
    //Output: The value of the $k$th smallest element in $A[l..r]$
    $s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
    if $s = k - 1$ return $A[s]$
    else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
    else $Quickselect(A[s+1..r], k - 1 - s)$
```

Quickselect : Example

Find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15

0	1	2	3	4	5	6	7	8
s	*i*							
4	1	10	8	7	12	9	2	15
	s	*i*						
4	1	10	8	7	12	9	2	15
	s						*i*	
4	1	10	8	7	12	9	2	15
		s					*i*	
4	1	2	8	7	12	9	10	15
		s						*i*
4	1	2	8	7	12	9	10	15
2	1	**4**	8	7	12	9	10	15

0	1	2	3	4	5	6	7	8
			s	*i*				
			8	7	12	9	10	15
				s	*i*			
			8	7	12	9	10	15
				s				*i*
			8	7	12	9	10	15
			7	**8**	12	9	10	15

Quickselect : Running Time

- How efficient is quickselect?
  Partitioning an n-element array always requires n − 1 key comparisons. If it produces the split that solves the selection problem without requiring more iterations, then for this best case we obtain **$C_{best}(n) = n - 1 \in \Theta(n)$**

- Unfortunately, the algorithm can produce an extremely unbalanced partition of a given array, with one part being empty and the other containing n − 1 elements. In the worst case, this can happen on each of the n − 1 iterations. (For a specific example of the worst-case input, consider, say, the case of k = n and a strictly increasing array)

$$C_{worst}(n) = (n - 1) + (n - 2) + . . . + 1 = (n - 1)n/2 \in \Theta(n^2)$$

- The partition-based algorithm solves a somewhat more general problem of identifying the **k** smallest and **(n − k)** largest elements of a given list, not just the value of its kth smallest element

Quick Sort

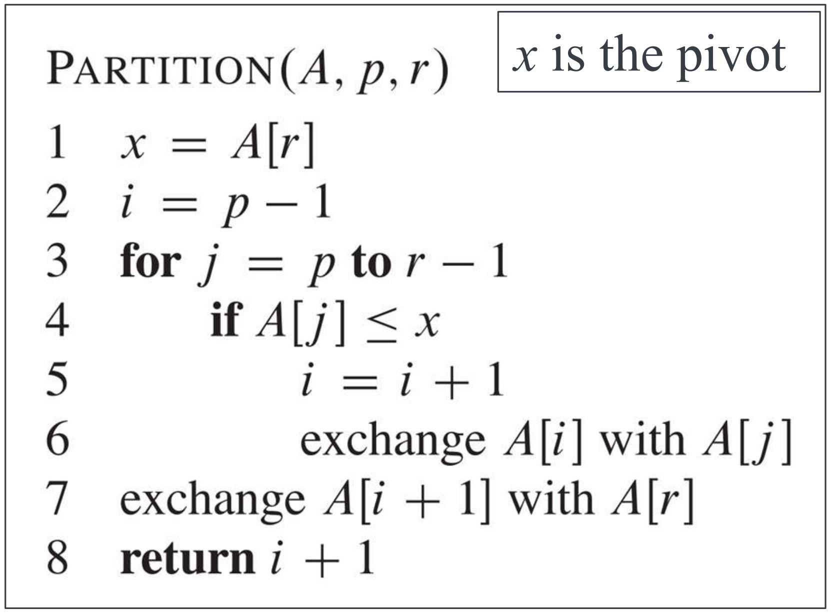
A sorting algorithm based on the divide-and-conquer strategy.

- **Divide**: partition $A[p \dots r]$ into two subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$, such that each element in $A[p \dots q-1]$ is $\leq$ $A[q]$ and $A[q]$ is $\leq$ in each element in $A[q+1 \dots r]$.
- **Conquer**: We sort the two subarrays by recursive calls to QUICKSORT.
- **Combine**: No need to combine the subarrays, because they are sorted in place.

```
QUICKSORT$(A, p, r)$
1    **if** $p < r$
2    **then** $q \leftarrow$ PARTITION$(A, p, r)$
3         QUICKSORT$(A, p, q-1)$
4         QUICKSORT$(A, q+1, r)$
```

*Initial call is QUICKSORT$(A, 1, n)$

Quick Sort



```
PARTITION$(A, p, r)$        $x$ is the pivot
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \leq x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

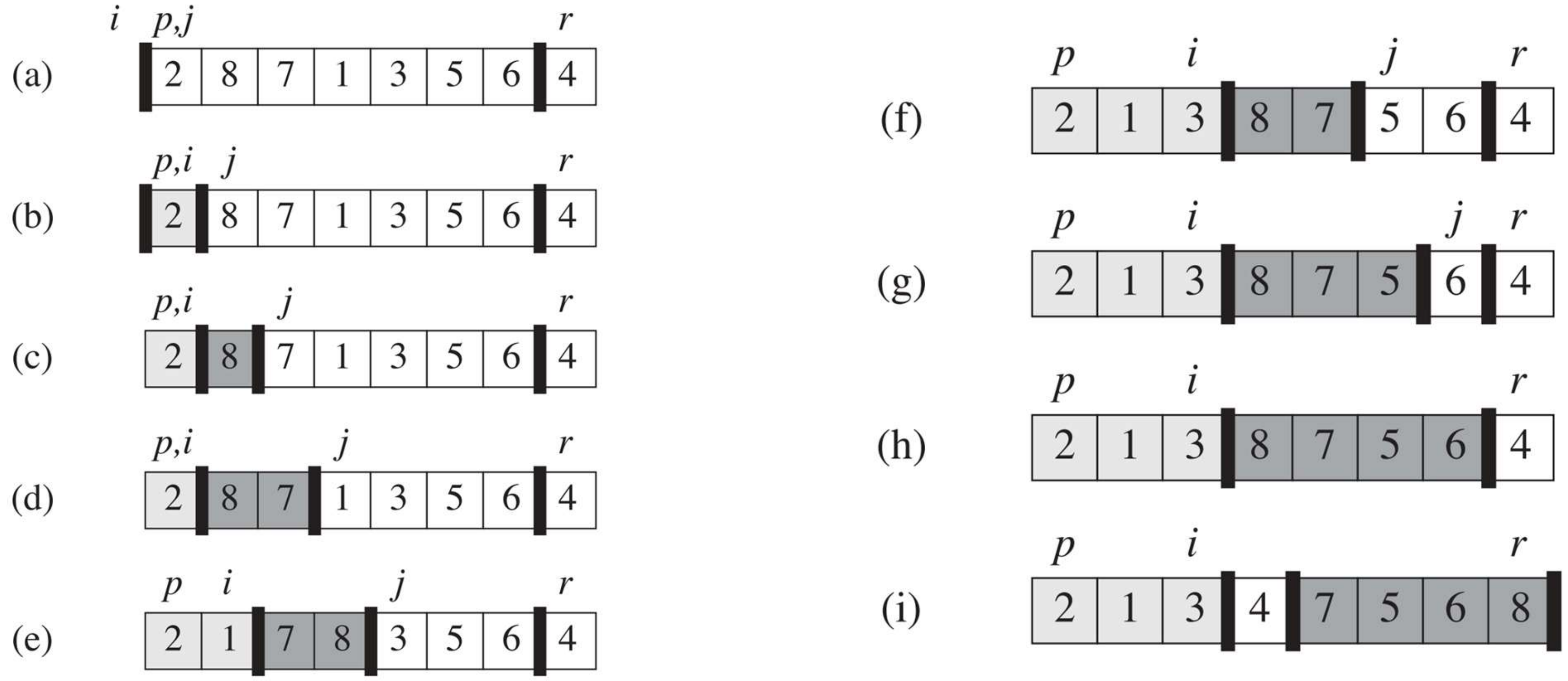
- PARTITION rearranges the subarray in place.
- PARTITION always selects the last element $A[r]$ in the subarray $A[p \dots r]$ as the pivot (the element around which to partition).
- As the procedure executes, the array is partitioned into four regions, some which may be empty:

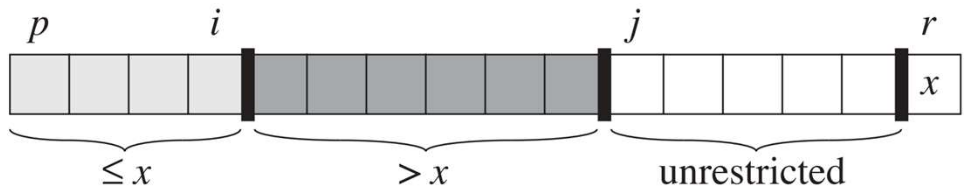
Loop invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot.
2. All entries in $A[i + 1 \dots j - 1]$ are $>$ pivot.
3. $A[r]$ =pivot.

The additional region $A[j \dots r - 1]$ consists of elements that have not yet been processed. We do not yet know how they compare to the pivot element.

Quick Sort


(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8


p
i
j
r
x
≤ x
> x
unrestricted

Quick Sort

Correctness using the loop invariant

- **Initialization:** The subarray $A[p \ldots i]$ and $A[i+1 \ldots j-1]$ are empty before the start of the loop. The loop invariant conditions are therefore satisfied.
- **Maintenance:** Inside of the loop – we swap $A[j]$ and $A[i+1]$ and increase i if $A[j] \leq$ pivot element. The loop counter j is incremented.
- **Termination:** Loop terminates for $j = r$. We have three regions $A[p \ldots i] \leq$ pivot, $A[i+1 \ldots r-1] >$ pivot and $A[r] =$ pivot element.
- At the end of PARTITION, we swap the pivot element in its correct position and then return its position.

Quick Sort

The running time of QUICKSORT depends on the partitioning of the subarrays.

- QUICKSORT is fast as MERGE-SORT if the partitioned subarrays are balanced (even sized).
- QUICKSORT is slow as INSERTION SORT if the partitioned subarrays are unbalanced (uneven sized).

Worst case: Subarrays completely unbalanced.

- Have 0 elements in one subarray and $n-1$ elements in the other subarray.
- The recurrence running time:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= \boldsymbol{O}(n^2) \end{aligned}$$

- The running time is like INSERTION-SORT.
- The worst-case running time occurs when QUICKSORT takes a sorted input array, but INSERTION-SORT runs in $O(n)$ time in this case.

Best case: Subarrays are always completely balanced.

- Each subarray has $\leq n/2$ elements.
- The recurrence running time:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) \\ &= \boldsymbol{\Omega}(n \lg n) \end{aligned}$$

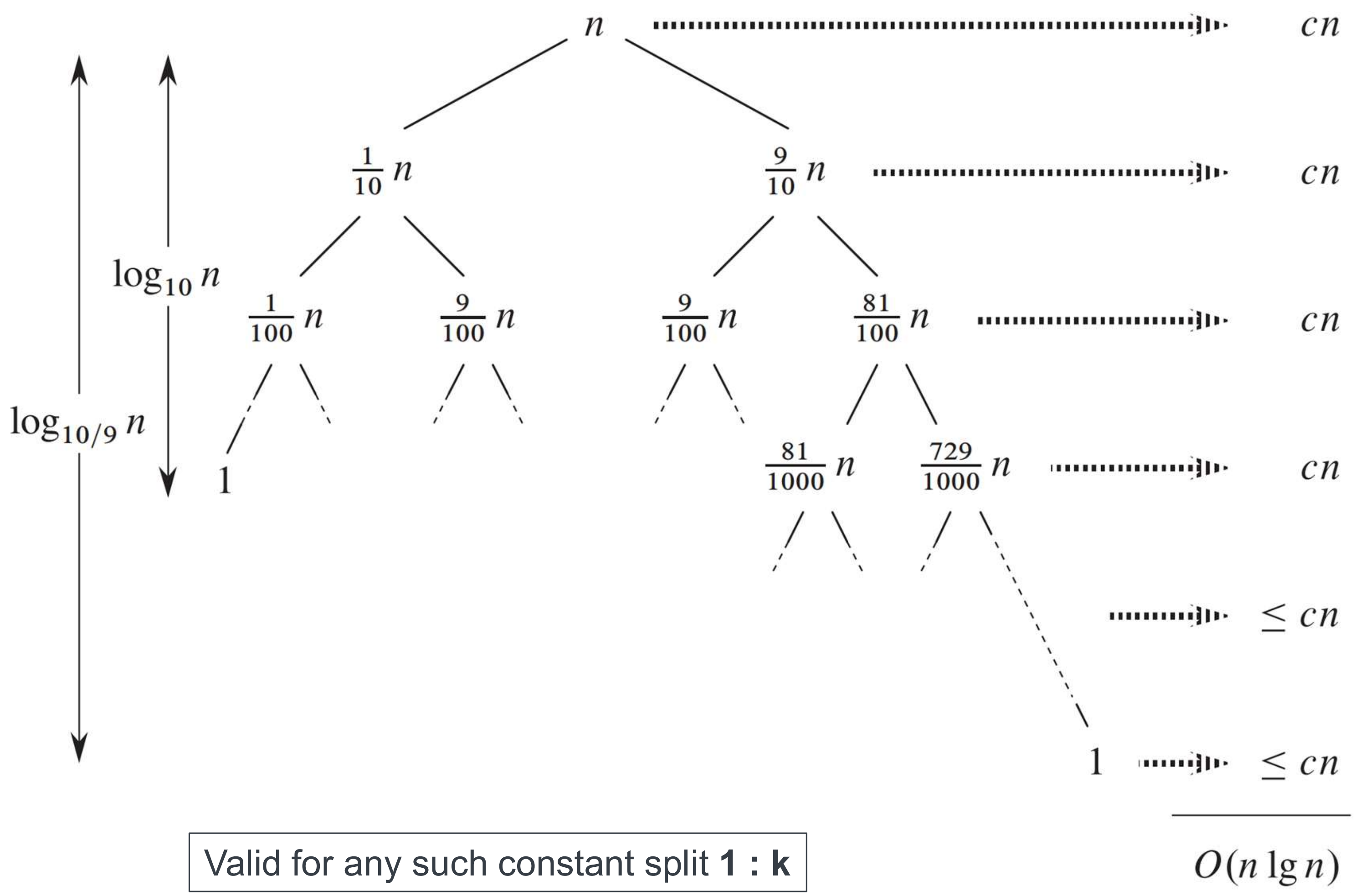
Quick Sort

Balanced partitioning:

- We assume that PATITION always produces a 9 to 1 split (seems unbalanced!! but?).
- Then the recurrence is

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$
$$= O(n \lg n)$$

- In the recursion tree, we get $\log_{10} n$ full levels and $\log_{10/9} n$ levels that are nonempty.
- As long as it's a constant, the base of the log does not matter in asymptotic notation.
- Any split of constant proportionality will yield a recursion tree of depth $\Theta(\lg n)$.



Valid for any such constant split **1 : k**

Randomized Quicksort

- We introduce randomization in order to improve on the quicksort algorithm.
- One option would be to use a random permutation of the input array.
- We use random sampling instead which is picking one element at random.
- Instead of using $A[r]$ as the pivot element we randomly pick an element from the subarray.

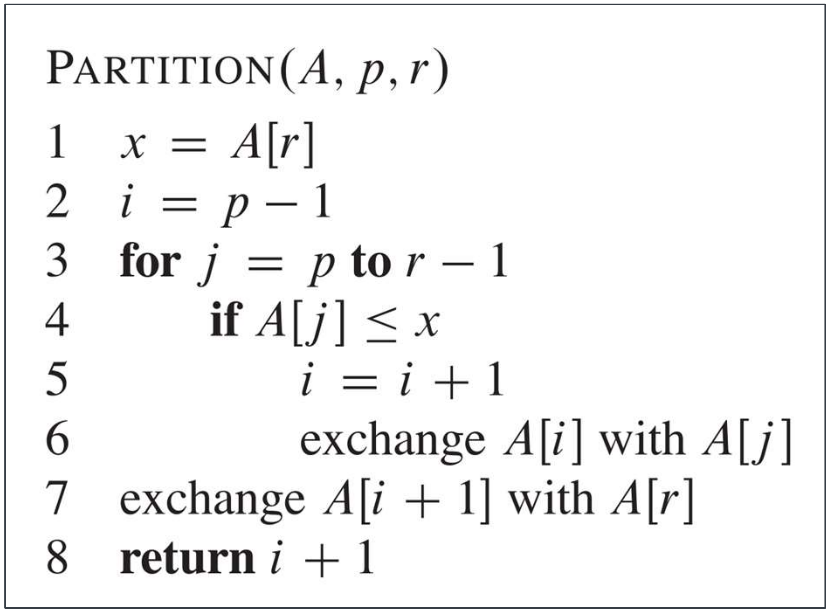
```
RANDOMIZED-PARTITION($A, p, r$)
1    $i = \mathrm{RANDOM}(p, r)$
2    $EXCHANGE\ A[r] \leftrightarrow A[i]$
3    return $\mathrm{PARTITION}(A, p, r)$
```

*Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.

```
RANDOMIZED-QUICKSORT($A, p, r$)
1    if $p < r$
2        $q = \mathrm{RANDOMIZED - PARTITION}(A, p, r)$
3        $\mathrm{RANDOMIZED - QUICKSORT}(A, p, q - 1)$
4        $\mathrm{RANDOMIZED - QUICKSORT}(A, q + 1, r)$
```

*Randomization of quicksort stops any specific type of array from causing worst-case behavior. For example, an already-sorted array causes worst-case behavior in non-randomized quicksort, but not in randomized-quicksort.

Randomized Quicksort



```
PARTITION(A, p, r)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \le x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

- Let, **X** be the number of comparisons performed in line-4 over the entire execution over an n-element array.
- Each time the partition is executed, a pivot element is ordered and never included in future calls. So, partition can't be executed more than n times.
- T(n) = ***O***(n+**X**)

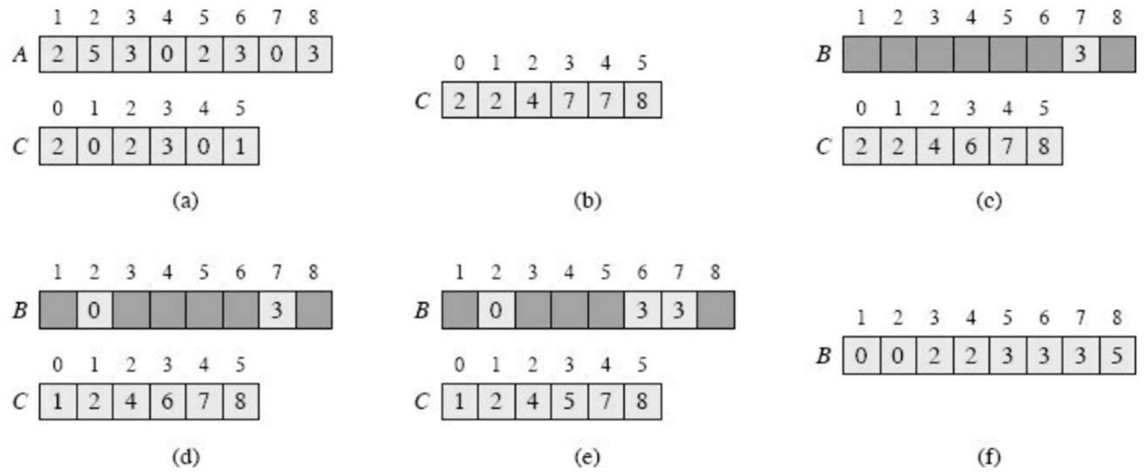
Counting Sort

Counting sort: Non-comparison sorting algorithm.

Our key assumption is that the numbers to be sorted are integers from the set $\{0,1,\dots,k\}$.

Input: $A[1 \dots n]$, where $A[j] \in \{0,\dots,k\}$ for $j = 1,\dots,n$. The array A, values n and k are given as parameters.
Output: Sorted array $B[1,\dots,n]$. B is a parameter and already allocated.
Auxiliary storage: $C[0 \dots k]$.

```
COUNTING-SORT(A,B,n,k)
1   create new array $C[0,\ldots,k]$
2   for $(0 \le i \le k)$ do
3       $C[i] = 0$
4   for $(1 \le j \le n)$ do
5       $C[A[j]] = C[A[j]] + 1$
6   for $(1 \le i \le k)$ do
7       $C[i] = C[i] + C[i-1]$
8   for $(n \ge j \ge i)$ do
9       $B\left[C[A[j]]\right] = A[j]$
10      $C[A[j]] = C[A[j]] - 1$
```



- Counting sort is stable.
- Means that keys with the same value appear in the same order in the output as the appeared in the input.
- The last loop in the algorithm ensures this property.

Counting Sort

Analysis:

- Running time $\Theta(n + k)$ which is $\Theta(n)$ if $k = O(\text{n})$

Practical?

- Not a good idea to use it to sort 32-bit values.
- Might not be a good idea for 16-bit values.
- Probably a good idea for 8-bit or 4-bit value. Strongly depends on the number of values n.

Memory consumption can be a problem

- The auxiliary storage C necessary for goes from $0 \text{ to } k$.
- The 32-bit integers we need 16GB of auxiliary storage. We need a 32-bit counter for each of the $0 \text{ to } k = 2^{32} - 1$.
- ⇒ We will use counting sort within radix sort

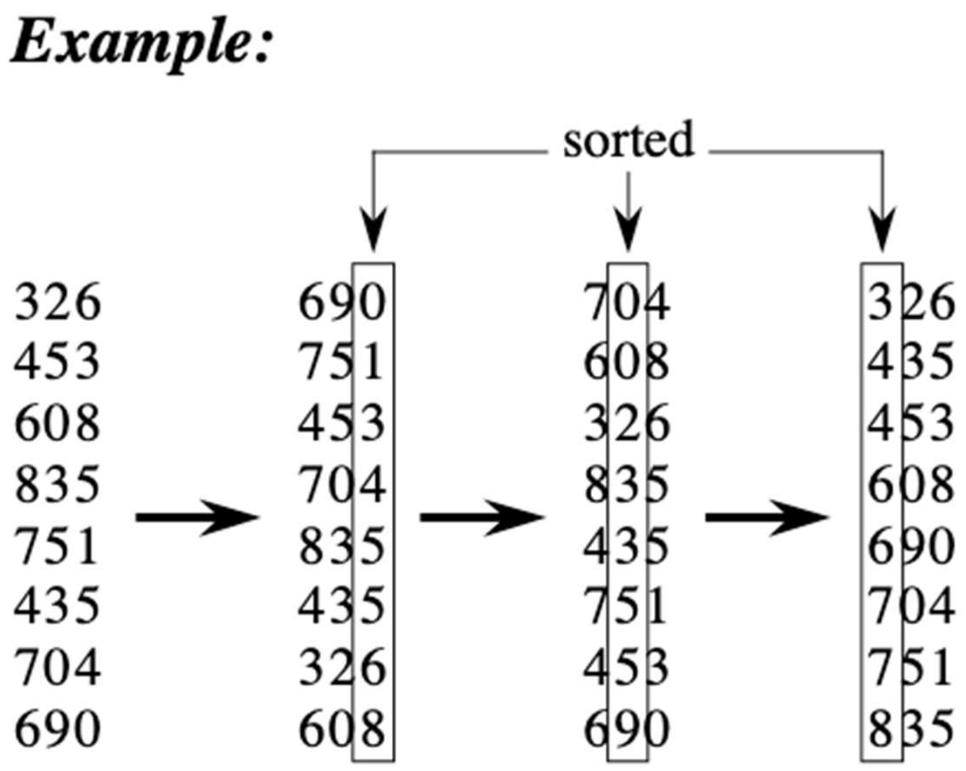
Radix Sort

- Goes back to IBM and census in early 1900.
- Card sorters would work on one column at a time.
- Key idea: Sort least significant digits first.

```
RADIX-SORT(A, d)
1       for $(1 \leq i \leq d)$ do
2           sort Array A on digit $i$ using stable sorting algorithm
```

d is the highest-order digit

Radix Sort


Example:
sorted
326 453 608 835 751 435 704 690
690 751 453 704 835 435 326 608
704 608 326 835 435 751 453 690
326 435 453 608 690 704 751 835

Radix Sort

Correctness:

- We use induction on the number of passes (loop variable i).
- We assume that the digits $1, \dots, i-1$ are sorted.
- We show that a stable sorting algorithm on digit i leaves the digits $1, \dots, i-1$ sorted.
    - If 2 digits in position i are different, ordering by position i is correct, and positions $1, \dots, i-1$ are irrelevant.
    - If 2 digits in position i are equal, then the numbers are already in the right order (by inductive hypothesis). The stable sort on digit i leaves them in the right order.

Analysis:

- We assume that we use counting sort as the intermediate sort.
- Running time $\Theta(n+k)$ per pass (digits in range $0, \dots, k$).
- We have to do d passes (for loop).
- The total running time is $\Theta\big(d(n+k)\big) \Rightarrow \Theta(dn), \text{if } k = O(n)$.

Summary

- We have completed sorting algorithms.
- Next Lecture: We will discuss Trees.
- Assignment-2 is out!

Contents of this presentation are partially adapted from
My CS385 (Fall2022)
and from
Prof. In Suk Jang CS590 (Summer 2021 Lecture-5)
and are also based on
Book Chapter- 7 & 8, Introduction to Algorithms by *Cormen, Leiserson, Rivest, & Stein*

THANK **YOU**

Stevens Institute of Technology