Ontario Tech Science

Kourosh Davoudi
kourosh@ontariotechu.ca

Lecture 4: Sort Algorithms II

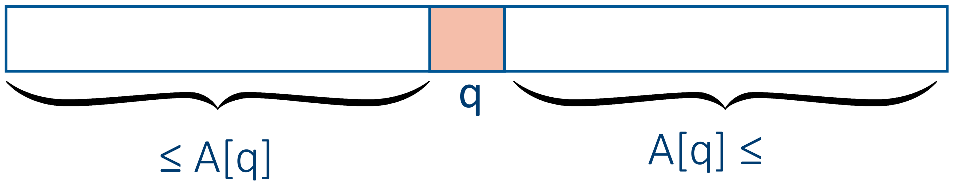
CSCI 3070U: Design and Analysis of Algorithms

Learning Outcomes

- Sorting Algorithms
  - Quick Sort
  - Linear Time Sort Algorithms

Quick Sort Foundations

- Quicksort uses a divide-and-conquer approach.
- To sort a subarray A[p .. r]
  - **Divide**: partition A[p .. r] into two (possibly empty) subarrays A[p .. q-1] and A[q+1 .. r] such that A[p .. q-1] ≤ A[q] and A[q] ≤ A[q+1 .. r]



  - **Conquer**: sort the subarrays using Quicksort recursively.
  - **Combine**: simple concatenation of A[p .. q-1], A[q] and A[q+1 .. r] produces the correct ordering

Quick Sort

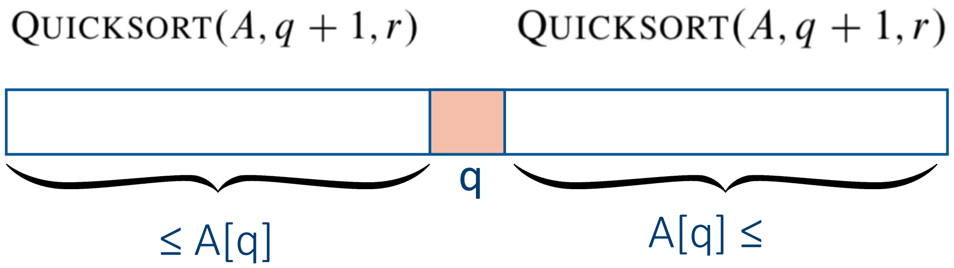
QUICKSORT(A, p, r)

if $p < r$

  $q = $ PARTITION(A, p, r)

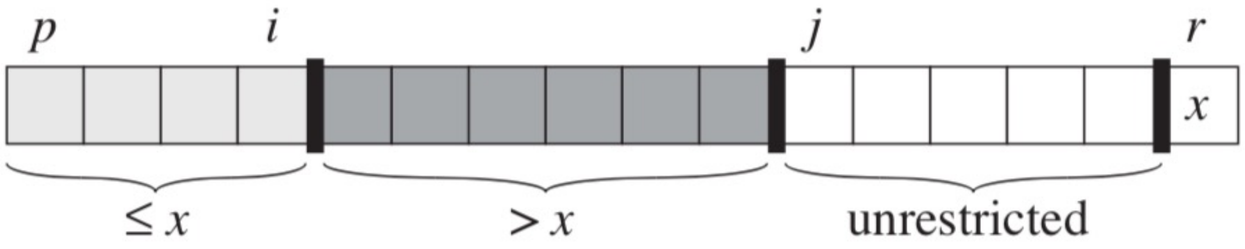
  QUICKSORT$(A, p, q - 1)$

  QUICKSORT$(A, q + 1, r)$



Partitioning in Quicksort

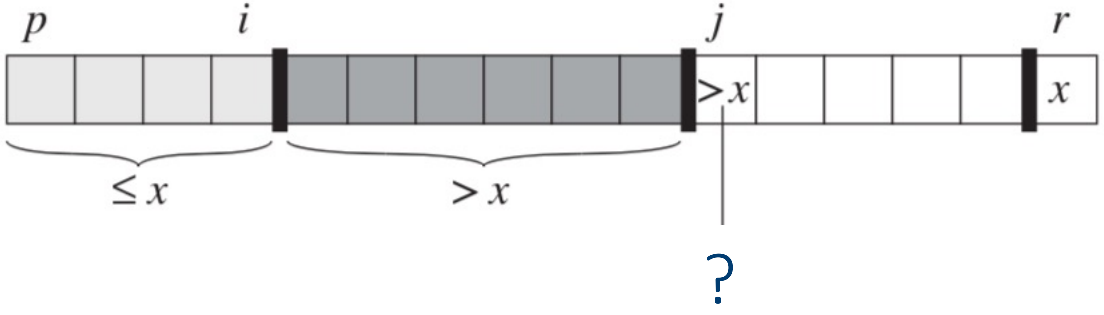
- Partitions subarray A[p .. r] by using the last element A[r] as a pivot element



- The four regions maintained by the procedure PARTITION on a subarray A[p .. r]

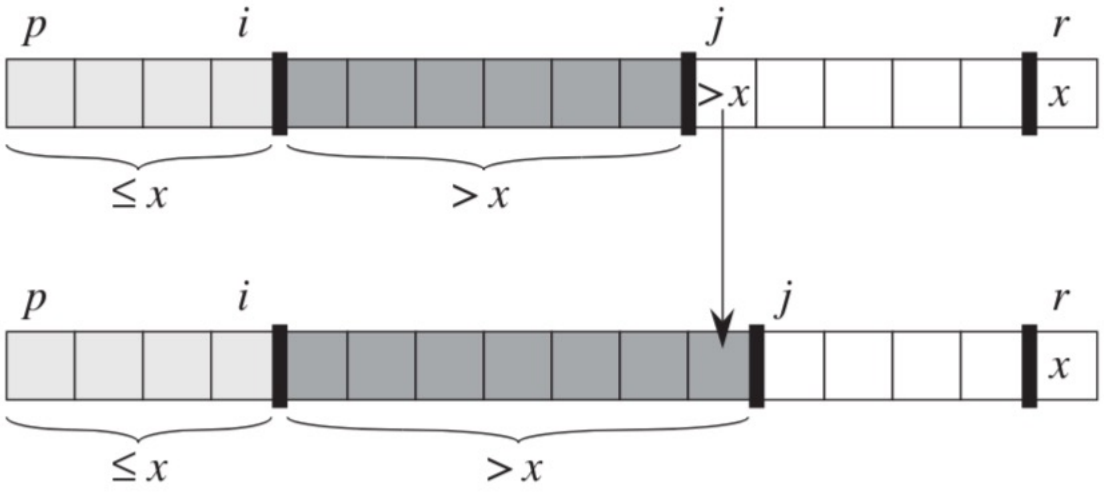
Partitioning in Quicksort

- Case 1: If A[j] > x,



Partitioning in Quicksort

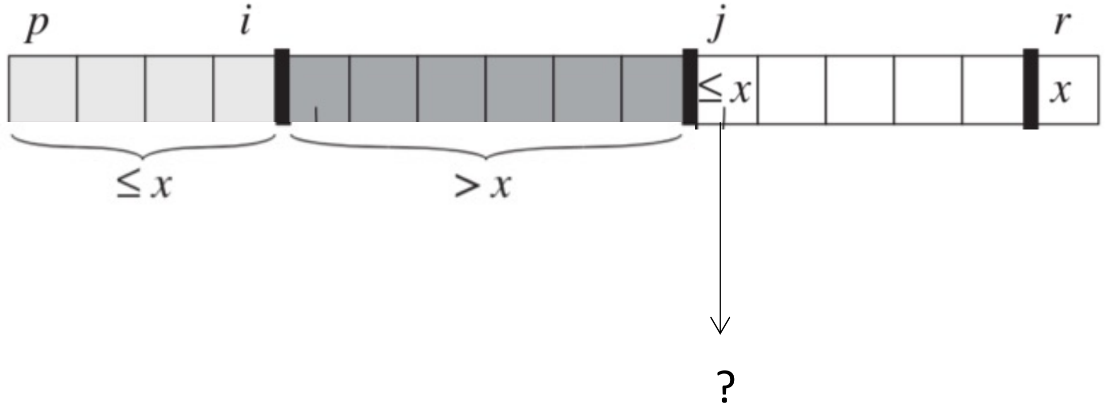
- Case 1: If A[j] > x,



the only action is to increment j

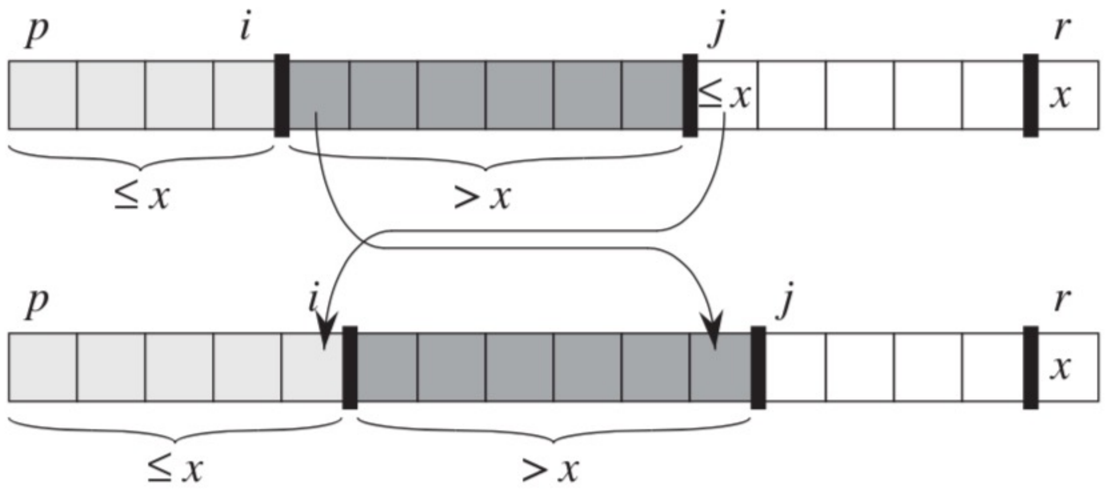
Partitioning in Quicksort

- Case 1: If A[j] ≤ x,



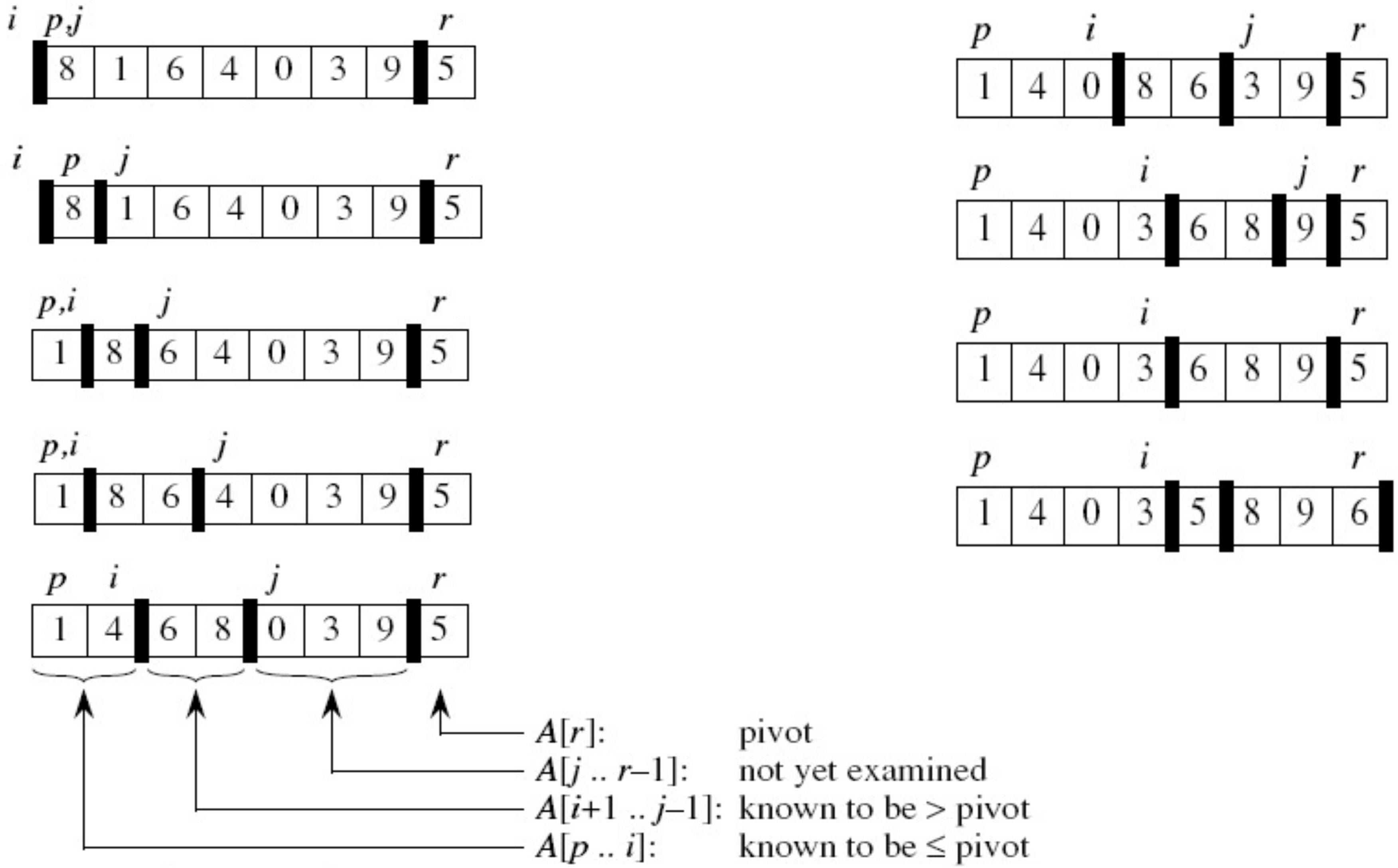
Partitioning in Quicksort

- Case 1: If A[j] ≤ x,



Index i is incremented, A[i] and A[j] are swapped, and then j is incremented.

Partitioning in Quicksort



Partitioning in Quicksort

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \leq x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return $i + 1$

$$\Theta(n)$$

Quick Sort Performance

- Worst Case:

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

$$T(n) = \Theta(n^2)$$

- Best Case:

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

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

Lower Bounds for Sorting

- How fast can we sort?
  - All sorts seen so far are

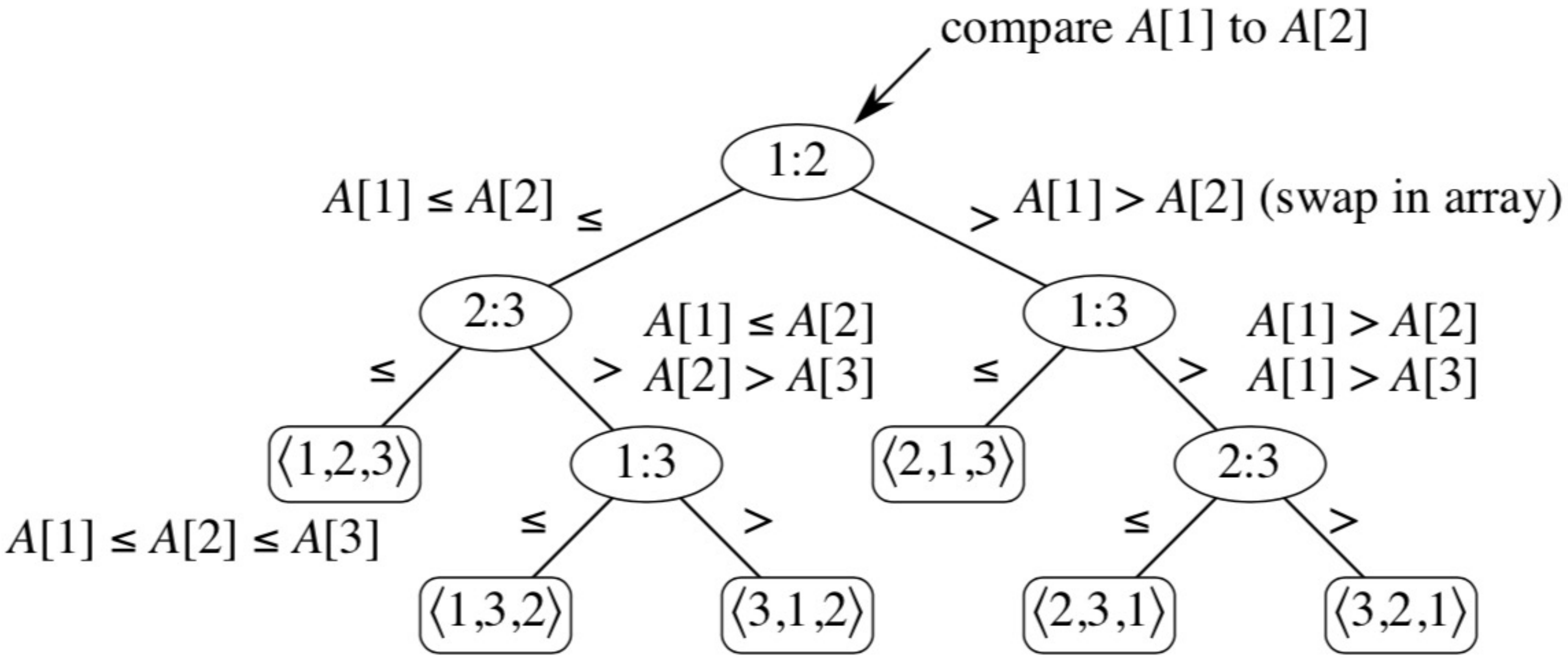
$$\Omega(n \log n)$$

  - We'll show that $\Omega(n \log n)$ is a lower bound for comparison sorts.

  - We use decision trees for this purpose:
    - The important factor is number of comparison and decision tree help us track it !

Lower Bounds for Sorting

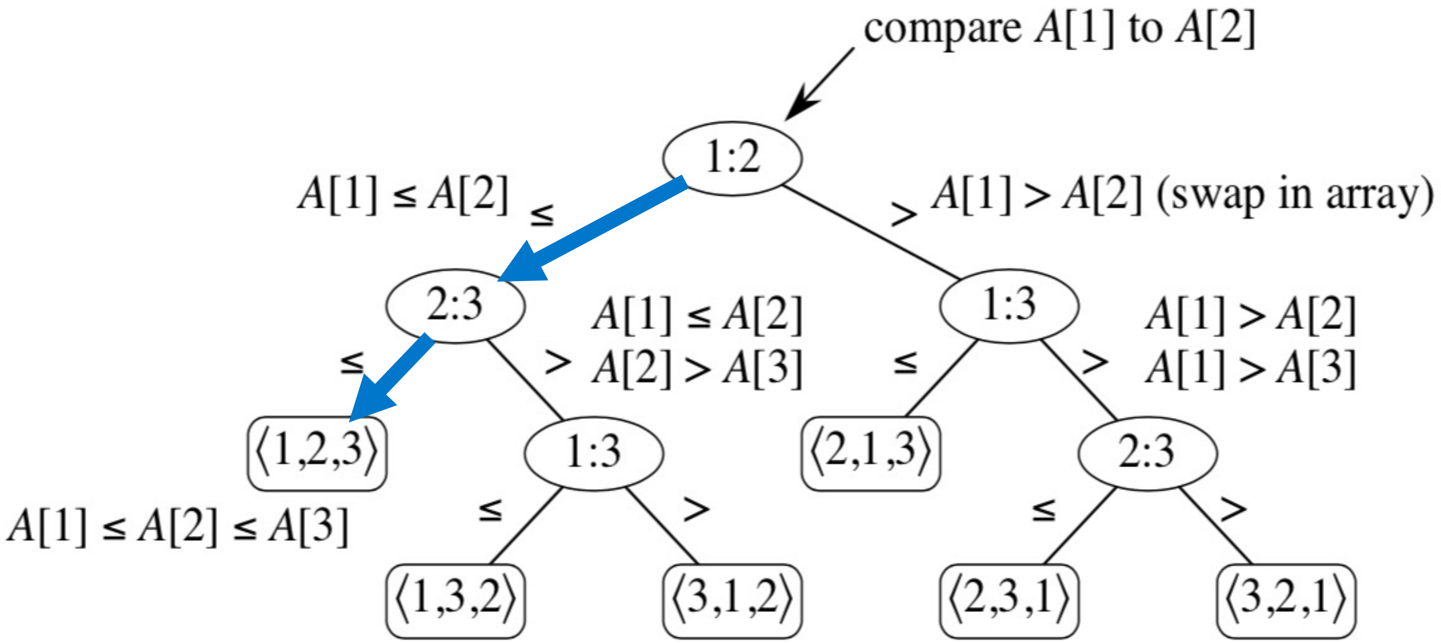
- For insertion sort on 3 elements
  - Lets track number of comparisons !



Each leaf is a permutation of orders

Lower Bounds for Sorting

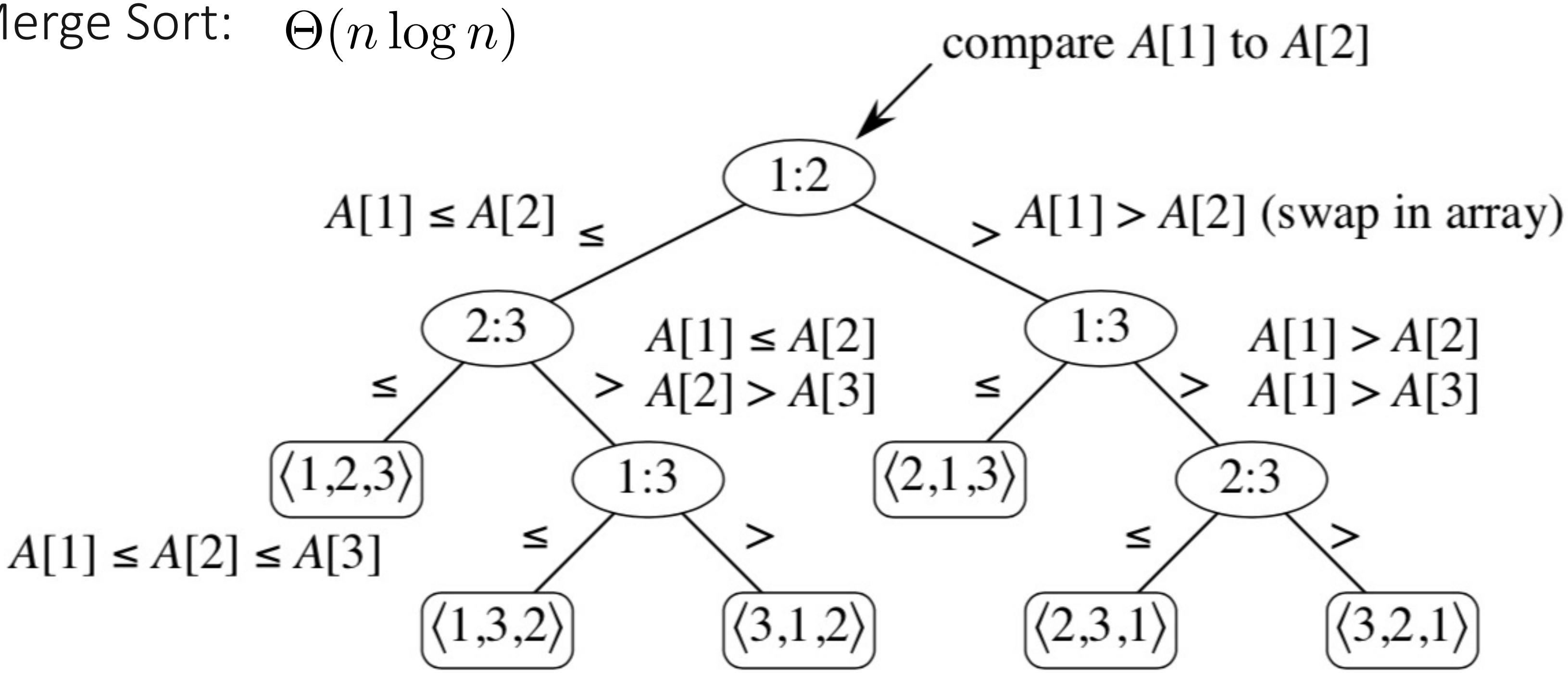
- How many leaves on the decision tree? $\geq n!$
  - because every permutation appears at least once
- A particular trace of the algorithm is a simple path from the root to a leaf node



Height of the tree = the time complexity of the algorithm

Lower Bounds for Sorting

- What is the length of the longest path from root to leaf?
  - Depends on algorithms:
    - Insertion Sort: $\Theta(n^2)$
    - Merge Sort: $\Theta(n \log n)$



Lower Bounds for Sorting

- **Lemma**: Any binary tree of height h and l leaves, we have $l \leq 2^h$
- **Theorem**: Any decision tree that sorts n elements has height $\Omega(n \log n)$

Proof

- $l \geq n!$
- By lemma, $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Take logs: $h \geq \lg(n!)$
- Use Stirling's approximation: $n! > (n/e)^n$

$$\begin{aligned} h &\geq \lg(n/e)^n \\ &= n \lg(n/e) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) . \end{aligned}$$

Stirling's approximation

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Linear Time Sorting

- Comparison sorting lower bound: $n \log n$
- So, how is linear time sorting possible?
  - We don’t use comparison between elements to sort
  - Linear sorting algorithms only work with numeric keys !

Counting Sort
Radix Sort
Bucket Sort

Counting Sort

- *Assumption*: numbers to be sorted are integers in

$$\{0, 1, \ldots, k\}$$

- Input: $A[1 \,..\, n]$, where $A[j] \in \{0, 1, \ldots, k\}$ for $j = 1, 2, \ldots, n$

- Output: $B[1 \,..\, n]$, sorted. B is assumed to be already allocated and is given as a parameter

- Auxiliary storage: $C[0 \,..\, k]$

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
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

...

Counting Sort

- Counting sort is **stable**
  - Keys with same value appear in same order in output as they did in input
- What is it good for?
  - Small k
  - Integers are 16-bit or 32-bit which are too big for count sort because it would require an auxiliary array of

$C[1 .. 2^{32}]$!

COUNTING-SORT(A, B, n, k)

let $C[0..k]$ be a new array
for $i = 0$ **to** k
  $C[i] = 0$
for $j = 1$ **to** n
  $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
  $C[i] = C[i] + C[i-1]$
for $j = n$ **downto** 1
  $B[C[A[j]]] = A[j]$
  $C[A[j]] = C[A[j]] - 1$

$\Theta(n + k)$, which is $\Theta(n)$ if $k = O(n)$.

Radix Sort Idea

- Key Ideas:
  - View each number as a multi-digit word.
  - Each digit can be arbitrary bits long.
  - Sort from the least significant digit to the most significant digit using any **stable** sorting algorithm.

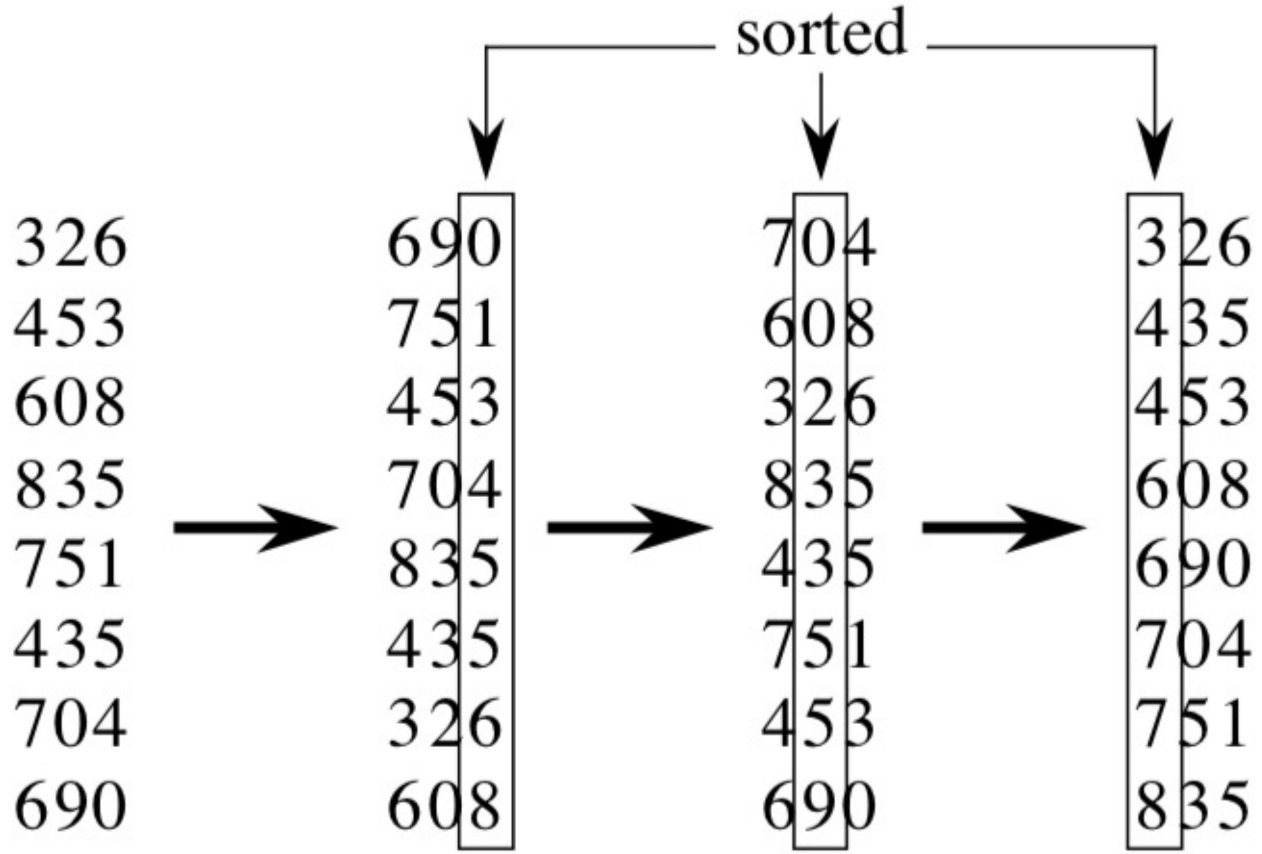
RADIX-SORT(A, d)

for $i = 1$ **to** d

use a stable sort to sort array A on digit i

Radix Sort

- Example:



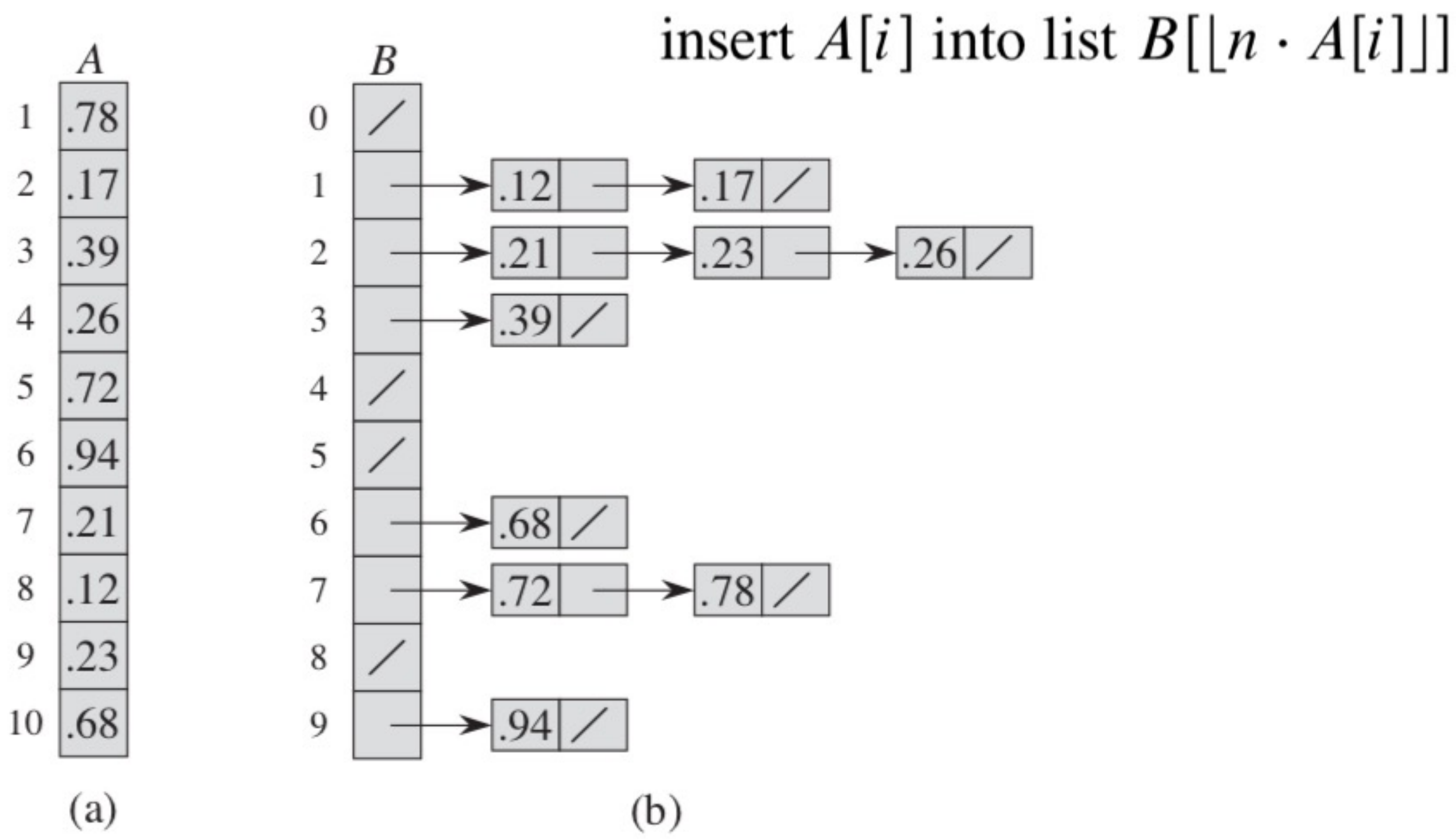
Radix Sort Time Analysis

- Assume that we use counting sort as the intermediate sort
  - $\Theta(n+k)$ per pass (digits in range $0,\ldots,k$)
  - d passes
  - $\Theta(d(n+k))$ total
  - If $k = O(n)$, the complexity is $\Theta(dn)$

Bucket Sort

- *Assumption*: the input is generated by a random process that distributes elements uniformly over [0, 1)
- General Idea
  - Divide [0,1) into n equal-sized *buckets*
  - Distribute the n input values into the buckets
  - Sort each bucket.
  - Then go through buckets in order, listing elements in each one

Bucket Sort



insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
A
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
B
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94
(a)
(b)

Bucket Sort

- **Input:** $A[1..n]$, where $0 \le A[i] \le 1$ for all i
- **Auxiliary array:** $B[0..n\text{-}1]$ of linked lists, each list initially empty

```
BUCKET-SORT(A, n)
  let B[0..n − 1] be a new array
  for i = 0 to n − 1
      make B[i] an empty list
  for i = 1 to n
      insert A[i] into list B[⌊n · A[i]⌋]
  for i = 0 to n − 1
      sort list B[i] with insertion sort
  concatenate lists B[0], B[1], ..., B[n − 1] together in order
  return the concatenated lists
```

Bucket Sort Time Complexity

- Average Case:
  - Assume n_i = the number of elements placed in bucket $B[i]$.

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Bucket Sort Time Complexity

- Average Case:
  - Assume n_i = the number of elements placed in bucket $B[i]$.

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}\left[n_i^2\right]) \quad (\mathrm{E}[aX] = a\mathrm{E}[X])
\end{aligned}$$

Bucket Sort Time Complexity

- Average Case:

$$= \quad \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2])$$

  - Claim

$$\mathrm{E}[n_i^2] = 2 - (1/n) \text{ for } i = 0, \ldots, n-1$$

Define indicator random variables:

- $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$
- $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$
- $n_i = \sum_{j=1}^{n} X_{ij}$

Bucket Sort Time Complexity

- Average Case:

$$
\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \mathrm{E}\left[X_{ij}X_{ik}\right] \quad \text{(linearity of expectation)}
\end{aligned}
$$

Bucket Sort Time Complexity

- Average Case:

$$
\begin{aligned}
\mathrm{E}\left[X_{ij}^2\right] &= 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\} \\
&= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} \\
&= \frac{1}{n}
\end{aligned}
$$

$\mathrm{E}[X_{ij} X_{ik}]$ for $j \neq k$: Since $j \neq k$, X_{ij} and X_{ik} are independent random variables

$$
\begin{aligned}
\Rightarrow \mathrm{E}[X_{ij} X_{ik}] &= \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] \\
&= \frac{1}{n} \cdot \frac{1}{n} \\
&= \frac{1}{n^2}
\end{aligned}
$$

Bucket Sort Time Complexity

- Average Case:

$$
\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \sum_{j=1}^{n} \frac{1}{n} + 2\sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + 2\binom{n}{2}\frac{1}{n^2} \\
&= 1 + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 1 + 1 - \frac{1}{n} \\
&= 2 - \frac{1}{n} \qquad \blacksquare \text{ (claim)}
\end{aligned}
$$

Therefore:

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}
$$

Wrap-Up

- We Learned
  - Quick sort as an important sorting algorithm
  - Lower bound on sorting algorithm
  - Linear time sort algorithms
    - Their assumptions
    - Case studies
      - Counting Sort
      - Radix Sort
      - Bucket Sort
  - Probabilistic Time Complexity Analysis