CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

Chapter 8. Sorting in Linear Time

Sorting Algorithms

- Until now, we have seen sorting algorithms with time complexity $\Theta(n^2)$ and $\Theta(n \log n)$.

Algorithm	Worst-case	Average-case
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$

- These algorithms have one thing is common: sorting is done based only on comparisons between the input elements.
- We call these algorithms comparison sorts (comparison-based sorting).

8.1 Lower Bounds for Sorting

Comparison Sorts

- A comparison sort uses only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \dots, a_n \rangle$.

- Given two elements a_i and a_j, it performs tests such as $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$ in order to determine their relative order.

- It may not inspect the values of the elements or gain order information about them in any other way.

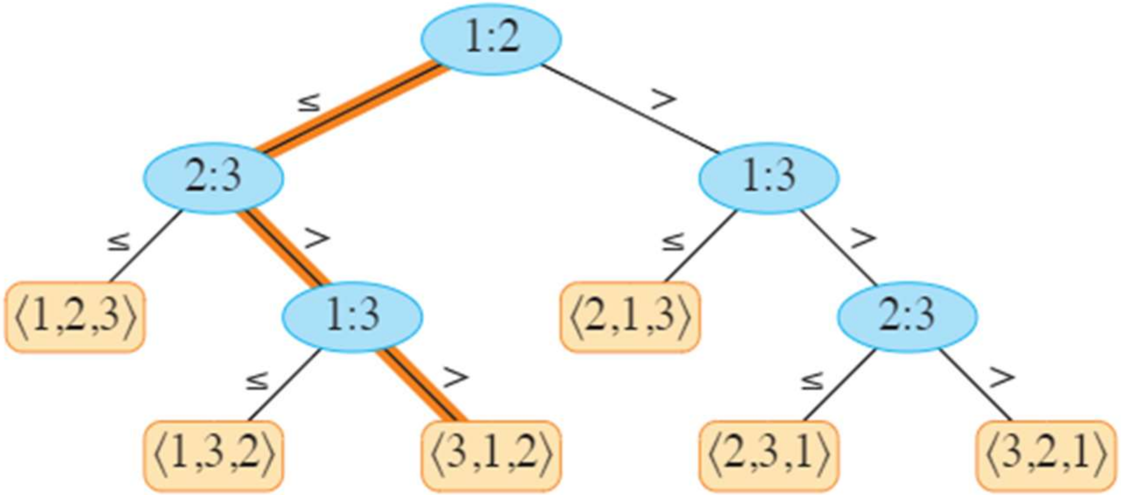
Comparison Sorts: Assumptions

- Without loss of generality, we assume that all input elements are distinct.
  - The same proof will apply for the case when elements are not distinct.

- With the above assumption, we can assume that all comparisons have the form $a_i \leq a_j$.
  - $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, and $a_i < a_j$ all yield identical information about the relative order of a_i and a_j.

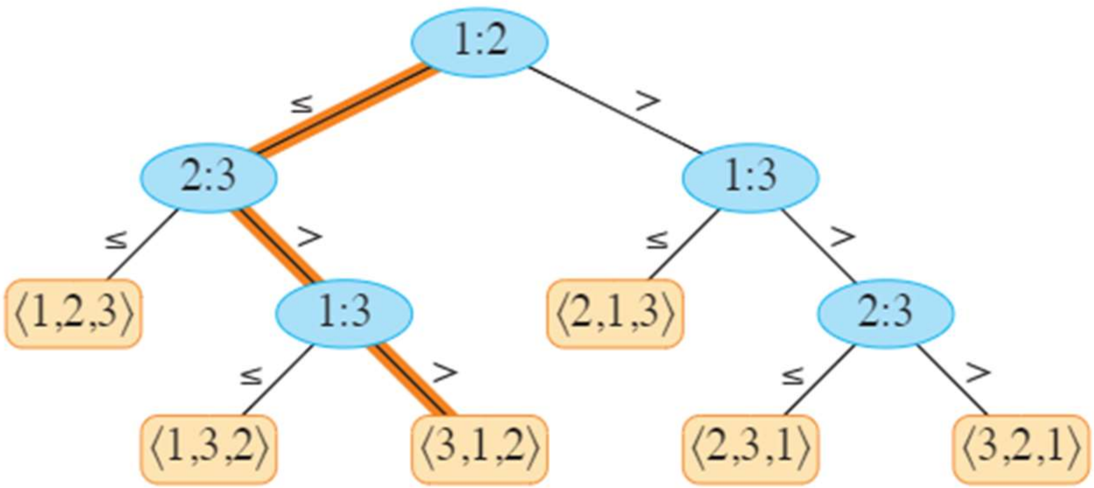
Comparison Sorts: Decision-Tree Model

- Decision Tree
  - A full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm
  - Full binary tree: a tree in which each node is either a leaf or has both children



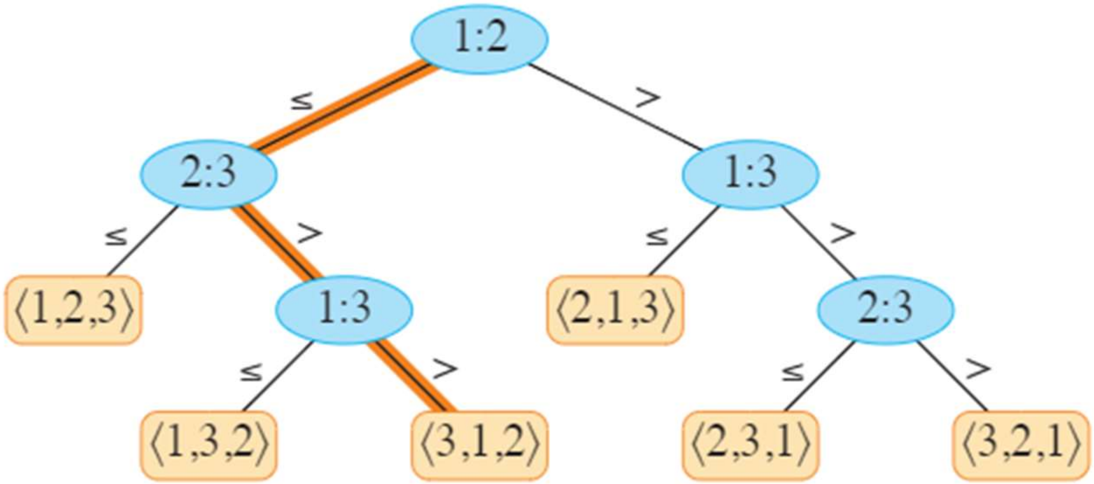
Comparison Sorts: Decision-Tree Model

- Each internal node is annotated by $i{:}\,j$ for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence.
- Each leaf node is annotated by a permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$.
- Indices in the internal nodes and the leaves refer to the original positions of the array elements at the start of the sorting algorithm.
- The execution of the comparison sort corresponds to tracing a simple path from the root of the decision tree down to a leaf.
- Each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree.



Comparison Sorts: Lower Bound

- The worst-case number of comparisons
  - The length of the longest simple path from the root of a decision tree to any of its reachable leaves
  - The height of its decision tree



Comparison Sorts: Lower Bound

- Theorem 8.1
  - Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.
- Proof
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort with n elements. Thus, we have $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h leaves, we have
  - $n! \leq l \leq 2^h$
  - Taking logarithms,
  - $h \geq \log n! = \Omega(n \log n)$

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

Stirling's appoximation

- Heapsort and merge sort are asymptotically best strategies for comparison sort!

8.2 Counting Sort

Counting Sort

- Assumption: Each of the n input elements is an integer in the range 0 to k, for some integer k.

- Under this assumption, the counting sort runs in $\Theta(n + k)$ time.
  - If $k = O(n)$, counting sort runs in $\Theta(n)$ time.

Counting Sort: Strategy

- For each input element, counting sort first determines the number of elements less than or equal to x.

- It then uses this information to place element x directly into its position in the output array.
  - If 17 elements are less than or equal to x, then x belongs in output position 17.
  - We need slight modification to handle elements with the same value, so that they are not placed in the same position.

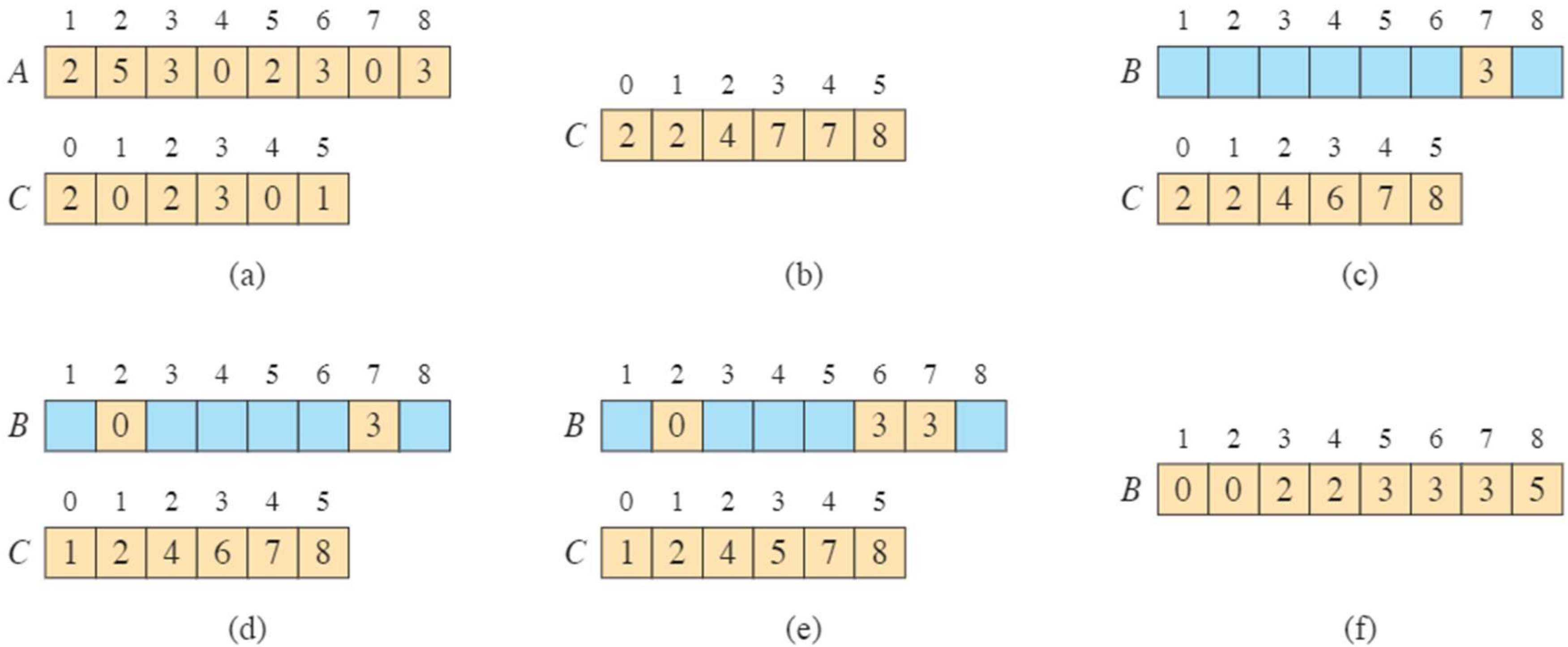
Counting Sort: Algorithm

- Algorithm COUNTING-SORT

```
COUNTING-SORT(A, n, k)
let B[1 : n] and C[0 : k] be new arrays
for i = 0 to k
    C[i] = 0
for j = 1 to n
    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.
// Copy A to B, starting from the end of A.
for j = n downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1    // to handle duplicate values
return B
```

Counting Sort: Operation Example

- Algorithm COUNTING-SORT



Counting Sort: Analysis and Running Time

- Algorithm COUNTING-SORT
  - Lines 2-3: initializes array C to all zeros.
  - Lines 4-5: makes a pass over the array A to inspect each input element. Each time it finds an input element whose value is i, it increments $C[i]$.
  - Lines 7-8: determines for each $i = 0,1, \ldots, k$ how many input elements are less than or equal to i by keeping a running sum of the array C.
  - Lines 11-13: makes another pass over A, but in reverse, to place each element $A[j]$ into its correct sorted position in the output array B.
    - Line 13 is needed to handle duplicate values

- Running time of COUNTING-SORT
  - Lines 2-3: $\Theta(k)$
  - Lines 4-5: $\Theta(n)$
  - Lines 7-8: $\Theta(k)$
  - Lines 11-13: $\Theta(n)$
  - Overall: $\Theta(k + n)$

Comparison Sort: Remarks

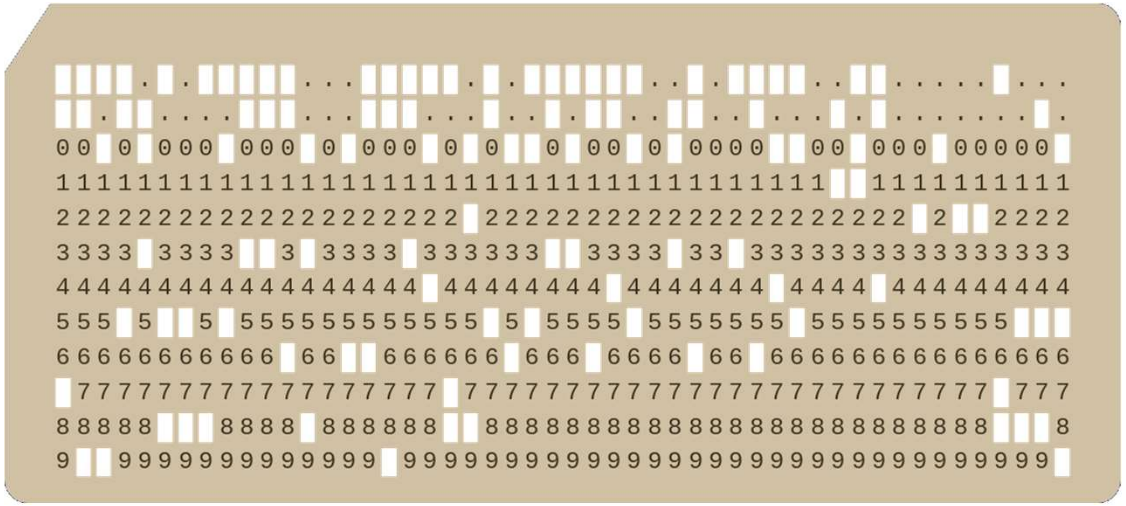
- In practice, we can use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.
- Counting sort can be asymptotically faster than $\Theta(n \log n)$ because it is not a comparison sort.
  - Counting sort uses actual values of the elements instead of comparisons.

- Counting sort is stable
  - Elements with the same value appear in the output array in the same order as they do in the input array.
  - It breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array.
  - Property of stability is important only when satellite data are carried around with the element being sorted.
  - Counting sort's stability is important for another reason: it is often used as a subroutine in radix sort. For radix sort to work correctly, counting sort must be stable.

8.3 Radix Sort

Radix Sort

- Used by the card-sorting machines for punch cards.
  - The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places.
  - The sorter can be mechanically programmed to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched.
  - An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.



Radix Sort: Strategy

- The radix sort solves the problem of card sorting by sorting on the least significant digit first.
- The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on.
- Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner.
- This process continues until the cards have been sorted on all d digits.
- Remarkably, at that point the cards are fully sorted on the d-digit number.

- Thus, only d passes through the deck are required to sort.

Radix Sort: Example and Requirements

- Radix sort

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

- In order for radix sort to work correctly, the digit sorts must be stable.
  - Suppose 457 precedes 458 after sorting the least significant digit.
  - When sorting the second least significant digit, their order must stay the same.

Radix Sort: Algorithm

- Algorithm RADIX-SORT

```
RADIX-SORT(A, n, d)
for i = 1 to d
    use a stable sort to sort array A[1 : n] on digit i
```

- For stable sort, COUNTING-SORT is commonly used.

Radix Sort: Running Time

- Lemma 8.3
  - Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time.

- Proof
  - Each pass over n d-digit numbers takes $\Theta(n+k)$ time.
  - There are d passes.

- When d is constant and $k = O(n)$, we can make radix sort run in linear time.

Radix Sort: Breaking a Number into Digits

- Lemma 8.4
  - Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)$ time if the stable sort it uses takes $\Theta(n + k)$ time for inputs in the range 0 to k.

- Proof
  - For a value $r \leq b$, view each key as having $d = \lceil b/r \rceil$ digits of r bits each.
  - Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$.
  - (For example, a 32-bit word can be viewed as having four 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = \frac{b}{r} = 4$.)
  - Each pass of counting sort takes $\Theta(n + k) = \Theta(n + 2^r)$ time and there are d passes, for a total running time of $\Theta\big(d(n + 2^r)\big) = \Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)$.

Radix Sort: Breaking a Number into Digits

- Given n and b, what value of $r \leq b$ minimizes the expression $(\frac{b}{r})(n + 2^r)$?
  - As r decreases, the factor b/r increases, but as r increases so does 2^r.
  - The answer depends on whether $b < \lfloor \log n \rfloor$.
  - If $b < \lfloor \log n \rfloor$, then $r \leq b$ implies $(n + 2^r) = \Theta(n)$.
  - Thus, choosing $r = b$ yields a running time of $\left(\frac{b}{b}\right)\left(n + 2^b\right) = \Theta(n)$, which is optimal.

  - If $b \geq \lfloor \log n \rfloor$, then choosing $r = \lfloor \log n \rfloor$ gives the best running time.
  - Choosing $r = \lfloor \log n \rfloor$ yields a running time of $\Theta(\frac{bn}{\log n})$.
  - As r increases above $\lfloor \log n \rfloor$, the 2^r term in the numerator increases faster than the r term in the denominator, and so increasing r above $\lfloor \log n \rfloor$ yields a running time of $\Omega(\frac{bn}{\log n})$.
  - If instead r were to decrease below $\lfloor \log n \rfloor$, then the b/r term increases and the $n + 2^r$ term remains at $\Theta(n)$.

Radix Sort: A Good Choice?

- Is radix sort preferable to a comparison-based sorting algorithm, such as quicksort?
- If $b = O(\log n)$, and $r \approx \log n$, then radix sort's running time is $\Theta(n)$, which appears to be better than quicksort's expected running time of $\theta(n \log n)$.
- However, the constant factors hidden in the Θ-notation differ.
- Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take significantly longer.

- Thus, which sorting algorithm to prefer depends on the input data and characteristics of the implementations and underlying machine
  - quicksort often uses hardware caches more effectively than radix sort
  - radix sort that uses counting sort does not sort in place, which many of the $\Theta(n \log n)$-time comparison sorts do. Thus, when primary memory storage is at a premium, an in-place algorithm such as quicksort could be the better choice.

8.4 Bucket Sort

Bucket Sort: Introduction

- Bucket sort is a sorting algorithm that has an average-case running time of $O(n)$.
- Bucket sort assumes that the input is drawn from a uniform distribution.
  - Bucket sort is fast because of the assumption.

- Assumptions: Counting sort vs. Bucket sort
  - Counting sort: input consists of integers in a small range
  - Bucket sort: input is generated by a random process that distributes elements uniformly and independently over the interval [0, 1).

Bucket Sort: Strategy

- Bucket sort divides the interval [0, 1) into n equal-sized subintervals, or buckets.
- Then, it distributes the n input numbers into the buckets.
- Since the inputs are uniformly and independently distributed over [0, 1), we do not expect many numbers to fall into each bucket.
- To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

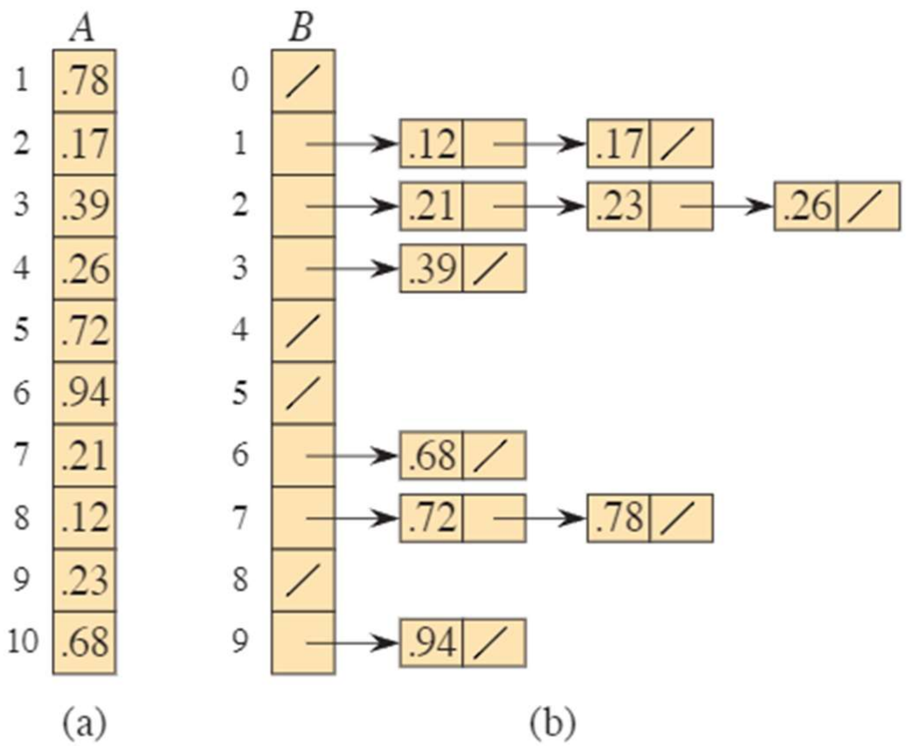
Bucket Sort: Algorithm

- The BUCKET-SORT procedure assumes tha the input is an array $A[1{:}n]$ and that each element $A[i]$ in the array satisfies $0 \leq A[i] < 1$.
- The code requires an auxiliary array $B[0{:}n-1]$ of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

```
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 the lists B[0], B[1], ..., B[n − 1] together in order
return the concatenated lists
```

Bucket Sort: Example

- The operation of bucket sort on an input array of 10 numbers.



Bucket Sort: Running Time

- In the BUCKET-SORT procedure, all lines except line 7 take $O(n)$ time in the worst case. We need to analyze the total time taken by the n calls to insertion sort in line 7.

- Let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time, the running time of bucket sort is:

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

- We now analyze the average-case running time of bucket sort by computing the expected value of the running time.

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

Bucket Sort: Running Time

- What is $E[n_i^2]$?

- First of all, we can see that each bucket i has the same value of $E[n_i^2]$, since each value in the input array A is equally likely to fall in any bucket.

- Let's view each random variable n_i as the number of successes in n Bernoulli trials. Success in a trial occurs when an element goes into bucket $B[i]$.
  – Success probability $p = 1/n$.
  – Failure probability $q = 1 - 1/n$.

- A binomial distribution counts n_i, the number of successes, in the n trials.
- $\mathrm{E}[n_i] = np$, $\mathrm{Var}[n_i] = npq$.
- $\mathrm{E}[n_i] = np = n\left(\frac{1}{n}\right) = 1$
- $\mathrm{Var}[n_i] = npq = 1 - 1/n$

Bucket Sort: Running Time

- $\mathrm{E}\left[n_i^2\right] = \mathrm{Var}[n_i] + \mathrm{E}^2[n_i]$
$$= \left(1 - \frac{1}{n}\right) + 1^2$$
$$= 2 - \frac{1}{n}$$

- Thus, $E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right)$
$$= \Theta(n) + n \cdot O\left(2 - \frac{1}{n}\right)$$
$$= \Theta(n).$$

Bucket Sort: Summary

- Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time.
  - As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr