UMC 201: Data Structures & Algorithms

Lecture 5: Lower bound for sorting; Sorting in linear time

Indian Institute of Science

Recap

- In the last lecture, we introduced the quicksort algorithm that has worst-case running time $O(n^2)$.
- Quicksort sorts the input array *in place*.

Recap

- In the last lecture, we introduced the quicksort algorithm that has worst-case running time $O(n^2)$.
- Quicksort sorts the input array *in place*.
- We then discussed the randomized quicksort algorithm, which has *expected running time* $O(n \log n)$.
- Randomized versions of quicksort are used in *software libraries* as they are efficient in practice.

Recap

- In the last lecture, we introduced the quicksort algorithm that has worst-case running time $O(n^2)$.
- Quicksort sorts the input array *in place*.
- We then discussed the randomized quicksort algorithm, which has *expected running time* $O(n \log n)$.
- Randomized versions of quicksort are used in *software libraries* as they are efficient in practice.

- Question. Is there a sorting algorithm whose run-time is asymptotically better than $\Theta(n \log n)$?

A lower bound for sorting

Comparison sort

- Observe that **merge sort** and **heap sort** (and randomized **quicksort**) have worst-case running time (expected running time) $O(n \log n)$.
- These algorithms share an interesting property: the sorted order they determine is *based only on comparisons between input elements*.

Comparison sort

- Observe that **merge sort** and **heap sort** (and randomized **quicksort**) have worst-case running time (expected running time) $O(n \log n)$.
- These algorithms share an interesting property: the sorted order they determine is *based only on comparisons between input elements*.
- We call such sorting algorithms comparison sorts.
- Today, we'll prove that any comparison sort must take $\Omega(n \log n)$ comparisons in the worst-case to sort n elements.

Comparison sort

- A comparison sort uses only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$.
- Given two elements a_i and a_j, it performs one of the tests $a_i < a_j, a_i \leq a_j, a_i = a_j, a_i \geq a_j$ or $a_i > a_j$ to determine their relative order.

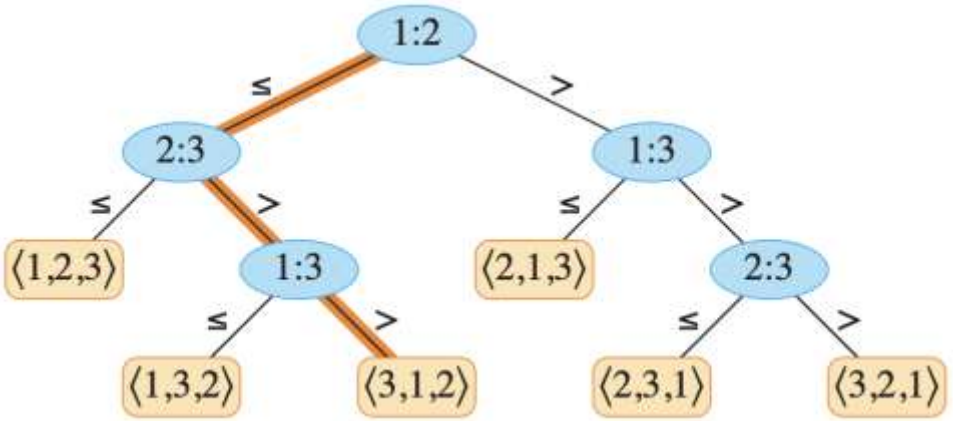
Comparison sort

- A comparison sort uses only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$.
- Given two elements a_i and a_j, it performs one of the tests $a_i < a_j, a_i \leq a_j, a_i = a_j, a_i \geq a_j$ or $a_i > a_j$ to determine their relative order.
- W.l.o.g all the input elements are *distinct*. Therefore, assume that all comparisons are of the form $a_i \leq a_j$.

Comparison sort

- A comparison sort uses only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$.
- Given two elements a_i and a_j, it performs one of the tests $a_i < a_j, a_i \leq a_j, a_i = a_j, a_i \geq a_j$ or $a_i > a_j$ to determine their relative order.
- W.l.o.g all the input elements are *distinct*. Therefore, assume that all comparisons are of the form $a_i \leq a_j$.
- We can view comparison sorts abstractly in terms of decision trees.

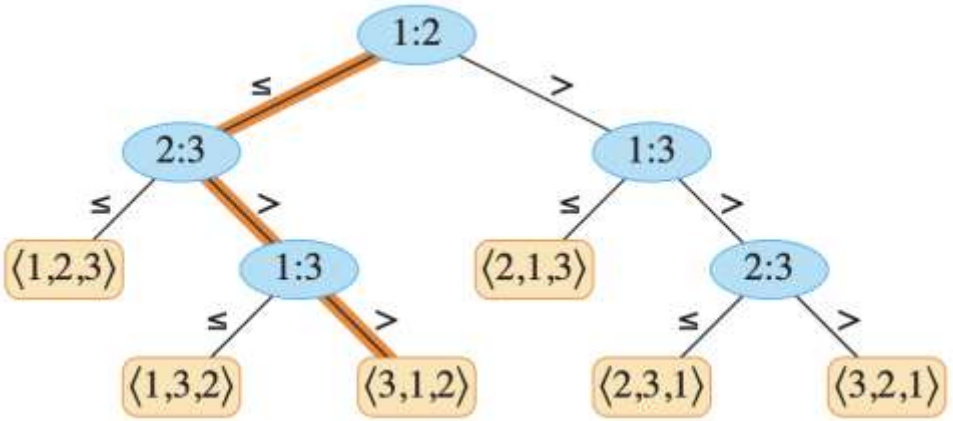
Decision trees



The decision tree for insertion sort on 3 elements

- A decision tree is a **full binary tree** (i.e., each node is either a leaf or has both children) that *represents the comparisons between elements* that are performed by a *particular sorting algorithm* operating on an *input of a given size*.

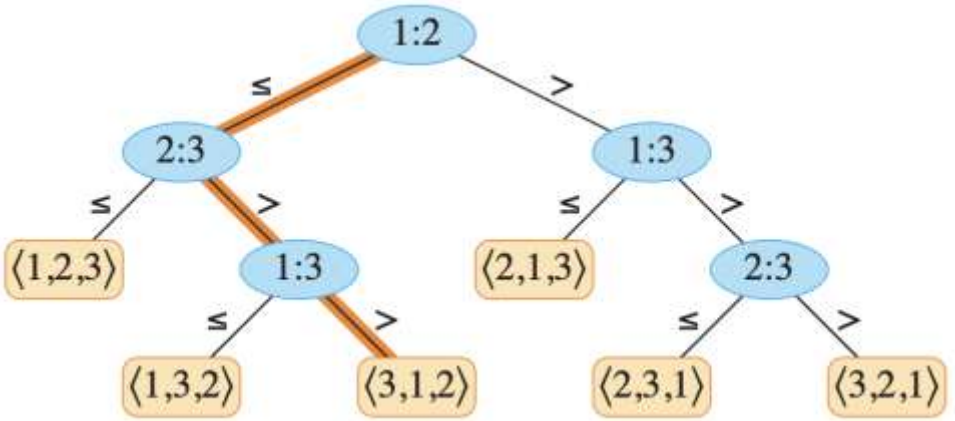
Decision trees



The decision tree for insertion sort on 3 elements

- A decision tree is a **full binary tree** (i.e., each node is either a leaf or has both children) that *represents the comparisons between elements* that are performed by a *particular sorting algorithm* operating on an *input of a given size*. Control, data movement, and other aspects of the algorithm are ignored.

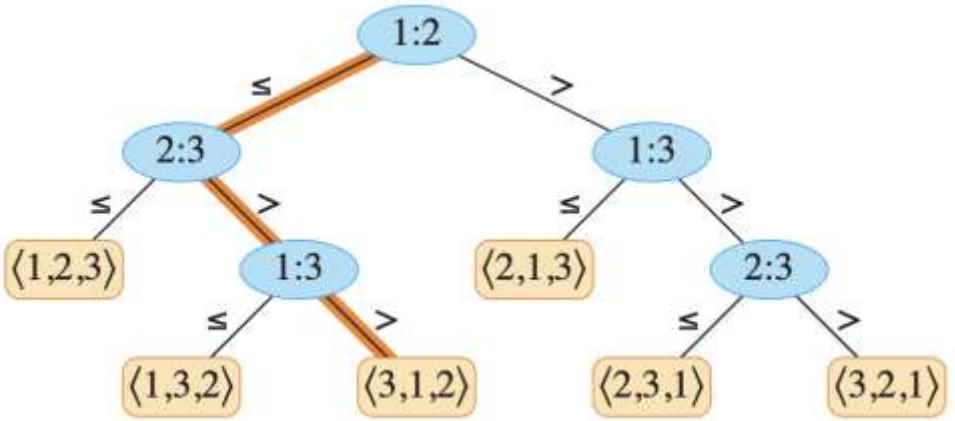
Decision trees



The decision tree for insertion sort on 3 elements

- A decision tree has each internal node annotated by $i:j$ for some $i,j \in [1,n]$, where n is the number of elements.

Decision trees

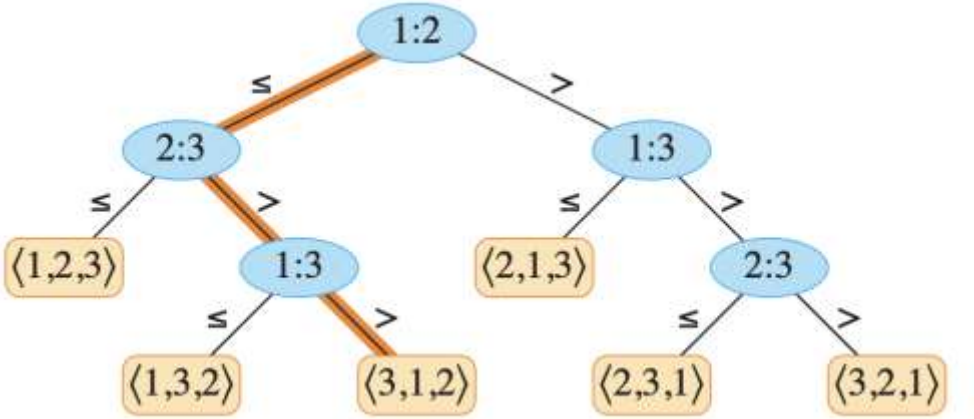


The decision tree for insertion sort on 3 elements

- A decision tree has each internal node annotated by $i:j$ for some $i,j \in [1,n]$, where n is the number of elements.

- We annotate each leaf by a permutation $(\pi(1), \pi(2), \ldots, \pi(n))$.

Decision trees

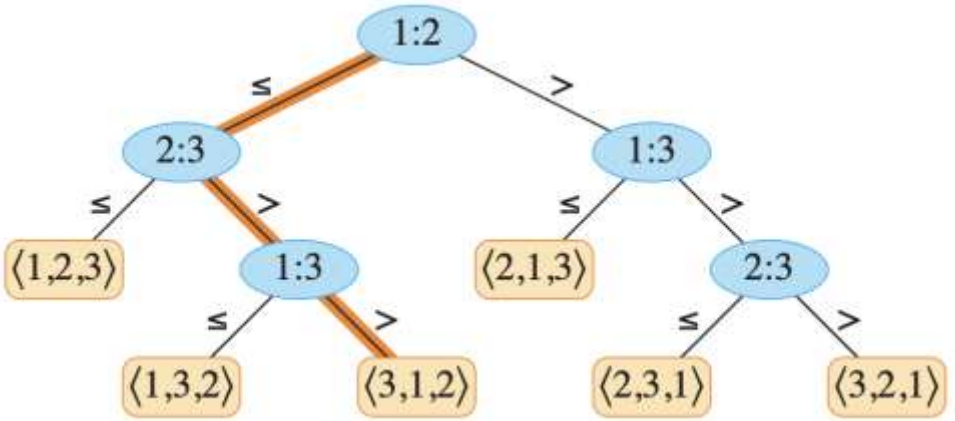


The decision tree for insertion sort on 3 elements

- A decision tree has each internal node annotated by $i:j$ for some $i, j \in [1, n]$, where n is the number of elements.

- We annotate each leaf by a permutation $(\pi(1), \pi(2), \ldots, \pi(n))$.
- Indices in the internal nodes and the leaves refer to the *original positions of the array elements* at the start of the sorting algorithm.

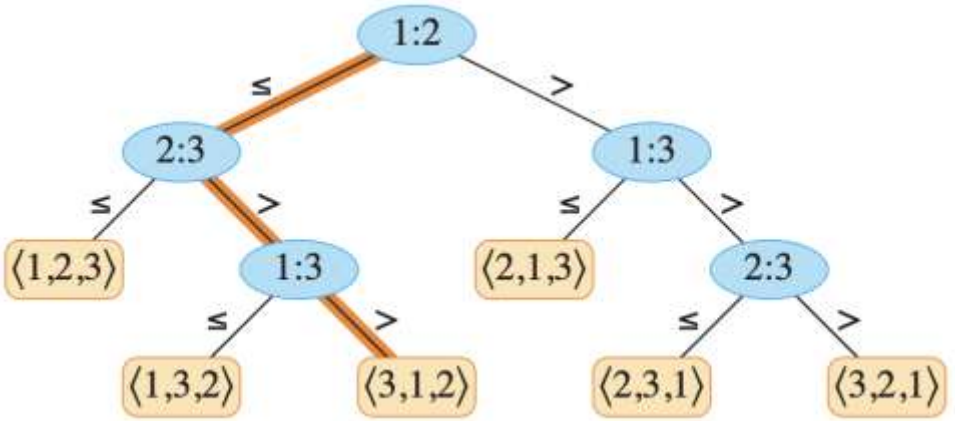
Decision trees



The decision tree for insertion sort on 3 elements

- An execution of the algorithm corresponds to *tracing a simple path* from the root to a leaf of the decision tree.

Decision trees

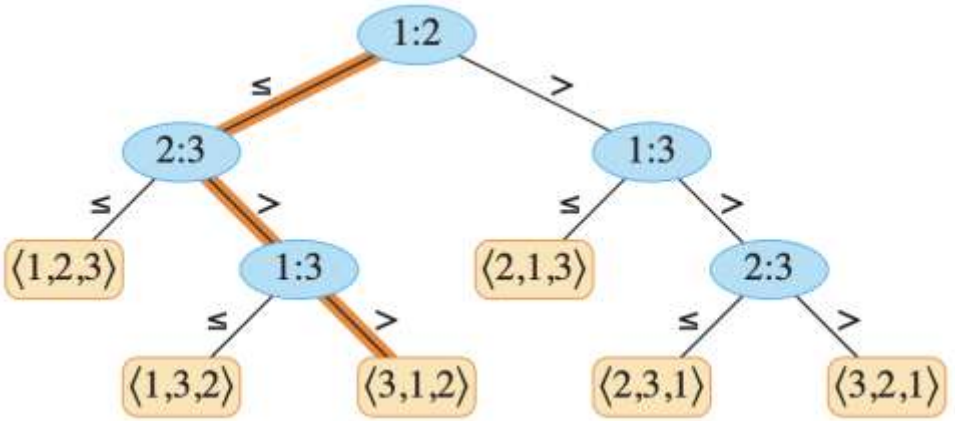


The decision tree for insertion sort on 3 elements

- An execution of the algorithm corresponds to *tracing a simple path* from the root to a leaf of the decision tree.

- Each internal node indicates a comparison $a_i \leq a_j$.

Decision trees

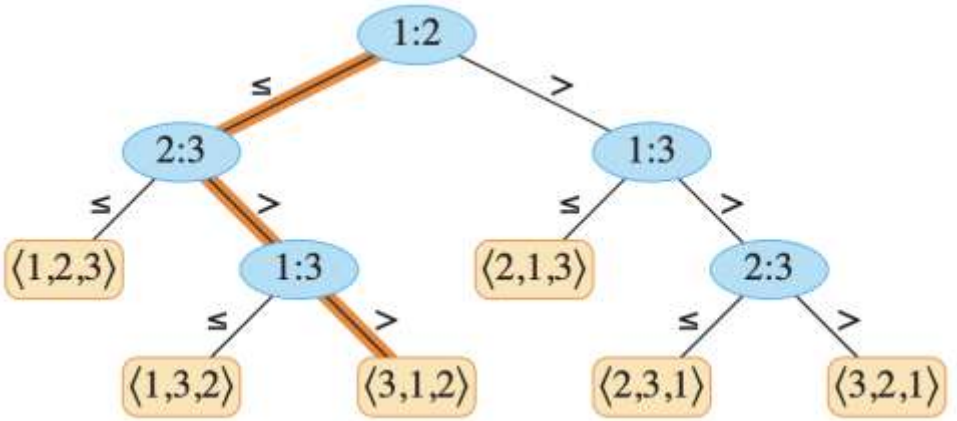


The decision tree for insertion sort on 3 elements

- An execution of the algorithm corresponds to *tracing a simple path* from the root to a leaf of the decision tree.

- Each internal node indicates a comparison $a_i \leq a_j$.
- The left subtree dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons when $a_i > a_j$.

Decision trees

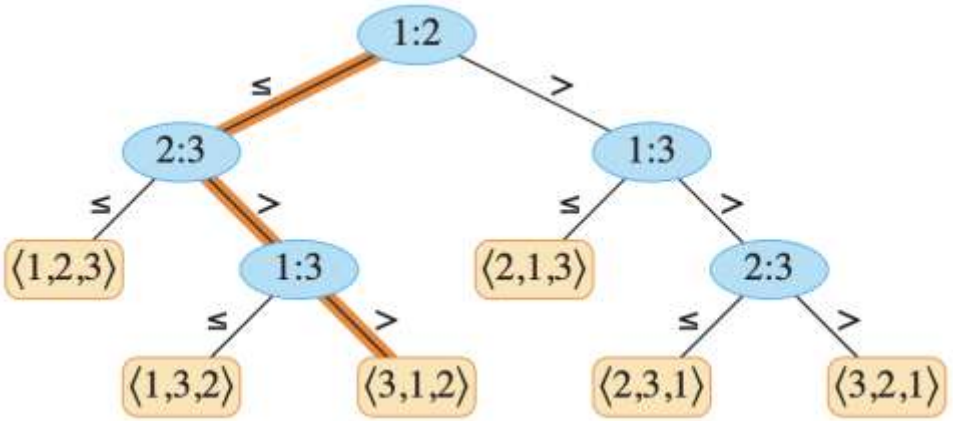


The decision tree for insertion sort on 3 elements

- An execution of the algorithm corresponds to *tracing a simple path* from the root to a leaf of the decision tree.

- Each internal node indicates a comparison $a_i \leq a_j$.
- The left subtree dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons when $a_i > a_j$.
- Arriving at a leaf, the sorting algorithm has established the order $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

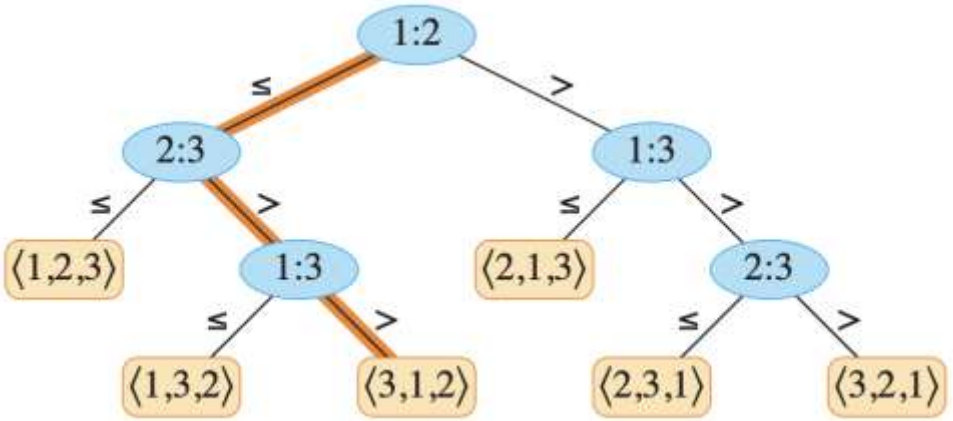
Decision trees



The decision tree for insertion sort on 3 elements

- Because any *correct sorting algorithm* must be able to produce *every permutation* of its input, each of the $n!$ permutations on n elements must appear as at least *one of the leaves of the decision tree*.

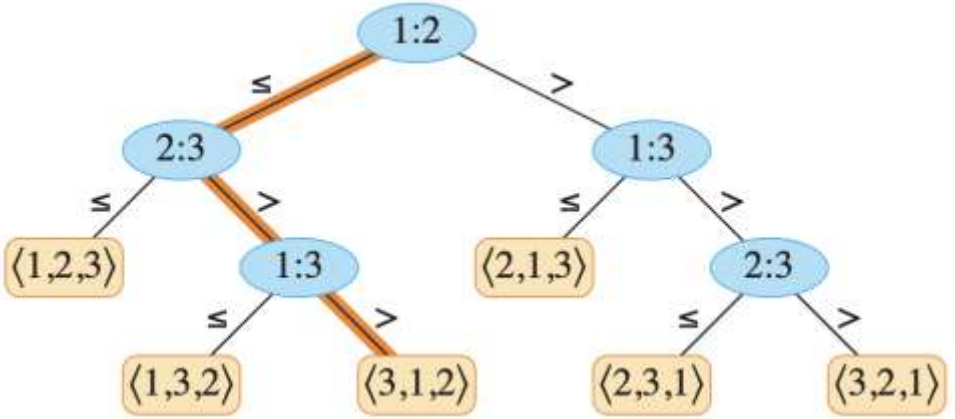
Decision trees



The decision tree for insertion sort on 3 elements

- Because any *correct sorting algorithm* must be able to produce *every permutation* of its input, each of the $n!$ permutations on n elements must appear as at least *one of the leaves of the decision tree*.
- Each of these leaves must be reachable from the root by a downward path.

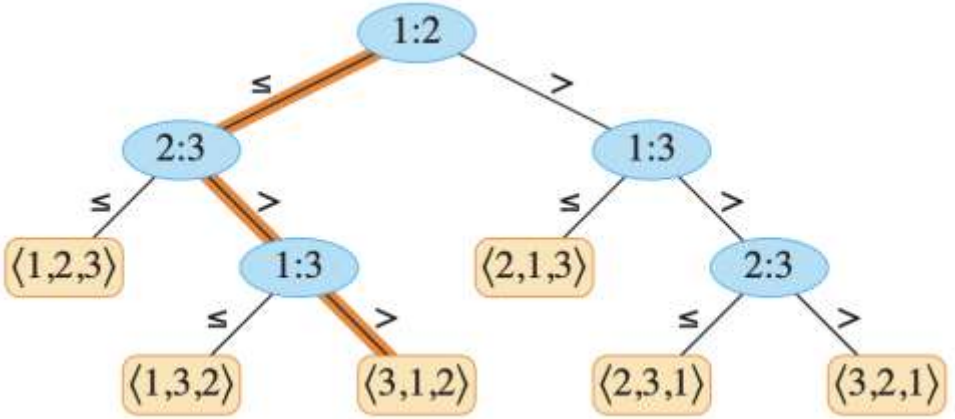
Decision trees



The decision tree for insertion sort on 3 elements

- Obs. The length of the *longest simple path* from the root of a decision tree to any of its reachable leaves represents the *worst-case number of comparisons* that the corresponding sorting algorithm performs.

Decision trees



The decision tree for insertion sort on 3 elements

- Obs. The length of the *longest simple path* from the root of a decision tree to any of its reachable leaves represents the *worst-case number of comparisons* that the corresponding sorting algorithm performs.
- Cor. A lower bound on the *height of all decision trees* in which every permutation appears as a reachable leaf is a lower bound on the *running time of any comparison sort* algorithm.

A lower bound for comparison sorts

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

A lower bound for comparison sorts

- Thm. 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 on n elements.

A lower bound for comparison sorts

- Thm. 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 on n elements. As each of the $n!$ permutations of the input appears as one or more leaves, we have $n! \leq l$.

A lower bound for comparison sorts

- Thm. 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 on n elements. As each of the $n!$ permutations of the input appears as one or more leaves, we have $n! \leq l$. Since a binary tree of height h has *no more* than 2^h leaves, we have $n! \leq l \leq 2^h$.

A lower bound for comparison sorts

- Thm. 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 on n elements. As each of the $n!$ permutations of the input appears as one or more leaves, we have $n! \leq l$. Since a binary tree of height h has *no more* than 2^h leaves, we have $n! \leq l \leq 2^h$. Hence, $h \geq \log(n!) = \Omega(n \log n)$ (by Stirling's formula).

A lower bound for comparison sorts

- Thm. 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 on n elements. As each of the $n!$ permutations of the input appears as one or more leaves, we have $n! \leq l$. Since a binary tree of height h has *no more* than 2^h leaves, we have $n! \leq l \leq 2^h$. Hence, $h \geq \log(n!) = \Omega(n \log n)$ (by Stirling's formula). From the preceding discussion, we have the required lower bound.

A lower bound for comparison sorts

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

- Note. The argument can be adapted to prove an $\Omega(n \log n)$ lower bound on the *expected running time* of ***randomized*** *comparison sort* algorithms.

A lower bound for comparison sorts

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

- Note. The argument can be adapted to prove an $\Omega(n \log n)$ lower bound on the *expected running time* of ***randomized*** *comparison sort* algorithms.

- Cor. Heapsort, merge sort, and randomized quicksort are *asymptotically optimal* comparison sorts.

Sorting in linear time: Counting sort, Radix sort, and Bucket sort

Counting sort

- Counting sort assumes that each of the n input elements is an integer *in the range* 0 *to* k.

- It runs in $\Theta(n+k)$ time, so that when $k = O(n)$, counting sort runs in $\Theta(n)$ time.

- The upper bound k on the input integers is used crucially to "beat" the $\Omega(n \log n)$ lower bound for comparison sorts.

Counting sort: The idea

- Counting sort first determines, for each input element x, the number of elements $\leq x$.

- It then uses this information to place element x directly into its position in the output array.

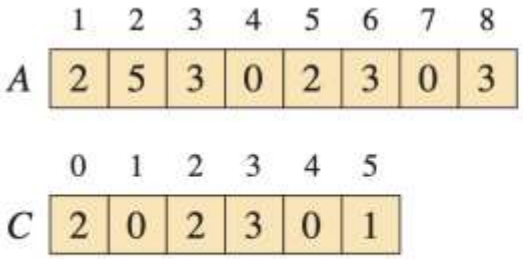
Counting sort: The idea

- Counting sort first determines, for each input element x, the number of elements $\leq x$.

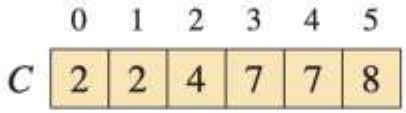
- It then uses this information to place element x directly into its position in the output array.

- A slight adjustment is required to handle the situation in which *multiple elements have the same value*. This also ensures that the algorithm is "stable".

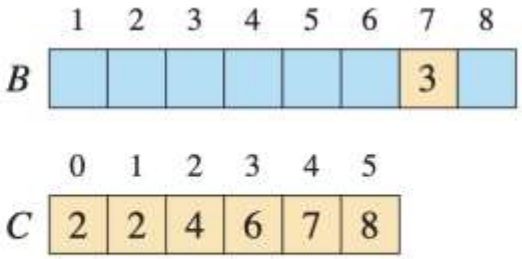
Counting sort: An example



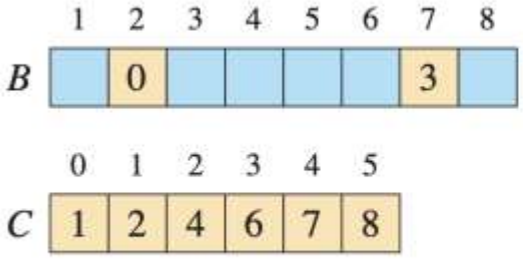
(a)



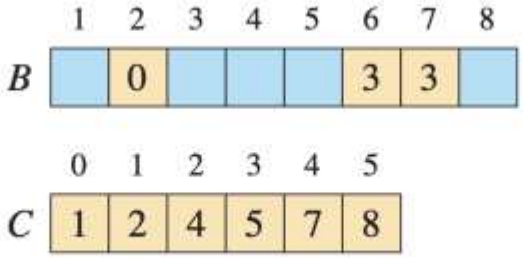
(b)



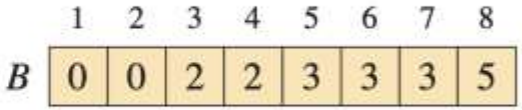
(c)



(d)



(e)



(f)

Counting sort: Pseudocode

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Counting sort: Pseudocode

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Notice the downward movement of the counter

Correctness. Follows directly from the idea.

Counting sort: Time complexity

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Time complexity. Can be easily shown to be $\Theta(n + k)$.

Counting sort: Time complexity

- In practice, we usually use counting sort when we have $k = O(n)$, in which case the run-time is $\Theta(n)$.

Counting sort: Time complexity

- In practice, we usually use counting sort when we have $k = O(n)$, in which case the run-time is $\Theta(n)$.
- Counting sort can "beat" the lower bound of $\Omega(n \log n)$ because it is *not a comparison sort*. Notice that **no** comparisons between input elements occur anywhere in the pseudocode. *The additional information about the upper bound k on the input elements is used crucially to avoid comparisons*.

Counting sort: Time complexity

- In practice, we usually use counting sort when we have $k = O(n)$, in which case the run-time is $\Theta(n)$.
- Counting sort can "beat" the lower bound of $\Omega(n \log n)$ because it is *not a comparison sort*. Notice that **no** comparisons between input elements occur anywhere in the pseudocode. *The additional information about the upper bound k on the input elements is used crucially to avoid comparisons*.
- The $\Omega(n \log n)$ lower bound *does not apply* when we depart from comparison based models.

Counting sort: Stability

- 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.
- The property of stability is usually important when *satellite data are carried around* with the elements being sorted.

Counting sort: Stability

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Notice the downward movement of the counter

Note. The algorithm *won't be stable* if j (in line 11) goes from 1 to n.

Counting sort: Stability

- 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.
- The property of stability is usually important when *satellite data are carried around* with the elements being sorted.

- Counting sort is often used as a subroutine in radix sort (which we'll discuss next). For radix sort to work correctly, counting sort must be stable.

Radix sort

- Radix sort is the algorithm used to sort *punched cards* in the first half of the 20th century. A punched card is a medium to store digital information.
- Radix sort is still used to sort *large volumes of integers* in *few digits*. It is used in high-performance GPU algorithms and in large databases to sort records that are keyed by *multiple fields*.

Radix sort

- Radix sort is the algorithm used to sort *punched cards* in the first half of the 20th century. A punched card is a medium to store digital information.
- Radix sort is still used to sort *large volumes of integers* in *few digits*. It is used in high-performance GPU algorithms and in large databases to sort records that are keyed by *multiple fields*.

- In order for radix sort to work correctly and efficiently, we *need an efficient stable sorting algorithm*.

Radix sort: The idea

- The input is a sequence of n many d-digit numbers.
- Radix sort solves the problem by sorting the numbers based on the *least significant digit first* using a *stable sorting algorithm*.

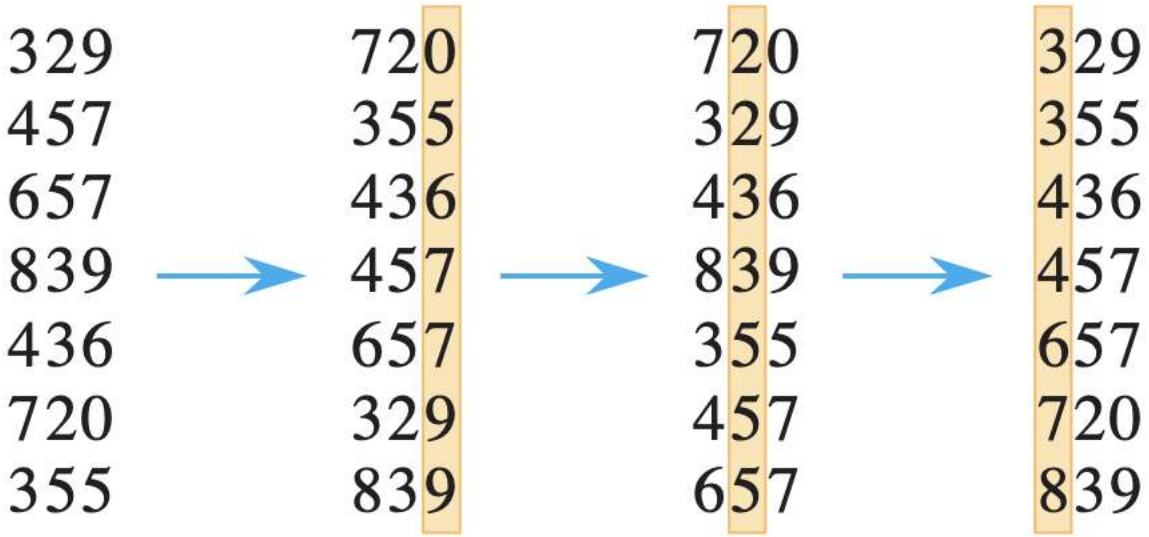
Radix sort: The idea

- The input is a sequence of n many d-digit numbers.
- Radix sort solves the problem by sorting the numbers based on the *least significant digit first* using a *stable sorting algorithm*.
- Then, it sorts the numbers again based on the *second-least significant digit*. And then, based on the *third-least significant digit*… and so on.

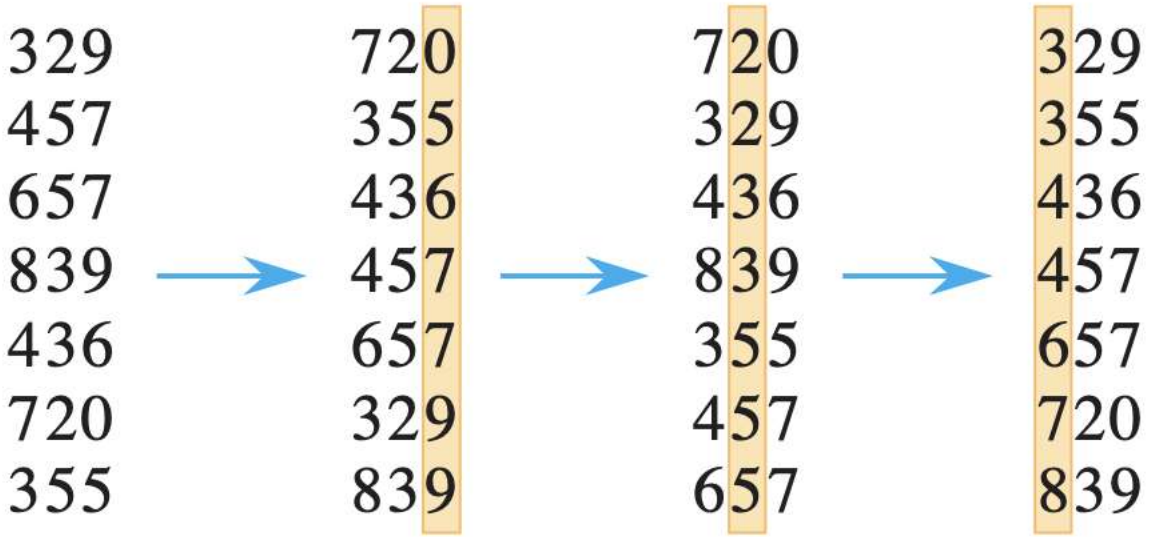
Radix sort: The idea

- The input is a sequence of n many d-digit numbers.
- Radix sort solves the problem by sorting the numbers based on the *least significant digit first* using a *stable sorting algorithm*.
- Then, it sorts the numbers again based on the *second-least significant digit*. And then, based on the *third-least significant digit*… and so on.
- The process continues until the numbers are sorted based on the *the most significant digit*.
- At this point, the numbers are fully sorted. Why?

Radix sort: An example

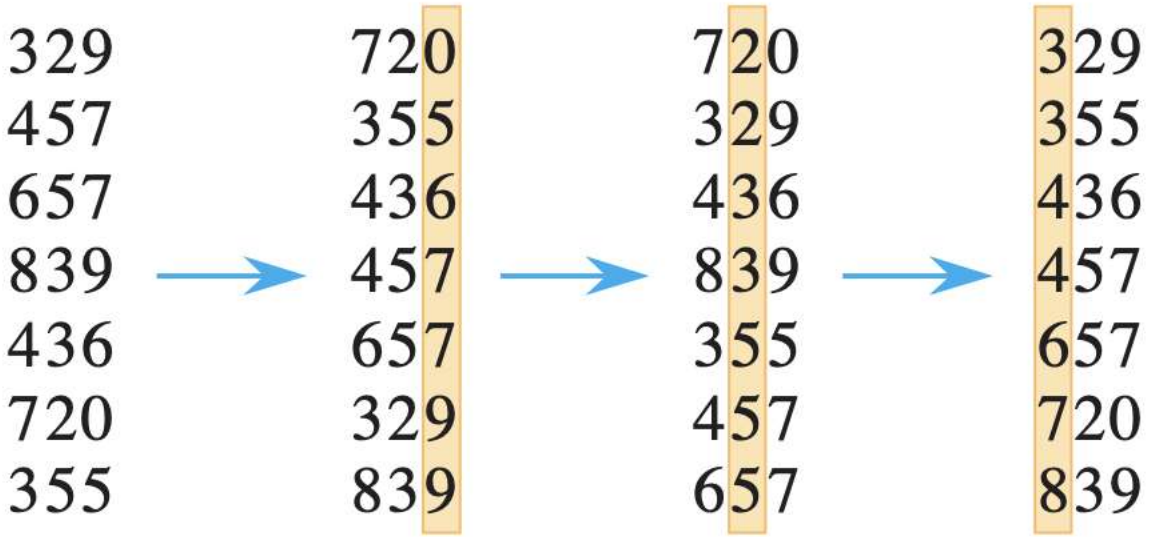


Radix sort: An example



- Correctness. The correctness of the algorithm follows easily *by induction on the digit being sorted*: Assume that after sorting the i-th l.s.d, the numbers *restricted to* the first i least significant digits are sorted.

Radix sort: An example



- Correctness. The correctness of the algorithm follows easily *by induction on the digit being sorted*: Assume that after sorting the i-th l.s.d, the numbers *restricted to* the first i least significant digits are sorted. Sorting based on the $(i+1)$-th l.s.d. using a *stable sorting algorithm* makes the numbers restricted to the first $i+1$ least significant digits sorted.

Radix sort: Pseudocode

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

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

Radix sort: Pseudocode

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

- Time complexity. Given n many d-digit numbers in which each digit can take up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta\big(d(n+k)\big)$ time if the stable sort it uses takes time $\Theta(n+k)$ time. *Counting sort* is commonly used as the stable sorting algorithm.

Radix sort: A useful lemma

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

Radix sort: A useful lemma

- Lemma. Given n many b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta\big((b/r)(n+2^r)\big)$ time if the stable sort it uses takes time $\Theta(n+k)$ 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 $k = 2^r - 1$.

Radix sort: A useful lemma

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

- Usefulness of the lemma. In practice problem 3, you'll show how to sort n integers in the range 0 to $n^3 - 1$ in $\Theta(n)$ time. (Applying Counting sort or any comparison sort directly won't help.)

Bucket sort

- Bucket sort assumes that the input is *drawn from a uniform distribution* and has an *average-case running time* of $\Theta(n)$.

Bucket sort

- Bucket sort assumes that the input is *drawn from a uniform distribution* and has an *average-case running time* of $\Theta(n)$.

- Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the *input is generated by a random process that distributes the elements uniformly and independently over the interval* $[0,1)$.

Bucket sort: The idea

- Bucket sort *divides the interval* $[0,1)$ *into* n *equal-sized subintervals*, or buckets, and then 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.

Bucket sort: The idea

- Bucket sort *divides the interval* $[0,1)$ *into* n *equal-sized subintervals*, or buckets, and then 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.

- We simply *sort the numbers in each bucket* and then go through the buckets in order.

Bucket sort: Pseudocode

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

- Note. The code uses an *auxiliary array* $B[0 : n-1]$ *of linked lists* (buckets) and assumes that there's a mechanism for maintaining such lists. We'll discuss **linked lists** in detail in a later lecture.

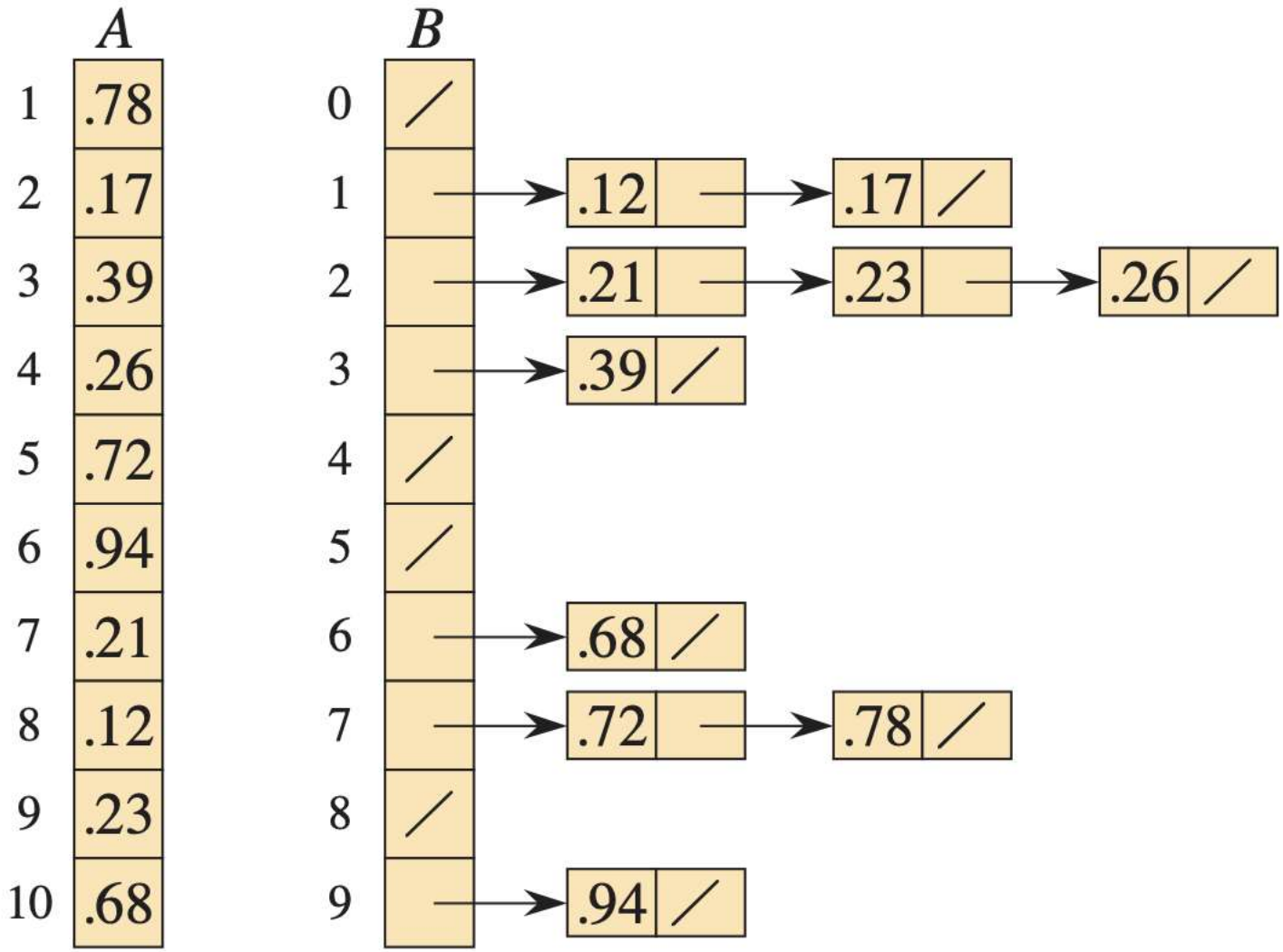
Bucket sort: Pseudocode

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

Bucket $A[i]$ goes into

- Note. The code uses an *auxiliary array* $B[0 : n - 1]$ *of linked lists* (buckets) and assumes that there's a mechanism for maintaining such lists. We'll discuss **linked lists** in detail in a later lecture.

Bucket sort: An example



Bucket sort: Time complexity

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

- Time complexity. To analyze the running time, observe that, together, all lines *except line 7* take $O(n)$ time. We need to analyze the time taken by the n calls to insertion sort in line 7.

Bucket sort: Time complexity

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

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

Bucket sort: Time complexity

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

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

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

(by linearity of expectation)

- Important note. The expectation is over the *randomness of the input*. Bucket sort algorithm ***does not*** *use randomness* (unlike randomized quicksort).

Bucket sort: Time complexity

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

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

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

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

Bucket sort: Time complexity

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

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

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

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

- Claim. $\mathrm{E}[n_i^2] = 2 - 1/n$.
- Assuming the claim, $\mathrm{E}[T(n)] = \Theta(n)$, i.e., the *average-case running time* of bucket sort is $\Theta(n)$.

Proof of the claim

- Claim. $\mathrm{E}[n_i^2] = 2 - 1/n$.
- *Proof.* View each random variable n_i as the *number of successes* in n independent trials. Success in a trail occurs when an element goes into bucket $B[i]$. Hence, success probability is $p = 1/n$ as every element is equally likely to fall in any bucket.

Proof of the claim

- Claim. $\mathrm{E}[n_i^2] = 2 - 1/n$.
- *Proof.* View each random variable n_i as the *number of successes* in n independent trials. Success in a trail occurs when an element goes into bucket $B[i]$. Hence, success probability is $p = 1/n$ as every element is equally likely to fall in any bucket.
- A *binomial distribution* counts n_i, the number of successes in the n trails.
- Hence, $\mathrm{E}[n_i] = np = 1$ & $\mathrm{Var}[n_i] = np(1-p) = 1 - 1/n$.

Proof of the claim

- Claim. $\mathrm{E}[n_i^2] = 2 - 1/n$.
- *Proof.* View each random variable n_i as the *number of successes* in n independent trials. Success in a trail occurs when an element goes into bucket $B[i]$. Hence, success probability is $p = 1/n$ as every element is equally likely to fall in any bucket.
- A *binomial distribution* counts n_i, the number of successes in the n trails.
- Hence, $\mathrm{E}[n_i] = np = 1$ & $\mathrm{Var}[n_i] = np(1-p) = 1 - 1/n$. Therefore, $\mathrm{E}[n_i^2] = \mathrm{Var}[n_i] + (\mathrm{E}[n_i])^2 = 2 - 1/n$.

Practice problems

Practice Problem 1

Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

Practice Problem 2

Suppose that the array being sorted contains only integers in the range 0 to k and that there are no satellite data to move with those keys. Modify counting sort to use just the arrays A and C, putting the sorted result back into array A instead of into a new array B.

Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a : b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

Practice Problem 3

Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

Practice Problem 4

An array A of size $n > 10$ is filled in the following way. For each element $A[i]$, choose two random variables x_i and y_i uniformly and independently from $[0, 1)$. Then set

$$A[i] = \frac{\lfloor 10x_i \rfloor}{10} + \frac{y_i}{n} .$$

Modify bucket sort so that it sorts the array A in $O(n)$ expected time.

Practice Problem 5

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time. (The desired order is the standard alphabetical order: for example, `a` $<$ `ab` $<$ `b`.)