ADVANCED ANALYSIS OF ALGORITHMS
CPS 5440

UNIT 6: SORTING IN LINEAR TIME: COUNTING SORT, RADIX SORT, AND BUCKET SORT.

COMPARISON SORTING REVIEW

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort sub-arrays
    - Linear-time merge step
  - Pro's:
    - $O(n \lg n)$ worst case - *asymptotically optimal for comparison sorts*
    - *Stable sort algorithm*
  - Con's:
    - Doesn't sort in place

COMPARISON SORTING REVIEW

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - Pro's:
    - $O(n \lg n)$ worst case - *asymptotically optimal for comparison sorts*
    - Sorts in place
  - Con's:
    - Fair amount of shuffling memory around
    - Not stable

COMPARISON SORTING REVIEW

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two sub-arrays, recursively sort
    - All of first sub-array $<$ all of second sub-array
  - Pro's:
    - $O(n \lg n)$ average case
    - Sorts in place
    - Fast in practice (*why?*)
  - Con's:
    - Not stable
    - $O(n^2)$ worst case
      - Naïve implementation: worst case on sorted input
      - Good partitioning makes this very unlikely.

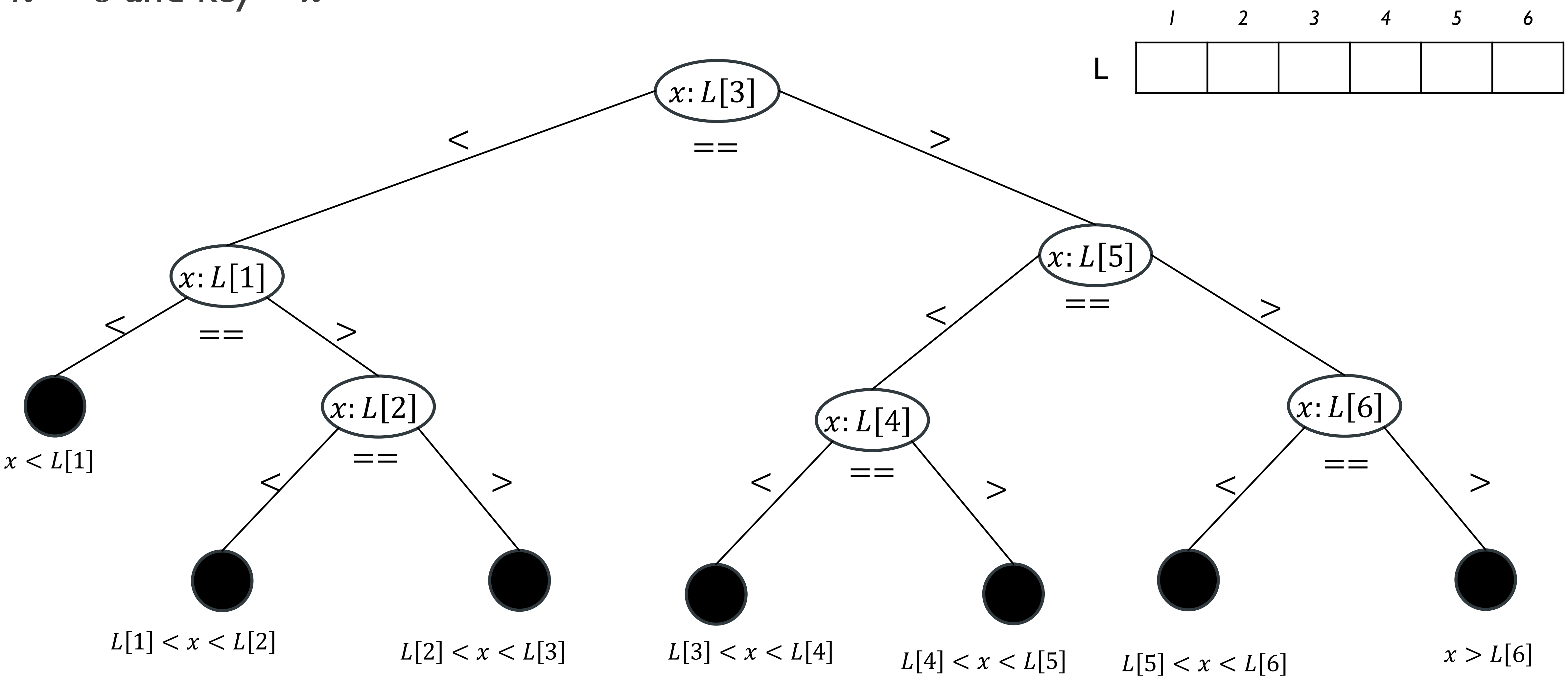
COMPARISON SORTING REVIEW

- Insertion sort:
  - Pro's:
    - Easy to code
    - Fast on small inputs (less than ~50 elements)
    - Fast on nearly-sorted inputs
  - Con's:
    - $O(n^2)$ worst case
    - $O(n^2)$ average case
    - $O(n^2)$ reverse-sorted case

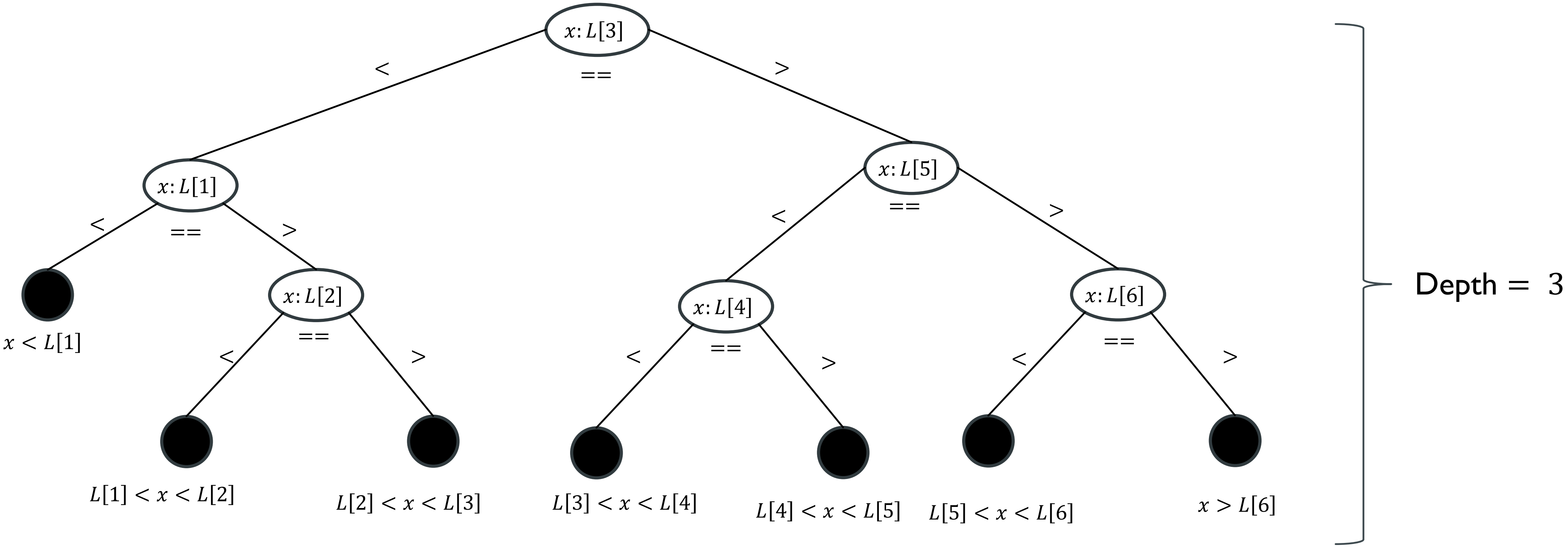
DECISION TREE

- Any comparison algorithm can be viewed as a tree of all possible comparisons and their outcomes, and resulting answers, for any particular n

- Draw the decision tree for binary search on a sorted list of six elements.
  $n = 6$ and key = x



LOWER BOUNDS FOR SEARCHING



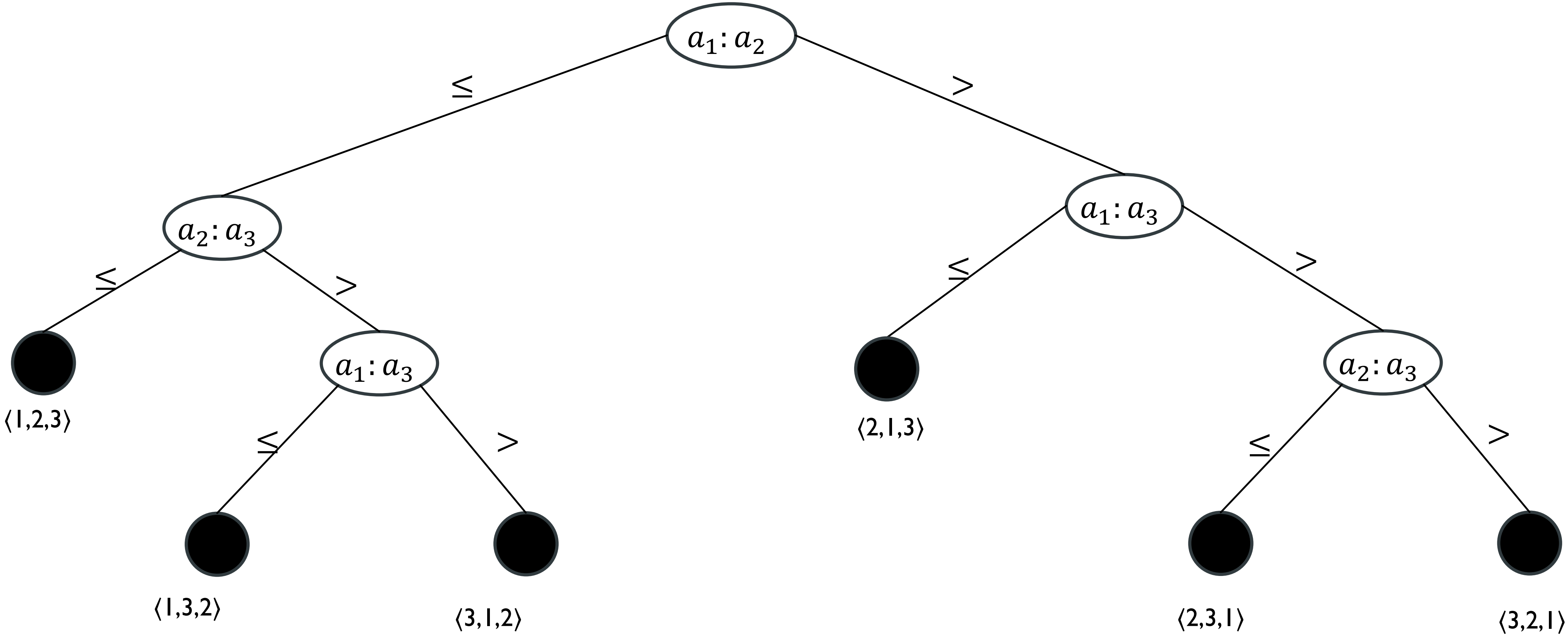
SEARCHING LOWER BOUND

- To find any leaf ($i.e.,$ possible solution including the optimal one), the height of tree comparisons must be performed
- Searching lower bound:
  - Finding a given item among them in comparison model requires $\Omega\ (\lg n)$
- Proof: Decision tree is binary and must have $\geq n$ leaves, one for each answer
  - $2^h\ \geq n$
  - $\Rightarrow height\ \geq \lg n$

LOWER BOUNDS FOR SORTING

- Draw the decision tree for insertion sort

	1	2	3
A	a_1	a_2	a_3



SORTING LOWER BOUND

- Since there are $n!$ permutations of n elements, each permutation representing a distinct sorted order, the tree must have at least $n!$ leaves.
- Since a binary tree of height h has no more than 2^h leaves, we have
  - $2^h \geq n!$
  - $\Rightarrow height \geq \lg(n!)$

SORTING LOWER BOUND

- $h \geq \lg(n!)$
- $= \lg(n\ (n-1)(n-2)(n-3)\ldots 1)$
- $= \lg(n) + \lg(n-1) + \lg(n-2) + \lg(n-3) + \cdots + \lg(2) + \lg(1)$
- $\sum_{i=1}^{n} \lg(i)$
- $\geq \sum_{i=\frac{n}{2}}^{n} \lg(i)$
- $\geq \sum_{i=\frac{n}{2}}^{n} \lg(i)$
- $= \sum_{i=\frac{n}{2}}^{n} \lg(\frac{n}{2})$
- $= \sum_{i=\frac{n}{2}}^{n} \lg(n) - 1$
- $= \frac{n}{2} \lg n - \frac{n}{2}$
- $\Omega(n \lg n)$

NON-COMPARISON BASED SORTING

- Many times we have restrictions on our keys
  - Deck of cards: Ace→King and four suites
  - Social Security Numbers
  - Employee ID's
- We will examine three algorithms which under certain conditions can run in $O(n)$ time.
  - Counting sort
  - Radix sort
  - Bucket sort

COUNTING SORT.

COUNTING SORT

- Depends on assumption about the numbers being sorted
  - Assume numbers are in the range $0..k$
- The idea is: for each input element x to count how many elements are less than x, and use this information to place x to the right place in the output sequence
- The algorithm:
  - Input: $A[1..n]$, where $A[j] \in \{1, 2, 3, \ldots, k\}$
  - Output: $B[1..n]$, sorted (not sorted in place)
  - Also: $C[0..k]$, for auxiliary storage

COUNTING SORT

```
CountingSort(A, B, 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]] += 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] contains the number of elements less than or equal to i
for j=n downto 1
    B[C[A[j]]] = A[j];
    C[A[j]] -= 1;
```

COUNTING SORT EXAMPLE

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

$k \in \{0, \ldots, 5\}$

1 let C[0..k] be a new array

	0	1	2	3	4	5
C						

```
2. for i=0 to k
3.      C[i]= 0;
```

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

COUNTING SORT EXAMPLE

```
4.  for j=1 to n
5.       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
$j = 1$	C	0	0	1	0	0	0
$j = 2$	C	0	0	1	0	0	1
$j = 3$	C	0	0	1	1	0	1
$j = 4$	C	1	0	1	1	0	1
$j = 5$	C	1	0	2	1	0	1
$j = 6$	C	1	0	2	2	0	1
$j = 7$	C	2	0	2	2	0	1
$j = 8$	C	2	0	2	3	0	1

COUNTING SORT EXAMPLE

```
7.  for i=1 to k
8.       C[i] = C[i] + C[i-1];
```

		0	1	2	3	4	5
	C	2	0	2	3	0	1
$i = 1$	C	2	2	2	3	0	1
$i = 2$	C	2	2	4	3	0	1
$i = 3$	C	2	2	4	7	0	1
$i = 4$	C	2	2	4	7	7	1
$i = 5$	C	2	2	4	7	7	8

COUNTING SORT EXAMPLE

```
for j=n down to 1
    B[C[A[j]]] = 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	2	4	7	7	8

		1	2	3	4	5	6	7	8
$j = 8$	B							3	
$j = 7$	B		0					3	
$j = 6$	B		0				3	3	
$j = 5$	B		0		2		3	3	
$j = 4$	B	0	0		2		3	3	
$j = 3$	B	0	0		2	3	3	3	
$j = 2$	B	0	0		2	3	3	3	5
$j = 1$	B	0	0	2	2	3	3	3	5

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

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

COUNTING SORT

```
CountingSort(A, B, 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]] += 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] contains the number of elements less than or equal to i
    for j=n down to 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Takes time $O(k)$

Takes time $O(n)$

What is the running time?

$$O(n+k)$$

COUNTING SORT

- Total time: $\boldsymbol{O(n+k)}$
  - Works well if $k = O(n)$ or $k = O(1)$
- This sorting is ***stable***.
  - A sorting algorithm is ***stable*** when numbers with the same values appear in the output array in the same order as they do in the input array.

COUNTING SORT REVIEW

- **Assumption:** input taken from **small** set of **numbers** of size k
- Basic idea:
  - Count number of elements less than you for each element.
  - This gives the position of that number – similar to selection sort.
- Pro's:
  - Fast / Stable
  - Asymptotically fast - $O(n + k)$
  - Simple to code
- Con's:
  - Doesn't sort in place.
  - Elements must be integers.
  - Requires $O(n + k)$ extra storage.

RADIX SORT.

RADIX SORT

- Sort on the Least Significant Digit, then the second LSD, etc.
- `RadixSort(A, d)`

```
        for i=1 to d
           StableSort(A) on digit i
```

RADIX SORT EXAMPLE

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

- The operation of radix sort on a list of seven 3-digit numbers.
- The leftmost column is the input.
- The remaining columns show the list after successive sorts on increasingly significant digit positions;
- Shading indicates the digit position sorted on to produce each list from the previous one.

RADIX SORT CORRECTNESS

- Sketch of an inductive proof of correctness (induction on the number of passes):
  - Assume lower-order digits $\{j\colon\ j < i\}$ are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

RADIX SORT

- *What sort is used to sort on digits?*
- Counting sort is the obvious choice:
  - Sort n numbers on digits that range from $1 \ldots b$
  - Time: $O(n \ + b)$
  - b: base (e.g., base 10)
- Each pass over n numbers with d digits takes time $O(n + b)$, so total time $O(dn + db)$
  - d = #digits = $lg_b k + 1$
  - When d is constant and $k = O(n)$, takes $O(n)$ time

RADIX SORT

- $O(dn + db)$
- $= O\left((n + b)lg_b k\right)$
- $(n + b)$ to be minimized $\Rightarrow b = \Theta\ (n)$
- $\Rightarrow O\left((n + n)lg_n k\right)$
- When k is polynomial on $n \Rightarrow O\left((n + n)lg_n n^c\right) = O\ (n)$

RADIX SORT REVIEW

- **Assumption:** input has d digits ranging from 0 to k
- Basic idea:
  - Sort elements by digit, starting with the *least* significant
  - Use a stable sort (like counting sort) for each stage
- Pros:
  - Asymptotically fast ($i.e.$, $O(n)$ when d is constant and $k = O(n)$)
  - Simple to code
- Con's:
  - Doesn't sort in place
  - Not a good choice for floating point numbers or arbitrary strings.

BUCKET SORT.

BUCKET SORT

- **Assumption**: input elements distributed uniformly over some known range, e.g., $[0,1)$, so all elements in A are greater than or equal to 0 but less than 1 .

```
Bucket-Sort(A)
1. n = length[A]
2. for i = 1 to n
3.        do insert A[i] into list B[floor of n*A[i]]
4. for i = 0 to n-1
5.        do sort list i with Insertion-Sort
6. Concatenate lists B[0], B[1],…,B[n-1]
```

BUCKET SORT

Bucket-Sort(A, *x*, *y*)

1. divide interval *[x, y)* into *n* equal-sized subintervals (buckets)
2. distribute the *n* input keys into the buckets
3. sort the numbers in each bucket (e.g., with insertion sort)
4. scan the (sorted) buckets in order and produce output array

<u>Running time of bucket sort:</u> $O(n)$ expected time

Step 1: $O(1)$ for each interval = $O(n)$ time total.

Step 2: $O(n)$ time.

Step 3: The expected number of elements in each bucket is $O(1)$, so total is $O(n)$

Step 4: $O(n)$ time to scan the n buckets containing a total of n input elements

BUCKET SORT EXAMPLE

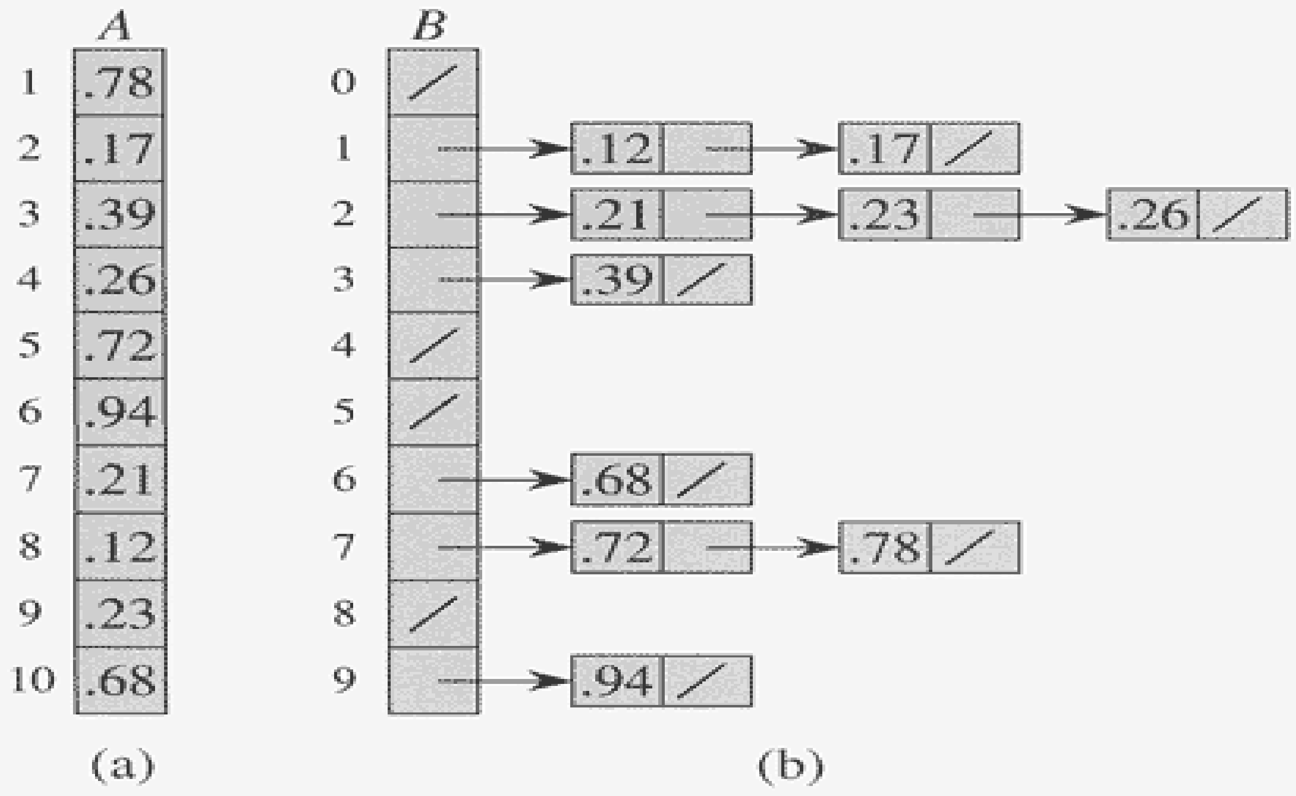


Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1\,..\,10]$. **(b)** The array $B[0\,..\,9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

BUCKET SORT REVIEW

- **Assumption:** input is uniformly distributed across a range
- Basic idea:
  - Partition the range into a fixed number of buckets.
  - Toss each element into its appropriate bucket.
  - Sort each bucket.
- Pro's:
  - Fast
  - Asymptotically fast (i.e., $O(n)$ when distribution is uniform)
  - Simple to code
  - Good for a rough sort.
- Con's:
  - Doesn't sort in place

SUMMARY OF LINEAR SORTING

	worst-case	average-case	best-case	in place
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	no
Radix Sort	$O(d(n + k'))$	$O(d(n + k'))$	$O(d(n + k'))$	no
Bucket Sort	$O(n)$	$O(n)$	$O(n)$	no

Counting sort assumes input elements are in range $[0,1,2,..,k]$ and uses array indexing to count the number of occurrences of each value.

Radix sort assumes each integer consists of d digits, and each digit is in range $[1,2,..,k']$.

Bucket sort requires advance knowledge of input distribution (sorts n numbers uniformly distributed in range in $O(n)$ time).

QUESTIONS/ANSWERS