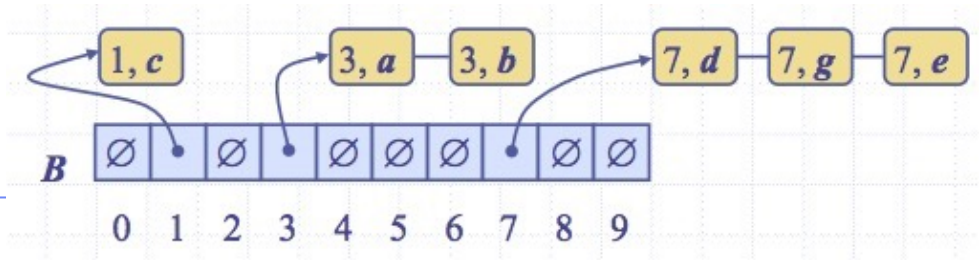
Lesson 8
PriorityQueueSort and RadixSort:
Discovering the Range of Natural Law



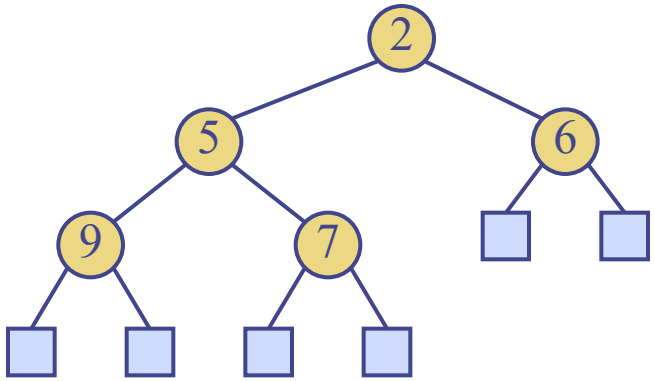
Wholeness of the Lesson

All the comparison-based sorting algorithms we have seen so far have at least one worst case for which running time is Ω(nlog n). Bucket Sort and its relatives, under suitable conditions, run in linear time in the worst case, but are not comparison-based algorithms. *Science of Consciousness*: Each level of existence has its own laws of nature. The laws of nature that operate at one level of existence may not apply to other levels of existence, e.g., macroscopic vs. subatomic, classical vs. quantum field level.

Overview

- Priority Queue ADT
- Sorting with a Priority Queue
- Bucket Sort
  - Each bucket contains elements with the same key
  - Disadvantage: requires too many buckets
- Radix Sort
  - By pairing with Bucket Sort, the number of buckets is reduced to a practical number
- Bucket Sort (Generic)
  - Number of buckets is reduced by allowing each bucket to contain keys within a range rather than one key per bucket
  - Disadvantage: Each bucket has to be sorted ==> buckets need to contain approximately the same number of elements

Priority Queues



Priority Queue ADT

- A priority queue stores a collection of sortable items
- Main methods of the Priority Queue ADT (Java Interface)
  - insertItem(k, e)
    inserts the item (k, e)
  - removeMin()
    removes the item with smallest key and returns its associated element
- Additional methods
  - minKey()
    returns, but does not remove, the smallest key of an item
  - minValue()
    returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market
  - Sorting

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- The Comparator Method returns a number as its return type indicating the relative size of its input arguments
  - compare(x, y)
  - returns a negative number if x<y, zero if x=y, and a positive number if x>y

Sorting with a Priority Queue

- We can use a priority queue to sort a set of comparable elements
  - Insert the elements one by one with a series of insertItem(e) operations
  - Remove the elements in sorted order with a series of removeMin() operations
- The running time of this sorting method depends on the priority queue implementation. Why?
- Would we need to change $S.remove(S.first())$ for efficiency, if S is a Sequence? What about $S.remove(S.last())$?

Algorithm *PQ-Sort*(*S*, *C*)
Input List *S*, comparator *C* for the elements of *S*
Output Sequence *S* sorted in increasing order according to *C*

PQ ← new priority queue using *C*	
while *S.size*() > 0 **do**	O(n)
e ← *S.remove* (S.first())	O(n)
PQ.insertItem(*e*, *e*)	O(n*?)
while *PQ.size*() > 0 **do**	O(n)
e ← *PQ.removeMin*()	O(n*?)
S.insertLast(*e*)	O(n)

Sequence-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the end of the sequence
  - removeMin, minKey, and minValue take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey, and minValue take $O(1)$ time since the smallest key is at the beginning of the sequence (requires circular array)

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

- Running time of Selection-sort:
  - Inserting the elements into the priority queue with n *insertItem* operations take $O(n)$ time
  - Removing the elements in sorted order from the priority queue with n *removeMin* operations take time proportional to

$$n + \ldots + 2 + 1$$

- Selection-sort runs in $O(n^2)$ time

Recall SelectionSort

Algorithm *SelectionSort (arr)*

Input Array *arr*

Output elements in *arr* are in sorted order

```
for i ← 0 to arr.length - 2 do
    min ← arr[i]
    minIndex ← i
    for j ← i + 1 to arr.length - 1 do // search for next smallest element
        if arr[j] < min then
            min ← arr[j]
            minIndex ← j
    if i != minIndex then
        swapElements(arr, i, minIndex) // place element in sorted location
```

Algorithm *swapElements(arr, i, j)*

```
temp ← arr[i]
arr[i] ← arr[j]
arr[j] ← temp
```

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence

- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with n *insertItem* operations take time proportional to

$$1 + 2 + \ldots + n$$

  - Removing the elements in sorted order from the priority queue with a series of n *removeMin* operations take $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Recall InsertionSort

Algorithm *InsertionSort*(*arr*)
Input Array *arr*
Output elements in *arr* are in sorted order

```
for i ← 1 to arr.length - 1 do
    j ← i
    insertElem ← arr[i]
// this loop corresponds to inserting into the PQ
    while j > 0 ∧ insertElem < arr[j - 1] do
        arr[j] ← arr[j - 1]        // shift element to right
        j ← j – 1
    arr[j] ← insertElem
```

Analysis of Heap Operations

- insertElem(e)
- removeMin()
- minimum()
- Helpers
  - upheap
  - downHeap

Analysis of Heap-based Priority Queue Operations

- insertItem(k, e)
- removeMin()
- minKey()
- minValue()
- size()
- isEmpty()

Analysis of Sorting with a Heap-based Priority Queue

- What is the running time of this sorting method if the priority queue is implemented as a Heap?

Algorithm *PQ-Sort*(*S*, *C*)
Input List *S*, comparator *C* for the elements of *S*
Output Sequence *S* sorted in increasing order according to *C*

PQ ← new priority queue using *C*
while *S.size*() > 0 **do**
e ← *S.remove* (S.*first*())
PQ.insertItem(*e*, *e*)
while *PQ.size*() > 0 **do**
e ← *PQ.removeMin*()
S.insertLast(*e*)

Analysis of Heap-Based Priority Queue

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insertItem and removeMin take $O(\log n)$ time
  - methods size, isEmpty, minKey, and minElement take $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Main Point

1. A heap is a binary tree that stores *keys* at each internal node and maintains the *heap-order* property on each path from root to leaf and the tree is *balanced*. The Heap ADT is the most efficient way of implementing a Priority Queue. The difference between a PQ and a Heap is that the PQ stores key-value items, rather than only keys.

Science of Consciousness: Pure consciousness is the field of wholeness, perfectly orderly, and complete. Experience of pure consciousness is the basis for efficient, life-supporting action that benefits individual and society.

Comparison-Based Sorting Algorithms

- All sorting algorithms discussed so far have used comparison as a central operation
- So far, all sorting algorithms discussed have a worst-case running time of $\Omega(n \log n)$
- It can be proven that sorting by key comparison requires at least nlog n comparisons to sort n elements, i.e., $\Omega(n \log n)$

Specialized Sorting: Bucket Sort

- The Context: Suppose we have an array A of n distinct integers a_1, a_2, …, a_n, all lying in the range 0..m-1.
- **Sorting strategy:**
  1. Create an array *bucket[]* of size *m*, entries initialized to 0. Create an output array *B* of size *n*
  2. Scan A. When a_i is encountered, increment the value *bucket*[a_i]
  3. Scan *bucket[]*. For each $j < m$ for which *bucket[j]* > *0*, copy *j* into next available slot in *B*.
  4. *Handling Duplicates (simple case):* Same algorithm, except in loading array B at the end, if bucket[j] = k, insert k copies of a_j into B.

Example of BucketSort

Input array A: [5, 8, 1, 2, 3, 5]

All values are ranged in [0,8], so m=9

bucket[] has size 9.

bucket[]:

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

Example of BucketSort

Input array A: [5, 8, 1, 2, 3, 5]

Scan the input array, When *A[i]* is encountered, increment the value *bucket[A[i]].*

bucket[]:

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

Example of BucketSort

Input array A: [5, 8, 1, 2, 3, 5]

Scan the input array, When *A[i]* is encountered, increment the value *bucket[A[i]].*

bucket[]:

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

Example of BucketSort

Input array A: [5, 8, 1, 2, 3, 5]

Scan the input array, When *A[i]* is encountered, increment the value *bucket[A[i]].*

bucket[]:

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

Example of BucketSort

Input array A: [5, 8, 1, 2, 3, 5]

Scan the input array, When *A[i]* is encountered, increment the value *bucket[A[i]].*

bucket[]:

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

Example of BucketSort

Input array A: [5, 8, 1, 2, 3, 5]

Scan the input array, When *A[i]* is encountered, increment the value *bucket[A[i]].*

bucket[]:

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

Example of BucketSort

Input array A: [5, 8, 1, 2, 3, 5]

Scan the input array, When *A[i]* is encountered, increment the value *bucket[A[i]].*

bucket[]:

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

Example of BucketSort

Input array A: [5, 8, 1, 2, 3, 5]

bucket[]:

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

Output array B of size n: Scan *bucket[].* For each *bucket[j] > 0*, copy j into next available slot in B.

1	**2**	**3**	**5**	**5**	**8**
0	1	2	3	4	5

Analysis of Bucket Sort

Step 1 requires O(m+n) → Create bucket array m size + output array n size.
Step 2 requires O(n) → Scan the Input array and update in the Bucket array.
Step 3 requires O(m) → Scan the Bucket the array and store the input value in the output array.

Specialized Sorting: Bucket Sort

- The Context: Suppose we have an array A of n distinct integers a_1, a_2, …, a_n, all lying in the range 0..m-1.
- **Sorting strategy:**
  1. Create an array *bucket[]* of size *m*, entries initialized to 0. Create an output array *B* of size *n*
  2. Scan A. When a_i is encountered, increment the value *bucket*[a_i]
  3. Scan *bucket[]*. For each $j < m$ for which *bucket[j]* > *0*, copy *j* into next available slot in *B*.
  4. *Handling Duplicates (simple case):* Same algorithm, except in loading array B at the end, if bucket[j] = k, insert k copies of a_j into B.

Analysis of Bucket Sort

** Therefore, BucketSort runs in O(m+n).
When m is O(n), BucketSort runs in O(n)

Example: When m is O(n), BucketSort runs in O(n).
Example: Sort 1000 integers all lying in the range 0..1999.

- *Handling Duplicates (simple case):* Same algorithm, except in loading array B at the end, if bucket[j] = k, insert k copies of a_j into B.

- If m is too big (i.e., not O(n)), BucketSort is not useful because scanning the bucket array is too costly

Bucket-Sort-Handling Duplicates

- Let S be a list containing n (key,element) items with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of lists (buckets)
  - Phase 1: Empty list L by moving each item (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the items of bucket $B[i]$ to the end of list L
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time
- Bucket-sort takes $O(n + N)$ time

Algorithm *bucketSort*(L, N)

Input List L of (key, element) items with keys in the range $[0, N-1]$

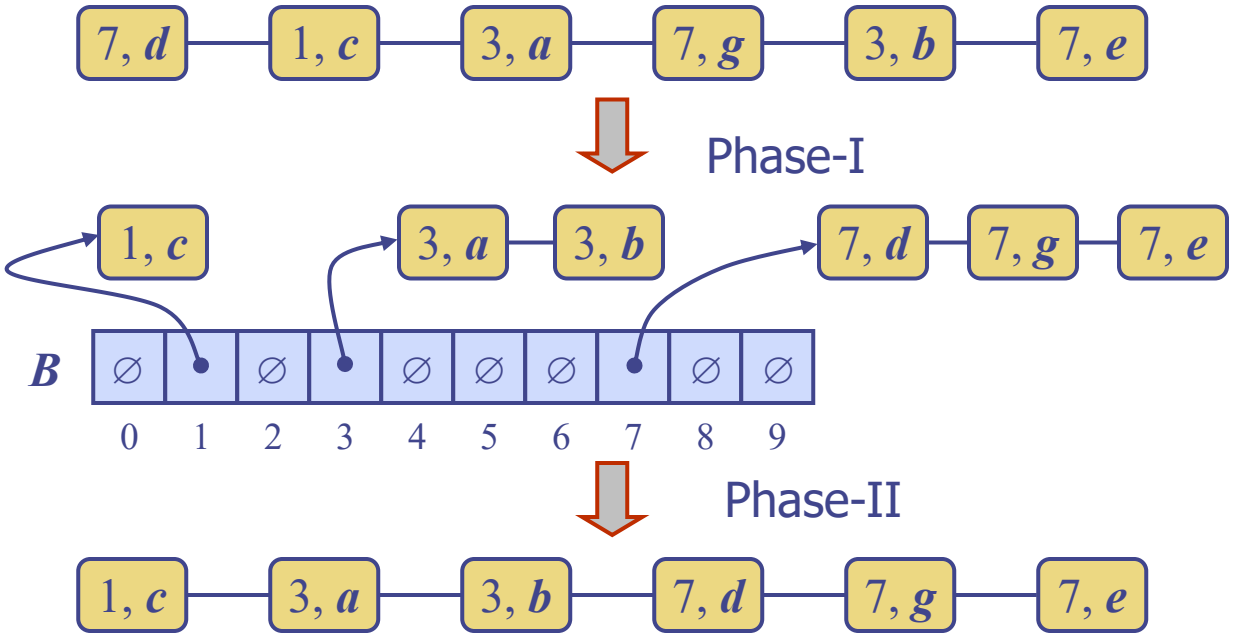
Output List L sorted by increasing keys

$B \leftarrow$ array of N empty lists	N
while ! *L.isEmpty*() **do**	n
$(k, o) \leftarrow$ *L.remove*(*L.first*())	n
B[*k*].*insertLast*((k, o))	n
for $i \leftarrow 0$ **to** $N-1$ **do**	N
while ! *B*[*i*].*isEmpty*() **do**	N+n
$f \leftarrow$ *B*[*i*].*first*()	n
$(k, o) \leftarrow$ *B*[*i*].*remove*(*f*)	n
L.insertLast((k, o))	n

T(n) is O(N+n)

Example

- Key range $[0, 9]$



Main Point

2. BucketSort is an example of a sorting algorithm that runs in O(n). This is possible only because BucketSort does not rely primarily on key comparisons in order to perform sorting.

   Science of Consciousness: This phenomenon illustrates two points from SCI. First, to solve a problem, often the best approach is to bring a new element to the situation (in this case, an array of buckets); this is the Principle of the Second Element. The second point is that different laws of nature are applicable at different levels of creation. Deeper levels are governed by more comprehensive and unified laws of nature.

Radix-Sort

- What do we do if BucketSort isn't efficient because the range is too big?
  - Since it would run in O(n+m)
- Radix-sort is a refinement of BucketSort that uses multiple bucket sorts
- Radix Meaning: The base of a number system,
  - e.g., base 10 in decimal numbers or base 2 in binary numbers. Also known as radix 10 or radix 2

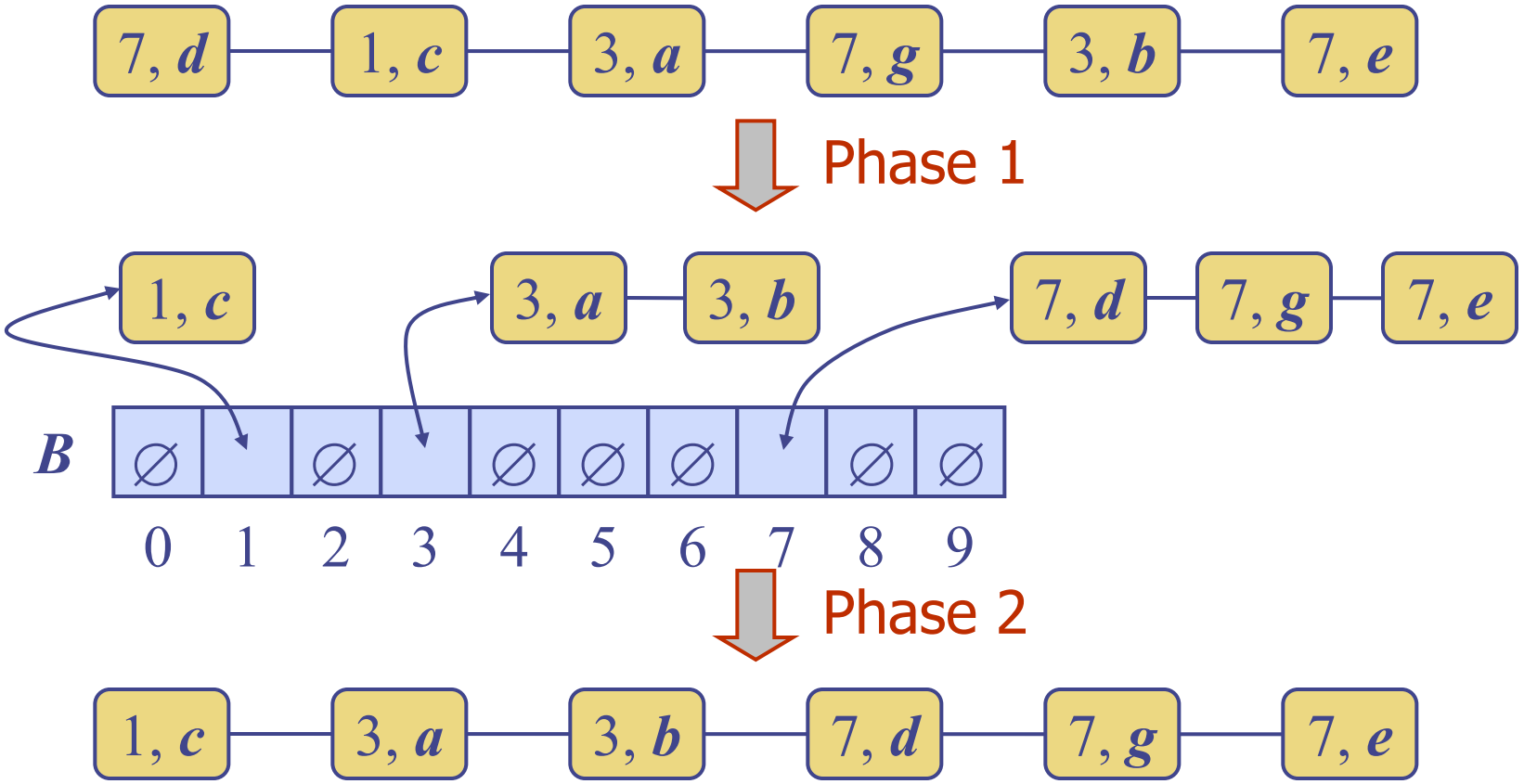
Stable Sorting

- **Stable** Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

- Not all sorting algorithms preserve this property

Example

- Key range $[0, 9]$



Radix-Sort

- The base of a number system
  - Eg: base 10 in decimal number numbers(radix 10) or base 2(radix 2) in binary numbers.
- Radix-sort is a generalization of BucketSort that uses multiple bucket arrays.
- Idea:
  - Bucket sort on one digit at a time
    - Number of buckets = Maximum number of digits
      - Eg: input value from the array is 5000, #buckets = 4
    - Starting with least significant digit
    - Keeping sort stable
  - Do one pass per digit
  - After k passes (digits), the last k digits are sorted

Example with Radix = 10

- Radix = 10

#Buckets = 3 [max digits]

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

Input: 478
537
9
721
3
38
143
67

First pass: bucket sort by ones digit (modulus by 10)

Order now: 721
3
143
537
67
478
38
9

Example

- Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Order was: 721, 3, 143, 537, 67, 478, 38, 9

Second pass: stable bucket sort by tens digit

Order now: 3, 9, 721, 537, 38, 143, 67, 478

Example

- Radix = 10

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

0	1	2	3	4	5	6	7	8	9
3 9 38 67	143			478	537		721		

Order was:

3
9
721
537
38
143
67
478

Third pass: stable
bucket sort by 100s digit

Order now:

3
9
38
67
143
478
537
721

Example with Radix = 7

- Example: Sort 48, 1, 6, 23, 37, 19 ,21
- Strategy is to use 2 bucket arrays, each of size 7 (7 is the *radix*)
- Based on observation that every k in [0,48] can be written:
  k = 7q + r where 0 <= q < 7, 0 <= r < 7
- **Procedure**:
  - Pass #1: Scan initial array and place values in the "remainders" *bucket r[]*
    – put *x* in *r[i]* if *x* % 7 = *i*. (Need to assume bucket array consists of lists)
  - Pass #2: Scan *r[],* reading from front of each list to back, and place values in the "quotients" *bucket q[]*
    – put *x* in *q[i]* if *x/7* = *i*.
  - Output: Scan *q[],* again reading lists front to back

Example with Radix = 7

- Example: Sort → [48, 1, 6, 23, 37, 19 ,21] , #Buckets = 2 [Max digits]

1. Create two buckets r and q with the size 7

r[0] q[0]
r[1] q[1]
r[2] q[2]
r[3] q[3]
r[4] q[4]
r[5] q[5]
r[6] q[6]

2. put x in r[i]
if x % 7 = i

r[0] = 21 % 7 = 0
r[1] = 1 % 7 = 1
r[2] = 23 % 7 = 2
37 % 7 = 2
r[3] =
r[4] =
r[5] = 9 % 7 = 5
r[6] = 48 % 7 = 6
6 % 7 = 6

r[0]: 21
r[1]: 1
r[2]: 23, 37
r[3]:
r[4]:
r[5]: 19
r[6]: 48, 6

3. Scan from r[i]
put *x* in *q[i]* if *x/7* = *i*

q[0] = 1 / 7 = 0
6 / 7 = 0
q[1] =
q[2] = 19 / 7 = 2
q[3] = 21 / 7 = 3
23 / 7 = 3
q[4] =
q[5] = 37 / 7 = 5
q[6] = 48 / 7 = 6

q[0]: 1, 6
q[1]:
q[2]: 19
q[3]: 21, 23
q[4]:
q[5]: 37
q[6]: 48

- Finally, we scan the "quotients" bucket array (q) and read the lists from front to back to get the sorted output.
- Sorted Output: 1, 6, 19, 21, 23, 37, 48

Analysis of RadixSort

- Running time is O(n+r) where
  - n = size of initial array (example: 7)
  - r = size of each bucket array (ex: 3)
- RadixSort runs in O(n) when size of each bucket array is O(n).

Summary of Sorting Algorithms

Algorithm	Time	Notes (pros & cons)
Shell-sort	$O(n^{3/2})$	fast for fairly large inputs, needs efficient random access
insertion-sort	$O(n^2)$	excellent for small inputs, fast for 'almost' sorted
heap-sort	$O(n \log n)$	in-place, needs efficient random access, optimal for key comparisons
PQ-sort	$O(n \log n)$	not in-place, fast but needs supplemental data structure
bucket-sort radix-sort	$O(m+\mathrm{n})$ $O(d(n+\mathrm{N}))$	if integer keys & keys known, could be faster than heap-sort

d(n+N) d- number of digits in the largest number
n is the number of elements in the input array
N is the base of the number system(radix) used for sorting.

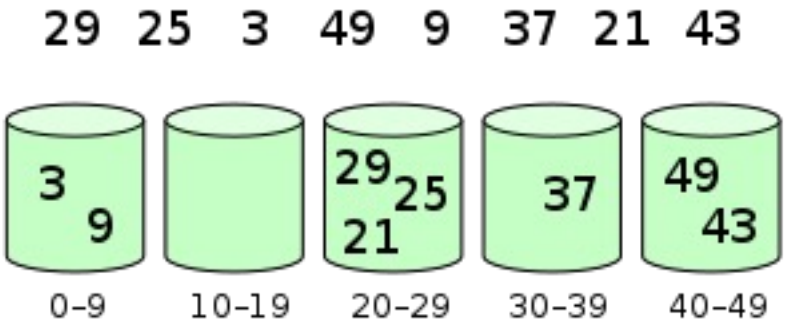
Main Point

3. A radix-sort does successive bucket sorts, one for each "digit" in the key beginning with the least significant digit going up to the most significant; it has linear running time.

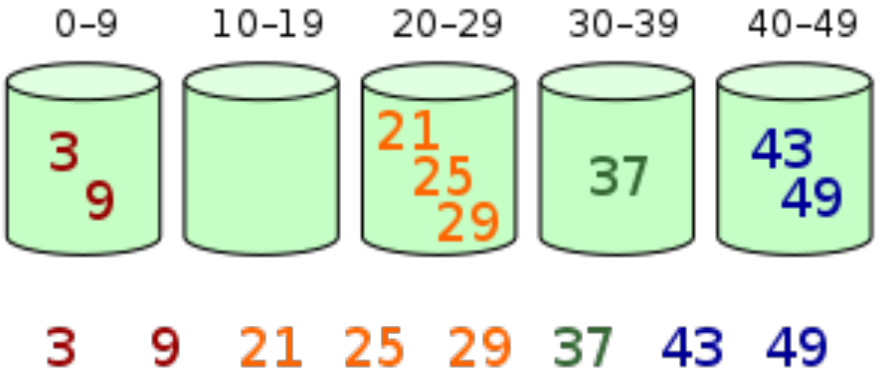
Science of Consciousness: The nature of life is to grow and progress; Natural Law unfolds in perfectly orderly sequence that gives rise to the universe, all of manifest creation.

Standard Bucket Sort (another version) [Optional]

- Three phases
  1. Distribution into buckets
  2. Sorting the buckets (the keys are not the same in the buckets)
  3. Combining the buckets



Elements are distributed among bins



Then, elements are sorted within each bin

1. Distribution

- Each key is examined once
  - a particular field of bits is examined or
  - some work is done to determine in which bucket it belongs
    - e.g., the key is compared to at most k preset values
- The item is then inserted into the proper bucket
- The work done in the distribution phase must be $\Theta(n)$

2. Sorting the Buckets

- Most of the work is done here
- O(m log m) operations are done for each bucket
  - m is the bucket size

3. Combining the Buckets

- The sorted sequences are concatenated
- Takes $\Theta(n)$ time

Analysis of Standard Bucket Sort

- If the keys are evenly distributed among the buckets and there are *k* buckets
  - Then the size of the buckets m = n/k
  - Thus the work (key comparisons) done would be
  - c m log m for each of the k buckets (c – constant time for comparisons)
    - We know that all comparison-based sorting algorithm needs atleast n log n
  - That is, the total work would be
  - k c (n/k) log (n/k) = c n log (n/k) [substitute m as n/k)

- If the number of buckets k = n/20, then the size of each bucket (n/k) is equal to 20, so the number of key comparisons would be
- c n log 20
- Thus, bucket sort would be linear when the input comes from a uniform distribution
- Note also that the larger the bucket size, the larger the constant (log m)

Main Point

3. We can prove that the lower bound on sorting by key comparisons in the best and worst cases is Ω(n log n). However, we can do better, i.e. linear time, but only if we have knowledge of the structure and distribution of keys. This knowledge is the basis for organizing a more efficient algorithm for sorting, but can only be applied in rare circumstances.

Science of Consciousness: Knowledge has organizing power. Deeper levels are governed by more comprehensive and unified laws of nature and have greater organizing power; pure knowledge has infinite organizing power.

Summary of Sorting Algorithms

Algorithm	Time	Notes (pros & cons)
insertion-sort		
Shell-sort		
heap-sort		
PQ-sort		
standard bucket-sort		

Summary of Sorting Algorithms

Algorithm	Time	Notes (pros & cons)
insertion-sort	$O(n^2)$	excellent for small inputs, fast for 'almost' sorted, simple, in-place
Shell-sort	$O(n^{3/2})$	fast even for fairly large inputs, simple, in-place
heap-sort	$O(n \log n)$	fast, as few comparisons as any sort, in-place
PQ-sort	$O(n \log n)$	not in-place, fast, excellent if input needs to be unchanged
Generic bucket-sort	$O(n \log(n/k))$	if keys can be distributed evenly into relatively small bucket sizes

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Comparison-based sorting algorithms can achieve a worst-case running time of $\Theta(n \log n)$, but can do no better.

2. Under certain conditions on the input, Bucket Sort and Radix Sort can sort in O(n) steps, even in the worst case. The nlog n lower bound does not apply because these algorithms are not comparison-based.

3. **Transcendental Consciousness** is the field of all possibilities and of pure orderliness. Contact with this field brings to light new possibilities and leads to spontaneous orderliness in all aspects of life.
4. **Impulses within Transcendental Consciousness**: The organizing power of pure knowledge is the lively expression of the Transcendent, giving rise to all expressions of intelligence.
5. **Wholeness moving within itself**: In Unity Consciousness, the organizing dynamics at the source of creation are appreciated as an expression of one's own Self.