Question

- Merge sort and Quick sort
  - worst-case running time is $O(n \lg n)$ and $O(n^2)$, respectively
- Are there better algorithms?
  - Can we do better (linear time algorithm) if the input has special structure (e.g., uniformly distributed, every number can be represented by d digits?)

CPT108 Data Structures and Algorithms

Lecture 13

Sorting

Counting Sort and Radix Sort

Counting sort and Radix sort

- Non-comparison-based sorting algorithms
- Work well when there is *limited range* of “*input values*”

Counting sort

- A sorting algorithm that sorts the elements by counting the number of occurrences of *unique* element, known as *keys*, in the array.

Counting sort

Algorithm

```
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 = 0 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

Example

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 = 0 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
```

	1	2	3	4	5	6	7	8	9	10
A	3	2	0	3	6	2	0	6	2	4

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

Counting sort

Example

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 = 0 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
```

	1	2	3	4	5	6	7	8	9	10
A	3	2	0	3	6	2	0	6	2	4

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

	0	1	2	3	4	5	6
(updated) C	2	2	5	7	8	8	10

Counting sort

Example (cont.)

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 = 0 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
```

current node $j = 10$

	1	2	3	4	5	6	7	8	9	10
A	3	2	0	3	6	2	0	6	2	4

$A[10] = 4$

	0	1	2	3	4	5	6
C	2	2	5	7	7	8	10

$C[4] = 8$

decrement counter value by 1 afterwards

	1	2	3	4	5	6	7	8	9	10
B								4		

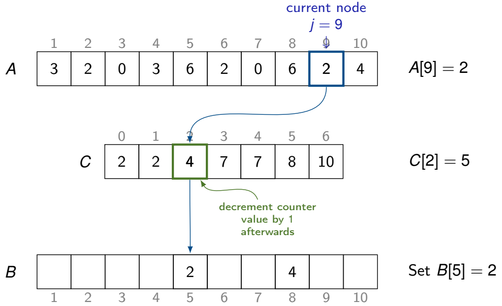
Set $B[8] = 4$

Counting sort

Example (cont.)

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 = 0 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
```

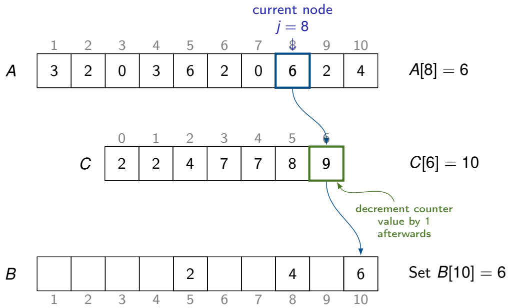


Counting sort

Example (cont.)

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 = 0 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
```

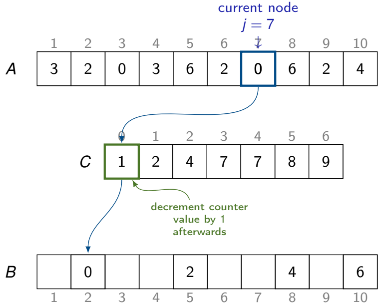


Counting sort

Example (cont.)

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 = 0 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
```



$A[7] = 0$

$C[0] = 2$

Set $B[2] = 0$

Counting sort

Example (cont.)

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 = 0 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
```

current node
$j = 6$

	1	2	3	4	5	6	7	8	9	10
A	3	2	0	3	6	2	0	6	2	4

$A[6] = 2$

	0	1	2	3	4	5	6
C	1	2	3	7	7	8	9

$C[2] = 4$

decrement counter value by 1 afterwards

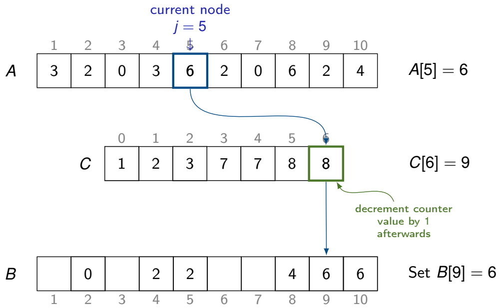
	1	2	3	4	5	6	7	8	9	10
B		0		2	2			4		6

Set $B[4] = 2$

Counting sort

Example (cont.)

```
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 = 0 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

Example (cont.)

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 = 0 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
```

current node
$j = 4$

	1	2	3	4	5	6	7	8	9	10
A	3	2	0	3	6	2	0	6	2	4

$A[4] = 3$

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

$C[3] = 7$

decrement counter value by 1 afterwards

	1	2	3	4	5	6	7	8	9	10
B		0		2	2		3	4	6	6

Set $B[7] = 3$

Counting sort

Example (cont.)

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 = 0 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
```

current node
$j = 3$

	1	2	3	4	5	6	7	8	9	10
A	3	2	0	3	6	2	0	6	2	4

$A[3] = 0$

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

$C[0] = 1$

decrement counter value by 1 afterwards

	1	2	3	4	5	6	7	8	9	10
B	0	0		2	2		3	4	6	6

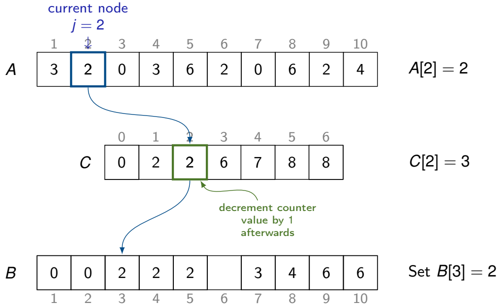
Set $B[1] = 0$

Counting sort

Example (cont.)

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 = 0 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

Example (cont.)

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 = 0 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
```

current node

$j = 1$

	1	2	3	4	5	6	7	8	9	10
A	3	2	0	3	6	2	0	6	2	4

$A[1] = 3$

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

$C[3] = 6$

decrement counter value by 1 afterwards

	1	2	3	4	5	6	7	8	9	10
B	0	0	2	2	2	3	3	4	6	6

Set $B[6] = 3$

Counting sort

Example (cont.)

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 = 0 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
```

	1	2	3	4	5	6	7	8	9	10
A	3	2	0	3	6	2	0	6	2	4

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

	1	2	3	4	5	6	7	8	9	10
B	0	0	2	2	2	3	3	4	6	6

Sorted Array

1	2	3	4	5	6	7	8	9	10
0	0	2	2	2	3	3	4	6	6

Counting sort

Some observations

- Assumes that each element of the n input is, or can be *mapped* to, an *non-negative* integer, i.e., the *key*, in the range from 0 to k
- It uses the input elements as the *indices* of the auxiliary array — *not* for the use of comparisons

Counting sort

Complexity

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 = 0 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
```

array initiation $\Theta(k)$

pass over the array to inspect each input element, $C[i]$ holds each element each to i $\Theta(n)$

determine the number of elements that are less than or equal to i $\Theta(k)$

place each element into its correct position in the output array $\Theta(n)$

$\Rightarrow$ *the overall time complexity is* $\Theta(n+k)$

Counting sort

Question

For sorting an array of the 100 largest cities by population, which sort do you think has a better worst-case execution time?

- A. Quick sort
- B. Insertion sort
- C. Counting sort

City	Population
Ahmedabad	7 681 000
Alexandria	5 086 000
Atlanta	5 572 000
Baghdad	6 812 000
Bangalore	11 440 000
⋮	⋮

Counting sort

Question (cont.)

In counting sort, it requires building of an array that is equal to the maximum value of the key, ie:

City	Population
Ahmedabad	7 681 000
Alexandria	5 086 000
Atlanta	5 572 000
Baghdad	6 812 000
Bangalore	11 440 000
⋮	⋮

Populations

Population	Count
⋮	0
5 022 998	0
5 022 999	0
5 023 000	1
5 023 001	0
⋮	0
5 051 999	0
5 052 000	1
5 052 001	0
⋮	0

Counting array

Resulted in a large number of zero-valued entries!

. . .

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

Counting sort

Summary (Geeksforgeeks.org, 2024a)

Advantages

- It has running time complexity of $O(n + k)$
- Generally performs faster than all comparison-based sorting algorithms, such as merge sort and quick sort, when the *range* of input values is small compare to the number of elements to be sorted
- Easy to code
- It is stable meaning that it preserve the relative order of elements with equal value

Disadvantages

- It does *not* work on decimal values
- Inefficient if the range of input values is very large
- It is *not* an in-place sorting algorithm, and requires an auxiliary space of $O(n + k)$ (i.e., k for counting, and n for outputs) for sorting the array elements

Radix sort

- Commonly use as a way to sort punch cards as early as 1923.

image source: `https://twobithistory.org/2018/06/23/ibm-029-card-punch.html`

- Exploits the concepts of *place value* by sorting numbers *digit by digit*
- Assumes that the “data” must be between a *range* of elements
- A procedure that uses a another *stable* sort (e.g., merge sort, quick sort, counting sort) as a subroutine
- Can be implemented to start with
  - Least significant digit (LSD)
  - Most significant digit (MSD)

Radix sort

Pseudocode

Find the maximum element *max* in the array
Find the number of digits *k* in *max*
For each *i* from 1 to *k*
    Sort the i^{th} least-significant digit of each element using a stable sort algorithm
    (If any element has less than *i* digits consider 0 at its place)

Algorithm

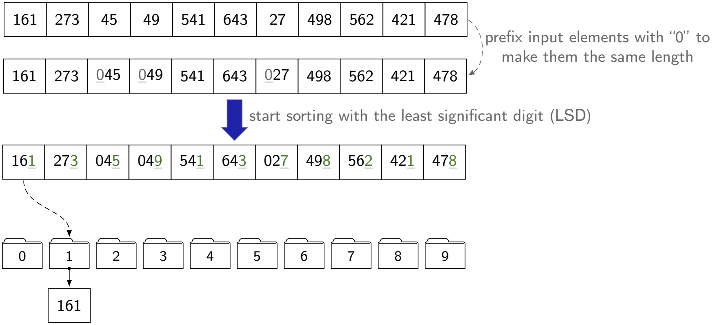
RADIX-SORT(A, n)

```
    // d: maximum number of digits (or length) of elements in the array
for i = 1 to d
    use a stable sort to sort array A[1 : n] on digit i
```

- Commonly used ***stable*** sorting algorithms include: *insertion sort*, *merge sort*, and *counting sort*

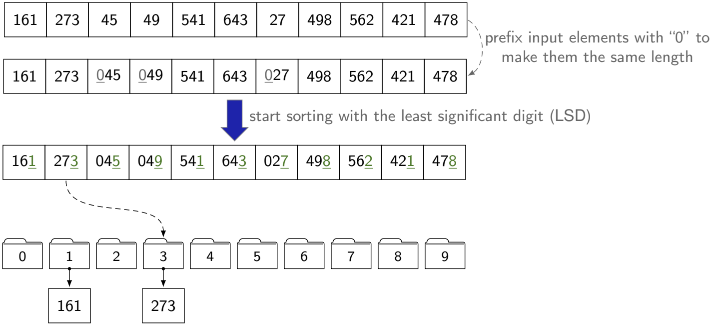
Radix sort

Example



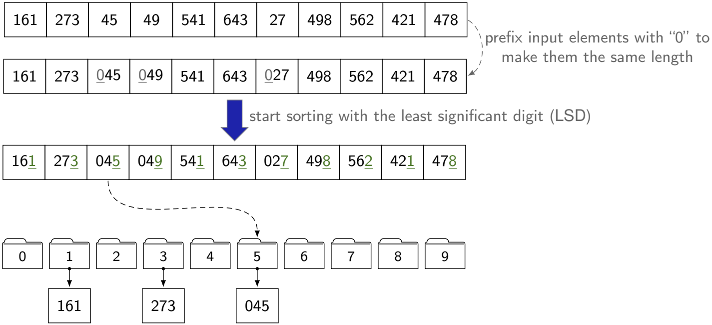
Radix sort

Example



Radix sort

Example



Radix sort

Example

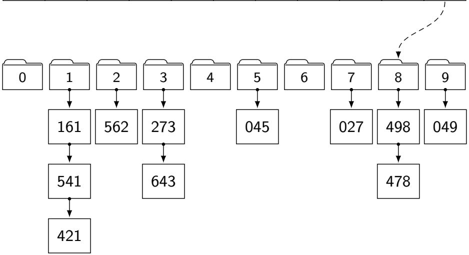
161	273	45	49	541	643	27	498	562	421	478

prefix input elements with "0" to make them the same length

161	273	045	049	541	643	027	498	562	421	478

start sorting with the least significant digit (LSD)

161	273	045	049	541	643	027	498	562	421	478

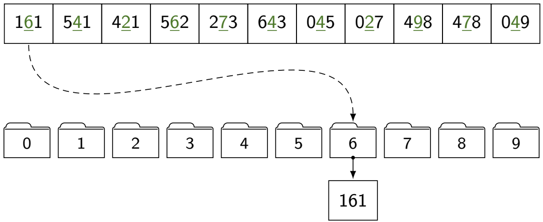


End of 1st iteration:

161	541	421	562	273	643	045	027	498	478	049

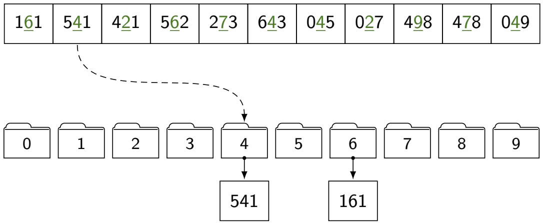
Radix sort

Example (cont.)



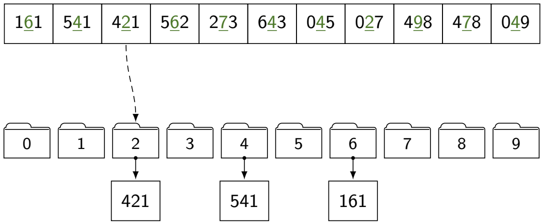
Radix sort

Example (cont.)



Radix sort

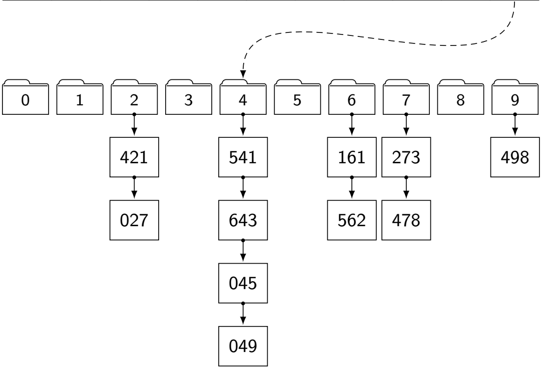
Example (cont.)



Radix sort

Example (cont.)

161	541	421	562	273	643	045	027	498	478	049

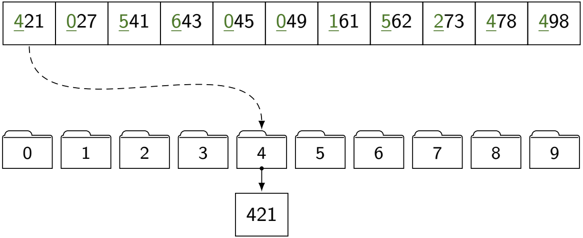


End of 2nd iteration:

421	027	541	643	045	049	161	562	273	478	498

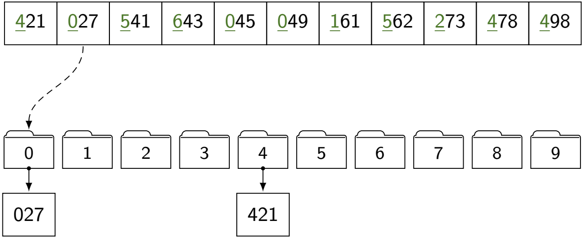
Radix sort

Example (cont.)



Radix sort

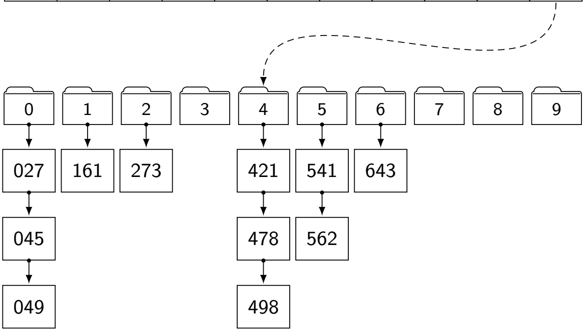
Example (cont.)



Radix sort

Example (cont.)

421	027	541	643	045	049	161	562	273	478	498



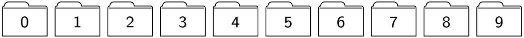
Sorted Array

027	045	049	161	273	421	478	498	541	562	643

Radix sort

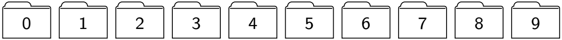
Exercise

7	16	58	129	347	153	48



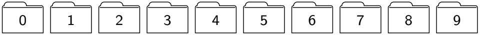
Radix sort

Exercise (cont.)



Radix sort

Exercise (cont.)



Radix sort

Question

How to sort string with radix sort?

- We can *map* each character into an integer and use the technique described above to do the sorting.

$$
\begin{aligned}
\texttt{<SPACE>} &\to 0 \\
\text{A} &\to 1 \\
\text{B} &\to 2 \\
\text{C} &\to 3 \\
&\vdots
\end{aligned}
$$

- *Can we do the same for counting sort?*

Radix sort

Complexity

RADIX-SORT(A, n)

// d: maximum number of digits (or length) of elements in the array

for $i = 1$ to d

use a stable sort to sort array $A[1:n]$ on digit i $\Theta(n+k)$ (if counting sort is used)

repeat d times

$\Rightarrow$ *the overall time complexity is* $\Theta(d(n+k))$

Radix sort

Summary (Geeksforgeeks.org, 2024b)

Advantages

- In practice, radix sort is often faster than other comparison-based sorting algorithms, such as quick sort and merge sort, for large datasets, especially when the keys have many digits

Drawbacks

- Its time complexity grows *linearly* with the number of digits, and so it is not as efficient for small datasets
- It requires *fixed size keys* to operate, which may not be the case for some types of data
- It is *not* an in-place algorithm as it uses a temporary count array
- It requires an auxiliary space of $O(n + k)$ for creating the buckets for each digit value and to copy the elements back to the original array after each digit has been sorted.

Counting sort and Radix sort

	Worst	Best
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$
Bubble sort	$O(n^2)$	$O(n^2)$
Improved bubble sort	$O(n^2)$	$O(n)$
Merge sort	$O(n \lg n)$	$O(n \lg n)$
Quick sort	$O(n^2)$	$O(n \lg n)$
Heap sort	$O(n \lg n)$	$O(n \lg n)$
Counting sort	$O(n + k)$	$O(n + k)$
Radix sort	$O(d(n + k))$	$O(d(n + k))$

Reading

- Chapter 8.2-8.3, Cormen (2022)
- “Lower bounds for sorting” Section 8.1, Cormen (2022) (Optional)
- “Bucket sort” Section 8.4, Cormen (2022) (Optional)

References I

- Geeksforgeeks.org (2024a). *Counting Sort — Data Structure and Algorithm Tutorials*. Online: https://www.geeksforgeeks.org/counting-sort/. [last accessed: 20 Mar 2024].
- — (2024b). *Radix Sort — Data Structure and Algorithm Tutorials*. Online: https://www.geeksforgeeks.org/radix-sort/. [last accessed: 20 Mar 2024].