Countingsort and Radixsort

COMS10017 - (Object-Oriented Programming and) Algorithms

Dr Christian Konrad

Countingsort

Countingsort
Input: Integer array $A \in \{0, 1, 2, \ldots, k\}^n$, for some integer k

Idea

Countingsort: Sorting Integers fast

Countingsort

Input: Integer array $A \in \{0, 1, 2, \ldots, k\}^n$, for some integer k

Idea

- For each element $x \in \{0, 1, \ldots, k\}$, count $\#$ elements $\leq x$

Countingsort: Sorting Integers fast

Countingsort

Input: Integer array $A \in \{0, 1, 2, \ldots, k\}^n$, for some integer k

Idea

- For each element $x \in \{0, 1, \ldots, k\}$, count $\#$ elements $\leq x$
- Put elements $A[i]$ directly into correct position

Countingsort: Sorting Integers fast

Countingsort
Input: Integer array $A \in \{0, 1, 2, \ldots, k\}^n$, for some integer k

Idea

- For each element $x \in \{0, 1, \ldots, k\}$, count $\#$ elements $\leq x$
- Put elements $A[i]$ directly into correct position
- **Difficulty:** Multiple elements have the same value

Algorithm

Require: Array A of n integers from $\{0, 1, 2, \ldots, k\}$, for some integer k

Let $C[0 \ldots k]$ be a new array with all entries equal to 0
Store output in array $B[0 \ldots n-1]$

for $i = 0, \ldots, n-1$ **do** {Count how often each element appears}
  $C[A[i]] \leftarrow C[A[i]] + 1$
for $i = 1, \ldots, k$ **do** {Count how many smaller (or equal) elements appear}
  $C[i] \leftarrow C[i] + C[i-1]$
for $i = n-1, \ldots, 0$ **do**
  $B[C[A[i]] - 1] \leftarrow A[i]$
  $C[A[i]] \leftarrow C[A[i]] - 1$
return B

Algorithm

Require: Array A of n integers from $\{0, 1, 2, \ldots, k\}$, for some integer k
Let $C[0 \ldots k]$ be a new array with all entries equal to 0
Store output in array $B[0 \ldots n-1]$

for $i = 0, \ldots, n-1$ **do** {Count how often each element appears}
  $C[A[i]] \leftarrow C[A[i]] + 1$
for $i = 1, \ldots, k$ **do** {Count how many smaller (or equal) elements appear}
  $C[i] \leftarrow C[i] + C[i-1]$
for $i = n-1, \ldots, 0$ **do**
  $B[C[A[i]] - 1] \leftarrow A[i]$
  $C[A[i]] \leftarrow C[A[i]] - 1$
return B

- Last loop processes A from right to left

Algorithm

Require: Array A of n integers from $\{0, 1, 2, \ldots, k\}$, for some integer k
  Let $C[0 \ldots k]$ be a new array with all entries equal to 0
  Store output in array $B[0 \ldots n-1]$

  for $i = 0, \ldots, n-1$ **do** {Count how often each element appears}
    $C[A[i]] \leftarrow C[A[i]] + 1$
  for $i = 1, \ldots, k$ **do** {Count how many smaller (or equal) elements appear}
    $C[i] \leftarrow C[i] + C[i-1]$
  for $i = n-1, \ldots, 0$ **do**
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$
  return B

- Last loop processes A from right to left
- $C[A[i]]$: Number of elements *smaller or equal* to $A[i]$

Algorithm

Require: Array A of n integers from $\{0, 1, 2, \ldots, k\}$, for some integer k
Let $C[0 \ldots k]$ be a new array with all entries equal to 0
Store output in array $B[0 \ldots n-1]$

for $i = 0, \ldots, n-1$ **do** {Count how often each element appears}
    $C[A[i]] \leftarrow C[A[i]] + 1$
for $i = 1, \ldots, k$ **do** {Count how many smaller (or equal) elements appear}
    $C[i] \leftarrow C[i] + C[i-1]$
for $i = n-1, \ldots, 0$ **do**
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$
return B

- Last loop processes A from right to left
- $C[A[i]]$: Number of elements *smaller or equal* to $A[i]$
- Decrementing $C[A[i]]$: Next element of value $A[i]$ should be left of the current one

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

	0	1	2	3	4	5	6	7
B								

for $i = n-1, \dots, 0$ **do**
  $B[C[A[i]] - 1] \leftarrow A[i]$
  $C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

	0	1	2	3	4	5	6	7
B							3	

for $i = n-1, \ldots, 0$ **do**
$\quad B[C[A[i]] - 1] \leftarrow A[i]$
$\quad C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

	0	1	2	3	4	5	6	7
B							3	

for $i = n-1, \dots, 0$ **do**
  $B[C[A[i]] - 1] \leftarrow A[i]$
  $C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8,\ k = 5$

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

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

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

	0	1	2	3	4	5	6	7
B		0					3	

for $i = n-1, \dots, 0$ **do**
$B[C[A[i]] - 1] \leftarrow A[i]$
$C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

	0	1	2	3	4	5	6	7
B		0					3	

for $i = n - 1, \dots, 0$ **do**
  $B[C[A[i]] - 1] \leftarrow A[i]$
  $C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8,\ k = 5$

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

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

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

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

for $i = n-1, \dots, 0$ **do**
$B[C[A[i]] - 1] \leftarrow A[i]$
$C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

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

for $i = n-1, \dots, 0$ **do**
$B[C[A[i]] - 1] \leftarrow A[i]$
$C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

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

for $i = n-1, \dots, 0$ **do**
  $B[C[A[i]] - 1] \leftarrow A[i]$
  $C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

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

for $i = n-1, \ldots, 0$ **do**
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

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

for $i = n-1, \dots, 0$ **do**
$B[C[A[i]] - 1] \leftarrow A[i]$
$C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8,\ k = 5$

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

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

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

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

for $i = n-1, \dots, 0$ **do**
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort: Example

Example: $n = 8$, $k = 5$

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

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

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

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

for $i = n-1, \dots, 0$ **do**
  $B[C[A[i]] - 1] \leftarrow A[i]$
  $C[A[i]] \leftarrow C[A[i]] - 1$

Analysis: Countingsort

Runtime:

$$
\begin{array}{l}
\textbf{for } i = 0, \ldots, n-1 \textbf{ do} \\
\quad C[A[i]] \leftarrow C[A[i]] + 1 \\
\textbf{for } i = 1, \ldots, k \textbf{ do} \\
\quad C[i] \leftarrow C[i] + C[i-1] \\
\textbf{for } i = n-1, \ldots, 0 \textbf{ do} \\
\quad B[C[A[i]] - 1] \leftarrow A[i] \\
\quad C[A[i]] \leftarrow C[A[i]] - 1
\end{array}
$$

Analysis: Countingsort

Runtime:

$O(n) + O(k) + O(n) = O(n + k)$

for $i = 0, \ldots, n-1$ **do**
    $C[A[i]] \leftarrow C[A[i]] + 1$
for $i = 1, \ldots, k$ **do**
    $C[i] \leftarrow C[i] + C[i-1]$
for $i = n-1, \ldots, 0$ **do**
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$

Analysis: Countingsort

Runtime:

$$O(n) + O(k) + O(n) = O(n + k)$$

- Countingsort has runtime $O(n)$ if $k = O(n)$

```
for i = 0, ..., n − 1 do
    C[A[i]] ← C[A[i]] + 1
for i = 1, ..., k do
    C[i] ← C[i] + C[i − 1]
for i = n − 1, ..., 0 do
    B[C[A[i]] − 1] ← A[i]
    C[A[i]] ← C[A[i]] − 1
```

Analysis: Countingsort

Runtime:

$$O(n) + O(k) + O(n) = O(n + k)$$

- Countingsort has runtime $O(n)$ if $k = O(n)$
- This beats the lower bound for comparison-based sorting

```
for i = 0, ..., n − 1 do
    C[A[i]] ← C[A[i]] + 1
for i = 1, ..., k do
    C[i] ← C[i] + C[i − 1]
for i = n − 1, ..., 0 do
    B[C[A[i]] − 1] ← A[i]
    C[A[i]] ← C[A[i]] − 1
```

Analysis: Countingsort

Runtime:

$O(n) + O(k) + O(n) = O(n + k)$

- Countingsort has runtime $O(n)$ if $k = O(n)$
- This beats the lower bound for comparison-based sorting

for $i = 0, \ldots, n-1$ **do**
  $C[A[i]] \leftarrow C[A[i]] + 1$
for $i = 1, \ldots, k$ **do**
  $C[i] \leftarrow C[i] + C[i-1]$
for $i = n-1, \ldots, 0$ **do**
  $B[C[A[i]] - 1] \leftarrow A[i]$
  $C[A[i]] \leftarrow C[A[i]] - 1$

Stable? In-place?

Analysis: Countingsort

Runtime:

$$O(n) + O(k) + O(n) = O(n + k)$$

- Countingsort has runtime $O(n)$ if $k = O(n)$
- This beats the lower bound for comparison-based sorting

```
for i = 0, ..., n − 1 do
    C[A[i]] ← C[A[i]] + 1
for i = 1, ..., k do
    C[i] ← C[i] + C[i − 1]
for i = n − 1, ..., 0 do
    B[C[A[i]] − 1] ← A[i]
    C[A[i]] ← C[A[i]] − 1
```

Stable? In-place? Yes, it is stable (important!)

Analysis: Countingsort

Runtime:

$O(n) + O(k) + O(n) = O(n + k)$

- Countingsort has runtime $O(n)$ if $k = O(n)$
- This beats the lower bound for comparison-based sorting

for $i = 0, \dots, n-1$ **do**
    $C[A[i]] \leftarrow C[A[i]] + 1$
for $i = 1, \dots, k$ **do**
    $C[i] \leftarrow C[i] + C[i-1]$
for $i = n-1, \dots, 0$ **do**
    $B[C[A[i]] - 1] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$

Stable? In-place? Yes, it is stable (important!) No, not in-place

Analysis: Countingsort

Runtime:

$O(n) + O(k) + O(n) = O(n + k)$

- Countingsort has runtime $O(n)$ if $k = O(n)$
- This beats the lower bound for comparison-based sorting

```
for i = 0, ..., n − 1 do
    C[A[i]] ← C[A[i]] + 1
for i = 1, ..., k do
    C[i] ← C[i] + C[i − 1]
for i = n − 1, ..., 0 do
    B[C[A[i]] − 1] ← A[i]
    C[A[i]] ← C[A[i]] − 1
```

Stable? In-place? Yes, it is stable (important!) No, not in-place

Correctness Loop Invariant

Radixsort

Radixsort
Input is an array A of d digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$

Radixsort
Input is an array A of d digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$

Examples

Radixsort

Radixsort

Input is an array A of d digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$

Examples

- $b = 2$, $d = 5$. E.g. 01101 (binary numbers)

Radixsort

Radixsort
Input is an array A of d digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$

Examples

- $b = 2$, $d = 5$. E.g. 01101 (binary numbers)
- $b = 10$, $d = 4$. E.g. 9714

Radixsort

Radixsort

Input is an array A of d digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$

Examples

- $b = 2$, $d = 5$. E.g. 01101 (binary numbers)
- $b = 10$, $d = 4$. E.g. 9714

Idea

Radixsort

Radixsort

Input is an array A of d digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$

Examples

- $b = 2$, $d = 5$. E.g. 01101 (binary numbers)
- $b = 10$, $d = 4$. E.g. 9714

Idea

- Iterate through the d digits

Radixsort

Input is an array A of d digits integers, each digit is from the set $\{0, 1, \ldots, b-1\}$

Examples

- $b = 2$, $d = 5$. E.g. 01101 (binary numbers)
- $b = 10$, $d = 4$. E.g. 9714

Idea

- Iterate through the d digits
- Sort according to the current digit

Radixsort (2)

Radixsort Algorithm

Radixsort (2)

Radixsort Algorithm

> **for** $i = 1, \ldots, d$ **do**
>   Use a stable sort algorithm to sort array A on digit i

(least significant digit is digit 1)

Radixsort (2)

Radixsort Algorithm

> **for** $i = 1, \ldots, d$ **do**
> Use a stable sort algorithm to sort array A on digit i

(least significant digit is digit 1)

Example

Radixsort (2)

Radixsort Algorithm

> **for** $i = 1, \ldots, d$ **do**
> Use a stable sort algorithm to sort array A on digit i

(least significant digit is digit 1)

Example

329
457
657
839
436
720
355

Radixsort (2)

Radixsort Algorithm

> **for** $i = 1, \ldots, d$ **do**
> Use a stable sort algorithm to sort array A on digit i

(least significant digit is digit 1)

Example

329		720
457		355
657		436
839	$\rightarrow$	457
436		657
720		329
355		839

Radixsort (2)

Radixsort Algorithm

> **for** $i = 1, \dots, d$ **do**
> Use a stable sort algorithm to sort array A on digit i

(least significant digit is digit 1)

Example

$$
\begin{array}{ccccc}
329 & & 720 & & 720 \\
457 & & 355 & & 329 \\
657 & & 436 & & 436 \\
839 & \rightarrow & 457 & \rightarrow & 839 \\
436 & & 657 & & 355 \\
720 & & 329 & & 457 \\
355 & & 839 & & 657
\end{array}
$$

Radixsort (2)

Radixsort Algorithm

> **for** $i = 1, \dots, d$ **do**
> Use a stable sort algorithm to sort array A on digit i

(least significant digit is digit 1)

Example

$$\begin{array}{ccccccc} 329 & & 720 & & 720 & & 329 \\ 457 & & 355 & & 329 & & 355 \\ 657 & & 436 & & 436 & & 436 \\ 839 & \rightarrow & 457 & \rightarrow & 839 & \rightarrow & 457 \\ 436 & & 657 & & 355 & & 657 \\ 720 & & 329 & & 457 & & 720 \\ 355 & & 839 & & 657 & & 839 \end{array}$$

Analysis

Radixsort (3)

Analysis

Lemma

Given n d-digit number in which each digit can take on up to b possible values. Radixsort correctly sorts these numbers in $O(d(n+b))$ time if the stable sort (e.g. Countingsort) it uses takes $O(n+b)$ time.

Radixsort (3)

Analysis

Lemma

Given n d-digit number in which each digit can take on up to b possible values. Radixsort correctly sorts these numbers in $O(d(n+b))$ *time if the stable sort (e.g. Countingsort) it uses takes* $O(n+b)$ *time.*

Proof

Radixsort (3)

Analysis

Lemma

Given n d-digit number in which each digit can take on up to b possible values. Radixsort correctly sorts these numbers in $O(d(n+b))$ *time if the stable sort (e.g. Countingsort) it uses takes* $O(n+b)$ *time.*

Proof Runtime is obvious. Correctness follows by induction on the columns being sorted. $\square$

Radixsort (3)

Analysis

Lemma

Given n d-digit number in which each digit can take on up to b possible values. Radixsort correctly sorts these numbers in $O(d(n+b))$ *time if the stable sort (e.g. Countingsort) it uses takes* $O(n+b)$ *time.*

Proof Runtime is obvious. Correctness follows by induction on the columns being sorted. □

Observe: If $d = O(1)$ and $b = O(n)$ then the runtime is $O(n)$!