COMP20007 Design of Algorithms

Input Enhancement Part 1: Distribution Sorting

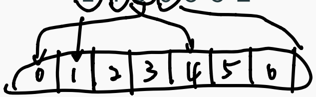
Daniel Beck

Lecture 17

Semester 1, 2020

Simple Distribution Sort

1 4 0 6 5 3 2



linear scan (allocate the value to the index position)

+ n size memory

Simple Distribution Sort

1 4 0 6 5 3 2

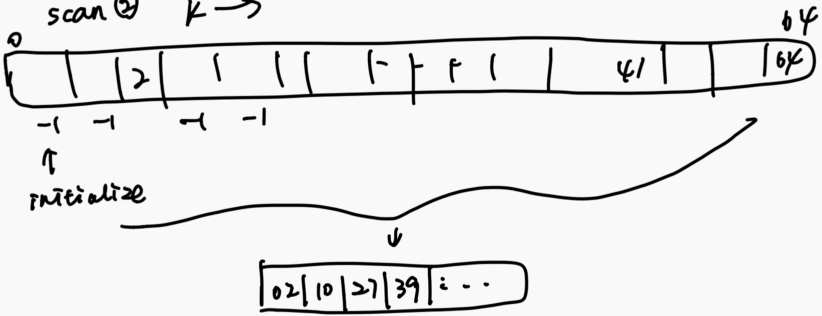
Looks $\Theta(n)$ even in worst case! Is it really?

Simple Distribution Sort

scan ① $n \rightarrow$

10 41 02 64 53 39 27

scan ② $k \rightarrow$



Simple Distribution Sort

10 41 02 64 53 39 27

$\Theta(n + k)$ worst case.

since $k >> n$ $\Theta(k)$

Simple Distribution Sort

10 41 02 10 41 10 10

multiple 10

instead of store value, store count.

Simple Distribution Sort

10 41 02 10 41 10 10

Use the auxiliary array to $\boxed{\text{store counts.}}$

Counting Sort

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

Counting Sort

$k=9+1=10$ 6 3 3 8 1 0 8 7 9 2 5 3 5 3 1 8 7 6 5 1 2 1 5 3

$\Theta(k)$

	0	1	2	3	4	5	6	7	8	9
Key	0	1	2	3	4	5	6	7	8	9
Counts	1	4	2	5	0	4	2	2	3	1

Counting Sort

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

Key	0	1	2	3	4	5	6	7	8	9	
Counts	1	4	2	5	0	4	2	2	3	1	
Counts($\boxed{shift}$)	0	1	4	2	5	0	4	2	2	3	1

Counting Sort

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

Key	0	1	2	3	4	5	6	7	8	9	
Counts	1	4	2	5	0	4	2	2	3	1	
Counts(shift)	0	1	4	2	5	0	4	2	2	3	1

Key	0	1	2	3	4	5	6	7	8	9	
Counts(acc)	0	1	5	7	12	12	16	18	20	23	24

0+1 0+1+4

0+1+4+2

Counting Sort

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

θ(n)

Key	0	1	2	3	4	5	6	7	8	9	
Counts	1	4	2	5	0	4	2	2	3	1	
Counts(shift)	0	1	4	2	5	0	4	2	2	3	1

θ(k)

if k << n, then θ(n)

θ(k)

Key	0	1	2	3	4	5	6	7	8	9	
Counts(acc)	0	1	5	7	12	12	16	18	20	23	24

tell you the starting position

θ(n) 0 1 1 1 1 2 2 3 3 3 3 3 5 5 5 5 6 6 7 7 8 8 8 9

0 1 2 3 4 5 6 7

Counting Sort

function $\textsc{Counting Sort}(A[0..n-1])$

  for $j \leftarrow 0$ **to** k **do**

    $C[j] \leftarrow 0$ initialise the count

  for $i \leftarrow 0$ **to** $n-1$ **do**

    $C[A[i]\underbrace{+1}_{\text{shift}}] \leftarrow C[A[i]+1]+1$ ▷ (shift)

  for $j \leftarrow 1$ **to** k **do**

    $C[j] = C[j] + C[j-1]$ cumulative counts.

  for $i \leftarrow 0$ **to** $n-1$ **do**

? → $B[C\boxed{[A[i]]}] \leftarrow A[i]$ key

    $C[A[i]] \leftarrow C[A[i]]+1$

  return ~~$B[1..n]$~~ B[0 ··· n-1]

✓ i=0

B[C[A[0]]] ← 6

6

16

C[6] ← 1

i=1

B[C[A[i]]] ← 3

Counting Sort

$$\frac{3}{\downarrow \; 7}$$

$C[3] \Leftarrow 8$

Questions!

Counting Sort

Questions!

- Stable?
- In-place?

Counting Sort

Counting Sort

- Stable: **Yes**, but depends on how it is implemented.

Counting Sort

- Stable: **Yes**, but depends on how it is implemented.
- In-place: **No**, requires $\Theta(n + k)$ memory.

Counting Sort

- Stable: **Yes**, but depends on how it is implemented.
- In-place: **No**, requires $\Theta(n + k)$ memory.

Take-home message: Counting Sort only works for integer keys and it works best when the key range is small.

Bucket Sort

Bucket Sort

- Generalisation of Counting Sort.

Bucket Sort

- Generalisation of Counting Sort.
- Split data into k buckets.

Bucket Sort

- Generalisation of Counting Sort.
- Split data into k buckets.
- Sort each bucket separately.

Bucket Sort

- Generalisation of Counting Sort.
- Split data into k buckets.
- Sort each bucket separately.
- Concatenate results.

Bucket Sort

- Generalisation of Counting Sort.
- Split data into k buckets.
- Sort each bucket separately.
- Concatenate results.

If K is the maximum key value and $k = K$, then Bucket Sort becomes Counting Sort.

if $k = K$, we don't need to sort each bucket separately

Bucket Sort

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

Bucket Sort

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

Bucket 1 ($A[i] < 3$): 1 0 2 1 1 2 1
Bucket 2 ($3 <= A[i] < 6$): 3 3 5 3 5 3 5 5 3
Bucket 3 ($A[i] >= 6$): 6 8 8 7 9 8 7 6

Bucket Sort

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

allocate a size

Bucket 1 ($A[i] < 3$): 1 0 2 1 1 2 1 → $\theta(n)$

Bucket 2 ($3 <= A[i] < 6$): 3 3 5 3 5 3 5 5 3 → $\theta(n)$

Bucket 3 ($A[i] >= 6$): 6 8 8 7 9 8 7 6 → $\theta(n)$

Sorted Bucket 1 ($A[i] < 3$): 0 1 1 1 1 2 2

Sorted Bucket 2 ($3 <= A[i] < 6$): 3 3 3 3 3 5 5 5 5

Sorted Bucket 3 ($A[i] >= 6$): 6 6 7 7 8 8 8 9

Bucket Sort

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

Bucket 1 ($A[i] < 3$): 1 0 2 1 1 2 1
Bucket 2 ($3 <= A[i] < 6$): 3 3 5 3 5 3 5 5 3
Bucket 3 ($A[i] >= 6$): 6 8 8 7 9 8 7 6

Sorted Bucket 1 ($A[i] < 3$): 0 1 1 1 1 2 2
Sorted Bucket 2 ($3 <= A[i] < 6$): 3 3 3 3 3 5 5 5 5
Sorted Bucket 3 ($A[i] >= 6$): 6 6 7 7 8 8 8 9

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

Bucket Sort

function BUCKET SORT($A[0..n-1], k$) ↑ number of bucket to set

$K \leftarrow$ max key value

for $j \leftarrow 0$ **to** $k-1$ **do**

INITIALISE($B[j]$) → initialise the bucket

for $i \leftarrow 0$ **to** $n-1$ **do** $\boxed{?}$

INSERT($B[\lfloor k \times A[i]/K \rfloor], A[i]$) → $\dfrac{A[i]}{\frac{K}{k}}$

for $j \leftarrow 0$ **to** $k-1$ **do**

AUXSORT($B[j]$)

return CONCATENATE($B[0..k-1]$)

Bucket Sort

function $\textsc{Bucket Sort}(A[0..n-1], k)$
  $K \leftarrow$ max key value
  for $j \leftarrow 0$ **to** $k-1$ **do**
    $\textsc{Initialise}(B[j])$
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $\textsc{Insert}(B[\lfloor k \times A[i]/K \rfloor], A[i])$
  for $j \leftarrow 0$ **to** $k-1$ **do**
    $\textsc{AuxSort}(B[j])$
  return $\textsc{Concatenate}(B[0..k-1])$

Stable?

Bucket Sort

Bucket Sort

Some properties depend on AuxSort:

Bucket Sort

Some properties depend on AUXSORT:

- Stability. if you use stable sorting algorithm
  (eg. insertion sort, mergesort)
  stable

  if unstable algorithm
  (eg. quicksort)

  unstable

Bucket Sort

Some properties depend on AuxSort:

- Stability.
- Worst case complexity. (assuming $k = \Theta(n)$)

eg. if use insertion sort $\Theta(n^2)$

then it become the worst case for bucket sort

if you don't set your bucket properly, all elements come to one bucket, then it depends on the algorithm you use

Bucket Sort

Some properties depend on AuxSort:

- Stability.
- Worst case complexity. (assuming $k = \Theta(n)$)

Average complexity: $\Theta(n + \frac{n^2}{k} + k)$. Linear if $k = \Theta(n)$.

$\underbrace{\Theta(n + \frac{n^2}{k}}_{\text{depend on bucket}}$

if $k = cn$

$\frac{n^2}{k} = \frac{n^{\cancel{2}}}{c\cancel{n}}$

$\Theta(n + \frac{n}{c} + cn)$

$\in \Theta(n)$

Bucket Sort

Some properties depend on $\textsc{AuxSort}$:

- Stability.
- Worst case complexity. (assuming $k = \Theta(n)$)

Average complexity: $\Theta(n + \frac{n^2}{k} + k)$. Linear if $k = \Theta(n)$.

Take-home message: Compared to Counting Sort, Bucket Sort provides more control over how much memory to use, but it is slower in the worst case.

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011

since < 8 , so 3 bit

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011

Bucket 1: 110 000 010 110 010

Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011

will be the least significant bit

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011

not sort for each bucket
directly concatenate

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011

look at 2nd digite

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011
000 001 101 101 001 101 001 001 101 110 010 110 010 011 011 111 011 011 111 011

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011
000 001 101 101 001 101 001 001 101 110 010 110 010 011 011 111 011 011 111 011
Bucket 1: 000 001 001 001 001 010 010 011 011 011 011 011
Bucket 2: 101 101 101 101 110 110 111 111

look at 1st digit

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011
000 001 101 101 001 101 001 001 101 110 010 110 010 011 011 111 011 011 111 011
Bucket 1: 000 001 001 001 001 010 010 011 011 011 011 011
Bucket 2: 101 101 101 101 110 110 111 111
000 001 001 001 001 010 010 011 011 011 011 011 101 101 101 101 110 110 111 111

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011
000 001 101 101 001 101 001 001 101 110 010 110 010 011 011 111 011 011 111 011
Bucket 1: 000 001 001 001 001 010 010 011 011 011 011 011
Bucket 2: 101 101 101 101 110 110 111 111
000 001 001 001 001 010 010 011 011 011 011 011 101 101 101 101 110 110 111 111

0 1 1 1 1 2 2 3 3 3 3 3 5 5 5 5 6 6 7 7

Radix Sort

Assumptions:

Radix Sort

Assumptions:

- Maximum key length is known in advance.

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Start sorting from least to the most significant digit. (also possible to do in reverse)

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Start sorting from least to the most significant digit. (also possible to do in reverse)
By using Bucket ~~Sort~~ with max buckets we have guaranteed $\Theta(n)$ performance per pass.

eg. binary → 2 bucket

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Start sorting from least to the most significant digit. (also possible to do in reverse)
By using Bucket Sort with max buckets we have guaranteed $\Theta(n)$ performance per pass.
Total worst case performance is $\Theta(n \times \boxed{len(k)})$?

Radix Sort

function $\textsc{Radix Sort}(A[0..n-1], k)$
 for $j \leftarrow 0$ **to** $len(k)$ **do**
  $A \leftarrow \textsc{AuxSort}(A, k[j])$

Radix Sort

function RADIX SORT($A[0..n-1], k$)
  for $j \leftarrow 0$ **to** $len(k)$ **do** from the least significant digit to the most significant digit
    $A \leftarrow$ AUXSORT($A, k[j]$)
    $\Theta(n\,len(k))$
    ↳ Bucket sort with max Bucket $\Theta(n)$

- Typically, AUXSORT is Bucket Sort with max buckets but can be any sorting algorithm as long as it is stable (why?)

Radix Sort

function $\textsc{Radix Sort}(A[0..n-1], k)$
  for $j \leftarrow 0$ **to** $len(k)$ **do**
    $A \leftarrow \textsc{AuxSort}(A, k[j])$

- Typically, $\textsc{AuxSort}$ is Bucket Sort with max buckets but can be any sorting algorithm as long as it is stable (why?)

Take-home message: Radix Sort can be very fast (faster than comparison sorting) if keys are short (need to known in advance).

Summary

- Distribution Sorting is a sorting paradigm that trades memory for speed.
- Relies on more assumptions, unlike Comparison Sorting algorithms:
  - Counting Sort, Bucket Sort: positive integer keys, with max bound known.
  - Radix Sort: more general but max key length must be known and keys should have lexicographical order.

In Practice

In Practice

- Distribution Sort is not as widely used as Comparison Sort:

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
  - Less general.
  - Require more memory.
  - In practice, good sorting algorithms can be very close to $\Theta(n)$ already (Timsort).

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
  - Less general.
  - Require more memory.
  - In practice, good sorting algorithms can be very close to $\Theta(n)$ already (Timsort).
- However, they can be very useful as part of a more complex algorithm.

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
  - Less general.
  - Require more memory.
  - In practice, good sorting algorithms can be very close to $\Theta(n)$ already (Timsort).
- However, they can be very useful as part of a more complex algorithm.

if k known

strings

?

$O(n)$

worst case

  - Radix Sort is used to construct suffix arrays.
  - Controlled environment with guaranteed short key size.

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
  - Less general.
  - Require more memory.
  - In practice, good sorting algorithms can be very close to $\Theta(n)$ already (Timsort).
- However, they can be very useful as part of a more complex algorithm.
  - Radix Sort is used to construct suffix arrays.
  - Controlled environment with guaranteed short key size.

Next lecture: string matching revisited.