Lecture 8
Counting sort, Bucket sort, Radix sort

CS 161 Design and Analysis of Algorithms
Ioannis Panageas

Counting sort

Counting sort

- Let A be the input array, B the output array. Assume there are n items.

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
  - To implement the algorithm in a modern programming language directly from the algorithm description:

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
  - To implement the algorithm in a modern programming language directly from the algorithm description:
    - Allocate each array to be one entry larger than it actually is

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
  - To implement the algorithm in a modern programming language directly from the algorithm description:
    - Allocate each array to be one entry larger than it actually is
    - Ignore location 0.

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
    - To implement the algorithm in a modern programming language directly from the algorithm description:
        - Allocate each array to be one entry larger than it actually is
        - Ignore location 0.
- Main idea of CountingSort: Suppose A contains exactly j elements $\leq x$

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
  - To implement the algorithm in a modern programming language directly from the algorithm description:
    - Allocate each array to be one entry larger than it actually is
    - Ignore location 0.
- Main idea of CountingSort: Suppose A contains exactly j elements $\leq x$
  - If x only appears once in A, then x should go in in $B[j]$.

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
  - To implement the algorithm in a modern programming language directly from the algorithm description:
    - Allocate each array to be one entry larger than it actually is
    - Ignore location 0.
- Main idea of CountingSort: Suppose A contains exactly j elements $\leq x$
  - If x only appears once in A, then x should go in in $B[j]$.
  - If x appears more than once in A and we want a stable sort:

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
  - To implement the algorithm in a modern programming language directly from the algorithm description:
    - Allocate each array to be one entry larger than it actually is
    - Ignore location 0.
- Main idea of CountingSort: Suppose A contains exactly j elements $\leq x$
  - If x only appears once in A, then x should go in in $B[j]$.
  - If x appears more than once in A and we want a stable sort:
    - Last occurrence of x in A should go in $B[j]$

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
  - To implement the algorithm in a modern programming language directly from the algorithm description:
    - Allocate each array to be one entry larger than it actually is
    - Ignore location 0.
- Main idea of CountingSort: Suppose A contains exactly j elements $\leq x$
  - If x only appears once in A, then x should go in in $B[j]$.
  - If x appears more than once in A and we want a stable sort:
    - Last occurrence of x in A should go in $B[j]$
    - Next-to-last occurrence of x should go in $B[j-1]$
    - etc.

Counting sort

Counting sort

- Assume:

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.
  - We make several passes over the data to set the values in the locator array before we do the actual sort.

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.
  - We make several passes over the data to set the values in the locator array before we do the actual sort.
  - At the start of the final (sorting) pass, `locator[x]` contains the number of elements $\leq$ `x`

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.
  - We make several passes over the data to set the values in the locator array before we do the actual sort.
  - At the start of the final (sorting) pass, `locator[x]` contains the number of elements $\leq$ `x`
- On the final pass:

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.
  - We make several passes over the data to set the values in the locator array before we do the actual sort.
  - At the start of the final (sorting) pass, `locator[x]` contains the number of elements $\leq$ `x`
- On the final pass:
  - Process the input array `A` from right to left (!). This makes the counting sort a stable sorting algorithm.

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.
  - We make several passes over the data to set the values in the locator array before we do the actual sort.
  - At the start of the final (sorting) pass, `locator[x]` contains the number of elements $\leq$ `x`
- On the final pass:
  - Process the input array `A` from right to left (!). This makes the counting sort a stable sorting algorithm.
  - When a value of `x` is encountered in the input array `A`:

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.
  - We make several passes over the data to set the values in the locator array before we do the actual sort.
  - At the start of the final (sorting) pass, `locator[x]` contains the number of elements $\leq$ `x`
- On the final pass:
  - Process the input array `A` from right to left (!). This makes the counting sort a stable sorting algorithm.
  - When a value of `x` is encountered in the input array `A`:
    - Copy the value into location `locator[x]` in the output array. That is, store it in location `B[locator[x]]`

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.
  - We make several passes over the data to set the values in the locator array before we do the actual sort.
  - At the start of the final (sorting) pass, `locator[x]` contains the number of elements $\leq$ `x`
- On the final pass:
  - Process the input array `A` from right to left (!). This makes the counting sort a stable sorting algorithm.
  - When a value of `x` is encountered in the input array `A`:
    - Copy the value into location `locator[x]` in the output array. That is, store it in location `B[locator[x]]`
    - Decrement `locator[x]`

Code for Counting sort

```
def CountingSort(A, B, n , k)
    //Initialize:  set each locator[x] to
          the number of entries ≤ x
    for x = 1 to k do locator[x] = 0
    for i = 1 to n do locator[A[i]] = locator[A[i]] + 1
    for x = 2 to k do
        locator[x] = locator[x] + locator[x-1]
    //Fill output array, updating locator values
    for i = n down to 1 do
        B[locator[A[i]]] = A[i]
        locator[A[i]] = locator[A[i]] - 1
```

Code for Counting sort

```
def CountingSort(A, B, n , k)
    //Initialize:  set each locator[x] to
          the number of entries ≤ x
    for x = 1 to k do locator[x] = 0
    for i = 1 to n do locator[A[i]] = locator[A[i]] + 1
    for x = 2 to k do
        locator[x] = locator[x] + locator[x-1]
    //Fill output array, updating locator values
    for i = n down to 1 do
        B[locator[A[i]]] = A[i]
        locator[A[i]] = locator[A[i]] - 1
```

Analysis: $O(n+k)$ running time.

Counting Sort Example

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

Counting Sort Example

A:

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

`locator`:

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

Counting Sort Example

A:

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

`locator`:

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

B:

1	2	3	4	5	6	7	8	9	10

Counting Sort Example

A:

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

`locator`:

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

B:

1	2	3	4	5	6	7	8	9	10

Counting Sort Example

A:

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

`locator`:

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

B:

1	2	3	4	5	6	7	8	9	10

Counting Sort Example

A:

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

`locator`:

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

B:

1	2	3	4	5	6	7	8	9	10
			4						

Counting Sort Example

A:

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

`locator`:

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

B:

1	2	3	4	5	6	7	8	9	10
			4						

Counting Sort Example

A:

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

`locator`:

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

B:

1	2	3	4	5	6	7	8	9	10
			4						

Counting Sort Example

A:

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

`locator`:

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

B:

1	2	3	4	5	6	7	8	9	10
			4						

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A:

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

`locator`:

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

B:

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

Counting Sort Example

A: Done!

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

`locator`:

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

B:

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

Bucket Sort

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.
- Three phases:

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.
- Three phases:
  1. Distribute keys into buckets

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.
- Three phases:
  1. Distribute keys into buckets
  2. Sort keys in each bucket

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.
- Three phases:
  1. Distribute keys into buckets
  2. Sort keys in each bucket
  3. Combine buckets.

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.
- Three phases:
  1. Distribute keys into buckets
  2. Sort keys in each bucket
  3. Combine buckets.
- Simplest approach is to divide the space of possible keys into equal sized buckets.

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.
- Three phases:
  1. Distribute keys into buckets
  2. Sort keys in each bucket
  3. Combine buckets.
- Simplest approach is to divide the space of possible keys into equal sized buckets.
- Typically use insertion sort in phase 2.

Bucket Sort Example

Bucket Sort Example

Sort the following keys in the range 0-999, using 10 equal-size buckets:

661 74 835 140 198 923 113 642 467 449

Bucket Sort Example

Sort the following keys in the range 0-999, using 10 equal-size buckets:

661 74 835 140 198 923 113 642 467 449

1. Distribute

0: 74
1: 140 198 113
2:
3:
4: 467 449
5:
6: 661 642
7:
8: 835
9: 923

Bucket Sort Example

Sort the following keys in the range 0-999, using 10 equal-size buckets:

661 74 835 140 198 923 113 642 467 449

1. Distribute

0:	74
1:	140 198 113
2:	
3:	
4:	467 449
5:	
6:	661 642
7:	
8:	835
9:	923

2. Sort

0:	74
1:	113 140 198
2:	
3:	
4:	449 467
5:	
6:	642 661
7:	
8:	835
9:	923

Bucket Sort Example

Sort the following keys in the range 0-999, using 10 equal-size buckets:

661 74 835 140 198 923 113 642 467 449

1. Distribute

0:	74
1:	140 198 113
2:	
3:	
4:	467 449
5:	
6:	661 642
7:	
8:	835
9:	923

2. Sort

0:	74
1:	113 140 198
2:	
3:	
4:	449 467
5:	
6:	642 661
7:	
8:	835
9:	923

3. Combine

74
113
140
198
449
467
642
661
835
923

Analysis of Bucket Sort

Analysis of Bucket Sort

n = number of items to sort

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets

Analysis of Bucket Sort

$$\begin{aligned} n &= \text{number of items to sort} \\ b &= \text{number of buckets} \\ s_i &= \text{number of items in bucket } i \; (i = 0, \ldots, b-1) \end{aligned}$$

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time
1. Distribution	$O(n)$

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_i s_i^2)$

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_i s_i^2)$
3. Combining buckets	$O(b)$

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_i s_i^2)$
3. Combining buckets	$O(b)$

Total running time is:

$$O\left(n + b + \sum_{i=1}^{b} s_i^2\right)$$

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_i s_i^2)$
3. Combining buckets	$O(b)$

Total running time is:

$$O\left(n + b + \sum_{i=1}^{b} s_i^2\right)$$

- Worst case: $O(n^2)$.

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_i s_i^2)$
3. Combining buckets	$O(b)$

Total running time is:

$$O\left(n + b + \sum_{i=1}^{b} s_i^2\right)$$

- Worst case: $O(n^2)$.
- Best case: $O(n)$.

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_i s_i^2)$
3. Combining buckets	$O(b)$

Total running time is:

$$O\left(n + b + \sum_{i=1}^{b} s_i^2\right)$$

- Worst case: $O(n^2)$.
- Best case: $O(n)$.
- Average case: $O(n)$ if certain assumptions are satisfied (next slide)

Analysis of Bucket Sort

$$
\begin{aligned}
n &= \text{number of items to sort} \\
b &= \text{number of buckets} \\
s_i &= \text{number of items in bucket } i \ (i = 0, \ldots, b-1)
\end{aligned}
$$

Phase	Running time
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_i s_i^2)$
3. Combining buckets	$O(b)$

Total running time is:

$$O\left(n + b + \sum_{i=1}^{b} s_i^2\right)$$

- Worst case: $O(n^2)$.
- Best case: $O(n)$.
- Average case: $O(n)$ if certain assumptions are satisfied (next slide)
- Storage: is $O(n + b)$.

Average running time of Bucket Sort

The following result is proved in [CLRS]:

Assume:

1. *The number of buckets is equal to the number of keys (i.e., if $b = n$)*
2. *The keys are distributed independently and uniformly over the buckets*

Then the expected total cost of the intra-bucket sorts is $O(n)$.

Radix Sort

Radix Sort

- Useful for sorting multi-field keys in lexicographic order

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on.

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:
    - clown comes before dog

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:
    - clown comes before dog
    - cat comes before clown

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
    - Words in dictionaries:
        - clown comes before dog
        - cat comes before clown
        - car comes before cat

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:
    - clown comes before dog
    - cat comes before clown
    - car comes before cat
  - Dates: `(year, month, day)`

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:
    - clown comes before dog
    - cat comes before clown
    - car comes before cat
  - Dates: `(year, month, day)`
  - Multi-digit numbers: (3-digit numbers in this example)

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:
    - clown comes before dog
    - cat comes before clown
    - car comes before cat
  - Dates: `(year, month, day)`
  - Multi-digit numbers: (3-digit numbers in this example)
    - 293 represented as (2,9,3)

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:
    - clown comes before dog
    - cat comes before clown
    - car comes before cat
  - Dates: `(year, month, day)`
  - Multi-digit numbers: (3-digit numbers in this example)
    - 293 represented as (2,9,3)
    - 71 represented as (0,7,1)

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:
    - clown comes before dog
    - cat comes before clown
    - car comes before cat
  - Dates: `(year, month, day)`
  - Multi-digit numbers: (3-digit numbers in this example)
    - 293 represented as (2,9,3)
    - 71 represented as (0,7,1)

Radix Sort:

Radix Sort:

Radix sort:

Radix Sort:

Radix sort:

- Sorts on each field in the key, one at a time

Radix Sort:

Radix sort:

- Sorts on each field in the key, one at a time
- Sorts on on least-significant field first

Radix Sort:

Radix sort:

- Sorts on each field in the key, one at a time
- Sorts on on least-significant field first
- Uses a stable sort

Radix Sort:

Radix sort:

- Sorts on each field in the key, one at a time
- Sorts on on least-significant field first
- Uses a stable sort
  - Recall: A sorting algorithm is stable if whenever two keys are equal, the algorithm preserves their order (i.e., does not reverse them.)

Radix Sort:

Radix sort:

- Sorts on each field in the key, one at a time
- Sorts on on least-significant field first
- Uses a stable sort
  - Recall: A sorting algorithm is stable if whenever two keys are equal, the algorithm preserves their order (i.e., does not reverse them.)

```
def radix_sort(A,n);
   for field ranging from rightmost (least significant)
         to leftmost (most significant):
      sort A on field using a stable sort
```

Radix Sort:

Radix sort:

- Sorts on each field in the key, one at a time
- Sorts on on least-significant field first
- Uses a stable sort
  - Recall: A sorting algorithm is stable if whenever two keys are equal, the algorithm preserves their order (i.e., does not reverse them.)

```
def radix_sort(A,n);
   for field ranging from rightmost (least significant)
         to leftmost (most significant):
      sort A on field using a stable sort
```

Radix Sort Example:

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)

661 74 835 140 198 923 113 642 467 449

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)

661 74 835 140 198 923 113 642 467 449

661
074
835
140
198
923
113
642
467
449

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)

661 74 835 140 198 923 113 642 467 449

661	⇒	140
074		661
835		642
140		923
198		113
923		074
113		835
642		467
467		198
449		449

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)

661 74 835 140 198 923 113 642 467 449

661		140		113
074		661		923
835		642		835
140		923		140
198	⇒	113	⇒	642
923		074		449
113		835		661
642		467		467
467		198		074
449		449		198

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)

661 74 835 140 198 923 113 642 467 449

661		140		113		074
074		661		923		113
835		642		835		140
140		923		140		198
198	⇒	113	⇒	642	⇒	449
923		074		449		467
113		835		661		642
642		467		467		661
467		198		074		835
449		449		198		923

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)

661 74 835 140 198 923 113 642 467 449

661		140		113		074
074		661		923		113
835		642		835		140
140		923		140		198
198	⇒	113	⇒	642	⇒	449
923		074		449		467
113		835		661		642
642		467		467		661
467		198		074		835
449		449		198		923

Note the importance of stability.

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)

661 74 835 140 198 923 113 642 467 449

661		140		113		074
074		661		923		113
835		642		835		140
140		923		140		198
198	⇒	113	⇒	642	⇒	449
923		074		449		467
113		835		661		642
642		467		467		661
467		198		074		835
449		449		198		923

Note the importance of stability.

We break the ties first, and stability makes sure the ties remain broken correctly.

Analysis of Radix Sort

Analysis of Radix Sort

Assume

Analysis of Radix Sort

Assume

- n is the number of items

Analysis of Radix Sort

Assume

- n is the number of items
- b is the size of each range

Analysis of Radix Sort

Assume

- n is the number of items
- b is the size of each range
  - Example:
    - Each field of each item is a numbers in the range `0..b-1`.

Analysis of Radix Sort

Assume

- n is the number of items
- b is the size of each range
  - Example:
    - Each field of each item is a numbers in the range `0..b-1`.
    - This is true if the numbers we are sorting are integers represented in base b.

Analysis of Radix Sort

Assume

- n is the number of items
- b is the size of each range
  - Example:
    - Each field of each item is a numbers in the range `0..b-1`.
    - This is true if the numbers we are sorting are integers represented in base b.
- d is the number of fields we are sorting

Analysis of Radix Sort

Assume

- n is the number of items
- b is the size of each range
  - Example:
    - Each field of each item is a numbers in the range `0..b-1`.
    - This is true if the numbers we are sorting are integers represented in base b.
- d is the number of fields we are sorting
  - For example, if each item is a base b number with d digits. (i.e., between 0 and $b^d - 1$, inclusive).

Analysis of Radix Sort

Assume

- n is the number of items
- b is the size of each range
  - Example:
    - Each field of each item is a numbers in the range `0..b-1`.
    - This is true if the numbers we are sorting are integers represented in base b.
- d is the number of fields we are sorting
  - For example, if each item is a base b number with d digits. (i.e., between 0 and $b^d - 1$, inclusive).
- Each field is sorted using Bucket Sort or Counting Sort

Analysis of Radix Sort

Assume

- n is the number of items
- b is the size of each range
  - Example:
    - Each field of each item is a numbers in the range `0..b-1`.
    - This is true if the numbers we are sorting are integers represented in base b.
- d is the number of fields we are sorting
  - For example, if each item is a base b number with d digits. (i.e., between 0 and $b^d - 1$, inclusive).
- Each field is sorted using Bucket Sort or Counting Sort

Then the running time of radix sort is $O(d(n + b))$.

Deterministic Selection:
Find k-th element

Recall QuickSelect

`quickSelect`(S, k)

If n is small, brute force and return.
Pick a random $x \in S$ and put rest into:
  L, elements smaller than x
  G, elements greater than x
if $k \leq |L|$ **then**
  quickSelect(L, k)
else if $k == |L| + 1$ **then**
  return x
else
  quickSelect$(G, k - (|L| + 1))$

Deterministic Selection

Instead of picking x at random:

- Divide S into $g = \lceil n/5 \rceil$ groups
- Each group has 5 elements (except maybe g^{th})
- Find median of each group of 5
- Find median of those medians
- Let x be that median.

We call this the "medians of 5" method.

Selecting Median of 5 Example

870	647	845	742	372	882	691	341	461	596
989	151	100	729	101	397	825	587	363	283
595	524	930	259	133	955	620	970	430	280
839	139	735	590	782	913	378	474	255	739
875	150	791	779	792					

Deterministic Select

`DeterministicSelect`(S, k)

If n is small, brute force and return.

Pick $x \in S$ via medians-of-5 and put rest into:

L, elements smaller than x

G, elements greater than x

if $k \leq |L|$ **then**

DeterministicSelect(L, k)

else if $k == |L| + 1$ **then**

return x

else

DeterministicSelect$(G, k - (|L| + 1))$

Deterministic Selection

Let's visualize: how does pivot compare to list?

Demo Re-visualized

- Each column was a group of five.
- Each column is sorted
- Columns are ordered based on median-of-5
- Which cells are in L? G? Either?

100	283	255	133	341				
101	363	378	259	461				
151	397	474	524	596	620	735	742	791
				691	955	782	845	792
				882	970	839	870	875

Deterministic Selection

- How few elements *must be* smaller than pivot?

- How few *must be* non-smaller than pivot?

- How many could be in either group?