Advanced Sorting Algorithms (Bucket, Count and Radix Sort)

Overview

- Introduction to Bucket sort
- Conceptual Understanding of Bucket sort
- Introduction to Radix sort
- Implementation details of Radix sort
- Applications of Radix sort

Bucket Sort

- The bucket sort makes assumptions about the data being sorted
  - Consequently, we can achieve better than $\Theta(n \ln(n))$ run times

Supporting Example

Suppose we are sorting a large number of mobile numbers, for example, all mobile numbers in Karachi (approximately four million)

We could use quick sort, however that would require an array which is kept entirely in memory

Supporting Example

Instead, consider the following scheme:

- Create a bit set (an array of bool) with $10\,000\,000$ bit

```
bitset<M> A ;
```

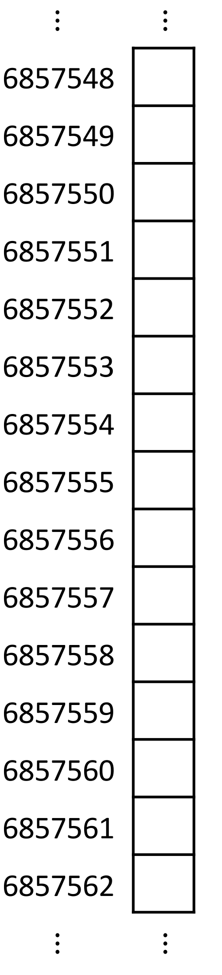
- Set each bit to 0 (indicating false)

  Default value of bitset is 0

- For each phone number, set the bit indexed by the phone number to 1 (true)
- Once each phone number has been checked, walk through the array and for each bit which is 1, record that number

Supporting Example

For example, consider this section within the bit array



Supporting Example

For each phone number, set the corresponding bit

– For example, 6857550 is a mobile number

⋮	⋮
6857548	✓
6857549	✓
6857550	
6857551	
6857552	
6857553	✓
6857554	
6857555	✓
6857556	
6857557	
6857558	✓
6857559	
6857560	
6857561	✓
6857562	✓
⋮	⋮

Supporting Example

For each phone number, set the corresponding bit

– For example, 6857550 is a mobile number

⋮	⋮
6857548	✓
6857549	✓
6857550	✓
6857551	
6857552	
6857553	✓
6857554	
6857555	✓
6857556	
6857557	
6857558	✓
6857559	
6857560	
6857561	✓
6857562	✓
⋮	⋮

Supporting Example

At the end, we just take all the numbers out that were checked:

…,6857548, 6857549, 6857550,
6857553, 6857555, 6857558,
6857561, 6855762, …

⋮	⋮
6857548	✓
6857549	✓
6857550	✓
6857551	
6857552	
6857553	✓
6857554	
6857555	✓
6857556	
6857557	
6857558	✓
6857559	
6857560	
6857561	✓
6857562	✓
⋮	⋮

Sorting Example

In this example, the number of phone numbers (4 000 000) is comparable to the size of the array (10 000 000)

The run time of such an algorithm is $\Theta(n)$:

- we make one pass through the data,
- we make one pass through the array and extract the phone numbers which are true

Algorithm

This approach uses very little memory and allows the entire structure to be kept in main memory at all times

We will term each entry in the bit vector a *bucket*

We fill each bucket as appropriate

Example

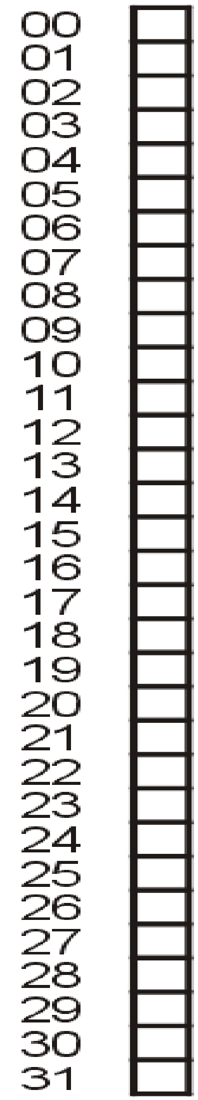
Consider sorting the following set of unique integers in the range 0, ..., 31:

20 1 31 8 29 28 11 14 6 16 15

27 10 4 23 7 19 18 0 26 12 22

Create an bit-vector with 32 buckets

– This requires 4 bytes



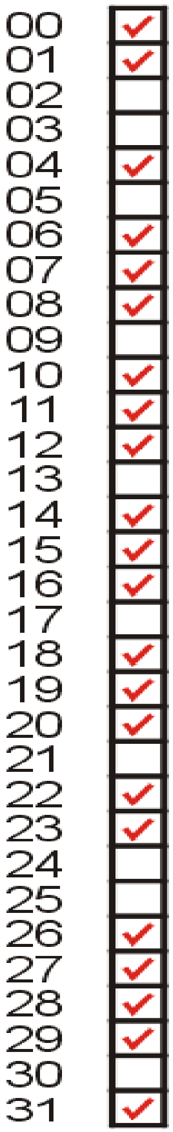
Example

For each number, set the corresponding bucket to 1

Now, just traverse the list and record only those numbers for which the bit is 1 (true):

0 1 4 6 7 8 10 11 12 14 15

16 18 19 20 22 23 26 27 28 29 31



Counting Sort

Modification: what if there are repetitions in the data

- In this case, a bit vector is insufficient

Two options, each bucket is either:

- a counter, or
- a linked list

The first is better if objects in the bin are the same

Example

Sort the digits

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

We start with an array of 10 counters, each initially set to zero:

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

Example

Moving through the first 10 digits

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

we increment the corresponding buckets

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

Example

Moving through remaining digits

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

we continue incrementing the corresponding buckets

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

Example

We now simply read off the number of each occurrence:

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

For example

- There are seven 2s
- There are two 4s

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

Run-time summary

Bucket sort always requires $\Theta(m)$ memory

The run time is $\Theta(n + m)$

Drawback

We must assume that the number of items being sorted is comparable to the possible number of values

- For example, if we were sorting $n = 20$ integers from 1 to one million, bucket sort may be argued to be $\Theta(n + m)$, however, in practice, it may be less so

Motivation for Radix Sort

By assuming that the data falls into a given range, we can achieve $\Theta(n)$ sorting run times

As any sorting algorithm must access any object at least once, the run time must be $\Theta(n)$

It is possible to use bucket sort in a more complex arrangement in radix sort if we want to keep down the number of buckets

Radix Sort

Suppose we want to sort 10 digit numbers with repetitions

- We could use bucket sort, but this would require the use of 10^{10} buckets
- With one byte per counter, this would require 9 GiB

This may not be very practical…

Radix Sort

Consider the following scheme

- Given the numbers

  16 31 99 59 27 90 10 26 21 60 18 57 17

- If we first sort the numbers based on their last digit only, we get:

  90 10 60 31 21 16 26 27 57 17 18 99 59

- Now sort according to the first digit:

  10 16 17 18 21 26 27 31 57 59 60 90 99

Radix Sort

The resulting sequence of numbers is a sorted list

Thus, consider the following algorithm:

- Suppose we are sorting decimal numbers
- Create an array of 10 queues
- For each digit, starting with the least significant
  - Place the ith number in to the bin corresponding with the current digit
  - Remove all digits in the order they were placed into the bins in the order of the bins

Example 01

Sort the following decimal numbers:

86 198 466 709 973 981 374 766 473 342

First, interpret 86 as 086

Example 01

Next, create an array of 10 queues:

0				
1				
2				
3				
4				
5				
6				
7				
8				
9				

Example 01

Push according to the 3rd digit:

086 198 466 709 973 981 374 766 473 342

0				
1	98**1**			
2	34**2**			
3	97**3**	47**3**		
4	37**4**			
5				
6	08**6**	46**6**	76**6**	
7				
8	19**8**			
9	70**9**			

and dequeue: 98**1** 34**2** 97**3** 47**3** 37**4** 08**6** 46**6** 76**6** 19**8** 70**9**

Example 01

Enqueue according to the 2nd digit:

981 342 973 473 374 086 466 766 198 709

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

and dequeue: 7**0**9 3**4**2 4**6**6 7**6**6 9**7**3 4**7**3 3**7**4 9**8**1 0**8**6 1**9**8

Example 01

Enqueue according to the 1st digit:

709 342 466 766 973 473 374 981 086 198

0	086			
1	198			
2				
3	342	374		
4	466	473		
5				
6				
7	709	766		
8				
9	973	981		

and dequeue: **0**86 **1**98 **3**42 **3**74 **4**66 **4**73 **7**09 **7**66 **9**73 **9**81

Example 01

The numbers

086 198 342 374 466 473 709 766 973 981

are now in order

The next example uses the binary representation of numbers, which is even easier to follow

Example 02

Sort the following base 2 numbers:

1111 11011 11001 10000 11010 101 11100 111 1011 10101

First, interpret each as a 5-bit number:

01111 11011 11001 10000 11010 00101 11100 00111 01011 10101

Next, create an array of two queues:

0							
1							

Example 02

Place the numbers

01111 11011 11001 10000 11010 00101 11100 00111 01011 10101

into the queues based on the 5^{th} bit:

0	10000	11010	11100					
1	01111	11011	11001	00101	00111	01011	10101	

Remove them in order:

10000 11010 11100 01111 11011 11001 00101 00111 01011 10101

Example 02

Place the numbers

10000 11010 11100 01111 11011 11001 00101 00111 01011 10101

into the queues based on the 4^{th} bit:

0	10000	11100	11001	00101	10101			
1	11010	01111	11011	00111	01011			

Remove them in order:

10000 11100 11001 00101 10101 11010 01111 11011 00111 01011

Example 02

Place the numbers

10000 11100 11001 00101 10101 11010 01111 11011 00111 01011

into the queues based on the 3rd bit:

0	10000	11001	11010	11011	01011			
1	11100	00101	10101	01111	00111			

Remove them in order:

10000 11001 11010 11011 01011 11100 00101 10101 01111 00111

Example 02

Place the numbers

10000 11001 11010 11011 01011 11100 00101 10101 01111 00111

into the queues based on the 2^{nd} bit:

0	10000	00101	10101	00111				
1	11001	11010	11011	01011	11100	01111		

Remove them in order:

10000 00101 10101 00111 11001 11010 11011 01011 11100 01111

Example 02

Place the numbers

10000 00101 10101 00111 11001 11010 11011 01011 11100 01111

into the queues based on the 1^{st} bit:

0	00101	00111	01011	01111				
1	10000	10101	11001	11010	11011	11100		

Remove them in order:

00101 00111 01011 01111 10000 10101 11001 11010 11011 11100

Example 02

The numbers

`00101  00111  01011  01111  10000  10101  11001  11010  11011  11100`

are now in order

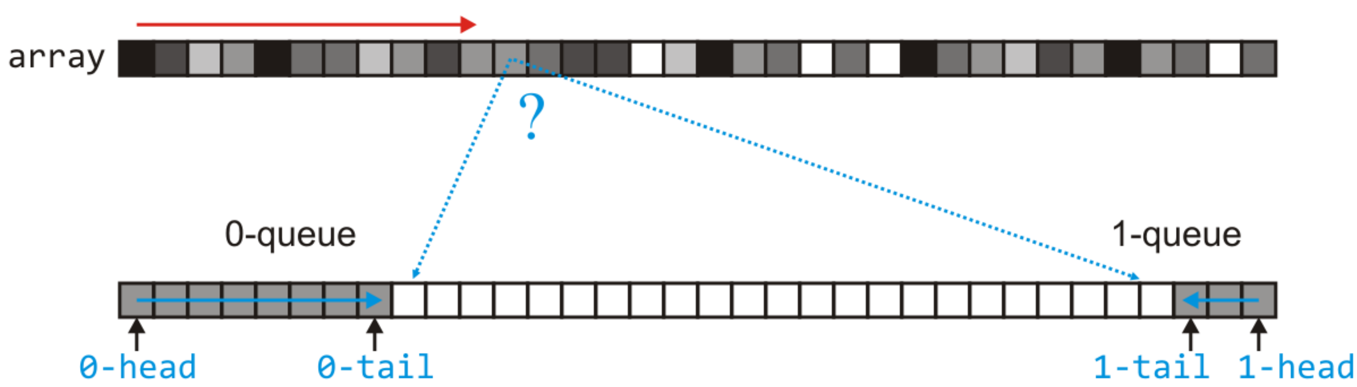
This required $5n$ enqueues and dequeues

- In this case, it $n = 10$

Sorting Binary Numbers

The implementation of multiple queues requires a lot of memory

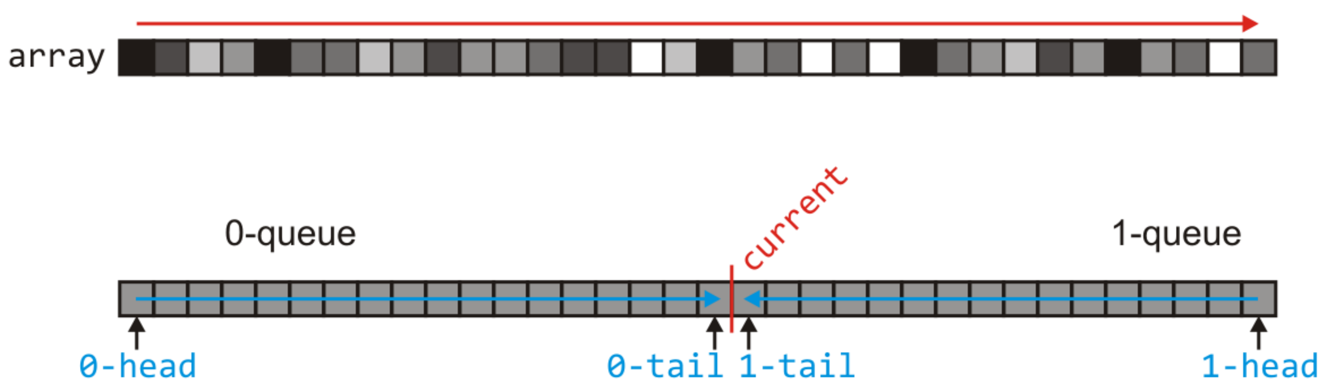
- Note, however, that the sum of the entries in the two queues is always n
- Create a new array of size n for a two-ended queues where
  - If the relevant bit is 0, enqueue it at at the front
  - Otherwise, the relevant bit is 1, so enqueue it at the back



Sorting Binary Numbers

Once we finish, the two queues have n entries

- Now, suppose the 0-queue has `current` entries
  - To iterate through the entries: go from `0` to `current - 1`,
    then from `n - 1` down to `current`
  - Go to the next bit and now use the original array as the queue



Example

Consider sorting the following 20 3-bit numbers

110	111	001	011	101	010	000	100	010	001	111	010	001	011	101	010	111	101	000	100

- These are the numbers

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

Example

Allocate a new array

110	111	001	011	101	010	000	100	010	001	111	010	001	011	101	010	111	101	000	100

Example

Sort on the least-significant bit

110	111	001	011	101	010	000	100	010	001	111	010	001	011	101	100	111	101	000	100

110	010	000	100	010	010	010	000	100	101	111	101	011	001	111	001	101	011	001	111

Example

Consider the original array as empty

110	010	000	100	010	010	010	000	100	101	111	101	011	001	111	001	101	011	001	111

Example

Sort on the intermediate bit

000	100	000	100	001	101	001	001	101	101	111	011	111	011	111	010	010	010	010	110

110	010	000	100	010	010	010	000	100	101	111	101	011	001	111	001	101	011	001	111

Example

Consider the other array as empty

000	100	000	100	001	101	001	001	101	101	111	011	111	011	111	010	010	010	010	110

Example

Sort on the most-significant bit

000	100	000	100	001	101	001	001	101	101	111	011	111	011	111	010	010	010	010	110

000	000	001	001	001	010	010	010	010	011	011	111	111	111	110	101	101	101	100	100

Example

Now we must copy back, swapping the second half

000	000	001	001	001	010	010	010	010	011	011	111	111	111	110	101	101	101	100	100

Example

Now we must copy back, swapping the second half

000	000	001	001	001	010	010	010	010	011	011	100	100	101	101	101	110	111	111	111

000	000	001	001	001	010	010	010	010	011	011	111	111	111	110	101	101	101	100	100

Example

Deleting the temporary array and we are finished

000	000	001	001	001	010	010	010	010	011	011	100	100	101	101	101	110	111	111	111

– Thus,

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

is now

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

Summary

Radix sort uses bucket sort on each digit of a set of numbers

- Interesting in theory, less useful in practice
- Useful only if sorting numbers with significant duplication
- The idea is used elsewhere