Linear Sorting

Dr. Timothy Kimber

March 2018

Recalling Comparison Sorts

The running time of these comparison sort algorithms

- Mergesort
- Heapsort
- Quicksort (expected)

are all $O(N \log N)$.

- Not possible for a comparison sort algorithm to do better

However, there are sorting methods that achieve $O(N)$ performance.

Counting Sort

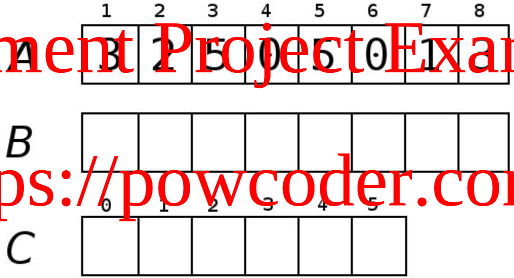
The Counting Sort algorithm sorts integers from a known range

- The key operation is to count the occurrences of all values

Counting Sort(Input: $A = [A_1, \ldots, A_N]$, k)

- For $i = 0$ to k
  - $C[i] = 0$ `<-- one entry per value in the range`
- For $j = 1$ to N
  - $C[A[j]] = C[A[j]] + 1$ `<-- count how many A[j] there are`
- For $i = 1$ to k
  - $C[i] = C[i] + C[i-1]$ `<-- how many less than or equal to i`
- For $j = N$ to 1
  - $B[C[A[j]]] = A[j]$
  - $C[A[j]] = C[A[j]] - 1$
- Return B

Counting Sort



- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

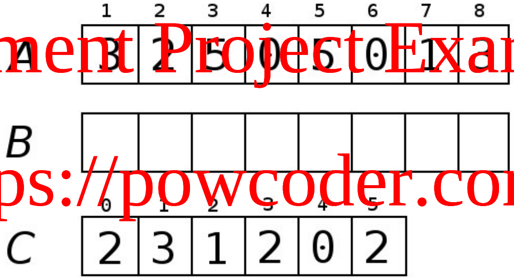
Counting Sort

	1	2	3	4	5	6	7	8
A	3	2	5	0	5	0	1	3
B								

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

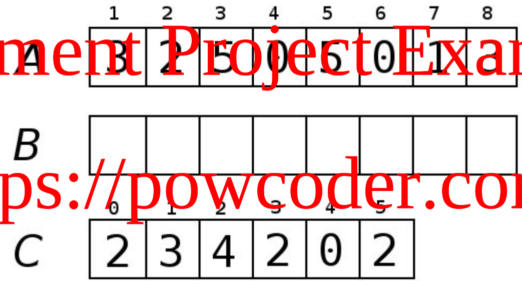
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



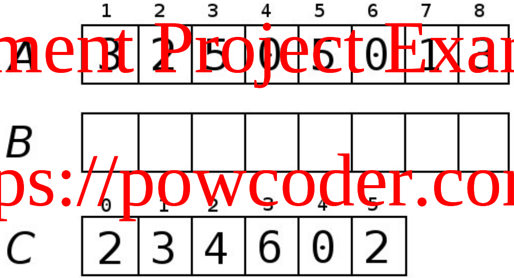
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



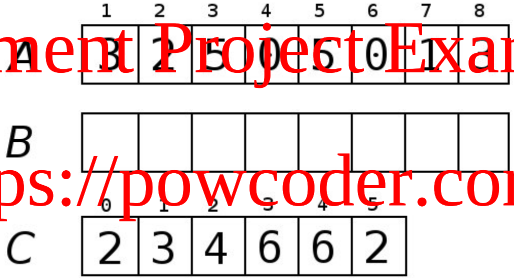
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



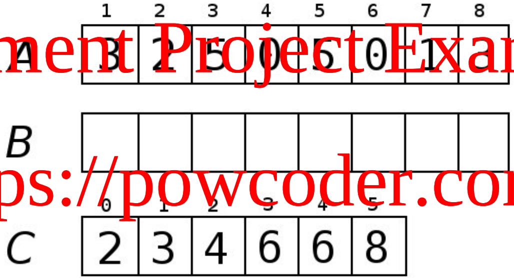
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



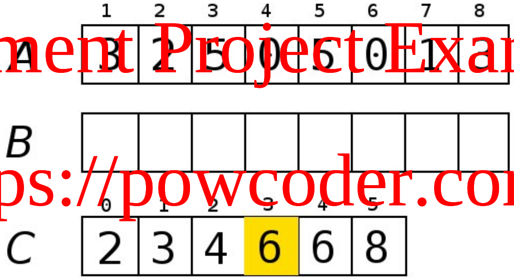
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



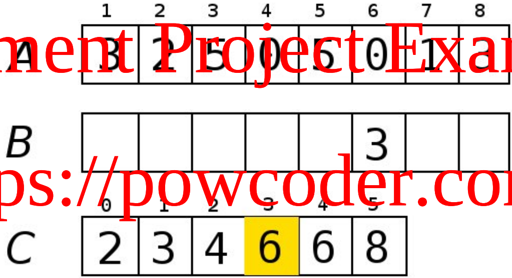
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



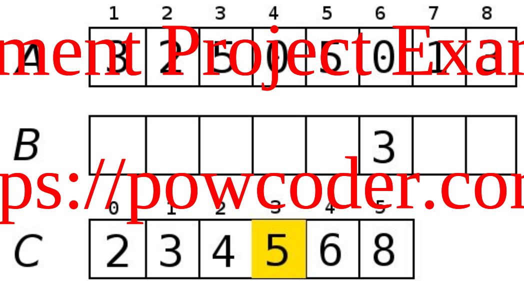
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



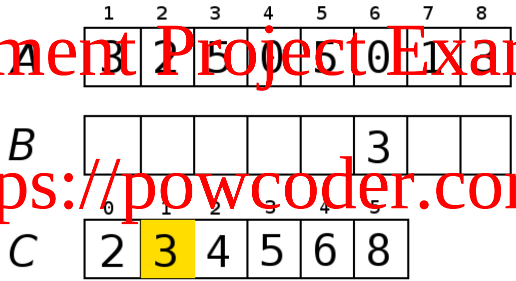
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



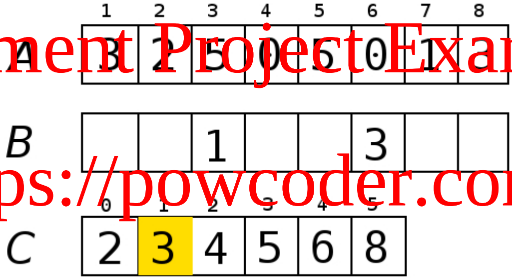
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



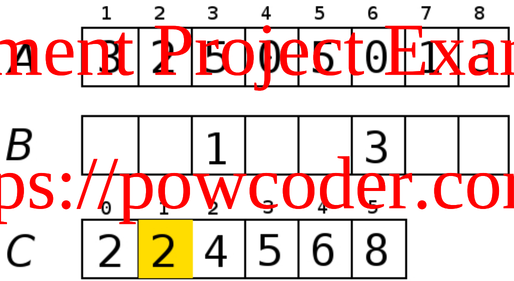
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



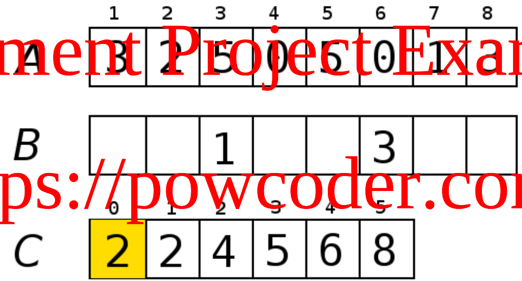
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



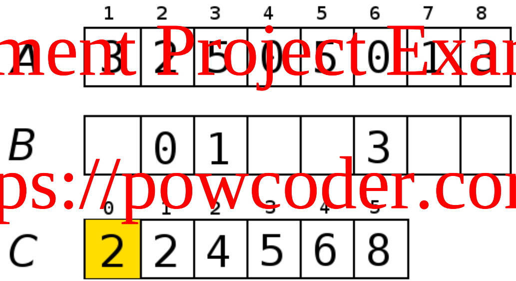
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



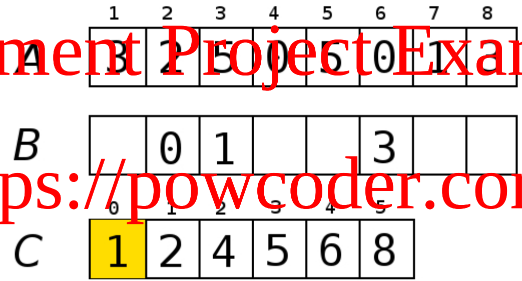
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



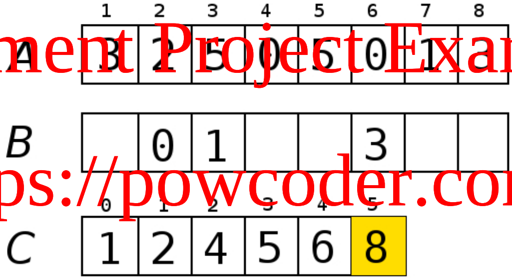
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



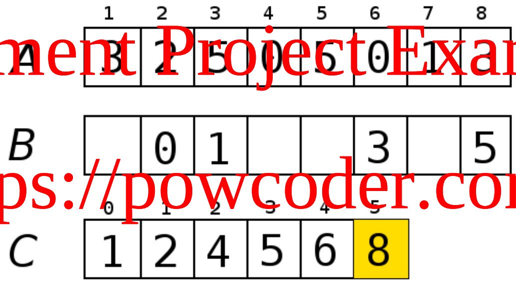
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort

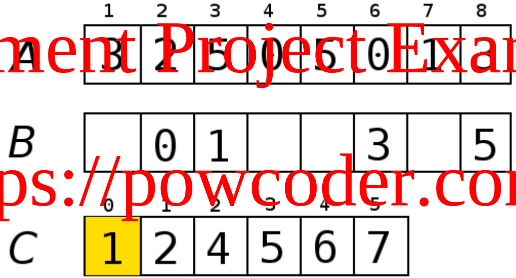
	1	2	3	4	5	6	7	8
A	3	2	5	0	5	0	1	3

B		0	1			3		5

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

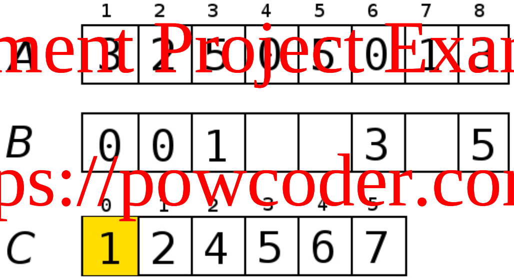
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

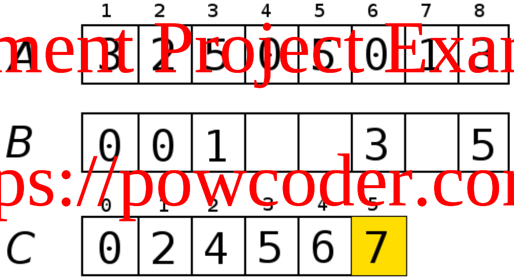
Counting Sort

	1	2	3	4	5	6	7	8
A	3	2	5	0	5	0	1	3
B	0	0	1			3		5

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

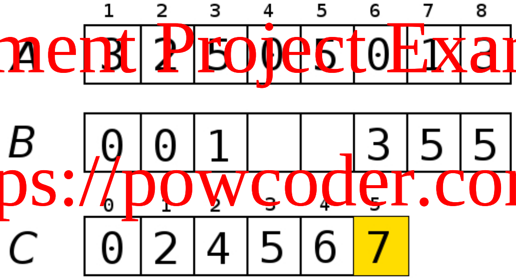
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



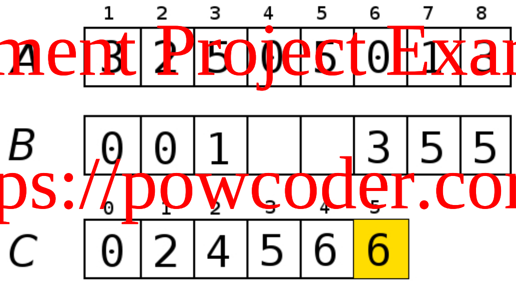
- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort

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

B	0	0	1			3	5	5

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

- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort

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

B	0	0	1	2		3	5	5

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

- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort

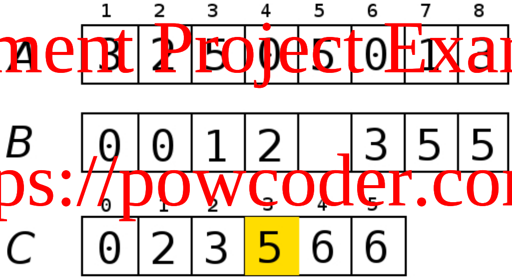
	1	2	3	4	5	6	7	8
A	3	2	5	0	5	0	1	3

B	0	0	1	2		3	5	5

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

- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort



- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort

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

B	0	0	1	2	3	3	5	5

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

- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort

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

B	0	0	1	2	3	3	5	5

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

- Counts of each value are saved into C
- Next the counts are accumulated
- Now $C[i]$ holds number of values $\leq i$
- Finally copy contents of A to correct positions in B using C

Counting Sort Time

Counting sort makes two passes through the input and two passes through the count table C

- So, the time taken is ...

Counting Sort(Input: $A = [A_1, \ldots, A_N]$, k)

- For $i = 0$ to k
  - $C[i] = 0$ `<-- one entry per value in the range`
- For $j = 1$ to N
  - $C[A[j]] = C[A[j]] + 1$ `<-- count how many A[j] there are`
- For $i = 1$ to k
  - $C[i] = C[i] + C[i-1]$ `<-- how many less than or equal to i`
- For $j = N$ to 1
  - $B[C[A[j]]] = A[j]$
  - $C[A[j]] = C[A[j]] - 1$
- Return B

Properties

Counting Sort runs in $\Theta(N + k)$ time.

Question

Under what circumstances does this become $O(N)$ time?

Counting Sort is also stable

- 'Different' 3s stay in the same order
- Can be important when the values are linked to other data
- This property is used by the next algorithm

Radix Sort

Radix Sort is used to sort a set of *d*-digit values

535		089
158		134
189		158
134	$\longrightarrow$	189
840		535
558		558
089		840

- It makes *d* passes through the data
- Each pass sorts on the *i*th digit only

Radix Sort

Radix Sort is used to sort a set of *d*-digit values

53**5**
15**8**
18**9**
13**4**
84**0**
55**8**
08**9**

- Counter-intuitively, the first sort is on the least significant digit
- It allows counting sort to be used per digit, over a much smaller range
- e.g. For decimal numbers there are 10 values to sort on

Radix Sort

Radix Sort is used to sort a set of *d*-digit values

84**0**
13**4**
53**5**
15**8**
55**8**
18**9**
08**9**

- Counter-intuitively, the first sort is on the least significant digit
- It allows counting sort to be used per digit, over a much smaller range
- e.g. For decimal numbers there are 10 values to sort on

Radix Sort

Radix Sort is used to sort a set of *d*-digit values

8**4**0
1**3**4
5**3**5
1**5**8
5**5**8
1**8**9
0**8**9

- Counter-intuitively, the first sort is on the least significant digit
- It allows counting sort to be used per digit, over a much smaller range
- e.g. For decimal numbers there are 10 values to sort on

Radix Sort

Radix Sort is used to sort a set of *d*-digit values

1**3**4
5**3**5
8**4**0
1**5**8
5**5**8
1**8**9
0**8**9

- Counter-intuitively, the first sort is on the least significant digit
- It allows counting sort to be used per digit, over a much smaller range
- e.g. For decimal numbers there are 10 values to sort on

Radix Sort

Radix Sort is used to sort a set of d-digit values

134
535
840
158
558
189
089

- Counter-intuitively, the first sort is on the least significant digit
- It allows counting sort to be used per digit, over a much smaller range
- e.g. For decimal numbers there are 10 values to sort on

Radix Sort

Radix Sort is used to sort a set of *d*-digit values

089
134
158

189
535
558
840

- Counter-intuitively, the first sort is on the least significant digit
- It allows counting sort to be used per digit, over a much smaller range
- e.g. For decimal numbers there are 10 values to sort on

Radix Sort

The algorithm is simple to state

Radix Sort(Input: $A = [A_1, \ldots, A_N]$, d)

- For $i = 0$ to d
  - Use a stable sort to sort A on digit i

- Counting Sort can implement the stable sort efficiently

The Radix

Radix Sort(Input: $A = [A_1, \ldots, A_N]$, d)

- For $i = 0$ to d
  - Use a stable sort to sort A on digit i

Discussion

You are sorting N numbers with Radix sort. You can *choose* what base the numbers will be represented in within the sort procedure.

- What base would you choose?
- Why?

The Radix

Assuming we have N numbers

- Expressed in base B
- Each with up to d digits

Radix sort takes $d(N + B)$ time.

- Base B has values in the range 0 to $(B - 1)$
- So, there are B distinct values to count

A base that is $O(N)$, e.g. base N, will limit the number of digits compared to some smaller base, while not dominating the time for each pass.

Binary

Binary representation allows you to pick any power of 2 as a base very cheaply. Assuming we have N numbers

- Each number has b bits
- Split the number into digits each comprising r bits

Radix Sort runs in $\Theta((b/r)(N + 2^r))$ time (if the stable sort takes $\Theta(N + k)$ time to sort values in the range $0 \dots k$).

- Each number has b/r digits
- Choose $r \sim \log_2(N)$ gives $\sim N$ values per digit

Under the assumption that $b = O(\log_2 N)$ the running time of Radix Sort is $\Theta(N)$. In practice, constant factors may mean that Quicksort is faster.