Non-Comparison Based Sorting

- We will examine three algorithms which under certain conditions can run in O(n) time.
  - Counting sort
  - Bucket sort
  - Radix sort

- Stable sort
  - A sorting algorithm where the order of elements having the same key is not changed in the final sequence.

Counting Sort

- Depends on assumption about the numbers being sorted
  - Assume numbers are in the range $1..k$
- The algorithm:
  - Input: A[1..n], where A[j] $\in \{1, 2, 3, \ldots, k\}$
  - Output: B[1..n], sorted (not sorted in place)
  - Also: Array C[1..k] for auxiliary storage

Counting Sort

```
CountingSort(A, B, k)
        for i=1 to k
                C[i]= 0;
        for j=1 to n
                C[A[j]] += 1;
        for i=2 to k
                C[i] = C[i] + C[i-1];
        for j=n downto 1
                B[C[A[j]]] = A[j];
                C[A[j]] -= 1;
```

This is called a ***histogram***.

Counting Sort Example

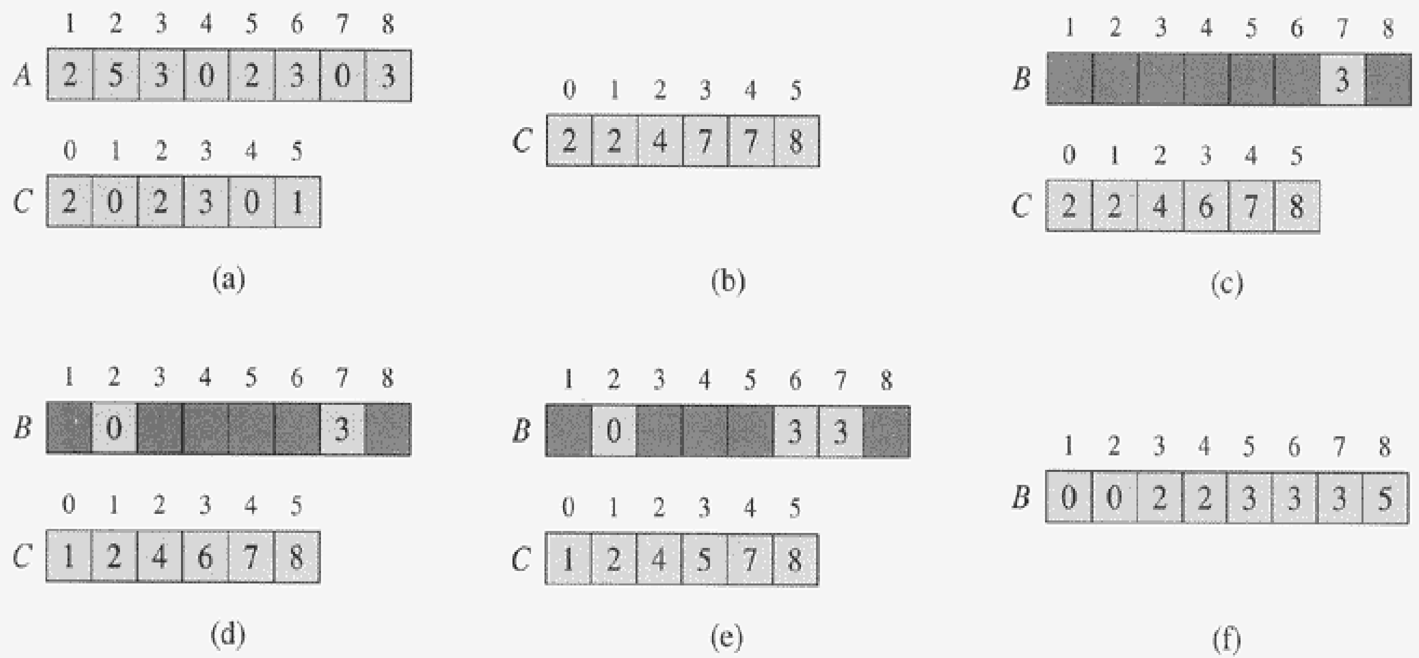


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Counting Sort

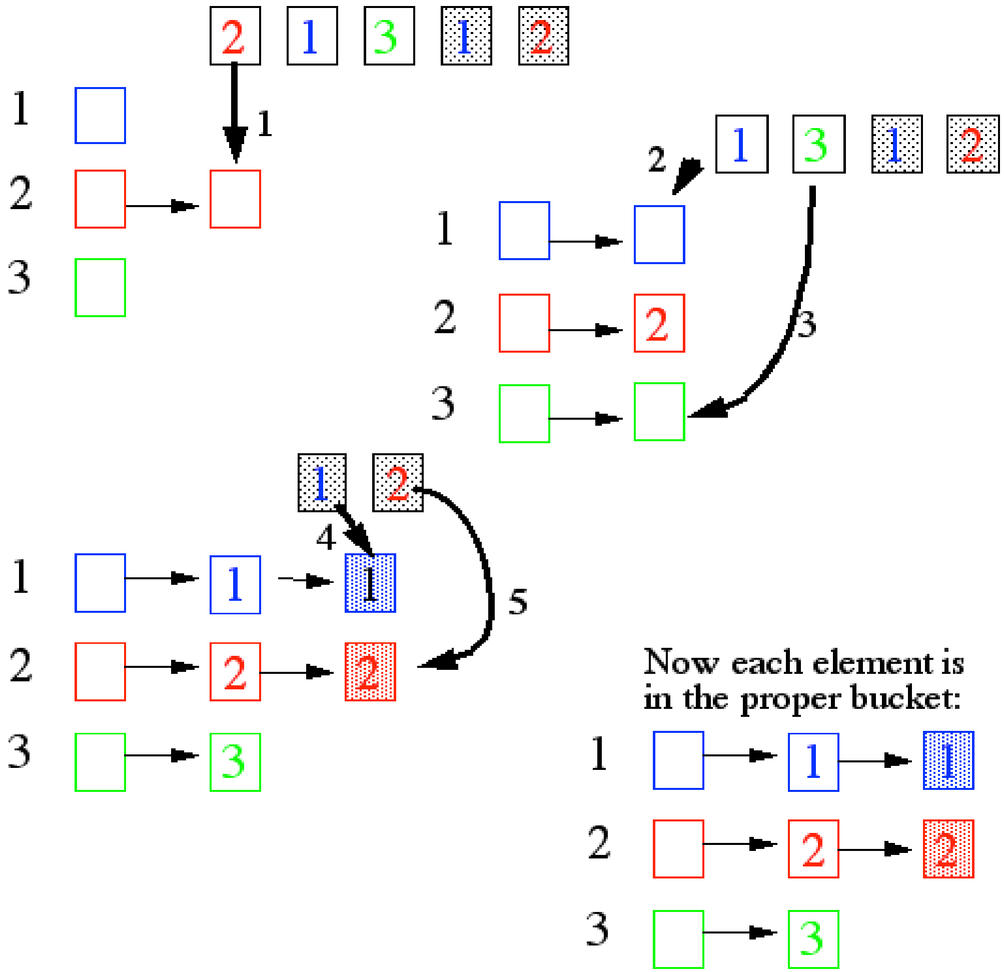
- Total time: $O(n + k)$
  - Works well if $k = O(n)$ or $k = O(1)$
- *Why don't we always use counting sort?*
  - Depends on range k of elements.
- *Could we use counting sort to sort 32 bit integers? Why or why not?*

Bucket Sort

- Bucket sort
  - Assumption: the keys are in [0, N)
  - Basic idea:
    1. Create *N* linked lists (*buckets*) to divide interval [0,N) into subintervals of size $\Theta(1)$
    2. Add each input element to appropriate bucket
    3. (Sort and) concatenate the buckets
  - Expected total time is O(n + N), with n = size of original sequence
    - if N is O(n) → sorting algorithm in O(n) !

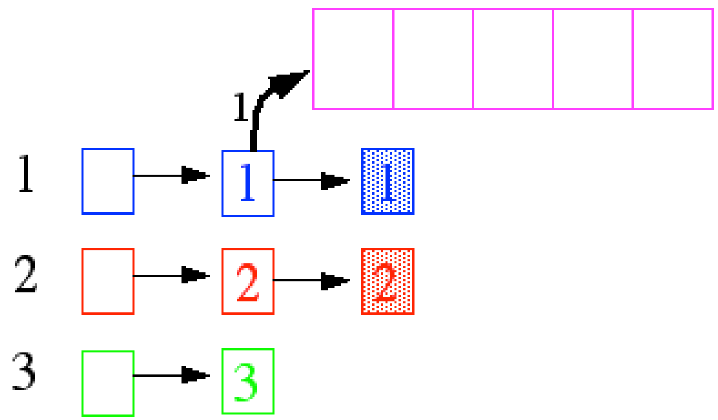
Bucket Sort

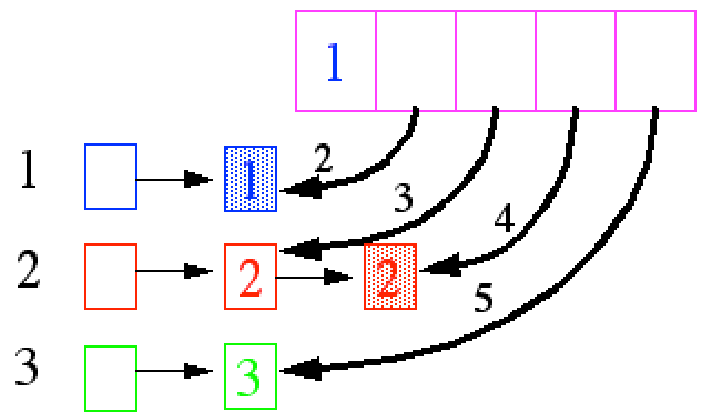
Each element of the array is put in one of the N "buckets"



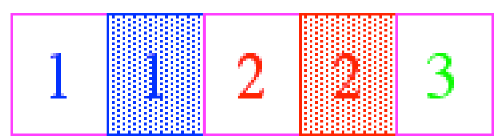
Bucket Sort

Now, pull the elements from the buckets into the array





At last, the sorted array (sorted in a stable way):



Does it Work for Real Numbers?

- What if keys are not integers?
  - Assumption: input is n reals from [0, 1)
  - Basic idea:
    - Create N linked lists (*buckets*) to divide interval [0,1) into subintervals of size $1/N$
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution → O(1) bucket size
    - Therefore the expected total time is O(n)

Radix Sort

- Used to sort punched card readers for census tabulation in early 1900's by IBM.
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - (Decimal digits use only 10 places!)
  - Problem: only one column can be sorted on at a time

Radix Sort

- Intuitively, you might sort on the most significant digit, then the second most significant, etc.
- Problem: lots of intermediate piles of cards to keep track of
- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit I
```

- Example: 216 579 626 571 023 189 169 573

Radix Sort

- *Can we prove it will work?*
- Inductive argument:
  - Assume lower-order digits {j: j<i}are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

- *What sort will we use to sort on digits?*
- Bucket sort is a good choice:
  - Sort n numbers on digits that range from 0..k
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and $k=O(n)$, takes $O(n)$ time

Radix Sort Example

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
  - Running time: $4(1 \text{ million} + 2^{16}) \approx 4$ million operations
- Compare with typical O(n lg n) comparison sort
  - Requires approx lg n = 20 operations per number being sorted
  - Total running time $\approx$ 20 million operations

Radix Sort

- In general, radix sort based on bucket sort is
  - Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice
- *Can radix sort be used on floating-point numbers?*

Summary: Radix Sort

- Radix sort:
  - Assumption: input has d digits ranging from 0 to k
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like bucket sort) for each stage
  - Each pass over n numbers with 1 digit takes time $O(n+k)$, so total time $O(dn+dk)$
    - When d is constant and $k=O(n)$, takes $O(n)$ time
  - Fast, Stable, Simple
  - Doesn't sort in place