Data Structures

Bucket sorting

- Stable sorting:
  - Consider a sequence, S = $((k_1, e_1), (k_2, e_2), \ldots, (k_n, e_n))$
  - $(k_i, e_i), (k_j, e_j) \epsilon$ S such that $k_i=k_j$
  - If (k_i, e_i) precedes (k_j, e_j) in S before sorting, the same is the case after soring S
- Stability is important, since applications may want to preserve the initial ordering of the items with the same key

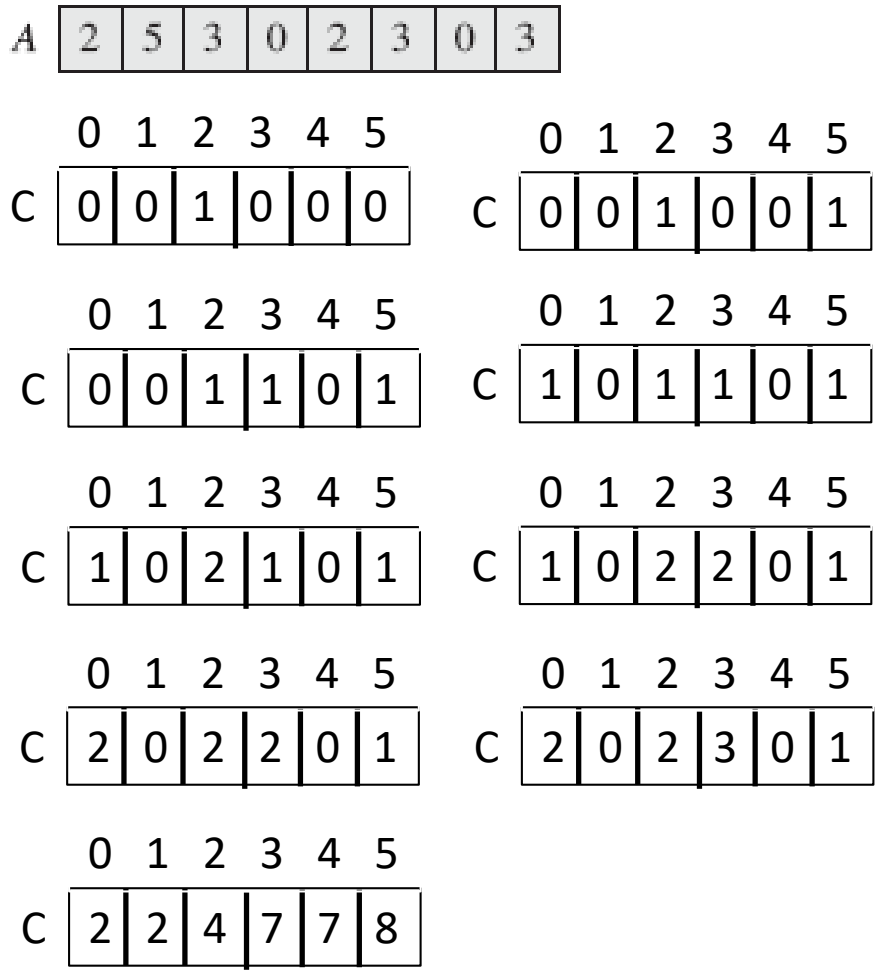
Counting sort

- Assumes that n input elements (integers) are in the range 0 to k, for some integer k
- If k is O(n), the complexity is Θ(n)
- General principle:
  - For each input element x, counts the number of elements that are less than x
  - Based on this information positions x in its correct position
- Has to be modified slightly if input has duplicates

Counting sort

Algorithm Counting_Sort(A, B, k)
{A[0. . n-1] is the input array; B[0. . n-1] holds the sorted output}

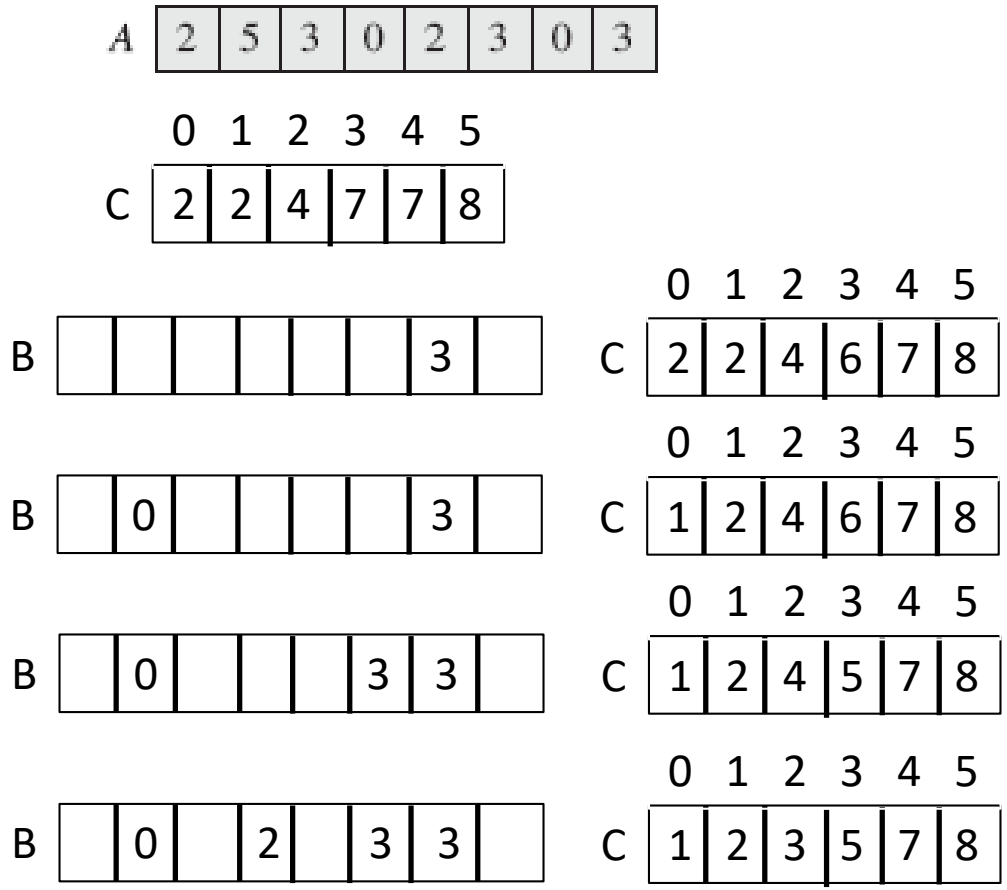
```
let C[0. . k] be a temporary array
for i ← 0 to k do
        C[i] ← 0
for j ←0 to n-1 do
        C[A[j]] = C[A[j]] +1
for i ← 1 to k
        C[i] = C[i] + C[i-1]
for j ← n-1 to 0
        B[C[A[j]]-1] = A[j]
        C[A[j]] = C[A[j]] -1
```



Counting sort

Algorithm Counting_Sort(A, B, k)
{A[0. . n-1] is the input array; B[0. . n-1] holds the sorted output}

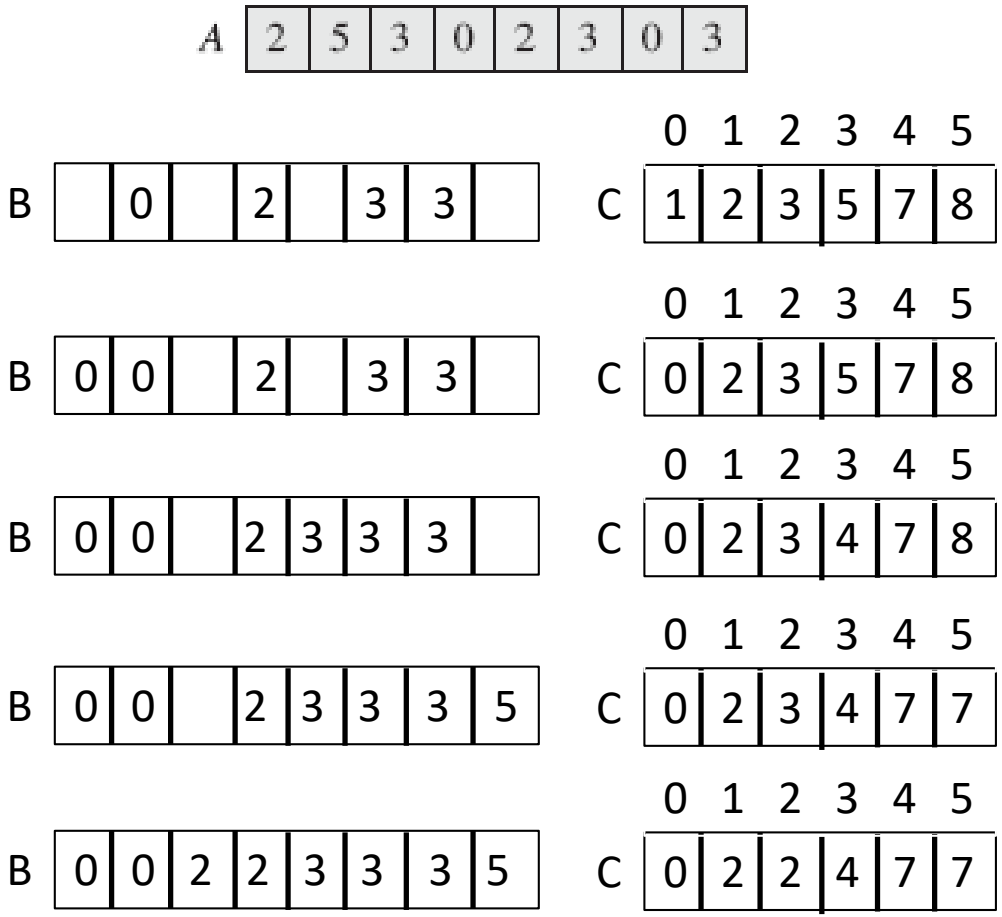
```
let C[0. . k] be a temporary array
for i ← 0 to k do
        C[i] ← 0
for j ←0 to n-1 do
        C[A[j]] = C[A[j]] +1
for i ← 1 to k
        C[i] = C[i] + C[i-1]
for j ← n-1 to 0
        B[C[A[j]]-1] = A[j]
        C[A[j]] = C[A[j]] -1
```



Counting sort

Algorithm Counting_Sort(A, B, k)
{A[0. . n-1] is the input array; B[0. . n-1] holds the sorted output}

```
let C[0. . k] be a temporary array
for i ← 0 to k do
        C[i] ← 0
for j ←0 to n-1 do
        C[A[j]] = C[A[j]] +1
for i ← 1 to k
        C[i] = C[i] + C[i-1]
for j ← n-1 to 0
        B[C[A[j]]-1] = A[j]
        C[A[j]] = C[A[j]] -1
```



Counting sort

```
Algorithm Counting_Sort(A, B, k)
{A[0. . n-1] is the input array; B[0. . n-1] holds the sorted output}
        let C[0. . k] be a temporary array
        for i ← 0 to k do
                C[i] ← 0
        for j ←0 to n-1 do
                C[A[j]] = C[A[j]] +1
        for i ← 1 to k
                C[i] = C[i] + C[i-1]
        for j ← n-1 to 0
                B[C[A[j]]-1] = A[j]
                C[A[j]] = C[A[j]] -1
```

Complexity: θ(k) + θ(n) + θ(k) + θ(n) is θ(k+ n)

Radix sort

- How to apply bucket sort in general contexts?
- Consider a sequence of items (k, l), where k and l are integers in the range [0, N-1], for some integer $N \geq 2$
- Lexicographic (dictionary) convention, where $(k_1, l_1) < (k_2, l_2)$ if
  - $K_1 < k_2$ or
  - $K_1 = k_2$ and $l_1 < l_2$
- Can generalize this definition to tuples of d numbers (d > 2)

Radix sort

- Sorts sequence S = ((k_1, e_1), (k_2, e_2), . . ., (k_n, e_n)) by applying bucket sort twice on S
- Using ordering key once and then using the second component
- In which order we should use bucket sort?
- S = ((3, 3) (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))
- S_1 = ((1, 5), (1, 2), (1, 7), (2, 5), (2, 3), (2, 2), (3, 3), (3, 2))
- $S_{1, 2}$ = ((1, 2), (2, 2), (3, 2), (2, 3), (3, 3), (1, 5), (2, 5), (1, 7))
- S_2 = ((1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7))
- $S_{2,1}$ = ((1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3))

Radix sort

- First using second component and then by using first component
- If two elements have equal values for second component, then their relative order in the starting sequence is preserved
- We can generalize this sorting to tuples with more than two components

Radix sort

- Consider a sequence of “n” numbers each with at most “d” digits
- Perform bucket sort with respect to the least significant digit
- Continue bucket sort with the next (previous) to least significant digit and end with the most significant digit

Algorithm Radix_Sort(A, d)

for i ← 1 to d

use a stable sorting algorithm to sort A on digit i

Radix sort

064, 008, 216, 512, 027, 729, 000, 001, 343, 125

000, 001, 512, 343, 064, 125, 216, 027, 008, 729

000, 001, 008, 512, 216, 125, 027, 729, 343, 064

000, 001, 008, 027, 064, 125, 216, 343, 512, 729

Radix sort

Algorithm Radix_Sort(A, d)

for i ← 1 to d

use a stable sorting algorithm to sort A on digit i

- Let S be a sequence of n key-element items, each of which has a key $(k_1, k_2, \ldots, k_d)$, where k_i is an integer in the range [0, N-1], for some integer $N \geq 2$; S can be sorted in time O(d(n+N)) using radix sort

Comparison of sorting algorithms

- If implemented well, insertion sort runs in O(n + k) time, where k is the number of inversions in the input
- What is an inversion?
  - A pair of elements that are out of order
- Consider the input: 34, 8, 64, 51, 32, 21
- Inversions: (34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21), (32, 21)

34	8	64	51	32	21
8	34	64	51	32	21
8	34	64	51	32	21
8	34	51	64	32	21
8	32	34	51	64	21
8	21	32	34	51	64

Comparison of sorting algorithms

- Insertion sort is excellent if the input size is small say less 50
- Effective when number of inversions is small
- $O(n^2)$ (worst-case)
- The merge sort runs in O(n log n) time in the worst-case
- Overhead makes it difficult to implement merge sort in place
- The expected running time of quick sort is O(n log n)
- Worst-case running time is $O(n^2)$
- If we have to sort by integer keys, or d-tuples of integer keys, then we can use bucket or radix sort, where [0, N-1] is the range for keys
- O(d(n+N)) (d =1 for bucket sort)