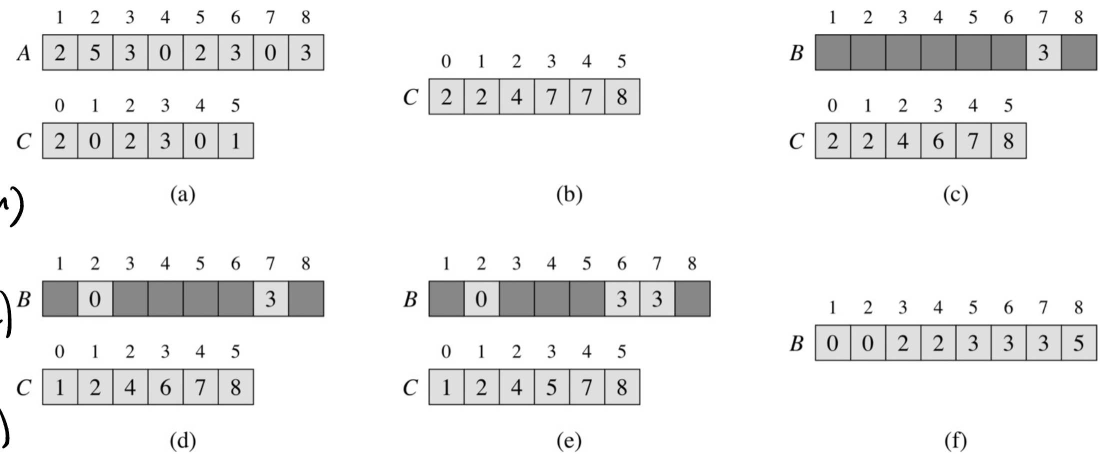
Counting Sort

- Assumes each element is an integer in the range 0 to k for some integer k
- When $k = O(n)$, the sort runs in $\Theta(n)$ time
- Determines for each element x the number of elements less than x
  - Use this to place x directly in its position
- Requires extra storage

k must be reasonably sized since we need an extra array of size k+1. We also need an extra array of size n

COUNTING-SORT(A, B, n, k)

```
let C[0..k] be a new array
for i = 0 to k                       Θ(k)
    C[i] = 0
for j = 1 to n
    C[A[j]] = C[A[j]] + 1            Θ(n)
for i = 1 to k
    C[i] = C[i] + C[i - 1]           Θ(k)
for j = n downto 1
    B[C[A[j]]] = A[j]                Θ(n)
    C[A[j]] = C[A[j]] - 1
```



$$T(n) = \Theta(k) + \Theta(n) + \Theta(k) + \Theta(n) = \Theta(n+k)$$

if $k = O(n)$ then $T(n) = \Theta(n)$

Sorting Stability

- A sort is stable if the items with the same value appear in the same order in the output as they do in the input

- Stability matters when the item's value is a key and there is additional data associated with the item (satellite data)

Radix Sort

- Works for items with a fixed number of digits or characters or fields
- Starting with the least significant digit, sort the items by that digit using a stable sort, then work your way through the rest of the digits in increasing significance

RADIX-SORT(A, d)

for $i = 1$ **to** d

use a stable sort to sort array A on digit i

digit 1 is least significant
" d is most "

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Lemma 8.4 in the book

Given n d-digit numbers in which each digit can take on up to k possible values, Radix Sort correctly sorts these numbers in $\Theta(d(n+k))$ if using a $\Theta(n+k)$ algorithm like counting sort to sort the digits

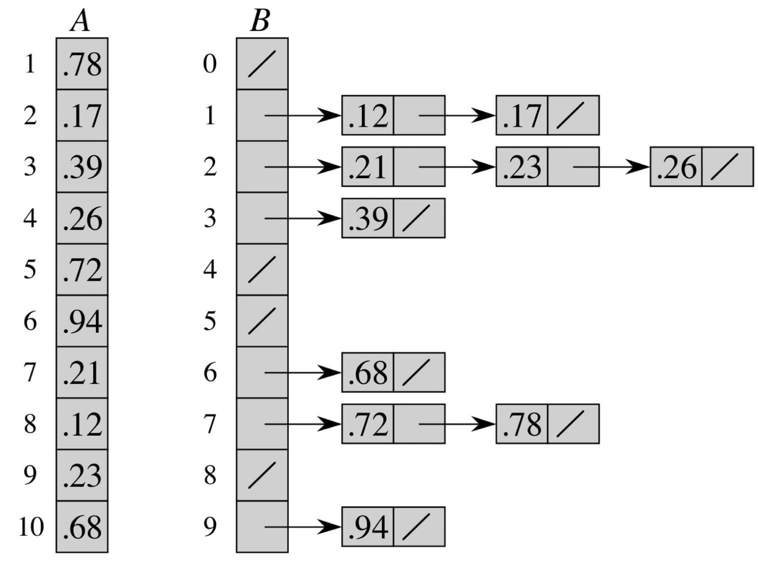
If d is constant and $k = O(n)$, radix sort is $\Theta(n)$ time

- Works for unsigned binary numbers with a fixed number of bits
    - "digits" can be bits or bytes
    - Requires overhead since counting sort is not in-place
    - Quicksort is still often faster

Bucket Sort

- Assumes input is drawn from a uniform distribution
- Each element is from the interval $[0, 1)$ (could be 0 but not 1)
- To sort n items, divide the interval into n equal sized buckets
- Distribute the n items into the buckets
- Sort each bucket
- Each bucket will have very few elements (1 on average)

```
BUCKET-SORT(A, n)
  let B[0 . . n − 1] be a new array
  for i = 0 to n − 1
      make B[i] an empty list
  for i = 1 to n
      insert A[i] into list B[⌊n · A[i]⌋]
  for i = 0 to n − 1
      sort list B[i] with insertion sort
  concatenate lists B[0], B[1], . . . , B[n − 1] together in order
  return the concatenated lists
```



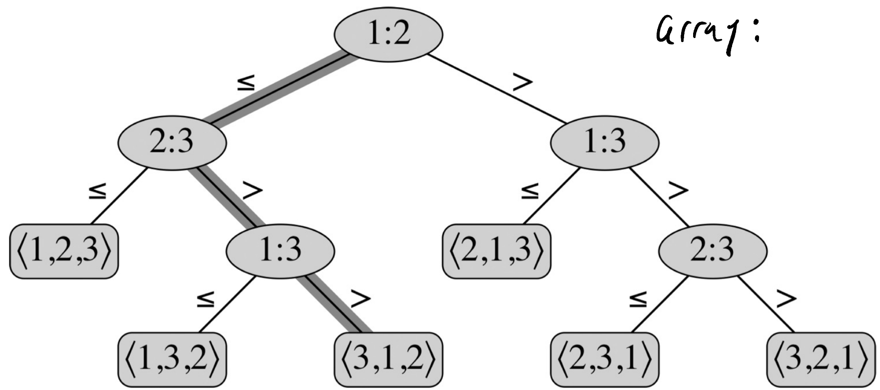
(a) (b)

- Insertion sort is quadratic ($\Theta(n^2)$), but we expect that the sum of squares of the bucket sizes is linear in the number of elements
- Average case is $\Theta(n)$
- See the book for proof

- Previous sorts we looked at were comparison sorts
  - insertion, merge, heap, quick

- The worst case running time for any comparison sort is $\Omega(n \lg n)$

Decision tree for arriving at all possible permutations of a 3-element array:



There are $n!$ permutations of an n-element list

This is a binary tree, so height $h \geq \lg(n!)$

$$h = \Omega(n \lg n)$$ (See equation 3.19 in the book)

A comparison sorting algorithm that is $\Theta(n \lg n)$ time

is asyptotically optimal