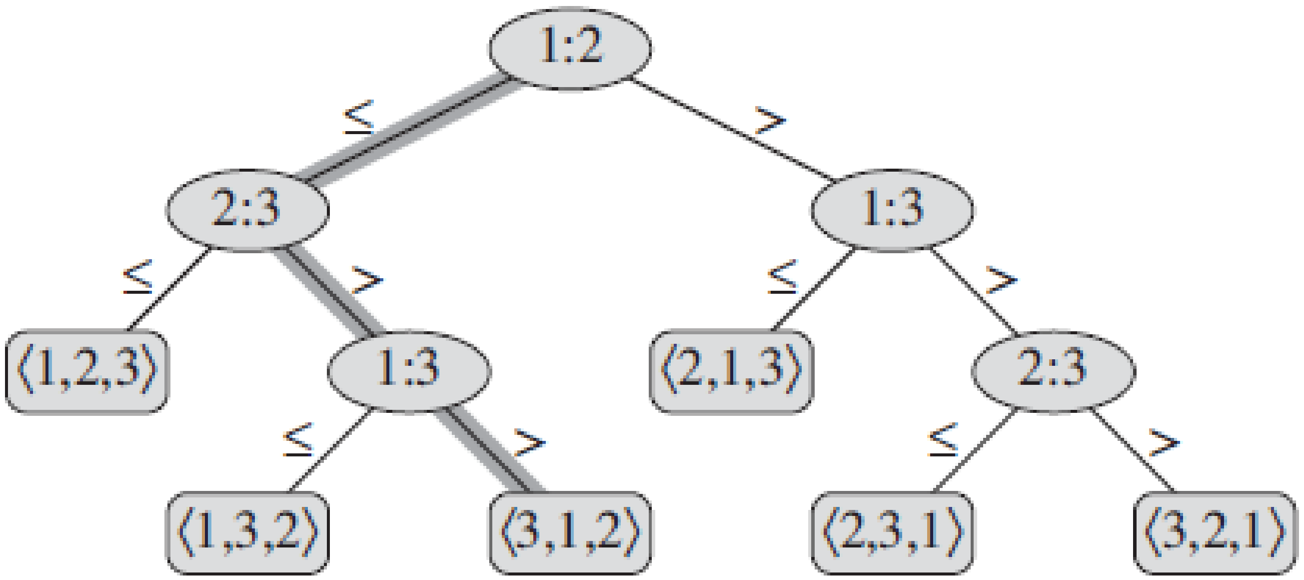
Sorting in Linear Time

Counting sort, Radix sort, Bucket sort

Decision tree model for comparison sort



Decision tree model

- Each node represents a comparison between elements A[i] and A[j]
- The result of comparison directs the search path
- If A[i] < A[j], the search takes left path and right otherwise
- Each leaf node represents one of the n! possible permutations of the input sequence
- Thus, the leaf node that is obtained in the end corresponds to the correct sorted sequence
- For any comparison based sorting algorithm to be correct, there must be n! leaf nodes in it and every leaf node must be reachable from the root

Lower Bound for Comparison sort

- For any algorithm, we can represent its working in the form of the decision tree, the height of the decision tree represents a bound on the worst case
- Suppose, the decision tree has height ‘h’ and has ‘l’ reachable nodes
- A binary tree with height h can have at most 2^h leaf nodes

$$\Rightarrow l \leq 2^h$$

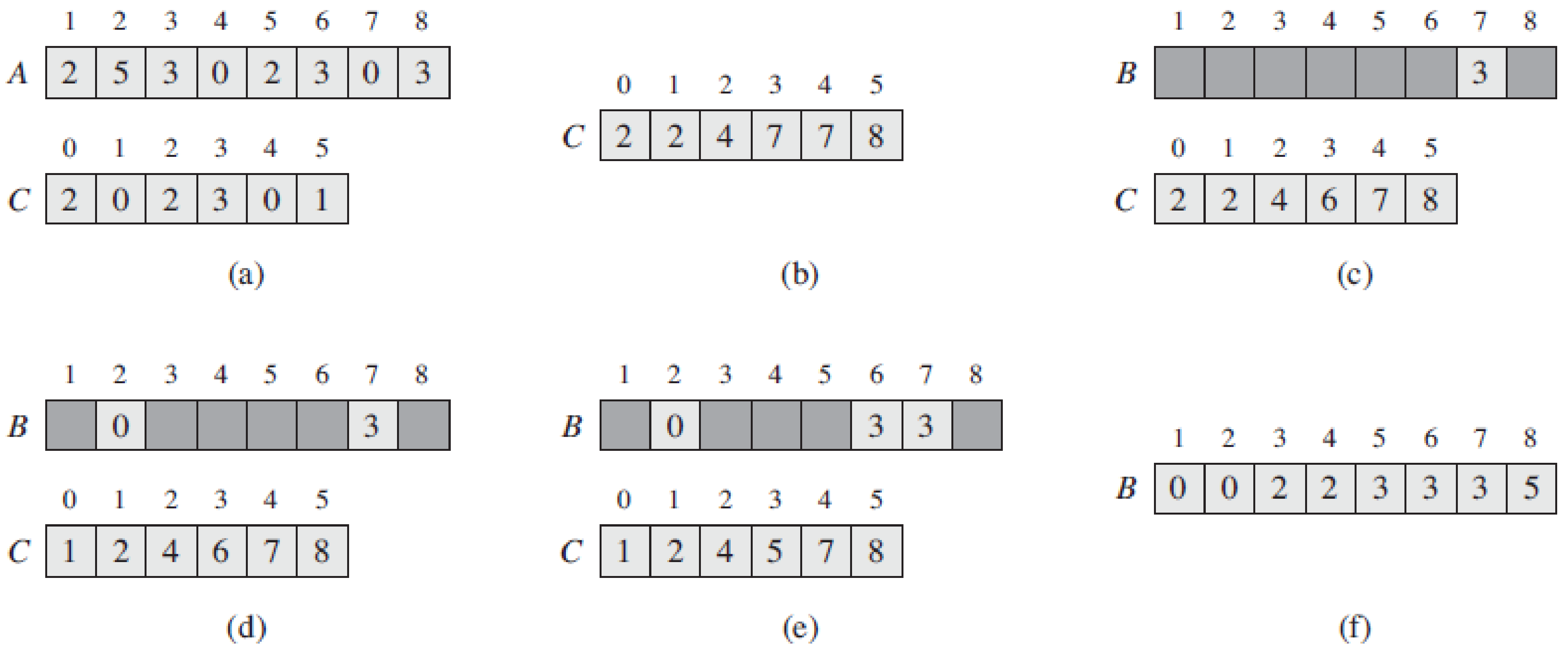
- Further, since all possible permutations of the input sequence must appear as reachable leaf nodes for the algorithm to be correct we have,

$$n! \leq l$$
$$\Rightarrow n! \leq 2^h$$
$$\Rightarrow h \geq \lg(n!)$$

- Since nlgn is a lower bound on lg(n!) we have

$$h = \Omega(nlgn)$$

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```


1 2 3 4 5 6 7 8
A 2 5 3 0 2 3 0 3
0 1 2 3 4 5
C 2 0 2 3 0 1
(a)
0 1 2 3 4 5
C 2 2 4 7 7 8
(b)
1 2 3 4 5 6 7 8
B 3
0 1 2 3 4 5
C 2 2 4 6 7 8
(c)
1 2 3 4 5 6 7 8
B 0 3
0 1 2 3 4 5
C 1 2 4 6 7 8
(d)
1 2 3 4 5 6 7 8
B 0 3 3
0 1 2 3 4 5
C 1 2 4 5 7 8
(e)
1 2 3 4 5 6 7 8
B 0 0 2 2 3 3 3 5
(f)

Analysis

- Runs in $\theta(n + k)$ time
- Often used when k = O(n) when it runs in $\theta(n)$ time
- Beats the lower bound of comparison sort i.e. $\Omega(nlgn)$
- It is a *Stable* sorting algorithm: numbers in the output array appear in the same order as in the input array
- Useful in Radix sort

Radix Sort

```
RADIX-SORT(A, d)
  for i = 1 to d
      use a stable sort to sort array A on digit i
```

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

Analysis

- For a sequence of n numbers in the range 0 to k, each having d-digits, the algorithm runs in $\theta\left(d(n+k)\right)$ time
- Runs in linear time when d is constant and $k = O(n)$
- Usually less preferred over quicksort
- Quicksort can use memory caches etc. in better manner
- Counting sort which is used as an intermediate sorting method, does external sorting which can be avoided in quicksort

Proof that counting sort works

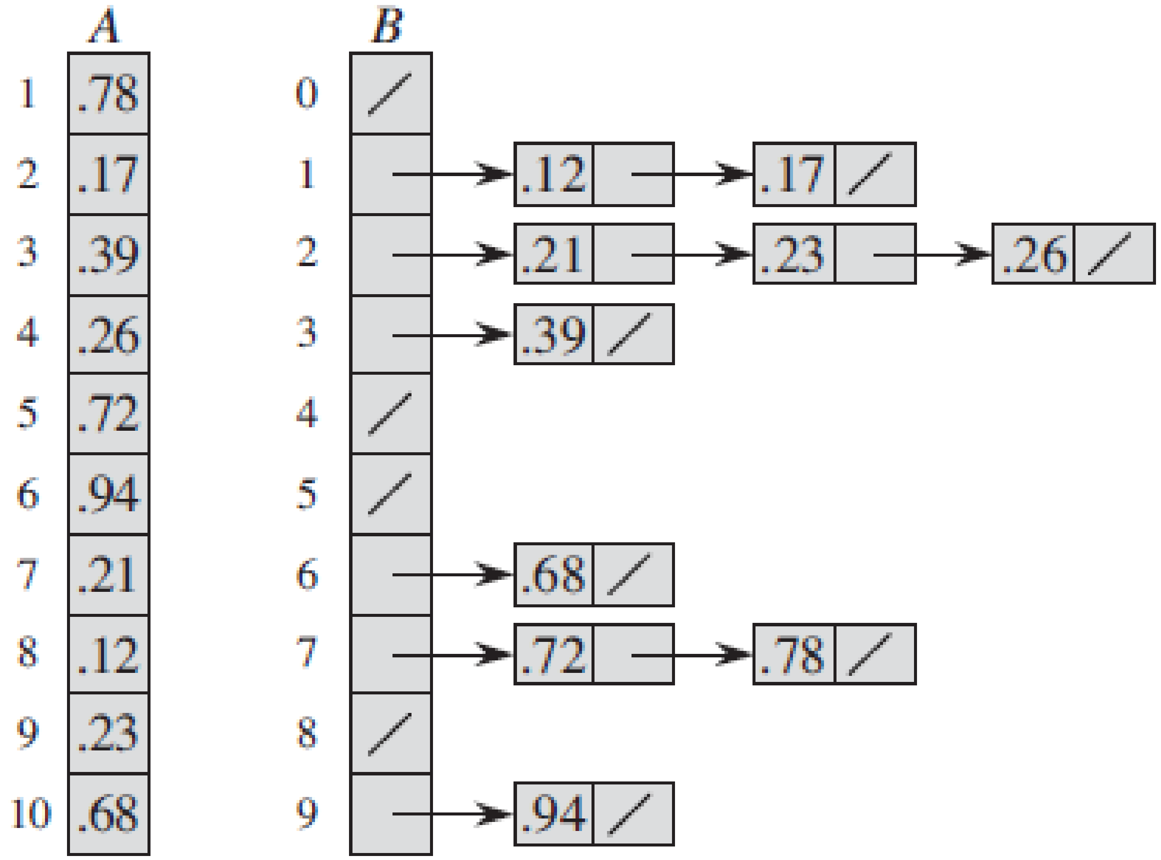
- Induction method is used to prove that it works
- For a 1-digit number, it sorts the array correctly
- For 2-digit number sequence, the sorting on the most significant digits maintains the lower order sorting order for tie and therefore, the sorting is correct for 2-digit
- Assume true for n-digit and prove the same for n+1th digit
- This proves that the counting sort works correctly

Bucket Sort

- Assumes that the input is drawn from a uniform distribution
- Average running time is O(n)
- Input is generated using a random process and uniformly and independently distributed over [0, 1)
- For sorting the interval is divided into n equal-size sub-intervals called *buckets*
- Numbers are distributed among the buckets
- We expect each bucket to have only small number of elements
- Each bucket is then sorted and elements are listed in order

BUCKET-SORT(A)

```
let B[0..n-1] be a new array
n = A.length
for i = 0 to n-1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n-1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n-1] together in order
```



A
B
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94

(a) (b)

Correctness of Bucket Sort

- For any two elements A[i] and A[j]
- Suppose $\mathrm{A[i]} \leq \mathrm{A[j]} \Rightarrow \lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$
- Therefore, either A[i] will go to same bucket as A[j] or it will go to a bucket lower than that of A[j]
- In either case, it will be placed before A[i] in the final sorted array
- Thus, the algorithm is correct

Complexity Analysis

- Each step takes O(n) time except lines 7 and 8 that perform sorting of each bucket using insertion sort
- The running time can be described as:

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} O({n_i}^2)$$

- Since, the input is a uniform distribution we can calculate the expected running time by calculating the expectation of the above equation

$$E[T(n)] = E\left[\theta(n) + \sum_{i=0}^{n-1} O({n_i}^2)\right]$$

$$= \theta(n) + \sum_{i=0}^{n-1} E[O({n_i}^2)]$$

$$= \theta(n) + \sum_{i=0}^{n-1} O[E({n_i}^2)]$$

- $E[{n_i}^2]$ value will be same for all the buckets as the elements are distributed uniformly
- It can be shown that $E[{n_i}^2] = 2 - 1/n$
- Thus, the average case running time of bucket sort comes out to be

$$\theta(n) + n \times O\left(2 - \frac{1}{n}\right) = \theta(n)$$

- Thus, bucket sort runs in linear time if the input is drawn from uniform distribution
- The linear running time can also be obtained if the input is not drawn from a uniform distribution as long as the sum of squares of bucket sizes is linear in total number of elements