Chapter 8: Sorting in Linear Time

Lower Bound for Sorting

Comparison sort: A sorting algorithm is based only on comparisons between the input elements.

Comparison sorts can be viewed abstractly in terms of decision trees.

Decision trees: Given an input sequence $< a_1, a_2, \ldots, a_n >$,

- Each internal node is denoted by $a_i : a_j$, for $1 \leq i, j \leq n$.
- Each leaf node is denoted by a permutation $< \pi(1), \pi)2), \ldots, \pi(n) >$.
- Each path from the root to a leaf corresponds to an execution of the sorting algorithm for a specific input.
- The left branch of an internal node means $a_i \leq a_j$.
  The right branch for an internal node means $a_i > a_j$.
- There are $n!$ permutations for n elements $\Longrightarrow$ there are at least $n!$ leaf nodes.

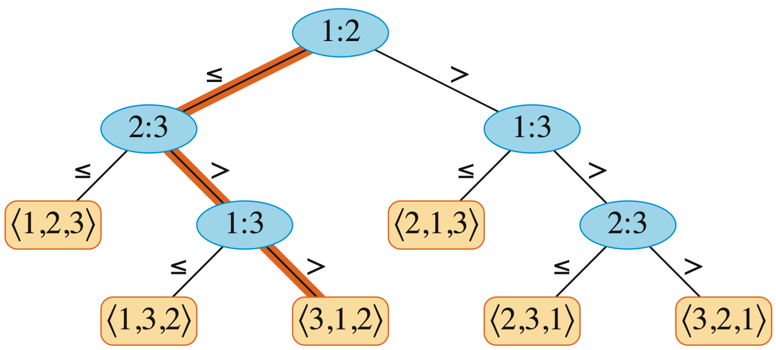
```
Ex: The decision tree for insertion sort with 3 elements.
                         a1:a2
                      /         \
                 a2:a3           a1:a3
                /     \         /     \
            <1,2,3>  a1:a3  <2,1,3>    a2:a3
                    /     \           /     \
                <1,3,2> <3,1,2>   <2,3,1> <3,2,1>
```

```
Ex: The decision tree for selection sort with 3 elements.
                         a1:a2
                     /          \
               a1:a3              a2:a3
             /      \           /      \
          a2:a3    a1:a2       a1:a3    a1:a2
        /     \      /        /     \         \
  <1,2,3> <1,3,2> <3,1,2>  <2,1,3> <2,3,1>  <3,2,1>
```

There are $n!$ permutations for n elements $\Longrightarrow$ the tree has at least $n!$ leaves.
Let h be the height of the tree $\Longrightarrow$ the tree has no more than 2^h leaves.
Thus,

$$n! \leq 2^h \Longrightarrow h \geq \log(n!) \Longrightarrow h = \Omega(n \log n)$$

The height of a decision tree means the number of comparisons for sorting in the worst-case.



$\Longrightarrow$ All comparison sorts have a lower bound running time $\Omega(n \log n)$ for the worst-case.

$\Longrightarrow$ It is impossible to find a new comparison based sorting algorithm that is asymptotically better than merge sort.

However, some non-comparison based sorting algorithms may run in linear time.

Counting Sort

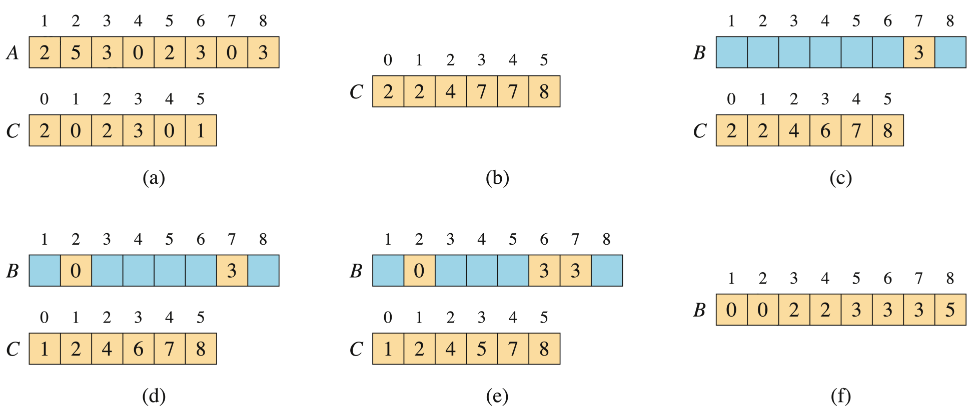
Counting sort assumes that each of the n input elements is an integer within a range $[0..k]$, for some integer k.

An input array $A[1:n]$, an output array $B[1:n]$, and a temporary working storage $C[0:k]$ are necessary for this algorithm. Thus, counting sort does not sort in place.

During the execution of counting sort, $C[i]$ maintains the # of elements less than or equal to i. For each element j in A, put it into B at position $C[j]$.

```
Counting-Sort(A, n, k)
let B[1:n] and C[0:k] be new arrays
for i = 0 to k
    C[i] = 0
for j = 1 to n
    C[A[j]] = C[A[j]] + 1
// C[i] contains the # of elements that is equal to i
for i = 1 to k
    C[i] = C[i] + C[i-1]
// C[i] now contains the # of elements less than or equal to i
// Copy A to B, starting from the end of A
for j = n downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1 // to handle duplicate values
```

Ex:



- Running time analysis:

  Counting-Sort's running time is $\Theta(n+k)$.

  If $k = O(n)$, then $\Theta(n+k) = \Theta(n)$. It's a linear time!

- Counting sort is a **stable** sorting algorithm: elements with the same value in the output array should be in the same order as they do in the input array.

  - Insertion Sort: stable (if no "=" sign in comparison)
  - Selection Sort: stable (if no "=" sign in comparison)
  - Merge Sort: stable (if the "=" sign is in the comparison)
  - Heap Sort: not stable (exchange $A[1] \longrightarrow A[n]$)
  - Quick Sort: not stable.
    Ex: input: $< 5, 5', 5'', 3, 4 >$ and the output is $< 3, 4, 5'', 5, 5' >$.

Radix Sort

The Radix-Sort sorts by the least significant digit first, then by the 2nd least significant digit, ...
The sorting algorithm used to sort each digit should be stable; otherwise Radix-Sort will not work.

```
Ex:
    213           321           312           123
    312           312           212           132
    123           212           213           212
    212  stable   132  stable   321  stable   213
    321  ------>  213  ------>  123  ------>  312
    132           123           132           321
      ^            ^            ^
```

```
Ex:
    213           321               312            123
    312           312               213 <-         132
    123           212               212 <-         213 <-
    212  stable   132  not stable   321  stable    212 <-
    321  ------>  213  ----------->  123  ------>   312
    132           123               132            321
      ^            ^                ^
```

Another example:

```
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
```

```
Radix-Sort(A, d)
1. for i = 1 to d
2.     use a stable sort to sort array A[1:n] on digit i
```

Two questions:

- Why does the algorithm need to use a stable sort to sort each digit?
- Why does the sorting start from sorting the least significant digit first?

`Running time analysis`: Suppose all n numbers have d or less digits.

If we use Counting-Sort as the sorting algorithm to sort each digit, then the running time for Radix-Sort is $d \cdot \Theta(n+k) = \Theta(dn+dk)$

If $k = O(n)$ and d is a constant, then the running time becomes $\Theta(n)$.

It's a linear time!

Recommended Exercise: Show how to sort n integers in the range 0 to $n^2 - 1$ in $O(n)$ time.

Solution:

We will assume that each digit has value in the range $0..n-1$, that is $k = n$. That is, as if the numbers are written in radix-n or base-n (instead of the usual base 2 or base 10).

Counting sort now requires $O(n+k) = O(n)$ time.

Then each number will have two digits, so $d = 2$ as the range of the numbers is $[0..n^2 - 1]$.

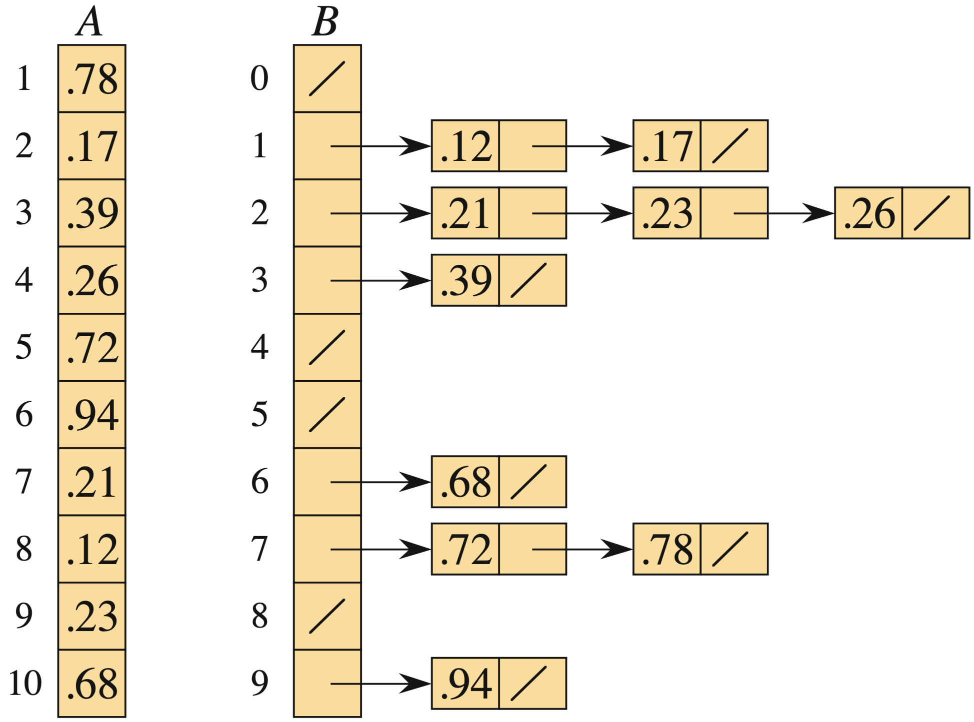
Since $d = 2$, radix-sort requires two passes of Counting sort that each take $O(n)$ time. This the total run-time for radix-sort for this type of input is $O(n)$.

Bucket sort

Assumption: input is drawn from the range $[0..1)$ with a uniform probability distribution.

Average case is $O(n)$.

```
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[floor(n A[i])]
    for i = 0 to n - 1
       sort B[i] using insertion sort
    concatenate the lists B[0], B[1],...,B[n-1] together in order
    return the concatenated lists
```



(a) (b)

If each list is of size $O(1)$, then the run-time is $O(n)$. The average case analysis requires advanced math so we will skip it here.

Notes: If the key values are uniformly distributed, we can still get $O(n)$ average case run time if we know the probability distribution. How can we take advantage of knowing the probability distribution to get all buckets to be of size $O(1)$ on the average?