Sorting in Linear Time

Lower Bounds for Sorting: Insertion-Sort, Merge-Sort, Heapsort, and Quicksort are comparisons sorts since all key movements are based on the outcomes of key comparisons. How fast can a comparison sort run on the average?

First we look at sorting just 3 keys. Sort3(x, y, z) rearranges x, y, and z so that $x \le y \le z$.

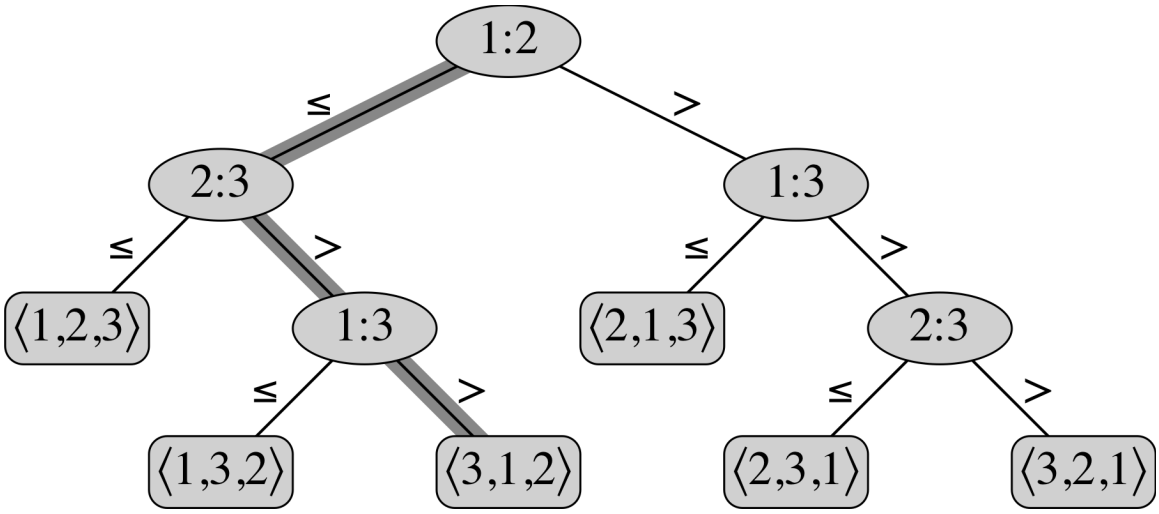
Sort3(x, y, z)

```
if x ≤ y
   then    if y ≤ z
              then    {do nothing, A is already sorted}
              else   if x ≤ z
                        then exchange y ↔ z
                        else exchange x ↔ y
                             exchange x ↔ z

   else   if x ≤ z
             then exchange x ↔ y
             else  if y ≤ z
                      then   exchange x ↔ z
                             exchange x ↔ y
                      else    exchange x ↔ z
```

The flow chart for the Sort3 comparisons is the decision tree. Sort3 enters at the root and exits at one of the leaves. The exit depends on the ordering of x, y, and z; for example, if $x \le y \le z$ then Sort3 exits at the leftmost leaf.

There are 6 leaves because 3 keys can be permuted 3! = 6 different ways. The paths to 2 of the leaves go through 2 comparisons and the paths to the other 4 leaves go through 3 comparisons. If each permutation is equally likely then Sort3 will perform (2*2+ 4*3)/6 = 16/6 = 2.667 comparisons on the average.



Is there another comparison sort algorithm for 3 keys that goes through less comparisons on the average? No. The flow chart for any such algorithm must be a binary tree with 6 leaves. Any 6-leaf binary decision tree has 5 comparison nodes. The minimum average path length is attained when the tree is balanced as much as possible with 1 node at the root, 2 nodes on the second level, and 2 nodes on the third level.

Any 3-key comparison sort algorithm must go through at least 16/6 comparisons on the average.

Now we look at sorting n keys with a comparison sort algorithm. There are n! permutations of the n keys so the binary decision tree for the algorithm has n! leaves.

To minimize the average path length, the tree should be balanced as much as possible. Let p=ceil(lg n!). The tree has p levels with $n!–2^{p-1}$ comparison nodes in the lowest level and all higher levels completely filled. The paths to $2n!–2^p$ leaves go through p comparisons and the paths to $2^p - n!$ leaves go through p-1 comparisons. The average path length is $p+1-2^p/n!$ comparisons. Stirling's approximation to n! is $(2\pi n)^{1/2}(n/e)^n$ so p = ceil(lg n!) is Θ(n lg n)

The average running time for a comparison sort algorithm on n keys is Ω(n lg n). Heapsort and Quicksort both reach this lower bound. Since Heapsort runs in Θ(n lg n) time at worst, the lower bound for the worst-case of any comparison sort is also Ω(n lg n)

Counting Sort

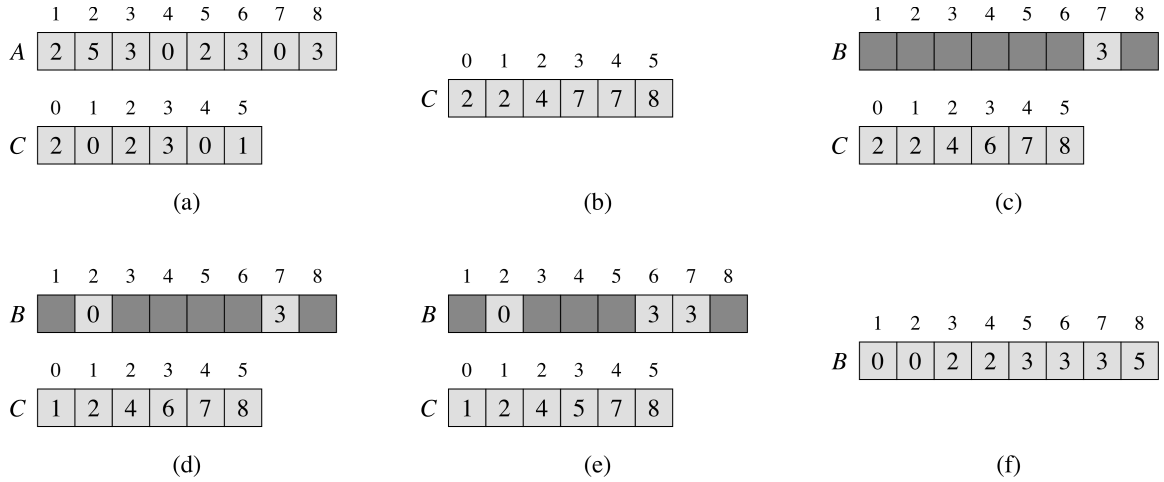
Counting sort assumes that the keys are integers in the range of 1 to k where k is O(n). Besides the input array, A[1 .. n], Counting sort uses two auxiliary arrays, B[1 .. n] to hold the sorted output and C[1 .. k] for temporary working storage.

```
COUNTNG-SORT(A, B, k)
1   let C[0 .. k] be a new array
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to A.length
5         C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i -1]
9   // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] – 1
```

Counting sort is **stable** (keys with same value appear in same order in output as they did in input) because of how the last loop works.

Insertion sort and Merge sort are stable while Heap-sort and quick-sort are not stable.

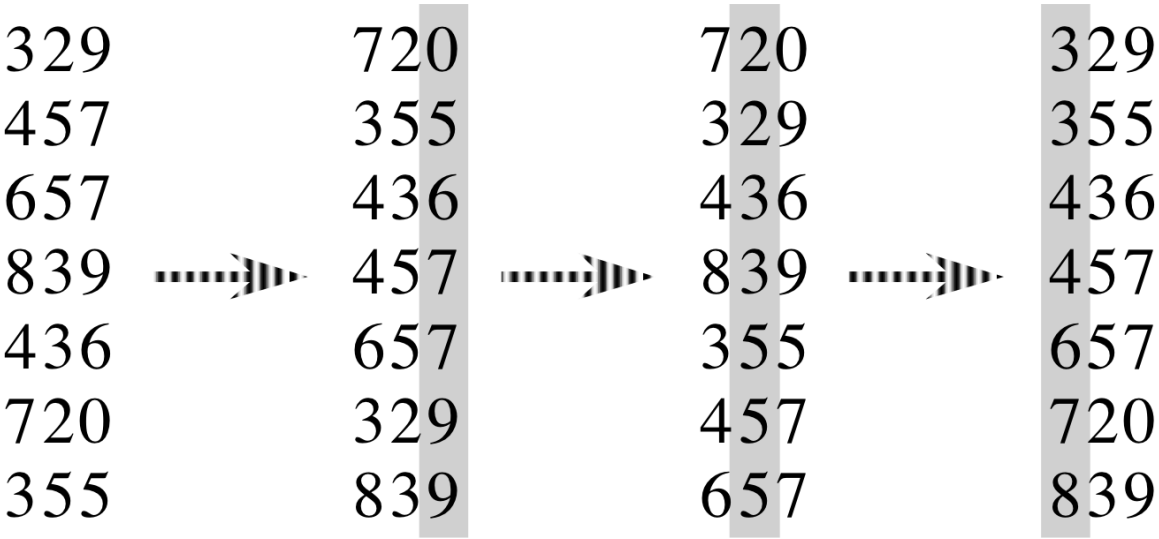
Analysis: $\Theta(n + k)$, which is $\Theta(n)$ if $k = O(n)$



Radix Sort

Suppose that the values to be sorted are written as *d-digit* numbers. Use some stable sort to sort them by last digit. Then stable sort them by the second least significant digit, then by the third, etc.

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



Correctness

- Induction on number of passes (i in pseudocode)
- Assume digits 1, 2, …, i-1 are sorted

- Show that a stable sort on digit i leaves digits 1, …, i sorted:
    - If 2 digits in position i are different, ordering by position i is correct, and positions 1, …, i-1 are irrelevant.
    - If 2 digits in position i are equal, numbers are already in the right order (by inductive hypothesis). The stable sort on digit i leaves them in the right order.

This argument shows why it's so important to use a stable sort for intermediate sort.

Analysis
Assume that we use counting sort as the intermediate sort.

- $\Theta(n + k)$ per pass (digits in range 0, …, k)
- *d* passes
- $\Theta(d(n + k))$ total
- If $k = O(n)$, time $= \Theta(dn) = \Theta(n)$

Bucket Sort

Assumes the input is generated by a random process that distributes elements uniformly over [0, 1).

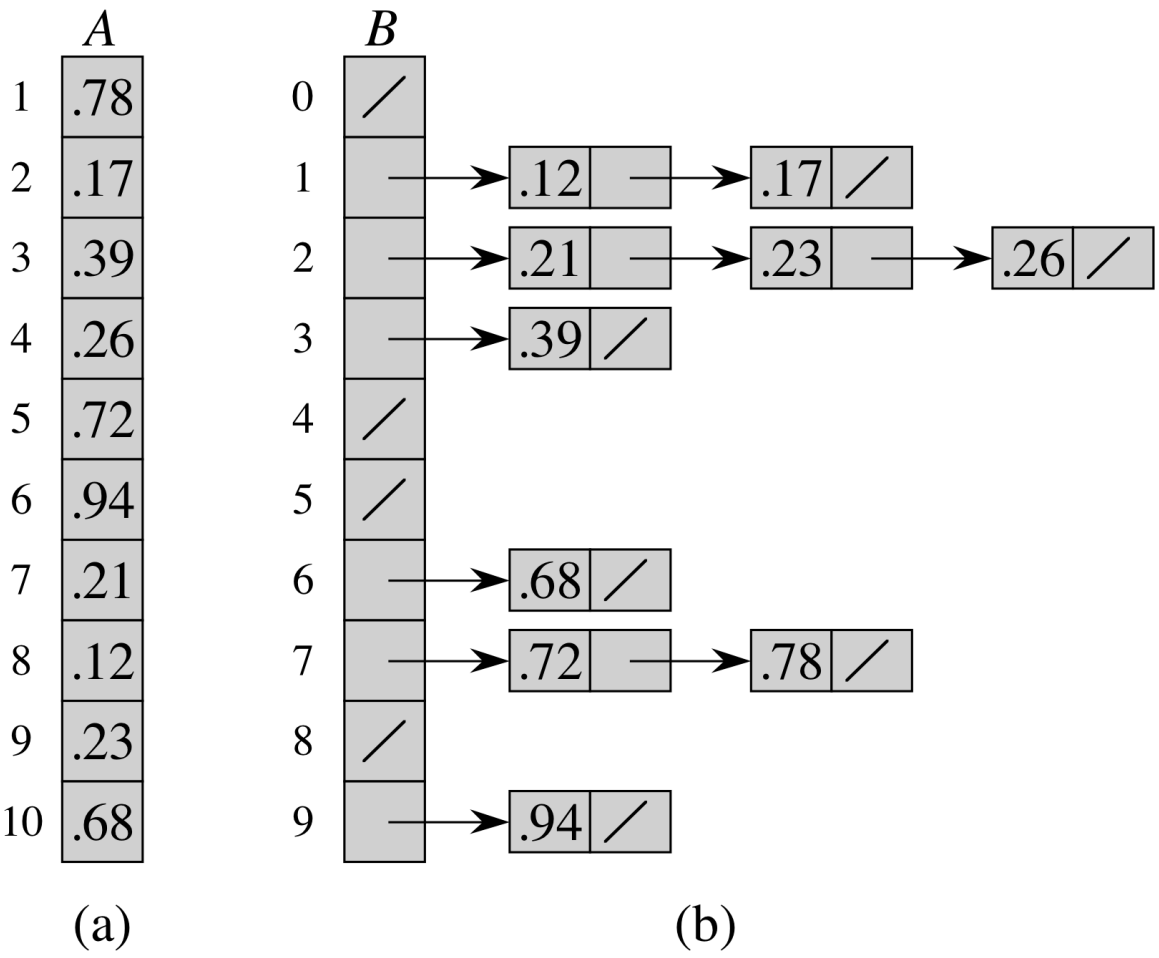
Idea:

- Divide [0, 1) into n equal-sized buckets.
- Distribute the n input values into the buckets.
- Sort each bucket.
- Then go through buckets in order, listing elements in each one.

Input: $A[1 .. n]$, where $0 \leq A[i] < 1$ for all i.
Auxiliary array: $B[0 .. n-1]$ of linked lists, each list initially empty.

BUCKET-SORT(A)

```
1   let B[0 .. n-1] be a new array
2   n = A.length
3   for i = 0 to n – 1
4       make B[i] an empty list
5   for i = 1 to n
6       insert A[i] into list B[⌊n.A[i]⌋]
7   for i = 0 to n -1
8       sort list B[i] with insertion sort
9   concatenate the lists B[0], B[1], …, B[n – 1] together in order
```



Correctness: Consider A[i], A[j]. Assume without loss of generality that A[i] ≤ A[j]. Then $\lfloor n.A[i] \rfloor \le \lfloor n.A[j] \rfloor$. So A[i] is placed into the same bucket as A[j] or into a bucket with a lower index.

- If same bucket, insertion sort fixes up.
- If earlier bucket, concatenation of lists fixes up.

Analysis:

- Relies on no bucket getting too many values.
- All lines of algorithm except insertion sorting take $\Theta(n)$ altogether.
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket implies $O(n)$ sort time for all buckets.
- We "expect" each bucket to have few elements, since the average is 1 element per bucket.
- Not a comparison sort. Used a function of key values to index into an array.
- We use probability to analyze an algorithm whose running time depends on the distribution of inputs.
- Different from a randomized algorithm, where we use randomization to impose a distribution.
- With bucket sort, if the input isn't drawn from a uniform distribution on [0, 1), all bets are off (performance-wise, but the algorithm is still correct).