CPSC 535: Advanced Algorithms

Instructor: Dr. Doina Bein

Importance of Sorting

https://users.cs.duke.edu/~reif/courses/alglectures/skiena.lectures/lecture4.1.pdf

Sorting problem

- The sorting problem's lower bound is Ω(n log n), which means that every algorithm that solves the sorting problem has time complexity O(n log n) or slower.
- ***Theorem****: The sorting problem has a lower bound of Ω(n log n) that applies to all comparison-based sorting algorithms.*
- But there are sorting algorithms with O(n) time complexity. How?
- If we sort without comparisons, i.e. without comparing entire elements with each other, but maybe parts of it, for example digits of it
- Counting sort, radix sort, and bucket sort, all have O(n)

(Chapter 8 of CLRS)

Counting Sort

(https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec5.pdf)

- No comparisons between elements
- Input: A[1..n], an array with n elements to be sorted such that the elements are drawn from A[j] $\in$ {1, 2, …, k}
  Output: B[1..b] (which is the sorted A)
  Auxiliary storage: C[1..k]
- Algorithm:

```
for i=1 to k do C[i] = 0
for j=1 to n do
  C[A[j]] = C[A[j]] + 1          // C[i] = |{ key == i }|
for i=2 to k do
  C[i]=C[i] + C[i-1]        // C[i] = |{ key ≤ i }|
for j=n downto 1 do
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j]] - 1
```

More on Counting Sort

- See the slides 13 through 27 at https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec5.pdf
- Time complexity: O(n+k); if k = O(n) then counting sort takes O(n). Why?
- Because counting sort is not a comparison-based sort. No comparison takes place between elements.
- Counting sort is a *stable* sort: it preserves the input order among equal elements. What other sorting algorithm is stable? Answer: mergesort.

Radix Sort

- Origin: Herman Hollerith's card sorting machine for US census in 1890.
- It is a digit by digit sort, sorting on *least significant digit first* using some auxiliary stable sort; Hollerith's idea was to sort starting with the most significant digit which is bad.
- Tow elements are not compared by their whole value, but by their digits
- Read the slides 32 through 35 at https://users.cs.duke.edu/~reif/courses/alglectures/demleis.lectures/lec5.pdf
- Radix sort has $O(d \cdot n)$ time where the values to be sorted are in the range $0..n^d-1$ (i.e. at most d digits)

Example

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Bucket Sort

- Assumes that the input is drawn from a uniform distribution over the interval [0,1), i.e. the input is generated by a random process that distributes elements uniformly and independently over the interval [0,1)
- **Input**: n numbers from the interval [0,1), uniformly distributed

  Output: sorted n values
- Bucket sort steps:

1. Divide the interval [0,1) into n equal-sized subintervals (or buckets)
2. Distribute each input value into the corresponding bucket
3. Simply sort the numbers in each bucket
4. Go through the buckets in order, listing the elements in each

Example

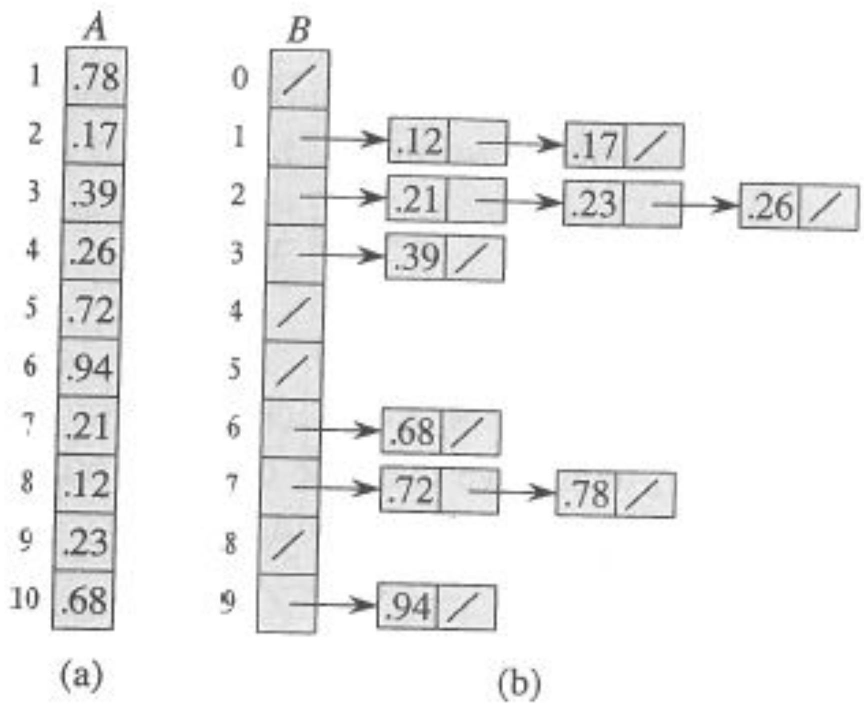

Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1\,..\,10]$. **(b)** The array $B[0\,..\,9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket Sort (contd.)

- We expect few numbers (i.e. a constant number) in each subinterval
- We use insertion sort to sort the elements in each subinterval, which is $O(1)$ expected
- Average-case is $\Theta(n)$ as long as the sum of the squares of the bucket sizes is linear in the total number of elements