CSE331 Introduction to Algorithm
Lecture 9: Can we Sort in Linear Time?

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 11, 2017

1. Introduction
2. Preliminary
3. Comparison-based sorting
4. Lower bound for sorting
5. Counting sort
6. Bucket sort

Introduction

- Last lecture on sorting.
- We have seen several sorting algorithms.
- The best ones run in $O(n \log n)$ time.
- Problem: Can we do better?
- This lecture shows that it is impossible, under a fairly general model of computation.
- We also give two algorithms that sort in linear time if we make some assumptions on the input.

- **Reference**: Section 8.1, 8.2 and 8.4 of the textbook Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.

Permutations

- Example: the permutations of $\{a, b, c\}$ are (a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a).

Definition

A *permutation* of a finite set S is an ordered sequence of all the elements of S, with each element appearing exactly once.

- From CSE232:

Proposition

Every set S with n elements has $n!$ permutations.

Binary Trees

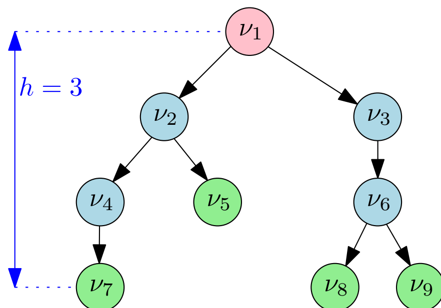


Figure: A binary tree rooted at ν_1, with 4 leaves $\nu_5, \nu_7, \nu_8, \nu_9$, with internal nodes $\nu_1, \nu_2, \nu_3, \nu_4, \nu_6$. The tree has $n = 9$ nodes and height $h = 3$.

- Still from CSE232:

Definition

A *binary tree* is a tree in which each node has at most two children.

Proposition

A binary tree with height h has at most 2^h *leaves.*

Comparison-Based Sorting

Definition

Comparison-based sorting algorithms only obtain information about the input array $[a_1, \ldots, a_n]$ by comparing pairs of input items. More precisely, these algorithms perform tests $a_i > a_j$, which return the answer YES or NO

- Examples: INSERTION SORT, MERGE SORT, QUICKSORT.
- Remark: Tests $a_i \leqslant a_j$ are equivalent, after exchanging the answers YES and NO.

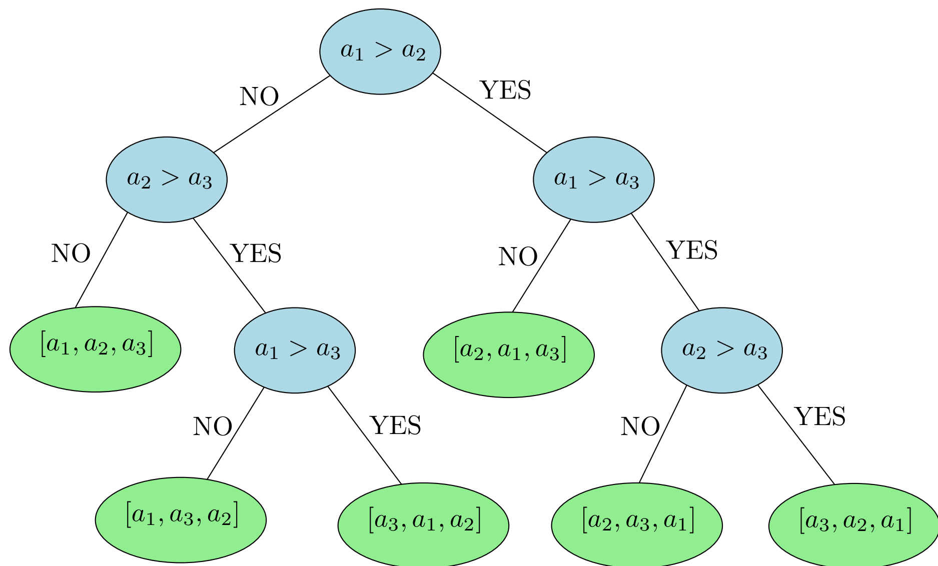
Comparison-Based Sorting

Insertion Sort

1: **procedure** INSERTION SORT($A[1 \dots n]$)
2: **for** $j \leftarrow 2, n$ **do**
3: $\text{key} \leftarrow A[j]$
4: $i \leftarrow j - 1$
5: **while** $i > 0$ and $A[i] > \text{key}$ **do**
6: $A[i+1] \leftarrow A[i]$
7: $i \leftarrow i - 1$
8: $A[i+1] \leftarrow \text{key}$

- INSERTION SORT only performs comparisons $A[i] > A[j]$.

Decision tree for Insertion sort on 3 elements



Lower Bound for Sorting

Theorem

Any comparison-based sorting requires $\Omega(n \log n)$ comparisons in the worst case.

Proof.

Any comparison-based sorting algorithm can be represented as a decision tree where each internal node is labeled by a test $a_i < a_j$, and each leaf contains exactly one permutation (see previous slide). In order for the algorithm to correctly sort the array, each permutation must appear in at least one leaf. So the tree has at least $n!$ leaves, and has height at least $\log(n!)$. As $\log(n!) = \Theta(n \log n)$, the height of the tree is $\Omega(n \log n)$. One of the leaves is at depth $\Omega(n \log n)$, so the algorithm needs to perform $\Omega(n \log n)$ comparisons in order to sort the corresponding permutation. □

Lower Bound for Sorting

- The lower bound holds for comparison-based sorting algorithms, which is a fairly general model.
- It is a *worst-case* lower bound, so it may not hold if some extra assumptions are made on the input.
- It may not hold if we use a more powerful model of computation.
- Next we present COUNTING SORT, which is faster if the range of input values is restricted.

Counting Sort

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Input A

0	1	2	3	4	5
2	0	2	0	3	1

Count C

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Input A

0	1	2	3	4	5
2	2	4	4	7	8

Cumulative count C

Counting Sort

Input A

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Output B

1	2	3	4	5	6	7	8

Auxiliary array C

0	1	2	3	4	5
2	2	4	4	7	8

Counting Sort

Input A

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Output B

1	2	3	4	5	6	7	8
						4	

Auxiliary array C

0	1	2	3	4	5
2	2	4	4	7	8

Counting Sort

Input A

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Output B

1	2	3	4	5	6	7	8
						4	

Auxiliary array C

0	1	2	3	4	5
2	2	4	4	6	8

Counting Sort

Input A

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Output B

1	2	3	4	5	6	7	8
	0					4	

Auxiliary array C

0	1	2	3	4	5
2	2	4	4	6	8

Counting Sort

Input A

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Output B

1	2	3	4	5	6	7	8
	0				4	4	

Auxiliary array C

0	1	2	3	4	5
1	2	4	4	6	8

Counting Sort

Input A

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Output B

1	2	3	4	5	6	7	8
	0		2		4	4	

Auxiliary array C

0	1	2	3	4	5
1	2	4	4	5	8

Counting Sort

	1	2	3	4	5	6	7	8
Input A	2	5	4	0	2	4	0	4

	1	2	3	4	5	6	7	8
Output B	0	0		2		4	4	

	0	1	2	3	4	5
Auxiliary array C	1	2	3	4	5	8

Counting Sort

	1	2	3	4	5	6	7	8
Input A	2	5	4	0	2	4	0	4

	1	2	3	4	5	6	7	8
Output B	0	0		2	4	4	4	

	0	1	2	3	4	5
Auxiliary array C	0	2	3	4	5	8

Counting Sort

Input A

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Output B

1	2	3	4	5	6	7	8
0	0		2	4	4	4	5

Auxiliary array C

0	1	2	3	4	5
0	2	3	4	4	8

Counting Sort

Input A

1	2	3	4	5	6	7	8
2	5	4	0	2	4	0	4

Output B

1	2	3	4	5	6	7	8
0	0	2	2	4	4	4	5

Auxiliary array C

0	1	2	3	4	5
0	2	3	4	4	7

Counting Sort

- We assume all keys are in $\{0, \dots, k\}$ and B is the output array.

Pseudocode

```
1: procedure COUNTING SORT(A[1...n], B[1...n], k)
2:     C[0...k] ← new array
3:     for i ← 0, k do
4:         C[i] ← 0
5:     for j ← 1, n do                         ▷ Record in C[i] the
6:         C[A[j]] ← C[A[j]] + 1               ▷ number of keys equal to i.
7:     for i ← 1, k do                         ▷ Record in C[i] the
8:         C[i] ← C[i] + C[i − 1]              ▷ number of keys ⩽ i.
9:     for j ← n downto 1 do
10:        B[C[A[j]]] ← A[j]
11:        C[A[j]] ← C[A[j]] − 1
```

Counting Sort

Theorem

If all the keys are in $\{0, \dots, k\}$, then an array of size n is sorted in $\Theta(n+k)$ time by COUNTING SORT.

- In particular, if $k = O(n)$, it runs in $\Theta(n)$ time.
- It does not contradict our lower bound because:
  - Counting sort is not comparison based. In fact, it does *not* compare keys.
  - Instead, it uses keys as indices in the auxiliary array C.
- In addition, it is slow when k is large, for instance when $k = \Omega(n^2)$ it runs in $\Omega(n^2)$ time.

Counting Sort

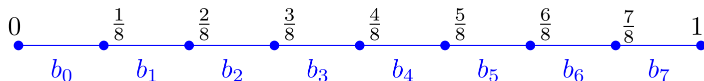
- The last loop counts from n down to 1.
- The algorithm is still correct if the loop goes from 1 to n.
- However, the algorithm would not be *stable* anymore:

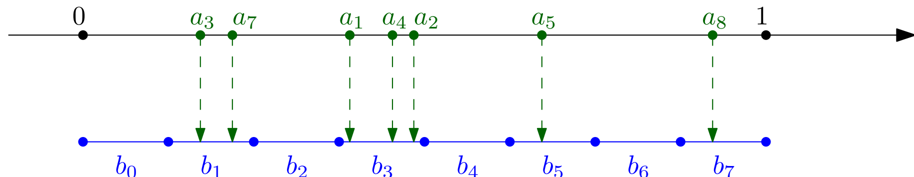
Definition

A sorting algorithm is *stable* if every two records with the same key appear in the same order in the input array and the output array.

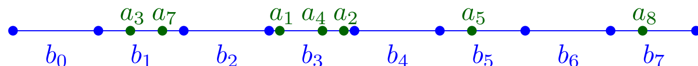
- This is not very interesting when we just sort numbers, but if we sort, say, a list of people according to their age, then the order of their other attributes (ID number ...) will not be modified.
- COUNTING SORT, MERGE SORT and INSERTION SORT are stable, but not QUICKSORT.

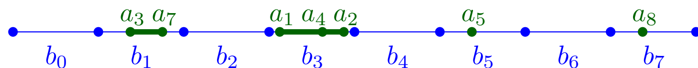
Bucket Sort



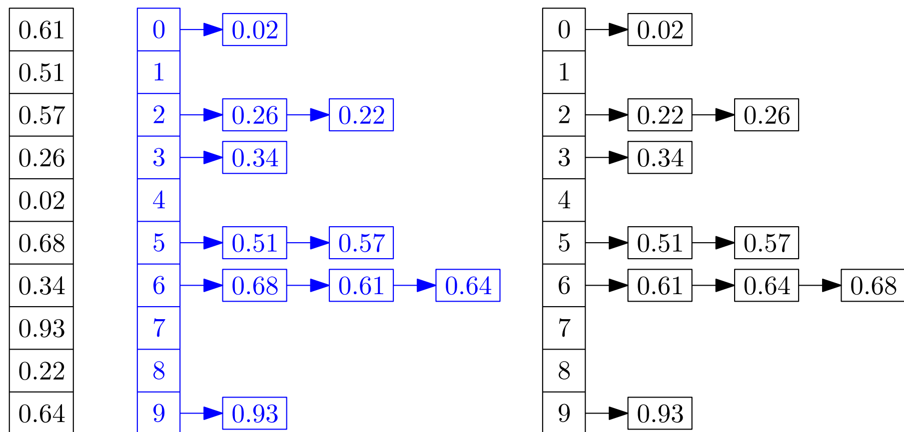


Bucket Sort





Bucket Sort



Bucket Sort

- INPUT: $a_1, \ldots, a_n \in [0, 1)$
  - It means $0 \leqslant a_i < 1$ for all i.
- We create n *buckets* $b_j = \left[\frac{j}{n}, \frac{j+1}{n}\right)$, $j = 0, \ldots n-1$.
- Each a_i is placed in the bucket b_j such that $a_i \in b_j$.
- We sort each bucket separately using insertion sort.
- Finally we concatenate the sorted lists corresponding to buckets $b_0, \ldots b_{n-1}$.

Bucket Sort

Pseudocode

```
1: procedure BUCKET SORT(A[1...n])
2:     B[0...n−1] ← new array
3:     for j ← 0, n−1 do
4:         B[j] ← empty list
5:     for i ← 1, n do
6:         j ← ⌊n.A[i]⌋
7:         insert A[i] into B[j]
8:     for j ← 0, n−1 do
9:         INSERTION SORT(B[j])
10:    return B[0].B[1]...B[n−1]          ▷ Concatenation of sorted lists
```

Bucket Sort

- At line 7, we insert $A[i]$ at the end of $B[j]$ if we need the algorithm to be stable. It can be done in $O(1)$ time by keeping a pointer to the tail of the list.
- If we don't count the calls to INSERTION SORT, it is clear that BUCKET SORT takes $\Theta(n)$ time.
- Worst case: If all $A[i]$ fall in the same bucket, $\Omega(n^2)$ time.
- Best case: If each bucket contains one input number, $O(n)$ time.
- Average case?
  - BUCKET SORT is deterministic, so we will determine the expected running time $\mathbb{E}[T(n)]$ over a distribution of input numbers.
  - We will assume that the input numbers are chosen *uniformly and independently at random* in $[0, 1)$.

Bucket Sort: Analysis

- Let n_j denote the number of input numbers a_i that fall in bucket b_j:

$$n_j = |\{a_i \mid a_i \in b_j\}|$$

Proposition

Bucket Sort *runs in* $\Theta(n + \sum_j n_j^2)$ *time.*

- By linearity of expectation, it implies that the expected running time is

$$\mathbb{E}\left[T(n)\right] = \Theta\left(n + \sum_{j=1}^{n} \mathbb{E}\left[n_j^2\right]\right)$$

- So we need to determine $\mathbb{E}\left[n_j^2\right]$.

Bucket Sort: Analysis

- Let X_i be the following indicator random variable:

$$X_i = \begin{cases} 0 & \text{if } a_i \notin b_0 \\ 1 & \text{if } a_i \in b_0 \end{cases}$$

- Then $n_0 = \sum_{i=1}^{n} X_i$, and

$$n_0^2 = \sum_{i=1}^{n} \sum_{j=1}^{n} X_i X_j = \sum_{i=1}^{n} X_i^2 + \sum_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant n \\ i \neq j}} X_i X_j$$

- By linearity of expectation, it follows that

$$\mathbb{E}\left[n_0^2\right] = \sum_{i=1}^{n} \mathbb{E}\left[X_i^2\right] + \sum_{i \neq j} \mathbb{E}\left[X_i X_j\right]$$

Bucket Sort: Analysis

- X_i^2 only takes values 0 or 1, so

$$\mathbb{E}\left[X_i^2\right] = 1.\Pr[X_i^2 = 1] + 0.\Pr[X_i^2 = 0]$$
$$= \Pr[X_i^2 = 1]$$
$$= \frac{1}{n}$$

because a_i is chosen uniformly at random in $[0, 1)$.

Bucket Sort: Analysis

- Suppose $i \neq j$.
- $X_i X_j$ only takes values 0 or 1, so

$$\begin{aligned}\mathbb{E}\left[X_i X_j\right] &= 1.\Pr[X_i X_j = 1] + 0.\Pr[X_i X_j = 0] \\ &= \Pr[X_i X_j = 1]\end{aligned}$$

- $\Pr[X_i X_j = 1]$ is the probability that a_i and a_j fall in b_0.
- It is $1/n^2$ because a_i and a_j are chosen independently, so

$$\mathbb{E}\left[X_i X_j\right] = \frac{1}{n^2}$$

Bucket Sort: Analysis

- It follows that

$$\begin{aligned}
\mathbb{E}\left[n_0^2\right] &= \sum_{i=1}^{n} \mathbb{E}\left[X_i^2\right] + \sum_{i \neq j} \mathbb{E}\left[X_i X_j\right] \\
&= \sum_{i=1}^{n} \frac{1}{n} + \sum_{i \neq j} \frac{1}{n^2} \\
&= 1 + \frac{n(n-1)}{n^2} \\
&= 2 - \frac{1}{n}
\end{aligned}$$

- As n_0 plays no special role, we also have

$$\mathbb{E}\left[n_j^2\right] = 2 - \frac{1}{n} \quad \text{for all } 1 \leqslant j \leqslant n$$

Bucket Sort: Analysis

- We just proved $1 \leqslant \mathbb{E}\left[n_j^2\right] < 2$, so it follows from the analysis on Slide 31 that:

Theorem

The expected running time of BUCKET SORT *is* $\Theta(n)$ *when the* n *input numbers are chosen uniformly and independently at random in* $[0, 1)$.

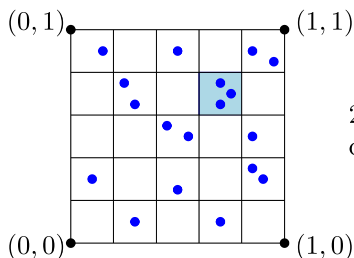
- So BUCKET SORT is very efficient if the input numbers are distributed uniformly at random.
- But in the worst case, for instance if the distribution is skewed and many input numbers fall in the same bucket, it runs in quadratic time.
- So it does not contradict our lower bound.

Bucket Sort: Analysis

- Here "Expected" running time has a very different meaning from our analysis of QUICKSORT:
- QUICKSORT is very efficient on worst-case input. It can only be slow if we are extremely unlucky with the random choices of pivot. In practice it never happens.
- On the other hand BUCKET SORT performs poorly on worst case input. In practice it happens, because data is often skewed.

Bucket Sort: Concluding Remarks

- The approach of BUCKET SORT is called *bucketing*.



2D bucketing. The blue bucket contains 3 input points.

- It also applies to multidimensional data, using a uniform grid for instance.
- Example: Fixed radius near-neighbor searching. See David Mount's computational geometry course, Lecture 33.

Bucket Sort: Concluding Remarks

- Problem: How can we apply BUCKET SORT if the input numbers are not in $[0, 1)$?