CSE331 Introduction to Algorithm
Lecture 15: Can we Sort in Linear Time?

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 23, 2021

1 Introduction

2 Preliminary

3 Comparison-based sorting

4 Lower bound for sorting

5 Counting sort

Introduction

- 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 an algorithm that sorts in linear time if we make some assumptions on the input.
- **Reference**: Section 8.1 and 8.2 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: Discrete Mathematics:

Proposition

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

Binary Trees

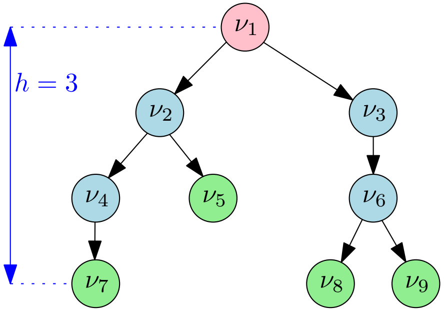


Figure: A binary tree rooted at ν_1, with 4 leaves $\nu_5, \nu_7, \nu_8, \nu_9$, with internal nodes $\nu_1, \nu_2, \nu_3, \nu_4, \nu_6$. The tree has $n = 9$ nodes and height $h = 3$.

- 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 and QUICKSORT.

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

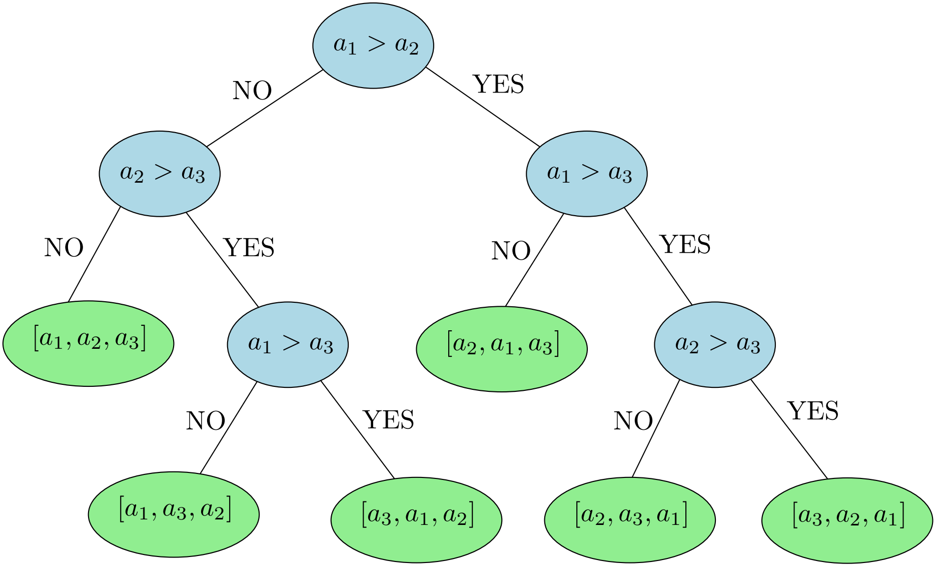
Comparison-Based Sorting

INSERTION SORT

```
procedure INSERTIONSORT($A[1 \dots n]$)
    for $j \leftarrow 2, n$ do
        key $\leftarrow A[j]$
        $i \leftarrow j - 1$
        while $i > 0$ and $A[i] > \text{key}$ do
            $A[i+1] \leftarrow A[i]$
            $i \leftarrow i - 1$
        $A[i+1] \leftarrow$ 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 algorithm 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. $\square$

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

	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	1	2	3	4	5
Auxiliary array C	2	2	4	4	7	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							4	

	0	1	2	3	4	5
Auxiliary array C	2	2	4	4	7	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							4	

	0	1	2	3	4	5
Auxiliary array C	2	2	4	4	6	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					4	

	0	1	2	3	4	5
Auxiliary array C	2	2	4	4	6	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				4	4	

	0	1	2	3	4	5
Auxiliary array C	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

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	

Auxiliary array C

0	1	2	3	4	5
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

	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	2	4	4	4	5

	0	1	2	3	4	5
Auxiliary array C	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 CountingSort($A[1 \dots n], B[1 \dots n], k$)
2:     $C[0 \dots k] \leftarrow$ new array
3:     for $i \leftarrow 0, k$ do
4:         $C[i] \leftarrow 0$
5:     for $j \leftarrow 1, n$ do                    ▷ record # keys equal to $i$ in $C[i]$
6:         $C[A[j]] \leftarrow C[A[j]] + 1$
7:     for $i \leftarrow 1, k$ do                    ▷ record # keys $\leqslant i$ in $C[i]$
8:         $C[i] \leftarrow C[i] + C[i-1]$
9:     for $j \leftarrow n$ downto 1 do              ▷ copy $A[j]$ to the right position
10:        $B[C[A[j]]] \leftarrow A[j]$
11:        $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting Sort

Theorem

If all the keys are in $\{0, \ldots, k\}$, 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.

Stable Sorting

year	name
2016	Alice
2014	Bob
2015	Carol
2015	David
2016	Emily
2016	Fred
2015	Gary

Input

year	name
2014	Bob
2015	Carol
2015	David
2015	Gary
2016	Alice
2016	Emily
2016	Fred

Stable sorting

year	name
2014	Bob
2015	David
2015	Carol
2015	Gary
2016	Emily
2016	Fred
2016	Alice

Unstable sorting

- COUNTING SORT, MERGE SORT and INSERTION SORT are stable, but not QUICKSORT.