Comparison Lower Bound of Sorting and Non-Comparison Sorting

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

We already know that the sorting problem on n elements can be settled in $O(n \log n)$ time. In this lecture, we will prove that this is the best time complexity possible in an important class of algorithms known as the comparison-based algorithms. Specifically, we will prove that all such algorithms must incur $\Omega(n \log n)$ time.

As the second step, we will partially circumvent this obstacle by presenting an algorithm that beats the $\Omega(n \log n)$ bound when the data domain has a small size.

Given an array A with length n, there are $n!$ different permutations of the elements therein.

Example

If $n = 3$, then there are 6 permutations:

$$A[1], A[2], A[3]$$
$$A[1], A[3], A[2]$$
$$A[2], A[1], A[3]$$
$$A[2], A[3], A[1]$$
$$A[3], A[1], A[2]$$
$$A[3], A[2], A[1]$$

The goal of the sorting problem is essentially to decide which of the $n!$ permutations corresponds to the final sorted order.

Comparison-Based Algorithm

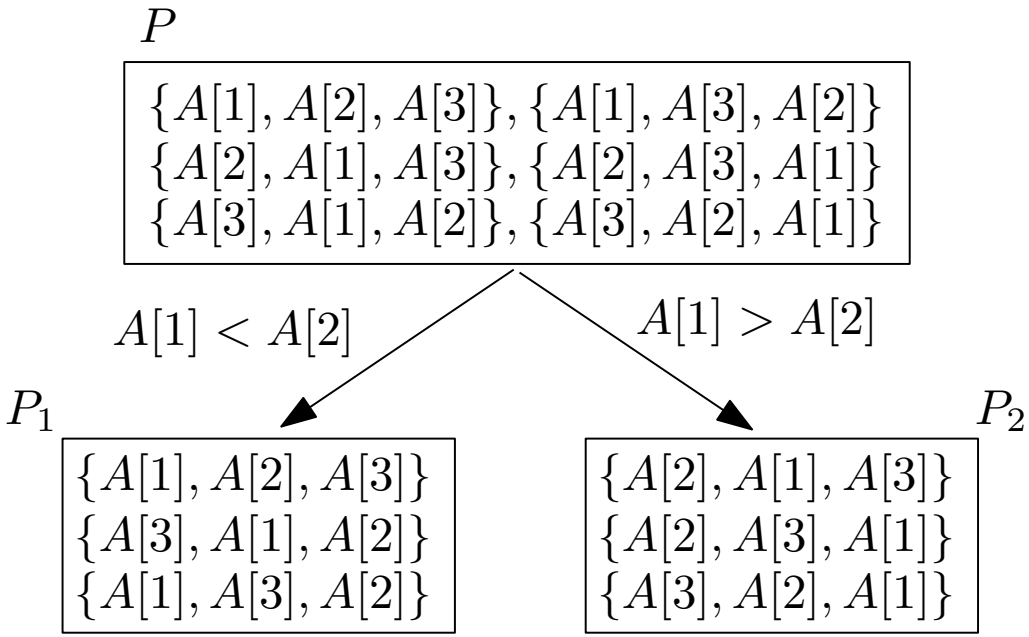
Intuitively, this is an algorithm that uses only comparisons to infer the ordering of the elements in A.

Formally, such an algorithm works by continuously shrinking a pool P of possible permutations.

- At the beginning, P contains all the $n!$ permutations.
- Every comparison allows the algorithm to discard all those permutations in P that are inconsistent with the comparison's result.
- Eventually, P has only 1 permutation left, which is thus the final sorted order.

In other words, at any moment, all the permutations that remain in P are possible results. The algorithm cannot terminate as long as $|P| \geq 2$.

Shrinking the Pool: An Example



In general, each comparison allows us to shrink P to either P_1 or P_2.

Comparison-Based Algorithm: The Framework

Framework

1. P = all the $n!$ permutations of A
2. **while** $|P| > 1$
3. make a comparison between elements e_1 and e_2
4. **if** $e_1 < e_2$ **then**
5. $P = P_1$, where P_1 is the set of permutations in P consistent with $e_1 < e_2$
6. **else**
7. $P = P_2$, where P_2 is the set of permutations in P consistent with $e_1 > e_2$
8. **return** the permutation in P

Various algorithms differ in how they implement Step 3.

A Worst-Case Lower Bound

- Note that one of P_1 and P_2 contains at least half of the permutations in P (i.e., either $|P_1| \geq |P|/2$ or $|P_2| \geq |P|/2$).
- The worst case happens when P always shrinks to the larger set between P_1 and P_2.
- In this case, the size of P shrinks by at most half after each comparison.
- Hence, the number of comparisons required before $|P|$ decreases to 1 is $\log_2(n!)$.

The next slide shows $\log_2(n!) = \Omega(n \log n)$.

A Worst-Case Lower Bound

$$\begin{aligned}
\log_2(n!) &= \sum_{i=1}^{n} \log_2 i \\
&\geq \sum_{i=n/2}^{n} \log_2 i \\
&\geq (n/2)\log_2(n/2) \\
&= \Omega(n \log n).
\end{aligned}$$

We now conclude that any comparison-based algorithm must incur $\Omega(n \log n)$ time sorting n elements in the worst case.

It is important to keep in mind that the above lower bound argument breaks apart as soon as we move away from the comparison-based class. Indeed, currently, the world's fastest deterministic algorithm for sorting n integers guarantees $O(n \log \log n)$ time (but in a variant of the RAM model that is slightly more powerful than the one we have been using). While that algorithm is too advanced for this course, next we will gain some idea about what a non-comparison-based algorithm can be, and how it can do better than $\Omega(n \log n)$.

Let us slightly re-define the sorting problem.

The Sorting Problem (in A Small Domain)

Problem Input:

A set S of n integers is given in an array of length n. Every integer is in the range of $[1, U]$, where the integer U is inside the CPU.

- The fact that S is a set (i.e., all the integers therein are distinct) implies that $U \geq n$.

Goal:

Design an algorithm to store S in an array where the elements have been arranged in ascending order.

Counting Sort

Step 1: Let A be the array storing S. Create an array B of length U. Initialize B by setting all its cells to 0.

Step 2: Carry out the following for every $i \in [1, n]$:

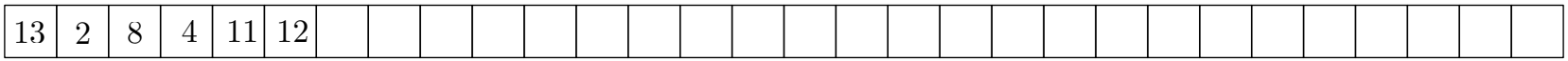
- Set x to the value of $A[i]$.
- Set $B[x] = 1$.

Step 3: Now we will generate the sorted order in A with one more scan of B:

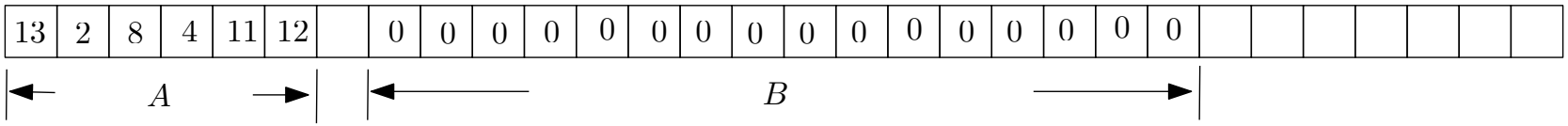
- Clear all the elements in A.
- Carry out the following for every $x \in [1, U]$:
  - If $B[x] = 0$, do nothing; otherwise, append integer x to A.

Example

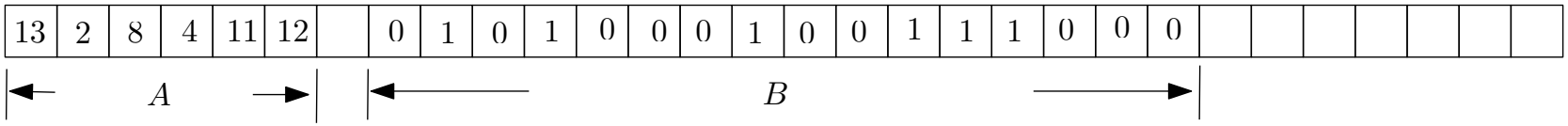
At the beginning



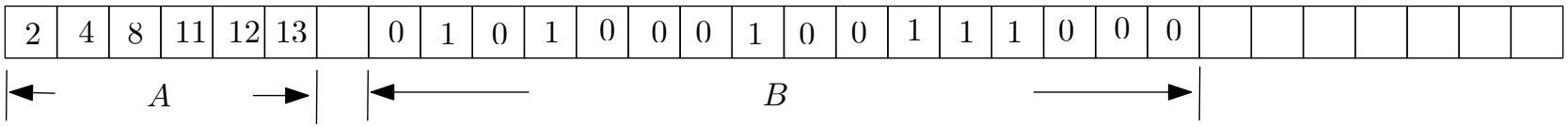
Initialize array B



Setting n cells of B to 1



Final sorted list



Analysis of Counting Sort

Steps 1 and 3 take $O(U)$ time, while Step 2 $O(n)$ time.

Therefore, the overall running time of counting sort is $O(n+U) = O(U)$.

For small $U = O(n)$ (e.g., $1000n$), the counting sort runs in $O(n)$ time.

Observe how counting sort gets around being "comparison-based"—in fact, it does not compare any pair of elements! Instead, it figures out the element ordering using ideas drastically different from doing comparisons: that is, by way of array B.

It is important to remember that counting sort does not improve merge sort! In fact, its time complexity $O(n + U)$ is incomparable to the time complexity $O(n \log n)$ of merge sort. For example, when $U = O(n)$, counting sort is faster, but when $U = \Omega(n^2)$, merge sort is faster.