Combinatorics II

Kon Yi

February 24, 2026

Abstract

Lecture note of Combinatorics II.

Contents

Chapter 1

Sorting Algorithm

Lecture 1

24 Feb.

Problem 1.0.1. Given n distinct numbers $a_1, a_2, \ldots, a_n$, sort them in increasing order.

- Input: n distinct numbers $a_1, a_2, \ldots, a_n$.
- Output: a permutation $\pi \in S_n$ s.t.

$$a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}.$$

Observation. Given x, y, is $x < y$?

Remark 1.0.1. When analyzing an algorithm, we consider the worst-case input and count operations, where operations are defined according to context.

Algorithm 1.1: Brute force

Go through all $\pi \in S_n$ one by one and test each permutation to see if it is correct.

Remark 1.0.2. For running time, testing a permutation takes $\leq n-1$ comparisons, and there are $n!$ possible permutations, so

$$\# \text{ of comparisons} \leq n!(n-1).$$

Question. What is efficient? How does the running time scale as the input size n increases? Ideally, the running time should be polynomial $O(n^c)$ for some $c \in \mathbb{R}$.

Algorithm 1.2: Insertion Sort

Maintain a sorted partial list, add one element at a time until the entire list is sorted.

Algorithm 1.3: Binary insertion sort

`// As before, we maintain a sorted prefix.`

Insert a_i into sorted list $a_1 < a_2 < \cdots < a_{i-2} < a_{i-1}$. Compare a_i to the middle element. If similar, insert into the first half. If bigger, insert into the second half. `// Either way reduce number of possible positions by half each time`

Remark 1.0.3. Thus, we can insert in $\approx \log_2 i$ many comparisons each time, and thus in total we need $O(n \log_2 n)$ comparisons.

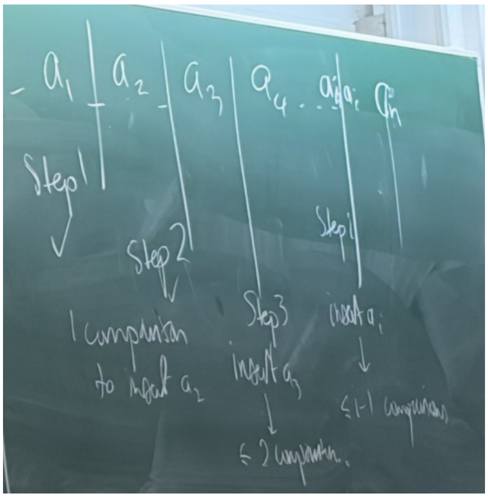
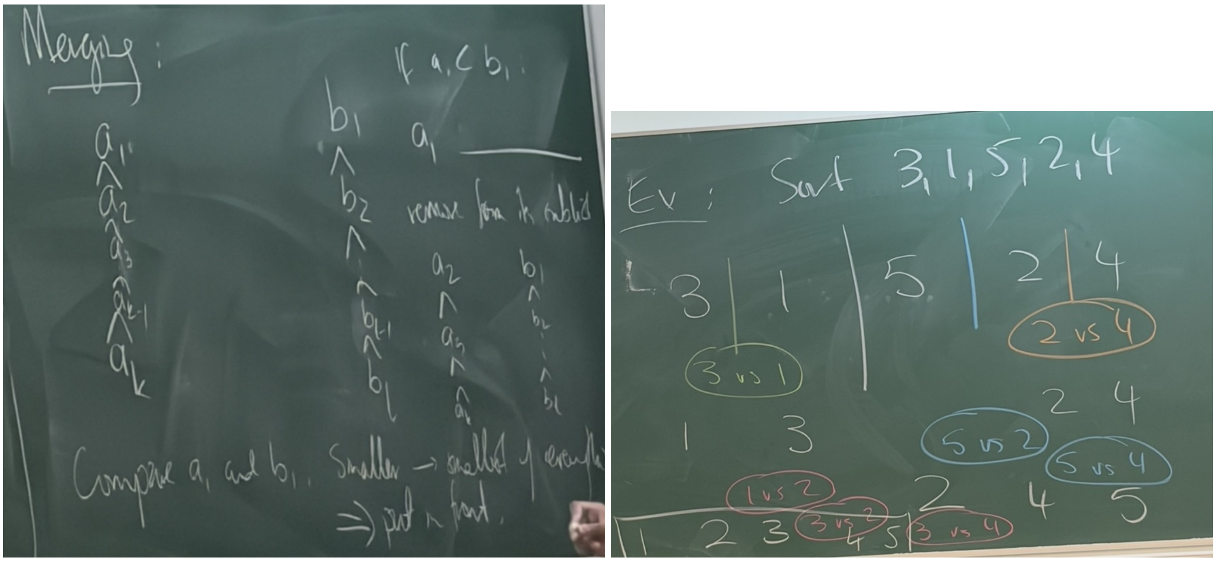


Figure 1.1: Insertion sort

Algorithm 1.4: Merge Sort

```
// Recursive algorithm
```
Split the list into two equal halves.
Sort each half.
Merge the two sorted lists into one. `// Compare the head of two list, remove the smaller one from original list then put it in the back of a new list, which is empty initially, and repeat this comparison step.`

Remark 1.0.4. Number of comparison to merge $\leq n - 1$.



(a) Merging

(b) An example of merge sort

Remark 1.0.5. Running time: Let $T(n)$ be the worst case running time for merge sort on a list of

n numbers. Then $T(1) = 0$, and we know

$$T(n) \leq \underbrace{T\left(\lfloor\frac{n}{2}\rfloor\right)}_{\substack{\text{sorting the}\\\text{first half}}} + \underbrace{T\left(\lceil\frac{n}{2}\rceil\right)}_{\substack{\text{sorting the}\\\text{second half}}} + \underbrace{n-1}_{\text{merge}}.$$

Now suppose $n = 2^k$, $k \in \mathbb{N}$, then

$$\begin{cases} T(1) = 0 \\ T\left(2^k\right) = 2T\left(2^{k-1}\right) + 2^k - 1 \text{ for } k \geq 1. \end{cases}$$

Thus,

$$\begin{aligned} T\left(2^k\right) &= 2T\left(2^{k-1}\right) + 2^k - 1 \\ &= 2\left[2T\left(2^{k-2}\right) + 2^{k-1} - 1\right] + 2^k - 1 \\ &= \cdots = 2^i T\left(2^{k-i}\right) + i2^k - \left(2^i - 1\right), \end{aligned}$$

and if we let $i = k$, then

$$T\left(2^k\right) = 2^k T(1) + k2^k - 2^k + 1 = k2^k - 2^k + 1 = n\log_2 n - n + 1.$$

Thus, $T(n) \leq n\log_2 n - n + 1$ when $n = 2^k$.

Question (Lower bounds). Can we do better?

Answer. No, but to prove this, we need to show that any other algorithm, no matter how weird, takes as long. ⊛

Question. Suppose we have an algorithm A for sorting n numbers. Now can we lower-bound its runtime?

Information Theory approach

Each comparison gives ≤ 1 bit of information. At the end, we get the information of the permutation to sort the numbers, and since there are $n!$ possible permutations, so the number of time needed is $\geq \log_2 n!$.

Appendix