Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Sorting Algorithms

- In week #1,

  we have seen two different sorting algorithms.

  - The ***Insertion Sort algorithm*** - $O(n^2)$ running time
  - The ***Merge Sort algorithm*** - $O(n \log n)$ running time

- A very natural question to ask is
  - Can it be done *even faster* ?

An $\Omega(n \log n)$-Time Lower Bound for Comparison-based Sorting Algorithms

$O(n \log n)$ is the best we can do,
unless further information (assumption) about the data is given.

Comparison-Based Sorting Algorithms

- A *comparison-based sorting algorithm* sorts the given set of data using only "***comparisons***" between *the elements of the data*.
  - Here, a comparison between any elements (cells) a and b refers to the following question

    " ***Should*** $\boldsymbol{a}$ ***be placed before*** $\boldsymbol{b}$ in the final sorted order ? "

  - No further assumption / operation on the input data set is required for the sorting procedure.

Comparison-Based Sorting Algorithms

- A *comparison-based sorting algorithm* sorts the given set of data using only "***comparisons***" between *the elements of the data*.
  - No further assumption / operation on the input data set is required for the sorting procedure.

- For example,
  - Insertion Sort / Bubble Sort / Selection Sort
  - ***Merge Sort*** / ***Quick Sort*** / *Heap Sort*

  are all such sorting algorithms.

$\Omega(n \log n)$-Lower-bound on Time Complexity

- We have the following theorem for comparison-based sorting algorithms.

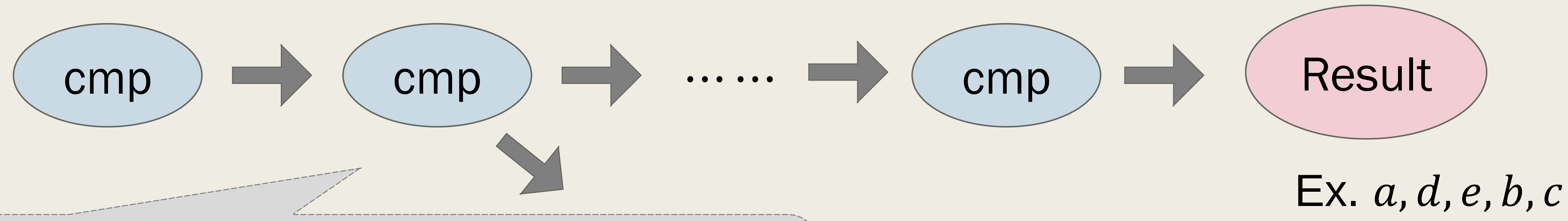
> **<u>Theorem.</u>**
>
> Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case for sorting n given numbers.

- How can we prove such a statement?
  - Just because we do not know how to do it doesn't mean that it doesn't exist, right?

Theorem.

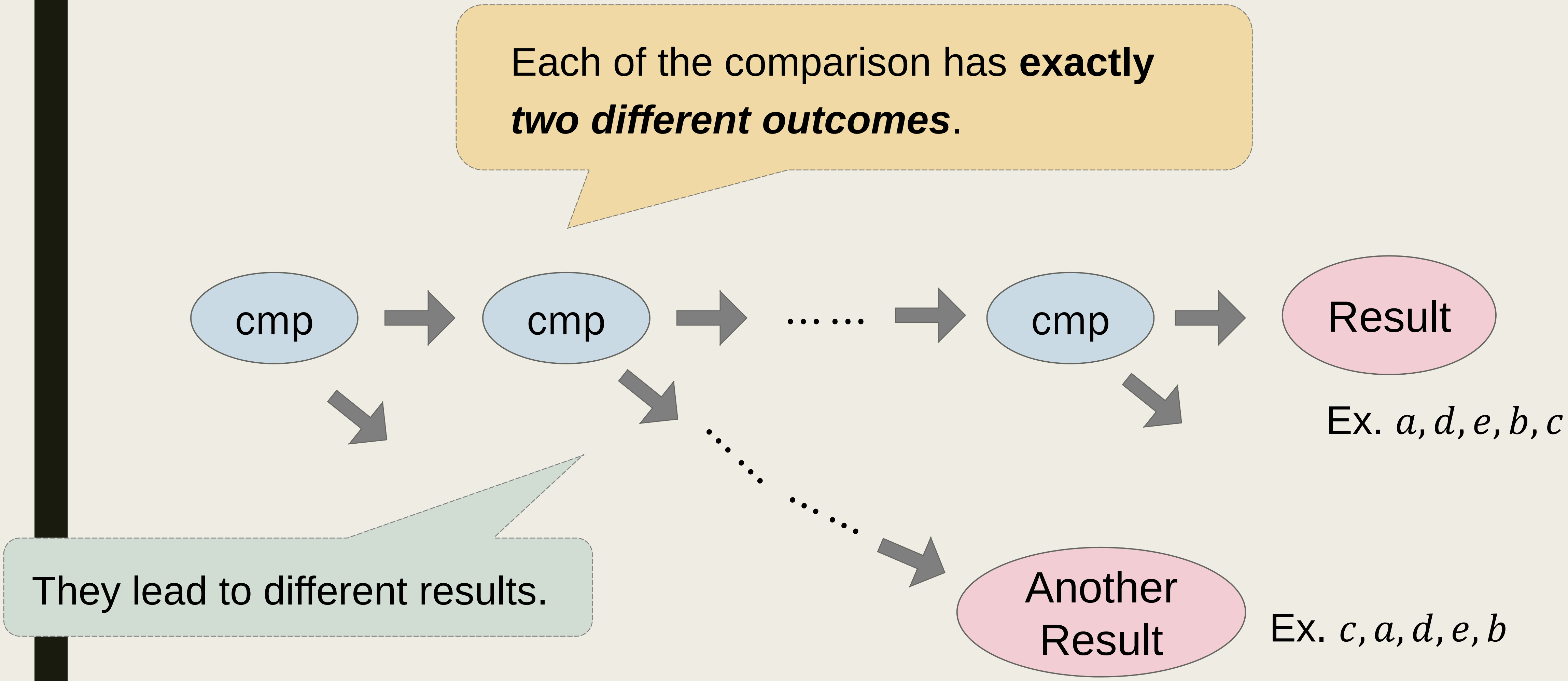
Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case for sorting n given numbers.

- The execution of *any comparison-based sorting algorithm* consists of a sequence of comparisons between the given numbers.

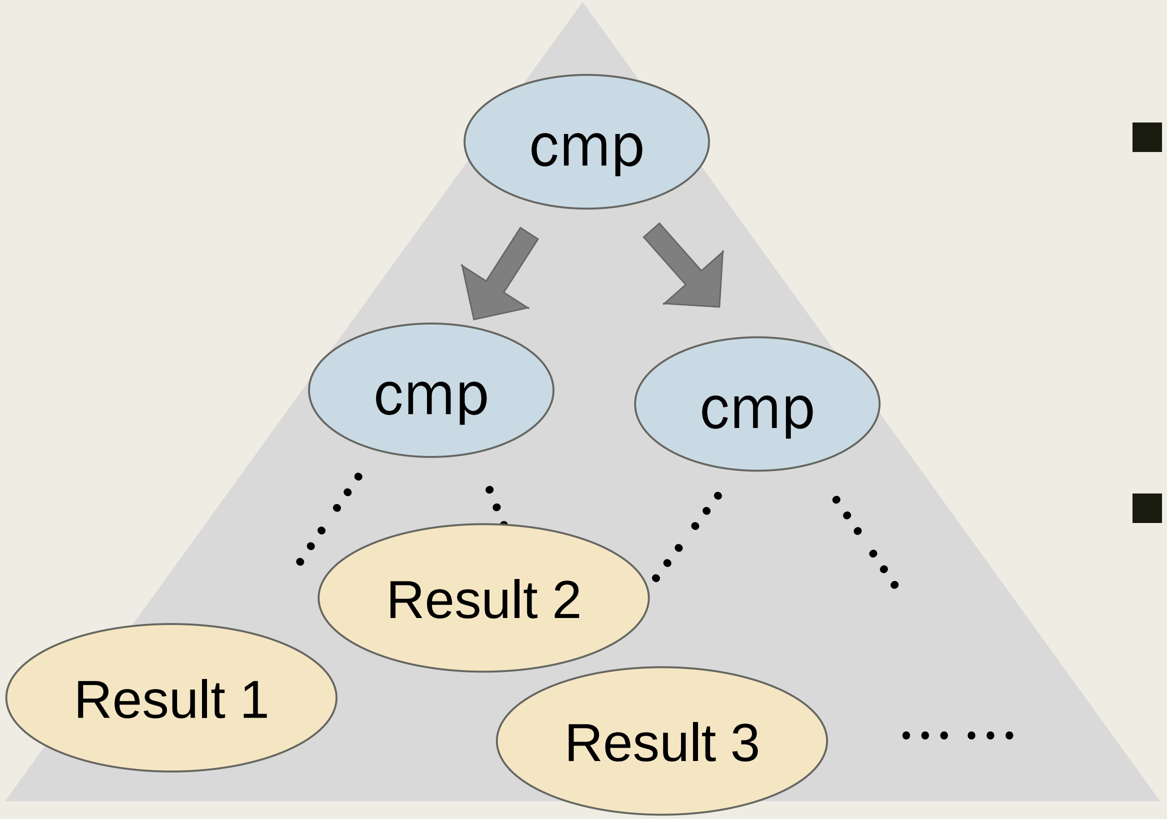


Each of the comparison has exactly two different outcomes.

- The execution of *<u>any comparison-based sorting algorithm</u>* consists of a sequence of comparisons between the given numbers.



- Hence, the execution process of a comparison-based sorting algorithm in fact ***corresponds to a binary tree***, where the leaf nodes are *the set of all possible results*.
  - *Abstract Decision Tree* Model (ADT model)



- A sequence of k comparisons can ***classify*** at most 2^k different sorted results.

- There are $n!$ potentially different input sequences.

- A sequence of k comparisons can ***classify*** at most 2^k different types of input sequences.
- There are $n!$ potentially different input sequences.
- Hence, any comparison-based sorting algorithm needs at least

$$\log(n!) = \Theta(n \log n)$$

number of comparisons to classify the input correctly,
where in the above we use the *Stirling's approximation* for $n!$,

$$n! \;=\; \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

(Non-Comparison-based)

Linear-Time Sorting Algorithms

We need to know additional properties of the input data.

Sorting in Linear Time

- We will introduce three different types of such algorithms.
  - ***<u>Counting Sort</u>*** Algorithm
    - Used when the input has only a small number of *<u>distinct values</u>*.
  - ***<u>Radix Sort</u>*** Algorithm
    - Used when the *input elements can be <u>represented</u>* by a small number of *<u>digits</u>* from a small set of *<u>alphabets</u>*.
  - ***<u>Bucket Sort</u>*** Algorithm
    - Used when the input distribution is close to *<u>uniformly random</u>*.

Counting Sort

Sort by counting the number of elements.

Counting Sort

- The counting sort algorithm is used when the input numbers are ***selected from a small subset***.

  - For example,

    for all $1 \leq i \leq n$, $a_i \in \{1,2,\ldots,k\}$ for some constant k.

  - In this case,

    we can simply "***count***" *the number of appearances* of value i in the input sequence *for each possible* $i \in \{1,2,\ldots,k\}$.

Counting Sort

- The counting sort algorithm is used when the input numbers are selected from a small subset $\{1,2,\dots,k\}$ for some constant k.

  - In this case,

    we can simply "***<u>count</u>***" *the number of appearances* of value i in the input sequence *<u>for each possible</u>* $i \in \{1,2,\dots,k\}$.

  - The time it takes will be $O(n+k)$.

Counting Sort

- For example,
  suppose that the input numbers come from $\{1, 2, 3, 4\}$.

- Then, in the sequence $2, 1, 2, 3, 1, 2, 3$,

  - The appearances of the four elements are $2, 3, 2, 0$, respectively.

  - Hence, the resulted sorted sequence will be

$$1, 1, 2, 2, 2, 3, 3 .$$

- CountingSort($A[\,1,2,\ldots,n\,], n, k$)

A. Initialize $C[\,0,\ldots,k\,]$ to be zero.

B. For $j \leftarrow 1$ to n, do the following.
   - Increase $C\big[A[j]\big]$ by 1.

C. For $j \leftarrow 1$ to k, do the following.
   - Set $C[j] = C[j] + C[j-1]$.

D. For $j = n$ to 1, do the following.
   - Set $B\Big[C\big[A[j]\big]\Big]$ to be $A[j]$.
   - Decrease $C\big[A[j]\big]$ by 1.

Now $C[i]$ counts the number of appearances of i.

Now $C[i]$ counts the number of elements that are ***at most i***.

Now $B[1,\ldots,n]$ will be the resulted sorted array.

Some Notes

- This algorithm works as long as the input elements can be ***represented*** by $\{1,2,\dots,k\}$ for a small k.
  - For example, $\{\,a, b, \dots, z\,\}$ can be mapped to $\{1,2,\dots,26\}$.
  - The mapping needs to be done efficiently.
- The counting sort algorithm maintains *the original order* of the elements that have the same ranking.
  - It is a ***stable sorting algorithm***.

Q: How does counting sort guarantee this property?

Stable Sorting Algorithm

- Example. Suppose that a company has three departments “A”, “B”, and “C”, and the employees are identified by the department he/she is working at and also a serial number.
  - Ex. B-010, C-123, A-015, A-016, A-003, B-003.
- Suppose that we want to sort the IDs of the employees only according to the departments.
- Then, a stable sorting algorithm will always produce the list

  A-015, A-016, A-003, B-010, B-003, C-123.

Stable Sorting Algorithm

- The following sorting algorithms are stable by default.
  - Insertion Sort, Bubble Sort, Selection Sort.
  - Counting Sort.

- Nevertheless, with an $O(n)$ extra storage space, all sorting algorithms can be made stable.

Radix Sort

Sort by elements digit by digit.

Radix Sort

- The radix sort works when the input elements can be represented by a string with **a small length** and **a small set of alphabets**.
  - Numbers between 0 and 999.
  - Strings with length 10.
  - IDs of citizens in Taiwan.
  - etc.

Radix Sort

- The radix sort algorithm considers the digits of the representation one by one, *from* ***the least significant*** *to* ***the most significant***.
  - For each digit considered, it uses a stable sorting algorithm, e.g., *counting sort*, to sort the elements according to that digit.

- RadixSort($A[\,1, 2, \ldots, n\,], n, d, k$) - d: number of digits,

  k: number of values for each digit

A. For $j \leftarrow 1$ to d, do the following.
   - Use counting sort to sort A according to the i-th digit.

Radix Sort

Lemma.

Given n $\boldsymbol{d}$**-digit** numbers in which ***each digit*** can take on up to $\boldsymbol{k}$ ***possible values***, the radix sort algorithm correctly sorts these numbers in $\Theta\big(d(n+k)\big)$ time.

- The time complexity is straight-forward.
- For the correctness,
  observe that at the end of the j-th iteration, all the numbers with the same $(d-j)$-digits prefix are sorted in order.

Radix Sort

- For numbers represented by ***b*-bits binary strings**, we have the following tricks for any $0 < r \leq b$.
  - Divide the string into substrings of length r.

<u>Lemma.</u>

Given n ***b*-bit** numbers and any possible $r \leq b$, the radix sort algorithm correctly sorts these numbers in $\Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)$ time.

- When $b = O(\log n)$ and $r = \log n$, radix sort works in $\Theta(n)$ time!

Further Discussion

- Very often, the input numbers are represented by binary strings of constant length.
  - Hence, radix sort gives a running time guarantee of $\Theta(n)$.

- Does it mean that radix sort is the best sorting algorithm in this circumstance?
  - In theory, yes.
  - In practice, it depends.

Further Discussion

- Does it mean that radix sort is the best sorting algorithm in this circumstance? Ans: In practice, it depends.

  For example,

  – Radix sort requires an extra $O(n)$ storage, while some $O(n \log n)$ algorithm, such as ***quick-sort***, sorts the number ***in place***.

  – Very often, the hidden constant in $\Theta(n)$ is comparable to the $O(\log n)$ factor for the divide-and-conquer sorting algorithms.

  – Quick-sort may perform better due to better CPU cache usage and compiler optimization, etc.

Bucket Sort

Works in linear time when the input is ***uniformly random***.

Bucket Sort

- The bucket sort algorithm works extremely well when the ***input numbers*** are *drawn from a* ***uniform distribution***.
- The idea of this algorithm is to divide the possible range of input numbers into n *equal-sized subintervals*.
  - Since the numbers are drawn from uniform distribution, there are $\boldsymbol{O(1)}$ ***elements*** in each sub-interval ***in expectation***.
  - Hence, any sorting algorithm can be used to sort these elements in expected $O(1)$ time.

This requires formal proofs, though.

Bucket Sort

- BucketSort($A[\,1, 2, \ldots, n\,], n, R$) - R: range of input numbers

---

A. Initialize $B[\,0, \ldots, n-1\,]$ to be n empty lists.

B. For $j \leftarrow 1$ to n, do the following.
   - Insert $A[j]$ into the list $B\left[\left\lfloor n \cdot \frac{A[j]}{R} \right\rfloor\right]$.

C. For $j \leftarrow 0$ to $n-1$, do the following.
   - Sort the elements in $B[j]$ using Insertion Sort algorithm.

D. Concatenate the lists $B[0], \ldots, B[n-1]$ in order
   to obtain the resulted sorted list.

The Analysis

- For any $0 \leq i < n$,
  let n_i be the number of elements in list $B[i]$.

  – Then,

$$n_i \;=\; \sum_{1 \leq j \leq n} X_{i,j} \;,$$

  where $X_{i,j}$ denotes the indicator variable
  for the event that the j-th element falls in the list $B[i]$.

- Let $T(n)$ be the running time of the Bucket sort algorithm.
  - Then,

$$T(n) = \Theta(n) + \sum_{0 \le i < n} O\left(n_i^2\right) .$$

  - Hence,

$$\mathrm{E}[\, T(n) \,] = \Theta(n) + \sum_{0 \le i < n} O\left(E\left[n_i^2\right]\right) ,$$

where

$$E\left[\, n_i^2 \,\right] = E\left[\sum_{1 \le j \le n} X_{i,j}^2 + \sum_{\substack{1 \le j,k \le n \\ j \ne k}} X_{i,j} \cdot X_{i,k} \right]$$

$$= \sum_{1 \le j \le n} E\left[X_{i,j}^2\right] + \sum_{\substack{1 \le j,k \le n \\ j \ne k}} E\left[X_{i,j} X_{i,k}\right] .$$

- where $$E[\,n_i^2\,] = \sum_{1\le j\le n} E[X_{i,j}^2] + \sum_{\substack{1\le j,k\le n\\ j\ne k}} E[X_{i,j}X_{i,k}].$$

- Since the numbers are drawn from uniform distribution,

$$\Pr[\,X_{i,j} = 1] = \frac{1}{n} \quad \text{for all } 0 \le i < n, 1 \le j \le n.$$

Furthermore, $X_{i,j}$ and $X_{i,k}$ are independent for $j \ne k$.

- Hence,

$$\sum_{1\le j\le n} E[X_{i,j}^2] = 1 \quad \text{and} \quad \sum_{\substack{1\le j,k\le n\\ j\ne k}} E[X_{i,j}X_{i,k}] = n(n-1)\cdot\frac{1}{n^2} \le 1.$$

The Analysis

- Let $T(n)$ be the running time of the Bucket sort algorithm.

  Then,

$$
\begin{aligned}
\mathrm{E}[\,T(n)\,] &= \Theta(n) + \sum_{0\le i<n} O\left(E\left[n_i^2\right]\right) \\
&= \Theta(n) + \sum_{0\le i\le n} O\left(\sum_{1\le j\le n} E\left[X_{i,j}^2\right] + \sum_{\substack{1\le j,k\le n \\ j\ne k}} E\left[X_{i,j}X_{i,k}\right] \right) \\
&= \Theta(n) + \sum_{0\le i\le n} O(1) = \Theta(n).
\end{aligned}
$$