SORTING IN LINEAR TIME

Juan Mendivelso

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

CONTENTS

1. LOWER BOUNDS FOR SORTING

Comparison Sorts

- Sorting algorithms that are based on comparisons between the input elements.
- Comparison sorts include
  - Insertion Sort
  - Merge Sort
  - Quicksort
  - Heapsort
- The worst case of Merge Sort and Heapsort is O(n lg n).
- The average case of Quicksort is O(n lg n).
- A comparison sort requires Ω(n lg n).

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

The Decision-Tree Model

- We can view comparison sorts abstractly in terms of decision trees.
- A **decision tree** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.
- Control, data movement and all other aspects of the algorithm are ignored.

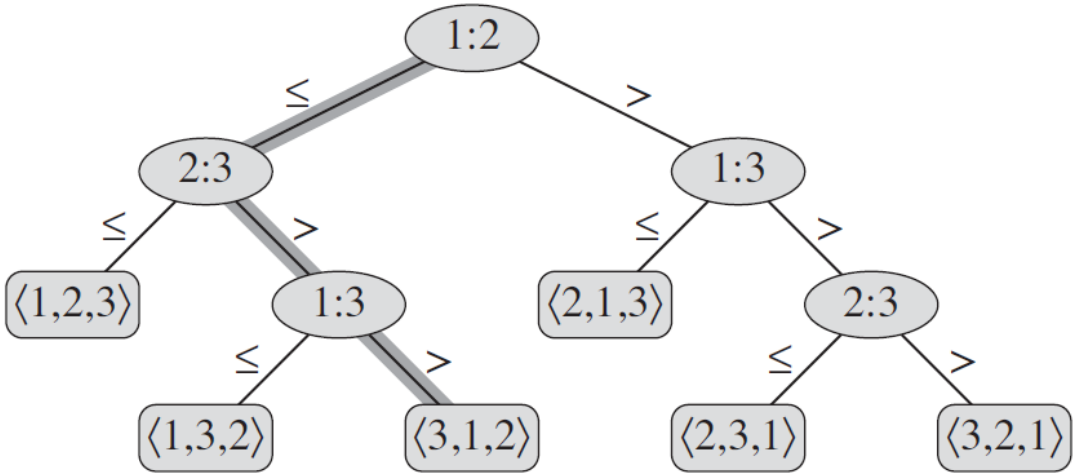
Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

The Decision-Tree Model

- In each internal node, a_i:a_j, $1 \leq i, j, \leq n$, represents a comparison.
- The left (right) subtree indicates that $a_i \leq a_j$ ($a_i > a_j$).
- Each leaf represents a permutation $\pi(1), \pi(2),\ldots, \pi(n)$ that indicates the sorted order of the input, i.e. $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$.
- Each of the n! permutations must appear as leaves.

Decision tree of Insertion Sort on array $A=a_1a_2a_3$ of size 3.

Example with 6,8,5:



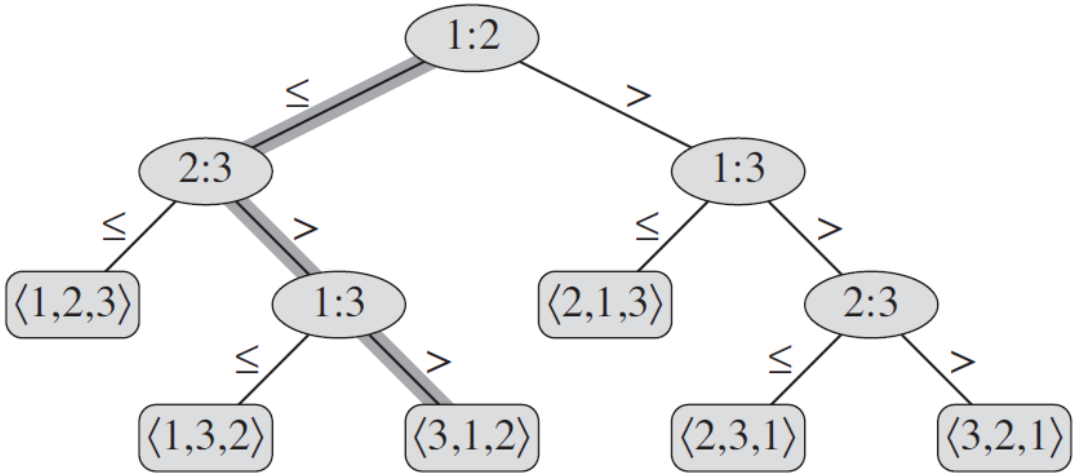
Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Bound for the Worst Case

- The length of the longest path from the root to a leaf is the number of comparisons in the worst case, i.e. the height of the decision tree.
- A bound on such height is a bound on any comparison sort.

Decision tree of Insertion Sort on array $A=a_1a_2a_3$ of size 3.

Example with 6,8,5:



Bound for the Worst Case

Theorem: Any comparison sort requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof:

- Let h be the height of a decision tree to sort n elements.
- There must be at least n! leaves that correspond to the permutations.
- Because it is a binary tree, the number of leaves is at most 2^h.

$$n! \leq 2^h$$
$$\lg(n!) \leq h$$
$$h = \Omega(n \lg n)$$

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

Corollary: Merge Sort and Heapsort are asymptotically optimal.

Sorting in linear time

- However, depending on the range or distribution on the input, we can sort arrays in linear time.
- With Counting Sort, we can sort in linear time if the range [0,k] of the values sorted is smaller than the number n of elements to be sorted., i.e. k =O(n).
- With Radix Sort, we can sort records of information that are keyed by multiple fields in linear time.
- With Bucket Sort, we can sort numbers drawn from a uniform distribution in linear time.

2. COUNTING SORT

Counting Sort

- The n input elements are integers in [0,k].
- If k=O(n), the algorithm runs in O(n).
- Idea: For each value x, determine how many elements < x there are. For instance, if there are 5 elements < x, there must be an x at position 5.
- In case several elements have the same value, we consider how many elements in A are ≤ x. We denote this by C[x].
- Then, we traverse A from right to left. Whenever a value x is found, we know there is an x at position C[x]. Then, we decrement C[x] to place other occurrences of x in previous positions.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Arrays Used

- We use the following arrays:
  - A[1..n]: Input array.
  - C[0..k]: At first, C[i] contains how many times i occurs at A. Then, it contains how many elements in A are ≤ i. It is decremented as soon as an occurrence of i is reached.
  - B[1..n]: Output array: same elements from A but sorted.

Pseudocode

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Example

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

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

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

(a)

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

(b)

	1	2	3	4	5	6	7	8
B							3	

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

(c)

	1	2	3	4	5	6	7	8
B		0					3	

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

(d)

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

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

(e)

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

(f)

Complexity Analysis

```
COUNTING-SORT(A, B, k)
1  let C[0..k] be a new array
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i - 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 for j = A.length downto 1
11     B[C[A[j]]] = A[j]
12     C[A[j]] = C[A[j]] - 1
```

- Lines 1- 3: θ(k)
- Lines 4 - 6: θ(n)
- Lines 7 - 9: θ(k)
- Lines 10-12: θ(n)
- Total Complexity: θ(n+k)
- Note that if k =O(n), then the time complexity is θ(n).

Stability

```
COUNTING-SORT(A, B, k)
1  let C[0..k] be a new array
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i - 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 for j = A.length downto 1
11     B[C[A[j]]] = A[j]
12     C[A[j]] = C[A[j]] - 1
```

- Is this algorithm in-place?
- Is this algorithm stable?
- What happens if we traverse from 1 to A.length in line 10?
- Counting-Sort is actually stable, i.e. numbers with the same value appear in the output array in the same order as they do in the input array.
- This property is important when satellite data are carried around with the element being sorted.
- Its stability is also relevant so it can be used by the radix sort algorithm.

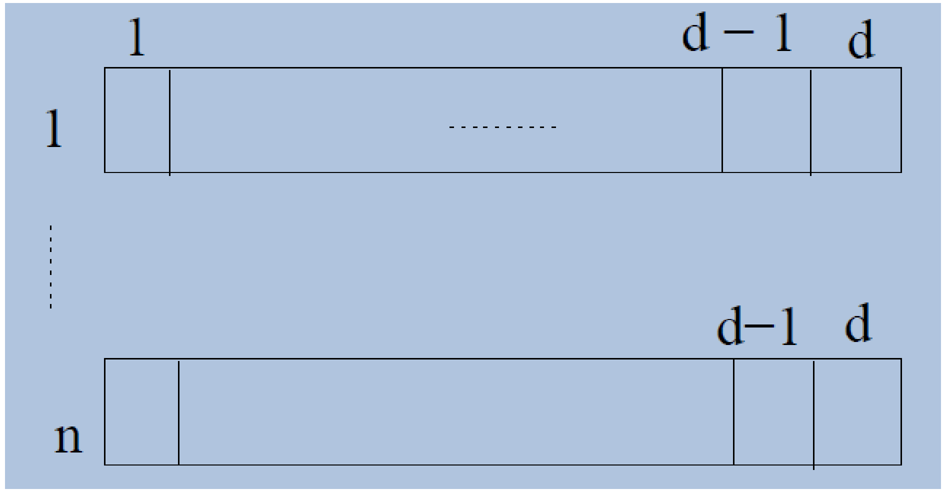
3. RADIX SORT

Introduction to Radix Sort

- What is the birthdate of a group of people?
- How can we sort them?

Radix Sort

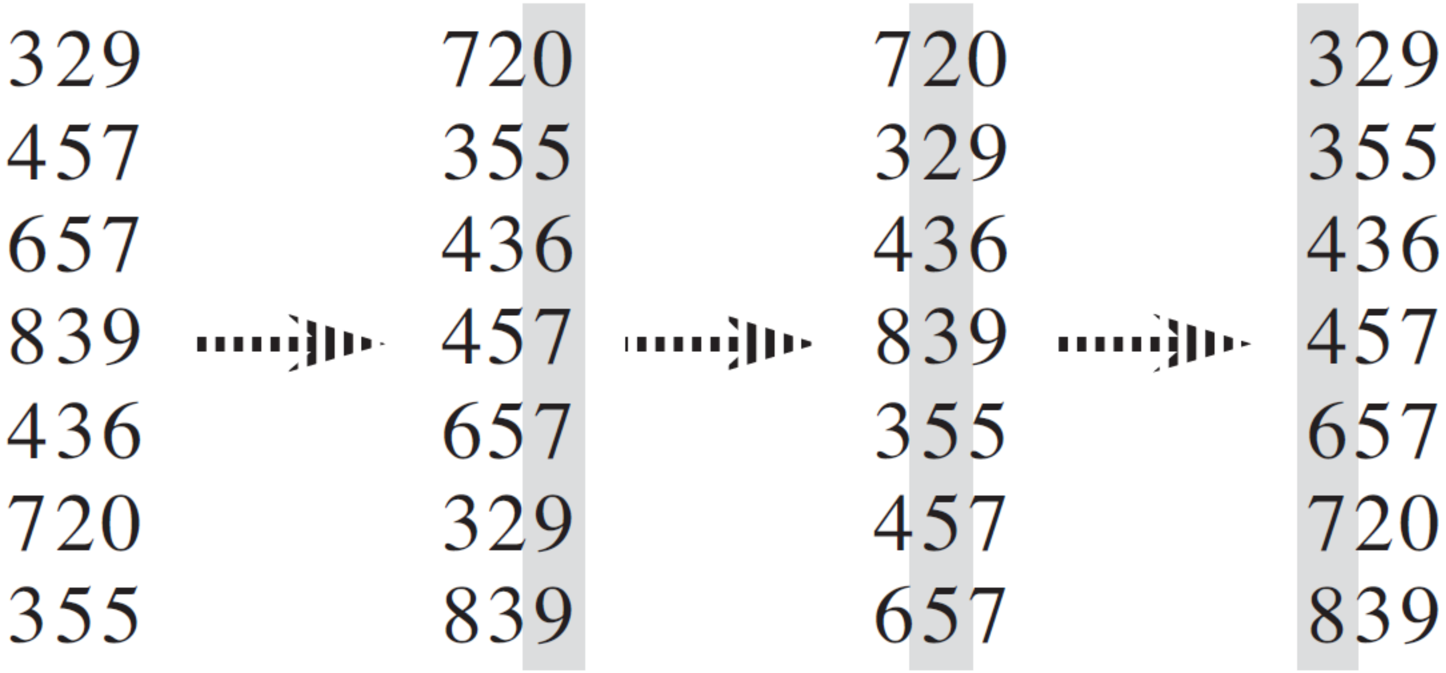
- It allows to sort records of information that are keyed by multiple fields in linear time.
- Particularly, we can sort n records of d digits where the first digit is the least significant and the d-th digit is the most significant.



Material Curso de Algoritmos – Julio López

Radix Sort

- Radix Sort proposes to sort the records on ascending order of the significance of their digits.



Pseudocode & Complexity

```
RADIX-SORT(A, d)
for i = 1 to d
        use a stable sort to sort array A on digit i
```

Lemma 1: Given n numbers of d digits in which each digit can take k possible values, Radix Sort can sort them in θ(d(n+k)).

Lemma 2: Given n numbers of b bits and some positive integer r ≤ b, Radix Sort can sort them in $\theta((b/r)(n+2^r))$.

Proof: Let r be the number of bits assigned to each digit. Then, there are $d = \left\lceil \frac{b}{r} \right\rceil$ digits and each of them can represent numbers in $[0, 2^r-1]$, i.e. $k=2^r$.

4. BUCKET SORT

Bucket Sort

- It assumes that the input is drawn from a uniform distribution.
- Its average-case running time is O(n).
- It is assumed that the input is generated by a random process that distributes elements uniformly and independently over the Interval [0,1).
- If the input is of size n, we use an array B[0..n-1] of buckets.
- The elements in the array are distributed into the buckets.
- Because the distribution of the data is uniform, we expect that each bucket receives one bucket.
- However, if it's not the case, the elements in each bucket are sorted with Insertion Sort.

Pseudocode

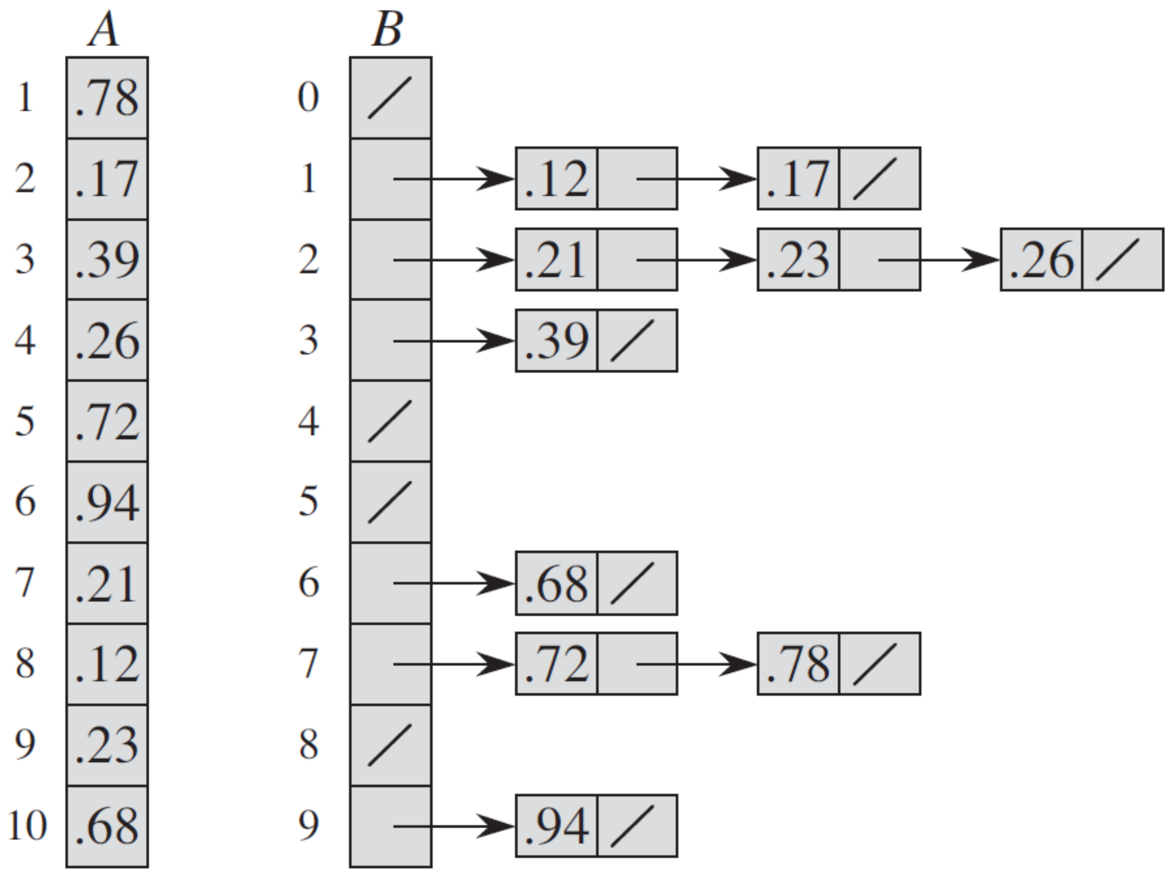
- Because 0 ≤ A[i] <1, we can assign into the bucket $\mathrm{B}[\lfloor nA[i] \rfloor]$.

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
        make B[i] an empty list
for i = 1 to n
        insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
        sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Example

BUCKET-SORT(*A*)

```
let B[0 . . n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```



(a) (b)

Correctness

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

- Consider two elements A[i] and A[j].
- Assume A[i] ≤ A[j].
- Since $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, either A[i] goes into the same bucket as A[j] or it goes in a bucket with a lower index.
- In the former case, lines 7-8 sorts them.
- In the latter case, line 9 puts them in the proper order.

Complexity Analysis

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

- Lines 3-4, 5-6 and 9 take θ(n).
- Line 6 depends on the size of the bucket i, denoted n_i.
- Then, the complexity of the algorithm is

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Complexity Analysis

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

- We now analyze the average-case running time:

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \\
&= \Theta(n) + n \cdot O(2 - 1/n) = \Theta(n) .
\end{aligned}$$

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n$$

Complexity Analysis

$$E\left[n_i^2\right] = 2 - 1/n$$

- Each bucket I has the same value of $E[n_i^2]$ since each input of the input array A is equally likely to fall in any bucket.

$$X_{ij} = I\{A[j] \text{ falls in bucket } i\} \text{ for } i = 0, 1, \ldots, n-1 \text{ and } j = 1, 2, \ldots, n.$$

$$n_i = \sum_{j=1}^{n} X_{ij}$$

Complexity Analysis

$$n_i = \sum_{j=1}^{n} X_{ij}$$

$$\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n}\mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} \mathrm{E}\left[X_{ij}X_{ik}\right] ,
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}\left[X_{ij}^2\right] &= 1^2\cdot\frac{1}{n} + 0^2\cdot\left(1-\frac{1}{n}\right) \\
&= \frac{1}{n} .
\end{aligned}$$

When $k \ne j$, the variables X_{ij} and X_{ik} are independent, and hence

$$\begin{aligned}
\mathrm{E}\left[X_{ij}X_{ik}\right] &= \mathrm{E}\left[X_{ij}\right]\mathrm{E}\left[X_{ik}\right] \\
&= \frac{1}{n}\cdot\frac{1}{n} \\
&= \frac{1}{n^2} .
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}\left[n_i^2\right] &= \sum_{j=1}^{n}\frac{1}{n} + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} \frac{1}{n^2} \\
&= n\cdot\frac{1}{n} + n(n-1)\cdot\frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n} ,
\end{aligned}$$

BIBLIOGRAPHY

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press. 2009.
- Images of Julio Cesar Lopez.