Lecture 8: Sorting in Linear Time

Running time of sorting algorithms

Do you still remember what these statements mean?

- Sorting algorithm $\mathcal{A}$ runs in $O(n \log n)$ time.
- Sorting algorithm $\mathcal{A}$ runs in $\Omega(n \log n)$ time.

So far, all algorithms have running time $\Omega(n \log n)$

- We didn't show this for heap sort, though it is true.

Q: Is it possible to design an algorithm that runs faster than $\Omega(n \log n)$?

A: No, if the algorithm is comparison-based.

Remark: A comparison-based sorting algorithm is more general, e.g., the sorting algorithm implemented in C++ STL.

Corollary: Heap sort has running time $\Theta(n \log n)$.

Lower bounds for problems

What does each of the following statements mean?

- Sorting can be done in $O(n \log n)$ time.
  - There is a sorting algorithm that runs in $O(n \log n)$ time.
- Sorting requires $\Omega(n \log n)$ time.
  - Any sorting algorithm must run in $\Omega(n \log n)$ time.

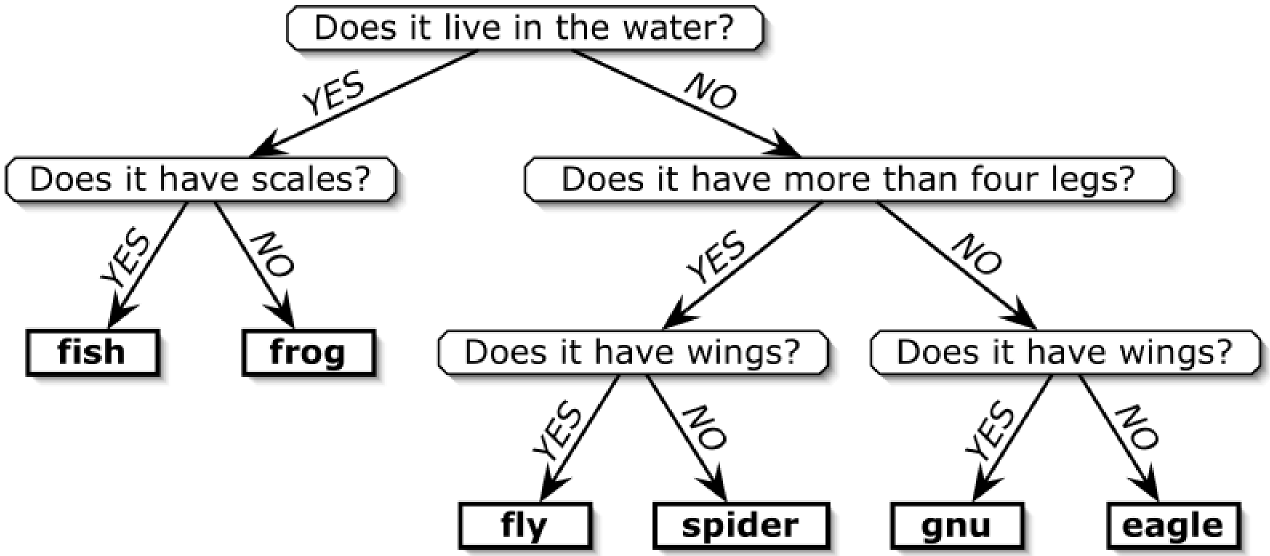
This is a very strong statement, which we unfortunately cannot prove.

- In fact, we have no lower bound higher than $\Omega(n)$ for sorting.

But we can show an $\Omega(n \log n)$ lower bound for comparison-based sorting algorithms

- More generally, in the decision-tree model of computation.

The decision-tree model

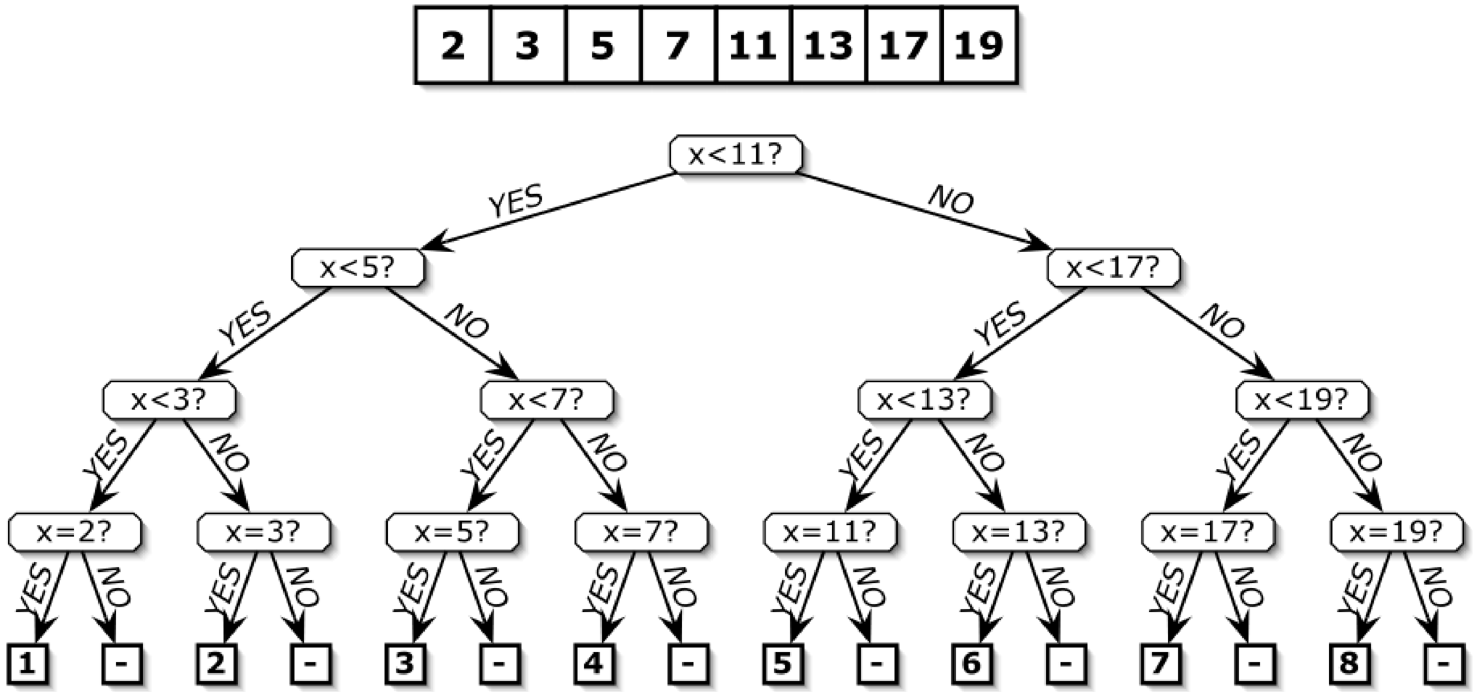


A decision tree to choose one of six animals.

An algorithm in the decision-tree model

- Solves the problem by asking questions with binary answers
- Cannot ask questions like "how many legs does it have?"
- The worst-case running time is the height of the decision tree.

The decision-tree for binary search



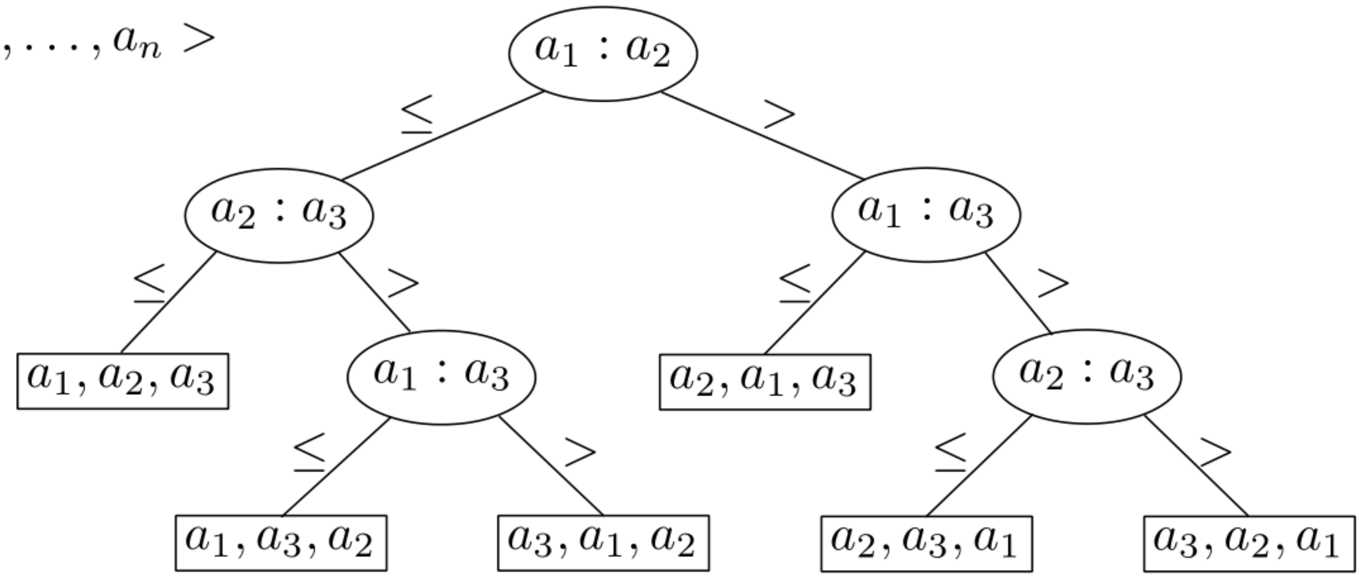
Theorem: Any algorithm for finding a given element in a sorted array of size n must have running time $\Omega(\log n)$ in the decision-tree model.

Proof:

- The algorithm must have at least n different outputs.
- The decision-tree has at least n leaves.
- Any binary tree with n leaves must have height $\Omega(\log n)$.

The decision-tree for sorting

Sort $< a_1, a_2, \ldots, a_n >$



Theorem: Any algorithm for sorting n elements must have running time $\Omega(n \log n)$ in the decision-tree model.

Pf:

- The algorithm must have at least $n!$ different outputs (there are so many different permutations).
- The decision-tree has at least $n!$ leaves.
- Any binary tree with n leaves must have height $\Omega(\log(n!)) = \Omega(n \log n)$.

Can we do better?

Implication of the lower bound

- Have to use non-comparison based algorithms.
- Don't use worse-case analysis

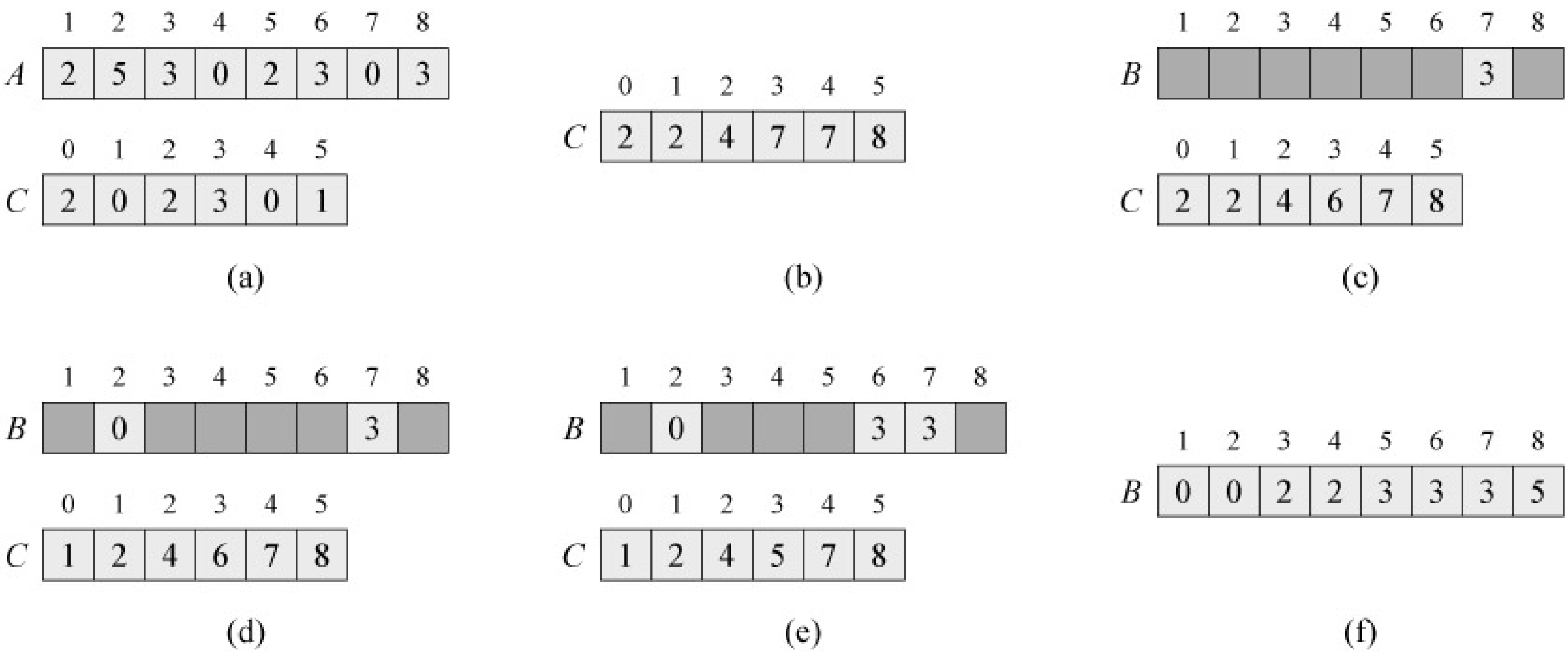
Integer sorting

- We will assume that the elements are integers from 0 to k.
- We will use both n and k to express the running time.
- Both $n < k$ are $n > k$ possible.

Exercise. Compare the following functions asymptotically:

$$n, \log k, n + 1000, n + k, n \log k\,, \max(n, k)\,, \min(n, k)$$

Counting sort



A: input array

B: output array

C: counting array, then position array

Counting sort

```
Counting-Sort(A,B):
let C[0..k] be a new array
for i ← 0 to k
    C[i] ← 0
for j ← 1 to n
    C[A[j]] ← C[A[j]] + 1
    // C[i] now contains the number of i's
for i ← 1 to k
    C[i] ← C[i] + C[i − 1]
    // C[i] now contains the number of elements ≤ i
for j ← n downto 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1
```

Running time: $\Theta(n + k)$
Working space: $\Theta(n + k)$

It is a stable sorting algorithm.

Radix sort

2 3 2 9		2 7 2 0		2 7 2 0		2 3 2 9		2 3 2 9
5 4 5 7		5 3 5 5		2 3 2 9		5 3 5 5		2 7 2 0
3 6 5 7		3 4 3 6		3 4 3 6		3 4 3 6		3 4 3 6
5 8 3 9	→	5 4 5 7	→	5 8 3 9	→	5 4 5 7	→	3 6 5 7
3 4 3 6		3 6 5 7		5 3 5 5		3 6 5 7		5 3 5 5
2 7 2 0		2 3 2 9		5 4 5 7		2 7 2 0		5 4 5 7
5 3 5 5		5 8 3 9		3 6 5 7		5 8 3 9		5 8 3 9

Radix-Sort(A,d):

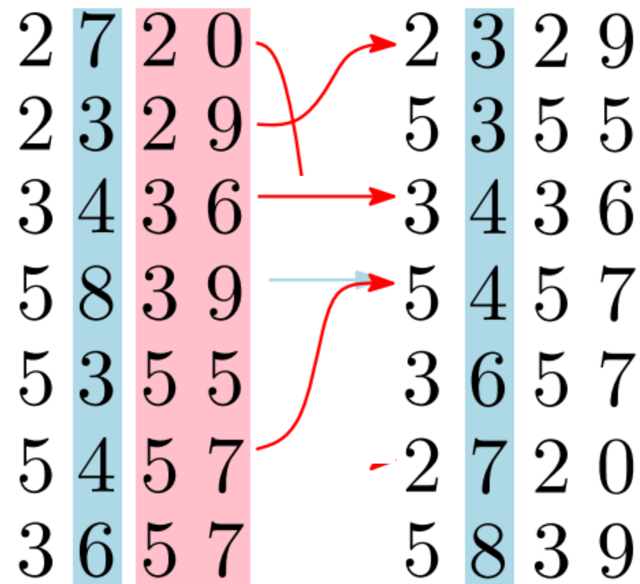
for $i \leftarrow 1$ to d

  use counting sort to sort array A on digit i

Radix sort: Correctness

Proof: (induction on digit position)

- Assume that the numbers are sorted by their low-order $i-1$ digits
- Sort on digit i
  - Two numbers that differ on digit i are correctly sorted by their low-order i digits
  - Two numbers equal on digit i are put in the same order as the input $\Rightarrow$ correctly sorted by their low-order i digits



Radix sort: Running time analysis

Q: How large should the "digits" be?

Analysis: Let each "digit" take values from 0 to $b-1$.

- Counting sort takes $\Theta(n+b)$ time.
- An integer $\leq k$ has $d = \log k / \log b$ such "digits".
- So the running time of radix sort is $\Theta\left(\frac{\log k}{\log b} \cdot (n+b)\right)$
- This is minimized when $b = \Theta(n)$.
- The running time is $\Theta(n \log_n k)$, which is $O(n)$ when $\log k = O(\log n)$

Q: When is this faster than $\Theta(n \log n)$?

A: When $\log_n k < \log n \Leftrightarrow \log k < \log^2 n$

Implementation:

- Choose $n \leq b < 2n$ such that b is a power of 2.
- Use bit-wise operation for more efficient implementation.

Summary of sorting algorithms

	Insertion sort	Merge sort	Quicksort	Heapsort	Radix sort
Running time	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log_n k)$
Working space	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n)$
Randomized	No	No	Yes	No	No
Cache performance	Good	Good	Good	Bad	Bad
Parallelization	No	Excellent	Good	No	No
Stable	Yes	Yes	No	No	Yes
Comparison-based	Yes	Yes	Yes	Yes	No

Bucket sort

Average-case analysis

- Bucket sort is the only algorithm for which we do an average-case analysis in this course.
- We will assume that the elements are drawn uniformly and independently from a range.
  - We can assume the range is $[0,1)$ without loss of generality.
  - They can be real numbers; while radix sort only works on integers

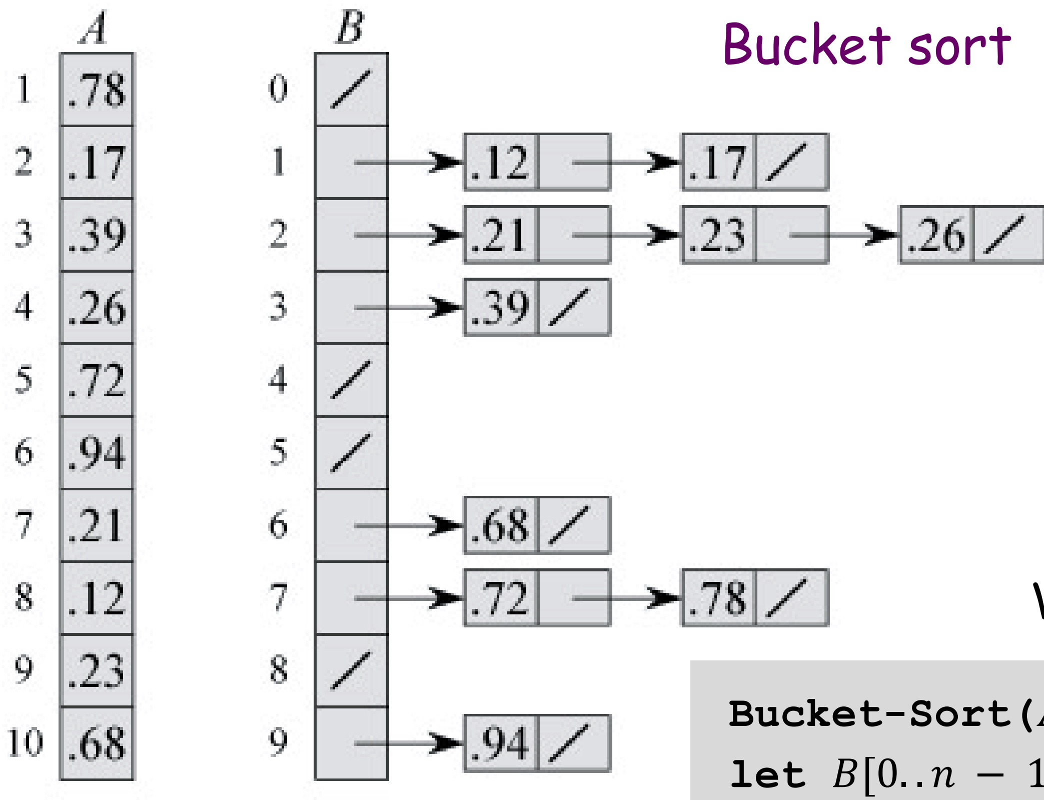
Non-comparison based

- Bucket sort is non-comparison based
- It breaks the lower bound by breaking both conditions.

Q: Is it possible to design a comparison-based sorting algorithm with expected or average running time better than $\Theta(n \log n)$?

A: No. (Proof omitted)

Bucket sort



(a) (b)

Worst-case: $\Theta(n^2)$

```
Bucket-Sort(A)
let B[0..n − 1] be a new array
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
```

Average-case analysis of bucket sort

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

$$E[T(n)] = E[n] + E\left[\sum_{i=0}^{n-1} n_i^2\right]$$

$$= n + \sum_{i=0}^{n-1} E\left[n_i^2\right]$$

Claim: $E\left[n_i^2\right] = 2 - \frac{1}{n}$

Pf: Let $X_{ij} = 1$ if $A[j]$ falls into bucket i.

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

Average-case analysis of bucket sort

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

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

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

Theorem: When the inputs are drawn from a range uniformly and independently, bucket sort runs in expected $\Theta(n)$ time.