CSCI 4470 Algorithms

Part II Sorting and Order Statistics

Chapter 8: Sorting in Linear Time

Sorting in Linear Time

Introduction

- Linear-time sorting algorithms are efficient for large datasets.
- They don't make comparisons but use other methods.
    - `Counting Sort`, `Radix Sort`, and `Bucket Sort` Algorithms

Lower Bounds for Sorting

- **Comparison sorts**: Any algorithm that sorts by comparing elements.
- **Lower bound of comparison sorts**: ***Proves that any comparison sort algorithm requires $\Omega(n \log n)$ time in the WORST CASE.***

Decision Tree Model

- A tool used to prove the lower bound of comparison sorts.
- ***Every comparison-based sorting algorithm has a decision tree***

Properties of Decision Trees

- **Nodes**: Represent decisions or comparisons.
- **Edges**: Represent the outcome of a decision.
- **Leaves**: Represent final outcomes or sorted permutations.

Height and Leaves

- **Height** h: h The maximum number of comparisons needed in the worst case.
- **Number of Leaves**: Given n elements, the total number of leaves is $n!$ (n factorial), representing all possible sorted permutations.
- **Inequality**: To have all possible sorted permutations, $2^h \geq n!$.

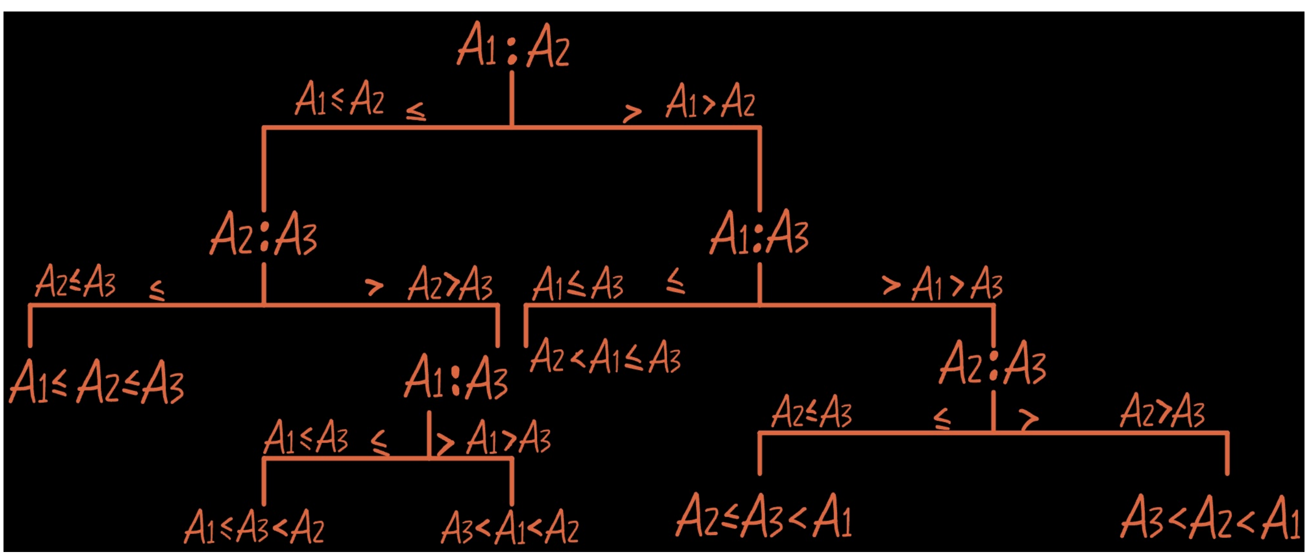
Height Boundaries

- The height h must satisfy $2^h \geq n!$ to ensure the tree has sufficient leaves.
- Using Stirling's approximation: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, the height is bounded by $h \geq n\log(n)$.
- **Logarithmic Boundaries**:
  - $h \geq \log(n!)$
  - $h \geq n\log(n)$

Decision Tree Model for Comparison Sort

Example of Decision Tree Given the permutation of 3 input values $\{A_1, A_2, A_3\}$, for an input of n elements, how many leaves do we have?

- ***In a decision tree for n elements, the number of leaves, representing all possible sorted permutations, is $n!$ (n factorial).***
- By the example, with 3 elements $\{A_1, A_2, A_3\}$, there are $3! = 6$ leaves, each corresponding to a unique permutation.
- **Decision Tree for**



  $\{A_1, A_2, A_3\}$:
  - Visualizes the comparisons between the three elements to determine all possible sorted permutations.

- **Left Subtree**: Represents “less than or equal to” comparisons.
- **Right Subtree**: Represents “greater than” comparisons.

- **First Level**: We start by comparing A_1 and A_2.
- **Second Level**: Depending on the result of the first comparison, we have two paths:
  - **Left Subtree**: If A_1 is less than or equal to A_2, we compare A_2 and A_3 and A_1 and A_3.
  - **Right Subtree**: If A_1 is greater than A_2, we compare A_1 with A_3 and A_2 with A_3.
- **Third Level**: Here, we reach the leaf nodes, which represent the sorted permutations of the input set.
- The sorted permutations corresponding to each leaf (L1 to L8):
- **L1**: $A_1 \leq A_2 \leq A_3$, **L2**: $A_1 \leq A_3 < A_2$, **L3**: $A_3 < A_1 \leq A_2$
- **L4**: $A_1 \leq A_2 < A_3$, **L5**: $A_2 < A_1 \leq A_3$, **L6**: $A_2 < A_3 < A_1$
- **L7**: $A_3 < A_2 < A_1$, **L8**: $A_2 < A_1 < A_3$

The height h of the decision tree:

- the height of the decision tree is based on the number of leaves, which is equal to the number of possible permutations ($n!$). The formula is:
  - $h \approx \log_2(n!)$
- By the example with 3 elements ($\{A_1, A_2, A_3\}$), we apply the formula:
  - $h \approx \log_2(3!) = \log_2(6) \approx 2.585$
- round up to get an integer value for the height:
  - $h = 3$
- ***This indicates that we will need up to 3 comparisons in the worst case.***

Given 3 input values $\{A_1, A_2, A_3\}$, we want to find out how many leaves we have in the decision tree.

1. **Number of Leaves:**
   - In a decision tree for n elements, the number of leaves is $n!$.
   - For $\{A_1, A_2, A_3\}$, there are $3! = 6$ leaves.
   - Each leaf corresponds to a unique permutation of the input values.
2. **Decision Tree Structure:**
   - **First Level:**
     - Compare A_1 and A_2.
   - **Second Level:**
     - If $A_1 \leq A_2$, compare A_2 and A_3 and A_1 and A_3.
     - If $A_1 > A_2$, compare A_1 with A_3 and A_2 with A_3.
   - **Third Level:**
     - Reach the leaf nodes, representing the sorted permutations of the input set.
3. **Sorted Permutations:**
   - Each leaf (L1 to L8) corresponds to a sorted permutation of $\{A_1, A_2, A_3\}$.
4. **Height of the Decision Tree:**
   - Formula: $h \approx \log_2(n!)$.
   - For $\{A_1, A_2, A_3\}$: $h \approx \log_2(3!) = \log_2(6) \approx 2.585$.
   - Round up: $h = 3$.
   - Indicates up to 3 comparisons in the worst case.

Determine Min Worst Case of The Decision Tree

1. Height and Leaves:

- The decision tree for a given comparison sort algorithm with input size = n has $n!$ leaves.
- Each leaf represents a distinct ordering of $a_1, a_2, \ldots, a_n$.
- h is the height of the tree.
- $2^h \geq n!$.

2. Inequality:

- $n! \geq \sqrt{2\pi n}$.
- $h \geq \log(n!)$.
- $h \geq n \log(n)$.

3. Maximum Number of Leaves in the Tree:

- If a binary tree has height h, the maximum number of leaves in the tree is 2^h.

Counting Sort Algorithm

- **Working**: Sorts integers within a range. It counts the occurrence of each element to sort them.
- **Time complexity**: $O(n + k)$, where n is the number of elements and k is the range of input values.
- **Definition**: Sorts integers within a specific range by counting occurrences.

Characteristics

- Non-comparison based.
- Assumes input of integers within a small range.
- Utilizes an auxiliary array for determining the position of each element.

```
Counting Sort(A[1 ... n], B[1 ... n],k)
let C[0 ... k] be a new array
for i = 0 to k   // initialize C
   C[i] = 0
   for j = 1 to A.length   //count keys
       C[A[j]] = C[A[j]] + 1
   for i = 1 to k    // accumulate C (determine rank of each key)
       C[i] = C[i] + C[i - 1]
for j = A.length downto 1    // place keys in B
   B[C[A[j]]] = A[j]
   C[A[j]] = C[A[j]] - 1
```

Algorithm Steps

```
Counting Sort(A[1..n], B[1 … n], k)
1. Let C[0 ... k] be a new array
2. Initialize C with zeros
3. Count the occurrence of each element in A and store in C
4. Update C to store the cumulative count
5. Build the sorted array B using C and A
```

Example of Counting Sort

Given: $A = \{3, 2, 4, 3, 0, 4, 3, 1, 2\}$
Step 1:, Find Max and Min for the range:

- Max is 4, Min is 0

Question: What are the contents of array C at different stages?

Step 2: After initialization of the Counting Array C (lines 1-2): $C = \{0, 0, 0, 0, 0\}$,

- The size of the Counting Array is $k + 1$, where k is 4, $C[k + 1] = C[5]$

Step 3: After counting occurrences (lines 3-4): $C = \{1, 1, 2, 3, 2\}$

- $C[0] = 1$, $C[1] = 1$, $C[2] = 2$, $C[3] = 3$, $C[4] = 2$
- The occurrences of the elements of A

Step 4: After cumulative count (line 5): $C = \{1, 2, 4, 7, 9\}$

- **After Cumulative Count (After line 7)**:
  - $C[0] = 1$ (remains the same)
  - $C[1] = C[1] + C[0] = 1 + 1 = 2$
  - $C[2] = C[2] + C[1] = 2 + 2 = 4$
  - $C[3] = C[3] + C[2] = 3 + 4 = 7$
  - $C[4] = C[4] + C[3] = 2 + 7 = 9$

Step 5: Resulting sorted array $B = \{0, 1, 2, 2, 3, 3, 3, 4, 4\}$
Step 5.1: Given the cumulative count array $C = \{1, 2, 4, 7, 9\}$ and the original array $A = \{3, 2, 4, 3, 0, 4, 3, 1, 2\}$,
Step 5.2 (Line 8-10): We iterate over array A from the end to the start.
Step 5.3: For each element $A[j]$ in array A, we do the following:

- Find the cumulative count $C[A[j]]$
- Place $A[j]$ at index $C[A[j]] - 1$ in array B
- Decrease the cumulative count $C[A[j]]$ by 1

Step 5.4:

- **For** $A[8] = 2$ (the last element in the array):
  - $C[2] = 4$, so we place 2 at index $4 - 1 = 3$ in array B: $B[3] = 2$ (0-indexed)
  - We then decrease $C[2]$ to 3: $C = \{1, 2, 3, 7, 9\}$
- **For** $A[7] = 1$:
  - $C[1] = 2$, so we place 1 at index $2 - 1 = 1$ in array B: $B[1] = 1$ (0-indexed)
  - We then decrease $C[1]$ to 1: $C = \{1, 1, 3, 7, 9\}$
- **For** $A[6] = 3$:
  - $C[3] = 7$, so we place 3 at index $7 - 1 = 6$ in array B: $B[6] = 3$ (0-indexed)
  - We then decrease $C[3]$ to 6: $C = \{1, 1, 3, 6, 9\}$

We continue this process for all elements in array A, updating array B and array C as we go. After processing all elements, we will obtain the sorted array B:

$$B = \{0, 1, 2, 2, 3, 3, 3, 4, 4\}$$

Analysis of Counting Sort

- **Time Complexity**: $T(n) = O(n + k)$. If $k = O(n)$, then $T(n) = O(n)$. **
- **Stability**: Counting sort is stable if the auxiliary array is used correctly.
- **In-Place**: Not in-place due to the auxiliary array.
- **Comparisons**: No element comparisons are made.
- Counting sort is stable and works in $O(n + k)$ where k is the range of input.

Reasonable Values for k 合理的 k 值

- Depends on n and memory capacity.
- Avoid large k values to prevent performance issues.
- Suitable for small bit-size numbers (like 4 or 8 bits) but not for larger bit sizes (like 16 or 32 bits).

Application

- The Counting sort is a subroutine in the Radix sort algorithm, a linear-time sorting algorithm.

Example of the Counting Sort 02

Example 02, $A = \{2, 5, 3, 0, 2, 3, 0, 3\}$

Step 1: Identify the Range

- Find the maximum and minimum values in the Input Array.
  - Min: 0, Max: 5

Step 2: Create a Count Array

- The size of Array C is $k + 1 = 5 + 1 = 6$

- The Counting Array $C = \{0, 0, 0, 0, 0, 0\}$
- After processing the first element (2) 處理第一個元素（2）後：
  - $C = \{0, 0, 1, 0, 0, 0\}$

Input Array	2	5	3	0	2	3	0	3
Index[i]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
Counting Array C	2	0	2	3	0	1		
Index[] of C $5 + 1 = 6$	C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	x	x
Cumulated Array C	2	2	4	7	7	8		
Index of C	C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	x	x

Step 3: Calculate Cumulative Counts 計算累積計數

- $C[0]$
  - Remains the same, $C[0] = 2$ 保持不變，$C[0] = 2$
- $C[1]$
  - $C[1] = C[1] + C[0]$
  - $C[1] = 0 + 2 = 2$
- $C[2]$
  - $C[2] = C[2] + C[1]$
  - $C[2] = 2 + 2 = 4$
- $C[3]$
  - $C[3] = C[3] + C[2]$
  - $C[3] = 3 + 4 = 7$
- $C[4]$
  - $C[4] = C[4] + C[3]$
  - $C[4] = 0 + 7 = 7$
- $C[5]$
  - $C[5] = C[5] + C[4]$
  - $C[5] = 1 + 7 = 8$

$$C = \{2, 2, 4, 7, 7, 8\}$$

Step 4: Build the Sorted Array

- Create an output array of the same size as the input array.
- Traverse the input array from end to start, and use the cumulative count array to find the correct position for each element in the output array.

Initial State

```
A = [2, 5, 3, 0, 2, 3, 0, 3]
C = [2, 2, 4, 7, 7, 8]
B = [-1, -1, -1, -1, -1, -1, -1, -1]
```

Input Array	2	5	3	0	2	3	0	3
Index[i]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
Cumulated Array C	2	2	4	7	7	8		
Index of C	C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	x	x
Sorted Array B								
Index[i]	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]

Step 1: Consider $A[7] = 3$

- Find $C[3] = 7$
- Set $B[7-1] = B[6] = 3$
- Decrease $C[3]$ by 1, $C[3] = 6$

```
B = [-1, -1, -1, -1, -1, -1, 3, -1]
C = [2, 2, 4, 6, 7, 8]
```

Step 2: Consider $A[6] = 0$

- Find $C[0] = 2$
- Set $B[2-1] = B[1] = 0$
- Decrease $C[0]$ by 1, $C[0] = 1$

```
B = [-1, 0, -1, -1, -1, -1, 3, -1]
C = [1, 2, 4, 6, 7, 8]
```

Step 3: Consider $A[5] = 3$

- Find $C[3] = 6$
- Set $B[6-1] = B[5] = 3$
- Decrease $C[3]$ by 1, $C[3] = 5$

```
B = [-1, 0, -1, -1, -1, 3, 3, -1]
C = [1, 2, 4, 5, 7, 8]
```

Step 4: Consider $A[4] = 2$

- Find $C[2] = 4$
- Set $B[4-1] = B[3] = 2$
- Decrease $C[2]$ by 1, $C[2] = 3$

```
B = [-1, 0, -1, 2, -1, 3, 3, -1]
C = [1, 2, 3, 5, 7, 8]
```

Counting Sort Algorithm_

- **Best Case**
  - **Scenario**: When all elements are the same.
  - **Time Complexity**: $O(n+k)$, where n is the number of elements and k is the range of input values.
- **Worst Case**
  - **Scenario**: When the elements are evenly distributed across a large range.
  - **Time Complexity**: $O(n+k)$. The worst case is the same as the best case because Counting Sort has a linear time complexity.
- **Average Case**
  - **Scenario**: In general scenarios where the input has a moderate range and distribution.
  - **Time Complexity**: $O(n+k)$. The average case is also linear, making Counting Sort efficient for small to moderate ranges of k.
- In all cases, the space complexity remains $O(n+k)$ because of the auxiliary arrays used in the algorithm.

Radix Sort

- **Working**: Sorts numbers digit by digit from least significant digit (LSD) to most significant digit (MSD) or vice versa.
- **Steps**:
  i. Determine the maximum number to know the number of digits.
  ii. Sort numbers using Counting Sort for each digit.
- **Stable Algorithm**: Maintains the relative order of records with equal keys.
- **Non-comparison based algorithm**: Does not use comparison operators to sort.
- **Time complexity**: $O(d(n+k))$, where d is the number of digits, n is the number of elements, and k is the range of input values.
- Processes each digit of the number.

- Starts from the least significant digit and moves to the most significant.
- Uses a stable sort (often counting sort) to sort each digit.

Analysis of Radix Sort

- Radix sort works in $O(nk)$ for n numbers of k digits.

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

- Use induction on length of input values (i.e., number of digits / bits) to prove this algorithm is correct
- This is left as an exercise

- **Example**: For numbers $170, 45, 75, 90, 802, 24, 2, 66$, sort based on units place, then tens place, then hundreds place.

initial	Step 1	Step 2	Step 3
-,-,-	-,-,↓	-,↓,-	↓,-,-
170	170	2	2
45	90	802	24
75	802	24	45
90	2	45	66
802	24	66	75
24	45	170	90
2	75	75	170
66	66	90	802

- **Example 02**, Array A{329, 457, 657, 839, 436, 720, 355}, to use Radix Sort

initial	Step 1	Step 2	Step 3
-,-,-	-,-,↓	-,↓,-	↓,-,-
329	720	720	329
457	355	329	355

initial	Step 1	Step 2	Step 3
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Complexity Analysis

- **Time Complexity**: $O(nk)$, where n is the number of elements and k is the number of digits in the maximum number.
- **Space Complexity**: $O(n + k)$, as it uses Counting Sort as a subroutine which requires additional space.
- **Stability**: Radix Sort is stable, meaning numbers with the same value appear in the output array in the same order as they do in the input array.

Note

- Radix Sort is efficient when k is not significantly larger than n.
- It's a non-comparative sorting algorithm.
- Best suited for sorting integers or strings.

Was used by the card-sorting machines to read punch cards

- How IBM made its money initially

The key is to sort digit by digit

- Start with least significant digit (or bit)
- Starting with the most significant digit requires extra storage.
- Sorting method used to sort each digit must be stable

1. First sort units digit (or bit)
2. Then sort tens digit (or twos bit)
3. Continue to most significant

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

- Use induction on length of input values (i.e., number of digits / bits) to prove this algorithm is correct

Bucket Sort

- **Working**: Distributes elements into buckets and then sorts each bucket individually.
- **Time complexity**: $O(n)$ average time under uniform distribution, but $O(n^2)$ in the worst case.
- Divides the interval of input numbers into equal-sized buckets.
- Distributes the numbers into buckets.
- Sorts each bucket and then gathers numbers back.

Analysis of Algorithms:

- Bucket sort assumes that input is uniformly distributed to achieve linear time.
- Worst Case is all numbers in one bucket Cost is $O(n^2)$

Limitations

- Counting sort is not suitable for sorting strings of varying length.
- Radix and bucket sort are not comparison-based and hence might not be suitable for all types of data.

Bucket Example:

- ***A[i] are the value of each element,***
  and n is the A.length which is 10 in the example

Index(A)	A	⌊A[i] × n⌋	B	Sorted
A[0]	.38	.38×10=⌊3.8⌋=3	0	
A[1]	.93	.93×10=⌊9.3⌋=9	1	.11
A[2]	.77	.77×10=⌊7.7⌋=7	2	.29
A[3]	.11	.11×10=⌊1.1⌋=1	3	.38 → .39
A[4]	.95	.95×10=⌊9.5⌋=9	4	.43
A[5]	.29	.29×10=⌊2.9⌋=2	5	.95
A[6]	.72	.72×10=⌊7.2⌋=7	6	

Index(A)	A	⌊A[i] × n⌋	B	Sorted
A[7]	.43	.43×10=⌊4.3⌋=4	7	.77 → .72
A[8]	.39	.39×10=⌊3.9⌋=3	8	
A[9]	.99	.99×10=⌊9.9⌋=9	9	.93 → .95 → .99

Analysis of the Average Run Time of Bucket Sort Algorithm

```
BUCKET-SORT (A)
1. n = A.length
2. let B[0 ... n-1] be a new array
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[⌊n·A[i]⌋]
7. for i= 0 to n-1
8.    sort list B[i] with insertion sort
// Cost is just O(c) constant
9. concatenate the lists B[0], B[1],
..., [n-1] together in order
```

All lines except line 8 (sort bucket) takes $O(n)$ time in the worst case
To analyze the average run time, we need to find $E[T(n)]$, which is given by:

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

Where n_i is the number of elements in bucket $B[i]$.

Step 1: Define $n_i

Define n_i as the number of elements in bucket i. It can be expressed in terms of X_{ij} as:

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

This equation means that we sum up all the X_{ij} for a given i, where X_{ij} is 1 if $A[j]$ falls in bucket i and 0 otherwise.

Step 2: Express $E[n_i^2]$

We aim to find $E[n_i^2]$, which can be expressed as:

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

Step 3: Expand the Square

Expanding the square gives:

$$E[n_i^2] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$

Step 4: Separate the Expectation

We can separate the expectation over the sum:

$$E[n_i^2] = \sum_{j=1}^{n}\sum_{k=1}^{n} E[X_{ij}X_{ik}]$$

Step 5: Find $E[X_{ij}X_{ik}]$

Now we need to find $E[X_{ij}X_{ik}]$ for two cases: when $j = k$ and when $j \neq k$.

1. When $j = k$:

$$E[X_{ij}X_{ik}] = E[X_{ij}^2]$$

   But since X_{ij} is a Bernoulli variable (it can only take values 0 or 1), $X_{ij}^2 = X_{ij}$. So,

$$E[X_{ij}^2] = E[X_{ij}] = \frac{1}{n}$$

2. When $j \neq k$:
   The events are independent, so

$$E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

Step 6: Substitute the Values Back
Now we substitute these values back into the equation for $E[n_i^2]$:

$$E[n_i^2] = \sum_{j=1}^{n}\sum_{k=1}^{n} E[X_{ij}X_{ik}]$$

This sum can be broken down into two parts: when $j = k$ and when $j \neq k$.

So we have:

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n}\sum_{k \neq j} E[X_{ij}X_{ik}]$$

Now substituting the values we found in step 5:

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{k \neq j} \frac{1}{n^2}$$

Step 7: Calculate the Sums

Now we calculate each sum separately:

1. $\sum_{j=1}^{n} \frac{1}{n} = 1$
2. $\sum_{j=1}^{n} \sum_{k \neq j} \frac{1}{n^2} = n(n-1)\frac{1}{n^2}$

Step 8: Final Calculation

Now we add these two results together to find $E[n_i^2]$:

$$E[n_i^2] = 1 + n(n-1)\frac{1}{n^2} = 1 + (n-1)\frac{1}{n} = 2 - \frac{1}{n}$$

Now we have found $E[n_i^2]$ as $2 - \frac{1}{n}$.

$$E[n_i^2] = 1 + n(n-1)\frac{1}{n^2} = 1 + (n-1)\frac{1}{n} = 2 - \frac{1}{n}$$

Let $n_i =$ number of elements in bucket $B[i]$

We are looking to find the expected time $E[T(n)]$, which is given by:

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

Substituting the values we found for $E[X_{ij}X_{ik}]$ when $j = k$ and $j \neq k$, we find:

$$E[n_i^2] = n\left(\frac{1}{n}\right) + n(n-1)\left(\frac{1}{n^2}\right) = 2 - \frac{1}{n}$$

So, we can now find $E[T(n)]$ using the formula we derived earlier:

$$T(n) = \Theta(n) + O\left(n\left(2 - \frac{1}{n}\right)\right) = \Theta(n) + O(n) = \Theta(n)$$

This shows that the average running time of the bucket sort algorithm is linear.

Conclusion

This analysis shows that the average running time of the bucket sort algorithm is linear, $\Theta(n)$.