**TUGAS EKSPLORASI MANDIRI**

**“Sorting By Counting”**

**2023234300048**



**Dosen Pengampu:**

**Randi Proska Sandra, S.Pd.,M.Sc.**

**Oleh:**

**Adelya Destriana Putri**

**22343033**

**PRODI INFORMATIKA**

**DEPARTEMEN ELEKTRONIKA**

**FAKULTAS TEKNIK**

**UNIVERSITAS NEGERI PADANG**

**2024**

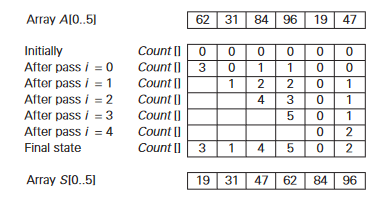
1. **Penjelasan Sorting By Counting**

Sorting by Counting, atau dalam Bahasa Indonesia dikenal sebagai pengurutan dengan penghitungan, adalah teknik pengurutan yang bekerja dengan cara menghitung jumlah kemunculan dari setiap elemen dalam array. Teknik ini sangat efisien untuk mengurutkan data di mana rentang nilai elemennya tidak terlalu besar dan diketahui. Salah satu implementasi umum dari ide ini adalah Counting Sort.

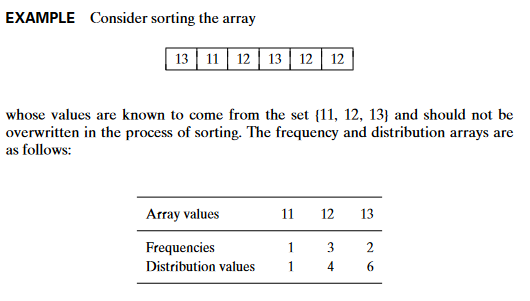
Langkah-langkahnya mencakup:

1. **Penentuan Rentang Nilai**: Identifikasi nilai minimum dan maksimum dalam array untuk mengetahui rentang nilai yang akan diurutkan.
2. **Inisialisasi Array Penghitung:** Inisialisasi array penghitung dengan ukuran yang sesuai berdasarkan rentang nilai, dan semua elemennya diatur ke 0.
3. **Menghitung Frekuensi Kemunculan:** Menghitung berapa kali setiap elemen muncul dalam array.
4. **Menghitung Distribusi Kumulatif:** Menambahkan frekuensi kemunculan sehingga kita mendapatkan distribusi kumulatif, yang menentukan posisi akhir dari setiap elemen dalam array yang diurutkan.
5. **Menempatkan Elemen pada Posisi yang Benar:** Memindahkan setiap elemen dari array asli ke array output sesuai dengan posisi yang diberikan oleh distribusi kumulatif.

Sebagai contoh pertama penerapan teknik peningkatan masukan, kita membahasnya aplikasi untuk masalah penyortiran. Salah satu gagasan yang cukup jelas adalah menghitung, untuk masing-masingnya elemen daftar yang akan diurutkan, jumlah total elemen lebih kecil dari ini elemen dan catat hasilnya dalam tabel. Angka-angka ini akan menunjukkan posisinya elemen dalam daftar yang diurutkan: misalnya, jika hitungannya adalah 10 untuk beberapa elemen, maka seharusnya demikian berada di posisi ke-11 (dengan indeks 10, jika kita mulai menghitung dengan 0) dalam pengurutan Himpunan. Dengan demikian, kita dapat mengurutkan daftar hanya dengan menyalin elemen-elemennya ke dalamnya posisi yang sesuai dalam daftar baru yang diurutkan. Algoritma ini disebut perbandingan jenis penghitungan (Gambar 7.1).



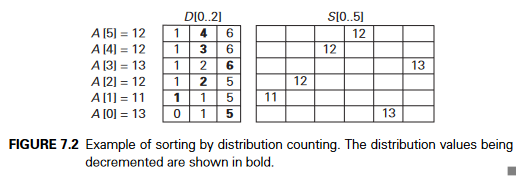
Gambar 7.1



Perhatikan bahwa nilai distribusi menunjukkan posisi yang tepat untuk kejadian terakhir. rences elemen-elemennya dalam array yang diurutkan terakhir. Jika kita mengindeks posisi array dari 0

ke n − 1, nilai distribusi harus dikurangi 1 untuk mendapatkan elemen yang sesuai posisi.

Akan lebih mudah untuk memproses array input dari kanan ke kiri. Misalnya, elemen terakhir adalah 12, dan karena nilai distribusinya adalah 4, kita tempatkan 12 ini di posisi 4 − 1 = 3 dari array S yang akan menampung daftar yang diurutkan. Lalu kita turunkan nilai distribusi 12 dengan 1 dan lanjutkan ke elemen berikutnya (dari kanan). dalam larik yang diberikan. Seluruh pemrosesan contoh ini digambarkan pada Gambar 7.2.



1. **Pseudocode Sorting By Counting**

ALGORITHM ComparisonCountingSort(A[0..n − 1])

//Sorts an array by comparison counting

//Input: An array A[0..n − 1] of orderable elements

//Output: Array S[0..n − 1] of A’s elements sorted in nondecreasing order

for i ← 0 to n − 1 do Count[i] ← 0

for i ← 0 to n − 2 do

for j ← i + 1 to n − 1 do

if A[i] < A[j ]

Count[j ] ← Count[j ] + 1

else Count[i] ← Count[i] + 1

for i ← 0 to n − 1 do S[Count[i]] ← A[i]

return S

Berapa efisiensi waktu dari algoritma ini? Itu harus kuadrat karena algoritma ini mempertimbangkan semua pasangan berbeda dari array n-elemen. Secara lebih formal, berapa kali operasi dasarnya, perbandingan A[i] < A[j ], dijalankan sama dengan jumlah yang telah kita temui beberapa kali:



Dengan demikian, algoritme membuat jumlah perbandingan kunci yang sama dengan pengurutan pilihan dan sebagai tambahan menggunakan sejumlah ruang ekstra linier.

ALGORITHM DistributionCountingSort(A[0..n − 1], l, u)

//Sorts an array of integers from a limited range by distribution counting

//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)

//Output: Array S[0..n − 1] of A’s elements sorted in nondecreasing order

for j ← 0 to u − l do D[j ] ← 0 //initialize frequencies

for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1 //compute frequencies

for j ← 1 to u − l do D[j ] ← D[j − 1] + D[j ] //reuse for distribution

for i ← n − 1 downto 0 do

j ← A[i] − l

S[D[j ] − 1] ← A[i]

D[j ] ← D[j ] − 1

return S

Dengan asumsi bahwa kisaran nilai array tetap, ini jelas merupakan linear algoritma karena hanya membuat dua lintasan berturut-turut melalui larik masukannya A. Ini adalah kelas efisiensi waktu yang lebih baik dibandingkan kelas penyortiran yang paling efisien algoritma—mergesort, quicksort, dan heapsort—yang pernah kami temui. Ini penting-Namun penting untuk diingat bahwa efisiensi ini diperoleh dengan mengeksploitasi sifat spesifik dari input yang dapat diurutkan berdasarkan penghitungan distribusi, selain itu pertukaran ruang dengan waktu.\

1. **Program Sorting By Counting**

**ALGORITHM ComparisonCountingSort(A[0..n − 1])**

print("===============================================")

print("==========Program Sorting By Counting==========")

print("==Program\_by\_22343033\_Adelya Destriana Putri==")

print("===============================================")

def counting\_sort(array):

n = len(array)

# Inisialisasi array Count

count = [0] \* n

# Menghitung frekuensi kemunculan setiap elemen

for i in range(n - 1):

for j in range(i + 1, n):

if array[i] < array[j]:

count[j] += 1

else:

count[i] += 1

# Membuat array output

output = [0] \* n

for i in range(n):

output[count[i]] = array[i]

return output

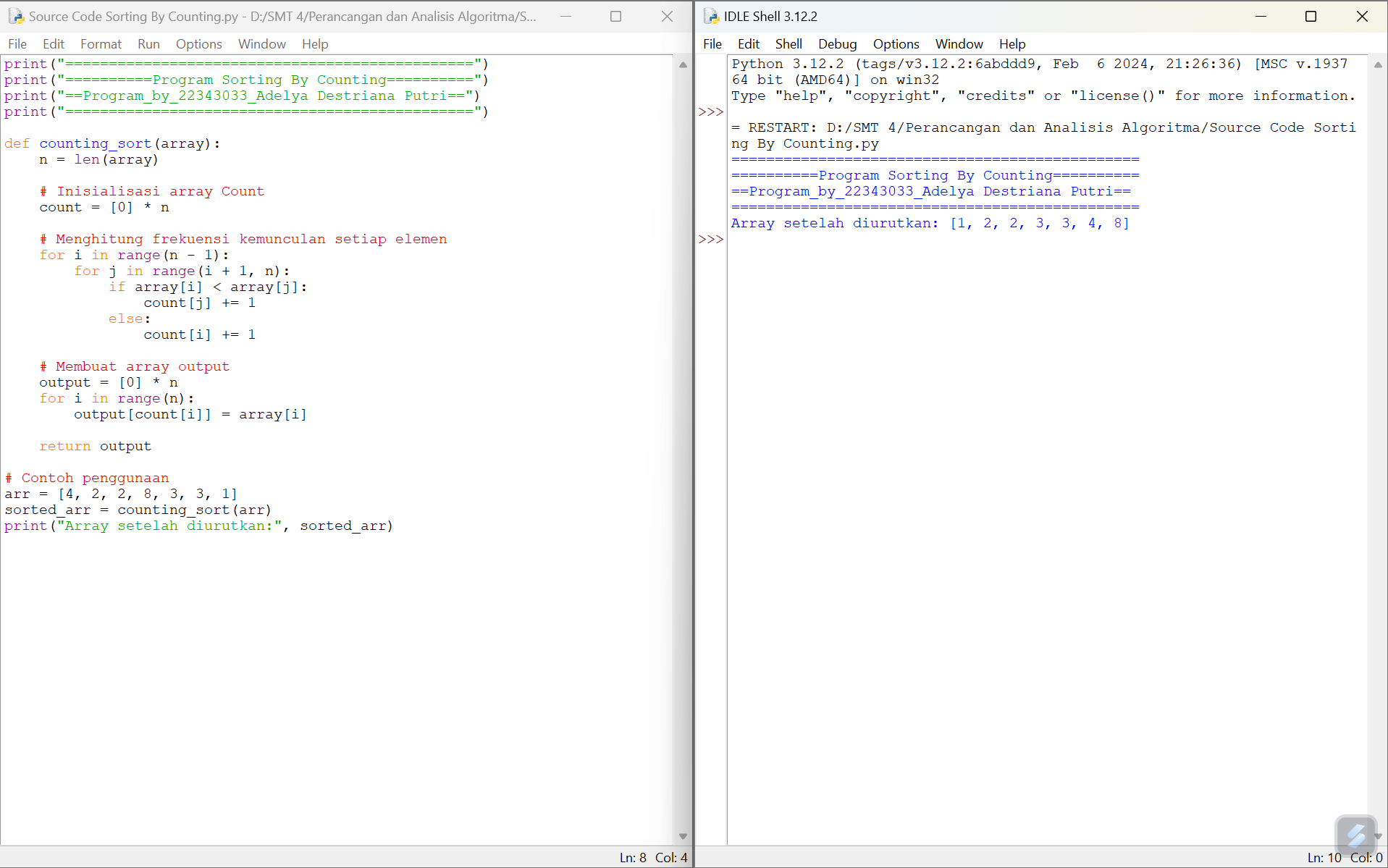
# Contoh penggunaan

arr = [4, 2, 2, 8, 3, 3, 1]

sorted\_arr = counting\_sort(arr)

print("Array setelah diurutkan:", sorted\_arr)

**Screenshot ALGORITHM ComparisonCountingSort(A[0..n − 1])**



**ALGORITHM DistributionCountingSort(A[0..n − 1], l, u)**

print("===============================================")

print("==========Program Sorting By Counting==========")

print("==Program\_by\_22343033\_Adelya Destriana Putri==")

print("===============================================")

def distribution\_counting\_sort(array, l, u):

n = len(array)

# Inisialisasi array D

D = [0] \* (u - l + 1)

# Menghitung frekuensi kemunculan setiap elemen

for i in range(n):

D[array[i] - l] += 1

# Menghitung distribusi kumulatif

for j in range(1, u - l + 1):

D[j] += D[j - 1]

# Membuat array output

output = [0] \* n

for i in range(n - 1, -1, -1):

j = array[i] - l

output[D[j] - 1] = array[i]

D[j] -= 1

return output

# Contoh penggunaan

arr = [4, 2, 2, 8, 3, 3, 1]

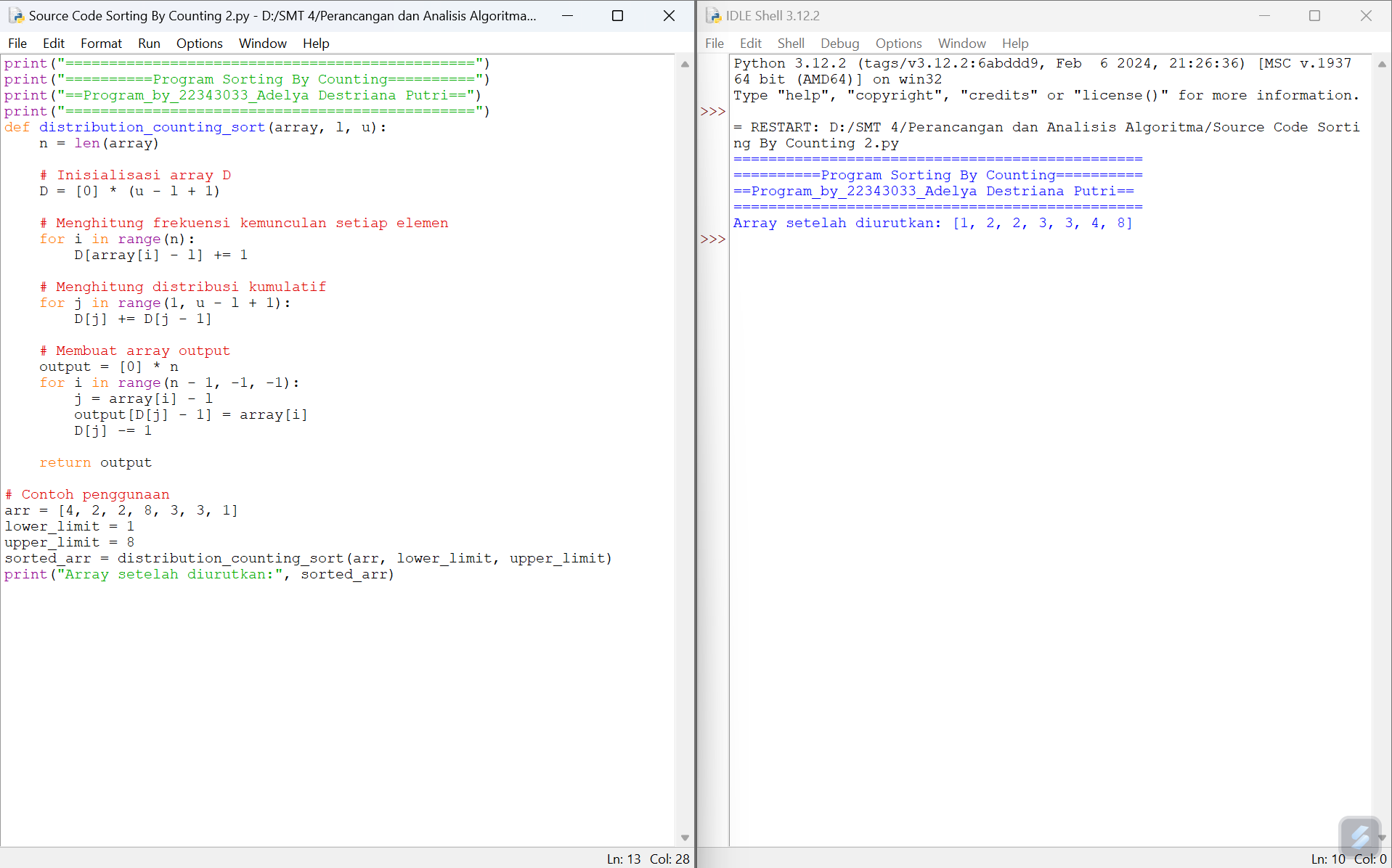
lower\_limit = 1

upper\_limit = 8

sorted\_arr = distribution\_counting\_sort(arr, lower\_limit, upper\_limit)

print("Array setelah diurutkan:", sorted\_arr)

**Screenshot ALGORITHM DistributionCountingSort(A[0..n − 1], l, u)**



1. **Analisis Kebutuhan Waktu**

Analisis Kebutuhan Waktu Program Sorting By Counting:

**Counting Sort:**

* Inisialisasi Array Count: Membutuhkan waktu konstan O(n) untuk menginisialisasi array Count dengan panjang yang sama dengan jumlah elemen dalam array.
* Menghitung Frekuensi Kemunculan: Dalam algoritma ini, terdapat loop bersarang yang mencakup iterasi sebanyak  kali, di mana n adalah jumlah elemen dalam array. Ini disebabkan oleh iterasi melalui setiap elemen dalam array dan membandingkannya dengan setiap elemen lainnya. Oleh karena itu, kompleksitas waktu untuk langkah ini adalah O(n^2).
* Menempatkan Elemen pada Posisi yang Benar: Setelah menghitung frekuensi kemunculan, langkah ini melibatkan iterasi melalui seluruh array untuk menempatkan elemen pada posisi yang benar berdasarkan informasi frekuensi kemunculan. Ini memerlukan waktu O(n).

Dengan demikian, kompleksitas waktu total Counting Sort adalah O(n^2), yang merupakan kompleksitas yang cukup tinggi terutama untuk dataset yang besar.

**Distribution Counting Sort:**

* Inisialisasi Array D: Memerlukan waktu konstan O(u - l + 1) untuk menginisialisasi array D dengan panjang yang sama dengan rentang nilai dari elemen-elemen dalam array.
* Menghitung Frekuensi Kemunculan: Memerlukan waktu O(n) untuk mengiterasi melalui array input dan meningkatkan frekuensi kemunculan setiap elemen dalam array D.
* Menghitung Distribusi Kumulatif: Memerlukan waktu O(u - l) untuk menghitung distribusi kumulatif dari array D.
* Menempatkan Elemen pada Posisi yang Benar: Setelah menghitung distribusi kumulatif, langkah ini melibatkan iterasi melalui seluruh array (dari belakang ke depan) untuk menempatkan elemen pada posisi yang benar dalam array output berdasarkan informasi distribusi kumulatif. Ini memerlukan waktu O(n).

Dengan demikian, kompleksitas waktu total Distribution Counting Sort adalah O(n + u - l ), yang jauh lebih efisien daripada Counting Sort untuk dataset dengan rentang nilai yang terbatas.

1. **Referensi**

Levitin, A. (2011). Introduction to the Design . Anany Levitin. — 3rd ed.