COUNTING SORT

DOSEN PEMBIMBING:

Randi Proska Sandra, M.Sc.

DISUSUN OLEH:

DZAKI SULTAN RABBANI (23343035)

UNIVERSITAS NEGERI PADANG

FAKULTAS TEKNIK

PRODI INFORMATIKA

SEPTEMBER 2025

1. PENJELASAN ALGORITMA

Dalam dunia ilmu komputer, algoritma pengurutan memainkan peran penting dalam mengatur data secara efisien untuk berbagai aplikasi. Algoritma pengurutan tradisional, seperti merge sort dan quick sort, mengandalkan perbandingan antar elemen untuk mencapai urutan. Namun, counting sort menawarkan pendekatan yang berbeda, yang memungkinkannya untuk melampaui batas algoritma berbasis perbandingan dalam skenario tertentu. Counting sort adalah algoritma pengurutan non-comparison-based yang mengurutkan elemen integer dengan memanfaatkan rentang nilai elemen. Berbeda dengan algoritma comparison-based seperti quicksort yang bergantung pada perbandingan elemen (O(n log n)), counting sort mencapai kompleksitas waktu O(n + k), di mana n adalah jumlah elemen dan k adalah rentang nilai (maksimum - minimum + 1). Algoritma ini ideal untuk data dengan rentang nilai terbatas, seperti nilai ujian (0–100) atau usia penduduk (1–120).

Algoritma ini ditemukan oleh Harold Seward pada tahun 1954 dan telah menjadi landasan dalam ilmu komputer karena kesederhanaan dan efisiensinya. Tidak seperti algoritma pengurutan berbasis perbandingan seperti bubble sort dan merge sort, counting sort beroperasi tanpa membuat perbandingan langsung antar elemen. Sebagai gantinya, ia menggunakan teknik penghitungan untuk menentukan posisi setiap elemen dalam array yang diurutkan. Algoritma ini sangat efisien ketika berhadapan dengan integer dalam rentang yang terbatas, menjadikannya pilihan yang sangat baik untuk berbagai aplikasi.

Prinsip Utama Algoritma

Counting sort didasarkan pada prinsip menghitung kemunculan setiap elemen unik dalam array masukan. Kemudian, ia menggunakan informasi ini untuk menentukan posisi setiap elemen dalam array yang diurutkan. Algoritma ini terdiri dari empat langkah utama:

1. Menentukan Rentang Nilai: Langkah pertama adalah menentukan rentang nilai dalam array masukan. Ini melibatkan pencarian nilai maksimum dan minimum dalam array. Rentang ini digunakan untuk membuat array bantu yang disebut array hitung. Jika nilai negatif ada dalam array masukan, offset dihitung untuk memastikan bahwa semua indeks dalam array hitung bersifat non-negatif.
2. Membuat Array Hitung: Array hitung digunakan untuk menyimpan frekuensi setiap elemen unik dalam array masukan. Ukuran array hitung sama dengan rentang nilai yang ditentukan pada langkah sebelumnya. Setiap indeks dalam array hitung sesuai dengan nilai dalam array masukan, dan nilai yang disimpan pada setiap indeks menunjukkan frekuensi nilai tersebut.
3. Menghitung Jumlah Kumulatif: Langkah selanjutnya adalah menghitung jumlah kumulatif elemen dalam array hitung. Jumlah kumulatif pada setiap indeks menunjukkan posisi elemen yang sesuai dalam array yang diurutkan. Transformasi ini

memungkinkan algoritma untuk menempatkan elemen secara langsung ke posisi yang benar dalam array keluaran6.

4. Membangun Array Hasil: Langkah terakhir adalah membangun array hasil dengan menempatkan setiap elemen ke posisi yang benar berdasarkan jumlah kumulatif yang dihitung pada langkah sebelumnya. Proses ini dilakukan dengan mengulangi array masukan dalam urutan terbalik dan menempatkan setiap elemen pada posisi yang sesuai dalam array hasil. Urutan terbalik memastikan bahwa algoritma stabil, artinya algoritma mempertahankan urutan relatif elemen yang sama.

2. PSEUDOCODE

```
FUNCTION CountingSort(arr)
    max_val ← Cari nilai maksimum dari arr
    min_val ← Cari nilai minimum dari arr
    rentang ← max_val - min_val + 1
    hitung ← Array berisi 0 sepanjang rentang
    hasil ← Array kosong sepanjang arr

    FOR i ← 0 KE panjang(arr) - 1
        offset ← arr[i] - min_val
        hitung[offset] ← hitung[offset] + 1

    FOR i ← 1 KE rentang - 1
        hitung[i] ← hitung[i] + hitung[i - 1]

    FOR i ← panjang(arr) - 1 KE 0 TURUN
        offset ← arr[i] - min_val
        posisi ← hitung[offset] - 1
        hasil[posisi] ← arr[i]
        hitung[offset] ← hitung[offset] - 1

    RETURN hasil
```

Dalam pseudocode ini:

- arr adalah array masukan yang akan diurutkan.
- max_val adalah nilai maksimum dalam array masukan.
- min_val adalah nilai minimum dalam array masukan.
- rentang adalah rentang nilai dalam array masukan, dihitung sebagai max_val - min_val + 1.
- hitung adalah array bantu yang digunakan untuk menyimpan frekuensi setiap elemen dalam array masukan.
- hasil adalah array yang diurutkan yang akan dikembalikan oleh algoritma

3. SOURCE CODE

```
def sortir_perhitungan(arr):
    if not arr:
        return []

    nilai_maks = max(arr)
    nilai_min = min(arr)
    rentang = nilai_maks - nilai_min + 1
    hitung = [0] * rentang
    hasil = [0] * len(arr)

    for angka in arr:
        offset = angka - nilai_min
        hitung[offset] += 1

    for i in range(1, rentang):
        hitung[i] += hitung[i-1]

    for i in range(len(arr)-1, -1, -1):
        offset = arr[i] - nilai_min
        posisi = hitung[offset] - 1
        hasil[posisi] = arr[i]
        hitung[offset] -= 1

    return hasil

# Contoh penggunaan
contoh_data = [3, -1, 5, 3, 0, 8, -2]
print(sortir_perhitungan(contoh_data))
```

4. ANALISIS KEBUTUHAN WAKTU

a) **Analisis Operasi *Assignment* dan Aritmatika**

Algoritma counting sort terdiri dari beberapa tahap, masing-masing dengan serangkaian operasi tertentu. Dengan menganalisis jumlah operasi assignment dan aritmatika di setiap tahap, kita dapat memperoleh pemahaman komprehensif tentang kompleksitas waktu algoritma.

- Tahap 1 (Mencari Maks/Min): Tahap ini melibatkan pencarian nilai maksimum dan minimum dalam array masukan. Jumlah operasi perbandingan yang diperlukan untuk tahap ini adalah 2n, di mana n adalah jumlah elemen dalam array.
- Tahap 2 (Menghitung Frekuensi): Tahap ini melibatkan iterasi melalui array masukan dan perhitungan frekuensi setiap elemen. Setiap iterasi memerlukan satu operasi assignment (offset = arr[i] - min) dan satu operasi aritmatika

(hitung[offset] += 1)3. Oleh karena itu, jumlah total operasi untuk tahap ini adalah n (operasi assignment) + n (operasi aritmatika) = 2n.

- Tahap 3 (Kumulatif): Tahap ini melibatkan perhitungan jumlah kumulatif elemen dalam array hitung. Setiap iterasi memerlukan satu operasi aritmatika (hitung[i] += hitung[i-1]). Jumlah iterasi untuk tahap ini adalah k-1, di mana k adalah rentang nilai dalam array masukan. Oleh karena itu, jumlah total operasi untuk tahap ini adalah k-1.
- Tahap 4 (Membangun Hasil): Tahap ini melibatkan iterasi melalui array masukan dalam urutan terbalik dan penempatan setiap elemen ke posisi yang benar dalam array hasil. Setiap iterasi memerlukan satu operasi assignment (offset = arr[i] - min), satu operasi aritmatika (posisi = hitung[offset] - 1), satu operasi assignment (hasil[posisi] = arr[i]), dan satu operasi decrement (hitung[offset] -= 1). Oleh karena itu, jumlah total operasi untuk tahap ini adalah n (operasi assignment) + n (operasi aritmatika) + n (operasi assignment) + n (operasi decrement) = 4n.

Jumlah total operasi untuk algoritma counting sort dihitung sebagai berikut:

- Operasi assignment: 2n (tahap 1) + n (tahap 2) + n (tahap 4) = 4n
- Operasi aritmatika: n (tahap 2) + (k-1) (tahap 3) + 2n (tahap 4) = 3n + k - 1

Oleh karena itu, jumlah total operasi adalah 4n + 3n + k - 1 = 7n + k - 1, yang menunjukkan bahwa kompleksitas waktu counting sort adalah O(n + k).

b) **Analisis Operasi Abstrak**

Selain menganalisis operasi assignment dan aritmatika, kita juga dapat menganalisis efisiensi counting sort dengan mempertimbangkan jumlah operasi abstrak yang dilakukan oleh algoritma. Operasi abstrak adalah operasi tingkat tinggi yang mencirikan perilaku algoritma.

- Perulangan Frekuensi: Operasi abstrak pertama adalah perulangan frekuensi, yang melibatkan iterasi melalui array masukan dan perhitungan frekuensi setiap elemen. Jumlah operasi untuk tahap ini adalah O(n), di mana n adalah jumlah elemen dalam array.
- Perulangan Kumulatif: Operasi abstrak kedua adalah perulangan kumulatif, yang melibatkan perhitungan jumlah kumulatif elemen dalam array hitung. Jumlah operasi untuk tahap ini adalah O(k), di mana k adalah rentang nilai dalam array masukan.
- Perulangan Penempatan: Operasi abstrak ketiga adalah perulangan penempatan, yang melibatkan iterasi melalui array masukan dalam urutan terbalik dan penempatan setiap elemen ke posisi yang benar dalam array hasil. Jumlah operasi untuk tahap ini adalah O(n), di mana n adalah jumlah elemen dalam array.

Oleh karena itu, jumlah total operasi abstrak untuk algoritma counting sort adalah O(n) + O(k) + O(n) = O(n + k), yang menegaskan bahwa kompleksitas waktu counting sort adalah O(n + k).

c) **Analisis Best-Case, Worst-Case, Average-Case**
Kompleksitas waktu algoritma counting sort juga dapat dianalisis dengan mempertimbangkan skenario best-case, worst-case, dan average-case.

- Best-Case: Skenario best-case untuk counting sort terjadi ketika data sudah diurutkan5. Dalam skenario ini, algoritma tetap harus melakukan semua tahap, tetapi jumlah operasi yang dilakukan diminimalkan. Kompleksitas waktu untuk skenario best-case adalah O(n + k), karena algoritma tetap harus mengulangi array masukan dan array hitung.
- Worst-Case: Skenario worst-case untuk counting sort terjadi ketika rentang nilai (k) sangat besar. Dalam skenario ini, kompleksitas waktu algoritma didominasi oleh ukuran array hitung, yang merupakan O(k). Oleh karena itu, kompleksitas waktu untuk skenario worst-case adalah O(n + k), tetapi algoritma menjadi tidak efisien karena dominasi k.
- Average-Case: Skenario average-case untuk counting sort terjadi ketika nilai-nilai dalam array masukan didistribusikan secara merata. Dalam skenario ini, kompleksitas waktu algoritma adalah O(n + k), karena algoritma harus mengulangi array masukan dan array hitung.

Tabel berikut meringkas kompleksitas waktu algoritma *counting sort* untuk berbagai skenario:

Skenario	Kompleksitas Waktu
Best-Case	O(n + k)
Worst-Case	O(n + k)
Average-Case	O(n + k)

5. DAFTAR REFERENSI

- Levitin, A. (2002). Introduction to the Design and Analysis of Algorithms (3rd ed.). Pearson.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
- Seward, H. (1954). “A Technique for High-Speed Sorting”. Communications of the ACM, 7(7), 419–420

- Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.
- GeeksforGeeks; Tutorialspoint; Wikipedia. “Counting Sort”. (Sumber daring diakses Maret 2025).

6. LINK GITHUB