# TUGAS PERANCANGAN DAN ANALISIS ALGORITMA
# BRUTE-FORCE STRING MATCHING


**DOSEN PENGAMPU:**

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

**OLEH:**

**Hasanul Fikri**

**23343040**

**PROGRAM STUDI INFORMATIKA**

**DEPARTEMEN ELEKTRONIKA**

**FAKULTAS TEKNIK**

**UNIVERSITAS NEGERI PADANG**

**2025**

## A. Penjelasan Algoritma Brute Force String Matching

Brute force string matching adalah algoritma yang digunakan untuk mencari kecocokan substring dalam sebuah teks dengan cara yang sangat sederhana dan langsung. Algoritma ini bekerja dengan membandingkan setiap karakter dalam pola (pattern) dengan substring yang terletak pada posisi yang berurutan dalam teks. Jika ditemukan kecocokan pada semua karakter dalam pola, maka algoritma akan mengembalikan posisi indeks substring yang cocok tersebut. Sebaliknya, jika ada karakter yang tidak cocok, pola akan digeser satu posisi ke kanan dan pencocokan dimulai kembali dari karakter pertama pola. Proses ini berlanjut hingga seluruh teks selesai diperiksa atau kecocokan ditemukan.

Secara rinci, algoritma ini pertama kali menyelaraskan pola dengan posisi pertama dalam teks, lalu membandingkan setiap karakter dalam pola dengan karakter yang sesuai dalam teks. Jika terjadi ketidaksesuaian, pola digeser satu posisi dan proses perbandingan diulang. Karena algoritma ini tidak menggunakan teknik penyaringan atau optimasi, kecepatannya tergolong lambat dengan kompleksitas waktu O(nm), di mana n adalah panjang teks dan m adalah panjang pola. Walaupun demikian, dalam kasus pencarian teks acak, algoritma ini dapat berjalan lebih cepat dari yang diperkirakan, meskipun tidak optimal jika dibandingkan dengan algoritma pencocokan string yang lebih canggih seperti Knuth-Morris-Pratt atau Boyer-Moore.

Algoritma Brute Force String Matching adalah pendekatan yang sangat sederhana untuk mencari kecocokan substring dalam sebuah teks. Algoritma ini mencocokkan pola dengan setiap substring teks, bergerak satu posisi ke kanan jika terjadi ketidaksesuaian. Meskipun mudah diterapkan, algoritma ini memiliki kelemahan utama dalam hal efisiensi dengan kompleksitas O(nm). Sebagai hasilnya, algoritma ini cocok digunakan untuk teks pendek atau ketika kecepatan bukanlah masalah utama, namun untuk teks yang lebih besar, algoritma yang lebih efisien seperti Boyer-Moore lebih disarankan.

## B. Pseudocode Algoritma Brute Force String Matching

```
ALGORITHM BruteForceStringMatch(T[0..n-1], P[0..m-1])

// Input: T - teks dengan n karakter, P - pola dengan m karakter

// Output: Indeks substring pertama yang cocok atau -1 jika tidak ada kecocokan

for i ← 0 to n - m

do

    j ← 0

    while j < m and P[j] = T[i + j]

do
```

j ← j + 1

if j = m then

return i  // Ditemukan kecocokan pada indeks i

return -1  // Tidak ditemukan kecocokan

## C. Source Code Python

```python
def brute_force_string_match(teks, pola):
    n = len(teks)
    m = len(pola)

    # Melakukan pencocokan dengan pergeseran pola
    for i in range(n - m + 1):
        j = 0
        while j < m and teks[i + j] == pola[j]:
            j += 1
        if j == m:  # Jika seluruh pola cocok
            return i  # Mengembalikan posisi kecocokan pertama
    return -1  # Jika tidak ditemukan kecocokan

# Contoh penggunaan
teks = "INI CONTOH SIMPEL DARI YANG PALING SIMPEL"
pola = "SIMPEL"
hasil = brute_force_string_match(teks, pola)
print("pola ditemukan di indeks ke:", hasil)
```

## D. Analisis Kebutuhan Waktu

### 1. Analisis menyeluruh

```
def brute_force_string_match(teks, pola):
    n = len(teks)   # 1 operasi untuk mendapatkan panjang teks (O(1))
    m = len(pola)   # 1 operasi untuk mendapatkan panjang pola (O(1))

    # Melakukan pencocokan dengan pergeseran pola
    for i in range(n - m + 1):   # Loop (O(n))
        j = 0   # 1 operasi untuk set j = 0 (O(1))
        while j < m and teks[i + j] == pola[j]:   # Loop dalam (O(m) dalam kasus terburuk per i)
            j += 1   # 1 operasi per perbandingan cocok (O(1))
        if j == m:   # 1 operasi per pengecekan kecocokan seluruh pola (O(1))
            return i   # Mengembalikan posisi kecocokan pertama (O(1))
    return -1   # Jika tidak ditemukan kecocokan (O(1))

# Contoh penggunaan
teks = "INI CONTOH SIMPEL DARI YANG PALING SIMPEL"
pola = "SIMPEL"
hasil = brute_force_string_match(teks, pola)
print("pola ditemukan di indeks ke:", hasil)
```

Total Kebutuhan Waktu:

- Best-Case: Jika pola ditemukan pada perbandingan pertama, waktu yang dibutuhkan hanya O(1).
- Worst-Case: Jika tidak ada kecocokan dan kita harus membandingkan setiap karakter, maka total waktu adalah O(n * m).

**2. Analisis berdasarkan operasi abstrak**

Untuk analisis ini, kita fokus pada jumlah operasi abstrak yang dilakukan oleh algoritma, tanpa memperhatikan detail implementasi seperti pergeseran indeks. Operasi utama dalam algoritma ini adalah:

- Perbandingan karakter: Setiap kali kita memeriksa apakah karakter dalam pola cocok dengan karakter dalam teks, ini dihitung sebagai satu operasi per perbandingan.
- Pergeseran indeks: Ketika dua karakter cocok, kita menggeser indeks **j** untuk melanjutkan perbandingan pola.
- Pengecekan kecocokan: Setelah selesai dengan perbandingan, kita memeriksa apakah seluruh pola cocok dengan substring teks.

Jumlah operasi perbandingan dalam kasus terburuk adalah:

- Loop luar sendiri melakukan iterasi sebanyak **n - m + 1** kali. Secara kasar, ini kira-kira **O(n)**
- Jumlah perbandingan dalam loop dalam: sekitar **m** per iterasi

Oleh karena itu, jumlah total perbandingan dalam kasus terburuk adalah **O(n * m).**

**3. Analisis berdasarkan kasus terbaik, rata-rata, dan terburuk**

Pada **kasus terbaik** (best-case) dari algoritma Brute Force String Matching, pola ditemukan pada perbandingan pertama di setiap posisi yang diuji dalam teks. Dalam situasi ini, algoritma akan memeriksa karakter pertama dari pola dan substring teks pada posisi yang diuji. Jika karakter pertama cocok, maka seluruh pola sudah cocok dengan substring tersebut, dan algoritma segera mengembalikan posisi kecocokan pertama tanpa melakukan perbandingan lebih lanjut. Hal ini menyebabkan algoritma hanya melakukan satu perbandingan karakter di loop dalam, sehingga waktu eksekusinya menjadi **O(1)**. Kasus ini sangat jarang terjadi, namun memberikan gambaran bahwa algoritma dapat bekerja dengan sangat cepat dalam kondisi yang menguntungkan.

Pada **kasus rata-rata** (average-case), kita mengasumsikan bahwa pola tidak ditemukan di awal, tetapi akan ditemukan setelah sejumlah perbandingan. Rata-rata, pada setiap posisi yang diuji dalam teks, kita mungkin akan mencocokkan sebagian besar pola, namun tidak seluruhnya. Dalam skenario ini, rata-rata perbandingan yang dilakukan dalam setiap iterasi loop dalam adalah sekitar setengah panjang pola, yakni **m/2**. Loop luar akan tetap melakukan iterasi sebanyak **n - m + 1** kali (atau kira-kira **n** kali), di mana **n** adalah panjang

teks dan **m** adalah panjang pola. Oleh karena itu, total waktu yang diperlukan untuk melakukan pencocokan adalah perkalian antara jumlah iterasi loop luar (sekitar **n**) dan rata-rata perbandingan dalam loop dalam (sekitar **m/2**), sehingga waktu eksekusi rata-rata untuk algoritma ini adalah **O(n * m)**. Ini berarti, meskipun algoritma tidak membutuhkan waktu sebanyak kasus terburuk, ia masih melakukan banyak perbandingan dan tetap tidak efisien untuk teks panjang atau pola besar.

Pada **kasus terburuk** (worst-case), pola tidak ditemukan sama sekali atau ditemukan hanya setelah memeriksa seluruh pola di setiap posisi teks. Dalam hal ini, untuk setiap iterasi dari loop luar, kita melakukan pemeriksaan seluruh pola, yang memakan waktu **m** perbandingan karakter. Loop luar akan berjalan sebanyak **n - m + 1** kali, yang kira-kira setara dengan **n** iterasi, dan dalam setiap iterasi tersebut kita melakukan **m** perbandingan di loop dalam. Oleh karena itu, total perbandingan yang dilakukan adalah hasil perkalian jumlah iterasi loop luar dengan jumlah perbandingan dalam loop dalam, yang menghasilkan waktu eksekusi **O(n * m)**. Ini adalah **kasus terburuk** yang terjadi ketika pola tidak cocok di hampir semua posisi teks, dan algoritma harus memeriksa seluruh pola di setiap posisi, menjadikannya sangat tidak efisien untuk teks yang lebih besar atau pola yang lebih panjang

## E. Referensi

Levitin, A. (2012). Introduction to the design & analysis of algorithms (3rd ed.). Pearson Education, Inc

## F. Lampiran Link Github

**https://github.com/piks16/23343040-Hasanul-Fikri-Bruteforce**