TUGAS

PERANCANGAN ANALISIS ALGORITMA

“Binary Exponentiation”

202423430076

DOSEN PENGAMPU:

Randi Proska Sandra, M.Sc.

OLEH:

Nama : Muhammad Devin Rahadi

NIM : 23343076

Program Studi : Informatika

PROGRAM STUDI INFORMATIKA
DEPARTEMEN ELEKTRONIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

2025

A. PENJELASAN PROGAM

Binary exponentiation adalah teknik efisien untuk menghitung hasil perpangkatan bilangan bulat secara optimal. Metode ini mengurangi jumlah operasi perkalian secara drastis dibandingkan dengan pendekatan naïf (brute force), yang harus melakukan perkalian sebanyak $\mathbf{n - 1}$ kali untuk menghitung $\mathbf{a^n}$. Dengan binary exponentiation, jumlah perkalian berkurang menjadi sekitar $O(\log n)$, membuatnya jauh lebih efisien, terutama saat menangani bilangan besar dalam bidang komputasi numerik dan kriptografi.

Konsep Dasar

Alih-alih melakukan perkalian berulang sebanyak nn kali, metode ini memanfaatkan representasi biner dari eksponen. Misalkan kita ingin menghitung ana^n, eksponen nn dapat dinyatakan dalam bentuk biner:

$$n = b_k 2^k + b_k - {}_1 2^{k-1} + \ldots + b_1 2^1 + b_0 2^0$$

Dengan representasi ini, perhitungan perpangkatan bisa diubah menjadi kombinasi perkalian berbasis bit. Teknik ini didasarkan pada dua prinsip utama:

- **Perpangkatan Bertahap**: Setiap kali kita melangkah ke bit berikutnya, hasil sementara dikuadratkan.
- **Perkalian Selektif**: Jika bit eksponen bernilai 1, maka hasil sementara dikalikan dengan nilai dasar aa.

Dua Pendekatan Binary Exponentiation

Terdapat dua metode utama untuk menerapkan binary exponentiation, yaitu **Left-to-Right Binary Exponentiation** dan **Right-to-Left Binary Exponentiation**.

1. **Left-to-Right Binary Exponentiation**
    - Dimulai dari bit paling signifikan ke bit paling rendah.
    - Inisialisasi nilai hasil dengan basis aa.
    - Untuk setiap bit eksponen dari kiri ke kanan, lakukan perpangkatan, dan jika bit tersebut bernilai 1, lakukan perkalian tambahan.
2. **Right-to-Left Binary Exponentiation**
    - Dimulai dari bit paling kecil ke bit paling besar.
    - Gunakan **accumulator** untuk menyimpan hasil sementara.
    - Setiap kali menemukan bit 1, kalikan accumulator dengan nilai basis yang dipangkatkan.

Kompleksitas dan Aplikasi

Keunggulan utama dari binary exponentiation adalah kompleksitasnya yang hanya sebesar $\boldsymbol{O(\log n)}$ dibandingkan metode naïf dengan $\boldsymbol{O(n)}$. Efisiensi ini sangat penting dalam berbagai aplikasi, terutama di bidang **kriptografi** seperti **RSA**, **modular exponentiation**, serta dalam pemrograman kompetitif yang sering kali memerlukan komputasi perpangkatan besar dalam waktu singkat.

Dengan optimalisasi ini, binary exponentiation menjadi salah satu algoritma fundamental dalam dunia komputasi, memberikan solusi efisien untuk masalah yang melibatkan perpangkatan besar.

B. PSEUCODE

```
ALGORITMA Pangkat_Biner(a, n)
Input: a (bilangan basis), n (eksponen, bilangan bulat non-negatif)
Output: hasil dari a^n

1. hasil ← 1
2. basis ← a
3. selama n > 0 lakukan:
   a. jika n mod 2 = 1, maka:
      - hasil ← hasil * basis
   b. basis ← basis * basis
   c. n ← n // 2  (geser bit ke kanan)
4. Kembalikan hasil
```

C. SOURCE CODE

```
print("NAMA : MUHAMMAD DEVIN RAHADI")
print("NIM : 23343076\n")

def pangkat_biner(basis, eksponen):
    """
    Fungsi untuk menghitung perpangkatan menggunakan metode Binary
Exponentiation.

    Parameter:
    basis (int) : Bilangan dasar yang akan dipangkatkan.
    eksponen (int) : Bilangan pangkat (harus bilangan bulat non-
negatif).

    Mengembalikan:
    int : Hasil dari basis^eksponen.
    """
    hasil = 1  # Inisialisasi hasil awal sebagai 1
    while eksponen > 0:
        if eksponen % 2 == 1:  # Jika eksponen ganjil
            hasil *= basis
        basis *= basis  # Basis dikalikan dengan dirinya sendiri
        eksponen //= 2  # Eksponen dibagi dua (geser bit ke kanan)
    return hasil

# Contoh Penggunaan
a = int(input("Masukkan bilangan basis: "))
n = int(input("Masukkan bilangan eksponen: "))

print(f"Hasil dari {a}^{n} adalah: {pangkat_biner(a, n)}")
```

D. ANALISIS KEBUTUHAN WAKTU

1. **Analisis Berdasarkan Jumlah Instruksi yang Dieksekusi**
   Pendekatan pertama dilakukan dengan menghitung jumlah instruksi utama dalam algoritma. Berikut adalah kode utama dari algoritma:

```
def pangkat_biner(basis, eksponen):
    hasil = 1             # 1 operasi assignment
    while eksponen > 0:    # Loop berjalan sekitar O(log n) kali
        if eksponen % 2 == 1:  # 1 operasi modulus + 1 perbandingan per iterasi
            hasil *= basis   # 1 perkalian + 1 assignment jika eksponen ganjil
        basis *= basis      # 1 perkalian + 1 assignment per iterasi
        eksponen //= 2      # 1 pembagian + 1 assignment per iterasi
    return hasil              # 1 operasi return
```

   Dalam proses eksekusi, terdapat beberapa jenis operasi utama yang dilakukan:

   - **Inisialisasi variabel**: hasil = 1 hanya dilakukan sekali sehingga memiliki **O(1)** kompleksitas.
   - **Loop while eksponen > 0**: Jumlah iterasi dalam loop ini bergantung pada jumlah bit dalam representasi biner eksponen, yang menyebabkan iterasi berjalan sebanyak **O(log n)** kali.
   - **Operasi dalam loop**: Pada setiap iterasi, terdapat beberapa operasi:
     - **Operasi modulus (%)**: Mengecek apakah eksponen adalah bilangan ganjil. Operasi ini dilakukan di setiap iterasi sehingga berjumlah **O(log n)** kali.
     - **Operasi perkalian (*)**: Jika eksponen ganjil, dilakukan perkalian tambahan, yang pada kasus terburuk terjadi setiap iterasi, yaitu **O(log n)** kali. Selain itu, basis juga selalu dikuadratkan setiap iterasi, sehingga juga **O(log n)**.
     - **Operasi pembagian (//)**: Eksponen dibagi dua setiap iterasi (eksponen //= 2), yang menyebabkan algoritma berhenti setelah sekitar $\log_2(n)$ langkah, sehingga operasinya juga **O(log n)**.

   Secara keseluruhan, algoritma melakukan sejumlah operasi utama dalam setiap iterasi, dan karena jumlah iterasi sekitar **O(log n)**, maka total jumlah operasi utama dalam skala besar tetap **O(log n)**.

2. **Analisis Berdasarkan Jumlah Operasi Abstrak**
   Pendekatan kedua dilakukan dengan menghitung jumlah operasi abstrak yang berpengaruh langsung terhadap performa algoritma. Kita fokus pada tiga operasi utama dalam algoritma ini, yaitu:

   - **Perkalian (*)**: Terdapat dua jenis perkalian dalam algoritma. Yang pertama adalah perkalian untuk menghitung hasil akhir (hasil *= basis), yang terjadi hanya jika bit eksponen bernilai 1. Karena dalam representasi

biner jumlah bit "1" dalam angka acak berkisar sekitar **setengah dari total bit**, maka dalam kasus rata-rata terjadi **O(log n) / 2 ≈ O(log n)** kali. Yang kedua adalah perkalian untuk menghitung basis kuadrat (basis *= basis), yang terjadi di setiap iterasi dan berjumlah **O(log n)** kali.

- **Pembagian (//)**: Operasi eksponen //= 2 dilakukan di setiap iterasi hingga eksponen mencapai 0, sehingga jumlahnya **O(log n)** kali.

- **Modulus (%)**: Operasi eksponen % 2 digunakan untuk mengecek apakah eksponen ganjil, yang dilakukan setiap iterasi, sehingga jumlahnya **O(log n)**.

  Dengan demikian, secara keseluruhan jumlah operasi abstrak dalam algoritma ini tetap **O(log n)**, karena iterasi dilakukan sebanyak **log n** kali dengan jumlah operasi dalam setiap iterasi tetap konstan.

3. **Analisis Berdasarkan Best-Case, Worst-Case, dan Average-Case**
   Pendekatan ketiga dilakukan dengan melihat kompleksitas algoritma berdasarkan kasus terbaik, kasus terburuk, dan kasus rata-rata.

   - **Best-Case (Kasus Terbaik)**: Kasus terbaik terjadi ketika eksponen bernilai **0**, misalnya pada perhitungan a0a^0a0, yang langsung mengembalikan nilai **1** tanpa perlu masuk ke dalam loop. Dalam kasus ini, jumlah operasi hanya **O(1)** karena tidak ada perhitungan lain yang perlu dilakukan.

   - **Worst-Case (Kasus Terburuk)**: Kasus terburuk terjadi ketika eksponen memiliki nilai yang sangat besar, seperti nilai maksimum dalam sistem komputer (contoh: n=231−1n = 2^{31}-1n=231−1 pada sistem 32-bit). Dalam kondisi ini, eksponen perlu dibagi dua terus-menerus hingga menjadi 0, yang menyebabkan jumlah iterasi mencapai sekitar **O(log n)**. Karena setiap iterasi membutuhkan beberapa operasi, kompleksitas waktu tetap **O(log n)** dalam kasus ini.

   - **Average-Case (Kasus Rata-Rata)**: Jika eksponen dipilih secara acak, rata-rata memiliki sekitar **setengah bit** yang bernilai 1, sehingga jumlah iterasi tetap sekitar **log n**. Meskipun tidak semua iterasi melakukan semua operasi, perbedaan ini tidak mengubah kompleksitas waktu secara signifikan. Oleh karena itu, dalam kasus rata-rata, kompleksitas waktu juga tetap **O(log n)**.

E. REFERENSI

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

F. LAMPIRAN LINK GITHUB

https://github.com/vondeastra/Perancangan_analisis_algoritma