Tugas 1 : Analisis Kompleksitas Waktu

Mata Kuliah : Analisis Algoritma

Syifa Fauziyah N. I. 140810160026

S-1 Teknik Informatika

Fakultas Matematika & Ilmu Pengetahuan Alam

Universitas Padjadjaran

Jalan Raya Bandung - Sumedang Km. 21 Jatinangor 45363

I. Code program

1.1 Pangkat iterasi

```cpp
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

main(){
    int pangkat, hasil, n;

    cout<<"Masukan nilai utama : "<<endl;
    cin>>n;
    cout<<"Masukan pangkat : "<<endl;
    cin>>pangkat;
    hasil = n;

    auto start = high_resolution_clock::now();

    for(int i = 0; i<pangkat-1; i++){
        hasil*=n;
    }

    cout<<"\nHasil = "<< hasil<<endl;

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(stop - start);

    cout << "Time taken by function: "
         << duration.count() << " nanoseconds" << endl;
}
```

1.2 Pangkat rekursif

```cpp
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

int Pangkat(int x,int y){
    if (y==0){
        return 1;
    }
    else{
        return (x*Pangkat(x,y-1));
    }

}
int main(int argc, char** argv) {
    int b,p;
    cout<<"masukkan bilangan : "<<endl;
    cin>>b;
    cout<<"masukkan pangkat : "<<endl;
    cin>>p;
    cout<<endl;
    auto start = high_resolution_clock::now();
    cout<<b<<"^"<<p<<"="<<Pangkat(b,p)<<endl;
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(stop - start);

    cout << "Time taken by function: "
         << duration.count() << " nanoseconds" << endl;
    return 0;
}
```

1.3 Linear search

```
#include<iostream>
#include<conio.h>
#include <chrono>
using namespace std;
using namespace std::chrono;
int main(int argc, char** argv)
{
    int arr[10], i, num, n, c=0, pos;
    cout<<"Enter the array size : "; cin>>n;
    cout<<"Enter Array Elements : ";
    for(i=0; i<n; i++)
    {
        cin>>arr[i];
    }
    cout<<"Enter the number to be search : ";
    cin>>num;
    auto start = high_resolution_clock::now();
    for(i=0; i<n; i++)
    {
        if(arr[i]==num)
        {
            c=1;
            pos=i+1;
            break;
        }
    }
    if(c==0)
    {
        cout<<"Number not found..!!";
    }
    else
    {
        cout<<num<<" found at position "<<pos;
    }
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(stop - start);
    cout << "Time taken by function: "
         << duration.count() << " nanoseconds" << endl;
    return 0;
}
```

1.4 Binary search

```
#include<iostream>
#include <chrono>
using namespace std;
using namespace std::chrono;
int main(int argc, char** argv)
{
    int n, i, arr[50], search, first, last, middle;
    cout<<"Enter total number of elements :"; cin>>n;
    cout<<"Enter "<<n<<" number :";
    for (i=0; i<n; i++)
    {
        cin>>arr[i];
    }
    cout<<"Enter a number to find :"; cin>>search;
    auto start = high_resolution_clock::now();
    first = 0;
    last = n-1;
    middle = (first+last)/2;
    while (first <= last)
    {
        if(arr[middle] < search)
            first = middle + 1;
        else if(arr[middle] == search)
        {
            cout<<search<<" found at location "<<middle+1<<"\n";
            break;
        }
        else
             last = middle - 1;
        middle = (first + last)/2;
    }
    if(first > last)
        cout<<"Not found! "<<search<<" is not present in the list.";
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(stop - start);
    cout << "Time taken by function: "
         << duration.count() << " nanoseconds" << endl;
    return 0;
}
```

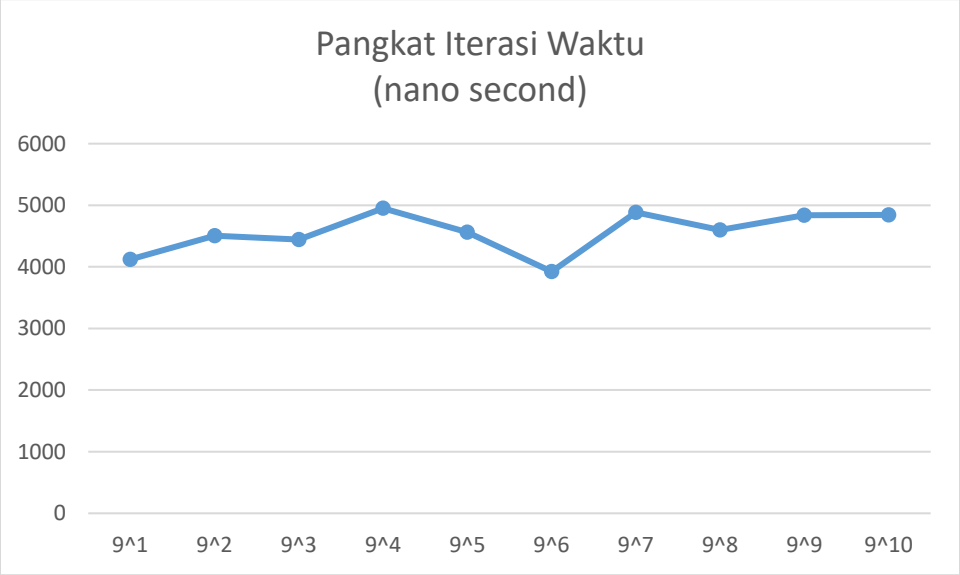
II. Spesifikasi Laptop

- RAM : 8GB
- Core i5
- Win 10

III. Hasil

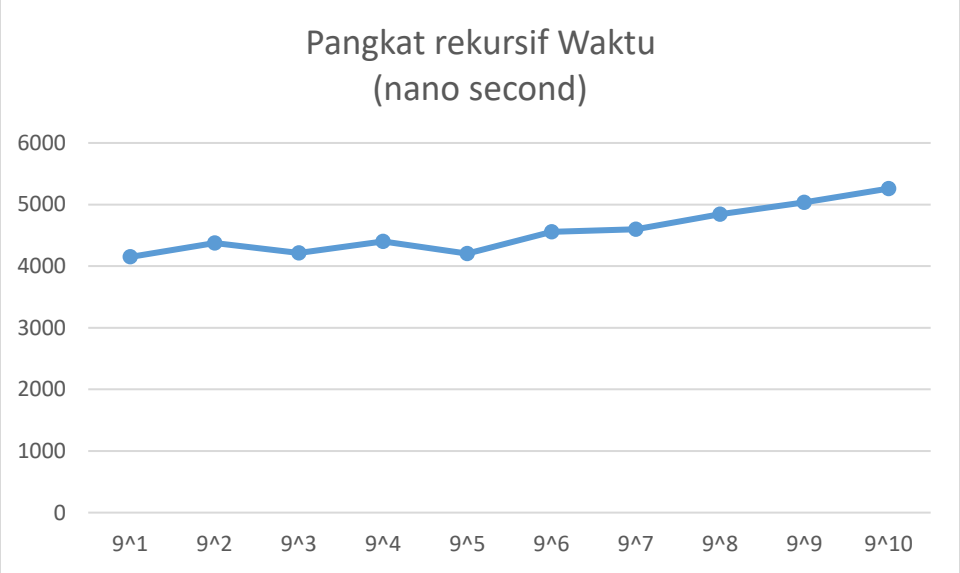
3.1 Pangkat iterasi

Pangkat Iterasi	
Input	**Waktu (nano second)**
9^1	4123
9^2	4506
9^3	4445
9^4	4951
9^5	4563
9^6	3923
9^7	4883
9^8	4600
9^9	4837
9^10	4843



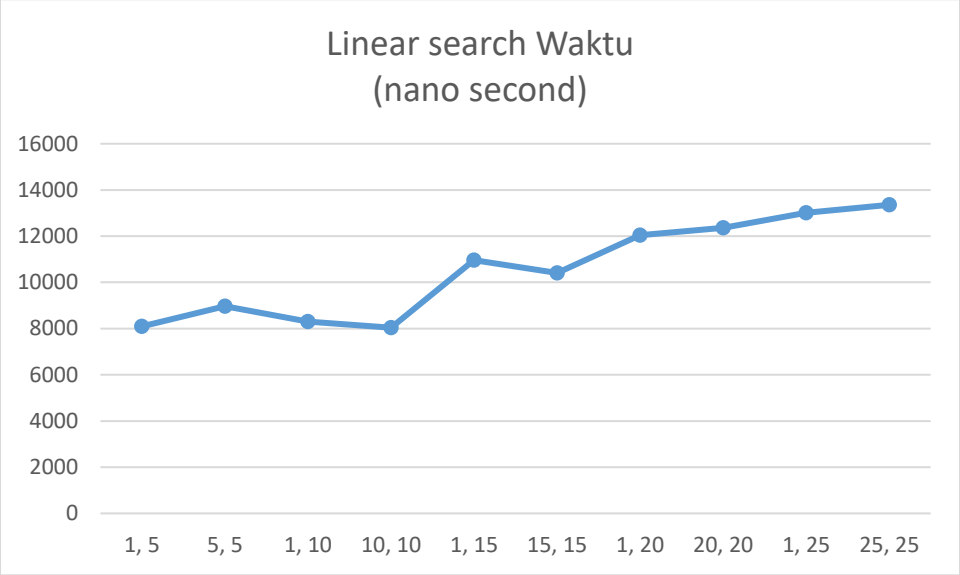
3.2 Pangkat rekursif

Pangkat rekursif	
Input	**Waktu (nano second)**
9^1	4150
9^2	4376
9^3	4215
9^4	4402
9^5	4202
9^6	4558
9^7	4598
9^8	4842
9^9	5037
9^10	5259



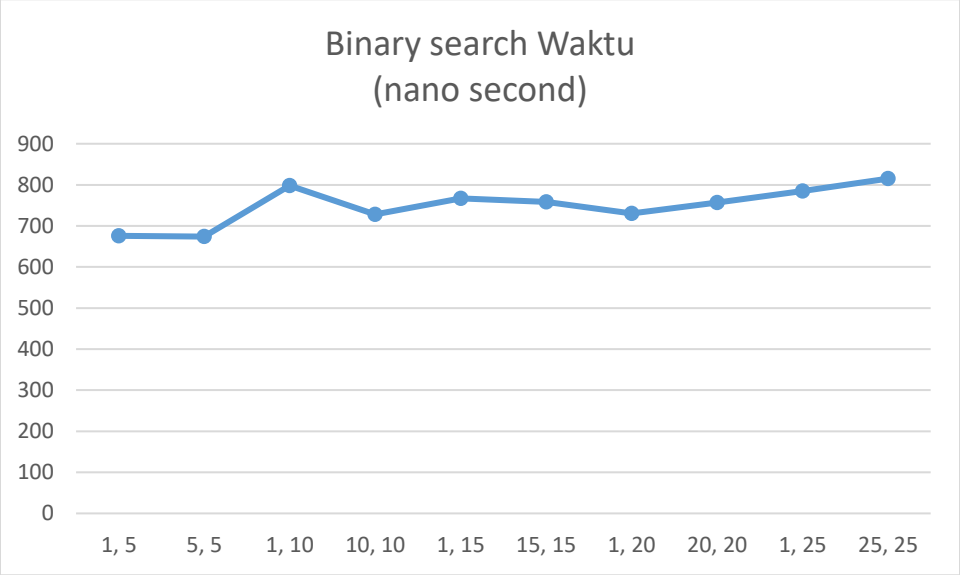
3.3 Linear search

Linear search	
Percobaan (letak data, jumlah data)	**Waktu (nano second)**
1, 5	8092
5, 5	8962
1, 10	8300
10, 10	8038
1, 15	10966
15, 15	10412
1, 20	12040
20, 20	12366
1, 25	13017
25, 25	13355



3.4 Binary search

Binary search	
Percobaan (letak data, jumlah data)	**Waktu (nano second)**
1, 5	676
5, 5	674
1, 10	798
10, 10	728
1, 15	767
15, 15	758
1, 20	730
20, 20	757
1, 25	785
25, 25	815



IV. Analisis

Dalam membuat program tidak hanya mengukur output yang dihasilkan benar atau salah. Selain memberikan hasil yang benar, efisiensi dari waktu eksekusi ataupun penggunaan memori dari algoritma adalah hal yang penting bagi sebuah algoritma. Seperti program yang telah dibuat adalah program mencari hasil pangkat suatu bilangan dan program pencarian.

Program untuk mencari hasil pemangkatan suatu bilangan dan program pencarian tidak hanya 1, tetapi bisa ada beberapa program yang bisa menghasilkan output yang benar. Namun, masalahnya manakah yang lebih efesien? Ada banyak cara mengukur ke efesienan suatu algoritma. Contohnya yang paling sederhana adalah melihat berapa langkah yang perlu dijalankan untuk menyelesaikan algoritma tersebut. Semakin banyaknya langkah yang dijalankan maka akan semakin lama runtime atau waktu yang diperlukan untuk menjalankan program tersebut. Untuk menggambarkan banyaknya langkah yang diperlukan fungsi pertumbuhan. Penulisan fungsi pertumbuhan ini dilakukan dengan menggunakan notasi asmtotik, yaitu salah satunya Big-O.

Program pangkat

Fungsi pangkat dapat dituliskan sebagai

$$F(x,y) = x^y$$

Sehingga banyaknya perulangan akan dipengaruhi oleh y. Contoh 2^4 atau pangkat(2,4). Maka dapat dijabarkan :

Hasil = 1 //inisialisasi awal
Hasil = 2 * Hasil
Hasil = 2 * Hasil
Hasil = 2 * Hasil
Hasil = 2 * Hasil
Return Hasil

Secara sederhana, dapat dituliskan sebagai berikut :

Baris Kode	**Jumlah Eksekusi**
`hasil = 1`	1
`hasil = x * hasil`	y
`return hasil`	1

Sehingga,

Y	Proses Perhitungan	Jumlah Langkah
1	2 + 1	3
10	2 + 10	12
100	2 + 100	102
1000	2 + 1000	1002
10000	2 + 10000	10002

Sesuai yang telah dijelaskan, bahwa kompleksitas bisa di lambangkan dengan Big-O maka untuk pencarian pangkat ini masuk kepada **O(n)** karena Algoritma dengan kompleksitas linear bertumbuh selaras dengan pertumbuhan ukuran data. Jika algoritma ini memerlukan 10 langkah untuk menyelesaikan kalkulasi data berukuran 10, maka ia akan memerlukan 100 langkah untuk data berukuran 100. Bisa dilihat juga dari hasil runtimenya, jika kita menambahkan data yang lebih banyak maka runtime lebih lama.

Program Search

- Linear Search

  Linear search melakukan pencarian dengan menelusuri elemen-elemen dalam list satu demi satu, mulai dari indeks paling rendah sampai indeks terakhir. Dilihat dari algoritmanya :

```
def linear_search(lst, search):
   for i in range(0, len(lst)):
       if lst[i] == search:
           print("Nilai ditemukan pada posisi " + str(i))
           return 0
   print("Nilai tidak ditemukan.")
   return -1
```

  Dengan menggunakan cara perhitungan yang sama pada perhitungan pangkat, kita bisa mendapatkan jumlah eksekusi kode seperti berikut (dengan asumsi n = len(lst)) maka :

Kode	**Jumlah Eksekusi**
for i in range(0, len(lst))	1
if lst[i] == search	*n*
print("Nilai ditemukan...	1
return 0	1
print("Nilai tidak ...	1
return -1	1

  Sehingga nilai kompleksitas dari linear search adalah 5+n, atau dapat dituliskan sebagai **O(n).**

- Binary Search

  Binary search adalah program dengan dasar membagi data menjadi beberapa bagian. Oleh karena itu binary search masuk kenalan **O(log n).** O(log n) adalah algoritma dengan kompleksitas logaritmik merupakan algoritma yang menyelesaikan masalah dengan membagi-bagi masalah tersebut menjadi beberapa

bagian, sehingga masalah dapat diselesaikan tanpa harus melakukan komputasi atau pengecekan terhadap seluruh masukan. Berikut adalah code binary search :

```
def binary_search(lst, search):
    lower_bound = 0
    upper_bound = len(lst) - 1

    while True:
        if upper_bound < lower_bound:
            print("Not found.")
            return -1

        i = (lower_bound + upper_bound) // 2

        if lst[i] < search:
            lower_bound = i + 1
        elif lst[i] > search:
            upper_bound = i - 1
        else:
            print("Element " + str(search) + " in " + str(i))
            return 0
```

Mari kita hitung jumlah langkah yang diperlukan untuk mendapatkan kelas kompleksitas dari binary search. Berikut adalah tahapan perhitungan untuk mendapatkan jumlah langkah yang diperlukan:

1. Langkah yang akan selalu dieksekusi pada awal fungsi, yaitu inisialisasi `lower_bound` dan `upper_bound`: **2 langkah**.
2. Pengecekan kondisi `while` (pengecekan tetap dilakukan, walaupun tidak ada perbandingan yang dijalankan): **1 langkah**.
3. Pengecekan awal (`if upper_bound < lower_bound`): **1 langkah**.
4. Inialisasi `i`: **1 langkah**.
5. Pengecekan kondisi kedua (`if lst[i] < search: ...`), kasus terburuk (masuk pada `else` dan menjalankan kode di dalamnya): **4 langkah**.

Dan setelah melalui langkah kelima, jika elemen belum ditemukan maka kita akan kembali ke langkah kedua. Perhatikan bahwa sejauh ini, meskipun elemen belum ditemukan atau dianggap tidak ditemukan, kita minimal harus menjalankan 2 langkah dan pada setiap perulangan while kita menjalankan 7 langkah. Sampai di titik ini, model matematika untuk fungsi Big-O yang kita dapatkan ialah seperti berikut:

f(n)=2+7(jumlah perulangan)

Pertanyaan berikutnya, tentunya adalah berapa kali kita harus melakukan perulangan? Berhentinya kondisi perulangan ditentukan oleh dua hal, yaitu:

1. Kondisi `upper_bound < lower_bound`, dan
2. Pengujian apakah `lst[i] == search`, yang diimplikasikan oleh perintah `else`.

Perhatikan juga bagaimana baik nilai upper_bound maupun lower_bound dipengaruhi secara langsung oleh i, sehingga dapat dikatakan bahwa kunci dari berhentinya perulangan ada pada i. Melalui baris kode ini:

```
i = (lower_bound + upper_bound) // 2
```

Kita melihat bahwa pada setiap iterasinya nilai i dibagi 2, sehingga untuk setiap iterasinya kita memotong jumlah data yang akan diproses (n) sebanyak setengahnya. Sejauh ini kita memiliki model matematika seperti berikut (konstanta 2 dihilangkan karena tidak berpengaruh):

$$f(n)=7f(n2)$$

yang jika diturunkan lebih lanjut akan menjadi:

$$f(n)=7f(n2)=7*(7f(n4))=49f(n4)=49*(7f(n8))...=7_kf(n2_k)$$

di mana kita ketahui kondisi dari pemberhentian perulangan adalah ketika sisa elemen list adalah 1, dengan kata lain:

$$\begin{aligned} \frac{n}{2^k} &= 1 \\ n &= 2^k \\ \log_2 n &= k \end{aligned}$$

Sehingga dapat dikatakan bahwa binary search memiliki kompleksitas $O(\log_2 n)$, atau sederhananya, $O(\log n)$.