Kompleksitas Algoritma (Fungsi Pertumbuhan)

Sukmawati Nur Endah

Departemen Informatika UNDIP

PR (Dikumpulkan)

- Buatlah algoritma pencarian beruntun (sequensial search)
- Hitunglah kompleksitas waktu terbaik, terburuk dan rata-rata dari algoritma tersebut!

Penyelesaian
Alg pencarian Beruntun

```
Procedure PencarianBeruntun (input  a1,a2,...an:integer, x :integer,
                                    output  idx:integer)
Dekalasi
    i : integer
    ketemu : boolean {bernilai true jika x ditemukan, atau false jika x tidak ditemukan}
Algoritma
    i ← 1
    ketemu ← false
    while (i ≤ n) and (not ketemu) do
      if ai = x then
            ketemu ← true
      else
            i ← i + 1
      endif
    endwhile
```

Lanjutan Algoritma Pencarian Beruntun

```
if ketemu then
        idx ← 1 {x ditemukan}
else
        idx ← 0 {x tidak ditemukan}
endif
```

Penyelesaian

- Algoritma ini membandingkan elemen larik dengan x, mulai dari elemen pertama sampai x ditemukan sampai elemen terakhir.
- Operasi dasar algoritma :
  - Operasi perbandingan elemen-eleman larik
- Kasus terbaik ➔ terjadi bila a1 = x
  - Operasi perbandingan hanya dilakukan sekali
  - T_{min} (n) = 1

Penyelesaian

- Kasus terburuk ➔ terjadi bila an = x atau x tidak ditemukan
  - Semua elemen akan dibandingkan
  - T_{max} (n) = n

- Kasus rata-rata
  - Jika x ditemukan pada posisi ke j, maka operasi perbandingan dilakukan sebanyak j kali
  - T_{avg} (n) = (1+2+ ...+n) / n

    = (½ n (n+1)) / n

    = (n+1) / n

Bagaimana dengan fungsi perpangkatan?

- Contoh

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

- Buatlah algoritmanya!

Algoritma Perpangkatan

```
def pangkat(x, y):
    hasil = 1
    for i in range(0, y):
        hasil = x * hasil
    return hasil
```

- Dengan algoritma diatas, berapa langkah untuk pangkat (2,1) ?

Alg Perpangkatan

```
hasil = 1
hasil = 2 * hasil
return hasil
```

- Terdapat 3 langkah
- Bagaimana dengan pangkat (2,2)? Ada berapa langkah?
- Bagaimana dengan pangkat (2,5)? Ada berapa langkah?

- Terkait jumlah langkah, apa yang dapat disimpulkan?

Alg Perpangkatan

- baris : hasil = x * hasil **dijalankan sebanyak y kali**

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

- **Fungsi pangkat akan selalu diselesaikan dalam 2+y**

Alg Perpangkatan

- Pengaruh y terhadap jumlah eksekusi

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

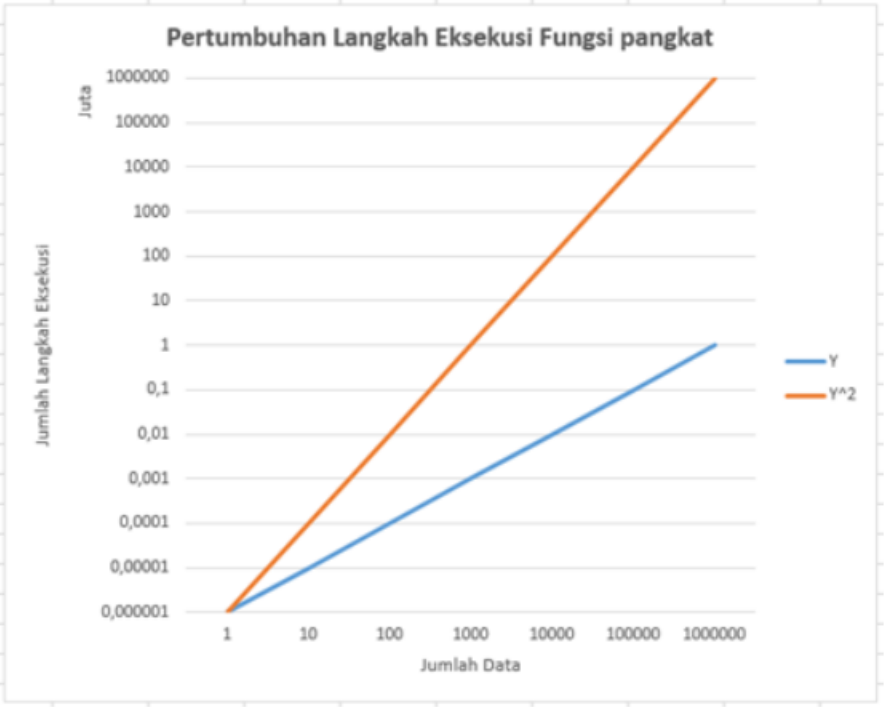
- Nilai angka 2 semakin tidak signifikan untuk y yang besar
- Untuk itu dapat disederhanakan fungsi pangkat dapat diselesaikan dalam y langkah

Bagaimana jika sebuah alg mempunyai y^2 langkah?

Y	Jumlah Langkah (y)	Jumlah Langkah (y^2)
1	1	1
10	10	100
100	100	10000
1000	1000	1000000
10000	10000	100000000

Fungsi Pertumbuhan

- Efisiensi algoritma dilihat dari ukuran pertumbuhan jumlah langkah eksekusi terhadap ukuran jumlah data
- Fungsi pangkat mempunyai fungsi pertumbuhan linier



Tingkat Pertumbuhan Fungsi Pangkat

Notasi Asimtotik

- Fungsi pertumbuhan mempunyai notasi ➔ notasi Asimtotik

- Notasi Asimtotik
  - Big O (O besar)
  - Big Ω (Omega Besar)
  - Big Θ (Tetha Besar)

Big O

- Notasi Big-O digunakan untuk :
  - mengkategorikan algoritma ke dalam fungsi yang menggambarkan batas atas (*upper limit*) dari pertumbuhan sebuah fungsi ketika masukan dari fungsi tersebut bertambah banyak

- Definisi:

  $T(n) = O(f(n))$ bila terdapat konstanta C dan n_0 sedemikian sehingga $T(n) \leq C.\ f(n)$, untuk $n \geq n_0$

O(1): Kompleksitas Konstan

- Berapapun ukuran data atau masukan yang diterima, algoritma dengan kompleksitas konstan akan *memiliki jumlah langkah yang sama untuk dieksekusi*

- merupakan algoritma **paling efisien** dari seluruh kriteria yang ada.

- Contoh : algoritma yang digunakan untuk menambahkan elemen baru ke dalam linked list

Fungsi Big-O

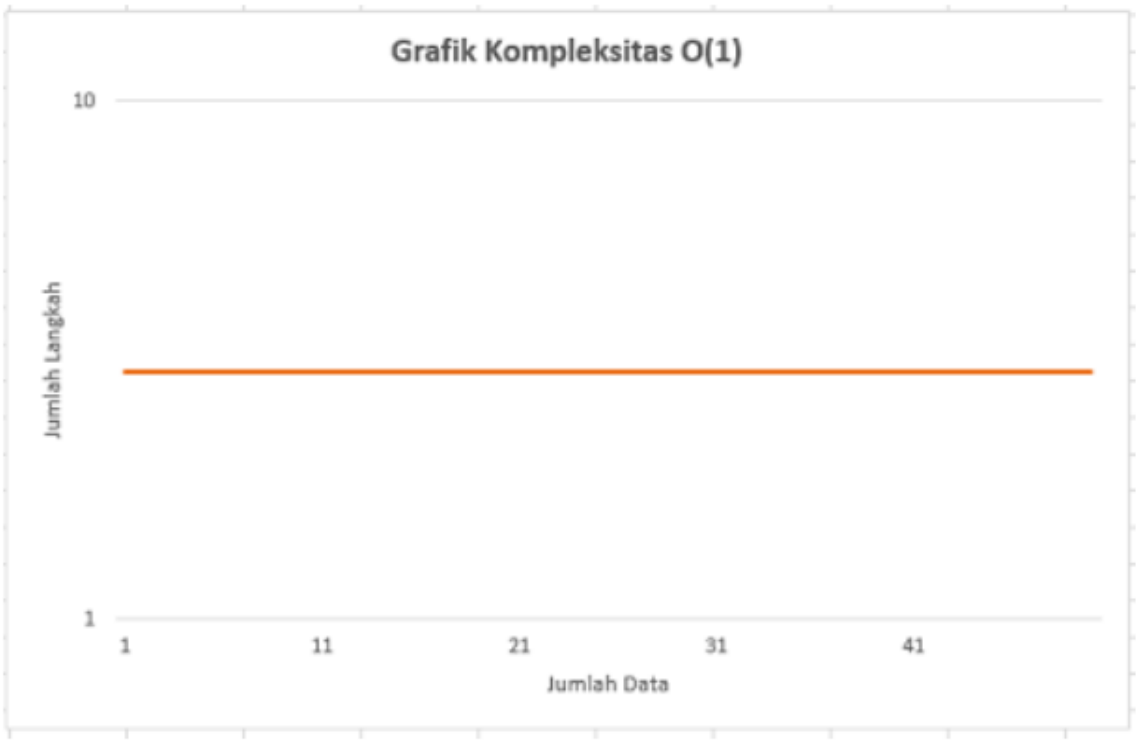
Fungsi Big-O	Nama
$O(1)$	Konstan
$O(\log n)$	Logaritmik
$O(n)$	Linear
$O(n \log n)$	n log n
$O(n^2)$	Kuadratik
$O(n^m)$	Polinomiale
$O(n!)$	Faktorial

O(1): Kompleksitas Konstan

- Contoh implementasi dengan bahasa C

```
void add_list(node *anchor, node *new_list)
{
    new_list->next = anchor->next;
    anchor->next = new_list;
}
```

O(1): Kompleksitas Konstan



Tingkat Pertumbunan Algoritma Kompleksitas Konstan

O(log n): Kompleksitas Logaritmik

- 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.

- Contoh : algoritma *binary search*

O(log n): Kompleksitas Logaritmik

- Implementasi binary search dengan python

```
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
```

O(log n): Kompleksitas Logaritmik

- Perhitungan

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:

$$f(n) = 2 + 7(\text{jumlah perulangan})$$

O(log n): Kompleksitas Logaritmik

- Berapa kali harus melakukan perulangan??
- Kondisi perulangan dilihat dari dua hal berikut :

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):

$$f(n) = 7f(\frac{n}{2})$$

O(log n): Kompleksitas Logaritmik

- Diturunkan lebih lanjut

$$
\begin{aligned}
f(n) &= 7f(\frac{n}{2}) \\
&= 7 * (7f(\frac{n}{4})) \\
&= 49f(\frac{n}{4}) \\
&= 49 * (7f(\frac{n}{8})) \\
&\dots \\
&= 7^k f(\frac{n}{2^k})
\end{aligned}
$$

O(log n): Kompleksitas Logaritmik

- kondisi dari pemberhentian perulangan adalah ketika sisa elemen list adalah 1

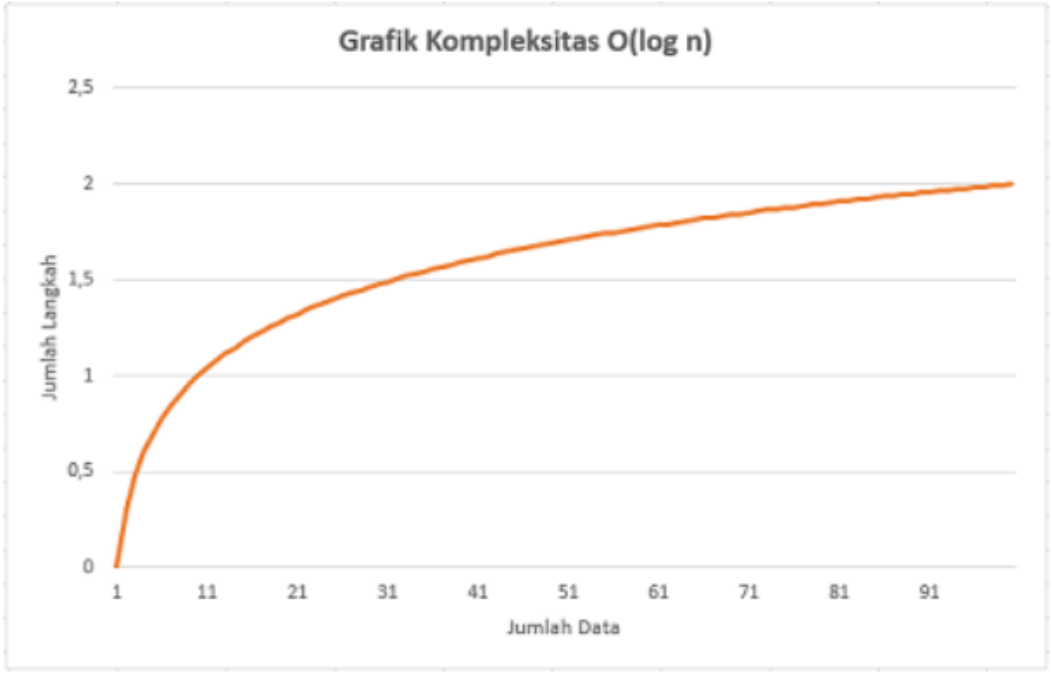
$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\log_2 n = k$$

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

O(log n): Kompleksitas Logaritmik



Tingkat Pertumbunan Algoritma Kompleksitas Logaritmik

O(n): Kompleksitas Linear

- Algoritma dengan kompleksitas linear bertumbuh selaras dengan pertumbuhan ukuran data.

- Contoh : algoritma perhitungan pangkat bilangan, algoritma linear search.

O(n): Kompleksitas Linear

- Contoh linear search pada python

```
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
```

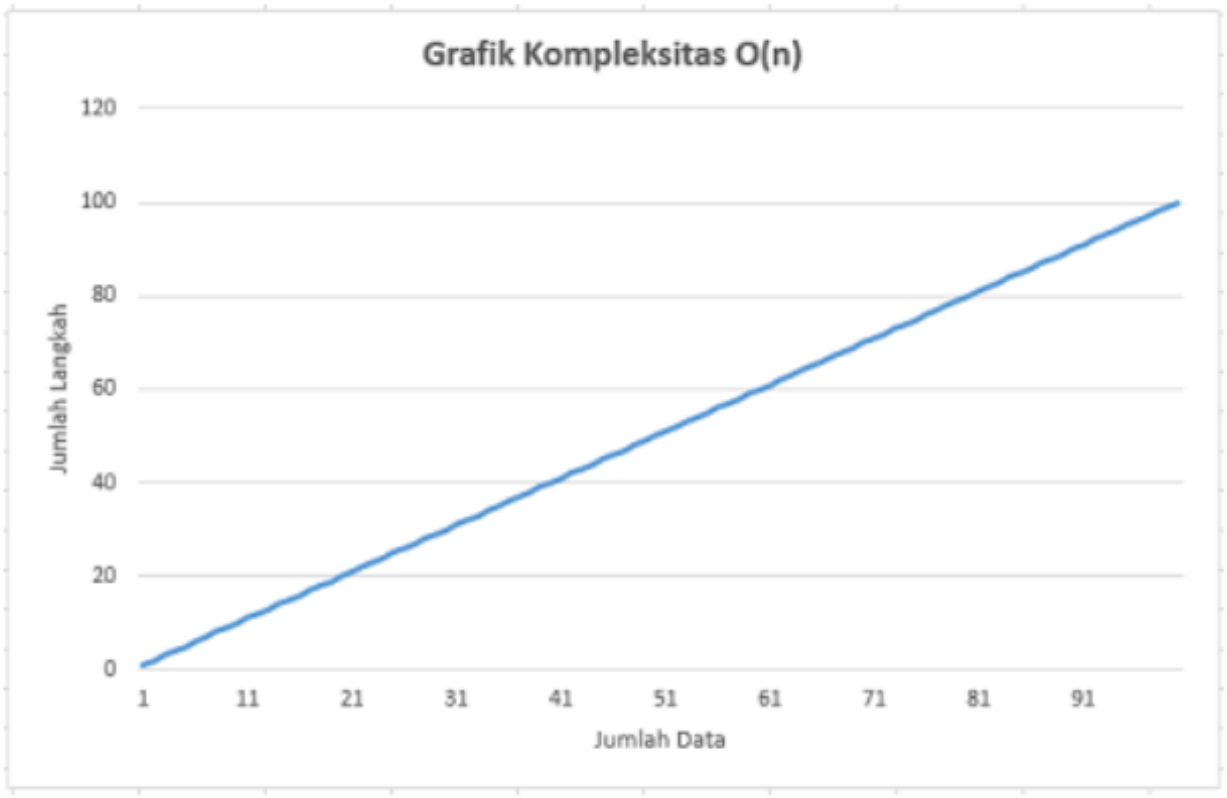
O(n): Kompleksitas Linear

- asumsi n = len(lst)

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

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

O(n): Kompleksitas Linear



Tingkat Pertumbunan Algoritma Kompleksitas Linear

O(n log n)

- Pada dasarnya algoritma kelas ini merupakan algoritma log*n* yang dijalankan sebenyak *n* kali.
- Contoh: misalkan kita diminta untuk mencari sepasang bilangan di dalam sebuah list yang jika ditambahkan akan bernilai 0. Asumsikan list yang diberikan sudah terurut.
- Salah satu solusi ;
  - dengan menelusuri seluruh list, satu demi satu (kompleksitas: *n*)
  - mencari elemen yang bernilai invers dari elemen sekarang menggunakan binary search (kompleksitas: log *n*).

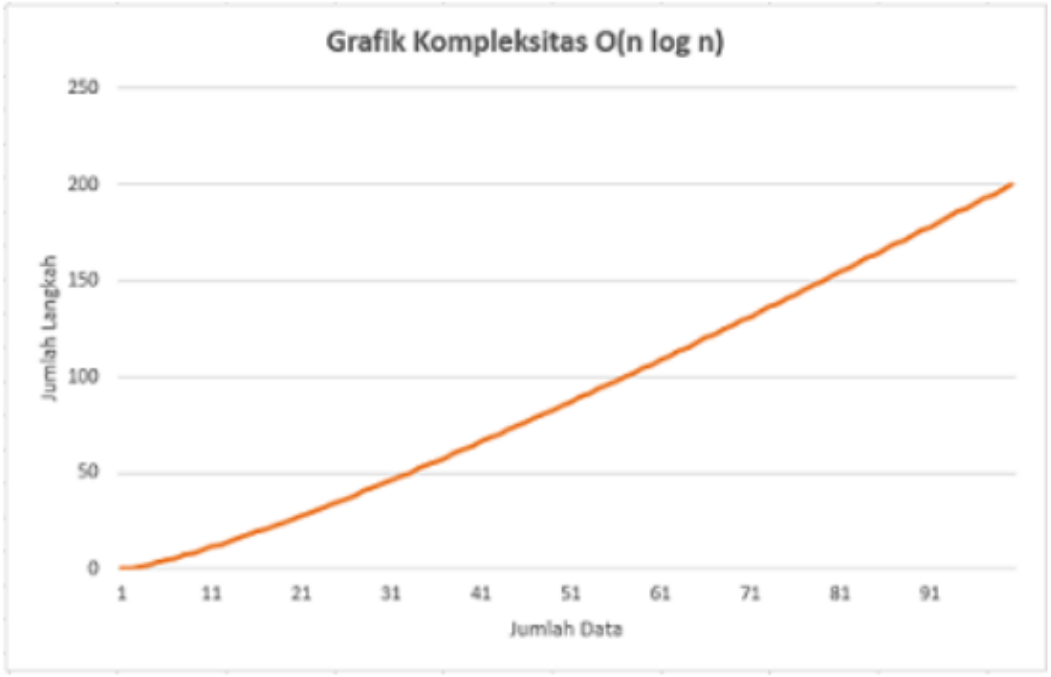
O(n log n)

- Contoh implementasi

```
def zero_sum(lst):
    n = len(lst)
    for i in range(0, n):
        j = binary_search(lst, -1 * lst[i])
        if j > i:
            n1 = str(lst[i])
            n2 = str(lst[j])
            print("Zero sum: " + n1 + " and " + n2 + "\n")
```

- Perhatikan bagaimana kita melakukan binary search sebanyak *n* kali, sehingga secara sederhana kompleksitas yang akan kita dapatkan adalah $n * \log n = n \log n$

O(n log n)



Tingkat Pertumbunan Algoritma Kompleksitas n log n

$O(n^m)$: Kompleksitas Polinomial

- Algoritma dengan kompleksitas polinomial merupakan salah satu kelas algoritma yang tidak efisien, karena memerlukan jumlah langkah penyelesaian yang jauh lebih besar daripada jumlah data.

- Contoh :...

$O(n^m)$: Kompleksitas Polinomial

- Contoh

```
def kali(a, b):
    res = 0
    for i in range(a):
        for j in range(b):
            res += 1
    return res
```

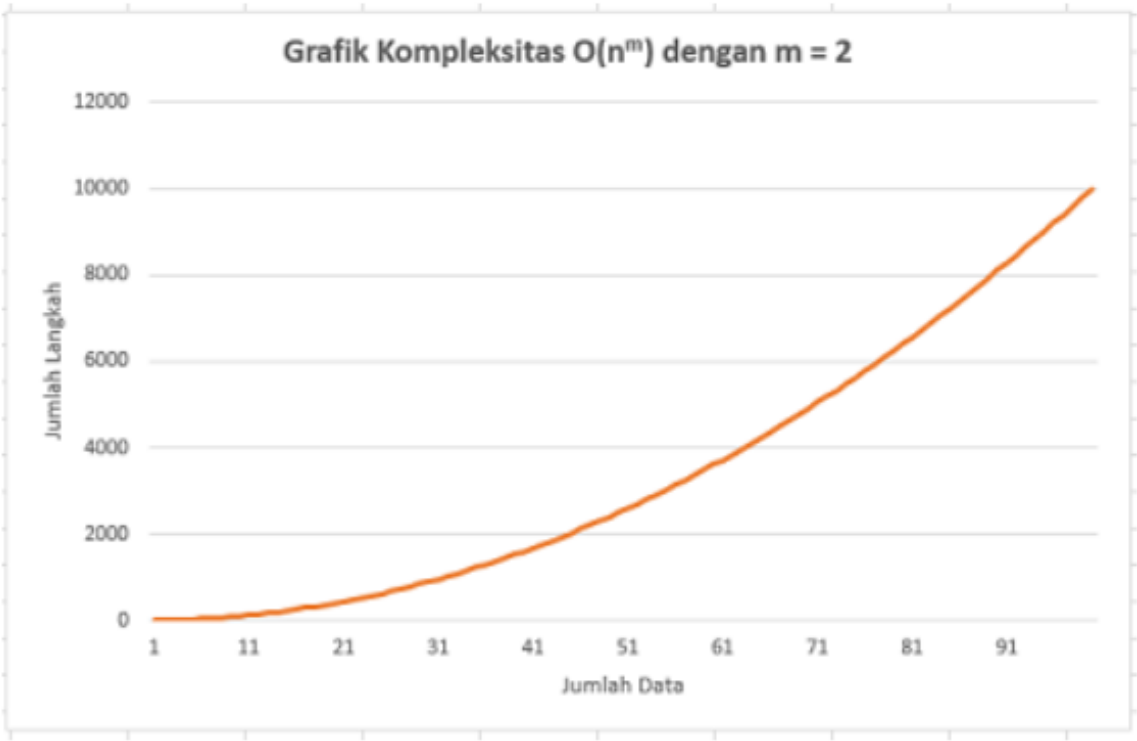
- Algoritma di atas melakukan perkalian antara *a* dan *b*, dengan melakukan penambahan 1 sebanyak *b* kali, yang hasilnya ditambahkan sebanyak *a* kali.

O(n^m): Kompleksitas Polinomial

Kode	Jumlah Langkah
`for i in range(b):`	a
`res += 1`	b

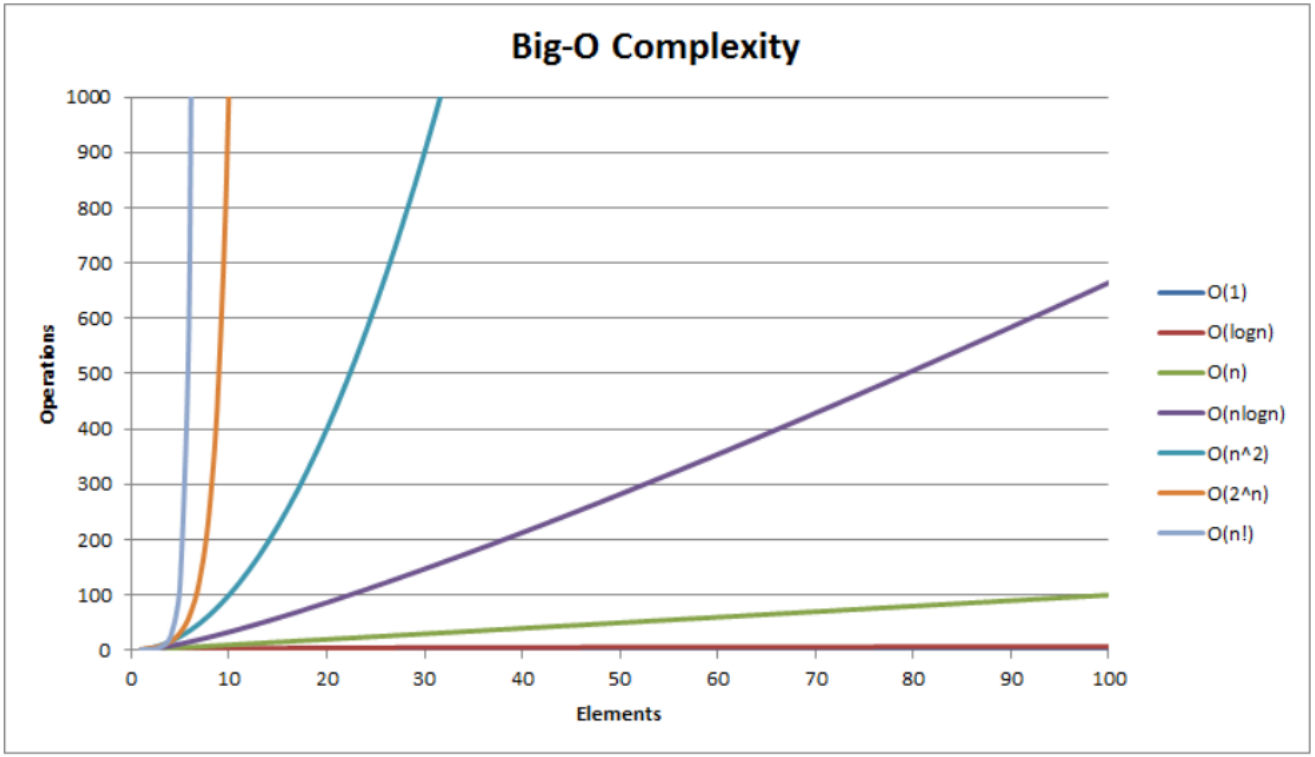
- Kompleksitas :a*b
- Jika a=b maka kompleksitasnya a^2, atau dapat ditulis sebagai n^2 yang diabstrakkan sebagai n^m,m=2

O(n^m): Kompleksitas Polinomial



Tingkat Pertumbunan Algoritma Kompleksitas Eksponensial

Perbandingan



Perbandingan Tingkat Pertumbuhan Tiap Kompleksitas

latihan

- Berapakah Big O untuk :
  - Algoritma Cari Elemen Terbesar
  - Algoritma Pengurutan