

# Modul 10

## Analisis Algoritma

Untuk setiap soal dan tantangan dalam pemrograman dan ilmu komputer, terdapat banyak algoritma yang bisa menyelesaikan soal dan tantangan itu. Pertanyaannya lalu: algoritma mana yang lebih cepat dan efisien? Algoritma mana yang lebih stabil? Algoritma mana yang lebih cocok untuk keadaan tertentu?

Analisis algoritma akan mempelajari pembandingan algoritma-algoritma berdasarkan banyaknya sumberdaya (memori, waktu, ...) yang dipakai tiap-tiap algoritma. Waktu yang diperlukan untuk menjalankan suatu algoritma seringkali bergantung pada banyaknya data yang diolah. Banyaknya data ini disimbolkan dengan  $n$ . Untuk suatu permasalahan, ada algoritma yang waktu pengerjaannya sebanding dengan  $n$  (saat  $n$  naik 10 kali, waktunya naik 10 kali). Untuk permasalahan yang lain, ada algoritma yang sebanding dengan  $n^2$  (saat  $n$  naik 10 kali, waktunya naik 100 kali). Ada pula yang sebanding dengan  $n \log n$ . Ada yang tidak bergantung pada  $n$ . Kita ingin algoritma yang efisien, yakni menjalankan tugasnya dengan benar dan secepat mungkin.

### 10.1 Sebuah contoh

#### Menjumlahkan bilangan 1 sampai $n$

Kamu diminta membuat program –memakai Python– yang akan menjumlahkan semua bilangan bulat dari 1 sampai  $n$ . Misal  $n = 10$ , maka si program akan mengembalikan 55 (karena  $55 = 1 + 2 + \dots + 10$ ). Bagaimana kamu membuatnya?

Cara pertama adalah cara ‘lugu’, yakni dengan menjumlahkannya betul-betul, seperti di bawah ini. (Ketiklah.)

```
1 def jumlahkan_cara_1(n):
2     hasilnya = 0
3     for i in range(1, n+1):
4         hasilnya = hasilnya + i
5     return hasilnya
```

Ketika dijalankan, tentu dia akan mengembalikan hasil yang diharapkan.

```
>>> jumlahkan_cara_1(10)
55
>>> jumlahkan_cara_1(100)
5050
```

Bagaimanakah kinerja (performance) algoritma ini? Jika  $n$  semakin besar, apakah perlu waktu yang lebih lama? Mestinya iya. Mari kita coba untuk mengukur waktu eksekusinya.

Pengukuran kinerja bisa dilakukan dengan berbagai cara. Salah satunya adalah dengan mendata waktu di awal eksekusi dan di akhir eksekusi, lalu membandingkan keduanya. Secara praktis, kita meng-import module `time` dan “membungkus” perintah eksekusi dengan penanda waktu. Tambah kode di atas untuk menjadi seperti di bawah ini.

```
1 import time
2
3 def jumlahkan_cara_1(n):
4     hasilnya = 0
5     for i in range(1, n+1):
6         hasilnya = hasilnya + i
7     return hasilnya
8
9 for i in range(5):          # mengulang lima kali
10    awal = time.time()       # menandai awal kerja
11    h = jumlahkan_cara_1(10000) # menjumlah 1 sampai sepuluh ribu
12    akhir = time.time()      # menandai akhir kerja, lalu mencetak
13    print("Jumlah adalah %d, memerlukan %.9f detik" % (h, akhir-awal))
```

Hasilnya adalah seperti di bawah ini (akan berbeda di komputermu):

```
Jumlah adalah 50005000, memerlukan 0.00400043 detik
Jumlah adalah 50005000, memerlukan 0.00200009 detik
Jumlah adalah 50005000, memerlukan 0.00199986 detik
Jumlah adalah 50005000, memerlukan 0.00099993 detik
Jumlah adalah 50005000, memerlukan 0.00200033 detik
>>>
```

Coba naikkan  $n$  menjadi satu juta (baris 11), lalu jalankan:

```
Jumlah adalah 500000500000, memerlukan 0.16300941 detik
Jumlah adalah 500000500000, memerlukan 0.15600896 detik
Jumlah adalah 500000500000, memerlukan 0.15900898 detik
Jumlah adalah 500000500000, memerlukan 0.15800905 detik
Jumlah adalah 500000500000, memerlukan 0.15700889 detik
>>>
```

Seperti diduga, perlu waktu lebih lama untuk menjumlahnya. Itu adalah cara pertama.

Sekarang kita akan mengetik algoritma yang lebih baik. Ingat bahwa jumlahan dari 1 sampai  $n$  bisa diringkas dengan rumus

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

sehingga kita mempunyai `jumlahkan_cara_2` sebagai berikut

```
1 def jumlahkan_cara_2(n):
2     return ( n*(n + 1) )/2
```

yang kalau kita tes dengan cara yang sama, kita akan mendapatkan seperti yang di bawah

```
Jumlah adalah 500000500000, memerlukan 0.00000001 detik
Jumlah adalah 500000500000, memerlukan 0.00000000 detik
>>>
```

Cobalah mengubah  $n$  menjadi seratus juta atau satu miliar. Apa yang terjadi? Waktu penggerjaannya tidak akan berubah. Algoritma di cara 2 ini tidak terpengaruh besarnya  $n$ .

Namun apa arti *benchmark* ini? Secara intuitif bisa kita lihat bahwa penyelesaian yang iteratif (cara 1) bekerja lebih banyak karena ada langkah program yang diulang-ulang. Juga, waktu yang diperlukan untuk menghitung di cara 1 bertambah dengan naiknya  $n$ .

Apakah angka kecepatan eksekusi di atas akan selalu di sekitar itu? Tidak. Jika kita menjalankan algoritma di atas di komputer yang berbeda, atau menerapkannya dengan bahasa pemrograman yang berbeda, kemungkinan kita akan mendapatkan hasil waktu yang berbeda. Kalau komputer yang dipakai lebih tua, waktu eksekusinya bisa jadi lebih lama.

Dengan demikian kita memerlukan sebuah alat ukur yang lebih baik untuk meng-karakterisasi algoritma-algoritma ini, dikaitkan dengan kecepatan eksekusinya. Karakterisasi ini terlepas dari bahasa pemrograman atau komputer yang dipakai. Pengukuran ini dengan demikian berguna untuk membandingkan antar algoritma yang implementasinya bisa di mana saja.

#### Pelajaran terpetik

- Dari contoh di atas, kita bisa melihat bahwa untuk suatu masalah, bisa jadi ada dua atau lebih algoritma bisa menyelesaikan. *Namun di antaranya ada yang lebih baik.*
- Kita ingin algoritma yang kecepatan pertumbuhannya (*growth rate*-nya) lambat. Artinya, dengan membesarnya  $n$  (yakni ukuran input), waktu eksekusinya tidak terpengaruh banyak.
- Kita memerlukan suatu konsep –berikut notasinya– untuk meng-karakterisasi algoritma-algoritma, sehingga kita bisa membandingkan antar algoritma dengan konsisten.

## 10.2 Notasi Big-O

Untuk mulai mengukur secara kuantitatif kinerja suatu algoritma, kita bisa saja mulai dari menghitung banyaknya operasi atau langkah yang dijalani, sebagai fungsi dari  $n$ . Sebut saja fungsi yang meng-aproksimasi ini adalah  $T(n)$ . Parameter  $n$  sering disebut “besarnya problem” (*size of the problem*).

### Ilustrasi $\mathcal{O}(f(n))$

Untuk algoritma cara 1 di atas, bisa kamu lihat bahwa  $T(n) = 1 + n$ . Bisa kita baca sebagai “ $T(n)$  adalah waktu yang diperlukan untuk menyelesaikan problem ini, dalam hal ini sebanyak  $n + 1$ ”. Dengan bertambah besarnya  $n$ , maka angka 1 nya menjadi semakin kurang signifikan. Maka kita katakan bahwa algoritma ini, dalam notasi big-O, mempunyai *running time*  $\mathcal{O}(n)$ .

Sebuah contoh lain. Misal suatu algoritma mempunyai banyak operasi<sup>1</sup> yang dirumuskan dengan  $T(n) = 5n^2 + 32n + 1440$ . Saat  $n$  kecil, misal 1 atau 2 atau 3, konstanta 1440 tampak dominan. Tapi dengan membesarnya  $n$ , maka suku  $n^2$  menjadi paling dominan. Saat  $n$  menjadi besar sekali, dua suku yang lain menjadi tidak signifikan untuk menentukan hasil akhir. Sehingga untuk menaproksimasi  $T(n)$  saat  $n$  cukup besar, dua suku lain itu bisa diabaikan dan kita fokus pada  $5n^2$ . Terlebih lagi, bila kita membandingkan kenaikan  $T(n)$  saat  $n$  berubah dari nilai besar ke nilai yang lebih besar lagi, maka koefisian angka 5 ini menjadi bisa diabaikan dan *running time* algoritma ini adalah  $\mathcal{O}(n^2)$ .

Kompleksitas waktu yang sering muncul dalam analisis algoritma adalah  $\mathcal{O}(1)$ ,  $\mathcal{O}(\log n)$ ,  $\mathcal{O}(\sqrt{n})$ ,  $\mathcal{O}(n)$ ,  $\mathcal{O}(n \log n)$ ,  $\mathcal{O}(n^2)$ ,  $\mathcal{O}(n^d)$ , dan  $\mathcal{O}(2^n)$ . Kita akan kembali ke sini lagi nanti.

BTW, algoritma cara 2 di atas mempunyai kompleksitas  $\mathcal{O}(1)$ .

### Formulasi

Notasi big-O bermakna “*order of magnitude*” dan ditulis dalam bentuk  $\mathcal{O}(f(n))$ . Idenya adalah kita ingin dapat mengungkapkan bagaimana waktu eksekusi suatu algoritma tumbuh semakin besar dengan naiknya  $n$ . Misal di satu contoh di atas, kita bisa mengatakan “algoritma ini *running time*-nya tumbuh sejalan dengan  $n$  kuadrat.” Atau kita bisa mengungkapkan “Algoritma pertama *tumbuhnya* lebih cepat daripada algoritma kedua –artinya algoritma kedua lebih baik.”

Awalnya adalah mencari sebuah fungsi yang dapat “mengatasi” atau “mengalahkan”  $T(n)$  saat  $n$  membesar, dan sekaligus sedekat mungkin terhadap  $T(n)$ .

**Definisi 10.1** Misalkan  $T(n)$  dan  $f(n)$  sebagai fungsi yang memetakan dari bilangan integer positif ke bilangan real. Maka kita katakan bahwa  $T(n)$  mempunyai order  $\mathcal{O}(f(n))$  jika ada konstanta real  $c$  dan konstanta integer  $n_0 \geq 1$  sedemikian rupa sehingga

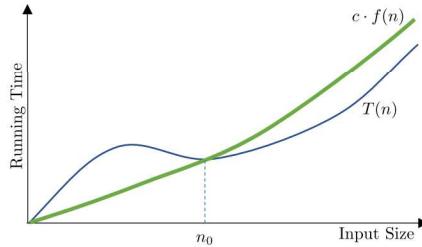
$$T(n) \leq c \cdot f(n), \quad \text{untuk } n \geq n_0.$$

Lihat Gambar 10.1. □

Sebagai tambahan, kita akan mencari suatu fungsi  $f(n)$  yang *paling simple* yang lalu  $c \cdot f(n)$  bisa menjadi *batas atas* yang paling ketat bagi fungsi  $T(n)$ .

---

<sup>1</sup>Untuk saat ini, tidak penting benar jenis operasinya; penjumlahan atau perkalian atau apapun. Yang penting bagaimana waktu eksekusinya *tumbuh* dengan semakin besarnya ukuran input.



**Gambar 10.1:** Ilustrasi notasi Big-O. Fungsi  $T(n)$  berada dalam  $\mathcal{O}(f(n))$ , karena  $T(n) \leq c \cdot f(n)$  untuk semua  $n \geq n_0$ . Dengan kata lain: untuk semua  $n$  di sebelah kanan  $n_0$ , nilai  $c \cdot f(n)$  selalu ‘mengalahkan’  $T(n)$ . Sebagai tambahan,  $c \cdot f(n)$  haruslah sekertai mungkin ke  $T(n)$ , dan fungsi  $f(n)$  haruslah se-simple mungkin.

**Contoh 10.2** Fungsi  $T(n) = 25n^4 + 8n^3 + 7n^2 + 9n + 4$  mempunyai order  $\mathcal{O}(n^4)$ .

Perhatikan bahwa  $25n^4 + 8n^3 + 7n^2 + 9n + 4 \leq (25 + 8 + 7 + 9 + 4)n^4 = cn^4$  untuk  $c = 53$  dan  $n_0 = 1$ .  $\square$

Sesungguhnya, kita bisa mengkarakterisasi kecepatan pertumbuhan semua fungsi polinomial.

**Proposisi 10.3** Jika  $T(n)$  adalah polinomial dengan derajat  $d$ , yakni,

$$T(n) = a_d n^d + a_{d-1} n^{d-1} + \dots + a_1 n + a_0,$$

di mana  $a_d > 0$ , maka  $T(n)$  berada dalam  $\mathcal{O}(n^d)$ .

**Sketsa bukti.** Perhatikan bahwa untuk  $n \geq 1$ , kita mempunyai  $n^d \geq \dots \geq n^2 \geq n \geq 1$ . Sehingga

$$a_d n^d + a_{d-1} n^{d-1} + \dots + a_2 n^2 + a_1 n + a_0 \leq (|a_d| + |a_{d-1}| + \dots + |a_2| + |a_1| + |a_0|) n^d.$$

Lalu kita tunjukkan bahwa  $f(n)$  berada dalam  $\mathcal{O}(n^d)$  dengan memilih konstanta  $c = |a_d| + \dots + |a_1| + |a_0|$  dan  $n_0 = 1$ .  $\square$

Jadi, pangkat tertinggilah yang menentukan kecepatan pertumbuhan polinomial itu. Di bawah ini ada beberapa contoh lagi, yang melibatkan fungsi fundamental lain. Kita bersandar pada fakta bahwa  $\log n \leq n$  untuk  $n \geq 1$ . Di sini, fungsi  $\log$  mempunyai basis 2. Jadi, sebagai contoh,  $\log 32 = 5$  karena  $2^5 = 32$ .

**Contoh 10.4**  $3n^2 + 5n \log n + 6n + 7$  berada dalam  $\mathcal{O}(n^2)$ .  $\square$

**Contoh 10.5**  $12n^3 + 10n \log n + 9$  berada dalam  $\mathcal{O}(n^3)$ .  $\square$

**Contoh 10.6**  $5 \log n + 3$  berada dalam  $\mathcal{O}(\log n)$ .

Perhatikan bahwa  $8 \log n \leq 5 \log n + 3$  untuk  $n \geq 2$ . Ingat bahwa  $\log n$  bernilai 0 saat  $n = 1$ . Itulah mengapa kita menggunakan  $n \geq n_0 = 2$  di sini.  $\square$

**Tabel 10.1:** Beberapa fungsi umum  $\mathcal{O}(f(n))$ , diurutkan dari waktu eksekusi cepat (kompleksitas rendah) ke waktu eksekusi semakin lambat (kompleksitas tinggi). Di sini log bermakna  $\log_2$ , yakni logaritma berbasis 2.

$f(n)$	Nama umum	Contoh algoritma
1	constant	Append, get item, set item. Menentukan suatu bilangan genap atau ganjil. Menemukan nilai median di suatu list yang sudah urut. Menghitung $(-1)^n$ .
$\log n$	logarithmic	Menemukan suatu item di sebuah list yang sudah urut menggunakan <i>binary search</i> .
$n$	linear	<i>Copy, insert, delete, iteration.</i> Menemukan sebuah item di sebuah list yang tidak urut.
$n \log n$	log-linear	Merge sort dan Quick sort (kasus rata-rata). <i>Fast Fourier Transform</i> .
$n^2$	quadratic	Mengalikan dua bilangan $n$ digit dengan algoritma sederhana. Selection sort. Insertion sort. Batas (di kasus terburuk) untuk algoritma yang biasanya cepat seperti Quick sort.
$n^3$	cubic	<i>Tree-adjoining grammar parsing.</i> Perkalian dua matriks $n \times n$ dengan ‘cara lugu’.
$2^n$	exponential	Solusi exact untuk <i>travelling salesman problem</i> memakai <i>dynamic programming</i> . <i>Towers of Hanoi problem</i> .

**Contoh 10.7**  $2^{n+2}$  berada dalam  $\mathcal{O}(2^n)$ .

Perhatikan bahwa  $2^{n+2} = 2^2 \cdot 2^n = 4 \cdot 2^n$ . □

**Contoh 10.8**  $2n + 35\log n$  berada dalam  $\mathcal{O}(n)$ . □

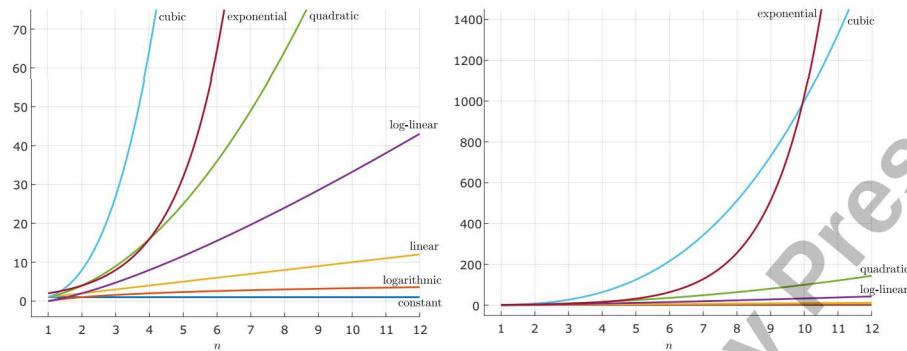
### Fungsi yang paling simple dan juga paling ketat

Umumnya kita harus menemukan notasi  $\mathcal{O}(\cdot)$  yang sedekat mungkin dengan fungsi yang akan dijelaskan. Sebagai contoh, jika kita punya  $T(n) = 5n^3 + 3n^2 + 4n + 5$ , maka bisa saja kita katakan bahwa  $T(n)$  berada dalam  $\mathcal{O}(n^5)$  atau berada dalam  $\mathcal{O}(n^4)$ . Tapi akan lebih tepat kalau kita katakan  $T(n)$  berada dalam  $\mathcal{O}(n^3)$ . Analoginya seperti di bawah.

Ada yang bertanya, “berapa lamakah berjalan dari Gedung J Lantai 1 ke Masjid Sudalmiyah Rais?” Ini bisa dijawab: “kurang dari sehari,” atau “tidak lebih dari 1 jam.” Ini secara teknis benar semua. Tapi akan lebih tepat (dan lebih membantu) kalau dijawab “berjalanlah ke arah utara dan kamu akan sampai dalam waktu kurang dari dua menit.”

### Perbandingan Fungsi-fungsi $\mathcal{O}(f(n))$ yang Umum

Terdapat beberapa fungsi yang umum ditulis dalam analisis algoritma. Ini ditampilkan dalam Tabel 10.1 dan grafik pertumbuhannya ditampilkan pada Gambar 10.2.



**Gambar 10.2:** Pertambahan kompleksitas / waktu ekskusi dengan membesarnya  $n$ . Kiri: zoom-in sumbu  $y$  grafik. Kanan: zoom-out sumbu  $y$  grafik. Tampak bahwa eksponensial pada suatu saat akan ‘mengatasi’ kubik. Lebih umum lagi –ini suatu fakta– adalah bahwa fungsi eksponensial akan ‘mengatasi’ fungsi polynomial  $n^d$  apapun, jika  $n$  cukup besar. Jadi, suatu kabar buruk jika suatu algoritma mempunyai kompleksitas eksponensial  $\mathcal{O}(2^n)$ .

### 10.3 Kasus terburuk, rata-rata, dan terbaik

Kinerja suatu algoritma seringkali bergantung pada nilai sebenarnya di datanya, tidak semata-mata pada ukuran/banyaknya data. Untuk algoritma yang seperti ini, kita perlu membedakan kinerjanya saat menghadapi kasus terbaik (*best case*), kasus terburuk (*worst case*), atau kasus rata-rata (*average case*). Ada algoritma yang pada kasus terburuk mempunyai kinerja  $\mathcal{O}(n^2)$ , namun pada kasus rata-rata kinerjanya adalah  $\mathcal{O}(n \log n)$ .

Sebagai contoh, untuk sebuah algoritma pengurutan<sup>2</sup>,

- Kasus terbaik adalah saat datanya sudah urut atau hampir urut.
- Kasus terburuk adalah saat data yang diterima terbalik urutannya.
- Kasus rata-rata adalah saat datanya acak.

Mari kita lihat perbedaan ini pada algoritma *Insertion Sort* yang sudah dibahas di Bab 5. Kita mulai dari *average case scenario* dulu, di mana data yang masuk berposisi acak. Siapkan kode *Insertion Sort* yang sudah kamu buat (dengan meng-import atau mengkopi kodennya), lalu ketik yang di bawah ini.

```

1 for i in range(5):
2     L = list(range(3000))
3     random.shuffle(L)      # Mengacak posisi elemen di list
4     awal = time.time()
5     U = insertionSort(L)
6     akhir = time.time()
7     print("Mengurutkan %d bilangan, memerlukan %.7f detik" % (len(L), akhir-awal))

```

Yang akan menghasilkan

<sup>2</sup>Tidak semua algoritma pengurutan mempunyai sifat seperti ini. Ada yang tidak terpengaruh urutan awal. Ada yang jika di awal urutannya terbalik, maka ini justru termasuk salah satu kasus terbaik.

```
Mengurutkan 3000 bilangan, memerlukan 1.1620665 detik
Mengurutkan 3000 bilangan, memerlukan 1.1730671 detik
Mengurutkan 3000 bilangan, memerlukan 1.1550660 detik
Mengurutkan 3000 bilangan, memerlukan 1.1370649 detik
Mengurutkan 3000 bilangan, memerlukan 1.1420655 detik
>>>
```

Sesudah itu, kita coba yang *worst case scenario* untuk algoritma ini. Yakni data yang masuk urutannya terbalik. Ikuti yang berikut.

```
1 for i in range(5):
2     L = list(range(3000))
3     L = L[::-1]          # Membalik urutan elemen di list
4     awal = time.time()
5     U = insertionSort(L)
6     akhir = time.time()
7     print("Mengurutkan %d bilangan, memerlukan %8.7f detik" % (len(L), akhir-awal))
```

yang menghasilkan

```
Mengurutkan 3000 bilangan, memerlukan 2.2251272 detik
Mengurutkan 3000 bilangan, memerlukan 2.2221272 detik
Mengurutkan 3000 bilangan, memerlukan 2.2151270 detik
Mengurutkan 3000 bilangan, memerlukan 2.2291274 detik
Mengurutkan 3000 bilangan, memerlukan 2.2141266 detik
>>>
```

Sekarang kita coba *best case scenario*, di mana data masukannya sudah sejak awal urut. Ikuti yang berikut ini.

```
1 for i in range(5):
2     L = list(range(3000))
3
4     awal = time.time()
5     U = insertionSort(L)
6     akhir = time.time()
7     print("Mengurutkan %d bilangan, memerlukan %8.7f detik" % (len(L), akhir-awal))
```

yang menghasilkan

```
Mengurutkan 3000 bilangan, memerlukan 0.0020003 detik
Mengurutkan 3000 bilangan, memerlukan 0.0009999 detik
Mengurutkan 3000 bilangan, memerlukan 0.0020001 detik
Mengurutkan 3000 bilangan, memerlukan 0.0020003 detik
Mengurutkan 3000 bilangan, memerlukan 0.0020001 detik
>>>
```

Terlihat bahwa *insertion sort* mempunyai kinerja yang berbeda-beda antara *average case*, *worst case*, dan *best case*.

Tidak semua algoritma pengurutan mempunyai sifat seperti ini. Fungsi pengurutan yang ada di Python, `sorted()`, yang memakai algoritma Timsort<sup>3</sup>, mempunyai sifat yang menarik. Lihat bagian latihan di akhir modul.

<sup>3</sup><https://en.wikipedia.org/wiki/Timsort>

## 10.4 Menganalisis kode Python

Seperti sudah diterangkan sebelumnya, analisis algoritma bermula dari operasi-operasi dasar (*basic operations*). Tapi apa yang dimaksud dengan operasi dasar? Operasi dasar adalah pernyataan dan pemanggilan fungsi yang waktu eksekusinya tidak bergantung pada nilai spesifik data yang sedang dimanipulasi. Berikut ini adalah contoh operasi dasar, tiap-tiap barisnya.

```
x = 5
y = x
z = x + y*8
d = x > 0 and x < 100
f = [3,2,4,5]
v = f[0:2]
```

Ketika ada dua operasi yang berurutan, maka kompleksitasnya dijumlahkan. Sebagai contoh, perhatikan dua operasi berurutan: penyisipan suatu item ke sebuah list dan lalu mengurutkan list itu. Kita tahu penyisipan satu item memakan waktu  $\mathcal{O}(n)$  dan pengurutan memerlukan waktu  $\mathcal{O}(n \log n)$ . Total kompleksitas waktunya adalah menjadi  $\mathcal{O}(n + \log n)$ . Namun kita hanya memperhatikan fungsi yang order-nya lebih tinggi<sup>4</sup>, sehingga kompleksitasnya kita tulis menjadi  $\mathcal{O}(n \log n)$ .

Jika kita mengulang suatu operasi, misal di `while`-loop atau `for`-loop, maka kita mengalikan kelas kompleksitasnya dengan banyaknya operasi.

Sebagai contoh, misal fungsi `myFun()` mempunyai kompleksitas waktu  $\mathcal{O}(n^2)$ , dan dieksekusi sebanyak  $n$  kali di dalam sebuah for-loop:

```
for i in range(n):
    myFun(...)
```

maka kompleksitas waktunya adalah  $\mathcal{O}(n \cdot n^2) = \mathcal{O}(n^3)$ .

Jika kita mempunyai loop di dalam loop (*nested loop*), maka kodennya akan dijalankan  $n^2$  kali, dengan anggapan kedua loop berjalan  $n$  kali. Sebagai contoh:

```
for i in range(n):
    for j in range(n):
        # pernyataan
```

Jika setiap pernyataan adalah konstan, dan dijalankan  $n \cdot n$  kali, maka bisa kita tulis bahwa kompleksitasnya adalah  $\mathcal{O}(n^2)$ .

Kode berikut mempunyai kompleksitas  $\mathcal{O}(\log n)$

```
count = 0
i = 32
while i >= 1:
    count = count + 1
    i = i//2
print(count)
```

Perhatikan bahwa setiap kali menghitung, nilai  $i$  dibagi 2. Ini akan membuat programnya lebih cepat selesai dibandingkan jika  $i$  dikurangi satu tiap penghitungan. Cobalah eksekusi kode di

<sup>4</sup>Yakni, fungsi yang ‘mengatasi’ saat  $n$  besar.

atas dengan  $n = 100$ . Di sini ukuran input dikecilkan jadi separuhnya setiap kali dijalankan ulang, sehingga untuk mencapai ukuran 1 banyaknya iterasi yang dibutuhkan adalah

$$\lfloor \log_2 n \rfloor + 1$$

yakni bilangan bulat terbesar yang kurang dari atau sama dengan<sup>5</sup>  $\log_2 n$ , tambah 1. Jika  $n = 100$  maka akan ada iterasi sebanyak  $\lfloor \log_2 100 \rfloor + 1 = 6 + 1 = 7$ .

## 10.5 Analisis Pewaktuan Menggunakan timeit

Module `timeit` dapat membantu untuk secara praktis melakukan pengukuran waktu eksekusi untuk kode-kode sederhana. Perhatikan dan ketiklah kode berikut, serta pahami cara kerjanya.

```
>>> from timeit import timeit
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000)
0.00342316301248502
>>> timeit('sqrt(2)', 'from math import sqrt', number=100000)
0.019461828997009434
>>> timeit('sqrt(2)', 'from math import sqrt', number=1000000)
0.1989576570049394
```

Pada contoh di atas, argumen pertama pada `timeit` adalah kode yang dijalankan / yang sedang dites. Argumen kedua, `setup`, adalah persiapan sebelum kode yang dites dijalankan (persiapan ini hanya sekali). Argumen kata kunci `number` adalah banyaknya pengulangan, default-nya satu juta jika argumen ini tidak diikutkan. Fungsi ini mengembalikan waktu dalam detik. Lanjutkan eksplorasi kamu dengan `timeit` ini.

```
>>> timeit("1+2") # Waktu untuk menghitung 1+2, diulang 1 juta kali.
0.01901585698942654
>>> timeit("sin(pi/3)", setup="from math import sin, pi")# sin(pi/3) diulang 1 juta kali
0.2774660010036314
>>> timeit("sin(1.047)", setup="from math import sin")# sin(1.047) diulang 1 juta kali
0.1839345560001675
```

Yang `sin(1.047)` lebih cepat dari `sin(pi/3)` karena tidak perlu operasi pembagian.

### 10.5.1 Melihat $\mathcal{O}(n^2)$ pada *nested loop*

Pada bagian ini kita akan menggambar hasil uji pewaktuan sebuah fungsi dengan kompleksitas  $\mathcal{O}(n^2)$ . Pastikan modul `matplotlib` terinstall di komputermu. Jika belum ada, install dari internet: `pip install matplotlib`. Gagasan dasarnya adalah

- Ada fungsi yang akan kita uji. Fungsi ini menerima satu argumen, dan mempunyai kalang bersusul (*nested loop*).
- Kita buat sebuah kalang lain, yang nilai iterasi-nya berjalan dari 1 sampai 1000, yang dipakai untuk menguji fungsi tersebut di atas. Nilai iterasi ini akan diumpulkan ke fungsi

<sup>5</sup>Lambang  $[x]$ , perintah `math.floor` di Python, bermakna “ $x$ , jika bukan bilangan bulat, akan dibulatkan ke bilangan bulat di bawahnya” yakni *menuju lantai*. Jika ingin menuju bilangan bulat di atasnya, orang menggunakan  $[x]$ , perintah `math.ceil` di Python: *menuju plafon*.

yang sedang diuji.

- Untuk tiap iterasi, eksekusinya hanya satu kali. Jadi, `number = 1` di `timeit`. Dan lamanya waktu eksekusi dicatat untuk tiap  $i$ , menggunakan `timeit`.
- Setelah pengujian berakhir, buat grafik yang menggambarkan lamanya waktu eksekusi dari data kecil ke besar.
- Tumpukkan grafik  $x^2$  – yang sudah diskala– sebagai pembanding.

Kodenya ada di bawah ini, dan hasilnya ditunjukkan di Gambar 10.3. Ketik dan jalankan.

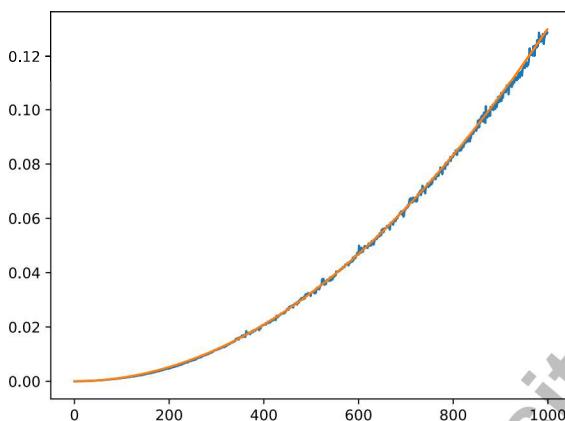
```

1 import timeit
2 import matplotlib.pyplot as plt
3
4 ## Ini fungsi nested loop yang akan diuji:
5 def kalangBersusuh(n):
6     for i in range(n):
7         for j in range(n):
8             i+j
9
10 ## Ini fungsi pengujinya:
11 def ujiKalangBersusuh(n):
12     ls=[]
13     jangkauan = range(1,n+1)
14     siap = "from __main__ import kalangBersusuh"
15     for i in jangkauan:
16         print('i =',i) # baris ini bisa dihidupkan atau dimatikan
17         t=timeit.timeit("kalangBersusuh(" + str(i) +")", setup=siap, number=1)
18         ls.append(t)
19     return ls
20
21 ## Pemanggilan pengujian:
22 n = 1000
23 LS = ujiKalangBersusuh(n) # dari 1 sampai 1000.
24 ## LS adalah list hasil uji kecepatan, dari n sedikit ke banyak.
25
26 ## Menggambar grafik. Di bawah ini saja yang diulang saat me-nyetel skala.
27 plt.plot(LS) # Mem-plot hasil uji
28 skala = 7700000 # ----- Setel skala ini sesuai hasilmu.
29 plt.plot([x*x/skala for x in range(1,n+1)]) # Grafik x^2 untuk pembanding.
30 plt.show() # Tunjukkan plotnya

```

Hasil grafik sesuai dengan ekspektasi kita. Perlu diingat bahwa ini mewakili kinerja algoritmanya sekaligus perilaku platform perangkat keras dan lunaknya.

Tentu saja, processor yang lebih cepat akan menghasilkan waktu eksekusi yang lebih cepat. Kinerja juga akan dipengaruhi oleh proses lain yang sedang berjalan, oleh kekangan memori, oleh kecepatan clock, dan lain sebagainya. Namun kenaikan waktu eksekusi relatif dari  $n$  yang kecil ke  $n$  yang lebih besar akan menunjukkan trend yang kurang lebih seragam. Pemilihan algoritma tidak hanya melihat  $\mathcal{O}(\cdot)$ , tapi juga kecepatan sebenarnya untuk data yang *typical* akan dipakai saat *production*.



Gambar 10.3: Hasil ujicoba kalang bersusuh, yang mempunyai kompleksitas  $\mathcal{O}(n^2)$ . Hasil akan berbeda angkanya untuk komputer yang berbeda, namun bentuk grafik akan kurang lebih sama.

## 10.6 Soal-soal untuk Mahasiswa

Sebelum mengerjakan soal-soal di bawah, pastikan bahwa kamu paham semua contoh soal di modul ini.

1. Kerjakan ulang contoh dan latihan di modul ini menggunakan modul `timeit`, yakni  
(a) `jumlahkan_cara_1` (b) `jumlahkan_cara_2` (c) `insertionSort`

Untuk `insertionSort`, kerjakan untuk ketiga kasusnya.

2. Python mempunyai perintah untuk mengurutkan suatu list yang memanfaatkan algoritma Timsort. Jika `g` adalah suatu list berisi bilangan, maka `g.sort()` kan mengurutkannya. Perintah yang lain, `sorted()` mengurutkan list dan *mengembalikan* sebuah list baru yang sudah urut. Selidikilah fungsi `sorted` ini menggunakan `timeit`:

- Apakah yang merupakan *best case* dan *average case* bagi `sorted()`?
- Confirm bahwa data input urutan terbalik *bukan* kasus terburuk bagi `sorted()`. Bahkan dia lebih cepat dalam mengurutkannya daripada data input random.

3. Untuk tiap kode berikut, tentukan *running time*-nya,  $\mathcal{O}(1)$ ,  $\mathcal{O}(\log n)$ ,  $\mathcal{O}(n)$ ,  $\mathcal{O}(n \log n)$ ,  $\mathcal{O}(n^2)$ , atau  $\mathcal{O}(n^3)$ , atau yang lain. Untuk memulai analisis, ambil suatu nilai  $n$  tertentu, lalu ikuti apa yang terjadi di kode itu.

- (a) loop di dalam loop, keduanya sebanyak  $n$ :

```
test = 0
for i in range(n):
    for j in range(n):
        test = test + i * j
```

- (b) loop di dalam loop, yang dalam bergantung nilai  $i$  loop luar:

```
test = 0
```

```
for i in range(n):
    for j in range(i):
        test = test + i * j
```

(c) dua loop terpisah:

```
test = 0
for i in range(n):
    test = test + 1
for j in range(n):
    test = test - 1
```

(d) *while* loop yang dipangkas separuh tiap putaran:

```
i = n
while i > 0:
    k = 2 + 2
    i = i // 2
```

(e) loop in a loop in a loop, ketiganya sebanyak  $n$ :

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            m = i + j + k + 2019
```

(f) loop in a loop in a loop, dengan loop dalam sebanyak nilai loop luar terdekat:

```
for i in range(n):
    for j in range(i):
        for k in range(j):
            m = i + j + k + 2019
```

(g) fungsi ini:

```
for i in range( n ) :
    if i % 3 == 0 :
        for j in range( n / 2 ) :
            sum += j
    elif i % 2 == 0 :
        for j in range( 5 ) :
            sum += j
    else :
        for j in range( n ) :
            sum += j
```

4. Urutkan dari yang pertumbuhan kompleksitasnya lambat ke yang cepat:

$$n \log_2 n \quad 4^n \quad 10 \log_2 n \quad 5n^2 \quad \log_4 n \quad 12n^6 \quad 2^{\log_2 n} \quad n^3$$

5. Tentukan  $\mathcal{O}( \cdot )$  dari fungsi-fungsi berikut, yang mewakili banyaknya langkah yang diperlukan untuk beberapa algoritma.

(a)  $T(n) = n^2 + 32n + 8$

(c)  $T(n) = 4n + 5n \log n + 102$

(b)  $T(n) = 87n + 8n$

(d)  $T(n) = \log n + 3n^2 + 88$

(e)  $T(n) = 3(2^n) + n^2 + 647$       (g)  $T(n, k) = 8n + k \log n + 800$

(f)  $T(n, k) = kn + \log k$       (h)  $T(n, k) = 100kn + n$

6. (Literature Review) Carilah di Internet, kompleksitas metode-metode pada object `list` di Python. *Hint:*

- Google `python list methods complexity`. Lihat juga bagian "Images"-nya
- Kunjungi <https://wiki.python.org/moin/TimeComplexity>

7. Buatlah suatu ujicoba untuk mengkonfirmasi bahwa metode `append()` adalah  $\mathcal{O}(1)$ . Gunakan `timeit` dan `matplotlib`, seperti sebelumnya.

8. Buatlah suatu ujicoba untuk mengkonfirmasi bahwa metode `insert()` adalah  $\mathcal{O}(n)$ . Gunakan `timeit` dan `matplotlib`, seperti sebelumnya.

9. Buatlah suatu ujicoba untuk mengkonfirmasi bahwa untuk memeriksa apakah-suatu-nilai-berada-di-suatu-list mempunyai kompleksitas  $\mathcal{O}(n)$ . Gunakan `timeit` dan `matplotlib`, seperti sebelumnya.

10. (Literature Review) Carilah di Internet, kompleksitas metode-metode pada object `dict` di Python.

11. (Literature Review) Selain notasi big-O  $\mathcal{O}(\cdot)$ , ada pula notasi big-Theta  $\Theta(\cdot)$  dan notasi big-Omega  $\Omega(\cdot)$ . Apakah beda di antara ketiganya?

12. (Literature Review) Apa yang dimaksud dengan *amortized analysis* dalam analisis algoritma?

*Soal-soal mengambil inspirasi dari buku-buku yang tercantum di Daftar Bacaan.*

## Daftar Bacaan

- [1] Benjamin Baka. *Python Data Structures and Algorithms: Improve the performance and speed of your applications.* Packt Publishing, 2017.
- [2] Wesley J. Chun. *Core Python Programming.* Prentice Hall, New Jersey, 2<sup>nd</sup> edition, 2006.
- [3] Wesley J. Chun. *Core Python Applications Programming.* Prentice Hall, New Jersey, 3<sup>rd</sup> edition, 2012.
- [4] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python.* John Wiley & Sons, New Jersey, 2013.
- [5] Narasimha Karumanchi. *Data Structures and Algorithmic Thinking in Python.* CareerMonk Publications, Bombay, 2016.
- [6] Kenneth A. Lambert and Martin Osborne. *Fundamentals of Python: From First Programs Through Data Structures.* John Wiley & Sons, New Jersey, 2011.
- [7] Kent D. Lee and Steve Hubbard. *Data Structures and Algorithms with Python.* Springer, Cham, Switzerland, 2015.
- [8] Brad N. Miller and David L. Ranum. *Problem Solving with Algorithms and Data Structures Using Python.* Franklin, Beedle & Associates, 2<sup>nd</sup> edition, 2011.
- [9] Rance D. Necaise. *Data Structures and Algorithms Using Python.* John Wiley & Sons, New Jersey, 2011.