

**D a s a r - d a s a r**  
**Algoritma**  
**Struktur Data**  
**Pemrograman**  
— dengan —  
**Bahasa Singkong**

**Dr. Noprianto**  
**Budiman**

**PT. Stabil Standar Sinergi**

# **Dasar-dasar Algoritma, Struktur Data, dan Pemrograman dengan Bahasa Singkong**

Penulis: Dr. Noprianto, Budiman

ISBN: 978-602-52770-8-5

Penerbit:

PT. Stabil Standar Sinergi

Alamat	Puri Indah Financial Tower Lantai 6, Unit 0612 Jl. Puri Lingkar Dalam Blok T8, Puri Indah, Kembangan, Jakarta Barat 11610
Website	<a href="http://www.singkong.dev">www.singkong.dev</a>
Email	<a href="mailto:info@singkong.dev">info@singkong.dev</a>

Hak cipta dilindungi undang-undang.

## Daftar Isi

Kata Pengantar .....	5
Persiapan .....	7
Mengurutkan Daftar Bilangan .....	9
Contoh Algoritma Greedy.....	41
Memulai Pemrograman Komputer .....	49
Profesional atau membantu pekerjaan.....	51
Mengetahui dan menggunakan sistem operasi.....	52
Mempelajari bahasa pemrograman.....	52
Bekerja dalam tim.....	54
Bekerja Dengan Array.....	57
Membuat Array .....	58
Bekerja Dengan Indeks.....	60
Iterasi Array .....	62
Menambahkan atau Mengurangi Elemen.....	63
Memeriksa Elemen.....	67
Membandingkan Array.....	69
Mendapatkan Slice Array .....	70
Nilai Acak .....	71
Menggunakan Array Dalam Array .....	71
Fungsi tambahan untuk Array Number .....	75
Stack dan Queue.....	77
Stack .....	77
Queue .....	80

Bekerja dengan Hash .....	83
Membuat Hash .....	87
Bekerja dengan Key .....	88
Bekerja dengan Value.....	89
Iterasi Hash .....	90
Menambahkan atau Menghapus Pasangan .....	90
Memeriksa Pasangan.....	91
Membandingkan Hash.....	92
Nilai Acak .....	93
Hash dan Array .....	94
Konversi Hash ke Array.....	94
Set, Matriks, dan Vektor .....	95
Set.....	95
Membuat Set .....	95
Memeriksa Kesamaan Set .....	96
Memeriksa Bagian Dari Set Lain .....	96
Mendapatkan Irisan.....	96
Mendapatkan Union.....	97
Selisih Set.....	97
Menghitung Produk.....	97
Menghitung Power Set .....	98
Memeriksa Relasi.....	98
Memeriksa Relasi Refleksif .....	99
Memeriksa Relasi Simetrik .....	99

Memeriksa Relasi Anti-Simetrik .....	99
Memeriksa Relasi Transitif .....	101
Memeriksa Fungsi.....	101
Memeriksa Fungsi Injektif .....	102
Memeriksa Fungsi Surjektif .....	103
Memeriksa Fungsi Bijektif .....	104
Matriks dan Vektor .....	104
Membuat Rectangular Array .....	105
Mendapatkan Ukuran.....	107
Vektor .....	108
Matriks Persegi .....	110
Matriks Diagonal.....	111
Matriks Identitas.....	111
Matriks Konstanta .....	112
Matriks Segitiga Atas .....	113
Matriks Segitiga Bawah .....	113
Matriks Nol .....	114
Matriks Simetrik .....	114
Penjumlahan Matriks.....	115
Pengurangan Matriks .....	116
Perkalian Matriks.....	117
Perkalian Silang Vektor .....	118
Perkalian Skalar .....	120
Trace Matriks Persegi .....	120

Transpose Matriks .....	121
Determinan Matriks .....	121
Inverse Matriks .....	122
Instalasi Java dan Menjalankan Singkong.jar .....	123
Windows.....	123
macOS.....	126
Linux .....	131
Chrome OS.....	134
Tambahan: JRE dan JDK.....	137
Tambahan: distribusi Java alternatif .....	138
Daftar Pustaka .....	141

## Kata Pengantar

Buku ini diawali dengan contoh mengurutkan daftar bilangan (37 halaman) dan contoh algoritma greedy (8 halaman). Pembahasan dimulai dari memahami permasalahan, yang dilanjutkan secara bertahap ke algoritma, struktur data, dan pembuatan program dalam bahasa pemrograman Singkong.

Pembahasan dilakukan secara mendetil dengan tujuan agar buku ini juga dapat digunakan sebagai pengantar bagi yang tertarik memulai pemrograman komputer.

Sebagai topik lanjutan, disertakan juga contoh-contoh bagaimana bekerja dengan array, stack, queue, hash, set, matriks, dan vektor di Singkong.

Jakarta, April 2024

Tim penulis.

---

Untuk referensi, dokumentasi lengkap, dan contoh-contoh bahasa Singkong, Anda mungkin ingin membaca buku-buku gratis berikut:

- Mengetahui dan Menggunakan Bahasa Pemrograman Singkong (ISBN: 978-602-52770-1-6, Dr. Noprianto).
- Contoh dan Penjelasan Bahasa Singkong: Dasar-dasar Aplikasi GUI (ISBN: 978-602-52770-3-0, Dr. Noprianto, Dr. Karto Iskandar, Benfano Soewito, Ph.D.).
- Contoh dan Penjelasan Bahasa Singkong: Mahir Bekerja dengan GUI (ISBN: 978-602-52770-4-7, Dr. Noprianto, Dr. Wartika, Dr. Ford Lumban Gaol).
- Contoh dan Penjelasan Bahasa Singkong: Bekerja dengan Database Relasional (ISBN: 978-602-52770-5-4, Dr. Noprianto, Dr. Maria Seraphina Astriani, Dr. Fredy Purnomo).
- Contoh dan Penjelasan Bahasa Singkong: Aplikasi Web dan Topik Lanjutan (ISBN: 978-602-52770-6-1, Dr. Noprianto, Dr. Buyung Sofianto Munir, Dr. Sarwo).

Semua buku tersebut juga dapat dibaca di: <https://singkong.dev>

---



## Persiapan

Siapkanlah sebuah komputer, yang dilengkapi layar, keyboard, dan mouse/trackpad. Walaupun dengan tablet atau ponsel juga memungkinkan secara teknis, akan diperlukan pengaturan/instalasi tambahan yang tidak dibahas dalam buku ini.

Untuk perangkat keras komputernya, dapat menggunakan spesifikasi komputer mulai dari yang terbaru ataupun yang telah dijual sekitar 20 tahun yang lalu. Yang penting, dapat menjalankan salah satu dari daftar sistem operasi berikut.

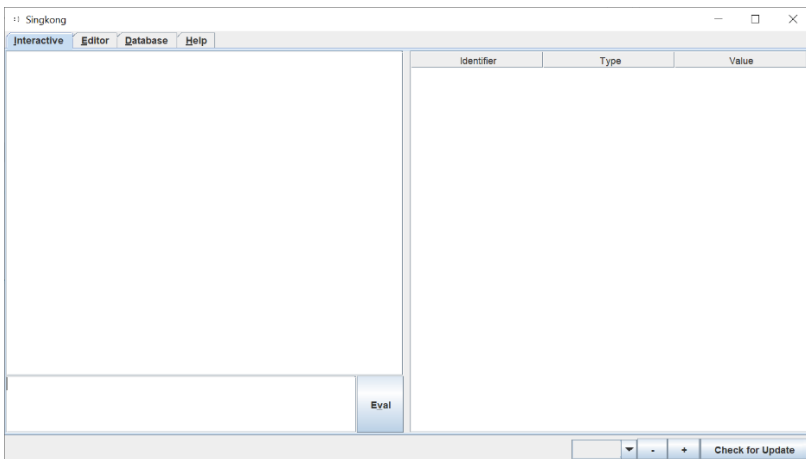
Interpreter Singkong (dan program yang Anda buat nantinya) dapat berjalan pada berbagai sistem operasi berikut:

- macOS (mulai dari Mac OS X 10.4 Tiger)
- Windows (mulai dari Windows 98)
- Linux (mulai yang dirilis sejak awal 2000-an; juga termasuk Raspberry Pi OS dan Debian di Android)
- Chrome OS (sejak tersedia Linux development environment, telah diuji pada versi 101 di chromebook)
- Solaris (telah diuji pada versi 11.4)
- FreeBSD (telah diuji pada versi 13.0 dan 12.1)
- OpenBSD (telah diuji pada versi 7.0 dan 6.6)
- NetBSD (telah diuji pada versi 9.2 dan 9.0)

Setelah komputer siap, lakukanlah instalasi Java, apabila belum terinstal sebelumnya. Secara teknis, Anda hanya membutuhkan Java Runtime Environment, versi 5.0 atau lebih baru. Versi 5.0 dirilis pada tahun 2004 (hampir 20 tahun lalu pada saat buku ini ditulis). Gunakan versi yang masih didukung secara teknis, apabila memungkinkan. *(Kenapa perlu menginstalasi Java? Karena interpreter Singkong ditulis dengan bahasa Java dan bahasa Singkong itu sendiri.)*

Sebagai langkah terakhir, downloadlah interpreter Singkong, yang akan selalu didistribusikan sebagai file jar tunggal (Singkong.jar). Downloadlah selalu dari <https://nopri.github.io/Singkong.jar>. Pada saat buku ini ditulis, ukurannya hanya 4,3 MB dan berisikan semua yang diperlukan untuk mengikuti semua pembahasan dalam buku ini. Pastikanlah Anda menggunakan versi terbaru.

Apabila Singkong.jar telah dapat dijalankan, maka persiapan sudah selesai. Perhatikanlah bahwa kita akan aktif pada tab Interactive untuk menguji kode program singkat secara langsung dan tab Editor untuk mengetikkan, menyimpan/membuka, dan menjalankan kode program yang lebih panjang (silahkan menggunakan editor pilihan Anda, apabila diperlukan).



Apabila langkah detil instalasi Java dan menjalankan Singkong.jar diperlukan, kita juga membahasnya di akhir buku, pada bab [Instalasi Java dan Menjalankan Singkong.jar](#).

## Mengurutkan Daftar Bilangan

Mari kita urutkan daftar bilangan berikut, dari kecil ke besar:

4 2 1 5 3

1. (**4** 2 1 5 3) Pertama, kita bandingkan antara 4 dan 2. Karena 4 lebih besar dari 2, kita tukar (posisinya) menjadi 2 dan 4. Kini, daftar kita berubah menjadi 2 4 1 5 3.
2. (2 **4** 1 5 3) Giliran 4 dan 1. Karena 4 lebih besar dari 1, kita tukar menjadi 1 dan 4. Daftar kita menjadi 2 1 4 5 3.
3. (2 1 **4** 5 3) Berikutnya, 4 dan 5 sesuai urutan yang kita harapkan. Sehingga, daftar kita tidak berubah.
4. (2 1 4 **5** 3) Terakhir, antara 5 dan 3. Karena 5 lebih besar dari 3, maka kita tukar. Daftar kita menjadi 2 1 4 3 5.

2 1 4 3 5

Kita sudah membandingkan semua pasangan angka dalam daftar, dengan melakukan 3 kali pertukaran. Apakah sudah terurut dari kecil ke besar? Rupanya belum. Maka, kita ulangi lagi langkah yang sama seperti sebelumnya.

1. (**2** 1 4 3 5) Untuk 2 dan 1, kita perlu tukar menjadi 1 dan 2. Daftar kita berubah menjadi 1 2 4 3 5.
2. (1 **2** 4 3 5) Untuk 2 dan 4, sesuai urutan dan oleh karenanya, daftar kita tidak berubah.

3. (1 2 **4 3** 5) Giliran 4 dan 3, dan karena 4 lebih besar dari 3, kita tukar, sehingga daftar kita berubah menjadi 1 2 3 4 5.
4. (1 2 3 **4 5**) Untuk 4 dan 5, urutannya sesuai, sehingga daftar kita tetap.

1 2 3 4 5

Kita kembali selesai membandingkan semua pasangan angka dalam daftar, dengan melakukan 2 kali pertukaran. Apakah sudah terurut? Sudah, bagi kita. Tapi, seperti sebelumnya, kita perlu cek:

1. (1 **2 3** 4 5) Karena 1 dan 2 sesuai urutan, kita tidak tukar. Daftar kita tetap.
2. (1 **2 3** 4 5) Karena 2 dan 3 sesuai urutan, kita tidak tukar. Daftar kita tetap.
3. (1 2 **3 4** 5) Karena 3 dan 4 sesuai urutan, kita tidak tukar. Daftar kita tetap.
4. (1 2 3 **4 5**) Karena 4 dan 5 sesuai urutan, kita tidak tukar. Daftar kita tetap.

1 2 3 4 5

Tidak ada pertukaran yang dilakukan. Daftar kita sudah terurut dari kecil ke besar.

Sampai tahap ini, kita sama sekali tidak melakukan pemrograman komputer. Kita bisa tuliskan langkah-langkahnya di kertas apabila mau.

Untuk daftar bilangan yang panjang pun, kita bisa melakukannya. Kenapa? Karena kita sudah menentukan langkah-langkah yang jelas untuk mencapai apa yang kita inginkan. Kita sebut ini sebagai algoritma.

**Catatan:** Nama algoritma pengurutan yang kita lakukan sebelumnya adalah Bubble sort, yang ditemukan pada tahun 1950-an.

Algoritma ini adalah contoh yang baik dalam mempelajari algoritma. Walau demikian, di era komputasi modern seperti saat ini misalnya, tersedia berbagai algoritma pengurutan lain yang lebih efisien.

Sekarang, mari kita cermati apa yang kita sudah lakukan sebelumnya, yaitu:

- A Kita terus mengulang selama daftar bilangan belum terurut dari kecil ke besar (masih terdapat yang harus ditukar).
- B Di dalam setiap perulangan tersebut, memeriksa pasangan dalam daftar. Ini tentunya juga mengulang, hanya jumlah perulangannya pasti, tergantung panjangnya daftar.
- C Ketika memeriksa, kita membandingkan apakah suatu bilangan lebih besar dari bilangan lain.

Terdapat langkah-langkah yang jelas, dimana kita melakukan perulangan dan mengambil keputusan (membandingkan dan menukar posisi bilangan apabila kondisi tertentu tercapai).

Sehari-hari, mengulang dan mengambil keputusan adalah hal yang umum dilakukan. Sebagai contoh, ketika berbelanja, kita mungkin mengunjungi kios-kios tertentu sampai kondisi tertentu tercapai. Misal ketika apa yang kita butuhkan sudah selesai dibeli. Untuk setiap barang yang akan dibeli, kita mungkin mengecek apakah harganya cocok, sebelum membeli.

Pada contoh mengurutkan bilangan sebelumnya, kita memiliki sebuah daftar bilangan. Penekanannya adalah pada sebuah daftar, dimana kita bisa memeriksa satu per satu item dalam daftar tersebut. Setidaknya, kita bisa merujuk pada item pertama, kedua, ketiga, dan seterusnya. Kita tahu dari mana kita harus memulai, dan kapan harus selesai.

Lebih lanjut, kembali ke daftar bilangan kita, kita mungkin perlu memastikan bahwa semua item dalam daftar tersebut hanyalah bilangan yang dapat dibandingkan.

Secara sederhana, daftar ini adalah contoh struktur data. Yaitu bagaimana data disusun atau distrukturkan, baik di memori (seperti daftar kita) ataupun di disk.

Sehari-hari, kita mungkin memiliki daftar belanja, yang hanya berisi barang apa yang perlu kita beli. Mungkin disertai dengan prioritas mana yang penting, mana yang bisa menyusul. Mungkin disertai juga dengan rentang harga yang cocok. Dengan demikian, kita dapat, diantaranya, dengan mudah beralih dari barang satu ke barang lain, dan menandai mana yang sudah dibeli.

Andai kita menulis setiap barang yang ingin dibeli pada selembar kecil kertas, dan menempatkan kertas-kertas tersebut secara tersebar, kita mungkin kerepotan ketika akan berbelanja.

Bisa kita lihat, cara kita menstrukturkan memiliki peran yang penting. Dengan kata lain, penting untuk menggunakan struktur yang tepat.

Lebih lanjut, kita dapat memanipulasi data yang tersimpan dalam daftar kita sebelumnya dengan algoritma, seperti halnya mengurutkan.

---

Mari kita kembali lagi ke contoh mengurutkan bilangan. Kita sudah memahami algoritmanya. Kita sudah tahu bahwa bilangan-bilangan dapat ditempatkan dalam daftar. Apabila kita perlu meminta bantuan kepada teman untuk mengurutkan daftar bilangan lain untuk kita, caranya dapat dibicarakan.

Tapi, lain ceritanya kalau kita perlu berbicara kepada komputer untuk mengurutkan daftar bilangan tertentu untuk kita.

Dengan apa kita berbicara kepada seorang teman? Tentunya dengan bahasa sehari-hari, yang saling dipahami satu sama lain.

Dengan apa kita berbicara dengan komputer? Dengan bahasa pemrograman. Akhirnya, kita sampai pada bagian pemrograman dari buku ini.

**Catatan:** Setidaknya, selama 80 tahunan terakhir (saat buku ini ditulis), terdapat ratusan bahasa pemrograman yang pernah dikembangkan. Dalam setiap dekade, puluhan bahasa pemrograman baru bisa dibuat. Sebagai contoh:

- Sebelum 1950: lebih dari 10 bahasa
- Pada 1950-an (50-an bahasa): FORTRAN, COBOL, dan LISP.
- Pada 1960-an (50-an bahasa): BASIC.
- Pada 1970-an (60-an bahasa): Pascal, C, SQL.
- Pada 1980-an (60-an bahasa): C++, Perl.
- Pada 1990-an (70-an bahasa): Python, Java, PHP, JavaScript.
- Pada 2000-an (50-an bahasa): C#, Go.
- Pada 2010-an (30-an bahasa): Dart, Swift.

Belakangan ini, bahasa baru yang dibuat tidaklah sebanyak misal 40 atau 50 tahun lalu, tapi, tetap saja ada. Misalkan bahasa pemrograman Singkong, yang kita gunakan dalam buku ini, yang dirilis pada akhir tahun 2019.

Sebelum kita mulai menulis kode, mari kita deskripsikan langkah-langkah dalam algoritma sebelumnya dengan penjelasan yang dapat kita pahami, dengan bantuan kata kunci yang umum ditemukan dalam bahasa pemrograman. Seperti menulis kode program, tapi tidak spesifik dengan bahasa tertentu, dan lebih ke arah yang dapat dipahami oleh manusia. Kita sebut ini pseudocode.

Apa gunanya? Diantaranya adalah untuk memudahkan kita memahami algoritma, tanpa terpaku pada bahasa pemrograman tertentu.

Mari kita contohkan.

procedure bubblesort (array: list of number)

    n <- length of array

    repeat

        swapped <- false

        i <- 1

        repeat

            if array[i - 1] > array [i]

                swap array[i - 1] and array [i]

                swapped <- true

        end if

        i <- i + 1

    until i = n

    until swapped = false

end procedure

Dengan hanya memahami teks dalam pseudocode tersebut, dan tidak terkait dengan sintaks bahasa pemrograman apapun, bisa kita lihat bahwa ini sama saja dengan ketika kita melakukan langkah-langkah dalam mengurutkan daftar bilangan:



- [Diwarnai merah] Kita mengulang sampai tidak ada yang perlu ditukar.
- [Diwarnai biru] Kita akan memeriksa setiap pasangan dalam daftar. Perhatikanlah bahwa kita menggunakan penanda indeks tertentu dalam daftar. Dalam hal ini, apabila kita memiliki daftar 4 2 1 5 3, penanda indeks adalah sebagai berikut (diwakili dengan i pada pseudocode tersebut):
  - o 4: i adalah 0 (indeks dimulai dari 0)
  - o 2: i adalah 1
  - o 1: i adalah 2
  - o 5: i adalah 3
  - o 3: i adalah 4
- [Diwarnai biru] Dengan demikian, untuk daftar 4 2 1 5 3, kita akan memeriksa dari indeks 1 (bilangan dengan nilai 2).
  - o Setiap kali perulangan, kita menambahkan 1 ke indeks.
  - o Setelah indeks sama dengan jumlah bilangan dalam daftar, perulangan selesai.
  - o Terakhir, kita memeriksa sampai indeks 4 (bilangan dengan nilai 3).
  - o Oleh karena itu, apabila n adalah jumlah bilangan dalam daftar, kita mengulang dari indeks 1 sampai n - 1 (apabila n adalah 5, maka indeks 1 sampai 4).
- [Diwarnai hijau] apabila indeks sebelumnya lebih besar daripada indeks saat ini, maka kita tukar. Misal ketika kita membandingkan 4 dan 2 dari **4 2 1 5 3**:
  - o Ketika i bernilai 1, maka indeks adalah 2.
  - o Maka, indeks sebelumnya adalah 0, dan bilangannya adalah 4.
  - o Apabila 4 lebih besar dari 2 (yang mana adalah benar), maka kita tukar menjadi 2 dan 4.

- [Diwarnai oranye] Ketika terjadi pertukaran, kita tandai bahwa swapped adalah true. Ingatlah kita berhenti mengulang hanya ketika swapped adalah false.

**Catatan:** Tidak terdapat standar yang universal dalam pseudocode. Kita perlu menuliskannya dengan jelas dan komplit, sehingga dapat dipahami.

Dalam contoh tersebut, `n <- length of array` artinya kita menyimpan nilai panjang daftar (berapa banyak bilangan) ke `n`. Untuk mengulang, di contoh ini kita menggunakan `repeat` dan `until`. Untuk menukar bilangan-bilangan, kita dapat menuliskan sebagai `swap` saja (yang idealnya dipahami oleh pembuat program).

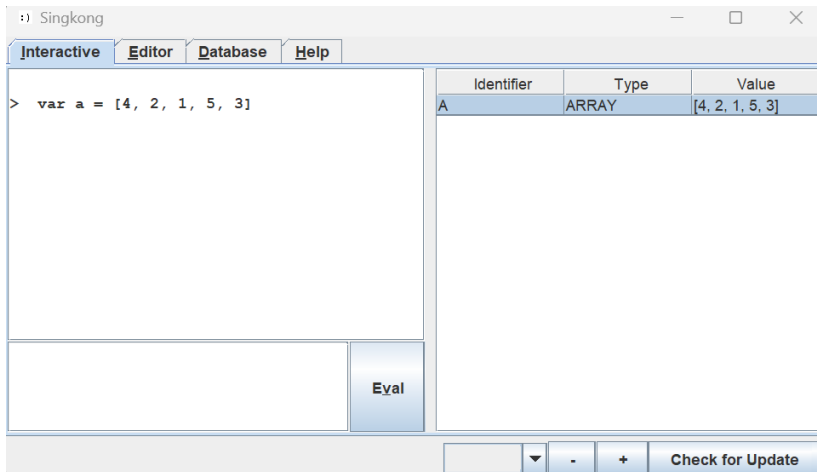
Sekarang, bagaimana kalau kita terjemahkan pseudocode sebelumnya menjadi kode bahasa pemrograman?

Karena buku ini adalah tentang bahasa pemrograman Singkong, maka kode program dituliskan dalam bahasa Singkong. Tanpa diperlukan pemahaman sebelumnya akan sintaks bahasa Singkong, mari kita bahas dari awal sampai algoritma tersebut berhasil diimplementasikan.

Jalankan `Singkong.jar` dan aktiflah terlebih dahulu pada tab `Interactive`. Pada tab ini, kita bisa ketikkan kode Singkong, menjalankan, dan menampilkan hasilnya. Langsung pada daftar bilangan kita, ketikkanlah (pada bagian kiri bawah):

```
var a = [4, 2, 1, 5, 3]
```

Kemudian, kliklah tombol `Eval`. Hasilnya akan tampak pada gambar di halaman berikut:



Kita akan sekaligus mengenal user interface interactive evaluator ini:

- Pada bagian kiri atas, apa yang kita ketikkan dan hasil evaluasinya akan ditampilkan.
- Pada bagian kiri bawah, kita mengetik kode untuk dievaluasi. Apabila tidak terdapat kesalahan, kode akan dievaluasi dan ditampilkan, termasuk hasil evaluasinya.
- Tabel di sebelah kanan memuat identifier, tipe, dan nilainya.

**Catatan:** Singkong adalah bahasa pemrograman dengan sistem tipe **dynamically typed**, dimana tipe data ditentukan secara dinamis pada saat program berjalan. Di contoh sebelumnya, `a` adalah sebuah ARRAY, walau kita tidak mendeklarasikan tipenya sebelum program berjalan. Kita hanya memberikan nilai berupa ARRAY.

Selama program berjalan, `a` bisa diubah menjadi tipe lain apabila kita berikan nilai dengan tipe lain tersebut. Misal: `var a = 12345` (dimana `a` menjadi sebuah NUMBER).

Kebalikan **dynamically typed** adalah **statically typed** (misal C dan Java), dimana pemeriksaan tipe dilakukan sebelum program berjalan.

**Catatan:** Kita dapat menggunakan interactive evaluator ini sebagai kalkulator. Cobalah misal 12345 pangkat 123 (operator pangkat di Singkong adalah ^), yang akan menghasilkan keluaran berikut (yang merupakan bilangan yang sangat besar):

```
> 12345 ^ 123
```

```
179227478536797075276952162319434197129926964430623405
351403914666844095303193142386105303128935260661331482
166609669142646381589155256961299625923906846736377224
598990446854741893321648522851663303862851165879753724
272728386042804116173040017014488023693807547724950916
588058455499429272048326934098750367364004488112819439
755556403443027523561951313385041616743787240003466700
321402142800004483416756392021359457461719905854364181
525061772982959380338841234880410679952689179117442108
690738677978515625
```

Untuk memberikan nilai ke suatu variabel, atau identifier, kita menggunakan statement var. Penamaan variabel memiliki aturan sederhana berikut:

1. Dimulai dengan huruf atau underscore dan dapat diikuti oleh huruf, underscore, atau digit.
2. Nama variabel harus belum digunakan sebagai nama fungsi bawaan.

Dalam hal ini, a adalah nama yang valid.

Di Singkong, sebuah daftar adalah sebuah ARRAY. Oleh karena itu, pada tabel identifier, tipenya adalah ARRAY (nantinya, kita akan

melihat tipe-tipe lain). Di Singkong, panjang ARRAY tidak dibatasi secara eksplisit dan dapat mengandung nilai berbagai tipe, termasuk ARRAY. Penanda indeks ARRAY dimulai dari 0, sampai panjang ARRAY dikurangi 1.

Untuk mendapatkan panjang ARRAY, kita menggunakan fungsi `len`. Contoh hasil evaluasi (dalam hal ini, `>` menandakan prompt ketika kode diketikkan, tapi karakter `>` tidak perlu diketikkan):

```
> var n = len(a)
> n
5

> n - 1
4
```

**Catatan:** Singkong tidak membedakan huruf besar dan huruf kecil (case-insensitive). Dalam hal ini, `a` sama saja dengan `A`, dan `len` sama saja dengan `LEN`, atau kombinasi huruf besar/kecil lainnya.

Untuk menggunakan fungsi, kita menuliskan nama fungsinya, diikuti oleh `(`, kemudian daftar argumen dipisahkan koma apabila ada, dan ditutup dengan `)`. Merujuk pada pemanggilan `len` sebelumnya, `len` adalah nama fungsi. Kita ingin mendapatkan panjang `a`, maka `a` adalah argumen bagi fungsi `len`.

Setiap fungsi memiliki jumlah argumen yang dapat diterima. Fungsi `len`, dalam hal ini, hanya menerima satu argumen. Apabila diberikan lebih, maka pesan kesalahan akan ditampilkan:

```
> len()
ERROR: [line: 1] wrong number of arguments, got=0, want=1

> len(a, a)
ERROR: [line: 1] wrong number of arguments, got=2, want=1
```

Fungsi, ketika menerima argumen, juga menentukan tipe apa yang diterima. Sehari-hari, kita mengetahui bahwa nama kita umumnya adalah deretan huruf (STRING), tanggal lahir adalah sebuah tanggal (DATE), dan umur adalah sebuah bilangan (NUMBER). Fungsi `len`, dalam hal ini menerima satu argumen dengan tipe STRING, HASH, atau ARRAY. Dari mana kita tahu aturan fungsi `len`? Kita bisa ketikkan `len` dan klik tombol Eval (atau tekan enter):

```
> len
built-in function: len: returns the length of
STRING, HASH, or ARRAY
arguments: 1: STRING, HASH, or ARRAY
return value: NUMBER
```

Untuk mendapatkan tipe sebuah identifier, kita bisa menggunakan fungsi `type`. Seperti contoh berikut:

```
> type(len)
"BUILTIN"

> type(a)
"ARRAY"

> type(n)
"NUMBER"
```

Sampai di sini, kita tahu bahwa `a` adalah sebuah ARRAY dan `n` (panjang ARRAY `a`) adalah sebuah NUMBER.

Kita siap melanjutkan ke pembahasan berikutnya.

---

Bagaimana kalau kita melakukan perulangan, misal untuk menampilkan 1 sampai 5? Kita bisa menggunakan beberapa cara di Singkong. Tapi, karena algoritma kita membutuhkan kita untuk mengulang terus menerus sampai kondisi tertentu tercapai, mari kita pelajari ini terlebih dahulu.

Di Singkong, kita bisa menggunakan repeat untuk mengulang terus menerus. Yang penting, kita harus memiliki kondisi untuk berhenti dari perulangan. Apabila tidak, maka program akan terus berjalan (dimana, untuk menghentikannya, kita meminta sistem operasi untuk melakukan terminasi).

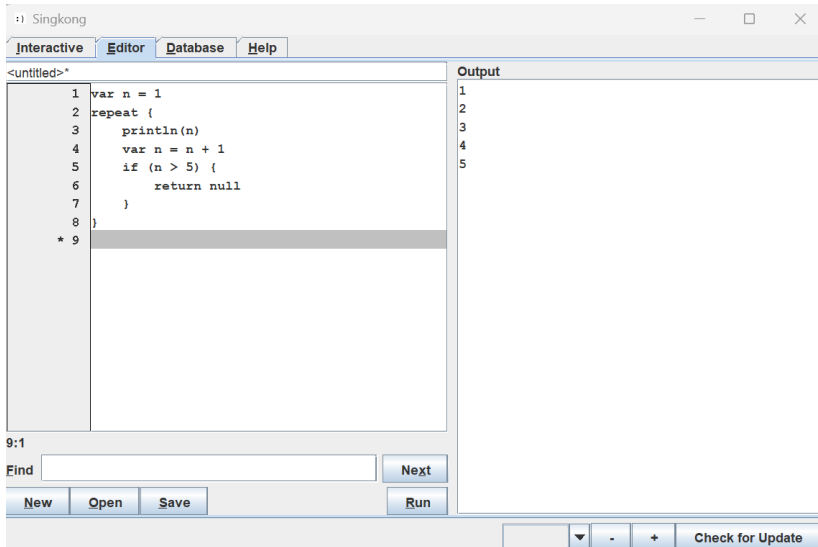
Aktiflah pada tab Editor, ketikkan kode berikut, dan kliklah tombol Run.

```
var n = 1
repeat {
    println(n)
    var n = n + 1
    if (n > 5) {
        return null
    }
}
```

Hasilnya adalah sebagai berikut (setelah message box tampil dan kita klik OK):

```
1
2
3
4
5
```

Message box dan tampilnya bilangan 1 sampai 5 adalah efek dari pemanggilan fungsi println, yang akan menampilkan, diikuti turun ke baris berikut.



Lihatlah. Kita sudah bisa mengulang dan membandingkan.

Untuk keluar dari perulangan `repeat`, kita menggunakan `return`. Dalam hal ini, kita `return null`, yang bertipe `NULL`, dan umumnya dihasilkan ketika terdapat kesalahan atau ketika kita tidak peduli dengan nilai yang dikembalikan. Sintaks `repeat` adalah:

```
repeat { statements }
```

Di Singkong, blok diawali dengan `{` dan diakhiri dengan `}`. Semua statement di dalam blok akan dikerjakan.

Untuk membandingkan, kita menggunakan `if`. Di Singkong, `if` digunakan untuk seleksi/kondisi/percabangan, dengan sintaks berikut:

```
if      (condition)      {consequences}      else
{alternatives}
```



Dalam hal ini, condition kita adalah  $n > 5$ . Apabila terpenuhi, blok consequences dikerjakan. Dalam hal ini, kita hanya keluar dari perulangan.

Dalam perulangan, kita menampilkan nilai  $n$ . Perhatikanlah kita kemudian tentukan nilai baru untuk  $n$ , sesuai nilai  $n$  ditambah 1 (var  $n = n + 1$ ). Apabila kita tidak lakukan, maka  $n$  akan tetap sama, dan kita tidak dapat keluar dari perulangan karena kondisi  $n > 5$  tidak pernah tercapai.

Sampai di sini, kita telah dapat melakukan perulangan terus menerus, dan melakukan perbandingan/seleksi tertentu.

**Catatan:** Kode sebelumnya dapat dituliskan juga sebagai berikut:

```
var n = 1 repeat { println(n) var n = n + 1 if (n > 5) { return null } }
```

Tapi, cukup repot untuk membacanya bukan? Bisa kita lihat, kita menuliskan dalam baris tersendiri atau indentasi tersendiri untuk memudahkan kita membaca kode program tersebut.

**Catatan:** Kita tidak dapat keluar dari Singkong.jar apabila editor masih berisi konten yang masih belum disimpan (pesan "Editor is open" akan ditampilkan). Untuk menyimpan kode yang diketik, kliklah tombol Save (dan simpanlah dengan nama file .singkong, misal hello.singkong). Untuk keluar tanpa menyimpan, kliklah New dan jawablah No ketika konfirmasi simpan file ditampilkan. Setelahnya, kita bisa keluar dari Singkong.jar.

Sekarang, bagaimana kalau kita melakukan perulangan, dengan fokus untuk bekerja dengan indeks dalam daftar? Kita akan menampilkan dari indeks 1 (bukan 0) sampai indeks panjang dikurangi 1, seperti bagian dari algoritma kita.

Ketikkanlah kode berikut pada tab Editor. Kode sebelumnya dapat disimpan apabila diinginkan.

```
var a = [4, 2, 1, 5, 3]
var n = len(a)

var i = 1
repeat {
    println(a[i])
    var i = i + 1
    if (i == n) {
        return null
    }
}
```

Pada daftar kita, yang tampil adalah seperti keluaran dari program kita (setelah message box tampil):

```
2
1
5
3
```

Walau kodenya kurang lebih sama dengan contoh sebelumnya, kita sudah melangkah lebih maju. Perhatikanlah beberapa hal berikut:

- [**Diwarnai merah**] Kita bekerja dengan daftar bilangan kita, sebagai variabel a. Untuk panjang kita, kita dapatkan dengan fungsi len, sebagai variabel n.
- [**Diwarnai biru**] Kita memulai dari indeks 1, bukan 0. Indeks disimpan sebagai variabel i.
- [**Diwarnai hijau**] Untuk mendapatkan bilangan pada indeks tertentu, kita menggunakan operator indeks [<indeks>]. Dengan demikian:

- a[0]: 4
  - a[1]: 2
  - a[2]: 1
  - a[3]: 5
  - a[4]: 3
- [Diwarnai orange] Pastikanlah kita menambahkan nilai indeks karena kita akan keluar dari perulangan setelah indeks sama dengan panjang daftar ( $i == n$ ). Di Singkong, untuk membandingkan operan yang kompatibel (misal NUMBER dan NUMBER), kita menggunakan operator `==`.
  - Perhatikanlah bahwa di kode tersebut, kita tidak pernah memproses `a[n]` karena begitu  $i == n$ , kita sudah keluar dari perulangan. Selalu `a[1]` sampai maksimal `a[n-1]`.

**Catatan:** Ketika indeks pada ARRAY diberikan nilai di luar rentang, maka kita mendapatkan null (tipe: NULL). Dalam ARRAY dengan 5 elemen, indeks yang valid berada pada rentang 0 sampai 4. Bagaimana kalau kita berikan indeks misal -5 atau 5? Hasilnya adalah null seperti contoh berikut:

```
> type(a[-5])
"NULL"

> type(a[5])
"NULL"
```

Bagaimanakah cara kita menukar bilangan pada indeks satu dengan indeks lain pada sebuah daftar? Mari kita contohkan dengan kode, yang diketikkan pada tab Interactive.

Kita sudah tahu bagaimana mendapatkan elemen tertentu dalam ARRAY berdasarkan indeks. Perhatikanlah contoh berikut:

```
> var a = [4, 2, 1, 5, 3]
```

```
> a[1]  
2
```

```
> a[0]  
4
```

Untuk menentukan nilai elemen tertentu berdasarkan indeks, kita perlu menggunakan fungsi `set`. Untuk ARRAY, argumen untuk fungsi `set` adalah:

- ARRAY itu sendiri
- Indeks
- Nilai elemen pada indeks

Misal:

```
> set(a, 0, 40)  
[40, 2, 1, 5, 3]
```

Kita kembalikan ke nilai awal:

```
> set(a, 0, 4)  
[4, 2, 1, 5, 3]
```

Untuk menukar indeks 0 dan indeks 1, kita perlu gunakan variabel bantu. Misal dengan nama `temp` (dari temporary).

```
> var temp = a[0]  
> temp  
4
```

Setelahnya, kita menempatkan nilai indeks 1 ke indeks 0:

```
> set(a, 0, a[1])  
[2, 2, 1, 5, 3]
```

Dan, akhirnya, menempatkan nilai variabel bantu `temp` ke indeks 1:

```
> set(a, 1, temp)
[2, 4, 1, 5, 3]
```

Pertukaran pun selesai dilakukan.

---

Mari kita gabungkan semua yang kita bahas sebelumnya. Kita bisa menyimpannya sebagai `bubblesort.singkong`.

```
var a = [4, 2, 1, 5, 3]
println(a)

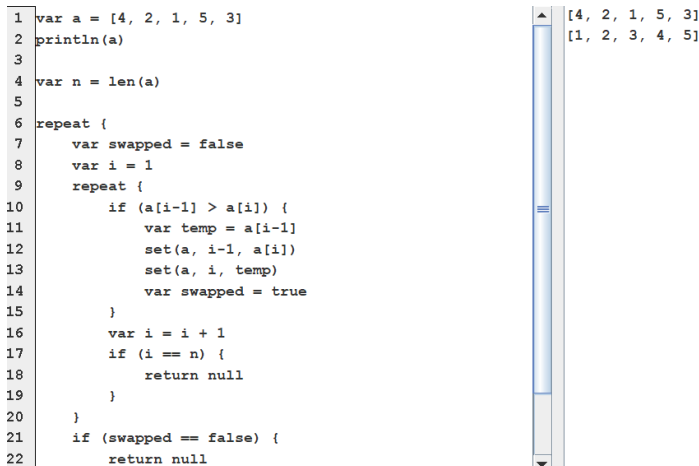
var n = len(a)

repeat {
    var swapped = false
    var i = 1
    repeat {
        if (a[i-1] > a[i]) {
            var temp = a[i-1]
            set(a, i-1, a[i])
            set(a, i, temp)
            var swapped = true
        }
        var i = i + 1
        if (i == n) {
            return null
        }
    }
    if (swapped == false) {
        return null
    }
}

println(a)
```

Perhatikanlah bahwa yang baru hanyalah ketika kita membuat variabel swapped dan mengulang sampai swapped bernilai false. Selebihnya, kita sudah bahas dalam contoh sebelumnya. Bedanya hanyalah, kita melakukan repeat dalam repeat.

Apabila program tersebut dijalankan, daftar sebelum diurutkan akan ditampilkan terlebih dahulu. Setelah pengurutan, kita kembali tampilkan.



The screenshot shows a Singkong IDE with a code editor on the left and a console on the right. The code in the editor is a bubble sort algorithm in Singkong. The console shows the output of the program, which is the array [1, 2, 3, 4, 5] after sorting the initial array [4, 2, 1, 5, 3].

```
1 var a = [4, 2, 1, 5, 3]
2 println(a)
3
4 var n = len(a)
5
6 repeat {
7   var swapped = false
8   var i = 1
9   repeat {
10    if (a[i-1] > a[i]) {
11      var temp = a[i-1]
12      set(a, i-1, a[i])
13      set(a, i, temp)
14      var swapped = true
15    }
16    var i = i + 1
17    if (i == n) {
18      return null
19    }
20  }
21  if (swapped == false) {
22    return null
```

[4, 2, 1, 5, 3]  
[1, 2, 3, 4, 5]

Bandingkanlah algoritma, pseudocode, dan kode Singkong kita.

**Catatan:** Apa yang kita tuliskan (disimpan sebagai file .singkong atau tidak), baik di tab Interactive ataupun Editor, adalah source code dalam bahasa Singkong. Ketika kita klik Eval atau Run, kode tersebut diinterpretasi oleh interpreter bahasa pemrograman Singkong (Singkong.jar), menjadi sesuatu yang dimengerti oleh komputer.

Kebalikan dari interpretasi (dengan interpreter) adalah kompilasi (dengan compiler). Pada Singkong, implementasi yang tersedia hanyalah interpretasi.

**Catatan:** Kita menuliskan langkah demi langkah apa yang program kita harus lakukan.

Apabila program kita harus mengulang sampai variabel swapped bernilai false, maka kita tuliskan demikian (termasuk tentunya kapan swapped dapat bernilai true). Apabila di dalam perulangan kita harus membandingkan, maka kita lakukan. Apabila kita harus menukar bilangan satu dengan lainnya, kita melakukannya misal dengan menggunakan variabel bantu.

Kita memberikan perintah apa yang harus dilakukan oleh program kita. Langkah demi langkah.

Ketika pemrogram menentukan langkah demi langkah control flow eksplisit dalam mencapai tujuan, pemrogram tersebut bekerja dalam paradigma imperatif. Bahasa Singkong adalah bahasa pemrograman yang bekerja secara imperative (imperatif).

Bukanlah yang kita bahas barusan terasa wajar? Kita membuat program dan tentunya perlu menuliskan langkah detilnya bukan?

Perkembangan dalam bahasa pemrograman dan pemrograman secara umum menghadirkan berbagai paradigma pemrograman, yang secara sederhana, dapat diartikan sebagai pendekatan bagaimana program dibuat.

Ketika control flow tidak eksplisit dan pemrogram dapat fokus pada hasil yang diinginkan (bukan langkah mencapainya), paradigmanya adalah declarative (deklaratif).

Sebagai contoh, apabila bekerja dengan sistem database relasional, kita menggunakan SQL. Dengan SQL, kita menentukan apa yang kita inginkan (misal: mendapatkan isi tabel; cukup sebaris perintah). Kita tidak perlu membuka file, baca dengan aturan tertentu, ulang, dan seterusnya. SQL adalah bahasa dengan paradigma deklaratif.

Apabila ingin benar-benar sama dengan langkah manual kita di awal bab ini, mari kita kopikan `bubblesort.singkong` ke `bubblesort_detail.singkong`. Kemudian, buka dan editlah `bubblesort_detail.singkong`, dengan isi sebagai berikut:

```
var a = [4, 2, 1, 5, 3]
println(a)

var n = len(a)
var t = 1
repeat {
    println("Tahap: " + t)
    var swapped = false
    var counter = 0
    var i = 1
    repeat {
        print(a)
        if (a[i-1] > a[i]) {
            print(" tukar " + a[i-1] + " dan " + a[i])
            var temp = a[i-1]
            set(a, i-1, a[i])
            set(a, i, temp)
            var swapped = true
            var counter = counter + 1
        } else {
            print(" tidak tukar " + a[i-1] + " dan " + a[i])
        }
        println(" " + a)
        var i = i + 1
        if (i == n) {
            return null
        }
    }
    print("Jumlah tukar: " + counter)
    println(" Daftar: " + a)
    var t = t + 1
    if (swapped == false) {
        return null
    }
}

println(a)
```



Ketika kita jalankan programnya (dan menutup message box berkali-kali), keluarannya akan membuktikan bahwa program kita melakukan yang sama persis dengan langkah manual kita:

```
[4, 2, 1, 5, 3]
Tahap: 1
[4, 2, 1, 5, 3] tukar 4 dan 2 [2, 4, 1, 5, 3]
[2, 4, 1, 5, 3] tukar 4 dan 1 [2, 1, 4, 5, 3]
[2, 1, 4, 5, 3] tidak tukar 4 dan 5 [2, 1, 4, 5, 3]
[2, 1, 4, 5, 3] tukar 5 dan 3 [2, 1, 4, 3, 5]
Jumlah tukar: 3 Daftar: [2, 1, 4, 3, 5]
Tahap: 2
[2, 1, 4, 3, 5] tukar 2 dan 1 [1, 2, 4, 3, 5]
[1, 2, 4, 3, 5] tidak tukar 2 dan 4 [1, 2, 4, 3, 5]
[1, 2, 4, 3, 5] tukar 4 dan 3 [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5] tidak tukar 4 dan 5 [1, 2, 3, 4, 5]
Jumlah tukar: 2 Daftar: [1, 2, 3, 4, 5]
Tahap: 3
[1, 2, 3, 4, 5] tidak tukar 1 dan 2 [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5] tidak tukar 2 dan 3 [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5] tidak tukar 3 dan 4 [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5] tidak tukar 4 dan 5 [1, 2, 3, 4, 5]
Jumlah tukar: 0 Daftar: [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Kita menggunakan operator + untuk menambahkan STRING dengan NUMBER, yang mana akan menjadi STRING untuk ditampilkan. Agar tetap dalam baris yang sama (tidak berlaku untuk message box), kita gunakan print dan pada akhirnya println untuk ke baris berikutnya.

**Catatan:** Selain **dynamically typed**, Singkong juga menggunakan sistem tipe **weakly typed**, yang dalam hal ini lebih banyak melakukan konversi tipe secara implisit. Misal, STRING + NUMBER otomatis menjadi STRING.

Kebalikan dari weakly typed adalah strongly typed (misal pada Java (static, strong) dan Python (dynamic, strong)) dengan aturan tipe lebih ketat dan lebih sedikit konversi secara implisit.

Contoh bahasa pemrograman selain Singkong yang dynamically typed dan weakly typed adalah JavaScript.

Apakah pembahasan kita sudah selesai? Sayangnya, belum. Apabila diperhatikan, kita hanya fokus pada daftar yang kita miliki. Ketika kita memiliki daftar lain untuk diurutkan, sejumlah kode repeat dalam repeat tersebut perlu dikopikan ulang.

Ketika mengopikan ulang, kode menjadi panjang, padahal yang dilakukan sama saja. Lebih repot lagi, apabila kita perlu mengubah kode tertentu, kita perlu mengubahnya juga di hasil kopinya. Apabila lupa, dan perubahannya harus dilakukan, bug atau dalam hal ini kesalahan yang dibuat oleh programmer, dapat terjadi.

Contoh solusi untuk hal ini adalah dengan membuat fungsi, yang menerima argumen tertentu. Dalam hal ini, argumen yang dimaksud adalah sebuah ARRAY. Ketika kita perlu mengurutkan bilangan lain, kita cukup panggil fungsi tersebut. Ketika ada kode dalam fungsi yang perlu diubah, kita cukup mengubahnya sekali. Bug mungkin tetap dapat terjadi, tapi kita cukup memperbaiki sekali, walaupun fungsi dipanggil berkali-kali.

Fungsi di Singkong memiliki aturan nama yang sama dengan identifier lain. Mari kita berikan nama bubblesort.

Aktiflah pada tab Editor dan kliklah tombol New untuk membuat file baru. Ketikkanlah kode berikut:

```
var bubblesort = fn(a) {  
    return a  
}  
  
println(bubblesort([4, 2, 1, 5, 3]))
```

Fungsi bubblesort kita memang tidak melakukan apa-apa. Buktinya, daftar kita tetap tidak terurut. Tapi, kita bisa melihat beberapa hal berikut:

- [Diwarnai merah] Fungsi kita membutuhkan sebuah argumen, yang nantinya kita akses sebagai a.
- Perhatikanlah bahwa statement dalam fungsi adalah sebuah blok tersendiri, diawali { dan diakhiri }.
- [Diwarnai biru] Fungsi kita harusnya mengembalikan daftar yang sudah terurut. Saat ini, kita mengembalikan apa yang dilewatkan, tanpa melakukan apa-apa. Tapi ini tetap fungsi yang valid.

Untuk melakukan bubblesort, bukalah kembali file bubblesort.singkong (dengan tombol Open pada Editor), dan sesuaikanlah isinya menjadi berikut:

```
var bubblesort = fn(a) {
  var n = len(a)
  repeat {
    var swapped = false
    var i = 1
    repeat {
      if (a[i-1] > a[i]) {
        var temp = a[i-1]
        set(a, i-1, a[i])
        set(a, i, temp)
        var swapped = true
      }
      var i = i + 1
      if (i == n) {
        return null
      }
    }
    if (swapped == false) {
      return null
    }
  }
  return a
}
```

```
println(bubblesort([4, 2, 1, 5, 3]))  
println(bubblesort([5, 4, 3, 2, 1]))
```

Kita hanya membuat fungsi (dengan argumen dan return) dan merapikan kode yang dibuat sebelumnya, menjadi isi fungsi. Tekanlah tab di setiap baris untuk membuat indentasi supaya kode program lebih nyaman terbaca. Sebagaimana disebutkan sebelumnya, indentasi ini tidaklah diwajibkan.

Di luar isi fungsi, kita memanggil fungsi untuk ARRAY-ARRAY yang berbeda. Ketika dijalankan, hasilnya adalah seperti berikut:

```
[1, 2, 3, 4, 5]
```

```
[1, 2, 3, 4, 5]
```

Sesuai apa yang inginkan bukan? Simpanlah file bubblesort.singkong tersebut. Pembahasan kita sudah hampir selesai.

**Catatan:** Kita telah membuat sebuah function (fungsi, tipe data: FUNCTION). Pada program yang lebih besar, kita dapat membuat berbagai function lain. Sebagai contoh, function untuk mendapatkan input user, melakukan validasi, dan membuat daftar dari input tersebut. Function ini kemudian bisa memanggil function bubblesort untuk mengurutkan.

Sebuah function idealnya melakukan atau menyediakan fungsionalitas yang spesifik. Dalam istilah lain, kita juga mengenalnya sebagai procedure (prosedur).

Ketika kita membuat program yang terdiri dari serangkaian prosedur yang dapat memanggil atau dipanggil prosedur-prosedur lainnya, kita bekerja dengan paradigma procedural (prosedural). Bisa kita lihat, paradigma ini juga adalah imperatif karena kita menentukan langkah-langkah control flow eksplisit.

Singkong adalah bahasa pemrograman prosedural. Contoh bahasa pemrograman prosedural lain adalah C dan Python.

Contoh paradigma populer lain adalah object-oriented (berorientasi objek). Pada pemrograman berorientasi objek, sebuah program dimodelkan dalam berbagai objek, yang mana dapat saling berinteraksi. Masing-masing objek dapat memiliki field (data) dan method (kode, prosedur). Pemrogram umumnya memodelkan sesuai dengan dunia nyata. Sebagai contoh, pada program penjualan produk, kita memiliki objek Produk dan Pelanggan. Contoh bahasa pemrograman berorientasi objek adalah Java dan Python. Singkong **tidak** berorientasi objek.

Sebuah bahasa pemrograman dapat mendukung beberapa paradigma. Sebagai contoh, dengan Python misalnya, kita dapat membangun program secara prosedural ataupun berorientasi objek.

Sekarang, bagaimana kalau kita tidak berikan ARRAY bilangan, melainkan ["singkong", "programming", "language"]? Bukalah kembali bubblesort.singkong dan tambahkan baris berikut di akhir file:

```
println(bubblesort(["singkong", "programming",  
"language"]))
```

Kemudian, jalankan kembali file tersebut. Sejumlah pesan kesalahan akan ditampilkan:

```
[1, 2, 3, 4, 5]  
[1, 2, 3, 4, 5]  
ERROR: [line: 7] unknown operator: STRING >  
STRING  
ERROR: [line: 7] unknown operator: STRING >  
STRING
```

```
ERROR: [line: 7] unknown operator: STRING >
STRING
ERROR: [line: 7] unknown operator: STRING >
STRING
```

Untuk dua baris pertama, kita berhasil mengurutkan karena memang berupa ARRAY dari NUMBER. Tapi, untuk baris terakhir, kita melewati ARRAY dari STRING. Algoritma kita hanya bisa bekerja untuk membandingkan NUMBER. Ketika ingin membandingkan "singkong" dan "programming", dengan operator >, kesalahan terjadi, yang mana adalah wajar.

Jadikanlah baris yang baru ditambahkan menjadi komentar, sehingga tidak diproses, walau tetap ada dalam kode:

```
#
println(bubblesort(["singkong", "programming", "language"]))
;
```

**Catatan:** Komentar diawali dengan # dan diakhiri ; serta dapat terdiri dari sejumlah baris. Pastikanlah menutup komentar dengan ; (terkadang ini terlupakan).

Ketika kita jalankan kembali, tentu saja tidak ada pesan kesalahan yang ditampilkan. Tambahkan baris berikut di akhir file:

```
println(bubblesort(42153))
```

Jalankan kembali, dan kita kembali akan menjumpai pesan kesalahan:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
ERROR: [line: 2] argument to "len" not
supported, got NUMBER
```

```
ERROR: [line: 2] argument to "len" not
supported, got NUMBER
```

Bisa kita lihat, fungsi `len` tidak dapat bekerja mendapatkan panjang sebuah `NUMBER`, karena memang tidak ada panjang yang bisa didapatkan.

Dengan demikian, kita perlu memastikan bahwa hanya `ARRAY` yang dilewatkan, dan harus berupa `ARRAY` dari `NUMBER`. Apabila bukan, kita kembalikan `null` (sebagai penanda terjadi kesalahan). Untuk kebutuhan tersebut, kita dapat menggunakan fungsi

`is_array_and_of`:

```
> is_array_and_of
```

built-in function: `is_array_and_of`: returns whether `is` `ARRAY` and each of its elements (expression, identifier, literal) is of specific type (represented as `STRING`, case-insensitive comparison).

arguments: 2: <any> and `STRING`

return value: `BOOLEAN`

Sisipkan kode berikut sebagai baris pertama fungsi `bubblesort` kita, sebelum `var n = len(a)`:

```
if (!is_array_and_of(a, "NUMBER")) {
    return null
}
```

Kini, apabila dipanggil dengan argumen yang valid ataupun tidak, fungsi kita bekerja dengan baik:

```
println(bubblesort([4, 2, 1, 5, 3]))
println(bubblesort([5, 4, 3, 2, 1]))
```

```
println(bubblesort(["singkong", "programming",  
"language"]))  
println(bubblesort(42153))
```

Hasilnya adalah:

```
[1, 2, 3, 4, 5]
```

```
[1, 2, 3, 4, 5]
```

```
null
```

```
null
```

Kita menggunakan operator `!`, yang berfungsi sebagai not.

- Apabila diberikan argumen yang tepat, fungsi `is_array_and_of` mengembalikan `true` (BOOLEAN). Not `true` adalah `false`, sehingga blok `return null` tidak dikerjakan.
- Apabila diberikan argumen yang tidak tepat, fungsi `is_array_and_of` mengembalikan `false`. Not `false` adalah `true`, dan blok `return null` dikerjakan, sehingga fungsi dihentikan.

**Catatan:** Masih terkait dynamically typed di Singkong, walau FUNCTION kita dapat menentukan berapa jumlah argumen yang valid, kita tidak dapat menentukan apa saja tipe argumennya. Dengan demikian, kita perlu lebih banyak melakukan pemeriksaan tipe secara manual.

Pada statically typed, pemeriksaan ini tidak diperlukan, karena pemeriksaan tipe dilakukan sebelum program berjalan (misal pada saat kompilasi dilakukan).

Sampai di sini, pembahasan kita sebenarnya sudah selesai.

---



Barangkali ada yang sedikit mengganjal. Fungsi mengurutkan bilangan tampaknya sering sekali dibutuhkan bukan? Apakah sebagai pembuat program, kita bahkan harus membuat fungsi ini sendiri?

Tentu saja, jawabannya adalah tidak. Terlepas dari kompleksitas algoritmanya, sebagai pembuat program, kita mungkin menginginkan apa yang kita butuhkan telah tersedia. Sebagai contoh, di Singkong, kita dapat menggunakan fungsi-fungsi berikut untuk mengurutkan: `sort_number`, `sort_boolean`, `sort_string`, `sort_array`, `sort_hash`, `sort_date`, `sort_rect_array_of_number_by_index`. Untuk membalik hasil pengurutan, kita bisa menggunakan fungsi `reverse`.

Sebagai contoh, kita bisa ketikkan pada tab Interactive:

```
> sort_number([4, 2, 1, 5, 3])
[1, 2, 3, 4, 5]
> reverse(sort_number([4, 2, 1, 5, 3]))
[5, 4, 3, 2, 1]
```

**NUMBER** (bilangan) diurutkan berdasarkan kecil besarnya.

```
> sort_boolean([true, false])
[false, true]
```

Pada **BOOLEAN**, `false` diasumsikan lebih kecil dari `true`.

```
> sort_string(["singkong", "programming",
"language"])
["language", "programming", "singkong"]
```

**STRING** diurutkan secara leksikografik.

```
> sort_array([[1,2,3], [], [1,2,3,4], [1,2]])  
[[], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
```

ARRAY diurutkan berdasarkan jumlah elemen di dalamnya.

```
> sort_hash([{"name": "Singkong"}, {}])  
[{}, {"name": "Singkong"}]
```

HASH diurutkan berdasarkan jumlah pemetaan di dalamnya.

```
> sort_date([@, @2000, day(@, 123)])  
[2000-01-01 00:00:00, 2024-03-22 22:29:00,  
2024-07-23 22:29:00]
```

DATE diurutkan berdasarkan waktu sebelum/sesudah.

Terlepas dari berbagai tipe data yang belum kita bahas (misal HASH dan DATE), kita melihat bahwa berbagai fungsi pengurutan telah disertakan.

Tapi, buku ini membahas algoritma. Pengurutan, misal dengan bubble sort, adalah contoh yang seru ketika membahas dasar-dasar algoritma dan implementasinya. Penulis senang dan banyak belajar kembali ketika menyajikan tulisan ini. Semoga Anda pun menikmati membacanya.

## Contoh Algoritma Greedy

**Penulis: Budiman**

Anggaplah kita sedang akan membayar belanjaan, dengan total 8.800. Ketika kita membayar dengan uang 10.000, kasir perlu mengembalikan 1.200.

Saat ini, kasir memiliki banyak uang koin 100, 200, 500, dan 1000. Namun, tidak terdapat koin dengan nilai 1.200. Oleh karena itu, kasir perlu memberikan kembalian yang terdiri dari gabungan beberapa koin.

Tentu saja, kasir bisa memberikan kembalian berupa:

- 12 keping koin 100.
- Atau 6 keping koin 200.
- Atau 2 keping koin 500 dan 2 keping koin 100. Total 4 koin.
- Atau, 1 keping koin 1.000 dan 1 keping koin 200. Total 2 koin.
- Dan kombinasi lainnya.

Supaya tidak keliru menghitung dan pelanggan tidak perlu menerima banyak koin, kasir ingin memberikan kembalian dengan sesedikit mungkin jumlah koin.

Ini adalah permasalahan yang kita hadapi, yang akan kita selesaikan dalam bentuk algoritma dan program.

---

Terkait kembalian 1.200 dengan keping koin paling sedikit, dengan mudah kita menjawab kombinasi 1.000 dan 200, bukan?

Tapi kita perlu menentukan algoritmanya, supaya berlaku untuk nilai kembalian lain. Selanjutnya, kita dapat tampilkan di layar, sehingga kasir cukup mengikuti petunjuk tersebut, dan tidak perlu repot menentukan kombinasi koin yang pas.

Dari mana kita menentukan 1.000 dan 200? Barangkali:

- Kita selalu menggunakan koin dengan nilai paling besar, sampai ini tidak dimungkinkan. Misal, dengan kembalian 1.200 dan koin dengan nilai terbesar adalah 1.000, kita hanya dapat memberikan 1 koin 1.000.
- Setelah itu, kembalian yang tersisa adalah 200. Kembali kita gunakan aturan yang sama:
  - o Koin 1.000 tidak dimungkinkan, karena kita hanya butuh 200.
  - o Koin 500 tidak dimungkinkan dengan alasan serupa.
  - o Koin 200 rupanya pas, sebanyak 1 keping.
- Karena dengan 1 keping koin 1.000 dan 1 keping 200 sudah sesuai, maka tugas kita pun selesai.

Perhatikanlah bahwa kita selalu memilih opsi yang optimal pada setiap tahap. Misal, dengan sebanyak mungkin koin dengan nilai paling besar. Andaikata kasir harus memberikan kembalian 4.000, maka pilihannya adalah 4 keping koin 1.000.

Cara yang kita lakukan adalah contoh dari algoritma greedy. Kita memilih yang paling optimal pada kondisi lokal yang dihadapi (misal set opsi yang dapat dipilih), untuk setiap tahapannya.

---

Dalam hal ini, kita tidak mengubah opsi yang telah dipilih sebelumnya (yang sudah dirasa paling optimal pada tahapan sebelumnya tersebut). Oleh karena itu, ada kemungkinan, solusi akhir yang dihasilkan bukanlah solusi yang paling optimal. Kita akan melihat contohnya nanti.

Sekarang, seperti pada bab sebelumnya, mari kita siapkan pseudocodenya.

```

sort array of coin in ascending order
c <- array of coin
s <- empty array
n <- amount
i <- length of c - 1
repeat
    repeat
        append c[i] to s
        n <- n - c[i]
    until n < c[i]
    i <- i - 1
until n = 0

```

Lebih sederhana dari contoh pengurutan sebelumnya bukan? Mari kita bahas:

- Pertama, kita urutkan array koin yang dimiliki, dari kecil ke besar. Dalam contoh kasir kita, ini akan berupa [100, 200, 500, 1000].
- Variabel yang kita gunakan adalah:
  - o c: array koin
  - o s: array yang saat ini kosong, namun nantinya akan menampung keping koin kembalian.
  - o n: uang kembalian
  - o i: berupa indeks array koin, dari koin dengan nilai paling besar. Dalam hal ini, indeks adalah sama dengan jumlah elemen dalam array dikurangi 1, karena indeks dimulai dari 0.
- [Diwarnai merah] Pada dasarnya, kita mengulang sampai jumlah uang kembalian adalah 0.

- [Diwarnai biru] Pada setiap tahap, kita:
  - o Menambahkan koin dengan nilai terbesar saat ini ke array koin kembalian.
  - o Mengurangi jumlah uang kembalian sebesar nilai koin tersebut.
  - o Mengulang sampai jumlah uang kembalian tersisa lebih kecil dari nilai koin.
- [Diwarnai hijau] Kita perlu pindah ke koin dengan nilai yang lebih kecil. Oleh karena itu, kita mengurangi indeks array. Sehingga, apabila sebelumnya koin 1.000, kita pindah ke koin 500, dan seterusnya.

Sekarang, mari kita lihat contoh programnya.

---

Ketikkanlah source code berikut di tab Editor dari Singkong.jar. Kita dapat pula menyimpannya misal sebagai coin.singkong.

```
var unsorted_c = [1000, 200, 100, 500]
var c = sort_number(unsorted_c)
println(c)
var n = 1200
var s = []

var i = len(c) - 1

repeat {
  repeat {
    if (n < c[i]) {
      return null
    }
    println("target_value: " + n + " ||
current_coin: " + c[i])
    s + c[i]
```

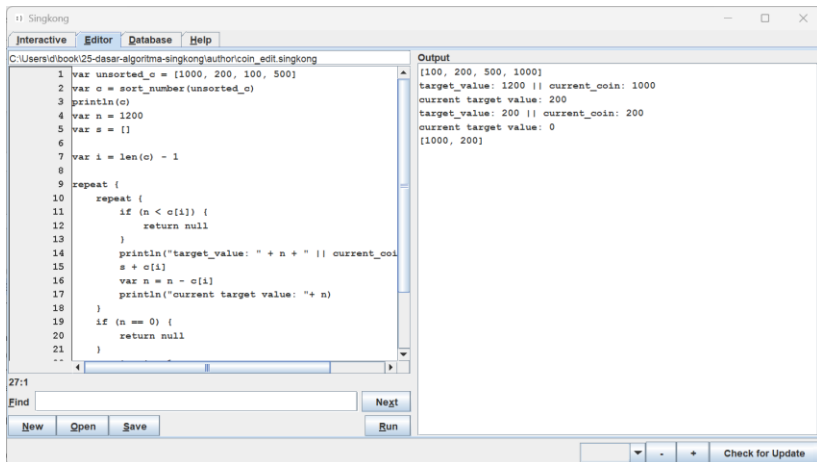
```

        var n = n - c[i]
        println("current target value: "+ n)
    }
    if (n == 0) {
        return null
    }
    var i = i - 1
}

println(s)

```

Kemudian, kliklah tombol Run.



Keluarannya adalah sebagai berikut:

```

[100, 200, 500, 1000]
target_value: 1200 || current_coin: 1000
current target value: 200
target_value: 200 || current_coin: 200
current target value: 0
[1000, 200]

```

Bisa kita lihat, program ini selalu menggunakan koin dengan nilai terbesar untuk dapat lebih cepat mengurangi jumlah uang kembalian, yang dianggap sebagai pilihan paling optimal pada setiap tahap.

---

Berikut adalah penjelasan kode programnya:

```
var unsorted_c = [1000, 200, 100, 500]
var c = sort_number(unsorted_c)
println(c)
var n = 1200
var s = []

var i = len(c) - 1
```

Pada tahap ini kita membuat variable-variable yang akan menampung data ketika menjalankan program, dan penambahan baris print untuk memperjelas proses ketika program di-run dalam editor. Berikut penjelasan singkat tentang variable yang digunakan:

- 1) **(unsorted\_c) dan (c)**: variable yang menampung set koin sebelum dan setelah disorting dengan fungsi bawaan `sort_number`. Tentu saja kita bisa menggunakan fungsi pengurutan yang dibuat di bab sebelumnya.
- 2) **(n)**: nilai yang akan dipecahkan dengan set koin **(c)**. Dalam hal ini, adalah jumlah uang kembalian.
- 3) **(s)**: array untuk menampung koin yang dibutuhkan untuk memecahkan nilai **(n)**.
- 4) **(i)**: variable yang menunjukkan indeks terbesar pada set **(c)**.



Untuk blok repeat berikut:

```
repeat {
    if (n < c[i]) {
        return null
    }
    println("target_value: " + n + " ||
current_coin: " + c[i])
    s + c[i]
    var n = n - c[i]
    println("current target value: "+ n)
}
```

Yang kita lakukan adalah:

- Menambahkan nilai koin (**c[i]**) ke array (**s**).
- Mengurangi nilai (**n**) dengan nilai **c[i]**.
- Keluar dari repeat setelah kondisi (**n < c[i]**) terpenuhi.

Sementara, untuk blok repeat luar:

```
repeat {

    if (n == 0) {
        return null
    }
    var i = i - 1
}
```

Kita keluar dari perulangan setelah jumlah uang kembalian (**n**) sudah 0. Apabila belum, kita pindah ke koin dengan nilai lebih kecil (indeks dikurangi 1).

---

Apakah algoritma kita selalu menghasilkan solusi yang optimal? Dengan koin kita saat ini [100, 200, 500, 1000], bisa kita lihat adalah ya.

Nilai kembalian	Hasil Program	Hasil Optimal
1.200	[1000, 200]	[1000, 200]
800	[500, 200, 100]	[500, 200, 100]

Akan tetapi, kalau kita ubah koin kita menjadi [100, 700, 1000] (anggaplah koin 700 benar-benar ada) dan uang kembalian adalah 1.500, maka program akan menghasilkan [1000, 100, 100, 100, 100, 100]. Ini tentu saja bukan yang paling optimal, yang seharusnya adalah [700, 700, 100].

Kemudian, untuk koin [200, 700, 1000] dan uang kembalian 1.100, program gagal menghitung (error), sementara hasil optimal yang seharusnya adalah [700, 200, 200].

Demikianlah contoh pembahasan algoritma greedy.

## Memulai Pemrograman Komputer

Di bab sebelumnya, kita mengenal sekilas algoritma, struktur data, dan pemrograman dengan bahasa Singkong. Mari kita bahas beberapa aspek lain yang mungkin berguna, ketika kita akan memulai pemrograman komputer.

Perkembangan teknologi hardware komputer memungkinkan hadirnya komputer yang berukuran lebih kecil dari sebuah kartu nama dan membutuhkan daya sekitar setengah Watt, sampai komputer super yang membutuhkan ruangan ratusan meter persegi, dengan jutaan core dan/atau membutuhkan daya jutaan Watt.

Ketika dikombinasikan dengan perubahan gaya hidup, komputer tidak hanya tersimpan di kantor atau di atas meja, melainkan juga ikut ke manapun kita pergi. Bukan sekedar untuk bekerja, melainkan dapat membuat hidup sehari-hari menjadi lebih mudah dan nyaman.

Komputer-komputer tersebut akan membutuhkan program mendasar, dalam berbagai bentuk. Baik sekedar untuk menjalankan perangkat dan menyediakan fungsionalitas spesifik (firmware), ataupun dalam bentuk berbagai sistem operasi yang umum kita kenal, seperti Windows, macOS, dan Linux (yang rata-rata telah dikembangkan selama puluhan tahun dan source codenya terdiri dari puluhan juta baris kode).

Sebagai pengguna komputer, kita umumnya bekerja dengan program tertentu, yang berjalan pada sistem operasi tersebut. Bisa berupa pengolah kata, misal untuk mengetik naskah buku ini. Atau program untuk mengelola pembukuan perusahaan. Atau web browser untuk mencari informasi lewat internet. Atau memainkan game, baik di ponsel, laptop, ataupun konsol game.

Program-program yang disebutkan tersebut tentunya perlu dibuat. Entah membutuhkan beberapa baris atau jutaan baris, ada orang yang perlu mengetikkan kode untuk membuat program tersebut. Walau, dengan bantuan program tertentu atau kecerdasan artifisial, pemrograman mungkin menjadi lebih mudah dan/atau kode yang perlu ditulis menjadi lebih sedikit.

Sama seperti berbagai bidang lain, kita perlu memulai untuk mencapai hasil tertentu. Apabila kita ingin memulai pemrograman komputer, memahami hal seputar pemrograman dan menulis beberapa baris kode akan membawa kita ke program yang lebih besar. Secara bertahap, kita akan beralih ke program yang lebih kompleks, yang mungkin membutuhkan pemahaman tertentu di luar teknis pemrograman itu sendiri. Sebagai contoh, kita perlu memahami dasar-dasar akuntansi untuk membuat program pendukung bisnis perusahaan.

Program tertentu, apabila tidak dibuat dengan baik, dapat berakibat serius dan mungkin saja fatal. Selain teknis pemrograman dan pemahaman baik akan bidang tertentu, kita juga akan membutuhkan tim kerja yang terkelola.

**Catatan:** Ada masa dimana komputer tidaklah umum. Internet dan ponsel seperti sekarang ini barangkali tidak terbayangkan.

Sekarang, semua serba terhubung. Sebuah microcontroller seharga beberapa puluh ribu bisa diprogram untuk membaca dari berbagai sensor, dan lalu terhubung ke internet. Akses internet tersedia meluas dan biayanya relatif terjangkau. Anak-anak di sekolah dasar mungkin telah belajar menggunakan ponsel dan komputer.

Masing-masing dari kita tentunya bisa membayangkan apa saja program komputer yang bisa dibuat dan peluang yang mengikutinya.

## Profesional atau membantu pekerjaan

Saat ini, pemrograman menjadi terjangkau dan tampaknya sekaligus menjadi kemampuan mendasar. Tapi, bagaimana kalau kita tidak ingin menjadikan pemrograman sebagai mata pencaharian? Bagaimana kalau kita sekedar ingin membuat program untuk kebutuhan sendiri?

Di dunia pengembangan program komputer, kita mengenal istilah end-user programming (dan end-user development serta end-user software engineering), dimana tujuan utamanya adalah untuk menghasilkan program komputer untuk digunakan sendiri. Mungkin untuk sekedar hobi atau mendukung pekerjaan. Pembuat programnya sendiri bisa saja berpengalaman dan menggunakan bahasa pemrograman yang tidak sederhana.

Kebalikannya adalah professional programming, dengan tujuan membuat program untuk digunakan pihak lain. Pemrogram dibayar untuk menghasilkan program dan memastikan program berjalan baik untuk sekian waktu tertentu. Pemrogram tersebut bisa saja tidak berpengalaman dan menggunakan bahasa pemrograman yang sederhana.

Prosesnya mungkin akan berbeda. Sebagai contoh, pemrogram end-user mungkin membuat program sebatas yang diperlukan, ketika terpikir saja. Pemrogram profesional mungkin mengharapkan adanya gambaran yang lebih eksplisit karena apabila kebutuhan tidak diketahui atau program terus dirombak, biaya pengembangan akan menjadi lebih mahal.

Pemrogram end-user mungkin sudah yakin ketika program berjalan dengan baik ketika digunakan oleh dirinya sendiri. Tapi, pemrogram profesional mungkin harus melakukan pengujian dengan berhati-hati, mempertimbangkan berbagai faktor dan kemungkinan.

Penulis berharap, buku ini juga mungkin dapat berguna apabila Anda sekedar membuat program untuk hobi atau kebutuhan sendiri.

## Mengenal dan menggunakan sistem operasi

Ada masa, untuk sekedar menggunakan sistem operasi komputer, kita membutuhkan pelatihan, membaca buku, atau setidaknya bertanya kepada orang yang lebih tahu.

Sekarang, kita mungkin cukup membiasakan diri. Mungkin butuh waktu dan bertanya, tapi tidak perlu sampai ikut pelatihan lengkap atau membaca buku utuh. Puluhan tahun dalam pengembangan, dengan puluhan juta baris kode, menjadikan sistem operasi komputer menjadi begitu nyaman dan canggih.

Apabila boleh, penulis menyarankan untuk memahami lebih baik sistem operasi yang digunakan, ketika kita ingin membuat program.

Setidaknya, kita perlu terbiasa dan nyaman untuk melakukan instalasi berbagai program. Kita juga mungkin perlu memahami dengan baik bagaimana melakukan konfigurasi dasar sistem operasi (atau, sekalian instalasi dari awal). Terbiasa bekerja dengan file sistem mungkin juga akan membantu. Sese kali melakukan troubleshooting kendala teknis penggunaan komputer tidaklah merugikan.

## Mempelajari bahasa pemrograman

Ketika kita menggunakan sistem operasi komputer, kita mungkin memberikan perintah, katakanlah, untuk menghapus file tertentu.

Di user interface grafikal (dengan mouse atau layar sentuh), kita cukup membuka misal file manager, memilih file, dan memilih item menu tertentu untuk menghapus file (atau dengan menekan tombol

tertentu pada keyboard, apabila ada). Tergantung sistem operasi yang digunakan, kita juga bisa bekerja pada command line interface untuk menjalankan program yang menghapus file, dengan cara mengetikkan nama program dan nama file.

Dalam contoh tersebut, perlukah kita membuat program hanya untuk sekedar menghapus file tersebut? Tidak. Kita menggunakan program yang sudah dibuat sebelumnya.

Sekarang, katakanlah, kita perlu melakukan sesuatu yang fungsionalitasnya tidak disediakan langsung oleh sistem operasi, dan program untuk melakukannya belum tersedia. Benar, seperti yang Anda sudah bayangkan, ada program yang perlu dibuat.

Ketika membahas algoritma di awal buku ini, kita membahas bahwa dalam kehidupan sehari-hari, kita dapat meminta bantuan kepada teman, dan menyampaikan detailnya dengan bahasa yang sudah sama-sama dipahami. Dengan sistem operasi komputer, untuk membuat program, kita berkomunikasi dengan bahasa pemrograman.

Dari sisi domain aplikasinya, kita mengenal bahasa pemrograman general purpose (membangun jenis aplikasi apa saja) atau domain-specific (dikhususkan untuk membangun jenis aplikasi tertentu). Sebagai contoh, Python, Java, dan Singkong adalah bahasa pemrograman general purpose. Sementara, SQL dikhususkan untuk bekerja dengan sistem database relasional.

Ketika kita berhak memutuskan sepenuhnya bahasa apa yang ingin digunakan, dan program yang kita ingin buat dimungkinkan untuk dibuat dengan bahasa tersebut, maka secara teknis, ini dapat menjadi alasan memilih sebuah bahasa.

Hanya saja, kadangkala, alasan tidak selalu bersifat teknis, seperti contoh berikut:

- Seseorang yang baru mempelajari pemrograman mungkin akan sering bertanya kepada rekan yang lebih terbiasa, dan sangat mungkin memilih untuk menggunakan bahasa yang sama.
- Seorang mahasiswa/i ilmu komputer mungkin diharuskan untuk mempelajari bahasa pemrograman tertentu, ketika mempelajari mata kuliah tertentu.
- Seseorang yang ingin berkontribusi pada pengembangan software tertentu perlu menyesuaikan diantaranya dengan bahasa pemrograman yang digunakan.
- Mahir dalam bahasa pemrograman tertentu mungkin menjadikan seseorang lebih mudah dalam mendapatkan pekerjaan.
- Seseorang mungkin cocok dengan produk-produk tertentu dari sebuah perusahaan teknologi, dan menggunakan bahasa pemrograman yang disediakan/diharuskan.
- Pada kurun waktu tertentu, sebuah bahasa mungkin menjadi sangat populer.
- Terdapat sejumlah bahasa pemrograman yang lebih umum di industri tertentu.
- Kita mungkin bekerja dalam tim.

### Bekerja dalam tim

Katakanlah kita membuat program untuk kebutuhan sendiri, dan kita dapat menunggu dengan sabar sampai program tersebut selesai dibuat, maka kita bisa bekerja sendirian saja.

Tapi, bagaimana kalau kita tidak dapat melakukan semuanya sendiri? Misal, kita mungkin nyaman dengan algoritma dan pemrograman, namun tidak dapat mendesain user interface dengan baik. Apabila kita paksakan untuk tetap melakukannya sendiri, program yang dibuat mungkin terasa lebih sulit untuk digunakan.



Yang lebih serius adalah misal kita memahami dengan baik pemrograman, tapi tidak memahami dasar-dasar akunting. Apabila kita paksakan untuk tetap membangun aplikasi pendukung bisnis, program yang dibuat mungkin saja keliru. Bukan sekedar sulit digunakan.

Seringkali, tantangan terbesar adalah memahami kebutuhan pengguna program, apabila ketika kita melakukannya secara profesional, untuk pihak lain. Tantangan komunikasi akan tetap ada, dan dibutuhkan anggota tim yang spesifik menangani kebutuhan ini.

Bahkan, ketika kita terbiasa dengan pemrograman pun, kita mungkin tidak memiliki waktu yang cukup. Dengan demikian, kita akan tetap membutuhkan rekan kerja.

Dalam membangun sesuatu yang lebih besar, bekerja dalam tim tidaklah dapat dihindari. Tapi, karena buku ini adalah tentang dasar-dasar algoritma, struktur data, dan pemrograman, mari kita fokuskan pada hal ini terlebih dahulu. Dengan demikian, kita dapat berkontribusi dengan baik ketika nantinya bekerja dalam tim.

*Halaman ini sengaja dikosongkan*

## Bekerja Dengan Array

Di contoh algoritma pengurutan bilangan, kita menyusun bilangan-bilangan dalam sebuah daftar, yang diimplementasikan dalam bahasa Singkong sebagai sebuah ARRAY.

Kapanpun kita perlu bekerja dengan daftar yang item di dalamnya memiliki posisi indeks tertentu, kita bisa gunakan ARRAY. Sebagai contoh: daftar bilangan, daftar belanja, daftar kontak, daftar produk, dan lain sebagainya.

Posisi indeks tertentu dalam hal ini artinya kita bisa merujuk pada elemen pertama, kedua, ketiga, keempat, terakhir, dan sebagainya. Tentu saja, kita bisa mengubah posisi ini, seperti halnya ketika kita menukar posisi bilangan dalam contoh pengurutan sebelumnya. Kita juga dapat mengetahui posisi elemen berdasarkan nilainya.

Lebih lanjut, kita mungkin perlu bekerja dengan sebagian dari daftar, misal mendapatkan lima elemen terakhir sebagai ARRAY baru.

Dan, yang mungkin sangat penting, kita bisa menambah elemen ke atau mengurangi elemen dari daftar (yang mana, ketika mengurangi, posisi indeks elemen tertentu dapat berubah).

Sebagai sebuah ARRAY, kita juga dapat memeriksa apakah terdapat elemen tertentu di dalamnya (kalau ada, berapa jumlahnya), atau memeriksa apakah item di dalamnya berupa tipe tertentu (misal, apakah merupakan bilangan).

Apabila diperlukan, kita pun dapat mengubah ARRAY, misal mengacak, mengurutkan, menjadikan set dengan hanya nilai unik, dan sebagainya.

Singkong juga mendukung rectangular ARRAY, misal ketika kita perlu bekerja dengan matriks.

Apabila kebutuhan kita dapat dipenuhi dengan ARRAY di Singkong, kita mungkin tidak memerlukan struktur lainnya.

## Membuat Array

Sintaks sebuah ARRAY adalah:

- Diawali dengan [
- Diikuti daftar elemen. Apabila daftar ini tidak kosong, maka antara elemen dipisahkan sebuah koma.
- Diakhiri dengan ]

Dengan demikian, berikut adalah sebuah ARRAY yang tidak memiliki elemen di dalamnya:

```
var a = []
```

Bisa kita lihat, panjangnya adalah 0:

```
> len(a)
0
```

Kita bisa menuliskan daftar elemen secara langsung, misal seperti contoh pengurutan sebelumnya:

```
var a = [4, 2, 1, 5, 3]
```

Sekarang, kalau kita ingin membuat ARRAY dari 1 sampai 100, apakah kita harus menuliskannya secara manual?

Tentu saja tidak. Kita bisa gunakan fungsi yang mengembalikan ARRAY, seperti contoh berikut:

```
> var a = range(1,101)
> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100]
```

Kalau kita perlu membuat ARRAY yang terdiri dari bilangan dengan kelipatan tertentu? Misal ganjil atau genap saja dari 1 sampai 100?

```
> range(1, 101, 2)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25,
27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49,
51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73,
75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97,
99]
```

```
> range(2, 102, 2)
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24,
26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48,
50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72,
74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96,
98, 100]
```

Tanpa fungsi bantu, kita bisa lakukan ini secara manual, walaupun kodenya lebih panjang dan waktunya lebih lama:

```
var a = []
var x = 1
```

```
repeat {
    if (x % 2 == 1) {
        a + x
    }
    var x = x + 1
    if (x > 100) {
        return null
    }
}
println(a)
```

Keluarannya sama saja:

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25,
27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49,
51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73,
75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97,
99]
```

Sejumlah fungsi bawaan bahasa Singkong bekerja dan atau mengembalikan ARRAY. Bacalah buku referensi Singkong: Mengenal dan Menggunakan Bahasa Pemrograman Singkong (ISBN: 978-602-52770-1-6, gratis, <https://nopri.github.io/singkong.pdf>) apabila diperlukan. Pada Singkong.jar, tab Help juga menyajikan referensi serupa.

## Bekerja Dengan Indeks

Sebagaimana dicontohkan sebelumnya, indeks ARRAY adalah selalu berupa NUMBER. Di Singkong, NUMBER dapat berupa bilangan bulat ataupun pecahan, namun indeks ARRAY selalu berupa bilangan bulat.

Indeks selalu dimulai dari 0 sampai panjang elemen dikurangi 1. Dengan demikian, apabila ARRAY terdiri dari 5 elemen, maka indeks

yang valid adalah selalu 0 sampai 4. Apabila kita mencoba bekerja dengan indeks di luar rentang yang valid, maka NULL akan dikembalikan. Tidak ada error yang menyebabkan program dihentikan.

Kita dapat mengakses indeksnya secara langsung dengan operator [], atau mengubah nilai pada indeks tertentu dengan fungsi `set`.

```
> var a = range(1, 6)
> a
[1, 2, 3, 4, 5]

> type(a[-5])
"NULL"

> type(a[5])
"NULL"

> a[0]
1

> a[1]
2

> a[2]
3

> a[3]
4

> a[4]
5

> set(a, 0, a[0] * 1000)
[1000, 2, 3, 4, 5]
```

```
> a
[1000, 2, 3, 4, 5]

> set(a, len(a)-1, a[len(a)-1] * 1000)
[1000, 2, 3, 4, 5000]
```

**Catatan:** kita menggunakan fungsi `type` untuk mendapatkan tipe. Dalam contoh kode di halaman sebelumnya, ini perlu, karena `NULL` tidak ditampilkan. Nilainya tetap `null`, hanya saja tidak ditampilkan.

Sebagai alternatif dari menggunakan indeks secara langsung, kita bisa menggunakan fungsi-fungsi bantu berikut:

```
> var a = range(1, 6)
> first(a)
1

> last(a)
5
```

## Iterasi Array

Ketika bekerja dengan `ARRAY`, seringkali kita perlu memeriksa setiap elemen di dalamnya. Kita sudah melakukan ini di contoh bubble sort sebelumnya, dengan perulangan `repeat`. Kita sajikan ulang versi sederhananya di sini:

```
var a = range(1, 5)
var i = 0
repeat {
  println(a[i])
  var i = i + 1
  if (i == len(a)) {
```



```
        return null
    }
}
```

Memang agak panjang karena harus memeriksa indeks ARRAY dan keluar dari perulangan secara manual. Alternatifnya adalah menggunakan fungsi bawaan `each`, seperti contoh berikut:

```
var a = range(1,5)
each(a, fn(e, i) {
    println(e)
})
```

Kita melewati sebuah FUNCTION pada argumen kedua fungsi `each`. FUNCTION tersebut harus menerima dua argumen, yang nantinya berupa elemen dan indeks ARRAY setiap kali perulangan dilakukan.

Dalam contoh tersebut, `e` adalah elemen dan `i` adalah indeks. Yang perlu diperhatikan adalah `println` dan statement lain dalam FUNCTION berada dalam scope tersendiri. Contoh efeknya adalah variabel yang dibuat dalam scope ini akan tidak tersedia begitu keluar dari scope (misal, ketika perulangan selesai).

## Menambahkan atau Mengurangi Elemen

ARRAY di Singkong dapat mengandung elemen dari berbagai tipe. Dengan demikian, kita bisa menambahkan semua tipe yang tersedia di Singkong ke dalam ARRAY yang sama. Sebagai contoh:

```
var a = [
    [],
    true,
    len,
```

```

        component("button", "hello"),
        database("org.apache.derby.jdbc.EmbeddedDriver",
"jdbc:derby:test;create=true", "", ""),
        @2024,
        fn() {},
        {"name": "Singkong"},
        null,
        12345,
        "Singkong"
    ]

println(a)

```

Apabila dijalankan, keluarannya kurang lebih adalah sebagai berikut (apabila pembuatan database gagal, misal karena tidak memiliki hak tulis pada direktori aktif, maka elemen ke-4 akan bernilai null):

```

[[[], true, built-in function: len: returns the length of
STRING, HASH, or ARRAY
arguments: 1: STRING, HASH, or ARRAY
return value: NUMBER
, COMPONENT: button (hello), DATABASE
(URL=jdbc:derby:test;create=true, user=,
driver=org.apache.derby.jdbc.EmbeddedDriver), 2024-01-01
00:00:00, fn()
{
}
, {"name": "Singkong"}, null, 12345, "Singkong"]

```

Lebih lanjut, ARRAY selalu bisa mengandung ARRAY dari ARRAY dari ARRAY, dan seterusnya.

Untuk menambahkan elemen ke ARRAY, kita bisa menggunakan operator `+`, fungsi `array_extend`, fungsi `array_extend_all`, atau fungsi `push`. Perhatikanlah contoh berikut:

```

> [1,2] + 3
[1, 2, 3]

```

```

> [1,2] + [3]
[1, 2, [3]]
> array_extend([1,2], [3])
[1, 2, 3]
> array_extend_all([[1,2], [3,4], 5])
[1, 2, 3, 4, 5]
> push([1,2], [3])
[1, 2, [3]]

```

Untuk menghapus elemen dari ARRAY, kita bisa menggunakan operator - atau fungsi `pop`. Perhatikanlah contoh berikut:

```

> [1,2,3,4,4] - 4
[1, 2, 3]
> pop([1,2,3,4])
[1, 2, 3]

```

Yang perlu diperhatikan di sini adalah menambahkan atau mengurangi elemen ARRAY dengan operator + atau - akan mengubah ARRAY itu sendiri, kecuali fungsi `push`, `pop`, `array_extend`, atau `array_extend_all` digunakan. Contoh berikut tidak mengubah ARRAY awalnya, karena ARRAY baru dihasilkan.

```

> var a = [1,2,3,4]
> var b = push(a, 5)
> a
[1, 2, 3, 4]

> b
[1, 2, 3, 4, 5]
> var c = pop(a)
> c
[1, 2, 3]

```

```

> a
[1, 2, 3, 4]

> var b = array_extend(a, [5])
> a
[1, 2, 3, 4]

> b
[1, 2, 3, 4, 5]

> var c = array_extend_all([a, [5,6], [7]])
> a
[1, 2, 3, 4]

> c
[1, 2, 3, 4, 5, 6, 7]

```

Contoh yang mengubah ARRAY awalnya:

```

> var a = [1,2,3,4]
> a+5
[1, 2, 3, 4, 5]

> a
[1, 2, 3, 4, 5]
> a - 5
[1, 2, 3, 4]

> a
[1, 2, 3, 4]

```

Dalam kondisi kita harus mengopi sebuah ARRAY ke ARRAY baru, kita bisa menggunakan fungsi `array_copy` dari modul `util`. Modul ini perlu di-load sebelumnya. Perhatikanlah contoh berikut:

```
> load_module("util")
> var a = ["hello", "world"]
> var b = array_copy(a)
> set(a, 1, "hello world")
["hello", "hello world"]

> println(a, b)
["hello", "hello world"]
["hello", "world"]
```

## Memeriksa Elemen

Dalam bekerja dengan ARRAY, sangat umum kita perlu memeriksa apakah ARRAY kosong atau berisi elemen tertentu. Untuk itu, kita bisa menggunakan fungsi `empty` atau `in`. Contoh:

```
> var a = [1,2,3]
> empty(a)
false

> in(a, 1)
true

> in(a, 2)
true

> in(a, 3)
true

> in(a, 4)
false
```

Sementara, apabila yang kita butuhkan adalah mengetahui indeks dari elemen tertentu atau berapa jumlah elemen tertentu dalam ARRAY, kita bisa menggunakan fungsi `index` dan `count`. Contoh:

```
> var a = [1,1,2,2,3,3,4]
> index(a, 1)
[0, 1]
```

```
> index(a, 2)
[2, 3]
```

```
> index(a, 3)
[4, 5]
```

```
> index(a, 4)
[6]
```

```
> count(a, 4)
1
```

```
> count(a, 5)
0
```

Fungsi lain seperti `is`, `is_array_and_of` dan `is_array_of` mungkin berguna untuk memeriksa apakah merupakan ARRAY dan merupakan ARRAY dengan elemen bertipe tertentu:

```
> is([1,2,3], "ARRAY")
true
```

```
> is_array_of([1,2,3], "NUMBER")
true
```

```
> is_array_of(123, "NUMBER")
ERROR: [line: 1] type mismatch: got=NUMBER and
STRING, want=ARRAY and STRING
```

```
> is_array_and_of([1,2,3], "NUMBER")
true
```

```
> is_array_and_of(123, "NUMBER")
false
```

## Membandingkan Array

Untuk membandingkan apakah dua ARRAY merupakan ARRAY yang sama, kita bisa menggunakan operator `==` (elemen sama, urutan sama) atau fungsi `array_equals` (elemen sama, urutan boleh beda).

Contoh dengan operator `==`:

```
> [] == []
true
> [1,2] == [1,2]
true
> [1,2] == [2,1]
false
> [[1,2]] == [[2,1]]
false
> [1,2] == [1,2,3]
false
```

Kebalikan operator `==` adalah `!=`.

Contoh perbandingan penggunaan operator `==` dengan fungsi `array_equals`:

```
> [1,2] == [2,1]
false
> array_equals([1,2], [2,1])
true
> array_equals([1,2], [1,2])
true
```

Untuk mendapatkan perbedaan antara dua ARRAY, kita bisa menggunakan fungsi `array_diff` dari modul `util`, seperti contoh berikut:

```
> load_module("util")
> array_diff([1,2,3,4,5], [1,3,5])
[2, 4]
```

### Mendapatkan Slice Array

Dalam bekerja dengan ARRAY, misal daftar bilangan, atau daftar produk, terkadang kita perlu mendapatkan sebagian (*slice*) dari ARRAY. Misal 5 elemen pertama atau 5 elemen terakhir. Untuk kebutuhan tersebut, kita dapat menggunakan fungsi `slice`. Contoh:

```
> var a = range(1, 10)
> a
[1, 2, 3, 4, 5, 6, 7, 8, 9]

> var b = slice(a, 0, 5)
> b
[1, 2, 3, 4, 5]
> var c = slice(a, len(a)-5, len(a)+1)
> c
[5, 6, 7, 8, 9]
```

Untuk mendapatkan slice ARRAY dari indeks 1, fungsi `rest` dapat digunakan:

```
> var d = rest(a)
> d
[2, 3, 4, 5, 6, 7, 8, 9]
```



## Nilai Acak

Untuk mendapatkan nilai acak dari ARRAY, kita bisa menggunakan fungsi `random`, seperti contoh berikut:

```
> random(range(1, 1000))  
819
```

Untuk mengacak isi ARRAY, kita bisa menggunakan fungsi `shuffle`. Fungsi ini mengubah ARRAY awalnya:

```
> var a = [1,2,3,4,5]  
> shuffle(a)  
[4, 5, 2, 3, 1]  
> a  
[4, 5, 2, 3, 1]
```

## Menggunakan Array Dalam Array

Bagaimana kalau kita perlu memiliki daftar produk, tapi yang kita simpan tidak hanya berupa nama? Sebagai contoh, kita perlu menyimpan kode produk, nama, harga, dan stok saat ini. Dengan demikian, alih-alih berupa bilangan seperti contoh pengurutan sebelumnya, kita memiliki data gabungan.

Dengan menggunakan ARRAY saja, kita bisa menggunakan ARRAY dalam ARRAY untuk menstrukturkan daftar produk kita. Karena kita tahu bahwa ARRAY bisa berisi elemen dengan tipe apapun, termasuk ARRAY (dan ARRAY dalam ARRAY, dan seterusnya), kita bisa menggunakan contoh struktur berikut:

```
var p = [  
  [4, "Produk D", 4000, 40],  
  [3, "Produk C", 3000, 30],  
  [2, "Produk B", 2000, 20],  
  [1, "Produk A", 1000, 10]  
]
```

Bagaimana kalau kita diminta untuk mengurutkan produk berdasarkan harganya, dari kecil ke besar (ascending)? Daftar produk kita malah terurut besar ke kecil kalau menurut kode, harga, dan stok. Bukan itu yang kita inginkan.

Mari kita kopikan fungsi bubblesort kita menjadi sort\_product berikut (kita bisa simpan dalam file sort\_product.singkong):

```
var sort_product = fn(a, idx) {
  if (!is_array_and_of(a, "ARRAY")) {
    return null
  }
  var n = len(a)
  repeat {
    var swapped = false
    var i = 1
    repeat {
      if (a[i-1][idx] > a[i][idx]) {
        var temp = a[i-1]
        set(a, i-1, a[i])
        set(a, i, temp)
        var swapped = true
      }
      var i = i + 1
      if (i == n) {
        return null
      }
    }
    if (swapped == false) {
      return null
    }
  }
  return a
}
```

```

var p = [
    [4, "Produk D", 4000, 40],
    [3, "Produk C", 3000, 30],
    [2, "Produk B", 2000, 20],
    [1, "Produk A", 1000, 10]
]
println(p)
sort_product(p, 2)
println(p)

```

Keluarannya adalah seperti berikut:

```

[[4, "Produk D", 4000, 40], [3, "Produk C", 3000,
30], [2, "Produk B", 2000, 20], [1, "Produk A",
1000, 10]]
[[1, "Produk A", 1000, 10], [2, "Produk B", 2000,
20], [3, "Produk C", 3000, 30], [4, "Produk D",
4000, 40]]

```

Lihatlah. Terurut dari kecil ke besar berdasarkan harga. Yang kita ubah sangatlah sedikit:

- Kita menambahkan argumen idx dan menggunakan idx ini dalam perbandingan:

```
var sort_product = fn(a, idx) {
```

dan:

```
    if (a[i-1][idx] > a[i][idx]) {
```

- Kita memeriksa bahwa ARRAY yang dilewatkan haruslah berupa ARRAY dalam ARRAY:

```
    if (!is_array_and_of(a, "ARRAY")) {
```

Sayangnya, fungsi kita:

- Belum memeriksa apakah setiap elemen adalah ARRAY data produk dengan 4 elemen di dalamnya. Kita bisa cek jumlah elemen, misalnya. Cobalah untuk menambahkan kodenya.
- Belum membandingkan selain NUMBER, misal nama produk. Bisa saja, kita membuat ARRAY dengan [nama produk 1, nama produk 2], kemudian mengurutkan dengan `sort_string` dan membandingkan dengan hasil pengurutan. Apabila sama, maka sudah terurut kecil ke besar, tidak perlu ditukar (apabila tidak sama, maka belum terurut, kita tukar). Kode yang diformat tebal berikut mungkin bisa diterapkan:

```
repeat {  
    var p1 = [a[i-1][idx], a[i][idx]]  
    var p2 = sort_string(p1)  
    if (p1 != p2) {  
        var temp = a[i-1]  
        set(a, i-1, a[i])  
        set(a, i, temp)  
        var swapped = true  
    }  
    var i = i + 1  
    if (i == n) {  
        return null  
    }  
}
```

- Belum menangani baik STRING atau NUMBER yang akan dibandingkan. Sebagai contoh, apabila tipe (dengan fungsi `type`) adalah NUMBER atau STRING, kita membandingkan dengan cara berbeda.

## Fungsi tambahan untuk Array Number

Fungsi-fungsi berikut mungkin berguna apabila kita memiliki ARRAY dari NUMBER, misal:

```
> var a = range(1, 10)
> a
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**Menjumlahkan semua elemen:**

```
> sum(a)
45
```

**Mendapatkan max dan min:**

```
> var a = range(1, 10)
> min(a)
1
```

```
> max(a)
9
```

**Mendapatkan rata-rata:**

```
> average(a)
5
```

**Mendapatkan mean, median, mode, range (dari modul util):**

```
> a + 8
[1, 2, 3, 4, 5, 6, 7, 8, 9, 8]

> load_module("util")
> mean(a)
5.3000
```

```
> median(a)
5.5000
```

```
> mode(a)
[8, 2]
```

```
> range_(a)
8
```

Kita telah membahas dasar-dasar bekerja dengan ARRAY. Untuk informasi selengkapnya, atau fungsi-fungsi lain terkait ARRAY, bacalah buku referensi Singkong atau tab Help Singkong.jar.

## Stack dan Queue

Contoh daftar bilangan yang akan diurutkan ataupun pembahasan ARRAY sebelumnya menggunakan ARRAY apa adanya. Kita tidak membutuhkan aturan tambahan.

Akan tetapi, kita mungkin perlu bekerja dengan algoritma yang membutuhkan aturan khusus, misal pada stack dan queue. Kita akan membahas dasar-dasarnya, dengan implementasi berupa ARRAY.

### Stack

Pada stack, kita menambahkan setelah elemen paling akhir, dan mengeluarkan dari elemen terakhir, yang kita kenal sebagai LIFO (Last In, First Out).

Analogi dalam kehidupan sehari-hari adalah misal pada tumpukan buku, dimana:

- Apabila ingin menyimpan buku lain ke tumpukan tersebut, kita cukup menumpuk di atasnya. Di stack, kita menyebut ini sebagai push.
- Apabila ingin mengambil buku tertentu, kita mengambil dari atas. Di stack, ini adalah pop.
- Kita mungkin ingin tahu buku apa saja yang ada di paling atas tumpukan. Di stack, ini adalah peek. Tidak ada yang berubah pada stack setelah peek.

Pada fitur undo pada pengolah kata misalnya, yang dapat kita undo langsung adalah yang terakhir kita lakukan sebelumnya. Pada kenyataannya, stack banyak digunakan dalam ilmu komputer.

Untuk bekerja dengan stack di Singkong, kita cukup menggunakan ARRAY.

Sebagai contoh, pertama, stack kita kosong:

```
> var s = []  
> s  
[]  
  
> type(s)  
"ARRAY"
```

Ukuran stack kita adalah panjang ARRAY, yang bisa didapatkan dengan `len`:

```
> len(s)  
0
```

Untuk menambahkan elemen, kita sudah membahasnya dalam [Menambahkan atau Mengurangi Elemen](#). Apabila analogi terhadap operasi stack diperlukan, kita bisa menggunakan fungsi `push`. Misal, kita `push` NUMBER 1, 2, dan 3 berikut:

```
> var s = push(s, 1)  
> var s = push(s, 2)  
> var s = push(s, 3)  
> s  
[1, 2, 3]  
  
> len(s)  
3
```

Perhatikanlah bahwa `push` membuat ARRAY baru, berbeda dengan operator `+` misalnya.



Untuk mengetahui elemen paling atas dalam stack kita, kita bisa menggunakan fungsi `last`.

```
> last(s)
3
```

```
> s
[1, 2, 3]
```

Dengan demikian, `pop` dari stack adalah kombinasi dari penggunaan fungsi `last` dan `pop`. Perhatikanlah bahwa `pop` akan membuat ARRAY baru. Contoh:

```
> var e = last(s)
> e
3
```

```
> var s = pop(s)
> s
[1, 2]
```

Ketika kita `peek` atau `pop` pada stack yang kosong, kondisi `underflow` akan terjadi. Di Singkong, `last` dan `pop` akan mengembalikan `null` apabila ARRAY kosong.

Contoh:

```
> type(last([]))
"NULL"
```

```
> type(pop([]))
"NULL"
```

## Queue

Pada queue, kita menambahkan setelah elemen paling akhir, dan mengeluarkan dari elemen yang paling awal, yang kita kenal sebagai FIFO (First In, First Out).

Analogi dalam kehidupan sehari-hari adalah misal pada antrian di kasir:

- Ketika kasir masih melayani pelanggan lain, pelanggan yang baru datang akan mengantri di akhir barisan. Pada queue, ini adalah enqueue.
- Ketika kasir selesai melayani pelanggan, yang berikutnya dilayani adalah yang berada di paling depan antrian. Pada queue, ini adalah dequeue.
- Kasir mungkin ingin mengetahui siapa saja pelanggan yang akan dilayani berikut, tanpa mengubah antrian itu sendiri. Pada queue, ini adalah peek.

Sama seperti pada stack, kita dapat implementasikan dengan ARRAY.

Contoh:

```
> var q = []
> q
[]

> type(q)
"ARRAY"

> len(q)
0
```

Untuk menambah antrian, cara sama saja dengan pada stack, yaitu menambah elemen pada ARRAY. Kali ini, mari kita gunakan operator

+, yang langsung mengubah ARRAY. Misal, kita menambahkan NUMBER 1, 2, dan 3.

```
> q + 1
[1]

> q + 2
[1, 2]

> q + 3
[1, 2, 3]

> q
[1, 2, 3]

> len(q)
3
```

Untuk peek, kita cukup menggunakan fungsi `first`. Contoh:

```
> first(q)
1

> q
[1, 2, 3]
```

Sementara, untuk dequeue, kita mengkombinasikan `first` dan `rest`, yang akan membuat ARRAY baru. Contoh:

```
> var e = first(q)
> e
1

> var q = rest(q)
> q
```

[2, 3]

Ketika kita peek atau dequeue pada queue yang kosong, kondisi underflow akan terjadi. Di Singkong, `first` dan `rest` akan mengembalikan null apabila ARRAY kosong.

Contoh:

```
> type(first([]))  
"NULL"
```

```
> type(rest([]))  
"NULL"
```

Apabila kita perlu mengimplementasikan stack ataupun queue dengan ukuran tetap tertentu, gunakanlah fungsi `len` untuk memeriksa kemungkinan operasi yang diijinkan.

## Bekerja dengan Hash

Tipe data ARRAY di Singkong telah dirancang untuk bisa diandalkan apabila kita perlu bekerja dengan daftar, himpunan, ataupun matriks (yang akan kita bahas di bab terpisah). Atau ketika kita perlu bekerja dengan stack atau queue seperti dibahas sebelumnya.

Hanya saja, indeks di ARRAY, seperti yang kita telah ketahui, selalu berupa bilangan bulat (dimulai dari 0). Dan, yang penting, indeks tersebut bisa berubah, misal karena ada elemen yang dihapus. Contoh:

```
> var a = [1,2,3,4,5]
> a[3]
4
```

Saat ini, `a[3]` bernilai 4.

```
> a - 4
[1, 2, 3, 5]
```

Kemudian, kita hapus 4 dari ARRAY. Kebetulan, indeksnya adalah 3.

```
> a[3]
5
```

Bisa kita lihat, elemen dengan nilai 5 bergeser dan menempati posisi indeks 3. Tadinya, indeksnya adalah 4.

---

Apabila kita membutuhkan indeks selain NUMBER dan indeks ini tidak berubah posisi, maka kita bisa menggunakan HASH di Singkong, yang memetakan pasangan dari key ke value, dengan key dan value dapat berupa tipe apa saja di Singkong.

Misal, kita bisa tulis ulang produk kita:

```
var p = [  
  {  
    "code": 4,  
    "name": "Produk D",  
    "price": 4000,  
    "stock": 40  
  },  
  {  
    "code": 3,  
    "name": "Produk C",  
    "price": 3000,  
    "stock": 30  
  },  
  {  
    "code": 2,  
    "name": "Produk B",  
    "price": 2000,  
    "stock": 20  
  },  
  {  
    "code": 1,  
    "name": "Produk A",  
    "price": 1000,  
    "stock": 10  
  }  
]
```

Benar. Kita menambahkan HASH dalam ARRAY, sehingga ARRAY kita menjadi ARRAY dari HASH.

Mari kita ubah fungsi pengurutan kita, untuk mengurutkan harga dari kecil ke besar. Kita simpan dalam `sort_product_hash.singkong` dan berikan nama fungsi `sort_product_hash`:

```

var sort_product_hash = fn(a, k) {
  if (!is_array_and_of(a, "HASH")) {
    return null
  }
  var n = len(a)
  repeat {
    var swapped = false
    var i = 1
    repeat {
      if (a[i-1][k] > a[i][k]) {
        var temp = a[i-1]
        set(a, i-1, a[i])
        set(a, i, temp)
        var swapped = true
      }
      var i = i + 1
      if (i == n) {
        return null
      }
    }
    if (swapped == false) {
      return null
    }
  }
  return a
}

```

Fungsinya mirip, dengan penyesuaian argumen dan pemeriksaan:

```

var sort_product_hash = fn(a, k) {
  if (!is_array_and_of(a, "HASH")) {
    return null
  }
}

```

Dan penyesuaian nama variabel:

```

  if (a[i-1][k] > a[i][k]) {

```

Saat ini, HASH produk kita menggunakan key berupa STRING. Akan tetapi, tipe apa saja bisa digunakan untuk key dan value.

---

Bagaimana kalau kita hapus sesuatu dari HASH?

```
> var h = {"code": 4, "name": "Produk D",  
"price": 4000, "stock": 40}  
> h["stock"]  
40  
  
> h - "stock"  
{"code": 4, "name": "Produk D", "price": 4000}  
  
> h  
{"code": 4, "name": "Produk D", "price": 4000}  
  
> h["stock"]  
> type(h["stock"])  
"NULL"
```

Bisa kita lihat, ketika kita hapus pemetaan dengan key “stock” dari HASH tersebut, karena key tersebut tidak lagi ditemukan, value yang dikembalikan adalah NULL. Tapi, tidak ada yang bergeser sebagaimana pada ARRAY.

Bahkan, ketika kita menggunakan NUMBER sebagai key (dan tentunya, bebas diawali berapa dan dapat berupa bilangan pecahan):

```
> var h = {10:1000, 20:2000, 30:3000, 40:4000,  
50:5000}  
> h[40]  
4000
```



```
> h - 40
{10: 1000, 20: 2000, 30: 3000, 50: 5000}

> type(h[40])
"NULL"
```

Bisa kita lihat, untuk key dengan nilai 40, karena pemetaannya sudah dihapus, valuenya adalah NULL.

---

Yang mungkin juga berguna adalah bahwa HASH di Singkong menjaga insertion-order. Dengan demikian, apabila key “code” ditambahkan lebih dulu, dibandingkan key “name”, maka ketika menampilkan, pasangan dengan key “code” akan selalu tampil lebih dahulu.

## Membuat Hash

Sintaks sebuah HASH adalah:

- Diawali dengan {
- Diikuti daftar pasangan, yang masing-masing dituliskan sebagai <key>:<value>. Apabila daftar ini tidak kosong, maka antara pasangan dipisahkan sebuah koma.
- Diakhiri dengan }

Dengan demikian, berikut adalah sebuah HASH yang tidak memiliki pasangan di dalamnya:

```
var h = {}
```

Bisa kita lihat, panjangnya adalah 0:

```
> len(h)
0
```

Kita bisa menuliskan pemetaan secara langsung, misal seperti contoh produk kita sebelumnya:

```
var h = {  
  "code": 4,  
  "name": "Produk D",  
  "price": 4000,  
  "stock": 40  
}
```

Sejumlah fungsi bawaan bahasa Singkong bekerja dan atau mengembalikan HASH. Bacalah buku referensi Singkong apabila diperlukan.

### Bekerja dengan Key

Pada contoh HASH product sebelumnya, kita bisa menggunakan fungsi `keys` untuk mendapatkan ARRAY semua key. Contoh:

```
> keys(h)  
["code", "name", "price", "stock"]
```

Kita dapat mengakses value berdasarkan key dengan operator `[]`, sama seperti ARRAY. Apabila kita mencoba bekerja dengan key yang tidak ditemukan, maka NULL akan dikembalikan. Tidak ada error yang menyebabkan program dihentikan.

```
> h["code"]  
4
```

```
> h["name"]  
"Produk D"
```

```
> h["price"]
4000

> h["stock"]
40

> h["STOCK"]
> type(h["STOCK"])
"NULL"
```

Perhatikanlah pada contoh terakhir, “stock” tidaklah sama dengan “STOCK”. Walaupun Singkong tidak membedakan huruf besar dan huruf kecil (case-insensitive), ini berlaku pada identifier (misal: nama variabel), dan bukan pada nilai.

Untuk mengubah value berdasarkan key tertentu, gunakanlah fungsi `set`, sama seperti `ARRAY`, hanya dengan key yang valid:

```
> set(h, "stock", 400)
{"code": 4, "name": "Produk D", "price": 4000,
"stock": 400}
```

## Bekerja dengan Value

Untuk mendapatkan semua value dari sebuah `HASH` (sebagai sebuah `ARRAY`), kita bisa menggunakan fungsi `values`.

Masih dengan contoh sebelumnya yang telah diubah, fungsi `values` mengembalikan:

```
> values(h)
[4, "Produk D", 4000, 400]
```

## Iterasi Hash

Iterasi pada HASH dilakukan berdasarkan key. Karena kita bisa mendapatkan semua key (sebagai ARRAY) dengan fungsi `keys`, maka kita cukup iterasi ARRAY ini dan mengakses value dengan operator `[]`. Sebagai contoh:

```
> h
{"code": 4, "name": "Produk D", "price": 4000,
"stock": 400}

> var k = keys(h)
> k
["code", "name", "price", "stock"]

> each(k, fn(e, i) {println(h[e])})
4
Produk D
4000
400
```

## Menambahkan atau Menghapus Pasangan

Untuk menambahkan pasangan key ke value baru, kita bisa menggunakan fungsi `set`. Kita tinggal set key yang belum ada sebelumnya. Contoh untuk key “variant” berikut:

```
> h
{"code": 4, "name": "Produk D", "price": 4000,
"stock": 400}

> set(h, "variant", [])
{"code": 4, "name": "Produk D", "price": 4000,
"stock": 400, "variant": []}
```

Kita juga bisa menggunakan operator + seperti contoh berikut:

```
> h + {"tag": ["sale"]}
{"code": 4, "name": "Produk D", "price": 4000,
"stock": 400, "variant": [], "tag": ["sale"]}
```

Untuk menghapus pasangan berdasarkan key, kita menggunakan operator -. Contoh:

```
> h - "tag"
{"code": 4, "name": "Produk D", "price": 4000,
"stock": 400, "variant": []}
```

Dalam kondisi kita harus mengopi sebuah HASH ke HASH baru, kita bisa menggunakan fungsi `hash_copy` dari modul `util`. Modul ini perlu diload sebelumnya. Perhatikanlah contoh berikut:

```
> load_module("util")
> var a = {"hello": "world"}
> var b = hash_copy(a)
> set(a, "hello", "hello world")
{"hello": "hello world"}

> println(a, b)
{"hello": "hello world"}
{"hello": "world"}
```

## Memeriksa Pasangan

Untuk memeriksa apakah sebuah HASH kosong, kita bisa menggunakan fungsi `empty`, seperti contoh berikut:

```
> empty({})
true
```

```
> empty(h)
false
```

Untuk mengetahui apakah terdapat pasangan dengan key tertentu, kita menggunakan fungsi `in` pada hasil kembalian `keys`. Contoh:

```
> keys(h)
["code", "name", "price", "stock", "variant"]

> in(keys(h), "name")
true
```

Untuk mengetahui apakah terdapat value tertentu, kita menggunakan fungsi `in` pada hasil kembalian `values`. Contoh:

```
> values(h)
[4, "Produk D", 4000, 400, []]

> in(values(h), 4)
true
```

## Membandingkan Hash

Untuk membandingkan HASH, kita menggunakan operator `==` (atau kebalikannya, `!=`). Contoh:

```
> {} == {}
true

> {1:2} == {1:2}
true

> {1:2} == {2:1}
false
```

Pada contoh berikut, 1 dan 1.0 dianggap sama, sehingga perbandingan mengembalikan true:

```
> {1.0:2} == {1:2}
true
```

Akan tetapi, 1 tidaklah sama dengan 1.0001, sehingga perbandingan mengembalikan false:

```
> {1:2, 3:4} == {1.0001:2, 3:4}
false
```

Bisa kita lihat, walau urutan berbeda, kedua HASH berikut dianggap sama:

```
> {1:2, 3:4} == {3:4, 1:2}
true
```

Evaluasi ekspresi akan dilakukan, sehingga kedua HASH berikut juga dianggap sama:

```
> {1:1+1, 3:2+2} == {3:4.0, 1.0:2.0}
true
```

## Nilai Acak

Untuk mendapatkan key acak dari HASH, kita bisa menggunakan fungsi `random`, seperti contoh berikut:

```
> random({1:2, 3:4, 5:6})
5
```

## Hash dan Array

Seperti halnya ARRAY dalam ARRAY, karena key dan value pada HASH bisa berupa tipe apa saja, kita juga bisa membuat pemetaan dengan key atau value berupa HASH. Misal:

```
> { {}: {} }  
{ {}: {} }
```

Akan tetapi, berbeda dengan ARRAY yang dimaksudkan sebagai daftar, terlepas dari apakah setiap elemennya juga berupa ARRAY atau bukan, kita umumnya tidak menggunakan HASH sebagai sebuah daftar.

Walaupun kita tahu bahwa HASH menjaga insertion-order, akan rumit apabila kita mengimplementasikan HASH sebagai daftar, karena harus membuat key sendiri yang unik (dan sekaligus memiliki arti tertentu). Pada ARRAY, ini tentunya otomatis karena indeksnya berupa bilangan bulat (dan mungkin memiliki arti tertentu).

Seperti contoh daftar produk, lebih umum kita menggunakan ARRAY dari HASH untuk daftar, dimana setiap elemennya adalah HASH.

## Konversi Hash ke Array

Apabila kita perlu mengkonversi dari HASH ke ARRAY, kita bisa menggunakan fungsi `array`. Dalam hal ini, ARRAY akan berisi elemen [key, value]. Contoh:

```
> array({1:2, 3:4, 5:6})  
[[1, 2], [3, 4], [5, 6]]
```

Kita telah membahas dasar-dasar bekerja dengan HASH. Untuk informasi selengkapnya, atau fungsi-fungsi lain terkait HASH, bacalah buku referensi Singkong atau tab Help Singkong.jar.



## Set, Matriks, dan Vektor

Singkong tidak menyediakan tipe set (himpunan), matriks, dan vektor. Apabila struktur tersebut diperlukan, kita bisa menggunakan berbagai fungsi yang disertakan dalam modul bawaan `set_util` dan `rect_array_util`. Secara internal, tipe `ARRAY` digunakan.

Di dalam bab ini, kita akan membahas contoh-contoh bekerja dengan set, matriks, dan vektor. Semua contoh berasal dari contoh implementasi dalam buku *Matematika Diskrit Komputasional dengan Bahasa Singkong* (ISBN: 978-602-52770-2-3). Apa yang kita sajikan di sini hanyalah implementasinya. Bacalah juga buku tersebut apabila dasar teorinya diperlukan.

### Set

Untuk bekerja dengan set (struktur tanpa duplikasi, tanpa urutan tertentu), kita perlu load modul `set_util`:

```
> load_module("set_util")
```

### Membuat Set

Untuk membuat set, kita dapat menggunakan fungsi `create_set_from_array`, yang menerima argumen berupa sebuah `ARRAY`, menghilangkan duplikasi elemen, dan mengembalikan sebuah `ARRAY` himpunan.

```
> var c = create_set_from_array([2, 2, 2, 4,
3, 1, 3])
> c
[2, 4, 3, 1]
```

Perhatikanlah bahwa kita bisa memodifikasi ARRAY yang dihasilkan sehingga tidak lagi menjadi set yang valid (misal: menambah elemen duplikat). Dalam hal ini, karena tidak tersedia tipe khusus set (melainkan hanya ARRAY), pemrogram diharapkan untuk memastikan agar tetap merupakan set yang valid.

### Memeriksa Kesamaan Set

Untuk memeriksa apakah dua set sama, kita dapat menggunakan fungsi `is_same_set`, yang akan sekaligus memeriksa apakah merupakan set yang valid. Contoh:

```
> is_same_set([2,2,2,4,3,1,3], [1,2,3,4])
true

> is_same_set([2,2,2,4,3,1,3], [1,2,3,4,5])
false
```

Perhatikanlah bahwa selain tidak adanya duplikasi, urutan tidaklah penting.

### Memeriksa Bagian Dari Set Lain

Untuk memeriksa apakah sebuah set merupakan bagian dari set lain, kita dapat menggunakan fungsi `is_sub_set` seperti contoh berikut:

```
> is_sub_set([2,2,2,4,3,1,3], [1,2,3,4,5])
true

> is_sub_set([1,2,3,4,5], [2,2,2,4,3,1,3])
false
```

### Mendapatkan Irisan

Untuk mendapatkan irisan antar dua set, kita menggunakan fungsi `set_intersection`. Berikut adalah contohnya:

```
> set_intersection([1,2,3,4,5], [2,4,6,8,10])  
[2, 4]
```

```
> set_intersection([1,2,3,4,5], [6,8,10,12])  
[]
```

## Mendapatkan Union

Fungsi `set_union` dapat digunakan untuk mendapatkan union antar dua set. Contoh:

```
> set_union([1,2,3,4,5], [2,4,6,8,10])  
[1, 2, 3, 4, 5, 6, 8, 10]
```

## Selisih Set

Gunakanlah fungsi `set_diff` untuk mendapatkan selisih antar dua set, seperti contoh berikut:

```
> set_diff([1,2,3,4,5], [2,4,6,8,10])  
[1, 3, 5]
```

## Menghitung Produk

Untuk menghitung cartesian product dari dua set, kita dapat menggunakan fungsi `set_product`. Contoh:

```
> set_product([1,2,3], [6,8])  
[[1, 6], [1, 8], [2, 6], [2, 8], [3, 6], [3, 8]]
```

## Menghitung Power Set

Untuk menghitung power set, gunakanlah fungsi `set_power`, seperti contoh berikut:

```
> var s = [1,2,3]
> var p = set_power(s)
> s
[1, 2, 3]

> p
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]

> len(s)
3

> len(p)
8

> set_power(["a", "b", "c"])
[[], ["c"], ["b"], ["b", "c"], ["a"], ["a", "c"], ["a", "b"], ["a", "b", "c"]]
```

## Memeriksa Relasi

Untuk memeriksa apakah set merupakan sebuah relasi, gunakanlah fungsi `is_relation_set`. Contoh:

```
> is_relation_set([1,2,3], [4,6,8], [[1,4], [3,6], [3,8]])
true

> is_relation_set([1,2,3], [4,6,8], [[1,5], [3,6], [3,8]])
false
```

### Memeriksa Relasi Refleksif

Untuk memeriksa relasi refleksif, kita dapat menggunakan fungsi `is_reflexive_relation_set` seperti contoh berikut:

```
> is_reflexive_relation_set([1,2,3], [[1,1],  
[2,2], [3,3]])  
true
```

```
> is_reflexive_relation_set([1,2,3], [[1,1],  
[2,2], [2,3]])  
false
```

### Memeriksa Relasi Simetrik

Untuk memeriksa relasi simetrik, fungsi `is_symmetric_relation_set` dapat digunakan. Contoh:

```
> is_symmetric_relation_set([1,2,3], [[1,1],  
[2,3], [3,2], [2,1], [1,2]])  
true
```

```
> is_symmetric_relation_set([1,2,3], [[1,1],  
[2,3], [3,2], [2,1]])  
false
```

### Memeriksa Relasi Anti-Simetrik

Untuk memeriksa relasi anti-simetrik, gunakanlah fungsi `is_antisymmetric_relation_set` seperti contoh berikut:

```
> is_antisymmetric_relation_set([1,2,3],  
[[1,3], [2,1], [3,2]])  
true
```

```
> is_antisymmetric_relation_set([1,2,3],  
[[1,1], [1,3], [2,1], [3,2]])
```

```
true
```

```
> is_antisymmetric_relation_set([1,2,3],  
[[1,2], [1,3], [2,3], [3,2]])  
false
```

```
> is_antisymmetric_relation_set([1,2,3],  
[[1,1]])  
true
```

```
> is_antisymmetric_relation_set([1,2,3], [])  
true
```

**Contoh relasi refleksif tapi tidak anti-simetrik:**

```
> is_reflexive_relation_set([1,2], [[1,1],  
[1,2], [2,2], [2,1]])  
true
```

```
> is_antisymmetric_relation_set([1,2],  
[[1,1], [1,2], [2,2], [2,1]])  
false
```

**Contoh relasi kosong tidak refleksif tapi anti-simetrik:**

```
> is_reflexive_relation_set([1], [])  
false
```

```
> is_antisymmetric_relation_set([1], [])  
true
```

**Contoh relasi simetrik dan anti-simetrik sekaligus:**

```
> is_symmetric_relation_set([1,2], [[1,1]])  
true
```

```
> is_antisymmetric_relation_set([1,2],
[[1,1]])
true
```

### Memeriksa Relasi Transitif

Untuk memeriksa relasi transitif, kita bisa menggunakan fungsi `is_transitive_relation_set`. Contoh:

```
> is_transitive_relation_set([1,2,3], [[1,3],
[3,1], [1,1], [1,3], [3,3]])
true
```

```
> is_transitive_relation_set([1,2,3], [[1,3],
[3,1], [1,1], [1,3], [3,3], [2,2]])
false
```

```
> is_transitive_relation_set([1,2], [[1,1],
[1,2]])
true
```

Contoh relasi transitif dan simetrik:

```
> is_transitive_relation_set([1,2], [[1,1],
[1,2], [2,1], [2,2]])
true
```

```
> is_symmetric_relation_set([1,2], [[1,1],
[1,2], [2,1], [2,2]])
true
```

### Memeriksa Fungsi

Untuk memeriksa apakah set merupakan sebuah fungsi (dalam matematika, bukan FUNCTION dalam bahasa Singkong), gunakanlah fungsi `is_function_set` seperti contoh berikut:

```

> is_function_set([1,2,3], [1,5], [[1,1],
[2,1], [3,5]], false)
true

> is_function_set([1,2,3], [1,5], [[1,1],
[2,1]], false)
false

> is_function_set([1,2,3], [1,5], [[1,1],
[2,1]], true)
true

> is_function_set([1,2,3], [1,5], [[1,1],
[1,5], [2,1], [3,5]], false)
false

> is_function_set([1,2,3], [1,5], [[1,1],
[1,5], [2,1], [3,5]], true)
false

```

Perhatikanlah bahwa fungsi `is_function_set` mendukung total function ataupun partial function, dengan melewati `true/false` pada parameter keempat. Apabila `false`, maka fungsi ini akan memeriksa dengan aturan total function.

### Memeriksa Fungsi Injektif

Fungsi `is_injective_function_set` dapat digunakan untuk memeriksa apakah merupakan fungsi injektif. Contoh:

```

> is_injective_function_set([1,2,3,4],
[1,5,7,8], [[1,1], [2,8], [3,5], [4,7]],
false)
true

```



```
> is_injective_function_set([1,2,3,4],  
[1,5,7,8], [[1,1], [2,1], [3,5], [4,7]],  
false)  
false
```

Contoh-contoh sebelumnya mengasumsikan total function. Apabila kita perlu bekerja dengan partial function, lewatkanlah true sebagai parameter keempat pada saat pemanggilan fungsi, seperti contoh berikut:

```
> is_injective_function_set([1,2,3,4],  
[1,5,7,8], [[1,1], [2,8], [3,5]], true)  
true
```

### Memeriksa Fungsi Surjektif

Untuk memeriksa apakah merupakan fungsi surjektif, gunakanlah fungsi `is_surjective_function_set` seperti contoh berikut:

```
> is_surjective_function_set([1,2,3,4],  
[1,5,7], [[1,1], [2,1], [3,5], [4,7]], false)  
true
```

```
> is_surjective_function_set([1,2,3,4],  
[1,5,7], [[1,1], [2,1], [3,5], [4,5]], false)  
false
```

Contoh-contoh sebelumnya mengasumsikan total function. Untuk partial function:

```
> is_surjective_function_set([1,2,3,4],  
[1,5,7], [[2,1], [3,5], [4,7]], true)  
true
```

## Memeriksa Fungsi Bijektif

Untuk memeriksa apakah merupakan fungsi bijektif, gunakanlah fungsi `is_bijective_function_set`. Contoh:

```
> is_bijective_function_set([1,2,3], [1,5,7],  
[[1,7], [2,5], [3,1]], false)  
true
```

```
> is_bijective_function_set([1,2,3], [1,5,7],  
[[1,7], [2,5], [3,5]], false)  
false
```

Karena injektif dan surjektif, kita bisa mendapatkan inverse dengan fungsi `inverse_bijective_function_set`, seperti contoh berikut:

```
> inverse_bijective_function_set([1,2,3],  
[1,5,7], [[1,7], [2,5], [3,1]], false)  
[[7, 1], [5, 2], [1, 3]]  
  
>  
println(inverse_bijective_function_set([1,2,3]  
, [1,5,7], [[1,7], [2,5], [3,5]], false))  
null
```

## Matriks dan Vektor

Dalam bahasa Singkong, matriks merupakan rectangular ARRAY, yang mana merupakan ARRAY yang setiap elemennya juga merupakan ARRAY, dan semua elemen tersebut masing-masingnya memiliki jumlah elemen yang sama.

Kita perlu load module bawaan `rect_array_util` terlebih dahulu:

```
> load_module("rect_array_util")
```

## Membuat Rectangular Array

Berikut adalah contoh pembuatan rectangular ARRAY:

```
> var a = [[12.5, 4.0, 15.2], [10.2, 1.2, 14.1], [21.4, 3.3, 12.3]]
```

Perhatikanlah bahwa pada dasarnya, rectangular ARRAY adalah ARRAY. Tidak ada keharusan bahwa ARRAY ini tetap rectangular. Sebagai contoh, kita bisa menambahkan elemen tertentu, yang menjadikan ARRAY tersebut tidak lagi rectangular.

Pada contoh berikut, kita akan mengopikan variabel ARRAY `a` ke `b`, supaya nilai awal `a` tidak berubah (kita module `util` terlebih dahulu untuk menggunakan fungsi `array_copy`):

```
> load_module("util")
> var b = array_copy(a)
> a == b
true
```

Keduanya masih merupakan rectangular ARRAY. Sekarang, mari kita tambahkan elemen ke `b`. Sebuah ARRAY di Singkong bisa menampung berbagai macam tipe, sehingga contoh berikut adalah sepenuhnya valid:

```
> b + [1]
[[12.5000, 4, 15.2000], [10.2000, 1.2000, 14.1000], [21.4000, 3.3000, 12.3000], [1]]
```

```
> b + null
[[12.5000, 4, 15.2000], [10.2000, 1.2000,
14.1000], [21.4000, 3.3000, 12.3000], [1],
null]
```

Sekarang, per definisi, a masih tetap merupakan rectangular ARRAY, dan b sudah bukan lagi merupakan rectangular ARRAY. Namun, keduanya masih ARRAY. Untuk memastikan yang mana yang rectangular dan mana yang bukan, gunakanlah fungsi `is_rect_array`. Contoh:

```
> is_rect_array(a)
true

> is_rect_array(b)
false
```

Lebih lanjut lagi, sebuah ARRAY tetap bisa rectangular, namun bisa berisikan tipe data yang tidak sama. Sebagai contoh, mari kita kopikan ARRAY a ke c:

```
> var c = array_copy(a)
> c + ["Singkong", "Programming", "Language"]
[[12.5000, 4, 15.2000], [10.2000, 1.2000,
14.1000], [21.4000, 3.3000, 12.3000],
["Singkong", "Programming", "Language"]]
```

```
> is_rect_array(c)
true
```

Dalam contoh tersebut, c tetaplah rectangular ARRAY, namun salah satu elemennya adalah ARRAY dari STRING, yang mungkin - untuk pembahasan dalam bab ini atau kebutuhan algoritma tertentu - bukanlah hal yang kita inginkan. Kita ingin misalnya, semua elemen

adalah ARRAY dari NUMBER. Dan, ini bisa kita pastikan dengan fungsi `is_rect_array_of`. Perhatikanlah contoh berikut:

```
> is_rect_array_of(a, "NUMBER")
true

> is_rect_array_of(c, "NUMBER")
false
```

Dengan demikian, kita tidak hanya saja memastikan bahwa sebuah ARRAY adalah rectangular. Kita juga memastikan setiap elemen dari elemennya adalah merupakan tipe tertentu (dalam hal ini, NUMBER).

Sekarang, kita bisa yakin untuk memroses ARRAY, setelah kita mengetahui bahwa ARRAY tersebut rectangular dan setiap elemennya adalah ARRAY tipe tertentu. Sama seperti contoh set, karena tidak terdapat tipe khusus matriks di Singkong, pemrogram memastikan – misal dengan berbagai fungsi – agar tetap merupakan rectangular ARRAY yang valid.

### Mendapatkan Ukuran

Untuk mendapatkan ukuran rectangular ARRAY, kita bisa gunakan fungsi `rect_array_size` dan `rect_array_size_of`. Contoh (mengembalikan [baris, kolom] apabila berhasil):

```
> rect_array_size(a)
[3, 3]

> rect_array_size(c)
[4, 3]

> rect_array_size_of(a, "NUMBER")
[3, 3]
```

```
> println(rect_array_size_of(c, "NUMBER"))
null
```

Selain ukuran, apabila kita perlu membandingkan rectangular ARRAY, gunakanlah operator `==` atau fungsi `array_equals` yang telah kita bahas sebelumnya.

## Vektor

Vektor adalah bentuk khusus dari matriks, yang hanya memiliki satu baris atau kolom. Dengan demikian, kita juga implementasikan dengan rectangular ARRAY.

Untuk rectangular ARRAY yang berisikan satu baris atau kolom, pastikanlah pertama-tama, bahwa ARRAY yang dimaksud adalah sebuah rectangular ARRAY. Jadi, misal pada contoh vektor kolom dengan dimensi 3x1 berikut, kita perlu menuliskannya sebagai:

```
> var b = [ [1], [4], [10] ]
> rect_array_size_of(b, "NUMBER")
[3, 1]
```

Ingatlah, kita **tidak** menuliskannya sebagai:

```
> var x = [1, 4, 10]
```

Karena, `x` tersebut bukanlah rectangular ARRAY di Singkong:

```
> println(rect_array_size_of(x, "NUMBER"))
null
```

Kita juga dapat menggunakan fungsi bantu `is_rect_array_column_of` dan `is_rect_array_column_of_number`, seperti contoh berikut:

```
> println(is_rect_array_column_of_number([ 1,
4, 10 ] ))
null
```

```
> is_rect_array_column_of_number([ [1], [4],
[10] ] )
true
```

```
> is_rect_array_column_of_number([ [1,1],
[4,4], [10,10] ] )
false
```

Untuk vektor baris, berikut adalah contohnya:

```
> var c = [ [5, 13, 8, 7] ]
> rect_array_size_of(c, "NUMBER")
[1, 4]
```

Sekali lagi, kita tidak menuliskannya sebagai:

```
> var x = [5, 13, 8, 7]
> println(rect_array_size_of(x, "NUMBER"))
null
```

Kita juga dapat menggunakan fungsi bantu `is_rect_array_row_of` dan `is_rect_array_row_of_number` seperti contoh berikut:

```
> println(is_rect_array_row_of_number([ 5, 13,
8, 7 ] ))
null
```

```

> is_rect_array_row_of_number([ [5, 13, 8, 7]
] )
true

> is_rect_array_row_of_number([ [5, 13, 8, 7],
[5, 13, 8, 7] ] )
false

```

### Matriks Persegi

Untuk rectangular ARRAY persegi di Singkong, dimana jumlah baris dan kolom adalah sama, kita cukup memeriksa hasil kembalian dari pemanggilan fungsi `rect_array_size` atau `rect_array_size_of`. Perhatikanlah contoh berikut:

```

> var a = [ [1,2], [2,1] ]
> var s = rect_array_size_of(a, "NUMBER")
> s
[2, 2]

```

Lebih mudah lagi, kita cukup menggunakan fungsi `is_square_rect_array_of` atau `is_square_rect_array_of_number`, seperti contoh berikut:

```

> is_square_rect_array_of_number([ [1,2],
[2,1] ])
true

> is_square_rect_array_of_number([ [1,2],
[2,1], [3,4] ])
false

```



```
> println(is_square_rect_array_of_number([
[0], [1,2] ]))
null
```

### Matriks Diagonal

Untuk memeriksa apakah suatu rectangular ARRAY merupakan matriks diagonal, kita bisa menggunakan fungsi `is_diagonal_rect_array_of_number`. Fungsi tersebut pada dasarnya melakukan perulangan untuk setiap elemen dalam rectangular ARRAY dan memastikan definisi matriks diagonal terpenuhi. Tentunya, sebelumnya rectangular ARRAY tersebut harus merupakan matriks persegi.

Contoh:

```
> is_diagonal_rect_array_of_number([
[2,0,0,0], [0,5,0,0], [0,0,3,0], [0,0,0,-4] ])
true
```

```
> is_diagonal_rect_array_of_number([
[1,2,3,4], [0,5,0,0], [0,0,3,0], [0,0,0,-4] ])
false
```

### Matriks Identitas

Untuk memeriksa apakah sebuah rectangular ARRAY merupakan matriks identitas, kita dapat menggunakan fungsi `is_constant_rect_array_of` atau `is_constant_rect_array_of_number`. Cara kerja fungsi tersebut mirip dengan pemeriksaan apakah rectangular ARRAY tersebut merupakan matriks diagonal. Akan tetapi, nilai untuk elemen pada kolom dan baris yang sama, adalah sama dengan 1.

Contoh:

```
> is_constant_rect_array_of_number([  
  [1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1] ],  
  1)  
true
```

```
> is_constant_rect_array_of_number([  
  [1,2,3,4], [0,1,0,0], [0,0,1,0], [0,0,0,1] ],  
  1)  
false
```

Atau, kita dapat menggunakan fungsi `is_identity_rect_array_of_number`, seperti contoh berikut:

```
> is_identity_rect_array_of_number([  
  [1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1] ])  
true
```

### Matriks Konstanta

Untuk memeriksa apakah sebuah rectangular ARRAY merupakan matriks konstanta, kita dapat gunakan cara yang sama seperti memeriksa apakah merupakan matriks identitas. Bedanya, nilai 1 yang dilewatkan dapat diganti dengan nilai lain. Berikut adalah contohnya:

```
> is_constant_rect_array_of_number([  
  [1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1] ],  
  5)  
false
```

```
> is_constant_rect_array_of_number([
[5,0,0,0], [0,5,0,0], [0,0,5,0], [0,0,0,5] ],
5)
true
```

Contoh pertama tentu akan mengembalikan false, karena walaupun merupakan matriks konstanta, nilai untuk elemen pada baris dan kolom yang sama adalah 1 dan bukan 5 sebagaimana diminta. Contoh kedua akan mengembalikan true.

### Matriks Segitiga Atas

Untuk memeriksa apakah sebuah rectangular ARRAY merupakan matriks segitiga atas, kita dapat menggunakan fungsi `is_upper_triangular_rect_array_of` atau `is_upper_triangular_rect_array_of_number` sebagai berikut.

```
> is_upper_triangular_rect_array_of_number([
[1,3,4], [0,2,1], [0,0,-5] ])
true
```

```
> is_upper_triangular_rect_array_of_number([
[1,0,0], [4,2,0], [5,-1,-5] ])
false
```

### Matriks Segitiga Bawah

Untuk memeriksa apakah sebuah rectangular ARRAY merupakan matriks segitiga bawah, kita dapat menggunakan fungsi `is_lower_triangular_rect_array_of` atau `is_lower_triangular_rect_array_of_number` sebagai berikut.

```
> is_lower_triangular_rect_array_of_number([
[1,3,4], [0,2,1], [0,0,-5] ])
false

> is_lower_triangular_rect_array_of_number([
[1,0,0], [4,2,0], [5,-1,-5] ])
true
```

### Matriks Nol

Untuk memeriksa apakah sebuah rectangular ARRAY merupakan matriks nol, kita dapat menggunakan fungsi `is_zero_rect_array_of` atau `is_zero_rect_array_of_number` seperti contoh berikut:

```
> is_zero_rect_array_of_number([ [0] ])
true

> is_zero_rect_array_of_number([ [0,0], [0,0]
])
true

> is_zero_rect_array_of_number([ [0,0,0,0],
[0,0,0,0], [0,0,0,0], [0,0,0,0] ])
true

> is_zero_rect_array_of_number([ [8,0,0,0],
[0,0,0,0], [0,0,0,0], [0,0,0,0] ])
false
```

### Matriks Simetrik

Untuk memeriksa apakah sebuah rectangular ARRAY merupakan matriks simetrik, kita dapat menggunakan fungsi `is_symmetric_rect_array_of` atau

`is_symmetric_rect_array_of_number` seperti contoh berikut:

```
> is_symmetric_rect_array_of_number([ [1,4,5],  
[4,2,-1], [5,-1,-5] ])  
true
```

```
> is_symmetric_rect_array_of_number([ [1,4,5],  
[4,2,-1], [5,-1,5] ])  
true
```

```
> is_symmetric_rect_array_of_number([ [1,4,5],  
[4,2,-1], [-5,-1,-5] ])  
false
```

### Penjumlahan Matriks

Untuk menjumlahkan dua rectangular ARRAY, kita dapat menggunakan fungsi `add_rect_array_of` atau `add_rect_array_of_number`. Di dalam fungsi tersebut, telah diperiksa apakah kedua rectangular ARRAY memang dapat ditambahkan. Apabila ya, hasil penjumlahan akan disimpan pada rectangular ARRAY baru (tidak memodifikasi kedua operan).

Berikut adalah contoh penjumlahan yang gagal:

```
> println(add_rect_array_of_number([ [7,8],  
[1,2,3], [6,5,4]], [[1,0,2], [-3,4,0], [2,-  
5,7]] ))  
null
```

Penjumlahan tersebut gagal (mengembalikan null) karena argumen pertama dalam fungsi bukanlah rectangular ARRAY.

Contoh berikut juga merupakan penjumlahan yang gagal (mengembalikan null), karena argumen kedua dalam fungsi bukanlah rectangular ARRAY:

```
> println(add_rect_array_of_number( [[7,8,9],  
[1,2,3], [6,5,4]], [[1,0,2], [-3,4,0], [2,-5]]  
))  
null
```

Contoh berikut juga gagal (mengembalikan null), karena walaupun keduanya adalah rectangular ARRAY, ukurannya berbeda:

```
> println(add_rect_array_of_number( [[7,8],  
[1,2]], [[1,0,2], [-3,4,0], [2,-5,7]] ))  
null
```

Berikut adalah contoh penjumlahan yang berhasil:

```
> add_rect_array_of_number( [[7,8,9],  
[1,2,3], [6,5,4]], [[1,0,2], [-3,4,0], [2,-  
5,7]] )  
[[8, 8, 11], [-2, 6, 3], [8, 0, 11]]
```

## Pengurangan Matriks

Untuk pengurangan dua rectangular ARRAY, kita dapat menggunakan fungsi `sub_rect_array_of` atau `sub_rect_array_of_number`. Di dalam fungsi tersebut, telah diperiksa apakah pengurangan dimungkinkan antara kedua rectangular ARRAY. Apabila ya, hasil pengurangan akan disimpan pada rectangular ARRAY baru (tidak memodifikasi kedua operan).

Sama seperti pada contoh penjumlahan sebelumnya, berikut adalah contoh-contoh yang gagal (mengembalikan null):

```
> println(sub_rect_array_of_number( [[7,8],
[1,2,3], [6,5,4]], [[1,0,2], [-3,4,0], [2,-
5,7]] ))
null

> println(sub_rect_array_of_number( [[7,8,9],
[1,2,3], [6,5,4]], [[1,0,2], [-3,4,0], [2,-5]]
))
null

> println(sub_rect_array_of_number( [[7,8],
[1,2]], [[1,0,2], [-3,4,0], [2,-5,7]] ))
null
```

Dan, berikut adalah contoh yang berhasil:

```
> sub_rect_array_of_number( [[7,8,9],
[1,2,3], [6,5,4]], [[1,0,2], [-3,4,0], [2,-
5,7]] )
[[6, 8, 7], [4, -2, 3], [4, 10, -3]]
```

## Perkalian Matriks

Untuk perkalian dua rectangular ARRAY, kita dapat menggunakan fungsi `mul_rect_array_of` atau `mul_rect_array_of_number`. Di dalam fungsi tersebut, telah diperiksa apakah perkalian dimungkinkan antara kedua rectangular ARRAY. Apabila ya, hasil perkalian akan disimpan pada rectangular ARRAY baru.

Pada contoh berikut, perkalian tidak dapat dilakukan (mengembalikan null) karena kolom rectangular ARRAY pertama tidak sama dengan baris pada rectangular kedua:

```

> println(mul_rect_array_of_number( [[1,4,5],
[4,2,-1], [5,-1,-5]], [[0,3,1], [1,-1,0]] ))
null

> rect_array_size_of([[1,4,5], [4,2,-1], [5,-
1,-5]], "NUMBER")
[3, 3]

> rect_array_size_of([[0,3,1], [1,-1,0]],
"NUMBER")
[2, 3]

```

Dalam hal ini, kolom (3) (pada [3,3]) tidak sama dengan baris (2) (pada [2,3]).

Berikut adalah contoh-contoh yang berhasil:

```

> mul_rect_array_of_number( [[1,4,5], [4,2,-
1], [5,-1,-5]], [[0,3,1], [1,-1,0], [2,0,2]] )
[[14, -1, 11], [0, 10, 2], [-11, 16, -5]]

> mul_rect_array_of_number( [[0,3,1], [1,-
1,0], [2,0,2]], [[1,4,5], [4,2,-1], [5,-1,-5]]
)
[[17, 5, -8], [-3, 2, 6], [12, 6, 0]]

```

## Perkalian Silang Vektor

Untuk perkalian silang (cross product) berdimensi 1x3 (3D) pada rectangular ARRAY vektor baris, kita dapat menggunakan fungsi `cross_product_3d_rect_array_row_of` atau `cross_product_3d_rect_array_row_of_number`, seperti contoh berikut:



```
> cross_product_3d_rect_array_row_of_number(
[[1, 2, 3]], [[4, 5, 6]])
[[-3, 6, -3]]

> rect_array_size_of(
cross_product_3d_rect_array_row_of_number([[1,
2, 3]], [[4, 5, 6]]), "NUMBER")
[1, 3]
```

Sementara, untuk cross product 3D pada rectangular ARRAY vektor kolom, kita dapat menggunakan fungsi `cross_product_3d_rect_array_column_of` atau `cross_product_3d_rect_array_column_of_number`, seperti contoh berikut:

```
>
cross_product_3d_rect_array_column_of_number(
[[1], [2], [3]], [[4], [5], [6]])
[[-3], [6], [-3]]

> rect_array_size_of(
cross_product_3d_rect_array_column_of_number([
[1], [2], [3]], [[4], [5], [6]]), "NUMBER")
[3, 1]
```

Perhatikanlah bahwa ini hanya berlaku pada 3D (1x3 atau 3x1). Contoh-contoh berikut akan mengembalikan null:

```
> println(
cross_product_3d_rect_array_row_of_number([[1,
2, 3, 4]], [[5, 6, 7, 8]]))
null
```

```
> println(
cross_product_3d_rect_array_column_of_number([
[1], [2], [3], [4]], [[5], [6], [7], [8]]))
null
```

### Perkalian Skalar

Untuk perkalian skalar dengan rectangular ARRAY, kita dapat menggunakan fungsi `mul_scalar_rect_array_of` atau `mul_scalar_rect_array_of_number`, seperti contoh berikut:

```
> mul_scalar_rect_array_of_number(2, [[1,4,5],
[4,2,-1], [5,-1,-5]])
[[2, 8, 10], [8, 4, -2], [10, -2, -10]]
```

### Trace Matriks Persegi

Untuk mendapatkan trace rectangular ARRAY persegi, kita dapat menggunakan fungsi `trace_rect_array_of` atau `trace_rect_array_of_number`. Berikut adalah contoh penggunaan fungsi:

```
> trace_rect_array_of_number([[7,0,-1],
[3,5,2], [0,4,3]])
15
```

Pastikanlah bahwa rectangular ARRAY yang dilewatkan adalah rectangular ARRAY persegi. Contoh berikut akan mengembalikan null:

```
> println(trace_rect_array_of_number([[7,0,-
1], [3,5,2]]))
null
```

## Transpose Matriks

Untuk mendapatkan transpose rectangular ARRAY, kita dapat menggunakan fungsi `transpose_rect_array_of` atau `transpose_rect_array_of_number`, seperti contoh berikut:

```
> transpose_rect_array_of_number([[1,2,-3],  
[4,2,-1], [5,-1,-5]])  
[[1, 4, 5], [2, 2, -1], [-3, -1, -5]]
```

Contoh berikut menghitung transpose dari transpose rectangular ARRAY. Keluarannya adalah sama seperti rectangular ARRAY awal:

```
> transpose_rect_array_of_number(  
transpose_rect_array_of_number([[1,2,-3],  
[4,2,-1], [5,-1,-5]]))  
[[1, 2, -3], [4, 2, -1], [5, -1, -5]]
```

## Determinan Matriks

Untuk menghitung determinan rectangular ARRAY persegi, kita dapat mempergunakan fungsi `determinant_rect_array_of` atau `determinant_rect_array_of_number`, seperti contoh berikut:

```
> determinant_rect_array_of_number([[6,1,1],  
[4,-2,5], [2,8,7]])  
-306
```

Apabila diterapkan pada rectangular ARRAY persegi berukuran 1x1, maka hasilnya adalah bilangan di dalam rectangular ARRAY tersebut:

```
> determinant_rect_array_of_number([[1]])  
1
```

## Inverse Matriks

Untuk mendapatkan inverse dari sebuah rectangular ARRAY, kita dapat menggunakan fungsi `inverse_rect_array_of` ataupun `inverse_rect_array_of_number`, seperti dicontohkan berikut:

```
> inverse_rect_array_of_number([[0,-3,-2],  
[1,-4,-2], [-3,4,1]])  
[[4, -5, -2], [5, -6, -2], [-8, 9, 3]]
```

Apabila dikalikan antara rectangular ARRAY dan kebalikannya, ataupun antara kebalikannya dan rectangular ARRAY tersebut, maka akan menghasilkan rectangular ARRAY identitas, seperti contoh berikut:

```
> mul_rect_array_of_number([[0,-3,-2], [1,-  
4,-2], [-3,4,1]],  
inverse_rect_array_of_number([[0,-3,-2], [1,-  
4,-2], [-3,4,1]]))  
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

```
> mul_rect_array_of_number(  
inverse_rect_array_of_number([[0,-3,-2], [1,-  
4,-2], [-3,4,1]]), [[0,-3,-2], [1,-4,-2], [-  
3,4,1]])  
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Tentu saja, sebuah rectangular ARRAY tidak akan memiliki kebalikan apabila bukan rectangular ARRAY persegi, atau determinannya adalah 0, yang mana fungsi akan mengembalikan null atau false (determinan=0), seperti contoh berikut:

```
> inverse_rect_array_of_number([[1,1], [2,2]])  
false
```

## Instalasi Java dan Menjalankan Singkong.jar

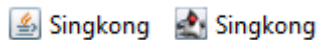
*Apabila diperlukan panduan teknis untuk instalasi Java, tutorial berikut membahas cara instalasi untuk sistem operasi Windows, macOS, Linux (Ubuntu), dan Chrome OS (dengan Linux development environment). Panduan ini dituliskan ulang berdasarkan konten dalam buku gratis: Mengenal dan Menggunakan Bahasa Pemrograman Singkong (ISBN: 978-602-52770-1-6, Dr. Noprianto, 2020-2024, diterbitkan oleh PT. Stabil Standar Sinergi).*

---

### Windows

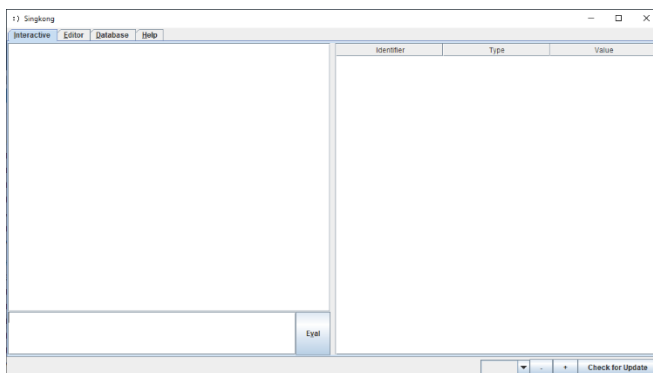
*Panduan ini ditulis berdasarkan Windows 10. Sesuaikanlah apabila diperlukan.*

Apabila icon Singkong.jar terlihat seperti salah satu dari gambar berikut:



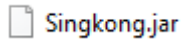
Maka, ada kemungkinan Java telah terinstal dan file .jar telah diasosiasikan untuk dibuka dengan Java.

Cobalah klik ganda pada Singkong.jar, atau klik kanan dan pilih Open. Apabila yang tampil adalah seperti gambar berikut, maka Java telah terinstal dan Singkong.jar dapat dijalankan.

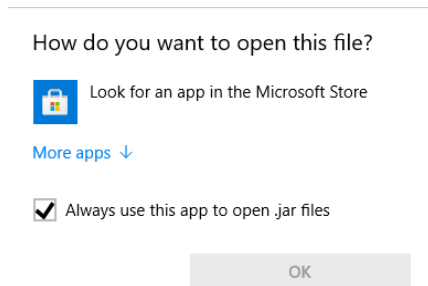


Apabila Singkong.jar ternyata dibuka dengan aplikasi lain (tampilan tidak sama dengan gambar), cobalah untuk klik kanan pada Singkong.jar dan pilih Open with atau Open with.... Dari menu atau pilihan yang ditampilkan, pilihlah yang mengandung kata Java atau OpenJDK. Apabila pilihan menu tersebut tidak tersedia, maka kemungkinan besar Java belum terinstal.

Apabila icon Singkong.jar terlihat seperti gambar berikut:

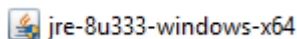


Atau, ketika diklik ganda, yang tampil adalah sebagai berikut:

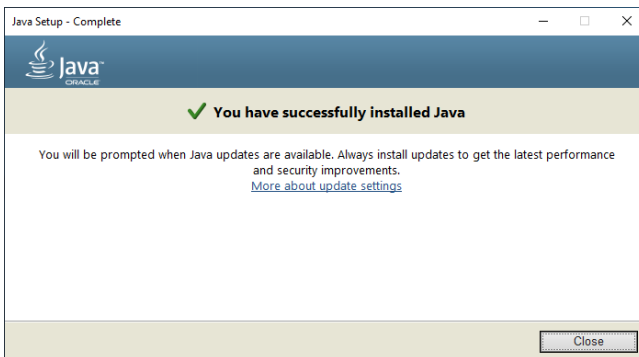
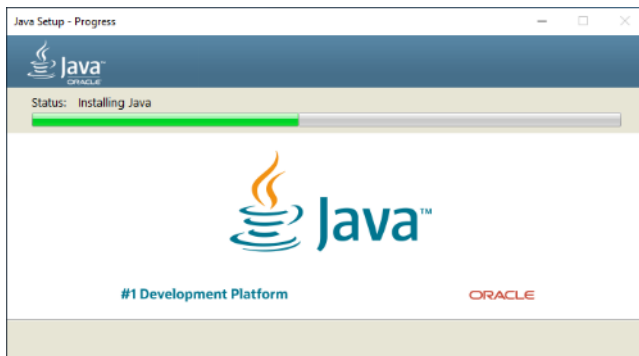
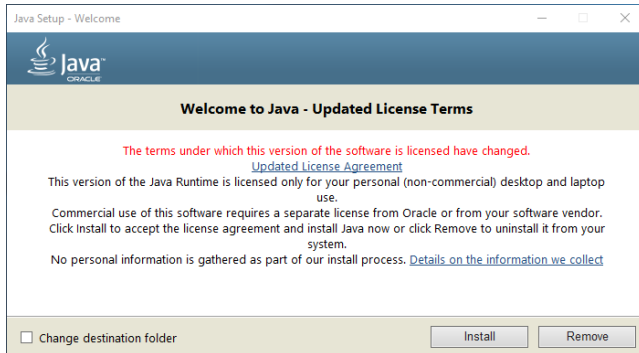


Maka, kemungkinan besar Java belum terinstal.

Untuk menginstal Java Runtime Environment, downloadlah dari java.com. Contoh file hasil download adalah seperti gambar berikut:



Jalankan file tersebut dan ikutilah langkah-langkah instalasi sebagaimana ditampilkan berikut (diawali klik pada tombol Install):



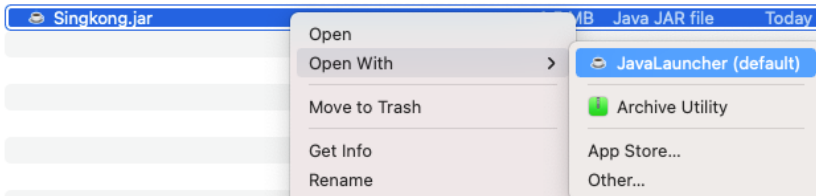
## macOS

*Panduan ini ditulis berdasarkan macOS 11. Sesuaikanlah apabila diperlukan.*

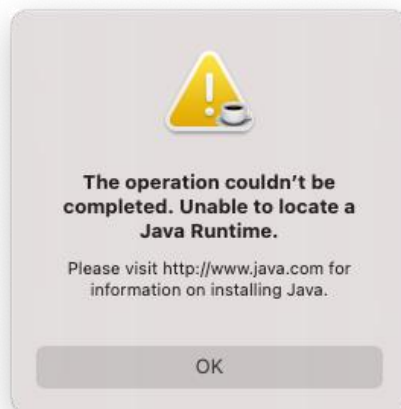
File Singkong.jar akan tampak seperti ini pada Finder:

 Singkong.jar

Cobalah control klik (tekan dan tahan tombol control sambil klik mouse atau trackpad) pada file tersebut. Gambar serupa berikut akan ditampilkan:

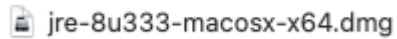


Pada menu yang tampil, pilihlah Open With, kemudian JavaLancer (default). Apabila yang tampil adalah pesan berikut:

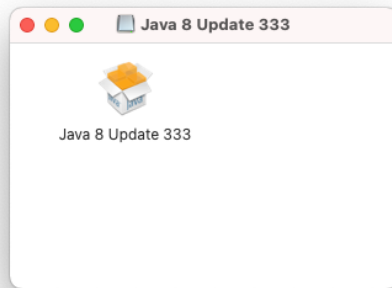




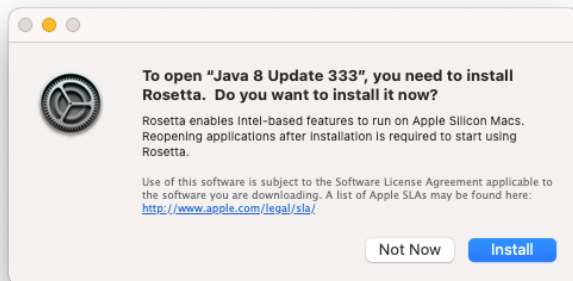
Maka, Java belum terinstal. Downloadlah dari java.com. Berikut adalah contoh file hasil download:



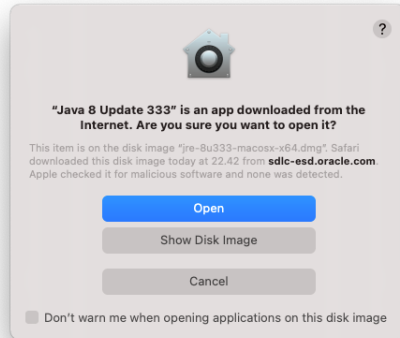
Klik gandalah untuk membuka disk image tersebut. Setelah terbuka, berikut adalah isinya:



Klik gandalah pada file installer tersebut. Apabila Mac yang digunakan menggunakan Apple Silicon, maka dialog konfirmasi untuk melakukan instalasi Rosetta mungkin akan ditampilkan:



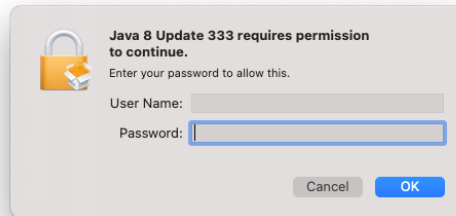
Apabila demikian, maka kliklah pada tombol Install dan tungguilah sampai selesai. Kemudian, jalankanlah kembali file installer sebelumnya. Dialog konfirmasi serupa berikut mungkin akan ditampilkan:



Kliklah tombol Open. Dialog instalasi berikut akan ditampilkan:



Kliklah tombol Install. Kemudian, masukkanlah password untuk pengguna yang memiliki hak instalasi software dan klik OK pada dialog yang tampil berikut:

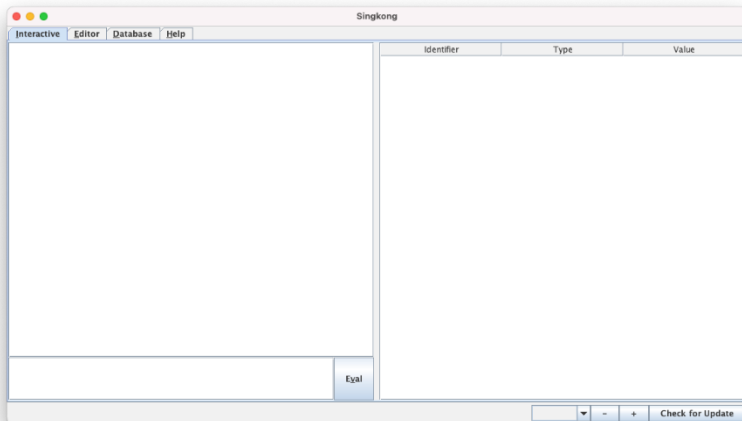


Setelah itu, tungguilah proses instalasi Java sampai selesai.





Cobalah untuk klik ganda file Singkong.jar. Apabila yang tampil adalah seperti gambar berikut, maka Java telah terinstal dan Singkong.jar dapat dijalankan.



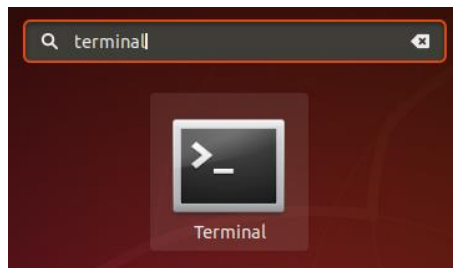
## Linux

*Panduan ini ditulis berdasarkan Ubuntu Linux 18.04, namun perintah command line yang digunakan dapat diterapkan pada Ubuntu versi lebih baru (misal 22.04). Sesuaikanlah apabila diperlukan.*

Kliklah tombol pada bagian bawah layar seperti contoh berikut:



Kemudian, ketikkanlah terminal pada layar pencarian aplikasi berikut:



Jalankan Terminal. Kemudian, ketikkanlah java (diikuti penekanan tombol Enter) pada terminal emulator yang ditampilkan, seperti contoh berikut:

```
$ java
Command 'java' not found
```

Apabila terdapat pesan not found seperti pada gambar, maka kemungkinan besar Java belum terinstal. Masih di layar terminal emulator tersebut, apabila pengguna yang login memiliki hak sudo (apabila tidak, gunakanlah akun pengguna lain yang memilikinya), jalankanlah perintah berikut (tanpa mengetikkan \$, diikuti Enter):

```
$ sudo apt-get update
```

Masukkanlah password pengguna yang sedang login tersebut apabila diminta:

```
[sudo] password for
```

Tunggulah sampai proses update berhasil. Setelah itu, jalankanlah perintah berikut untuk instalasi Java:

```
$ sudo apt-get install default-jre
```

Masukkanlah password pengguna yang sedang login apabila kembali diminta. Setelah itu, konfirmasilah untuk melakukan instalasi sejumlah paket yang tampil. Kemudian, tungguilah sampai selesai.

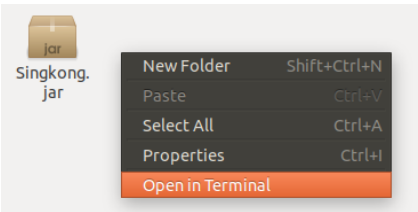
File Singkong.jar akan tampak seperti ini pada file manager:



Cobalah klik ganda pada file tersebut. Sebuah dialog dengan cuplikan pesan kesalahan berikut mungkin akan ditampilkan:

```
Singkong.jar  
is not  
marked as  
executable.
```

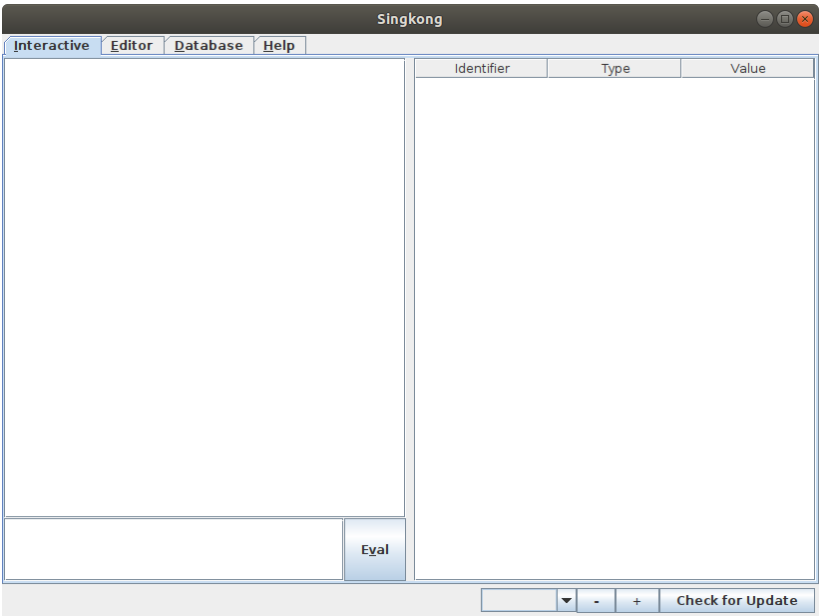
Pada area kosong daftar file di luar file Singkong.jar, klik kananlah dan jalankan Terminal, seperti contoh berikut:



Pada layar terminal emulator yang tampil, berikanlah perintah berikut:

```
$ chmod +x Singkong.jar
```

Setelah itu, kembalilah untuk klik ganda file Singkong.jar. Apabila yang tampil adalah seperti gambar berikut, maka Java telah terinstal dan Singkong.jar dapat dijalankan.



## Chrome OS

*Panduan ini ditulis berdasarkan Chrome OS versi 101 dengan Linux development environment telah didukung secara default. Sesuaikanlah apabila diperlukan.*

Bukalah Settings (pada bagian bawah layar) dan pilihlah pengaturan Advanced seperti berikut:

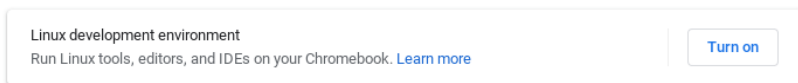
Advanced ▼

Kemudian, kliklah pada Developers:

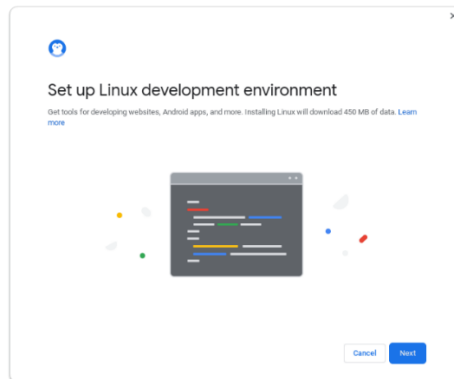
<> Developers

Pada bagian Linux development environment, kliklah tombol Turn on.

Developers

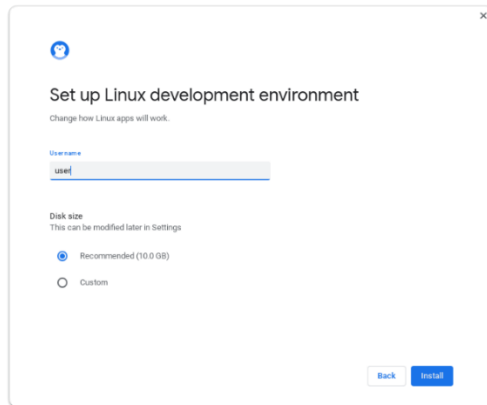


Dialog berikut akan ditampilkan. Kliklah tombol Next:

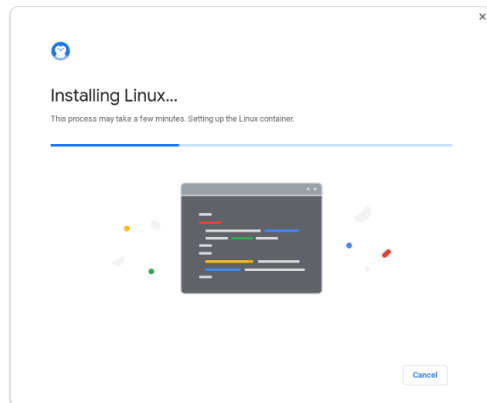




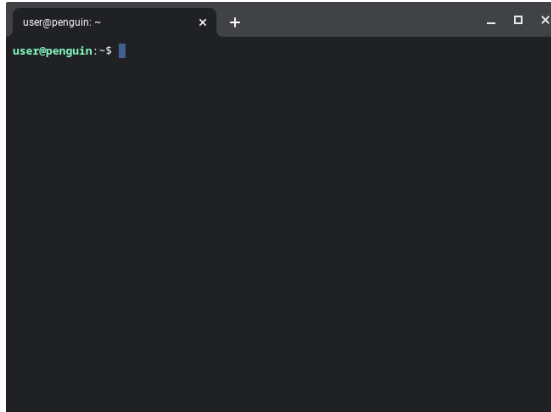
Kemudian, masukkanlah nama pengguna pada dialog yang tampil setelahnya dan kliklah tombol Install:



Setelah itu, proses download, instalasi, dan konfigurasi akan dilakukan. Tunggulah sampai selesai.



Layar terminal emulator akan ditampilkan setelah semua proses tersebut selesai. Berikut adalah contoh layarnya:



Berikanlah perintah berikut (tanpa \$, diikuti penekanan tombol Enter):

```
$ sudo apt-get update
```

Tunggulah sampai proses ini selesai. Kemudian, berikanlah perintah berikut untuk instalasi Java:

```
$ sudo apt-get install default-jre
```

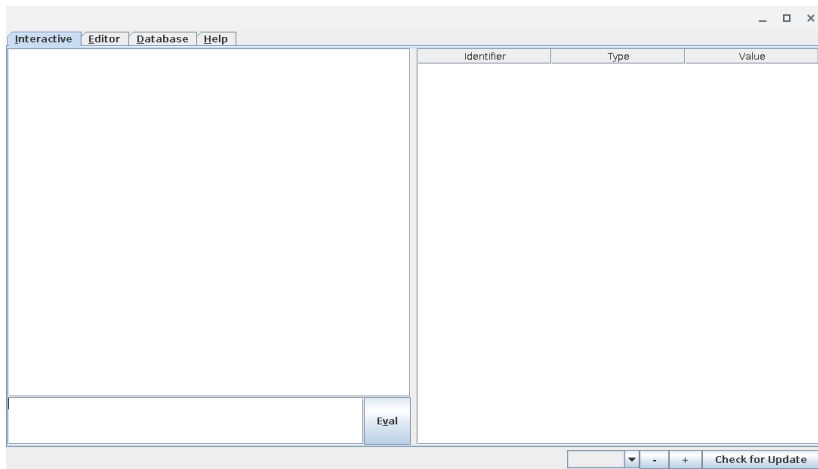
Tunggulah juga sampai selesai. Setelah itu, downloadlah Singkong.jar dengan perintah berikut:

```
$ wget https://nopri.github.io/Singkong.jar
```

Pada akhirnya, Singkong.jar dapat dijalankan dengan perintah berikut:

```
$ java -jar Singkong.jar
```

Apabila yang tampil adalah seperti gambar berikut, maka Java telah terinstal pada Linux development environment dan Singkong.jar dapat dijalankan.



### Tambahan: JRE dan JDK

- Singkong.jar hanya membutuhkan Java Runtime Environment (JRE). Dengan demikian, program yang ditulis dengan bahasa pemrograman Singkong pun hanya akan membutuhkan Java Runtime Environment dan Singkong.jar.
- Dengan demikian, instalasi Java Development Kit tidak diperlukan. Java Development Kit berisikan semua yang diperlukan untuk mengembangkan dan menjalankan program yang ditulis dengan bahasa Java.
- Pembahasan instalasi Java sebelumnya mengasumsikan instalasi Java Runtime Environment. Akan tetapi, apabila instalasi Java Development Kit yang dilakukan, pun tidak akan menjadi kendala teknis, karena semua yang diperlukan untuk menjalankan Singkong.jar pun tersedia dalam instalasi tersebut.
- Instalasi Java Development Kit dapat memberikan manfaat tambahan, seperti:

- Tersedianya program `jar`, yang dapat digunakan untuk bekerja dengan arsip `.jar`. Sebagai contoh, menambahkan file ke dalam arsip `.jar`. Ini berguna, sebagai contoh, ketika program yang ditulis dengan bahasa Singkong ingin didistribusikan sebagai file `.jar` tunggal (dibundel bersama `Singkong.jar` dan semua file yang diperlukan), yang dapat dijalankan pada komputer tujuan.
  - Tanpa program `jar` tersebut, penambahan file ke arsip `.jar` tetap dapat dilakukan dengan berbagai cara lain, seperti memanfaatkan fitur yang telah disediakan oleh sistem operasi, ataupun program yang dapat menambahkan file ke format `.zip`.
- Tersedianya program `javac`, yang dapat digunakan untuk melakukan kompilasi program yang ditulis dengan bahasa Java.
  - Program yang ditulis dengan bahasa Singkong dapat memanggil method Java, yang mungkin menyediakan fungsionalitas tambahan (di luar yang telah disediakan oleh bahasa Singkong).
  - Interpreter bahasa Singkong pun dapat diembed ke dalam program yang ditulis dengan bahasa Java.

### Tambahan: distribusi Java alternatif

Pembahasan instalasi Java sebelumnya mengasumsikan Java Runtime Environment yang didownload dari [java.com](http://java.com) (Windows dan macOS) ataupun yang disediakan oleh distribusi sistem operasi (Linux dan Linux development environment pada Chrome OS).

Apabila untuk berbagai alasan lain (termasuk alasan non-teknis) diperlukan distribusi Java alternatif, contoh yang dapat digunakan adalah Java Runtime Environment dan/atau Java Development Kit dari [adoptium.net](https://adoptium.net).

Di website tersebut tersedia Eclipse Temurin, yang merupakan distribusi OpenJDK dari Adoptium.

Instalasi distribusi Java alternatif tidak dibahas dalam buku ini.

*Halaman ini sengaja dikosongkan*

## Daftar Pustaka

Noprianto., 2024, Mengenal dan Menggunakan Bahasa Pemrograman Singkong, PT. Stabil Standar Sinergi.

Noprianto and Heryadi, Y., 2021, Matematika Diskrit Komputasional dengan Bahasa Singkong, PT. Stabil Standar Sinergi.

Buku ini diawali dengan contoh mengurutkan daftar bilangan (37 halaman) dan contoh algoritma greedy (8 halaman). Pembahasan dimulai dari memahami permasalahan, yang dilanjutkan secara bertahap ke algoritma, struktur data, dan pembuatan program dalam bahasa pemrograman Singkong.

Pembahasan dilakukan secara mendetil dengan tujuan agar buku ini juga dapat digunakan sebagai pengantar bagi yang tertarik memulai pemrograman komputer.

Sebagai topik lanjutan, disertakan juga contoh-contoh bagaimana bekerja dengan array, stack, queue, hash, set, matriks, dan vektor di Singkong.



Dr. Noprianto mengembangkan bahasa pemrograman Singkong dan interpretnya sejak akhir 2019.

Beliau menyukai pemrograman, dan mendirikan serta mengelola perusahaan pengembangan software dan teknologi Singkong.dev (PT. Stabil Standar Sinergi).

Noprianto menyelesaikan pendidikan doktor ilmu komputer dan telah menulis beberapa buku pemrograman (termasuk Python, Java, dan Singkong).

Buku dan softwarenya dapat didownload dari <https://nopri.github.io>



Budiman memulai karir pengembangannya dari mempelajari dasar-dasar pemrograman komputer pada tahun 2020.

Untuk ke depannya, beliau berharap dapat tetap mengembangkan diri secara positif seperti berkontribusi dalam penulisan buku ini.

**singkong.dev**

Alamat: Puri Indah Financial Tower  
Lantai 6, Unit 0612  
Jl. Puri Lingkar Dalam Blok T8  
Kembangan, Jakarta Barat 11610  
Email: [info@singkong.dev](mailto:info@singkong.dev)

ISBN 978-602-52770-8-5

