UIN SYARIF HIDAYATULLAH JAKARTA

URUTAN BERBASIS ARRAY

Syahrul Akbar Ramdhani
11230940000027
Dosen Pengampu: M. Irvan Septiar Musti, S. Si., M. Si.
Kelas 4A
Program Studi Matematika
Fakultas Sains dan Teknologi
Universitas Islam Negeri Syarif Hidayatullah Jakarta

DAFTAR ISI

- Jenis Urutan Python
- Efisiensi Jenis Urutan Python
- Array Tingkat Rendah
- Menggunakan Urutan Berbasis Array
- ARRAY Dinamis dan Amortisasi
- Kumpulan Data Multidimensi
- closing

JENIS URUTAN PYTHON

PERILAKU PUBLIK KELAS SEQUENCE

Perilaku Publik Kelas Sequence

1. Kesamaan Umum:
   - Semua kelas sequence mendukung akses elemen via indeks (misal: seq[2]).
   - Dapat membuat salinan (copy) atau slice (misal: seq[1:5]).
2. Perbedaan Utama:
   - list: Mutable (dapat diubah), mendukung operasi tambah/hapus elemen.
   - tuple: Immutable (tidak dapat diubah), efisien untuk data statis.
   - str: Immutable, khusus untuk data karakter.
3. Pentingnya Pemahaman Perilaku:
   - Kesalahan pemahaman (misal: cara membuat salinan atau slice) dapat menyebabkan bug.
   - Contoh: Representasi data multidimensi sebagai list of lists memerlukan model mental yang akurat.

DETAIL DAN ANALISIS

Detail Implementasi Internal

1. Prinsip Enkapsulasi vs. Efisiensi:
   - Meski prinsip OOP menganjurkan enkapsulasi (pengguna tidak perlu tahu detail internal), efisiensi program bergantung pada implementasi struktur data.
   - Contoh: Operasi append pada list diamortisasi O(1), tetapi insert(0, val) memerlukan O(n).
2. Alasan Mempelajari Implementasi:
   - Menghindari operasi yang tidak efisien (misal: penggunaan += pada string besar).
   - Memilih struktur data sesuai kebutuhan (misal: tuple untuk data tetap, list untuk data dinamis).

Analisis Asimtotik dan Eksperimental

1. Analisis Asimtotik:
   - Menggunakan notasi Big-O (dari Bab 3) untuk mendeskripsikan kompleksitas waktu operasi.
   - Contoh:
     - len(seq): O(1).
     - seq.insert(0, val): O(n).
2. Analisis Eksperimental:
   - Pengujian empiris untuk memvalidasi teori asimtotik.
   - Contoh: Mengukur waktu rata-rata operasi append vs insert untuk membuktikan O(1) vs O(n).

ARRAY TINGKAT RENDAH

PERILAKU PUBLIK KELAS SEQUENCE

Untuk memahami cara Python merepresentasikan tipe sequence, kita perlu terlebih dahulu memahami arsitektur komputer tingkat rendah. Memori utama komputer terdiri dari bit-bit informasi, yang biasanya dikelompokkan menjadi unit yang lebih besar bernama byte (1 byte = 8 bit). Setiap byte memiliki alamat memori unik, yaitu angka yang menunjukkan lokasi byte tersebut dalam sistem. Alamat ini memungkinkan komputer membedakan dan mengakses data di lokasi tertentu, seperti byte ke-2150 atau ke-2157. Alamat-alamat ini biasanya disusun secara berurutan, sesuai dengan tata letak fisik memori.

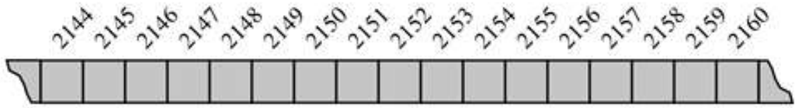


Figure 5.1: A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses.

Meskipun alamat memori disusun secara berurutan, perangkat keras komputer dirancang agar dapat mengakses setiap byte di memori utama secara efisien, berdasarkan alamatnya. Oleh karena itu, memori utama disebut sebagai RAM (Random Access Memory), karena waktu akses ke byte mana pun, seperti byte ke-8675309 atau ke-309, sama cepatnya, yaitu dalam waktu konstan atau O(1). Dalam pemrograman, bahasa pemrograman akan mencatat hubungan antara nama variabel (identifier) dan alamat memori tempat nilai tersebut disimpan. Untuk menyimpan sekelompok data yang saling terkait, seperti sepuluh skor tertinggi dalam sebuah gim, kita tidak perlu membuat sepuluh variabel terpisah—cukup satu nama dengan indeks untuk mengakses masing-masing nilai dalam kelompok tersebut.

PERILAKU PUBLIK KELAS SEQUENCE

Sekelompok variabel yang saling terkait dapat disimpan secara berurutan dalam bagian memori yang bersebelahan (contiguous), dan representasi seperti ini disebut sebagai array. Sebagai contoh nyata, sebuah string teks disimpan sebagai urutan karakter yang teratur. Di Python, setiap karakter direpresentasikan menggunakan Unicode, yang umumnya disimpan sebagai 16 bit (2 byte) per karakter. Jadi, string enam karakter seperti "SAMPLE" akan disimpan dalam 12 byte memori yang berurutan.

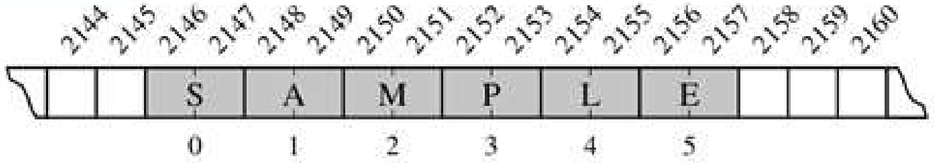


Figure 5.2: A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires two bytes of memory. The numbers below the entries are indices into the string.

PERILAKU PUBLIK KELAS SEQUENCE

Sebuah array dapat terdiri dari beberapa elemen (misalnya 6 karakter), dan meskipun membutuhkan 12 byte memori, kita tetap menyebutnya sebagai array enam elemen. Setiap lokasi dalam array disebut sel (cell) dan diakses menggunakan indeks angka, mulai dari 0. Misalnya, sel dengan indeks 4 berisi huruf 'L' dan disimpan di byte memori 2154 dan 2155. Semua sel dalam array harus menggunakan jumlah byte yang sama, agar akses ke elemen tertentu dapat dilakukan dalam waktu konstan (O(1)). Dengan mengetahui alamat awal array, ukuran per elemen, dan indeks yang diinginkan, kita bisa menghitung alamat memori sel tersebut menggunakan rumus:
alamat = start + (ukuran sel × indeks). Contohnya, jika array dimulai di alamat 2146, dan setiap karakter Unicode menggunakan 2 byte, maka sel ke-4 berada di alamat:
2146 + 2 × 4 = 2154.

Tentu saja, aritmatika untuk menghitung alamat memori dengan larik dapat ditangani secara otomatis. Oleh karena itu, sebuah program dapat membayangkan ekstraksi tingkat tinggi dari larik karakter seperti yang digambarkan pada Gambar 5.3

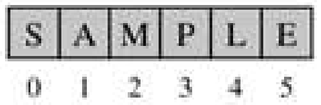


Figure 5.3: A higher-level abstraction for the string portrayed in Figure 5.2.

5.2.1 ARRAY REFERENSIAL

Untuk menyimpan daftar pasien di 200 tempat tidur rumah sakit yang diberi nomor 0 hingga 199, Python bisa menggunakan list misalnya seperti: ["Rene", "Joseph", "Janet", "Jonas", "Helen", "Virginia", ...] . Karena string memiliki panjang berbeda, Python tidak menyimpan string langsung dalam array. Sebagai gantinya, Python menggunakan array berisi referensi (alamat memori) ke string, sehingga efisien dan fleksibel dalam menangani data dengan ukuran bervariasi.

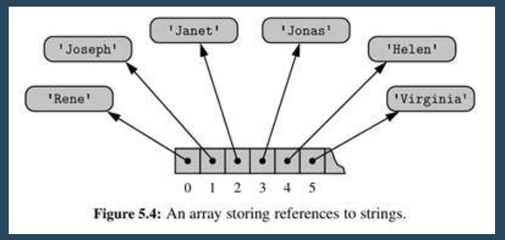


Figure 5.4: An array storing references to strings.

Meskipun ukuran elemen bisa berbeda-beda, alamat memori tiap elemen disimpan dalam jumlah bit yang tetap (misalnya 64-bit), sehingga Python dapat mengakses elemen list atau tuple dalam waktu konstan (O(1)) berdasarkan indeks. Dalam contoh daftar pasien rumah sakit, data bisa lebih kompleks dan disimpan sebagai objek, seperti instance dari kelas Patient. List tetap hanya menyimpan referensi ke objek-objek tersebut, dan untuk tempat tidur kosong, bisa digunakan referensi ke objek None.

List dan tuple di Python bersifat referensial, artinya mereka menyimpan referensi ke objek, bukan salinan objek itu sendiri. Karena itu, satu objek bisa muncul di beberapa list, atau satu list bisa berisi beberapa referensi ke objek yang sama. Misalnya, saat membuat slice dari sebuah list, Python menghasilkan list baru yang berisi referensi yang sama ke elemen-elemen dari list asli, bukan membuat salinan datanya

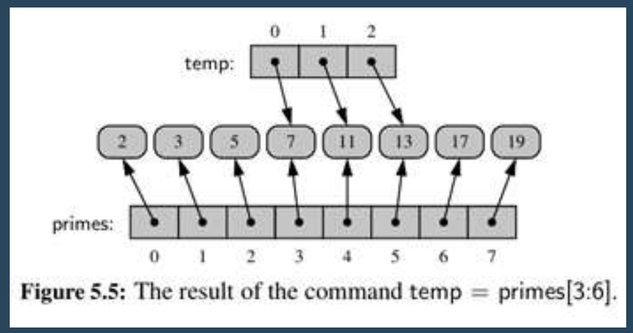


Figure 5.5: The result of the command temp = primes[3:6].

5.2.1 ARRAY REFERENSIAL

Jika elemen dalam list adalah objek immutable seperti integer, maka berbagi elemen antar list tidak menjadi masalah, karena objek tersebut tidak bisa diubah. Misalnya, saat menjalankan temp[2] = 15, yang terjadi bukan mengubah nilai integer lama, melainkan mengubah referensi di indeks ke-2 list temp agar menunjuk ke objek integer baru (15), tanpa memengaruhi list atau objek lain yang berbagi referensi sebelumnya.

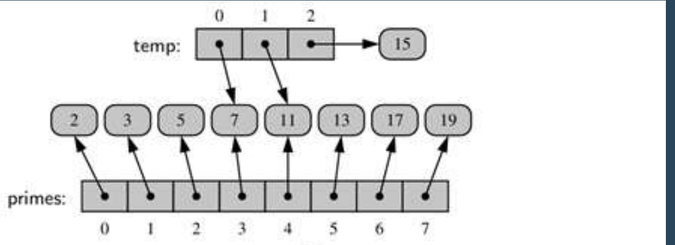


Figure 5.6: The result of the command temp[2] = 15 upon the configuration portrayed in Figure 5.5.

Saat membuat salinan list dengan backup = list(primes), yang dihasilkan adalah shallow copy, yaitu list baru yang tetap mereferensikan elemen yang sama dengan list asal. Jika elemen-elemennya bersifat immutable, hal ini tidak menjadi masalah. Namun jika elemen bersifat mutable, untuk membuat salinan sepenuhnya terpisah (deep copy), perlu digunakan fungsi deepcopy dari modul copy

5.2.1 ARRAY REFERENSIAL

Saat membuat list seperti counters = [0] * 8, Python menghasilkan list dengan 8 elemen bernilai nol. Namun secara teknis, semua elemen mereferensikan objek nol yang sama, bukan delapan objek nol yang berbeda. Seperti yang digambarkan pada gambar 5.7

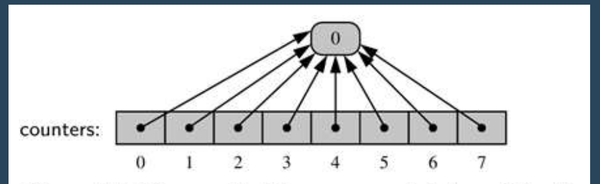


Figure 5.7: The result of the command data = [0] * 8.

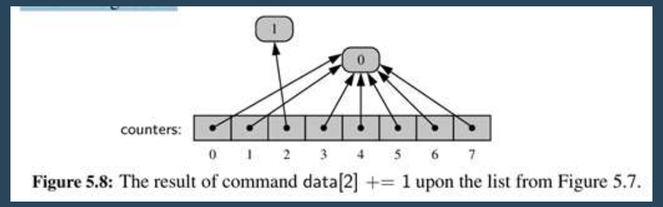


Figure 5.8: The result of command data[2] += 1 upon the list from Figure 5.7.

Meskipun semua elemen list counters = [0] * 8 mereferensikan objek nol yang sama, hal ini tidak masalah karena integer bersifat immutable. Misalnya, perintah counters[2] += 1 tidak mengubah nilai nol yang ada, melainkan membuat integer baru (nilai 1) dan mengubah referensi pada indeks ke-2 agar menunjuk ke nilai baru tersebut. Seperti yang digambarkan pada gambar 5.8.

Sebagai manifestasi akhir dari sifat referensial dari daftar, Perintah extend pada list akan menambahkan referensi elemen-elemen dari list lain, bukan menyalin elemen-elemennya. Jadi, list hasil extend akan menunjuk ke objek yang sama seperti list sumber.

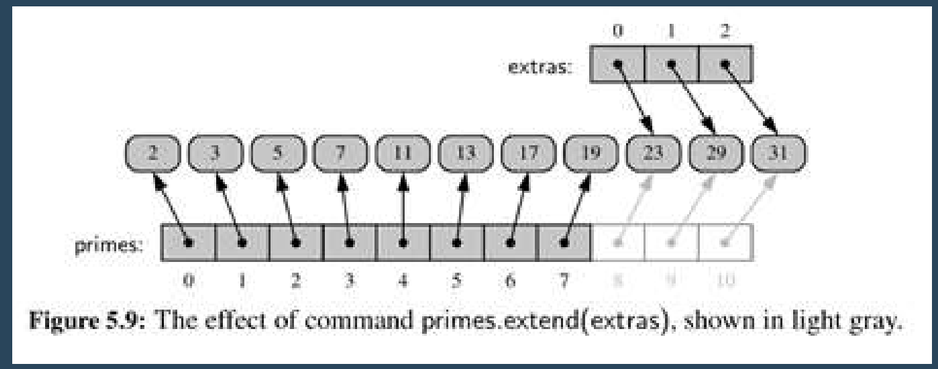
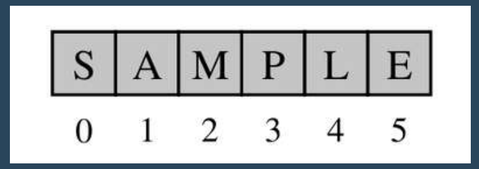


Figure 5.9: The effect of command primes.extend(extras), shown in light gray.

5.2.2 ARRAY RINGKAS DALAM PYTHON

Compact array dalam Python adalah struktur data yang menyimpan elemen-elemen secara langsung dalam memori secara berurutan (kontigu), tanpa menggunakan referensi ke objek lain. Ini berbeda dari list biasa yang menyimpan referensi ke objek-objek. Kita akan merujuk representasi yang lebih langsung ini sebagai larik padat karena larik ini menyimpan bit yang merepresentasikan
data utama (karakter, dalam kasus string)



Compact array di Python menyimpan data secara langsung (bukan referensi), sehingga lebih efisien dalam penggunaan memori dan performa. Misalnya, karakter string disimpan sebagai deretan byte (tiap karakter biasanya 2 byte untuk Unicode). Berbeda dengan struktur referensial yang membutuhkan tambahan 64-bit untuk menyimpan alamat memori tiap elemen.
Contohnya, menyimpan satu juta integer 64-bit dalam list biasa bisa menggunakan 4–5 kali lebih banyak memori dibanding array kompak. Hal ini karena setiap elemen disimpan sebagai objek dengan overhead tambahan.
Python menyediakan modul array untuk membuat array kompak, yang hemat memori dan cocok untuk data numerik dalam jumlah besar.

5.2.2 ARRAY RINGKAS DALAM PYTHON

Dukungan utama untuk compact array di Python tersedia melalui modul array, yang memungkinkan penyimpanan data primitif (seperti integer, float) secara efisien dan berurutan dalam memori.

- Untuk membuat array, digunakan kode tipe (type code) yang menunjukkan jenis data yang akan disimpan.
- Contoh: array('h', [2, 3, 5, 7, 11, 13, 17, 19]) membuat array berisi bilangan bulat bertipe signed short int ('h').
- Setiap elemen array menggunakan jumlah byte tetap tergantung pada jenis data.
- Tabel kode tipe (Tabel 5.1) menunjukkan berbagai jenis data (char, int, float, dll.) beserta jumlah byte yang umum digunakan.

Kode tipe ini berasal dari bahasa C, dan ukuran pastinya bisa berbeda tergantung sistem, tapi umumnya:

Code	C Data Type	Typical Number of Bytes
'b'	signed char	1
'B'	unsigned char	1
'u'	Unicode char	2 or 4
'h'	signed short int	2
'H'	unsigned short int	2
'i'	signed int	2 or 4
'I'	unsigned int	2 or 4
'l'	signed long int	4
'L'	unsigned long int	4
'f'	float	4
'd'	float	8

Table 5.1: Type codes supported by the array module.

SUSUNAN DINAMIS DAN AMORTISASI

DYNAMIC ARRAY PADA PYTHON LIST

Saat membuat larik tingkat rendah dalam sistem komputer, ukuran larik harus secara eksplisit dideklarasikan agar sistem dapat mengalokasikan bagian memori yang berurutan secara tepat. Sebagai contoh, Gambar 5.11 menampilkan larik 12 yang dapat disimpan di lokasi memori 2146 hingga 2157.

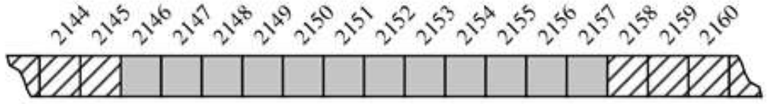


Figure 5.11: An array of 12 bytes allocated in memory locations 2146 through 2157.

Python menggunakan dynamic array untuk mengelola list, memungkinkan penambahan elemen tanpa batas meskipun kapasitas awal tetap. Sistem menyimpan list dalam array yang lebih besar dari jumlah elemennya. Jika kapasitas habis, Python membuat array baru yang lebih besar dan menyalin data lama ke dalamnya, mirip seperti kepiting pertapa yang berpindah cangkang. Bukti strategi ini dapat dilihat dengan fungsi getsizeof dari modul sys untuk mengukur penggunaan memori.

```
1 import sys                          # provides getsizeof function
2 data = [ ]
3 for k in range(n):                  # NOTE: must fix choice of n
4   a = len(data)                     # number of elements
5   b = sys.getsizeof(data)           # actual size in bytes
6   print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
7   data.append(None)                 # increase length by one
```

Code Fragment 5.1: An experiment to explore the relationship between a list's length and its underlying size in Python.

```
Length:   0; Size in bytes:   72
Length:   1; Size in bytes:  104
Length:   2; Size in bytes:  104
Length:   3; Size in bytes:  104
Length:   4; Size in bytes:  104
Length:   5; Size in bytes:  136
Length:   6; Size in bytes:  136
Length:   7; Size in bytes:  136
Length:   8; Size in bytes:  136
Length:   9; Size in bytes:  200
Length:  10; Size in bytes:  200
Length:  11; Size in bytes:  200
Length:  12; Size in bytes:  200
Length:  13; Size in bytes:  200
Length:  14; Size in bytes:  200
Length:  15; Size in bytes:  200
Length:  16; Size in bytes:  200
Length:  17; Size in bytes:  272
Length:  18; Size in bytes:  272
Length:  19; Size in bytes:  272
Length:  20; Size in bytes:  272
Length:  21; Size in bytes:  272
Length:  22; Size in bytes:  272
Length:  23; Size in bytes:  272
Length:  24; Size in bytes:  272
Length:  25; Size in bytes:  272
Length:  26; Size in bytes:  352
```

Code Fragment 5.2: Sample output from the experiment of Code Fragment 5.1.

DYNAMIC ARRAY PADA PYTHON LIST

Eksperimen menunjukkan bahwa sebuah list kosong di Python sudah menggunakan memori sebesar 72 byte. Ini karena setiap objek di Python menyimpan informasi internal seperti referensi ke kelasnya. Struktur internal list diperkirakan memiliki atribut seperti:

- n: jumlah elemen saat ini,
- _capacity: kapasitas maksimum list saat ini,
- _A: referensi ke array yang dialokasikan.

Saat elemen pertama ditambahkan, penggunaan memori meningkat dari 72 ke 104 byte (bertambah 32 byte). Ini sesuai dengan alokasi memori untuk menyimpan 4 referensi objek (4 × 8 byte).

- Setelah elemen kelima ditambahkan, memori meningkat lagi menjadi 136 byte, menunjukkan kapasitas ditingkatkan menjadi 8 referensi objek tambahan. Kenaikan kapasitas ini terjadi secara bertahap dan efisien.
- Ketika elemen ke-17 dimasukkan, memori naik menjadi 272 byte, cukup untuk menyimpan hingga 25 referensi objek. Hal ini menunjukkan bahwa Python menyesuaikan kapasitas list secara dinamis dan mengalokasikan memori secara bertahap untuk menghindari alokasi berulang.
- Akhirnya, karena list hanya menyimpan referensi, memori yang digunakan tidak termasuk ukuran data sebenarnya yang direferensikan oleh list, hanya ukuran untuk mengelola referensinya. Analisis ini penting untuk memahami kinerja array dinamis dan amortisasi biaya operasinya.

5.3.1 MENERAPKAN ARRAY DINAMIS

Meskipun Python sudah menyediakan list dengan implementasi dynamic array yang efisien, penting untuk memahami bagaimana cara kerjanya. Ketika array internal (larik) sudah penuh dan elemen baru ingin ditambahkan, proses berikut dilakukan:

- 1. Alokasikan larik baru B dengan kapasitas yang lebih besar.
- 2. Set B[i] = A[i], untuk i = 0,...,n-1, di mana n menyatakan jumlah item saat ini.
- 3. Set A = B, yaitu, kita selanjutnya menggunakan B sebagai larik yang mendukung daftar.
- 4. Masukkan elemen baru ke dalam larik yang baru.

Ilustrasi dari proses ini ditunjukkan pada Gambar 5.12

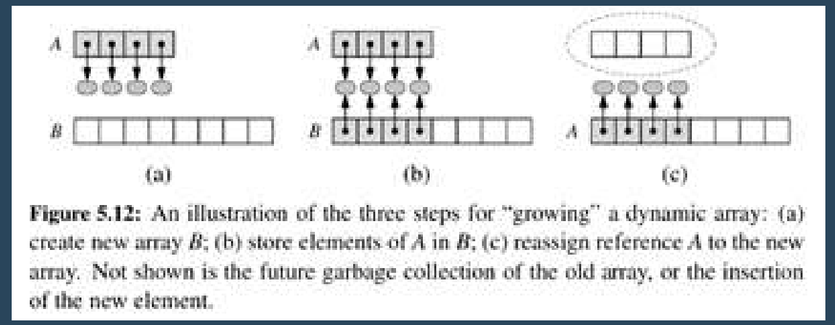


Figure 5.12: An illustration of the three steps for "growing" a dynamic array: (a) create new array *B*; (b) store elements of *A* in *B*; (c) reassign reference *A* to the new array. Not shown is the future garbage collection of the old array, or the insertion of the new element.

Masalah terakhir yang perlu dipertimbangkan adalah seberapa besar kapasitas array baru yang harus dibuat. Umumnya, kapasitas baru dibuat dua kali lipat dari array sebelumnya yang telah penuh. Analisis matematis untuk mendukung pilihan ini akan dijelaskan di bagian 5.3.2.
Pada Code Fragment 5.3, ditunjukkan implementasi konkret dari dynamic array di Python melalui kelas DynamicArray. Kelas ini meniru antarmuka list Python dengan fungsionalitas terbatas, seperti metode append, serta akses ke panjang (len) dan elemen (getitem). Untuk membuat array tingkat rendah, digunakan modul ctypes, namun penjelasan detail tentang modul ini diabaikan karena tidak digunakan di bagian lain buku ini. Pembuatan array dilakukan melalui metode privat make_array, dan proses perluasan kapasitas dilakukan dalam metode resize.

IMPLEMENTASI PADA PYTHON

```python
import ctypes                                   # provides low-level arrays

class DynamicArray:
  """A dynamic array class akin to a simplified Python list."""

  def __init__(self):
    """Create an empty array."""
    self._n = 0                                 # count actual elements
    self._capacity = 1                          # default array capacity
    self._A = self._make_array(self._capacity)  # low-level array

  def __len__(self):
    """Return number of elements stored in the array."""
    return self._n

  def __getitem__(self, k):
    """Return element at index k."""
    if not 0 <= k < self._n:
      raise IndexError('invalid index')
    return self._A[k]                           # retrieve from array
```

```python

  def append(self, obj):
    """Add object to end of the array."""
    if self._n == self._capacity:               # not enough room
      self._resize(2 * self._capacity)          # so double capacity
    self._A[self._n] = obj
    self._n += 1

  def _resize(self, c):                         # nonpublic utitity
    """Resize internal array to capacity c."""
    B = self._make_array(c)                     # new (bigger) array
    for k in range(self._n):                    # for each existing value
      B[k] = self._A[k]
    self._A = B                                 # use the bigger array
    self._capacity = c

  def _make_array(self, c):                     # nonpublic utitity
    """Return new array with capacity c."""
    return (c * ctypes.py_object)( )            # see ctypes documentation
```

Code Fragment 5.3: An implementation of a DynamicArray class, using a raw array from the ctypes module as storage.

5.3.2 ANALISIS AMORTISASI ARRAY DINAMIS

Analisis ini membahas efisiensi waktu eksekusi operasi pada array dinamis, khususnya operasi append. Meskipun satu penambahan elemen bisa memakan waktu Ω(n) saat kapasitas array habis dan perlu diganti, penggunaan strategi penggandaan kapasitas array membuat operasi secara keseluruhan tetap efisien. Dengan pendekatan analisis amortisasi, biaya tinggi dari operasi langka (seperti penggantian array) disebarkan ke banyak operasi murah, sehingga total waktu eksekusi tetap efisien. Teknik ini dianalogikan seperti pembayaran dengan “cyber-dollar”, di mana setiap operasi membayar biaya tetap dan menyisihkan dana untuk membiayai operasi mahal di masa depan.

CONTOH GAMBAR

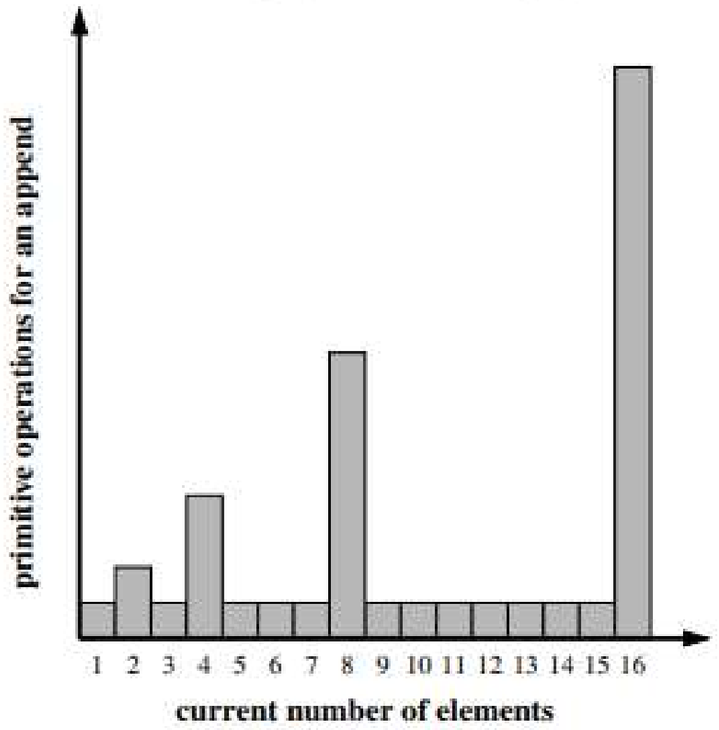


Figure 5.13: Running times of a series of append operations on a dynamic array.

PROPOSISI DAN JUSTIFIKASI

Proposisi 5.1:
Misalkan S adalah sebuah urutan (sequence) yang diimplementasikan menggunakan array dinamis dengan kapasitas awal satu, menggunakan strategi penggandaan ukuran array ketika penuh. Total waktu untuk melakukan serangkaian nn operasi append pada S, dimulai dari S yang kosong, adalah O(n).
Justifikasi:

- Asumsikan 1 cyber-dolar cukup untuk membayar eksekusi setiap operasi append pada S, kecuali waktu yang dihabiskan untuk memperbesar array.
- Perbesar array dari ukuran k ke 2^k membutuhkan k cyber-dolar (untuk inisialisasi array baru).
- Setiap operasi append dibebankan 3 cyber-dolar (overcharge):
  - 1 cyber-dolar untuk biaya append dasar.
  - 2 cyber-dolar disimpan sebagai "tabungan" di sel tempat elemen dimasukkan

Mekanisme Tabungan:

- Overflow terjadi ketika array S memiliki 2^i elemen, dan ukuran array saat itu adalah 2^i.
- Perbesar array dari 2^i ke 2^{i+1} membutuhkan 2^i cyber-dolar.
- Tabungan cyber-dolar tersedia di sel 2^{i-1} hingga 2^{i-1} (karena overflow sebelumnya terjadi saat elemen melebihi 2^{i-1} dan tabungan di sel tersebut belum terpakai).

Skema Valid:

- Total biaya untuk nn operasi append = 3n cyber-dolar.
- Amortized running time per operasi = O(1), sehingga total waktu nn operasi = O(n).

Kesimpulan:
Strategi ini menjamin bahwa biaya mahal resize array dapat dibiayai oleh tabungan cyber-dolar dari operasi sebelumnya, sehingga efisiensi amortisasi tercapai.

CONTOH GAMBAR

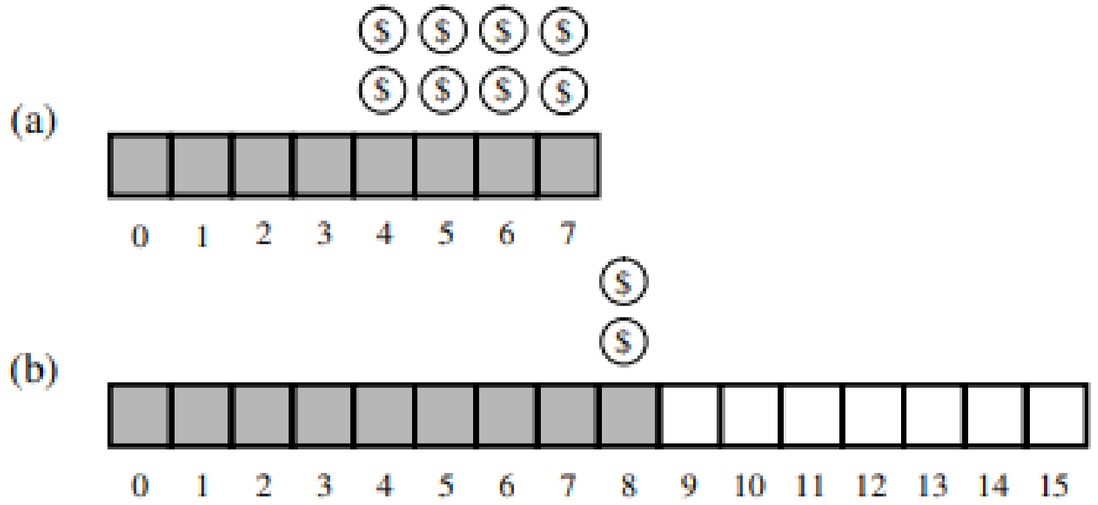


Figure 5.14: Illustration of a series of append operations on a dynamic array: (a) an 8-cell array is full, with two cyber-dollars "stored" at cells 4 through 7; (b) an append operation causes an overflow and a doubling of capacity. Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table. Inserting the new element is paid for by one of the cyber-dollars charged to the current append operation, and the two cyber-dollars profited are stored at cell 8.

PENINGKATAN KAPASITAS SECARA GEOMETRIK

Meskipun bukti Proposisi 5.1 menggunakan penggandaan ukuran array, batas O(1) amortisasi per operasi dapat dibuktikan untuk setiap peningkatan kapasitas berbasis deret geometrik (lihat Bagian 2.4.2). Pemilihan basis deret geometrik melibatkan kompromi antara efisiensi waktu eksekusi dan penggunaan memori:

- Basis Besar (contoh: 2):
  - Jika operasi terakhir memicu resize, array akan 2x lebih besar dari yang diperlukan → memori terbuang.
- Basis Kecil (contoh: 1.25):
  - Memori lebih efisien, tetapi resize terjadi lebih sering → biaya runtime tambahan.

Namun, batas O(1) amortisasi tetap berlaku asalkan penambahan kapasitas sebanding dengan ukuran array saat ini, meskipun dengan konstanta faktor lebih besar (misal: cyber-dolar per operasi >3).

HINDARI PENINGKATAN KAPASITAS DENGAN DERET ARITMATIKA

Untuk menghindari alokasi memori berlebihan, mungkin terlihat menarik untuk menggunakan strategi penambahan jumlah konstan sel (misal: +1, +2, +3) setiap kali array dinamis diresize. Sayangnya, strategi ini justru menghasilkan kinerja yang jauh lebih buruk:

- Kasus Ekstrem: Jika kapasitas hanya ditambah 1 sel setiap resize, setiap operasi append akan memicu resize. Akibatnya, total biaya menjadi 1+2+3+…+n (deret aritmatika) yang memiliki kompleksitas $\Omega(n^2)$.
- Penambahan 2/3 Sel: Meski sedikit lebih baik, biaya total tetap kuadratik (Gambar 5.13).

Mengapa Ini Bermasalah?

- Frekuensi Resize Tinggi: Penambahan kapasitas kecil tidak sebanding dengan pertumbuhan elemen, sehingga resize terjadi terus-menerus.
- Kontras dengan Deret Geometrik: Hanya deret geometrik (misal: penggandaan kapasitas) yang menjamin biaya amortisasi O(1).

CONTOH GAMBAR

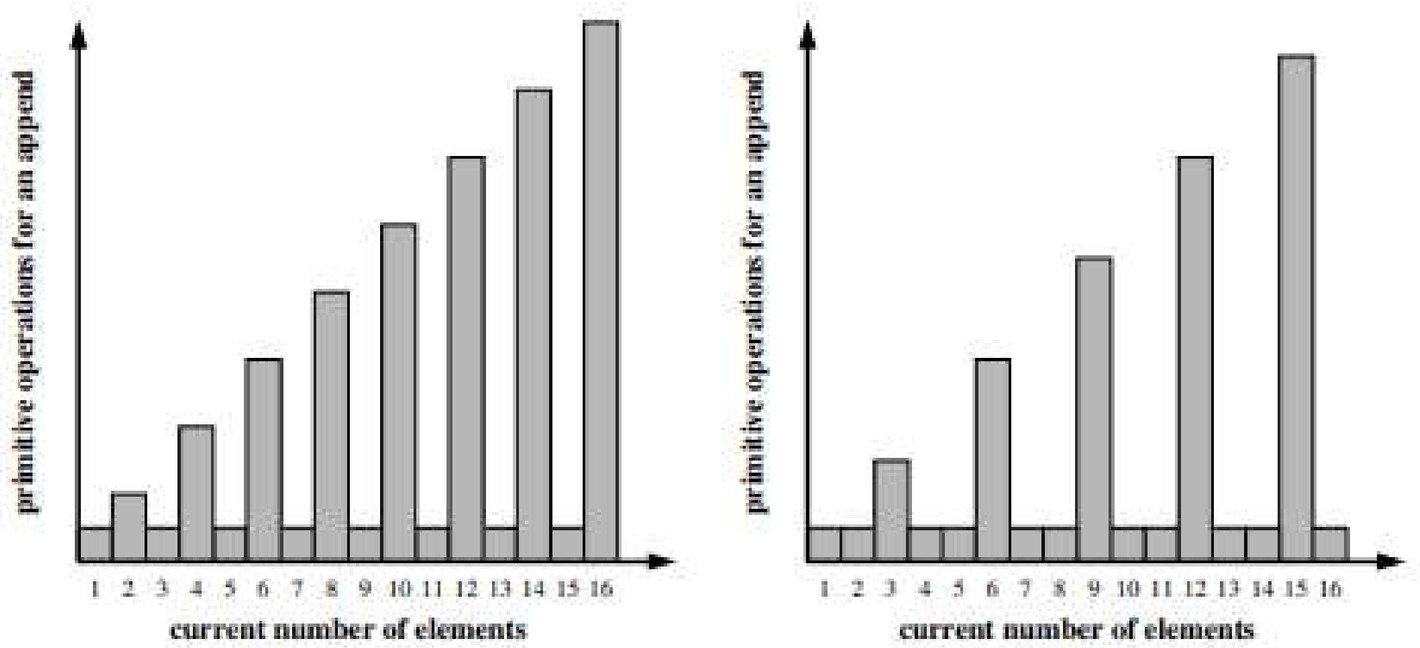


Figure 5.15: Running times of a series of append operations on a dynamic array using arithmetic progression of sizes. (a) Assumes increase of 2 in size of the array, while (b) assumes increase of 3.

PROPOSISI DAN JUSTIFIKASI

Proposisi 5.2: Melakukan serangkaian nn operasi append pada array dinamis (awalnya kosong) dengan peningkatan kapasitas tetap c per resize membutuhkan waktu $\Omega(n^2)$.

Justifikasi:
Setiap resize menambah kapasitas sebesar c (misal: 1000 sel).
Total waktu dihabiskan untuk inisialisasi array berukuran $c,2c,3c,\ldots,$ mc dengan $m=\lceil n/c \rceil$.
Total biaya = $c+2c+3c+\ldots+mc=c \cdot m(m+1) / 2$
Substitusi $m \approx n / c$:

$$\text{Total waktu} \geq n^2 / 2c \Rightarrow \Omega(n^2).$$

Mengapa Ini Bermasalah?

- Penambahan tetap (misal +1000 sel) awalnya terlihat besar, tetapi saat n*n* membesar, peningkatan ini menjadi tidak signifikan.
- Resize terlalu sering → Biaya total tumbuh secara kuadratik.

Pelajaran Penting:

1. Desain strategi resize sangat kritis:
    - Deret aritmatika (tambah tetap) → $\Omega(n^2)$.
    - Deret geometrik (tambah proporsional) → $O(n)$.
2. Analisis amortisasi membantu memahami trade-off desain struktur data.

(Lihat Proposisi 5.1 untuk perbandingan dengan strategi penggandaan kapasitas.)

PENGGUNAAN MEMORI DAN PENYUSUTAN ARRAY

1. Dampak Peningkatan Geometrik:Peningkatan kapasitas secara geometrik memastikan ukuran array akhir sebanding dengan total elemen, sehingga penggunaan memori efisien (O(n)).
2. Risiko Operasi Penghapusan: Jika struktur data (seperti list di Python) mendukung penghapusan elemen, kapasitas array mungkin tetap besar meski elemen sedikit → pemborosan memori.
3. Solusi: Penyusutan Array
   - Strategi Robust: Susutkan array ketika jumlah elemen turun di bawah 1/4 kapasitas (misal: kapasitas dijadikan setengah).
   - Manfaat: Kapasitas array tidak pernah melebihi 4x jumlah elemen → memori tetap O(n).
4. Tantangan:
   - Jika array terlalu sering resize (membesar dan mengecil bergantian), batas O(1) amortisasi tidak tercapai.
5. Studi Kasus:
   - Latihan C-5.16: Implementasi strategi penyusutan.
   - Latihan C-5.17 & C-5.18: Analisis amortisasi untuk memvalidasi efisiensi.

5.3.3 KELAS LIST PYTHON

Eksperimen pada Cuplikan Kode 5.1 dan 5.2 (di awal Bagian 5.3) memberikan bukti empiris bahwa kelas list di Python menggunakan bentuk array dinamis untuk penyimpanannya. Namun, pemeriksaan detail terhadap kapasitas array (lihat Latihan R-5.2 dan C-5.13) menunjukkan bahwa Python tidak menggunakan deret geometrik murni maupun deret aritmatik untuk strategi resize. Meski demikian, implementasi metode append di Python jelas menunjukkan perilaku waktu konstan teramortisasi (O(1)). Hal ini bisa dibuktikan secara eksperimen:

- Operasi append tunggal biasanya sangat cepat, sehingga sulit diukur akurat. Namun, operasi resize yang mahal kadang terdeteksi.
- Untuk mengukur biaya amortisasi per operasi, dilakukan serangkaian n operasi append pada list kosong, lalu dihitung rata-rata waktu per operasi. Fungsi untuk eksperimen ini terdapat di Cuplikan Kode 5.4.

Kesimpulan:
Strategi resize unik Python memastikan efisiensi amortisasi tanpa mengikuti pola geometrik/aritmatik ketat.

IMPLEMENTASI

```
1  from time import time                    # import time function from time module
2  def compute_average(n):
3    """Perform n appends to an empty list and return average time elapsed."""
4    data = []
5    start = time( )                        # record the start time (in seconds)
6    for k in range(n):
7      data.append(None)
8    end = time( )                          # record the end time (in seconds)
9    return (end − start) / n               # compute average per operation
```

Code Fragment 5.4: Measuring the amortized cost of append for Python's list class.

n	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
μs	0.219	0.158	0.164	0.151	0.147	0.147	0.149

Table 5.2: Average running time of append, measured in microseconds, as observed over a sequence of n calls, starting with an empty list.

Secara teknis, waktu yang diukur antara awal dan akhir eksperimen mencakup waktu iterasi loop for selain eksekusi operasi append. Hasil empiris untuk berbagai nilai n (Tabel 5.2) menunjukkan:

- Biaya rata-rata lebih tinggi pada dataset kecil, sebagian karena overhead iterasi loop.
- Varians alami terjadi karena resize terakhir berdampak signifikan relatif terhadap n.

Namun, secara keseluruhan, bukti menunjukkan bahwa waktu amortisasi per operasi append bersifat konstan (tidak tergantung pada n), yang mengonfirmasi efisiensi O(1).

EFISIENSI JENIS URUTAN PYTHON

5.4.1 KELAS LIST DAN TUPLE PYTHON

Perilaku non-mutasi (operasi yang tidak mengubah struktur) pada kelas list sama dengan yang didukung oleh kelas tuple. Namun, tuple umumnya lebih hemat memori karena bersifat immutable (tidak dapat diubah), sehingga tidak memerlukan array dinamis dengan kapasitas cadangan.

Efisiensi Asimtotik:

- Operasi seperti akses elemen (S[i]), pengecekan keanggotaan (x in S), dan len(S) memiliki kompleksitas O(1) untuk list dan tuple.
- Penggunaan Memori:
  - tuple: Memori yang digunakan persis sebanding dengan jumlah elemen (O(n)).
  - list: Memori lebih besar karena kapasitas array dinamis sengaja dilebihkan untuk mengakomodasi operasi mutasi di masa depan.

Tabel 5.3 merangkum efisiensi asimtotik ini, dengan penjelasan lebih lanjut tentang analisisnya.

Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Table 5.3: Asymptotic performance of the nonmutating behaviors of the list and tuple classes. Identifiers data, data1, and data2 designate instances of the list or tuple class, and n, n_1, and n_2 their respective lengths. For the containment check and index method, k represents the index of the leftmost occurrence (with $k = n$ if there is no occurrence). For comparisons between two sequences, we let k denote the leftmost index at which they disagree or else $k = \min(n_1, n_2)$.

OPERASI WAKTU KONSTAN DAN PENCARIAN NILAI DALAM LIST

Operasi Waktu Konstan

- len(data) (panjang data) dikembalikan dalam waktu konstan karena list/tuple secara eksplisit menyimpan informasi panjangnya.
- Efisiensi O(1) untuk sintaks data[j] (akses elemen indeks ke-j) dijamin oleh akses langsung ke array dasar (underlying array).

Pencarian Nilai dalam List

Metode count, index, dan operator in bekerja dengan mengiterasi elemen dari kiri ke kanan:

- count: Memeriksa semua elemen untuk menghitung kemunculan nilai → O(n).
- index dan in: Berhenti begitu menemukan kejadian pertama nilai yang dicari.
  - Kasus terbaik: O(1) (nilai ditemukan di posisi awal).
  - Kasus terburuk: O(n) (nilai di akhir/tidak ada).

Contoh Praktis:

Dengan data = list(range(10000000)) (list berisi 10 juta elemen):

- 5 in data selesai cepat karena 5 ada di indeks awal.
- 9999995 in data memerlukan iterasi hampir penuh.
- -5 in data gagal → iterasi semua elemen.

PERBANDINGAN LEKSIKOGRAFIS PADA URUTAN DAN PEMBUATAN INSTANSI BARU

Perbandingan Leksikografis pada Urutan

Perbandingan antara dua urutan dilakukan secara leksikografis (mirip pengurutan kamus):

- Kasus Terburuk: Evaluasi memerlukan iterasi sepanjang urutan terpendek, karena hasil baru pasti diketahui setelah salah satu urutan habis.
- Kasus Optimal: Jika perbedaan ditemukan di posisi awal, evaluasi langsung berhenti.
  - Contoh: [7, 3, ...] < [7, 5, ...] → hasil True langsung diketahui di elemen kedua (3 < 5).

Pembuatan Instansi Baru

Tiga perilaku terakhir pada Tabel 5.3 adalah operasi yang membuat instansi baru berdasarkan instansi yang sudah ada. Waktu eksekusi operasi ini bergantung pada proses konstruksi dan inisialisasi hasil baru, sehingga perilaku asimtotiknya sebanding dengan panjang hasil.

- Contoh:
  - data[6000000:6000008] hanya menghasilkan 8 elemen → dibuat hampir seketika.
  - data[6000000:7000000] menghasilkan 1 juta elemen → membutuhkan waktu lebih lama.

Kesimpulan:

- Ukuran hasil menentukan efisiensi operasi pembuatan instansi baru.
- Operasi dengan hasil kecil lebih cepat, sedangkan hasil besar memerlukan waktu linear terhadap ukurannya.

PERILAKU MUTASI PADA LIST

Perilaku Mutasi pada List

Efisiensi operasi mutasi (mengubah struktur data) pada kelas list dijelaskan dalam Tabel 5.3.

- Operasi Sederhana:
  - data[j] = val (mengubah elemen di indeks j) menggunakan metode __setitem__.
  - Waktu eksekusi O(1) karena hanya mengganti satu elemen tanpa mengubah kapasitas array.
- Operasi Kompleks (Tambah/Hapus Elemen):
  - Contoh: append, insert, pop.
  - Memerlukan analisis lebih mendalam karena dapat memicu perubahan kapasitas array (resize) → kompleksitas bervariasi (tergantung strategi resize).

Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse( )	$O(n)$
data.sort( )	$O(n \log n)$

*amortized

Table 5.4: Asymptotic performance of the mutating behaviors of the list class. Identifiers data, data1, and data2 designate instances of the list class, and n, n_1, and n_2 their respective lengths.

MENAMBAHKAN ELEMEN PADA LIST

Menambahkan Elemen ke List

Pada Seksi 5.3, metode append telah dibahas secara mendalam:

- Worst-case Ω(n) karena resize array, tetapi O(1) amortisasi.

Metode insert(k, value):

- Menyisipkan nilai di indeks k dengan menggeser elemen setelahnya.
- Implementasi (Cuplikan Kode 5.5):
  - Resize array → Biaya seperti append (amortized O(1)).
  - Menggeser elemen → Biaya O(n) karena pergeseran bisa melibatkan semua elemen (misal: insert di indeks 0).

Contoh Kasus:

- Insert di akhir list (seperti append) → pergeseran minimal (O(1)).
- Insert di awal list → semua elemen digeser → O(n).

Kesimpulan:

- Kompleksitas insert = O(n) karena pergeseran elemen (meski biaya resize diamortisasi).

```
1   def insert(self, k, value):
2     """Insert value at index k, shifting subsequent values rightward."""
3     # (for simplicity, we assume 0 <= k <= n in this verion)
4     if self._n == self._capacity:                    # not enough room
5       self._resize(2 * self._capacity)               # so double capacity
6     for j in range(self._n, k, -1):                  # shift rightmost first
7       self._A[j] = self._A[j-1]
8     self._A[k] = value                               # store newest element
9     self._n += 1
```

Code Fragment 5.5: Implementation of insert for our DynamicArray class.

ANALISIS KINERJA INSERT PADA POSISI BERBEDA

Analisis Kinerja Operasi insert pada Posisi Berbeda
Eksperimen mengukur waktu rata-rata operasi insert dalam tiga skenario:

1. Insert di Awal List:
   - Menggeser semua elemen → O(n) per operasi.
   - Contoh: Untuk N=1.000.000, waktu rata-rata 351.590 µs.
2. Insert di Tengah List:
   - Menggeser setengah elemen → Ω(n) (biaya ≈ separuh insert di awal).
   - Contoh: Untuk N=1.000.000, waktu rata-rata 175.383 µs.
3. Insert di Akhir List:
   - Tidak perlu menggeser elemen → O(1) (mirip append).
   - Waktu stabil ~0.4 µs, bahkan untuk N=1.000.000.

	N				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n // 2$	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

Table 5.5: Average running time of insert(k, val), measured in microseconds, as observed over a sequence of N calls, starting with an empty list. We let n denote the size of the current list (as opposed to the final list).

Kesimpulan:

- Insert di awal/tengah mahal untuk dataset besar (O(n)).
- Insert di akhir optimal (O(1)), seperti append.

MENGHAPUS ELEMEN DARI LIST

Menghapus Elemen dari List

Kelas list di Python menyediakan beberapa cara untuk menghapus elemen:

1. pop():
   - Menghapus elemen terakhir → O(1) diamortisasi (efisien, kecuali saat array dasar menyusut).
2. pop(k):
   - Menghapus elemen di indeks k → O(n−k).
   - Contoh:
     - pop(0) (hapus elemen pertama) → Ω(n) karena semua elemen digeser (Gambar 5.17).
3. remove(value):
   - Menghapus nilai pertama yang ditemukan → Ω(n).
     - Tahap 1: Pencarian nilai → O(n).
     - Tahap 2: Penggeseran elemen setelah nilai dihapus → O(n)

Figure 5.17: Removing an element at index k of a dynamic array.

```
1   def remove(self, value):
2     """Remove first occurrence of value (or raise ValueError)."""
3     # note: we do not consider shrinking the dynamic array in this version
4     for k in range(self._n):
5       if self._A[k] == value:                  # found a match!
6         for j in range(k, self._n - 1):        # shift others to fill gap
7           self._A[j] = self._A[j+1]
8         self._A[self._n - 1] = None            # help garbage collection
9         self._n -= 1                           # we have one less item
10        return                                 # exit immediately
11    raise ValueError('value not found')        # only reached if no match
```

Code Fragment 5.6: Implementation of remove for our DynamicArray class.

MEMPERLUAS LIST DENGAN METODE EXTEND DAN KONSTRUKSI LIST BARU

Python menyediakan metode extend untuk menambahkan semua elemen dari satu list ke akhir list lain. Contoh:
data.extend(other) # sama dengan loop: for element in other: data.append(element)
Perbandingan Efisiensi:

- extend lebih efisien daripada append berulang karena:
  a. Implementasi Native: Kode extend dijalankan di level terkompilasi (C), bukan Python interpreted.
  b. Overhead Rendah: Hanya 1 panggilan fungsi, bukan ribuan panggilan append.
  c. Resize Optimal: Kapasitas array dihitung sebelumnya → resize maksimal 1 kali.

Risiko Penggunaan append Berulang:

- Jika list other sangat besar, append berulang bisa memicu resize array berkali-kali → waktu eksekusi lebih lama.

Kesimpulan:

- Gunakan extend untuk menggabungkan list besar → lebih cepat dan hemat komputasi.

Konstruksi List Baru
Ada beberapa cara untuk membuat list baru di Python:

1. List Comprehension:
   - Contoh: kuadrat = [k*k for k in range(1, n+1)].
   - Efisiensi: Linear (O(n)), tetapi lebih cepat secara praktis dibandingkan loop dengan append karena optimasi internal.
2. Inisialisasi dengan Operator *:
   - Contoh: data = [0] * n → menghasilkan list dengan n elemen 0.
   - Keunggulan:
     - Sintaks singkat.
     - Lebih cepat karena alokasi memori dilakukan sekaligus, bukan bertahap.

5.4.2 KELAS STRING PYTHON
PENCOCOKAN POLA DALAM STRING

Analisis Efisiensi Operasi String di Python

1. Pentingnya String di Python:
   - String sangat penting di Python, dengan operator dan metode yang dibahas di Bab 1 dan Lampiran A.
   - Analisis efisiensi operasi string bergantung pada panjang string (n) atau pola (m).
2. Efisiensi Operasi String:
   - Operasi yang Menghasilkan String Baru (contoh: capitalize, strip):
     - Memerlukan waktu linear (O(n)) sesuai panjang string hasil.
   - Pengecekan Kondisi Boolean (contoh: islower, isalpha):
     - O(n) dalam kasus terburuk, tetapi bisa berhenti lebih cepat (short-circuit) jika kondisi sudah terpenuhi/tidak terpenuhi.
   - Operator Perbandingan (contoh: ==, <):
     - O(n) karena membandingkan karakter per karakter hingga ditemukan perbedaan atau akhir string.

Pencocokan Pola dalam String

Beberapa metode string seperti contains, find, index, count, replace, dan split memerlukan pencarian pola (substring) dalam string yang lebih besar.

- Implementasi Naif:
- Algoritma sederhana memeriksa semua kemungkinan posisi awal pola (dari indeks 0 hingga n−m+1) dan membandingkan m karakter di setiap posisi. Ini menghasilkan kompleksitas O(mn).
- Algoritma Optimal:
- Di Bagian 13.2, akan dijelaskan algoritma pencocokan pola (seperti Knuth-Morris-Pratt atau KMP) yang menyelesaikan masalah ini dalam waktu O(n).

Contoh:

- Mencari pola "abc" dalam string "aababcabc" → Algoritma naif membandingkan karakter di setiap posisi, sedangkan algoritma optimal melompati posisi yang tidak mungkin.

MENYUSUN STRING

Membuat String Besar dengan Efisien
Masalah dengan Penggabungan String Bertahap
Contoh kode berikut tidak direkomendasikan karena tidak efisien:

```
# PERINGATAN: jangan lakukan ini
letters = ""  # mulai dengan string kosong  for c in dokumen:if c.isalpha():
    letters += c  # menggabungkan karakter
```

- Penyebab Inefisiensi:
  - String bersifat immutable (tidak dapat diubah). Setiap operasi letters += c membuat string baru dan menyalin seluruh isi sebelumnya.
  - Kompleksitas waktu menjadi $O(n^2)$ (misal: untuk string panjang n, total operasi = 1+2+3+...+n).

Optimisasi Terbatas pada Beberapa Implementasi Python

- Beberapa versi Python menggunakan jumlah referensi (reference count) untuk mengoptimasi += pada string. Jika tidak ada variabel lain yang merujuk ke string tersebut, Python bisa memodifikasi string secara langsung.
- Masalah: Optimisasi ini tidak dijamin di semua lingkungan Python.

Solusi Efisien: Gunakan List dan join
Pendekatan Standar

```
temp = []  # mulai dengan list kosong  for c in dokumen:if c.isalpha():
    temp.append(c)  # tambahkan karakter ke list
letters = "".join(temp)  # gabungkan semua elemen list menjadi string
```

- Kompleksitas:
  - append ke list: O(n) (amortisasi).
  - join: O(n) (linear terhadap panjang string).
- Total waktu: O(n).

Optimisasi Lebih Lanjut dengan List Comprehension

```
letters = "".join([c for c in dokumen if c.isalpha()])
```

- Keunggulan:
  - Lebih ringkas dan cepat karena menghindari loop eksplisit.

Hindari List Sementara dengan Generator Comprehension

```
letters = "".join(c for c in dokumen if c.isalpha())
```

- Manfaat:
  - Tidak menyimpan seluruh data di memori sekaligus → hemat memori untuk dataset besar.

MENGGUNAKAN URUTAN BERBASIS ARRAY

5.5.1 MENYIMPAN SKOR TINGGI UNTUK PERMAINAN

- Aplikasi pertama yang dipelajari adalah menyimpan urutan entri skor tinggi (high score) dalam permainan video. Contoh ini mewakili banyak aplikasi yang memerlukan penyimpanan urutan objek, seperti rekaman pasien di rumah sakit atau daftar nama pemain dalam tim sepak bola. Fokusnya adalah pada entri skor tinggi karena aplikasi ini sederhana namun cukup kompleks untuk memperkenalkan konsep penting dalam struktur data.
- Setiap entri skor tinggi (GameEntry) minimal terdiri dari:

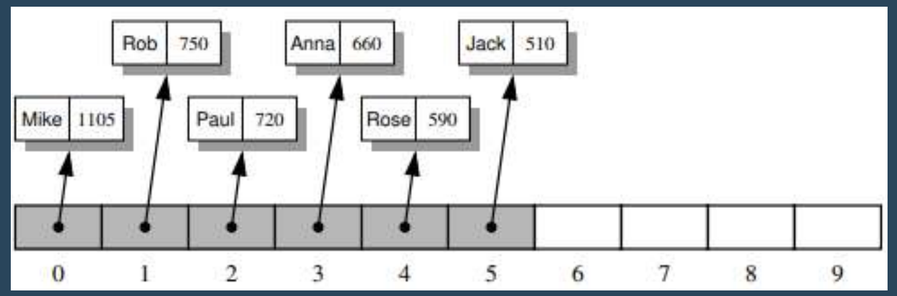
1. Score (skor): Nilai integer yang menunjukkan skor yang dicapai.
2. Name (nama): Nama pemain yang mencapai skor tersebut.

- Meskipun bisa ditambahkan detail lain (seperti tanggal atau statistik permainan), contoh ini disederhanakan hanya dengan dua komponen utama.

```
1  class GameEntry:
2    """Represents one entry of a list of high scores."""
3
4    def __init__(self, name, score):
5      self._name = name
6      self._score = score
7
8    def get_name(self):
9      return self._name
10
11   def get_score(self):
12     return self._score
13
14   def __str__(self):
15     return '({0}, {1})'.format(self._name, self._score) # e.g., '(Bob, 98)'
```

SEBUAH KELAS UNTUK NILAI TINGGI

- Untuk menyimpan daftar skor tertinggi (high scores), dikembangkan kelas bernama Scoreboard. Scoreboard memiliki batas maksimal entri yang dapat disimpan. Jika batas sudah terpenuhi, skor baru hanya bisa masuk jika lebih tinggi dari skor terendah yang ada di dalamnya. Ukuran scoreboard bisa bervariasi (misalnya 10, 50, atau 500 entri) dan ditentukan saat pembuatan objek.
- Secara internal, digunakan Python list bernama board untuk menyimpan objek GameEntry. List diinisialisasi dengan ukuran maksimal (diisi None terlebih dahulu) agar tidak perlu diubah ukurannya nanti. Entri disimpan berurutan dari skor tertinggi ke terendah, dimulai dari indeks 0.
- Implementasi lengkap kelas Scoreboard ditunjukkan dalam Code Fragment 5.8. Konstruktornya sederhana: perintah
- self.board = [None] * capacity
- membuat list dengan panjang yang diinginkan, tetapi semua entrinya masih None. Kami juga menggunakan variabel instansi n untuk melacak jumlah entri yang sudah terisi.
- Kelas Scoreboard juga menyediakan:
- Metode __getitem__ untuk mengakses entri di indeks tertentu (misal board[i]).
- Metode __str__ untuk menampilkan seluruh isi scoreboard dalam bentuk string (satu entri per baris).



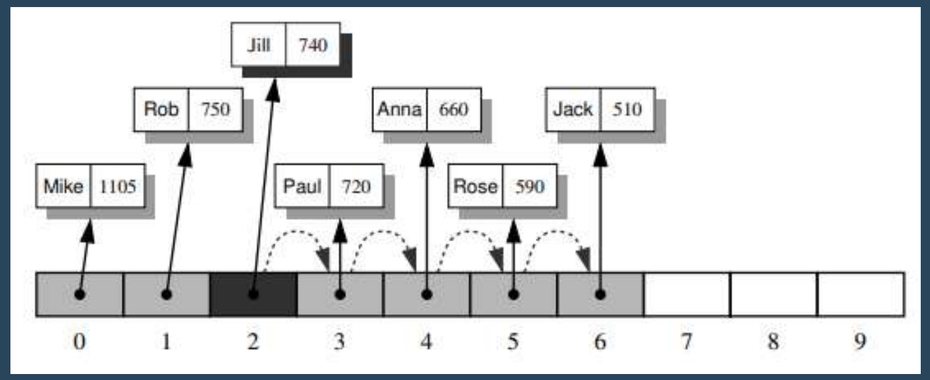
SEBUAH KELAS UNTUK NILAI TINGGI

```
1  class Scoreboard:
2    """Fixed-length sequence of high scores in nondecreasing order."""
3
4    def __init__(self, capacity=10):
5      """Initialize scoreboard with given maximum capacity.
6
7      All entries are initially None.
8      """
9      self._board = [None] * capacity          # reserve space for future scores
10     self._n = 0                              # number of actual entries
11
12   def __getitem__(self, k):
13     """Return entry at index k."""
14     return self._board[k]
15
16   def __str__(self):
17     """Return string representation of the high score list."""
18     return '\n'.join(str(self._board[j]) for j in range(self._n))
19
```

```
20   def add(self, entry):
21     """Consider adding entry to high scores."""
22     score = entry.get_score( )
23
24     # Does new entry qualify as a high score?
25     # answer is yes if board not full or score is higher than last entry
26     good = self._n < len(self._board) or score > self._board[-1].get_score( )
27
28     if good:
29       if self._n < len(self._board):         # no score drops from list
30         self._n += 1                         # so overall number increases
31
32       # shift lower scores rightward to make room for new entry
33       j = self._n - 1
34       while j > 0 and self._board[j-1].get_score( ) < score:
35         self._board[j] = self._board[j-1]    # shift entry from j-1 to j
36         j -= 1                               # and decrement j
37       self._board[j] = entry                 # when done, add new entry
```

MENAMBAHKAN ENTRI

- Metode paling menarik dalam kelas Scoreboard adalah add, yang bertanggung jawab menambahkan entri baru ke scoreboard. Tidak semua entri otomatis masuk ke daftar skor tinggi:
- Jika scoreboard belum penuh, entri baru akan langsung disimpan.
- Jika scoreboard sudah penuh, entri baru hanya disimpan jika lebih tinggi dari skor terendah (entri terakhir).
- Proses penambahan:
- Evaluasi kelayakan – Skor baru diperiksa apakah layak masuk scoreboard.
- Penyesuaian kapasitas – Jika scoreboard penuh, entri terendah dihapus untuk memberi ruang.
- Penyisipan di posisi tepat – Entri baru ditempatkan sesuai urutan (dari tertinggi ke terendah) dengan menggeser skor lebih rendah ke kanan.
- Implementasi mirip dengan metode insert pada list Python, tetapi tidak perlu menggeser entri None di akhir array.



- Implementasi teknis:
- Variabel j = self.n - 1 menunjukkan indeks tempat entri baru akan ditempatkan.
- Perulangan while (baris 34) memeriksa apakah entri di indeks j-1 memiliki skor lebih rendah daripada entri baru. Jika ya, entri digeser ke kanan dan j dikurangi, hingga posisi yang tepat ditemukan.

5.5.2 MENGURUTKAN URUTAN ALGORITMA INSERTION-SORT

1. Algoritma Insertion-Sort:
   - Metode pengurutan sederhana dengan cara menyisipkan elemen ke posisi yang tepat di dalam subarray terurut.
   - Langkah Utama:
     - Mulai dari elemen kedua (indeks 1).
     - Geser elemen ke kiri hingga posisinya sesuai dalam subarray terurut sebelumnya.
     - Ulangi untuk semua elemen hingga array terurut.
2. Pseudocode (Cuplikan Kode 5.9

Algoritma InsertionSort(A):
  Input: Array A dengan n elemen yang bisa dibandingkan
  Output: Array A terurut secara menaik
  untuk k dari 1 hingga n-1:
    Sisipkan A[k] ke posisi tepat di antara A[0], A[1], ..., A[k].

3. Implementasi Python (Cuplikan Kode 5.10):
   - Loop luar iterasi elemen dari indeks 1 hingga akhir.
   - Loop dalam menggeser elemen ke kiri hingga posisi tepat.
4. Kompleksitas Waktu:
   - Kasus Terburuk (array terbalik): $O(n^2)$ → setiap elemen harus digeser maksimal.
   - Kasus Terbaik (array hampir terurut): $O(n)$ → sedikit atau tidak ada geseran.

IMPLEMENTASI

```
1  def insertion_sort(A):
2    """Sort list of comparable elements into nondecreasing order."""
3    for k in range(1, len(A)):              # from 1 to n-1
4      cur = A[k]                           # current element to be inserted
5      j = k                                # find correct index j for current
6      while j > 0 and A[j−1] > cur:        # element A[j-1] must be after current
7        A[j] = A[j−1]
8        j −= 1
9      A[j] = cur                           # cur is now in the right place
```

Code Fragment 5.10: Python code for performing insertion-sort on a list.

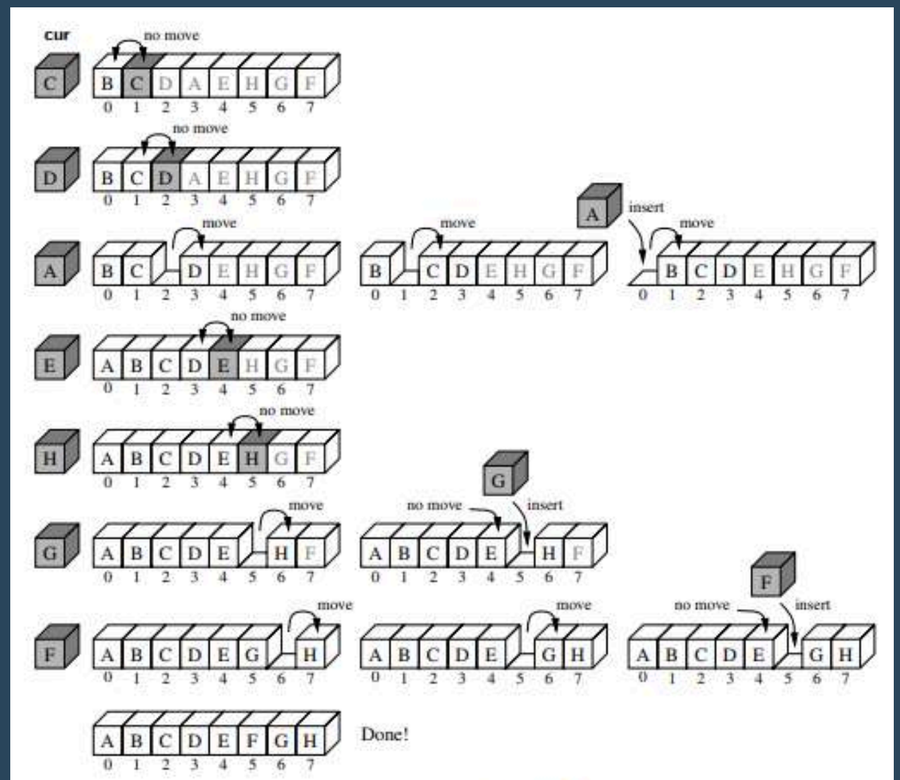


Figure 5.20: Execution of the insertion-sort algorithm on an array of eight characters. Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array, and shown as the cur value.

5.5.3 KRIPTOGRAFI SEDERHANA

Kriptografi adalah ilmu tentang pesan rahasia yang melibatkan enkripsi (mengubah teks biasa/plaintext menjadi teks acak/ciphertext) dan dekripsi (mengembalikan ciphertext ke plaintext). Salah satu skema enkripsi tertua adalah Caesar cipher, digunakan oleh Julius Caesar untuk mengamankan pesan militer.

Caesar Cipher bekerja dengan:

- Menggeser setiap huruf dalam alfabet sejumlah *r* posisi.
- Contoh: Jika *r* = 3, A → D, B → E, ..., Z → C (karena bersifat wrap-around).
- Rumus enkripsi: (indeks_huruf + r) mod 26.
- Dekripsi: (indeks_huruf - r) mod 26.

Implementasi Teknis:

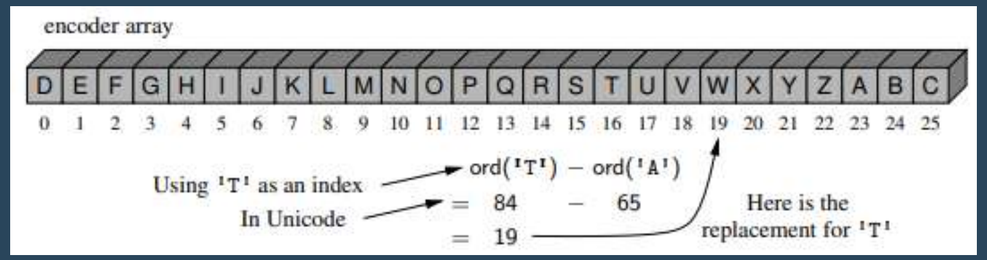
1. Konversi String ke List Karakter:
   - Karena string tidak bisa diubah (immutable), pesan diubah dulu menjadi list karakter (misal: list("bird") → ['b', 'i', 'r', 'd']).
   - Setelah dimodifikasi, list dikembalikan ke string dengan ''.join(['b', 'i', 'r', 'd']) → "bird".
2. Penggunaan Indeks Alfabet:
   - Huruf dianggap sebagai indeks array (A=0, B=1, ..., Z=25).
   - Operasi modulo (%) memastikan pergeseran melewati Z kembali ke A.

Kita dapat merepresentasikan aturan penggantian karakter dalam Caesar cipher menggunakan sebuah string terjemahan. Misalnya, untuk cipher dengan rotasi 3 karakter, string terjemahannya adalah DEFGHIJKLMNOPQRSTUVWXYZABC (A diganti D, B diganti E, dst).

Kunci implementasinya adalah:

1. Konversi karakter ke indeks numerik menggunakan fungsi ord() dan chr()
2. Menghitung posisi pengganti dengan rumus j = ord(c) - ord('A')
3. Menggunakan indeks untuk mengambil karakter pengganti dari string terjemahan

5.5.3 KRIPTOGRAFI SEDERHANA



Pada Code Fragment 5.11, diimplementasikan kelas Python untuk Caesar cipher dengan pergeseran (rotasi) bebas. Kelas ini mampu:

1. Membangun string terjemahan maju (enkripsi) dan mundur (dekripsi) berdasarkan nilai rotasi
2. Melakukan enkripsi dan dekripsi menggunakan metode utilitas privat _transform
3. Menghasilkan output seperti contoh:
   - Pesan terenkripsi: "WKH HDJOH LV LQ SODB; PHHW DW MRH'V."
   - Pesan terdekripsi: "THE EAGLE IS IN PLAY; MEET AT JOE'S."

```
1  class CaesarCipher:
2    """Class for doing encryption and decryption using a Caesar cipher."""
3
4    def __init__(self, shift):
5      """Construct Caesar cipher using given integer shift for rotation."""
6      encoder = [None] * 26                    # temp array for encryption
7      decoder = [None] * 26                    # temp array for decryption
8      for k in range(26):
9        encoder[k] = chr((k + shift) % 26 + ord('A'))
10       decoder[k] = chr((k - shift) % 26 + ord('A'))
11     self._forward = ''.join(encoder)         # will store as string
12     self._backward = ''.join(decoder)        # since fixed
13
14   def encrypt(self, message):
15     """Return string representing encripted message."""
16     return  self._transform(message, self._forward)
17
18   def decrypt(self, secret):
19     """Return decrypted message given encrypted secret."""
20     return  self._transform(secret, self._backward)
21
22   def _transform(self, original, code):
23     """Utility to perform transformation based on given code string."""
24     msg = list(original)
25     for k in range(len(msg)):
26       if msg[k].isupper( ):
27         j = ord(msg[k]) - ord('A')           # index from 0 to 25
28         msg[k] = code[j]                     # replace this character
29     return ''.join(msg)
30
31 if __name__ == '__main__':
32   cipher = CaesarCipher(3)
33   message = "THE EAGLE IS IN PLAY; MEET AT JOE'S."
34   coded = cipher.encrypt(message)
35   print('Secret: ', coded)
36   answer = cipher.decrypt(coded)
37   print('Message:', answer)
```

Code Fragment 5.11: A complete Python class for the Caesar cipher.

KUMPULAN DATA MULTIDIMENSI

STRUKTUR DATA MULTIDIMENSI DI PYTHON

1. Data Satu Dimensi vs. Multidimensi:
   - List, tuple, dan string di Python bersifat satu dimensi (menggunakan satu indeks).
   - Aplikasi seperti grafis, data geografis, atau analisis keuangan memerlukan struktur data dua dimensi (matriks) atau lebih.
2. Konsep Matriks (2D):
   - Matriks direpresentasikan dengan baris (indeks i) dan kolom (indeks j).
   - Indeks dimulai dari 0 (zero-indexed).
   - Contoh (Gambar 5.22):
     - Matriks stores dengan 8 baris dan 10 kolom.
     - stores[3][5] = 100 (baris 3, kolom 5).
     - stores[6][2] = 632 (baris 6, kolom 2).
3. Implementasi Matriks di Python:
   - Menggunakan list of lists (daftar berisi daftar).
   - Setiap elemen list utama merepresentasikan baris, dan list di dalamnya merepresentasikan kolom.
   - Contoh: data = [[22, 18, 709, 5, 33],[45, 32, 830, 120, 750],[4, 880, 45, 66, 61]]
   - Cara akses: data[1][3] → nilai di baris 1, kolom 3 = 120.
4. Keuntungan Representasi List of Lists:
   - Sintaks akses elemen intuitif: matrix[i][j].
   - Fleksibel untuk operasi baris/kolom.

MEMBANGUN LIST MULTIDIMENSI

Untuk membuat list satu dimensi berisi n nol, sintaks data = [0] * n dapat digunakan. Meskipun secara teknis ini membuat list dengan n referensi ke objek 0 yang sama, hal ini tidak menimbulkan masalah karena integer (int) di Python bersifat immutable (tidak dapat diubah). Namun, hati-hati saat membuat list multidimensi (list di dalam list). Misalnya, untuk membuat matriks r baris × c kolom berisi nol, beberapa pendekatan berikut salah:

- data = ([0] * c) * r

Ini menghasilkan list satu dimensi dengan panjang r * c (bukan list 2D), karena operasi perkalian hanya menggandakan elemennya.

- data = [[0] * c] * r

Meskipun menghasilkan list 2D, semua baris merujuk ke list yang sama. Jika satu elemen diubah (misal data[2][0] = 100), perubahan akan terlihat di semua baris karena aliasing.

Solusi yang benar adalah list comprehension. Gunakan list comprehension untuk memastikan setiap baris adalah list independen:

data = [[0] * c for _ in range(r)]

- [0] * c membuat satu baris berisi c nol.
- for _ in range(r) memastikan ekspresi [0] * c dievaluasi ulang untuk setiap baris, sehingga tidak ada aliasing.

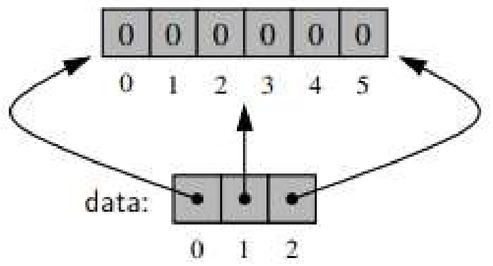
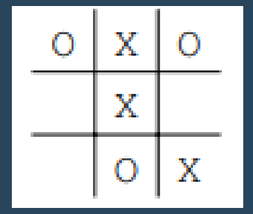


Figure 5.23: A flawed representation of a 3 × 6 data set as a list of lists, created with the command data = [[0] * 6] * 3. (For simplicity, we overlook the fact that the values in the secondary list are referential.)

ARRAY DUA DIMENSI DAN PERMAINAN POSISIONAL

1. Banyak permainan komputer—seperti strategi, simulasi, atau FPS—melibatkan objek yang berada dalam ruang dua dimensi. Untuk merepresentasikan "papan" permainan ini, Python menggunakan list of lists (list bersarang) sebagai struktur data yang alami.
2. Contoh: Permainan Tic-Tac-Toe
3. Tic-Tac-Toe adalah permainan sederhana di papan 3×3 di mana dua pemain (X dan O) bergantian menempatkan tanda mereka. Pemain menang jika berhasil membuat tiga tanda sejajar (mendatar, vertikal, atau diagonal).
4. Representasi Papan:
5. Papan direpresentasikan sebagai list of lists berisi karakter:
6. 'X' atau 'O' untuk tanda pemain.
7. ' ' (spasi) untuk kotak kosong.

[['O', 'X', 'O'],[' ', 'X', ' '],[' ', 'O', 'X']]



- Implementasi Kelas TicTacToe
- Sebuah kelas Python dapat dibuat untuk mengelola:
- Inisialisasi papan kosong.
- Metode mark(i, j) untuk menempatkan tanda

di posisi (i, j) dengan pengecekan:

  - Indeks valid.
  - Posisi belum terisi.
  - Permainan belum dimenangkan.
- Penentuan pemenang (cek tiga sejajar).
- Alternasi giliran pemain (X dan O).

```
1   game = TicTacToe( )
2   # X moves:              # O moves:
3   game.mark(1, 1);        game.mark(0, 2)
4   game.mark(2, 2);        game.mark(0, 0)
5   game.mark(0, 1);        game.mark(2, 1)
6   game.mark(1, 2);        game.mark(1, 0)
7   game.mark(2, 0)
8
9   print(game)
10  winner = game.winner( )
11  if winner is None:
12    print('Tie')
13  else:
14    print(winner, 'wins')
```

Code Fragment 5.12: A simple test for our Tic-Tac-Toe class.

ARRAY DUA DIMENSI DAN PERMAINAN POSISIONAL

```
1  class TicTacToe:
2    """Management of a Tic-Tac-Toe game (does not do strategy)."""
3
4    def __init__(self):
5      """Start a new game."""
6      self._board = [ [' '] * 3 for j in range(3) ]
7      self._player = 'X'
8
9    def mark(self, i, j):
10     """Put an X or O mark at position (i,j) for next player's turn."""
11     if not (0 <= i <= 2 and 0 <= j <= 2):
12       raise ValueError('Invalid board position')
13     if self._board[i][j] != ' ':
14       raise ValueError('Board position occupied')
15     if self.winner( ) is not None:
16       raise ValueError('Game is already complete')
17     self._board[i][j] = self._player
18     if self._player == 'X':
19       self._player = 'O'
20     else:
21       self._player = 'X'
22
```

```
23   def _is_win(self, mark):
24     """Check whether the board configuration is a win for the given player."""
25     board = self._board                          # local variable for shorthand
26     return (mark == board[0][0] == board[0][1] == board[0][2] or     # row 0
27             mark == board[1][0] == board[1][1] == board[1][2] or     # row 1
28             mark == board[2][0] == board[2][1] == board[2][2] or     # row 2
29             mark == board[0][0] == board[1][0] == board[2][0] or     # column 0
30             mark == board[0][1] == board[1][1] == board[2][1] or     # column 1
31             mark == board[0][2] == board[1][2] == board[2][2] or     # column 2
32             mark == board[0][0] == board[1][1] == board[2][2] or     # diagonal
33             mark == board[0][2] == board[1][1] == board[2][0])       # rev diag
34
35   def winner(self):
36     """Return mark of winning player, or None to indicate a tie."""
37     for mark in 'XO':
38       if self._is_win(mark):
39         return mark
40     return None
41
42   def __str__(self):
43     """Return string representation of current game board."""
44     rows = ['|'.join(self._board[r]) for r in range(3)]
45     return '\n-----\n'.join(rows)
```

TERIMAKASIH

ATAS PERHATIANNYA