Modul 6

Pengurutan lanjutan

Di modul sebelumnya kita sudah mengenal beberapa algoritma pengurutan seperti selection sort dan insertion sort. Algoritma-algoritma itu cukup sederhana dan intuitif, namun masih dirasa belum cukup cepat oleh para ahli. Riset di bidang ini terus dilakukan. Algoritma pengurutan yang menjadi bawaan suatu bahasa pemrograman pun berubah-ubah.

Di modul ini akan kita pelajari pengurutan yang sedikit lebih *advanced*, yakni *merge sort* dan *quick sort*. Sebelumnya akan kita tinjau sebuah problem yang menarik, yakni menggabungkan dua list yang sudah urut.

6.1 Menggabungkan dua list yang sudah urut

Terkadang kita perlu menggabungkan dua buah list yang sudah urut menjadi sebuah list baru. Perhatikan kode-kodel berikut

```
P = [2, 8, 15, 23, 37]
Q = [4, 6, 15, 20]
R = gabungkanDuaListUrut(P, Q)
print(R)
```

yang membuat dua list –yang sudah urut kecil ke besar– lalu memanggil sebuah fungsi buatan sendiri. Fungsi ini mengembalikan sebuah list baru yang dibuat dengan menggabungkan dua list tadi. Ketika dicetak akan muncul

```
[2, 4, 6, 8, 15, 15, 20, 23, 37]
```

Bagaimanakah algoritmanya? Bisa saja kita langsung menggabungkan keduanya dan lalu memanggil sebuah algoritma yang sudah kita buat sebelumnya[1]. Namun ini berarti kita tidak memanfaatkan kenyataan bahwa kedua list ini sudah urut. Kita bisa membuat algoritma yang lebih baik.

[1] Misal seperti ini:

```
def gabungkanDuaListUrut(A, B):
    C = A + B; insertionSort(C); return C
```

Misalkan kita akan mempunyai dua tumpukan lembar jawaban ujian yang masing-masingnya sudah urut NIM. Bagaimanakah menggabungkannya? Pertama lihat kertas yang paling atas di dua tumpukan itu. Bandingkan mana yang paling kecil NIM-nya. Ambil kertas yang paling kecil NIM-nya itu lalu taruh di meja secara terbalik untuk mulai membuat tumpukan baru. Lalu kembali lihat kedua tumpukan tadi. Bandingkan kedua NIM di kertas itu. ...

Demikian seterusnya. Akan ada suatu saat di mana salah satu tumpukan sudah habis sedangkan tumpukan yang satunya masih ada. Dalam hal ini, kita tinggal menumpukkan sisa tadi ke tumpukan yang baru. Berikut ini adalah salah satu implementasinya di Python.

```
def gabungkanDuaListUrut(A, B):
    la = len(A); lb = len(B)
    C = list()      # C adalah list baru
    i = 0; j = 0

    # Gabungkan keduanya sampai salah satu kosong
    while i < la and j < lb:
        if A[i] < B[j]:
            C.append(A[i])
            i += 1
        else:
            C.append(B[j])
            j += 1

    while i < la:       # Jika A mempunyai sisa
        C.append(A[i])  #   tumpukkan ke list baru itu
        i += 1          #   satu demi satu

    while j < lb:       # Jika B mempunyai sisa
        C.append(B[j])  #   tumpukkan ke list baru itu
        j += 1          #   satu demi satu

    return C
```

Cobalah melarikan program di atas untuk semua kemungkinan penggunaan. Perlu dicatat bahwa program di atas bukan satu-satunya cara untuk menyelesaikan problemnya. Banyak cara yang lain, namun yang ini dipilih sebagai *prelude* untuk algoritma *merge sort* di bawah.

6.2 Merge sort

Algoritma merge sort memakai strategi *divide and conquer*, bagi dan taklukkan. Algoritma ini adalah algoritma *recursive* yang secara terus menerus membelah (men-*split*) sebuah list menjadi dua. (Kotak di halaman 57 memberikan penyegaran tentang fungsi rekursif.) Jika list-nya kosong atau hanya berisi satu elemen, maka *by definition* dia sudah urut (ini adalah *base case*-nya).

Jika list-nya mempunyai item lebih dari satu, kita membelah list-nya dan secara rekursif memanggil `mergeSort` di keduanya. *Kalau* kedua paruh –*kanan* dan kiri– *sudah* urut, *operasi* penggabungan (*merge*) dilakukan. Proses penggabungan adalah proses mengambil dua list urut yang lebih kecil dan menggabungkan mereka menjadi satu list baru yang urut. Lihat

Penyegaran: Fungsi Rekursif

Seperti sudah kamu pelajari sebelumnya, fungsi rekursif[a] adalah fungsi yang memanggil dirinya sendiri. Sebagai contoh, misal kita diminta menghitung 7! (tujuh faktorial). Dari definisi, kita akan menghitungnya sebagai

$$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \quad (6.1)$$

Tapi perhatikan bahwa ini bisa juga ditulis sebagai

$$7! = 7 \times 6! \quad (6.2)$$

Tapi 6! juga bisa ditulis sebagai

$$6! = 6 \times 5! \quad (6.3)$$

Dan demikian seterusnya. Kapankah ini berakhir? Yakni saat mencapai angka 1, di mana,

$$1! = 1 \quad (6.4)$$

Sehingga ini merupakan *base case* (kejadian dasar) untuk fungsi faktorial kita. Dari sini kita tahu bahwa fungsi rekursif mempunyai tiga unsur:

- kejadian dasar (*base case*)
- kejadian rekursif (*recursive case*)
- hal yang memastikan bahwa semua *recursive cases* akan ter-reduksi ke *base case*[b].

Dari sini kita bisa membuat program faktorial di atas sebagai berikut.

```
def faktorial(a):
    if a==1:          ## base case
        return 1
    else:             ## recursive case
        return a*faktorial(a-1)
```

Perhatikan bahwa kita mempunyai kejadian dasar, kejadian rekursif, dan hal yang memastikan bahwa program ini akan konvergen. Fungsi rekursif adalah *fungsi yang memanggil dirinya sendiri*[c]. Cobalah jalankan program di atas dan periksa hasilnya.

[a] Bahasa Inggris: *recursive function*.

[b] Kalau tidak, program akan divergen, "meledak", dan tidak berakhir. Misal kalau tanda minus pada program faktorial di atas diganti dengan tanda plus.

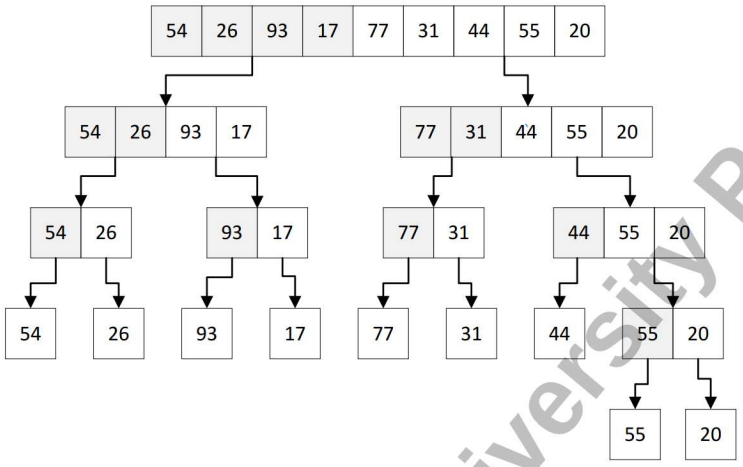
[c] Untuk paham fungsi rekursif, terlebih dahulu kamu harus paham fungsi rekursif. :-)

kembali halaman-halaman seblumnya.

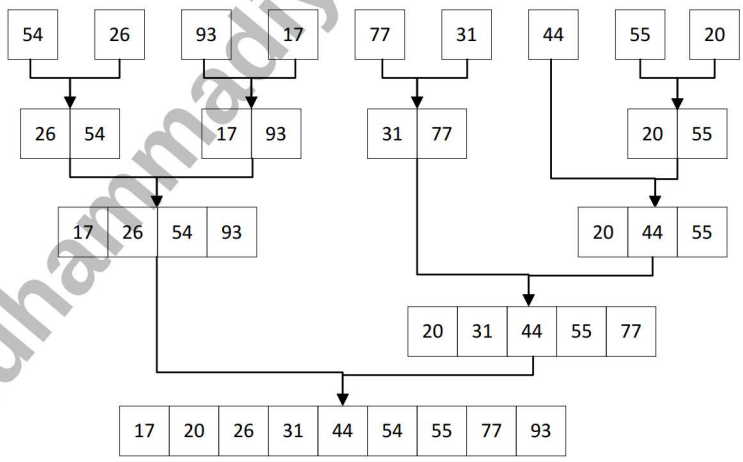
Gambar 6.1 menunjukkan[2] sebuah list yang dibelah-belah. Gambar 6.2 menunjukkan penggabungan list itu sampai urut. Perhatikan bahwa jumlah elemennya tidak harus pangkat dua (2, 4, 8, 16, ...) dan dengan demikian jumlah elemen kiri dan kanan pada suatu saat bisa berbeda. Ini tidak menjadi masalah.

Program merge sort dapat dilihat di bawah. Program ini mulai dengan menanyakan pertanyaan *base case*. Jika panjangnya list kurang dari atau sama dengan satu, kita berarti sudah mempunyai sebuah list yang urut dan tidak perlu pemrosesan lagi. Tapi, sebaliknya, jika panjang list itu lebih dari satu, kita lalu menggunakan operasi *slice* di Python untuk meng-ekstrak separuh kiri dan kanan. Perlu dicatat bahwa list-nya bisa jadi tidak mempunyai jumlah elemen yang genap. Ini tidaklah mengapa karena perbedaan jumlah elemen antara separuh kanan dan separuh kiri paling banyak satu.

[2] Gambar-gambar di bab ini diambil dari *Problem Solving with Algorithms and Data Structures Using Python*, oleh Bradley Miller dan David Ranum, penerbit Franklin, Beedle & Associates, 2011. Tersedia versi online-nya di `http://interactivepython.org/runestone/static/pythonds/index.html`



Gambar 6.1: Membelah list sampai tiap sub-list berisi satu elemen atau kosong. Sesudah itu digabung seperti ditunjukkan di Gambar 6.2.



Gambar 6.2: Menggabungkan list satu demi satu.

```
1  def mergeSort(A):
2      #print("Membelah     ", A)
3      if len(A) > 1:
4          mid = len(A) // 2       # Membelah list.
5          separuhKiri = A[:mid]   # Slicing ini langkah yang expensive sebenarnya,
6          separuhKanan = A[mid:]  #   bisakah kamu membuatnya lebih baik?
7
8          mergeSort(separuhKiri)  # Ini rekursi. Memanggil lebih lanjut mergeSort
9          mergeSort(separuhKanan) #   untuk separuhKiri dan separuhKanan.
10
11         # Di bawah ini adalah proses penggabungan.
12         i=0  ;  j=0  ;  k=0
13         while i < len(separuhKiri) and j < len(separuhKanan):
14             if separuhKiri[i] < separuhKanan[j]:  # while-loop ini
15                 A[k] = separuhKiri[i]             #   menggabungkan kedua list, yakni
16                 i = i + 1                         #   separuhKiri dan separuhKanan,
17             else:                                 #   sampai salah satu kosong.
18                 A[k] = separuhKanan[j]            # Perhatikan kesamaan strukturnya
19                 j = j + 1                         #   dengan proses penggabungan
20             k=k+1                                 #   dua list urut.
21
22         while i < len(separuhKiri):   # Jika separuhKiri mempunyai sisa
23             A[k] = separuhKiri[i]     #   tumpukkan ke A
24             i = i + 1                 #   satu demi satu.
25             k = k + 1                 #
26
27         while j < len(separuhKanan):  # Jika separuhKanan mempunyai sisa
28             A[k] = separuhKanan[j]    #   tumpukkan ke A
29             j = j + 1                 #   satu demi satu.
30             k = k + 1
31     #print("Menggabungkan", A)
```

Larikan program di atas dengan memanggilnya seperti ini

```
alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

Setelah fungsi mergeSort ini dipanggil pada separuh kiri dan separuh kanan (baris 8–9), kita anggap list itu semua sudah diurutkan[3]. Baris-baris selanjutnya (baris 12–30) bertugas menggabungkan dua list yang lebih kecil untuk menjadi list yang lebih besar. Perhatikan bahwa operasi penggabungan menempatkan kembali elemen-elemen ke list yang asli satu demi satu dengan secara berulang mengambil elemen terkecil dari list yang sudah urut[4].

Fungsi `mergeSort` di atas telah dilengkapi sebuah perintah `print` (baris 2) untuk menunjukkan isi list yang sedang diurutkan di awal tiap pemanggilan. Terdapat pula sebuah perintah `print` (baris 31) untuk memperlihatkan proses penggabungan. Aktifkan baris-baris itu[5] dan jalankan programnya. Hasilnya kurang lebih adalah sebagai berikut.

[3] Ingat, ini adalah fungsi rekursif.

[4] Inilah kegunaan index k di program itu. Lihat bahwa struktur program penggabungan ini (baris 12–30) sama dengan program kita di halaman 56, kecuali bahwa di sini kita tidak membuat list baru dan ada index k. List yang kita diminta mengurutkan adalah juga list tempat kita menaruh hasil pengurutannya.

[5] Yakni, hilangkan tanda # di baris-baris itu. Jangan lupa untuk memberi kembali tanda itu saat tes kecepatan nanti.

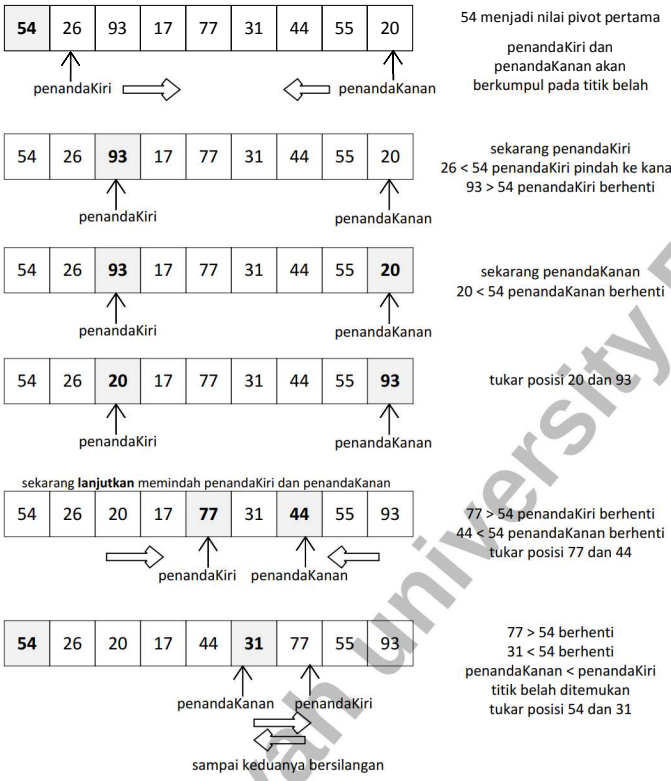
```
Membelah      [54, 26, 93, 17, 77, 31, 44, 55, 20]
Membelah      [54, 26, 93, 17]
Membelah      [54, 26]
Membelah      [54]
Menggabungkan [54]
Membelah      [26]
Menggabungkan [26]
Menggabungkan [26, 54]
Membelah      [93, 17]
Membelah      [93]
Menggabungkan [93]
Membelah      [17]
Menggabungkan [17]
Menggabungkan [17, 93]
Menggabungkan [17, 26, 54, 93]
Membelah      [77, 31, 44, 55, 20]
Membelah      [77, 31]
Membelah      [77]
Menggabungkan [77]
Membelah      [31]
Menggabungkan [31]
Menggabungkan [31, 77]
Membelah      [44, 55, 20]
Membelah      [44]
Menggabungkan [44]
Membelah      [55, 20]
Membelah      [55]
Menggabungkan [55]
Membelah      [20]
Menggabungkan [20]
Menggabungkan [20, 55]
Menggabungkan [20, 44, 55]
Menggabungkan [20, 31, 44, 55, 77]
Menggabungkan [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Bisa dilihat bahwa proses pembelahan pada akhirnya menghasilkan sebuah list yang bisa langsung digabungkan dengan list lain yang urut.

Latihan 6.1 Memakai bolpen merah atau biru, tandai dan beri nomer urut eksekusi proses pada Gambar 6.1 dan 6.2, dengan mengacu pada output di atas[6]. □

Merge sort ini secara umum bekerja lebih cepat dari insertion sort ataupun selection sort. (Dapatkah kamu melihat mengapa demikian?) Namun program di atas masih dapat ditingkatkan efisiensinya. Di program itu kita masih menggunakan operator *slice* (seperti `A[:mid]` dan `A[mid:]`), yang cukup memakan waktu. Kita meningkatkan efisiensi program dengan tidak memakai operator slice ini, dan lalu mem-*pass* index awal dan index akhir bersama list-nya saat kita memanggilnya secara rekursif. *Kamu dapat mengerjakannya sebagai latihan. Kamu perlu memisah fungsi* `mergeSort` *di atas menjadi beberapa fungsi, mirip halnya dengan apa yang dilakukan algoritma Quick Sort di bawah.*

[6]Jika kamu membaca modul ini dari file pdf-nya, maka cetaklah halaman-halaman yang relevan, atau kamu dapat menulis ulang dua gambar itu.



Gambar 6.3: ***Quick sort.*** Mencari titik belah untuk 54. Tujuan langkah ini adalah membuat semua elemen yang lebih kecil dari 54 berada di sebelah kiri dan semua elemen yang lebih besar dari 54 berada di sebelah kanan (meskipun di dalamnya sendiri belum urut). Angka 54 sendiri akan sudah berada di tempat yang tepat. Ini bisa dilihat di gambar selanjutnya, yaitu Gambar 6.4.

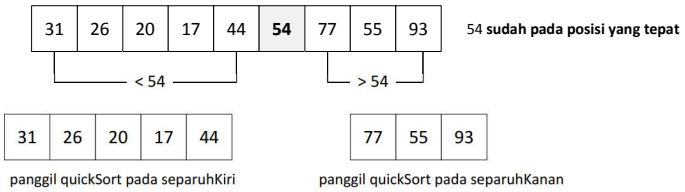
6.3 Quick sort

Algoritma quick sort memakai 'belah dan taklukkan' untuk mendapatkan keuntungan yang sama dengan merge sort, dengan tidak memerlukan penyimpanan tambahan. Harga yang harus dibayar adalah bahwa ada kemungkinan list itu tidak dibelah sama panjang.

Sebuah quicksort pertama-tama memilih sebuah nilai di antara nilai-nilai yang ada di list, yang disebut **nilai pivot**. Banyak cara untuk memilih nilai pivot itu[7]. Kita saat ini memilih nilai yang paling kiri (berarti elemen pertama). Peran nilai pivot ini adalah untuk membantu dalam proses membelah list ini.

Sesudah itu kita membuat semua elemen yang lebih kecil dari pivot berada di sebelah kiri dan semua elemen yang lebih besar dari pivot berada di sebelah kanan, serta pivotnya sendiri berada di antar dua sub-list ini. Pada contoh di Gambar 6.3, elemen pivot kita adalah $\boxed{54}$.

[7]Misal: yang paling kiri, yang paling kanan, secara acak, atau median-dari-tiga



Gambar 6.4: Keadaan setelah Gambar 6.3. Sekarang semua elemen di sebelah kiri 54 adalah lebih kecil dari 54, dan semua elemen di sebelah kanan 54 adalah lebih besar dari 54 (meskipun di masing-masingnya sendiri belum urut). Dengan demikian ***54 sudah menempati posisi yang tepat di list itu***. Lalu, secara rekursif kedua paruh kiri dan kanan kembali di-*feed* ke algoritma quick sort.

Kita mulai dengan menaikkan penandaKiri satu demi satu sampai kita menemui sebuah nilai yang lebih besar dari pivot. Kita lalu menurunkan penandaKanan sampai kita menemukan sebuah nilai yang lebih kecil dari pivot. Pada saat ini kita telah menemukan dua item yang posisinya salah (dilihat dari titik belahnya). Untuk contoh yang sedang kita bahas di Gambar 6.3, ini adalah 93 dan 20. Sekarang kita dapat menukar posisi dua elemen ini dan prosesnya diulang kembali.

Pada saat di mana penandaKanan menjadi lebih kecil dari penandaKiri, kita berhenti. Posisi penandaKanan sekarang adalah titik belahnya. Nilai pivot dapat ditukar posisinya dengan isi titik belahnya dan sekarang nilai pivotnya sudah berada pada posisi yang tepat (Gambar 6.4). Semua item di sebelah titik belah adalah lebih kecil dari nilai pivotnya dan semua item di sebelah kanan adalah lebih besar dari nilai pivotnya. List ini sekarang dapat dibagi dua pada titik belah itu dan lalu algoritma quick sort dapat kembali dipanggil secara rekursif pada kedua belahan itu.

Fungsi `quickSort` pada kode di bawah ini memanggil fungsi rekursif `quickSortBantu`. Fungsi `quickSortBantu` mulai dengan base case yang sama dengan merge sort (lihat kode `mergeSort` di halaman 59). Jika panjang list-nya kurang dari atau sama dengan satu, berarti sudah terurut. Jika lebih besar dari satu, maka list itu dapat dipartisi dan diurutkan secara rekursif. Fungsi `partisi` mengimplementasikan proses yang dijelaskan sebelumnya.

Fungsi `quickSort` adalah interface ke luar:

```
1 def quickSort(A):
2     quickSortBantu(A, 0, len(A) - 1 ) # memanggil quickSortBantu di
3
```

Fungsi `quickSortBantu` adalah fungsi rekursif yang dipanggil berulang-ulang:

```
4 def quickSortBantu(A, awal, akhir):
5     if awal < akhir:
6         titikBelah = partisi(A, awal, akhir)      # Atur elemen dan dapatkan titikBelah.
7         quickSortBantu(A, awal, titikBelah - 1)  # Ini rekursi untuk belah sisi kiri
8         quickSortBantu(A, titikBelah + 1, akhir) #   dan belah sisi kanan.
9
```

Fungsi `partisi` adalah untuk mencari titik belah dan mengatur ulang posisi elemen-elemen

agar menyesuaikan terhadap `nilaiPivot`:

```
10 def partisi(A, awal, akhir):
11     nilaiPivot = A[awal] # Di sini nilaiPivot kita ambil dari elemen yang paling kiri.
12
13     penandaKiri = awal + 1 # Posisi awal penandaKiri. Lihat Gambar 6.3.
14     penandaKanan = akhir   # Posisi awal penandaKanan.
15
16     selesai = False
17     while not selesai: # loop di bawah adalah untuk mengatur ulang posisi semua elemen
18
19         while penandaKiri <= penandaKanan and \   # penandaKiri bergerak ke kanan,
20                 A[penandaKiri] <= nilaiPivot:     #  sampai ketemu suatu nilai yang
21             penandaKiri = penandaKiri + 1         #  lebih besar dari nilaiPivot
22
23         while A[penandaKanan] >= nilaiPivot and \ # penandaKanan bergerak ke kiri,
24                 penandaKanan >= penandaKiri:      #  sampai ketemu suatu nilai yang
25             penandaKanan = penandaKanan - 1       #  lebih kecil dari nilaiPivot
26
27         if penandaKanan < penandaKiri:            # Kalau dua penanda sudah bersilangan,
28             selesai = True                        #  selesai & lanjut ke penempatan pivot
29         else:
30             temp = A[penandaKiri]                 # Tapi kalau belum bersilangan,
31             A[penandaKiri] = A[penandaKanan]      #  tukarlah isi yang ditunjuk oleh
32             A[penandaKanan] = temp                #  penandaKiri dan penandaKanan
33
34     temp = A[awal]               # Kalau acara tukar menukar posisi sudah selesai,
35     A[awal] = A[penandaKanan]    #  kita lalu menempatkan pivot pada posisi yang tepat,
36     A[penandaKanan] = temp       #  yakni posisi penandaKanan. Lihat Gambar 6.3 dan 6.4.
37                                  #  Posisi penandaKanan adalah juga titikBelah.
38     return penandaKanan     # Fungsi ini mengembalikan titikBelah ke pemanggil
```

Sebelumnya telah kita bahas bahwa pemilihan pivot bisa memakai berbagai cara. Khususnya, kita bisa mencoba menghindari pembelahan yang berpotensi tidak seimbang dengan menggunakan teknik median-dari-tiga. Untuk memilih nilai pivot, kita melihat elemen pertama, tengah, dan terakhir[8]. Untuk contoh kita di atas ini adalah 54, 77, 20. Dari ketiganya, pilih median-nya. Dalam contoh kita ini berarti 54. (Kebetulan ini juga yang kita pilih sebagai pivot awal di contoh kita.)

Ide dasarnya adalah ketika ada suatu kejadian di mana elemen paling kiri ternyata nilainya 'tidak cukup tengah', metode median-dari-tiga ini akan lebih baik dalam memilihkan 'nilai tengah'. Ini bisa berguna saat list awalnya sudah agak urut. Sebagai latihan, *ubahlah kode di atas untuk mengaplikasikan pemilihan pivot memakai metode median-dari-tiga.*

6.4 Soal-soal untuk Mahasiswa

Sebelum mengerjakan soal-soal di bawah, kerjakan dulu latihan-latihan di atas.

1. Ubahlah kode `mergeSort` dan `quickSort` di atas agar bisa mengurutkan list yang berisi object-object `mhsTIF` yang sudah kamu buat di Modul 2. Uji programmu secukupnya.

[8] Atau, dengan kata lain, elemen paling kiri, tengah, dan paling kanan.

2. Memakai bolpen merah atau biru, tandai dan beri nomer urut eksekusi proses pada Gambar 6.1 dan 6.2, dengan mengacu pada output di halaman 59.

3. Uji kecepatan. Ujilah `mergeSort` dan `quickSort` di atas (bersama metode sort yang kamu pelajari sebelumnya) dengan kode di bawah ini.

```
from time import time as detak
from random import shuffle as kocok
import time
k = range(6000)
kocok(k)
u_bub = k[:] ##
u_sel = k[:] ##  Deep copy.
u_ins = k[:] ##  Jangan lupa [:]-nya!
u_mrg = k[:] ##
u_qck = k[:] ##

aw=detak();bubbleSort(u_bub);ak=detak();print('bubble: %g detik' %(ak-aw) );
aw=detak();selectionSort(u_sel);ak=detak();print('selection: %g detik' %(ak-aw) );
aw=detak();insertionSort(u_ins);ak=detak();print('insertion: %g detik' %(ak-aw) );
aw=detak();mergeSort(u_mrg);ak=detak();print('merge: %g detik' %(ak-aw) );
aw=detak();quickSort(u_qck);ak=detak();print('quick: %g detik' %(ak-aw) );
```

   Tunjukkan hasil ujinya ke asisten praktikum[9].

4. Diberikan list `L = [80, 7, 24, 16, 43, 91, 35, 2, 19, 72]`, gambarlah *trace* pengurutan[10] untuk algoritma

   (a) merge sort

   (b) quick sort

Soal-soal di bawah ini sedikit lebih sulit. Kerjakanlah di rumah.

5. Tingkatkan efisiensi program `mergeSort` dengan *tidak* memakai operator slice (seperti `A[:mid]` dan `A[mid:]`), dan lalu mem-*pass* index awal dan index akhir bersama list-nya saat kita memanggil `mergeSort` secara rekursif. Kamu akan perlu memisah fungsi `mergeSort` itu menjadi beberapa fungsi, mirip halnya dengan apa yang dilakukan algoritma quick sort.

6. Apakah kita bisa meningkatkan efisiensi program `quickSort` dengan memakai metode median-dari-tiga untuk memilih pivotnya? Ubahlah kodenya dan ujilah.

7. Uji-kecepatan keduanya dan perbandingkan juga dengan kode awalnya.

8. Buatlah versi linked-list untuk program `mergeSort` di atas.

[9] Ekspektasi hasil: `mergeSort` dan `quickSort` hampir sama cepat, kedunya sekitar 20 kali lebih cepat dari `selectionSort` dan `insertionSort`, dan sekitar 100 kali lebih cepat dari `bubbleSort`.

[10] Kurang lebih sama dengan yang dimaksud soal nomer 2 di atas.