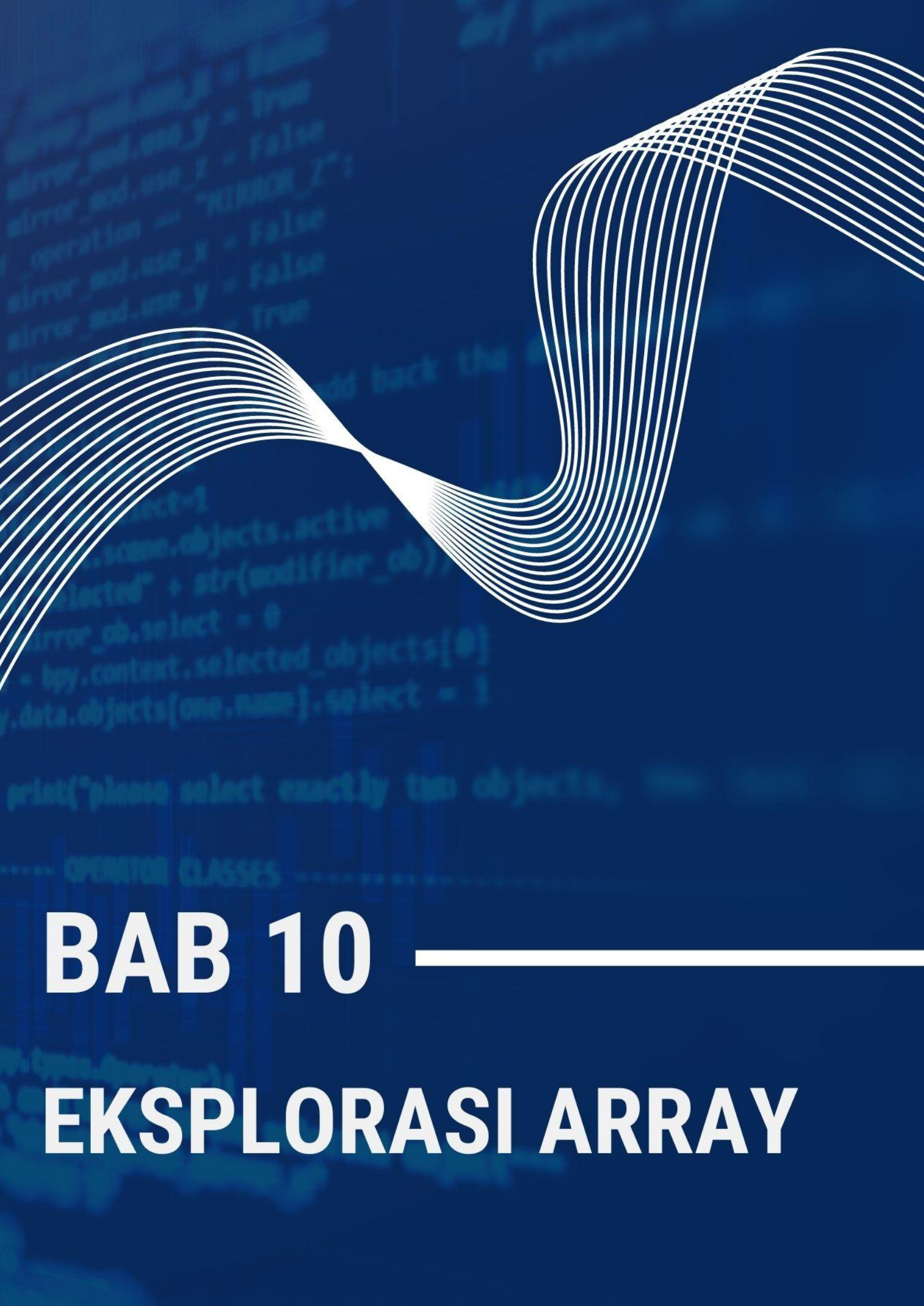


BAB 10

EKSPLORASI ARRAY

BAB 10
EKSPLORASI ARRAY

Capaian Pembelajaran

Setelah mempelajari materi dalam bab ini, mahasiswa diharapkan mampu menganalisis, menyampaikan pendapat, dan mengoperasikan Eksplorasi Array algoritma pemrograman pada sebuah kasus operasional bisnis.

Pokok Bahasan

1. Algoritma Pencarian pada Array
2. Implementasi Algoritma Pencarian pada Array
3. Algoritma Pengurutan pada Array
4. Implementasi Algoritma Pengurutan pada Array

Evaluasi Pembelajaran

Soal Latihan Eksplorasi Array

Pre Test
Eksplorasi Array

1. Apa yang kamu ketahui tentang algoritma pencarian pada array?
2. Bagaimana contoh implementasi dari algoritma pencarian pada array?
3. Apa yang kamu ketahui tentang algoritma pengurutan pada array?
4. Bagaimana contoh implementasi dari algoritma pengurutan pada array?

Pembahasan pada bab kesepuluh dari buku ini adalah tentang eksplorasi array sebagai lanjutan dari bab array yang telah kita pelajari sebelumnya. Pada bab ini kita akan membahas terkait dengan dua algoritma pada array yang nantinya akan berguna ketika kita melakukan praktik pemrograman. Dua algoritma tersebut adalah algoritma pencarian dan pengurutan. Selain itu kita juga akan mempelajari terkait implementasi dalam penggunaannya. Jadi tidak hanya terbatas pada teori atau ilmu hafalan.

Setelah memahami konsep teori beserta implementasi dari algoritma pencarian dan pengurutan pada array, kita akan mencoba mencari solusi dari permasalahan atau kendala yang ada dari latihan soal yang diberikan. Hal tersebut akan membantu kita untuk lebih memahami lebih dalam terkait dengan bab eksplorasi array.

10.1 Algoritma Pencarian pada Array

Pencarian adalah proses yang penting dalam melakukan pengolahan data. Proses pencarian merupakan menemukan nilai atau data tertentu di dalam kumpulan data yang memiliki tipe yang sama (baik yang bertipe dasar atau yang bertipe bentukan). Sebagai contoh, untuk melakukan update pada suatu data, langkah awal yang dilakukan adalah menemukan lokasi data tersebut di dalam kumpulannya. Jika sudah menemukan data yang dicari, maka kita dapat mengubah nilai dari data tersebut dengan data yang baru. Aktivitas awal yang sama juga dilakukan pada proses penambahan (insert) data baru, Proses penambahan data dimulai dengan rnencari apakah data yang akan ditambahkan sudah terdapat di dalam kumpulan. Jika sudah ada dan mengasumsikan tidak boleh ada duplikasi data maka data tersebut tidak perlu ditambakan, tetapi jika belum ada, maka tambahkan.

Dalam bab ini kita akan mempelajari algoritma pencarian data di dalam larik. Algoritma pencarian yang akan dibicarakan dimulai dengan algoritma pencarian yang paling sederhana (yaitu pencarian beruntun atau sequential search) hingga algoritma pencarian yang lebih maju yaitu pencarian bagidua (binary search).

Pertama-tama kita perlu menspesifikasi masalah pencarian di dalam larik. Di dalam bab ini kita mendefinisikan persoalan pencarian secara umum sebagai berikut: Diberikan larik L yang sudah terdefinisi elemen-elemennya, dan x adalah elemen yang bertipe sama dengan elemen larik L. Carilah x di dalam larik L.

Hasil atau keluaran dari persoalan pencarian dapat bermacam-macam, bergantung pada spesifikasi rinci dari persoalan tersebut, misalnya:

a. Pencarian hanya untuk memeriksa keberadaan x. Keluaran yang diinginkan misalnya pesan (message) bahwa x ditemukan atau tidak ditemukan di dalam larik.

   Contoh:

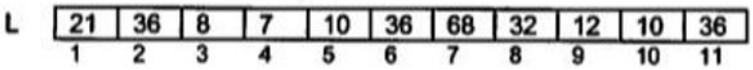
   ```
   write('ditemukan!')
   ```

   atau

   ```
   write('tidak ditemukan!')
   ```

b. Hasil pencarian adalah indeks elemen larik. Jika x ditemukan, maka indeks elemen larik tempat x berada diisikan ke dalam idx. Jika x tidak terdapat di dalam larik L, maka idx diisi dengan harga khusus, misalnya -1.

   Contoh: Perhatikan larik di bawah ini:

   Misalkan x = 68, maka idx = 7, dan bila x = 100, maka idx = -1.

c. Hasil pencarian adalah sebuah nilai boolean yang menyatakan status hasil pencarian. Jika s ditemukan, maka sebuah peubah bertipe boolean, misalnya "ketemu", diisi dengan nilai true. sebaliknya "ketemu" diisi dengan nilai false. Hasil pencarian ini selanjutnya disimpulkan pada bagian pemanggilan prosedur.

   Contoh: Perhatikan larik L di atas:

   Misalkan x = 68, maka ketemu = true, dan bila x = 100, maka ketemu = false.

Untuk kedua macam keluaran b dan c di atas, kita mengonsultasi basil pencarian setelah proses pencarian selesai dilakukan, bergantung pada kebutuhan. Misalnya menampilkan pesan bahwa x ditemukan (atau tidak ditemukan) atau memanipulasi nilai x.

Contoh:

```
(1)   if idx ≠ -1 then { x ditemukan }
            L(idx) ← L(idx) + 1 { manipulasi nilai x }
      endif

(2)   if ketemu then {yaitu, ketemu = true}
            write(x, 'tidak ditemukan')
      else
            write(x, 'ditemukan')
      endif
```

Hal lain yang harus diperjelas dalam masalah pencarian adalah mengenai duplikasi data. Apabila X yang dicari terdapat lebih dari satu banyaknya di dalam larik L, maka hanya X yang pertama kali ditemukan yang diacu algoritma pencarian selesai. Sebagai contoh, perhatikan larik di bawah ini:

L	21	36	8	7	10	36	68	32	12	10	36
	1	2	3	4	5	6	7	8	9	10	11

Larik L memiliki tiga buah nilai 36. Bila x = 36, maka algoritma pencarian selesai ketika X ditemukan pada elemen ke-z dan menghasilkan idx = 2 (atau menghasilkan ketemu = true jika mengacu pada keluaran pencarian jenis c). Elemen 36 lainnya tidak dipertimbangkan lagi dalam pencarian.

Algoritma pencarian yang akan dibahas di dalam bab ini adalah:

1. Algoritma pencarian beruntun (sequential search).
2. Algoritma pencarian bagidua (binary search).

Untuk masing-masing algoritma, tipe larik yang digunakan didefinisikan di bagian deklarasi global seperti di bawah ini. Larik L bertipe Lariklnt.

```
{ kamus data global }

DEKLARASI
const Nmaks = 100 {jumlah maksimum elemen larik}
type LarikInt = array [1..Nmaks] of integer
```

10.2 Implementasi Algoritma Pencarian pada Array

Pada implementasinya, algoritma pencarian pada array dibagi menjadi 2 bagian yaitu algoritma pencarian beruntun (sequential search) dan algoritma pencarian bagidua (binary search). Algoritma pencarian beruntun merupakan algoritma pencarian yang paling sederhana. Sedangkan algoritma pencarian bagidua adalah algoritma pencarian lanjutan yang lebih baik.

10.3 Algoritma Pencarian Beruntun

Algoritma pencarian yang yang paling sederhana, yaitu metode pencarian beruntun (sequential search). Nama lain algoritma pencarian beruntun adalah pencarian lurus (linear search).

Pada dasarnya, algoritma pencarian beruntun adalah proses membandingkan setiap elemen larik satu per satu secara beruntun, mulai dari elemen pertama, sampal elemen yang dicari ditemukan, atau seluruh elemen sudah diperiksa.

Contoh Perhatikan larik L di bawah ini dengan n = 6 elemen:

13	16	14	21	76	15
1	2	3	4	5	6

Misalkan nilai yang dicari adalah: x = 21

Elemen yang dibandingkan (berturut-turut): 13, 16, 14, 21 (ditemukan!)

Indeks larik yang dikemballkan: idx = 4

Misalkan nilai yang dicari adalah: x = 13

Elemen yang dibandingkan (berturut-turut): 13 (ditemukan!)

Indeks larik yang dikembalikan: idx = 1

Misalkan nilai yang dicari adalah: x = 15

Elemen yang dibandingkan (berturut-turut): 13, 16, 14, 21, 76, 21 (tidak ditemukan!)

Indeks larik yang dikembalikan: idx = -1

Berikut ini adalah contoh penerapan algoritma pencarian beruntun dengan pembandingan dilakukan sebagai kondisi pengulangan. Contoh ini tidak menggunakan peubah boolean dalam proses pencarian. Kita perlu tuliskan dua macam algoritmanya berdasarkan hasil yang diinginkan: Indeks larik atau nilai boolean. Selain itu, kita asumsikan jumlah elemen di dalam larik adalah n buah.

(a) Contoh Pembandingan elemen dilakukan sebagal kondisi pengulangan

(1) Hasil pencarian: sebuah peubah boolean yang bernilai true bila x ditemukan atau bernilai false bila x tidak ditemukan.

Setiap elemen larik L dibandingkan dengan x mulai dari elemen pertama, L[1]. Aksi pembandingan dilakukan selama indeks larik i belum melebihi n dan L[i] tidak sama dengan x. Aksi pembandingan dihentikan bila L[i] = x atau i = n. Elemen terakhir, L[n], diperiksa secara khusus. Keluaran yang dihasilkan oleh prosedur pencarian adalah sebuah peubah boolean (misal nama peubahnya ketemu) yang bernilai true jika x ditemukan, atau bernilai false jika x tidak ditemukan.

Algoritma pencarian beruntun untuk kategori hasil berupa nilai boolean dapat kita tulis sebagai sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch (input L : LarikInt, input n : integer,
      input x : integer, output ketemu : boolean)

{ Mencari keberadaan nilai x di dalam larik L[1..n]. }
{ K.Awal: x dan larik L[1..n] sudah terdefinisi nilainya. }
{ K.Akhir: ketemu bernilai true jika x ditemukan. Jika x tidak
ditemukan, ketemu bernilai false. }

DEKLARASI
i : integer { pencatat indeks larik }

ALGORITMA:
i ← 1
while (i < n) and (L[i] ≠ x) do
      i ← i + 1
```

```
endwhile
{ i = n or L[i] = x }
if L[i] = x then          { x ditemukan }
      ketemu ← true
else
      ketemu ← false      { x tidak ada di dalam larik L }
endif
```

(ii) Fungsi pencarian beruntun:

```
function SeqSearch1 (input L : LarikInt, input n : integer,
      input x : integer) → boolean

{ Mengembalikan nilai true jika x ditemukan di dalam larik
L[1..n], atau nilai false jika x tidak ditemukan. }

DEKLARASI
i : integer { pencatat indeks larik }

ALGORITMA:
i ← 1
while (i < n) and (L[i] ≠ x) do
      i ← i + 1
endwhile
{ i = n or L[i] = x }
if L[i] = x then          { x ditemukan }
      return true
else
      return false        { x tidak ada di dalam larik L }
endif
```

Perhatikanlah bahwa pada algoritma SeqSearchl di atas, pembandingan x dengan elemen larik dilakukan pada kondisi pengulangan. Apabila elemen larik yang ke-i tidak sama dengan x dan i belum sama dengan n, aktivitas pembandingan diteruskan ke elemen berikutnya (i ← i + 1).

Pembandingan dihentikan apabila L[i] = x atau indeks i sudah mencapai akhir larik (i = n). Perhatikan juga bahwa jika i sudah mencapai akhir larik, elemen terakhir ini belum dibandingkan dengan x. Pembandingan elemen terakhir dilakukan bersama-sama dengan penyimpulan hasil pencarian, Hasil pencarian disimpulkan di luar kalang while-do dengan pernyataan if (L[i] = x) then ...

Pernyataan if-then ini juga sekaligus memeriksa apakah elemen terakhir, L[n], sama dengan x. Jadi, pada algoritma SeqSearchl di atas, elemen terakhir diperiksa secara khusus,

Prosedur SeqSearchl dapat dipanggil dari program utama atau dari prosedur lain. Misal kita asumsikan prosedur SeqSearch1 dipanggil dari program utama. Misalkan program utama bertujuan untuk memeriksa keberadaan x di dalam larik, Jika x terdapat di dalarn larik maka ditampilkan pesan "ditemukan!", sebaliknya jika x tidak terdapat di dalam larik maka ditampilkan pesan "tidak ditemukan!".

Contoh program utama yang memanggil prosedur SeqSearch1:

```
PROGRAM Pencarian
{ Program untuk mencari nilai tertentu di dalam larik }

DEKLARASI
const Nmaks = 100       {jumlah maksimum elemen larik }
type LarikInt : array[1..Nmaks] of integer

L : LarikInt
x : integer { elemen yang dicari }
found : boolean   { true jika x ditemukan, false jika tidak }
n : integer { ukuran larik }

procedure BacaLarik(output L : LarikInt, input n : integer)
{ Mengisi elemen larik L[1..n] dengan nilai yang dibaca dari
piranti masukan }

procedure SeqSearch1(input L : LarikInt, input n : integer,
      input x : integer, output ketemu : boolean)
{ Mencari keberadaan nilai x di dalam larik L[1..n]. }

ALGORITMA:
read(n)           { tentukan banyaknya elemen larik }
BacaLarik(L, n)   { baca elemen-elemen larik L }
read(x)           { baca nilai yang dicari }
SeqSearch1(L, n, x, found)    { cari }
If found then     { found = true }
      write(x, 'ditemukan!')
else
      write(x, 'tidak ditemukan!')
endif
```

Prosedur BacaLarik yang disebutkan di dalam program utama di atas algoritmanya seperti di bawah ini:

```
procedure BacaLarik(output L : LarikInt, input n : integer)
{ Mengisi elemen larik L[1..n] dengan nilai yang dibaca dari
piranti masukan }
{ K.Awal: larik L belum terdefinisi elemen-elemennya. n sudah
berisi jumlah elemen efektif. n diasumsikan tidak lebih besar
dari ukuran maksimum larik ((Nmaks) }
{ K.Akhir: setelah pembacaan, sebanyak n buah elemen larik L
berisi nilai-nilai yang dibaca dari piranti masukan }

DEKLARASI
      i : integer { pencatat indeks larik }

ALGORITMA:
for i ← 1 to n do
      read(L[i])
endfor
```

Untuk pencarian beruntun yang berupa fungsi, contoh cara pemanggilan fungsi SeqSearch1:

```
read(x)
if not SeqSearch1(L, n, x) then
      write(x, 'tidak ditemukan!')
else
      { proses terhadap x }
      ...
endif
```

(2) Hasil pencarian: indeks elemen larik yang mengandung nilai x,

Setiap elemen larik L dibandingkan dengan x mulai dari elemen pertama, L[1] Aksi pembandingan dilakukan selama indeks larik i belum melebihi n dan L[i] tidak sama dengan x. Aksi pembandingan dihentikan bila L[i] : x atau i = N. Elemen terakhir, L[n], diperiksa secara khusus, Keluaran yang dihasilkan oleh prosedur pencarian adalah peubah idx yang berisi indeks larik tempat x ditemukan. Jika x tidak ditemukan, idx diisi dengan nilai -1

Algoritma pencarian beruntun untuk kategori hasil berupa indeks elemen larik dapat kita tulis yakni sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch2(input L : larik, input n : integer,
            input x : integer, output idx : integer)
{ Mencari keberadaan nilai x di dalam larik L[1..n] }
```

```
{ K.Awal: x dan elemen-elemen larik L[1..n] sudah terdefinisi }
{ K.Akhir: idx berisi indeks larik L yang berisi nilai x.
Jika x tidak ditemukan, maka idx diisi dengan nilai -15.  }

DEKLARASI
      i : integer { pencatat indeks larik }

ALGORITMA:
i ← 1
while (i < n) and (L(i) ≠ x) do
      i ← i + 1
endwhile
{ i = n or L(i) = x }
if L[i] = x then        { x ditemukan }
      idx ← i
else
      idx ← -i
endif
```

(ii) Fungsi pencarian beruntun:

```
function SeqSearch2(input L : larik, input n : integer,
            input x : integer) → integer
{ Mengembalikan indeks larik L[1..n] yang berisi x. Jika x tidak
ditemukan, maka indeks yang dikembalikan adalah -1.  }

DEKLARASI
      i : integer { pencatat indeks larik }

ALGORITMA:
i ← 1
while (i < n) and (L(i) ≠ x) do
      i ← i + 1
endwhile
{ i = n or L(i) = x }
if L[i] = x then        { x ditemukan }
      return i
else
      return -1
endif
```

Misalkan program yang memanggil prosedur SeqSearch2 bertujuan untuk menambahkan (append) nilai x ke dalam larik, namun sebelum penarnbahan itu harus ditentukan apakah x sudah terdapat di dalam larik. Jika x belum terdapat di dalam larik, maka x ditambahkan pada elemen ke- n + 1. Karena itu kita harus memastikan bahwa penambahan satu elemen baru tidak melampaui ukuran maksimum

larik (Nmaks). Setelah penambahan elemen baru ukuran larik efektif menjadi n + 1.

```
PROGRAM TambahElemenLarik
{ Program untuk menambahkan elemen baru pada ke dalam larik.
Elemen baru dibaca dari piranti masukan, lalu dicari apakah sudah
terdapat di dalam larik. Jika belum ada, tambahkan elemen baru
setelah elemen terakhir. }

DEKLARASI
const Nmaks = 100       {jumlah maksimum elemen larik }
type LarikInt : array[1..Nmaks] of integer

L : LarikInt
n : integer { ukuan larik L }
x : integer { elemen yang akan dicari }
idx : integer     { mencatat indeks elemen larik yang berisi X
}

procedure BacaLarik(output L : LarikInt, input n : integer)
{ Mengisi elemen larik L[1..n] dengan data integer }
procedure SeqSearch2(input L : LarikInt, input n : integer,
      input x : integer, output idx : integer)
{ Mencari keberadaan nilai x di dalam larik L[1..n]. }

ALGORITMA:
read(n)           { tentukan banyaknya elemen larik }
BacaLarik(L, n)   { baca elemen-elemen larik L }
read(x)
SeqSearch2(L, n, x, idx)      { cari x sebelum ditambahkan ke
dalam
L}
If idx ≠ -1 then
      write(x, 'sudah terdapat di dalam larik')
else        { x belum terdapat di dalam larik L, tambahkan x
pada
posisi ke-n+1 }
      n ← n + 1   { naikkan ukuran larik }
      L[n] ← x    { sisipkan x }
endif
```

Untuk pencarian beruntun yang berupa fungsi, contoh cara pemanggilan fungsi SeqSearch1:

```
read(x)
idx ← SeqSearch2(L, n, x)
If idx = -1 then
      write(x, 'tidak ditemukan!')
else
{ instruksi manipulasi terhadap L[idx] }
      ...
endif
```

10.4 Algoritma Pengurutan pada Array

Sebelumnya kita sudah membicarakan salah satu algoritrna pencarian, yaitu algoritma pencarian bagidua. Algoritma pencarian bagidua hanya dapat diterapkan jika elemen-elemen larik sudah terurut lebih dahulu. Di dalam bab ini akan dikemukakan beberapa algoritma pengurutan data

Masalah pengurutan rnerupakan persoalan yang menarik, karena terdapat puluhan algoritma pengurutan yang pernah dikemukakan orang. Tidak semua algoritma pengurutan akan dibahas di dalam bab ini, hanya beberapa pengurutan sederhana yang akan dijelaskan.

Pengurutan (sorting) adalah proses mengatur sekumpulan objek menurut urutan atau susunan tertentu (Nicklaus Wirth, 1976). Masalah pengurutan dapat ditulis sebagai berikut:

Diberikan larik L dengan n elemen yang sudah terdefinisi elemen-elemennya. Urutan Iarik tersebut sehingga tersusun secara menaik (ascending):

$L[1] \leq L[2] \leq L[3] \leq \ldots \leq L[n]$

atau secara menurun (descending):

$L[1] \geq L[2] \geq L[3] \geq \ldots \geq L[n]$

Data yang diurut dapat berupa data bertipe dasar atau tipe terstruktur (record). Jika data bertipe terstrukrur, maka harus dispesifikasikan berdasarkan field apa data tersebut diurutkan. Field yang dijadikan dasar pengurutan dikenal sebagai field kunci.

Berikut ini diberikan beberapa contoh data yang terurut:

i. 23, 27, 45, 67, 100, 130, 501
   (data bertipe integer terurut menaik)
ii. 50.27, 31.009, 20.3, 19.0, -5.2, -10.9
   (data bertipe riil terurut menurun)
iii. ‘Amir’, ‘Badu’, ‘Budi’, ‘Dudi’, ‘Eno’, ‘Rudi’, ‘Zamzami’
   (data bertipe string terurut menaik)
iv. ‘d’, ‘e’, ‘g’, ‘i’, ‘x’
   (data bertipe karakter terurut menaik)

v. <13596001, 'Eko', 'A'>, <13596006, 'Rizka', 'B'>, <13596007, 'Hamdi', 'D'>, <13596010, 'Rizal', 'C'>, <13596012, 'Raum', 'B'>
(data mahasiswa bertipe terstruktur terurut menaik berdasarkan field NIM)

Dalam kehidupan sehari-hari, entry di dalam buku telepon, kata di dalam kamus bahasa/istilah, dan entry di dalam ensiklopedi selalu terurut secara alfabetik. Kita juga sering melakukan pengururan seperti mencatat alamat teman berdasarkan nama, menyusun tumpukan koran berdasarkan tanggal, mengantri di pasar swalayan berdasarkan urutan waktu kedatangan di kasir, dan sebagainya.

Data yang sudah terurut memiliki beberapa keuntungan. Selain mempercepat waktu pencarian, dari data yang terurut kita dapat langsung memperoleh nilai maksimum dan nilai minimum. Untuk data numerik yang terurut menurun, nilai maksimum adalah elemen pertama larik, dan nilai minimum adalah elemen terakhir larik. Hal ini bermanfaat untuk mengetahui juara kelas misalnya, atau mengetahui peserta ujian Seleksi Penerimaan Mahasiswa Baru (SPMB) yang memperoleh skor nilai tertinggi. Pengurutan dikatakan stabil jika dua atau lebih data yang sama (atau identik) tetap pada urutan yang sama setelah pengurutan. Misalnya di dalam sekelompok data integer berikut terdapat 3 buat nilai 12 (diberi tanda petik ', ", dan "' untuk mengidentifikasi urutannya. 70, 12', 45, 10, 12", 60, 12"', 33, 50

Jika suatu metode pengurutan mengbasilkan susunan data terurut seperti berikut:

10, 12', 12", 12"', 33, 45, 50, 60, 70

maka pengurutannya dikatakan stabil dan metode pengurutannya disebut metode stabil. Tetapi jika suatu metode pengurutan menghasilkan susunan data terurut seperti contoh berikut:

10, 12", 12', 12"', 33, 45, 50, 60, 70

maka pengurutan dan metodenya kita katakan tidak stabil.

Kestabilan pengurutan mungkin penting atau tidak penting (Thomas W Parsons, 1995). Misalnya kita ingin menyusun data buku yang terurut secara alfabetik berdasarkan nama pengarang sekaligus terurut berdasarkan judul buku oleh pengarang tersebut (pengarang bisa menulis lebih dari satu buku). Jika kita menggunakan metode pengurutan yang stabil, maka kita urut berdasarkan judul buku lebih dahulu baru kemudian kita urut berdasarkan nama pengarang. Jika kita menggunakan metode pengurutan yang tidak stabil, maka kita tidak memperoleh hasil pengurutan yang kita inginkan

10.5 Implementasi Algoritma Pengurutan pada Array

Adanya kebutuhan terhadap proses pengurutan memunculkan bermacam-macam algoritma pengurutan. Banyak algoritma pengurutan yang telah ditemukan. Hal ini menunjukkan bahwa persoalan pengurutan adalah persoalan yang kaya dengan solusi algoritmik. Algoritma pengurutan yang sering ditemukan di dalam literatur-literatur komputer antara lain adalah:

1. Bubble Sort
2. Selection Sort (Maximum Sort dan Minimum Sort)
3. Insertion Sort
4. Heap Sort
5. Shell Sort
6. Quick Sort
7. MergeSort
8. Radix Sort
9. TreeSort

Pada bab ini kita tidak akan membahas semua algoritma pengurutan tersebut, tetapi hanya empat buah algoritma pengurutan yang sederhana saja, yaitu:

1. Metode Bubble Sort (Pengurutan Apung)
2. Metode Selection Sort (Pengurutan Seleksi)
3. Metode Insertion Sort (Pengurutan Sisip)

4. Metode Shell Sort (Pengurutan Shell)

Dua algoritma pertama (bubble dan selection sort) melakukan prinsip pertukaran elemen dalam proses pengurutan sehingga keduanya dinamakan pengurutan dengan pertukaran (exchange sorts), sedangkan dua algoritma terakhir melakukan prinsip geser dan sisip elemen dalam proses pengurutan (shift and insert sorts). Semua algoritma pengurutan selalu melakukan operasi perbandingan elemen larik untuk menemukan posisi urutan yang tepat.

Seperti halnya pada pencarian, algoritma pengurutan juga dapat diklasifikasikan sebagai algoritma pengurutan internal dan algoritma pengurutan eksternal.

1. Algoritma pengurutan internal, yaitu algoritma pengurutan untuk data yang disimpan di dalam memori komputer. Umumnya struktur internal yang dipakai untuk pengurutan internal adalah larik, sehingga pengurutan internal disebut juga pengurutan larik.
2. Algoritma pengurutan eksternal, yaitu metode pengurutan untuk data yang disimpan di dalam disk storage, disebut juga pengurutan arsip (file), karena struktur eskternal yang dipakai adalah arsip.

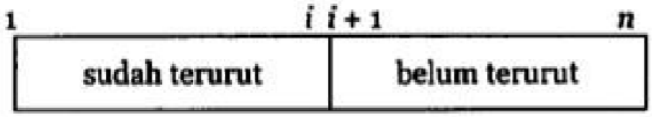
Untuk semua algoritma pengurutan yang akan dijelaskan di dalam bab ini, kita menggunakan tipe data larik yang didefinisikan di bagian deklarasi sebagai berikut:

```
DEKLARASI
const Nmaks = 1000      {jumlah maksimum elemen larik }
type LarikInt : array[1..Nmaks] of integer
```

10.6 Algoritma Bubble Sort

Algoritma pengurutan apung (bubble sort) diinspirasi oleh gelembung sabun yang berada di atas permukaan air. Karena berat jenis gelembung sabun lebih ringan daripada berat jenis air, maka gelembung sabun selalu terapung ke atas permukaan. Secara umum, benda-benda yang berat akan terbenam dan benda-benda yang ringan akan terapung ke atas permukaan.

Prinsip pengapungan di atas juga digunakan pada pengurutan apung. Apabila kita menginkan larik terurut menaik, maka elemen larik yang berharga paling kecil "diapungkan", artinya diangkat ke "atas" (atau ke ujung kiri larik) melalui proses pertukaran. Proses pengapungan ini dilakukan sebanyak n - 1 langkah (satu langkah disebut juga satu kali pass) dengan n adalah ukuran larik. Pada akhir setiap langkah ke-i, larik L[1..n] akan terdiri atas dua bagian yaitu bagian yang sudah terurut, yaitu L[1..i], dan bagian yang belum terurut, L[i + 1..n] (Gambar 10.1). Setelah langkah terakhir, diperoleh larik L[1..n] yang terurut menaik



Gambar 10. 1 Bagian larik yang belum terurut

Untuk mendapatkan larik yang terurut menaik, algoritma pengurutan apung secara global sebagal berikut:

Untuk setiap pass i = 1, 2, ..., n - 1, lakukan:
Mulai dari elemen k = n, n - 1, ..., i + 1, lakukan:
1.1 Bandingkan L[k] dengan L[k – 1].
1.2 Pertukarkan L[k] dengan L[k - 1] jika L[k] < L[k - 1].

Rincian setiap pass sebagai berikut:

Pass 1 : Mulai dari elemen ke-k = n, n-i, ..., 2, bandingkan L[k] dengan L[k-1]. Jika L[k] < L[k- 1], pertukarkan L[k] dengan L[k – 1]. Pada akhir langkah 1, elemen L[l] berisi harga minimum pertama.

Pass 2 : Mulai dari elemen ke-k = n, n - 1, ..., 3, bandingkan L[k] dengan L[k - 1]. Jika L[k] < L[k- 1], pertukarkan L[k] dengan L[k-1]. Pada akhir langkah 2, elemen L[2] berisi harga minimum kedua, larik L[1 .. 2] terurut, sedangkan L[3 .. n] belum terurut.

Pass 3 : Mulai dari elemen ke-k = n, n - 1, ..., 4, bandingkan L[k] dengan L[k – 1]. Jika L[k] < L[k – 1], pertukarkan L[k] dengan L[k-1]. Pada akhir langkah 3, elemen L[3] berisi harga minimum ketiga, larik L[1..3] terurut, sedangkan L[4..n] belum terurut

Pass n – 1 : Mulai darl elemen ke k = n, bandingkan L[k] dengan L[k – 1]. Jika L[k] < L[k-1], pertukarkan L[k] dengan L[k- 1]. Pada akhir pass n - 1, elemen L[n – 1] berisi nilai minimum ke- (n - 1) dan larik L[1..n - 1] terurut menaik (elemen yang tersisa adalah L[n], tidak perlu diurut karena hanya satu-satunya).

Tinjau larik L dengan n = 6 buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menaik:

25	27	10	8	76	21
1	2	3	4	5	6

← arah pembandingan

Pass 1:

k	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
k = 6	L[6] < L[5]? (21 < 76?)	Ya	25, 27, 10, 8, 21, 76
k = 5	L[5] < L[4]? (21 < 8?)	Tidak	25, 27, 10, 8, 21, 76
k = 4	L[4] < L[3]? (8 < 10?)	Ya	25, 27, 8, 10, 21, 76
k = 3	L[3] < L[2]? (8 < 27?)	Ya	25, 8, 27, 10, 21, 76
k = 2	L[2] < L[1]? (8 < 25?)	Ya	8, 25, 27, 10, 21, 76

Hasil akhir pass 1:

8	25	27	10	21	76
1	2	3	4	5	6

Pass 2: (berdasarkan hasil akhir pass 1)

K	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
k = 6	L[6] < L[5]? (76 < 21?)	Tidak	8, 25, 27, 10, 21, 76

k = 5	L[5] < L[4]? (21 < 10?)	Tidak	8, 25, 27, 10, 21, 76
k = 4	L[4] < L[3]? (10 < 27?)	Ya	8, 25, 10, 27, 21, 76
k = 3	L[3] < L[2]? (10 < 25?)	Ya	8, 10, 25, 27, 21, 76

Hasil akhir pass 2:

8	10	25	27	21	76
1	2	3	4	5	6

Pass 3: (berdasarkan hasil akhir pass 2)

k	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
k = 6	L[6] < L[5]? (76 < 21?)	Tidak	8, 10, 25, 27, 21, 76
k = 5	L[5] < L[4]? (21 < 27?)	Ya	8, 10, 25, 21, 27, 76
k = 4	L[4] < L[3]? (21 < 25?)	Ya	8, 10, 21, 25, 27, 76

Hasil akhir pass 3:

8	10	21	25	27	76
1	2	3	4	5	6

Pass 4: (berdasarkan hasil akhir pass 3)

k	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
k = 6	L[6] < L[5]? (76 < 27?)	Tidak	8, 10, 21, 25, 27, 76
k = 5	L[5] < L[4]? (27 < 25?)	Tidak	8, 10, 21, 25, 27, 76

Hasil akhir pass 4:

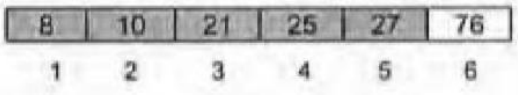
8	10	21	25	27	76
1	2	3	4	5	6

Pass 5: (berdasarkan hasil akhir pass 4)

K	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara

k = 6	L[6] < L[5]? (76 < 27?)	Tidak	8, 10, 21, 25, 27, 76

Hasil akhir pass 5:



Hasil akhir pass 5 menyisakan satu elemen (yaitu 76) yang tidak perlu diurutkan lagi, maim pengurutan selesai. Larik L sekarang sudah terurut menaik!

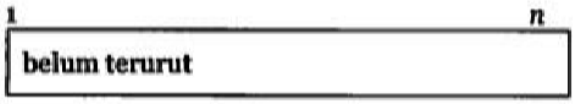
```
procedure BubbleSort1(input/output L : LarikInt, input n : integer)
{ Mengurutkan larik L[1..n] sehingga terurut menaik dengan
metode pengurutan apung}
{ K.Awal: Elemen larik L sudah terdefinisi nilai-nilainya.}
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
L[1] ≤ L[2] ≤ ... ≤ L[n]. }

DEKLARASI
      i : integer { pencacah untuk jumlah langkah }
      k : integer { pencacah untuk pengapungan pada langkah }
      temp : integer { pencacah bantu untuk pertukaran }

ALGORITMA:
for i ← 1 to n - 1 do
      for k ← n downto i + 1 do
            if L[k] < L[k-1] then

      {pertukaran L[k] dengan L[k-1]}
      temp ← L[k]
      L[k] ← L[k-1]
      L[k-1] ← temp
      endif
      endfor
endfor
```

Pengurutan apung merupakan algoritma pengurutan yang tidak efisien. Hal ini, disebabkan oleh banyaknya operasi pertukaran yang dilakukan pada setiap langkah pengapungan. Untuk ukuran larik yang besar, pengurutan dengan algoritma ini membutuhkan waktu yang lama. Karena alasan itu, maka algoritma pengurutan apung jarang digunakan dalam praktek pemrograman. Namun, kelebihan algoritma ini adalah pada kesederhanaanya dan mudah dipahami.

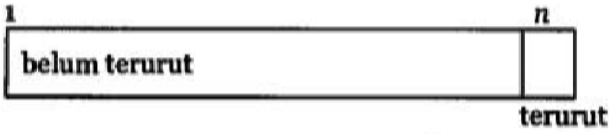
10.7 Algoritma Selection Sort

Algoritma pengurutan ini disebut selection sort (pengurutan seleksi) karena gagasan dasarnya adalah memilih elemen maksimum/minimum dari larik, lalu menempatkan elemen maksimum/minimum itu pada awal atau akhir larik (elemen terujung) (lihat Gambar 10.2). Selanjutnya elemen terujung tersebut "diisolasi" dan tidak disertakan pada proses selanjutnya. Proses yang sama diulang untuk elemen larik yang tersisa, yaltu memilih elemen maksimum/minimum berikutnya dan mempertukarkannya dengan elemen terujung larik sisa. Sebagaimana halnya pada algoritma pengurutan gelembung, proses memillh nilai maksimum/minimum dilakukan pada setiap pass. Jika larik berukuran n, maka jumlah pass adalah n-1.

Sebelum:



Sesudah:



Gambar 10. 2 Bagian larik yang terurut dan belum terurut pada algoritma pengurutan seleksi

Ada dua varian algoritma pengurutan seleksi ditinjau dari pemilihan elemen maksimum/ minimum, yaitu:

1. Algoritma pengurutan seleksi-maksimum, yaitu memilih elemen maksimum sebagai basis pengurutan. (yang akan kita bahas pada bab ini).
2. Algoritma pengurutan seleksi-minimum, yaitu memilih elemen minimum sebagai basis pengurutan.

10.8 Algoritma Pengurutan Seleksi-Maksimum

Untuk mendapatkan larik yang terurut menaik. algoritma pengurutan seleksi-rnaksimum ditulis secara garis besar sebagai berikut:

1. JumlahPass = n – 1
2. Untuk setiap pass i = 1, 2, ..., JumlahPass lakukan:
   2.1 cari elemen terbesar (maks) mulai dari elemen ke-i sampai elemen ke-n;
   2.2 pertukarkan maks dengan elemen ke-n;
   2.3 kurangi n satu (karena elemen ke-n sudah terurut).

Rincian aksi pada setiap pass, seperti di bawah ini:

Pass 1 : Cari elemen maksimum di dalam L[1 .. n].
Pertukarkan elemen maksirnum dengan elernen L[n].
Ukuran larik yang belum terurut = n - 1.

Pass 2 : Cari elemen maksimum di dalam L[i .. n - 1].
Pertukarkan elemen maksimum dengan elemen L[n - 1].
Ukuran larik yang belum terurut = n - 16.

Pass 3 : Cari elemen mnksimum di dalam L[1 .. n - 2].
Pertukarkan elemen maksimum dengan elemen L[n - 2].
Ukuran larik yang belum terurut = n - 3.

...

Pass n – 1 : Tentukan elemen maksimum di dalnm L[1..2]
Pertukarkan elemen maksimum dengan elemen L[2]
Ukuran larik yang belum terurut = 1.

Setelah pass n – 1, elemen yang tersisa adalah L[1], tidak perlu diurutkan lagi karena hanya satu-satunya.

Tinjau larik dengan n = 6 buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menaik dengan metode pengurutan seleksi-maksimum:

29	27	10	8	76	21
1	2	3	4	5	6

Pass 1:

Cari elemen maksimum di dalam larik L[1..6] => hasilnya: maks = L[5] = 76.

Pertukarkan maks dengan L[n], diperoleh:

29	27	10	8	21	76
1	2	3	4	5	6

Pass 2:

(berdasarkan susunan larik hasil pass 1)

Cari elemen maksimum di dalam larik L[1..5] => hasilnya: maks = L[1]=29.

Pertukarkan Maks dengan L[5], diperoleh:

21	27	10	8	29	76
1	2	3	4	5	6

Pass 3:

(berdasarkan susunan larik hasil pass 2)

Cari elemen maksimum di dalam larik L[1..4] => hasilnya: maks = L[2] = 27.

Pertukarkan maks dengan L[4], diperoleh:

21	8	10	27	29	76
1	2	3	4	5	6

Pass 4:

(berdasarkan susunan larik hasil pass 3)

Cari elemen maksimum di dalam larik L[1..3] => hasilnya: maks = L[1] = 2l.

Pertukarkan maks dengan L[3], diperoleh:

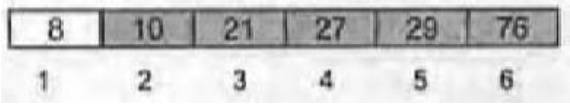
10	8	21	27	29	76
1	2	3	4	5	6

Pass 5:

(berdasarkan susunan larik hasil pass 4)

Cari elemen maksimum di dalam larikl[1..2] => hasilnya: maks = L[1] = 10.

Pertukarkan maks dengan L[2], diperoleh:



Tersisa satu elemen (yaitu 8), maka pengurutan selesai. Larik L sudah terurut menaik!

Jadi, pada setiap pass pengurutan terdapat proses mencari harga maksimum dan proses pertukaran dua buah elemen larik.

Algoritma pengurutan seleksi-maksimum selengkapnya sebagai berikut:

```
procedure SelectionSort1(input/output L : LarikInt, input n : integer)
{ Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan seleksi-maksimum}
{ K.Awal: Elemen larik L sudah terdefinisi harganya.}
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
L[1] ≤ L[2] ≤ ... ≤ L[n]. }

DEKLARASI
      i     : integer { pencacah pass }
      j     : integer { pencacah untuk mencari nilai maksimum }
      imaks       : integer { indeks yang berisi nilai maksimum sementara}
      maks  : integer { elemen maksimum }
      temp  : integer { peubah bantu untuk pertukaran }

ALGORITMA:
for i ← n downto 2 do          { jumlah pass sebanyak n - 1 }
      { cari elemen maksimum pada elemen L[1..i] }
      Imaks ← 1          { elemen pertama diasumsikan sebagai
                           elemen maksimum sementara }
      for j ← 2 to i do
            if L[j] > maks then
      imaks ← j
      maks ← L[j]
      endif
      endfor

      { pertukaran maks dengan L[1] }
      temp  ← L[i]
      L[i] ← maks
      L[imaks] ← Temp
endfor
```

Dibanding dengan algoritma pengurutan apung, algoritma pengurutan seleksi memiliki kinerja yang lebih baik. Alasannya, operasi pertukaran elemen hanya dilakukan sekali saja pada setiap pass, dengan demikian lama pengurutannya berkurang dibandingkan dengan metode pengurutan gelembung.

10.9 Algoritma Insertion Sort

Dari namanya, insertion sort (pengurutan sisip) adalah metode pengurutan dengan cara menyisipkan elemen larik pada posisi yang tepat. Pencarian posisi yang tepat dilakukan dengan menyisir larik, Selama penyisiran dilakukan pergeseran elemen larik. Metode pengurutan sisip cocok untuk persoalan menyisipkan elemen baru ke dalam sekurnpulan elemen yang sudah terurut.

10.10 Algoritma Pengurutan Sisip untuk Pengurutan Menaik

Untuk mendapatkan larik yang terurut menaik. algoritma pengurutan sisip secara garis besar ditulis sebagai berikut:

Untuk setiap pass i = 2, ..., n lakukan:

1. y ←L[i]
2. sisipkan y pada tempat yang sesuai di antara L[i] ... L[i]

Rincian setiap pass sebagai berikut:

Asumsikan: L[1] dianggap sudah pada tempatnya

Pass 2 : Elemen y = L[2] harus dicari tempatnya yang tepat di dalam L[1..2] dengan cara menggeser elemen L[1..1] ke kanan (atau ke bawah, jika anda membayangkan larik terentang vertikal) bila L[1..1] lebih besar daripada L[2]. Misalkan posisi yang tepat adalah k. Sisipkan L[2] pada L[k].

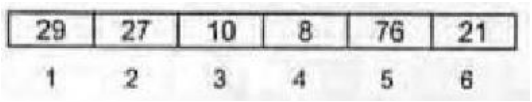
Pass 3 : Elemen y = L[3] harus dicari tempatnya yang tepat di dalam L[1..3] dengan cara menggeser elemen L[1..2] ke kanan (atau ke bawah) bila L[1..2] lebih besar daripada L[3]. Misalkan posisi yang tepat adalah k. Sisipkan L[3] pada L[k].

...

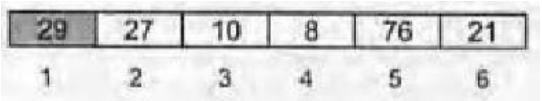
Pass n : Elemen y = L[n] harus dicari tempatnya yang tepat di dalam L[1..n] dengan cara menggeser elemen L[1..n - 1] ke kanan (atau ke bawah) bila L[1..n - 1] lebih besar daripada L[n]. Misalkan posisi yang tepat adalah k. Sisipkan L[n] pada L[k].

Hasil dari pass n: Larik L[i..n] sudah terurut menaik, yaitu
L[1] ≤ L[2] ≤ ... ≤ L[n].

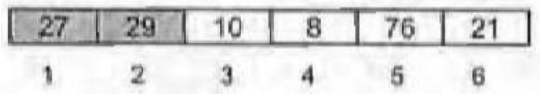
Tinjau larik dengan n = 6 buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menaik dengan metode pengurutan sisip:



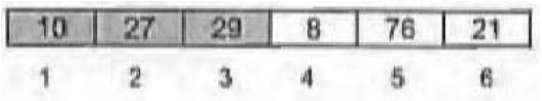
Asumsikan: Elemen y = L[1] = 29 dianggap sudah terurut.



Pass 2:
(berdasarkan susunan larik pada akhir pass 1)
Cari posisi yang tepat untuk y = L[2] = 27 di dalam L[1..2], diperoleh:



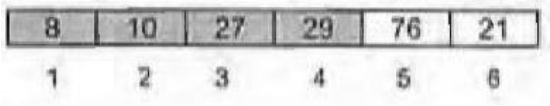
Pass 3:
(berdasarkan susunan larik pada akhir pass 2)
Cari posisi yang tepat untuk y = L[3] = 10 di dalam L[1..3], diperoleh:



Pass 4:

(berdasarkan susunan larik pada akhir pass 3)

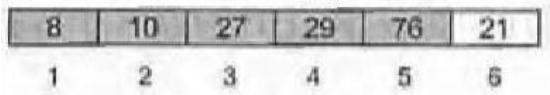
Cari posisi yang tepat untuk y: L[4] = 8 di dalam L[1..4], diperoleh:



Pass 5:

(berdasarkan susunan larik pada akhir pass 4)

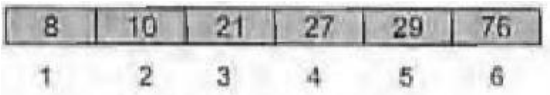
Cari posisi yang tepat untuk y = L[5] = 76 di dalam L[1..5], diperoleh:



Pass 6:

(berdasarkan susunan larik pada akhir pass 5)

Cari posisi yang tepat untuk y = L[6] = 21 di dalam L[1 .. 6], diperoleh:



Algoritma pengurutan sisip selengkapnya sebagai berikut:

```
procedure InsertionSort1(input/output L : LarikInt, input n : integer)
{ Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan sisip}
{ K.Awal: Elemen-elemen larik L sudah terdefinisi nilainya.}
{ K.Akhir: Elemen-elemen larik L terurut menaik sedemikian sehingga L[1] ≤
L[2] ≤ ... ≤ L[n]. }

DEKLARASI
      i     : integer { pencacah pass }
      j     : integer { pencacah untuk penelusuran larik }
      y     :  integer  {  peubah  bantu  agar  L[K]  tidak  ditimpa  selama
pergeseran }
      ketemu : boolean { untuk menyatakan posisi penyisipan ditemukan}

ALGORITMA:  { elemen L[1] dianggap sudah terurut }
for i ← 2 to n do       { mulai dari pass 2 sampai pass N }

      y ← L[i]
      { cari posisi yang tepat untuk y di dalam L[1..i-1] sambil menggeser
}
```

```
      j ← i - 1
      ketemu ← false

      while (j ≥ 1) and (not ketemu) do
            if y < L[j] then
      L[j + 1] ← L[j]   { geser }
      j ← j - 1
      else
            ketemu ← true
            endif
      endwhile
      { j < i or ketemu }
      L[j + 1] ← y { arsipkan y pada tempat yang sesuai }
endfor
```

Kelemahan algoritma pengurutan sisip terletak pada banyaknya operasi pergeseran yang diperlukan dalam mencari posisi yang tepat untuk elemen larik. Pada setiap pass i, operasi pergeseran yang diperlukan maksimum i - 1 kali. Untuk larik yang berukuran besar, jumlah operasi pergeseran meningkat secara kuadratik, sehingga pengurutan sisip kurang bagus untuk volume data yang besar.

10.11 Algoritma Shell Sort

Algoritma pengurutan Shell diberi nama sesuai nama penemunya (Donald Shell tahun 1959) (Thomas W Parsons, 1995). Algoritma ini merupakan perbaikan terhadap metode pengurutan sisip. Kelemahan metode pengurutan sisip sudah disebutkan pada bagian sebelum ini. Jika data pada posisi ke-1000 temyata posisi yang tepat adalah sebagai elemen kedua, maka dibutuhkan kira-kira 998 kali pergeseran elemen.

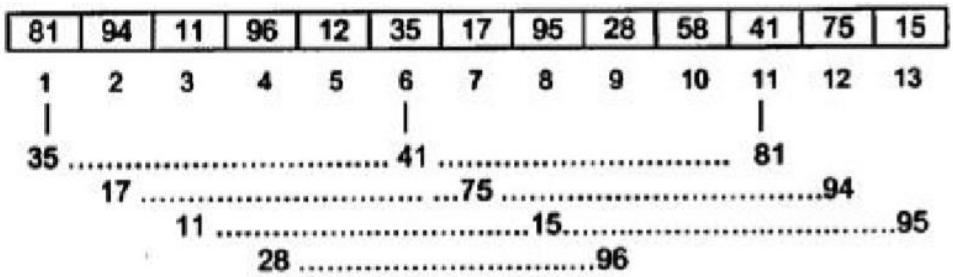
Untuk mengurangi pergeseran terlalu jauh, kita mengurutkan larik setiap k elemen dengan metode pengurutan sisip, misalkan kita urutkan setiap 5 elemen (k kita namakan juga step atau increment). Selanjutnya, kita gunakan nilai step yang lebih kecil, misalnya k = 3, lalu kita urut setiap 3 elemen. Begitu seterusnya sampai nilai k = 1. Karena nilai step selalu berkurang maka algoritma pengurutan Shell kadang-kadang dinamakan juga algoritma pengurutan kenaikan yang berkurang (diminishing increment sort).

Untuk memperjelas metode pengurutan Shell, tinjau pengurutan data integer berikut:

Data sebelum pengurutan:

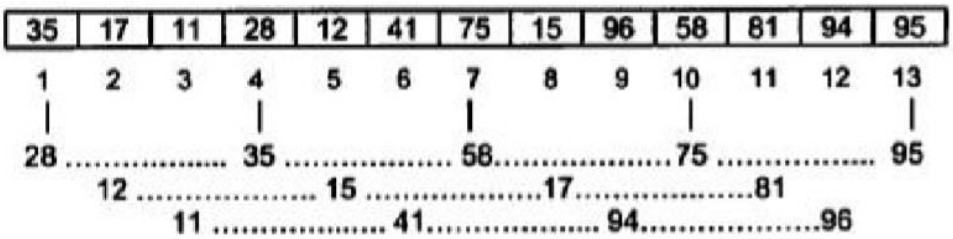
81	94	11	96	12	35	17	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

Pass 1 (step = 5): Urutkan setiap lima elemen



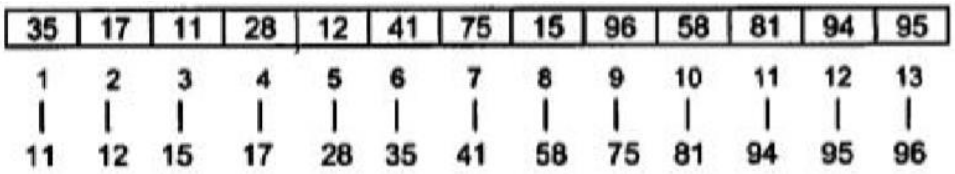
Baris pertama menyatakan elemen yang sudah terurut pada posisi 1, 6, dan 11. Baris kedua menyatakan elemen yang terurut pada posisi 2, 7, dan 12 Baris ketiga menyatakan elemen yang sudah terurut pada posisi 3, 9, dan 13 Baris keempat menyatakan elemen yang sudah terurut pada posisi 4 dan 9. Hasil pass pertama: 35, 17, 11, 28, 12, 41, 75, 15, 96, 58, 81, 94, 95.

Pass 1 (step = 3): Urutkan setiap tiga elemen



Hasil pass kedua: 28, 12, 11, 35, 15, 41, 58, 17, 94, 75, 81, 96, 95.

Pass 3 (step = 1): Urutkan setiap satu elemen



Hasil pass ketiga: 11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96.

Perhatikanlah bahwa pada pass yang terakhir (step = 1), pengurutan Shell menjadi sama dengan pengurutan sisip biasa.

Nilai-nilai step seperti 5, 3, dan 1 bukanlah angka "sihir" (magic). Kita dapat memilih nilai-nilai step yang lain yang bukan kelipatan dari step yang lain. Pemilihan step yang merupakan perpangkatan dari dua (seperti 8, 4, 2, 1) dapat mengakibatkan perbandingan elemen yang sama pada suatu pass akan terulang kembali pada pass berikutnya. Meskipun beberapa penelitian telah dibuat pada algoritma Shell, namun tidak seorang pun yang dapat membuktikan bahwa pemilihan step tertentu paling bagus di antara pemilihan step yang lain (Markus Robijanto Kusuma, 1991)

Secara garis besar, algoritma pengurutan Shell dituliskan sebagai berikut:

1. step ← n { n = ukuran larik }
2. While step > l do
   a. step ← step div 3 + 1
   b. For i ← 1 to step do
      Insertion Sort setiap elemen ke-step mulai dari elemen ke-i

Algoritma pengurutan Shell dibuat dengan pertama-tama memodifikasi algoritma pengurutan sisip sedemikian sehingga kita dapat menspesifikasikan titik awal pengurutan dan ukuran step (pada algoritma pengurutan sisip yang asli, titik awal pengurutan adalah elemen pertama dan ukuran step = 1). Modifikasi algoritma pengurutan sisip ini kita beri nama InsSort. Algoritma InSort dan ShellSort selengkapnya kita tulis berikut ini.

```
procedure InSort(input/output L : LarikInt, input n : start,
      step : integer)
{ Mengurutkan elemen-elemen larik L[start..n] sehingga tersusun menaik dengan
metode pengurutan sisip yang dimodifikasi untuk Shell Sort. }
```

```
{ K.Awal: Elemen-elemen larik L sudah terdefinisi nilainya.}
{ K.Akhir: Elemen-elemen larik pada kenaikan sebesar step terurut menaik }

DEKLARASI
      i     : integer { pencacah step }
      j     : integer { pencacah untuk penelusuran larik }
      y     : integer { peubah bantu yang menyimpan nilai L[K] }
      ketemu : boolean { untuk menyatakan posisi penyisipan ditemukan}

ALGORITMA:
{ elemen L[start] dianggap sudah terurut }
I ← start + step
while i ← n do
      y ← L[i]
      { sisipkan L[i] ke dalam bagian yang sudah terurut }
      { cari posisi yang tepat untuk y di dalam L[start..i-1] sambil
menggeser }
      j ← i - step
      ketemu ← false

      while (j ≥ 1) and (not ketemu) do
            if y < L[j] then
      L[j + step] ← L[j]      { geser }
      j ← j - step
      else
            ketemu ← true
            endif
      endwhile
      { j < i or ketemu }
      L[j + step] ← y { sisipkan y pada tempat yang sesuai }
      i  ← i + step
endwhile
```

Sebagaimana sudah dijelaskan sebelumnya, algoritma pengurutan Shell merupakan perbaikan terhadap metode pengurutan sisip. Namun, tidak seorang pun yang pernah dapat menganalisis algoritma pengurutan Shell secara tepat, karena pemilihan ukuran step (seperti 5, 3, 1, atau pendefinisian ukuran step dengan pernyataan step ← step div 3 + 1) didasarkan pada pertimbangan sugesti. Tidak ada aturan yang diketahui untuk menemukan ukurun step yang optimum, Jika step dipilih yang berdekatan, semakin banyak jumlah pass yang dihasilkan, tapi setiap pass mungkin lebih cepat. Jika ukuran step menurun dengan cepat, jumlah pass akan berkurang tetapi setiap pass menjadi lebih panjang (lama). Namun yang pasti, ukuran step terakhir adalah 1, sehingga pada akhir proses, larik diurutkan dengan pengurutan sisip biasa.

POST TEST

Soal dan Pembahasan tentang Materi Eksplorasi Array

1. Apa yang kamu ketahui tentang arsip beruntun?
2. Bagaimana contoh implementasi dari arsip beruntun?
3. Apa yang kamu ketahui tentang rekursif?
4. Bagaimana contoh implementasi dari rekursif?