1 Pencarian

Pencarian (*searching*) merupakan proses yang fundamental dalam pengolahan data. Proses pencarian adalah menemukan nilai (data) tertentu di dalam sekumpulan data yang bertipe sama (baik bertipe dasar atau bertipe bentukan). Di dalam Buku 1 Algoritma dan Pemrograman telah disebutkan bahwa aktivitas yang berkaitan dengan pengolahan data sering didahului dengan proses pencarian. Sebagai contoh, untuk mengubah (*update*) data tertentu, langkah pertama yang harus dilakukan adalah mencari keberadaan data tersebut di dalam kumpulannya. Jika data yang dicari ditemukan, maka data tersebut dapat diubah nilainya dengan data yang baru. Aktivitas awal yang sama juga dilakukan pada proses penambahan (*insert*) data baru. Proses penambahan data dimulai dengan mencari 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.

Data dapat disimpan secara temporer dalam memori utama atau disimpan secara permanen di dalam memori sekunder (*tape* atau *disk*). Di dalam memori utama, struktur penyimpanan data yang umum adalah berupa larik atau tabel (*array*), sedangkan di dalam memori sekunder berupa arsip (*file*). Bab 1 ini dititikberatkan pada algoritma pencarian data di dalam larik. Algoritma pencarian yang akan dibicarakan dimulai dengan algoritma pencarian yang paling sederhana (yaitu pencarian beruntun atau *sequential search*) sampai pada algoritma pencarian yang lebih maju yaitu pencarian bagidua (*binary search*).

1.1 Tinjauan Singkat Larik

Di dalam Buku 1 kita sudah membicarakan mengenai larik larik atau tabel. Untuk menyegarkan ingatan, upabab 1.1. ini meninjau (*review*) kembali secara singkat mengenai larik. Larik merupakan tipe data terstruktur.

Sebuah larik dapat dibayangkan sebagai sekumpulan kotak yang menyimpan sekumpulan elemen bertipe sama secara berturutan (*sequential*) di dalam memori komputer (di dalam Gambar 1.1 elemen-elemen larik disusun horizontal. Anda juga dapat membayangkan elemen-elemen larik disusun secara vertikal sehingga dinamai tabel). Setiap elemen larik data diacu melalui indeksnya. Karena elemen disimpan

secara berturutan, indeks larik haruslah suatu tipe yang juga mempunyai keterurutan (memiliki suksesor dan ada predesesor), misalnya tipe *integer* atau karakter. Jika indeksnya adalah *integer* maka keterurutan indeks sesuai dengan urutan *integer*. Jika indeks larik adalah karakter maka keterurutan indeks sesuai dengan urutan karakter. Tiap elemen larik langsung diakses dengan menspesifikasikan nama larik berikut indeksnya.

Untuk contoh-contoh larik pada Gambar 1.1, kita mendefinisikan nama dan tipenya di bagian deklarasi dari algoritma sebagai berikut:

```
DEKLARASI
  D  : array[1..11] of integer          { larik pada Gambar 1.1(a) }
  Kar: array[1..8] of character         { larik pada Gambar 1.1(b) }

  const n = 5   { jumlah data siswa }
  type Data = record <Nama: string, Usia:integer>
  Siswa : array[1..n] of Data          { larik pada Gambar 1.1(c) }
```

Algoritma 1.1 Contoh pendeklarasian larik

(a) **D**

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

(b) **Kar**

1	2	3	4	5	6	7	8
k	m	t	a	f	m	*	#

(c) **Siswa**

1	Ali	18
2	Tono	24
3	Amir	30
4	Tuti	21
5	Yani	22

Gambar 1.1 (a) Larik bertipe integer,
(b) larik bertipe karakter,
(c) larik bertipe terstrkur.
Angka 1, 2, 3, ... menyatakan indeks larik.

Contoh-contoh cara mengacu elemen larik pada Gambar 1.1:

```
D[2]
D[k] {mengacu elemen ke-k, dengan syarat k sudah terdefinisi nilainya}
Kar[5]
Siswa[1].Nama
Siswa[1].Usia
Siswa[j].Nama  { mengacu elemen ke-j, dengan syarat j sudah terdefinisi
                 nilainya
```

Di dalam Buku 1 telah dijelaskan beberapa proses sekuensial terhadap larik. Proses sekuensial terhadap larik antara lain menginisialisasi larik, mengisi elemen-elemen larik dengan data dari piranti masukan, menuliskan elemen-elemen larik ke piranti

tentu di dalam larik. keluaran, mencari elemen maksimum atau minimum, serta mencari elemen tertentu di dalam larik.

Di dalam Buku 2 ini, algoritma pencarian elemen tertentu di dalam larik akan dibahas lebih mendalam. Karena larik adalah struktur internal (yaitu, struktur yang direalisasikan di dalam memori utama), maka pencarian elemen di dalam larik disebut juga **pencarian internal**. Sedangkan pada **pencarian eksternal**, pencarian dilakukan terhadap sekumpulan data yang disimpan di dalam memori sekunder seperti *tape* atau *disk*. Penyimpanan data di di dalam *storage* bersifat permanen dengan tujuan agar data dapat dibaca kembali untuk pemrosesan lebih lanjut. Bab 1 ini hanya membicarakan pencarian internal saja. Metode pencarian eksternal diluar bahasan buku ini.

1.2 Persoalan Pencarian

Pertama-tama kita spesifikasikan persoalan pencarian elemen di dalam larik. Di dalam Bab ini kita mendefinisikan persoalan pencarian sebagai berikut:

Persoalan (umum)

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 (spek) 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 1.1 : Keluaran hasil pencarian berupa pesan

```
write  (x, 'ditemukan!') atau
write  (x, '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 1.2 : Keluaran hasil pencarian berupa indeks larik

Perhatikan larik pada Gambar 1.1(a).
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 *x* 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 1.3 : Keluaran hasil pencarian berupa nilai boolean

Perhatikan larik pada Gambar 1.1(a).
Misalkan *x* = 68, maka *ketemu* = *true*, dan bila *x* = 100, maka *ketemu* = *false*.

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

Contoh 1.4 : Konsultasi hasil pencarian

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

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

Hal lain yang harus diperjelas dalam spesifikasi persoalan 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 saja yang diacu dan algoritma pencarian selesai. Sebagai contoh, larik pada Gambar 1.1.(a) memiliki tiga buah nilai 36. Bila *x* = 36, maka algoritma pencarian selesai ketika *x* ditemukan pada elemen ke-2 dan menghasilkan *idx* = 2 (atau menghasilkan *ketemu* = *true* jika mengacu pada keluaran pencarian jenis b). Elemen 36 lainnya tidak dipertimbangkan lagi.

Metode pencarian yang akan dibahas di dalam Bab ini adalah:

1. Metode pencarian beruntun (*sequential search*).
2. Metode pencarian bagidua (*binary search*).

Untuk masing-masing algoritma dari kedua metode pencarian di atas, tipe larik yang digunakan dideklarasikan di dalam bagian deklarasi global seperti di bawah ini. Larik *L* adalah bertipe *LarikInt*.

```
{ kamus data global }

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

Algoritma 1.2 Deklarasi larik integer yang digunakan di dalam bab ini.

1.3 Metode Pencarian Beruntun

Di dalam Buku 1 kita telah mempelajari metode pencarian yang paling sederhana, yaitu metode **pencarian beruntun** (*sequential search*). Di dalam Bab ini kita

kemukakan kembali metode pencarian beruntun. Nama lain metode pencarian beruntun adalah **pencarian lurus** (*linear search*).

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

Contoh 1.5 : Pencarian pada sebuah larik

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 dikembalikan: $idx = 4$

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: $idx = -1$

Terdapat dua versi algoritma pencarian beruntun. Pada algoritma versi pertama, aksi pembandingan dilakukan di awal pengulangan, tepatnya pada kondisi pengulangan, sedangkan algoritma versi kedua, aksi pembandingan dilakukan di dalam badan pengulangan. Versi pertama tidak menggunakan peubah *boolean* dalam proses pencarian, sedangkan versi kedua menggunakan peubah *boolean*. Untuk masing-masing versi kita 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) Versi 1 (Pembandingan elemen dilakukan di awal 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 versi 1 untuk kategori hasil berupa nilai *boolean* dapat kita tulis sebagai sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch1(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
```

Algoritma 1.3 Prosedur pencarian beruntun (versi 1, hasil pencarian: boolean)

(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
```

Algoritma 1.4 Fungsi pencarian beruntun (versi 1, hasil pencarian: boolean)

Perhatikanlah bahwa pada algoritma `SeqSearch1` di atas, pembandingan *x* dengan elemen larik dilakukan di awal (bagian kondisional) kalang *while-do*. 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 `SeqSearch1` di atas, elemen terakhir diperiksa secara khusus.

Prosedur `SeqSearch1` 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 dalam 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
```

Algoritma 1.5 Contoh program utama pemanggilan prosedur Pencarian beruntun

Prosedur `BacaLarik` yang disebutkan di dalam program utama di atas algoritmanya adalah 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
```

Algoritma 1.6 Prosedur pembacaan elemen-elemen larik

Untuk pencarian beruntun yang berupa fungsi, œntoh 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 versi 1 untuk kategori hasil berupa indeks elemen larik dapat kita tulis sebagai 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 –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 }
     idx ← i
   else
     idx ← -1
   endif
```

Algoritma 1.7 Prosedur pencarian beruntun (versi 1, hasil pencarian: indeks elemen)

(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
```

Algoritma 1.8 Fungsi pencarian beruntun (versi 1, hasil pencarian: indeks elemen)

Misalkan program yang memanggil prosedur `SeqSearch2` bertujuan untuk menambahkan (*append*) nilai *x* ke dalam larik, namun sebelum penambahan 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       { ukuran 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
```

Algoritma 1.9 Program penambahan elemen larik

Untuk pencarian beruntun yang berupa fungsi, contoh cara pemanggilan fungsi `SeqSearch2`:

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

(b) Versi 2 (Pembandingan elemen dilakukan di dalam badan pengulangan)

Pada versi yang kedua ini, aksi pembandingan dilakukan di dalam badan pengulangan (jadi bukan di awal pengulangan seperti pada varian pertama). Untuk itu, kita memerlukan sebuah peubah *boolean* yang akan berfungsi untuk menyatakan apakah *x* sudah ditemukan. Misalkan peubah tersebut bernama *ketemu* yang pada mulanya diisi nilai *false*. Bila *x* ditemukan pada elemen ke-*i*, yaitu *L*[*i*] = *x*, maka

ketemu diisi dengan nilai *true*. Pengulangan dihentikan bila *ketemu* = *true*. Hasil pencarian disimpulkan di luar badan pengulangan.

Algoritma pencarian beruntun versi 2 lebih elegan dibandingkan dengan algoritma versi 1 karena semua elemen dibandingkan dengan cara yang sama. Algoritma pencariannya kita buat dua macam, yang pertama untuk hasil pencarian berupa berupa peubah *boolean*, dan algoritma kedua untuk hasil pencarian berupa indeks elemen larik.

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

Pada versi ini, peubah boolean *ketemu* diinisialisasi dengan nilai *false* dan indeks larrik *i* diisi dengan 1 (karena pembandingan dimulai dari elemen pertama). Setiap elemen *L* dibandingkan dengan *x* mulai dari elemen pertama. Jika *L*[*i*] sama dengan *x*, peubah *ketemu* diisi nilai *true* dan pengulangan dihentikan. Sebaliknya, jika *L*[*i*] tidak sama dengan *x*, pembandingan dilanjutkan untuk elemen berikutnya ($i \leftarrow i + 1$). Pada versi 2 ini, setiap elemen larik, termasuk elemen terakhir, diperiksa dengan cara yang sama. Keluaran yang dihasilkan adalah nilai yang disimpan di dalam peubah *ketemu*.

Algoritma pencarian beruntun versi 2 untuk kategori hasil berupa nilai *boolean* dapat kita tulis sebagai sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch3(input L : LarikInt, input n : integer,
                     input x : integer, output ketemu : boolean)

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

DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   i ← 1
   ketemu ← false
   while (i ≤ n ) and (not ketemu) do
     if L[i] = x then
        ketemu ← true
     else
        i ← i + 1
     endif
   endwhile
   { i > n or ketemu }
```

Algoritma 1.10 Prosedur pencarian beruntun (versi 2, hasil pencarian: boolean)

(ii) Fungsi pencarian beruntun:

```
function SeqSearch3(input L : LarikInt, input n : integer,
                    input x : integer) → boolean
{ Mengembalikan nilai true jika x ditemukan di dalam larik L[1..n]. Jika
x tidak ditemukan, nilai yang dikembalikan adalah false. }
```

```
DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   i ← 1
   ketemu ← false
   while (i ≤ n ) and (not ketemu) do
     if L[i] = x then
        ketemu ← true
     else
        i ← i + 1
     endif
   endwhile
   { i > n or ketemu }

   return ketemu
```

Algoritma 1.11 Fungsi pencarian beruntun (versi 2, hasil pencarian: boolean)

2. Hasil pencarian: indeks elemen larik yang mengandung nilai *x*.

Algoritmanya sama seperti `SeqSearch3` di atas, hanya saja setelah kalang *while-do* ditambahkan pernyataan *if-then* untuk memberikan hasil pencarian berupa indeks elemen larik (*idx*) yang berisi nilai *x*. Jika *x* tidak ditemukan maka *idx* diisi nilai -1.

Algoritma pencarian beruntun versi 2 untuk kategori hasil berupa indeks elemen larik dapat kita tulis sebagai sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch4(input L : LarikInt, 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 tempat x berada. Jika x tidak
  ditemukan, maka idx diisi nilai -1. }

DEKLARASI
   i : integer        { pencatat indeks larik }
   ketemu : boolean   { true bila x ditemukan, false bila tidak }

ALGORITMA:
    i ← 1
    ketemu ← false
    while (i ≤ n ) and (not ketemu) do
      if L[i] = x then
        ketemu ← true
      else
        i ← i+1
      endif
    endwhile
    { i > n or ketemu }
```

```
    if ketemu then     { x ditemukan }
      idx ← i
    else               { x tidak ditemukan }
      idx ← -1
    endif
```

Algoritma 1.12 Prosedur pencarian beruntun (versi 2, hasil pencarian: indeks elemen)

(i) Fungsi pencarian beruntun:

```
function SeqSearch4(input L : LarikInt, 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 }
   ketemu : boolean   { true bila x ditemukan, false bila tidak }

ALGORITMA:
    i ← 1
    ketemu ← false
    while (i ≤ n ) and (not ketemu) do
      if L[i] = x then
        ketemu ← true
      else
        i ← i+1
      endif
    endwhile
    { i > n or ketemu }

    if ketemu then     { x ditemukan }
      return i
    else               { x tidak ditemukan }
      return -1
    endif
```

Algoritma 1.13 Fungsi pencarian beruntun (versi 2, hasil pencarian: indeks elemen)

Kinerja Metode Pencarian Beruntun

Secara umum, metode pencarian beruntun berjalan lambat. Waktu pencarian sebanding dengan jumlah elemen larik. Misalkan larik berukuran *n* elemen. Maka, pada kasus di mana *x* tidak terdapat di dalam larik atau *x* ditemukan pada elemen yang terakhir, kita harus melakukan perbandingan dengan seluruh elemen larik, yang berarti jumlah perbandingan yang terjadi sebanyak *n* kali. Kita katakan bahwa waktu pencarian dengan algoritma pencarian beruntun sebanding dengan *n*. Bayangkan bila larik berukuran 100.000 buah elemen, maka kita harus melakukan perbandingan sebanyak 100.000 buah elemen. Andaikan satu operasi perbandingan elemen larik membutuhkan waktu 0.01 detik, maka untuk 100.000 buah perbandingan diperlukan waktu sebesar 1000 detik atau 16,7 menit. Semakin banyak

elemen larik, semakin lama pula waktu pencariannya. Karena itulah metode pencarian beruntun tidak bagus untuk volume data yang besar.

Metode Pencarian Beruntun pada Larik Terurut

Larik yang elemen-elemennya sudah terurut dapat meningkatkan kinerja algoritma pencarian beruntun. Jika pada larik tidak terurut jumlah perbandingan elemen larik maksimum *n* kali, maka pada larik terurut (dengan asumsi distribusi elemen-elemen larik adalah seragam atau *uniform*) hanya dibutuhkan rata-rata $n/2$ kali perbandingan. Hal ini karena pada larik yang terurut kita dapat segera menyimpulkan bahwa x tidak terdapat di dalam larik bila ditemukan elemen larik yang lebih besar dari x (pada larik yang terurut menaik) [TEN86].

Contoh 1.6 : Pencarian pada larik terurut

(a) Diberikan larik L tidak terurut:

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

maka, untuk mencari 15, dibutuhkan perbandingan sebanyak 6 kali.

(b) Misalkan larik L di atas sudah diurut menaik:

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

maka, untuk mencari 15, dibutuhkan perbandingan hanya 3 kali (secara rata-rata).

Prosedur `SeqSearch7` berikut adalah algoritma pencarian beruntun pada larik yang terurut menaik, yang mana merupakan modifikasi dari algoritma `SeqSearch1` dengan perubahan pada eskpresi kondisi $L[i] \neq x$ menjadi $L[i] < x$.

```
procedure SeqSearch7(input L : LarikInt, input n : integer,
                     input x : integer, output idx : integer)

{ Mencari keberadaan nilai X di dalam larik L[1..n] yang elemen-elemennya
  sudah terurut menaik. }
{ K.Awal: nilai x dan elemen-elemen larik L[1..n] sudah terdefinisi.
  Elemen-elemen larik L sudah terurut menaik. }
{ K.Akhir: idx berisi indeks larik L yang berisi nilai x. Jika x tidak
  ditemukan, maka idx diisi dengan nilai -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 }
      idx ← i
   else
      idx ← -1
   endif
```

Algoritma 1.14 Pencarian beruntun pada larik terurut

Metode Pencarian Beruntun dengan Sentinel

Jika pencarian bertujuan untuk menambahkan elemen baru setelah elemen terakhir larik, maka terdapat sebuah varian dari metode pencarian beruntun yang mangkus. Nilai *x* yang akan dicari sengaja ditambahkan terlebih dahulu pada elemen ke-*n*+1. Data yang ditambahkan setelah elemen terakhir larik ini disebut **sentinel**. Selanjutnya, pencarian beruntun dilakukan di dalam larik *L*[1..*n*+1]. Akibatnya, proses pencarian selalu menjamin bahwa *x* pasti berhasil ditemukan. Untuk menyimpulkan apakah *x* ditemukan pada elemen sentinel atau bukan, kita dapat mengetahuinya dengan melihat nilai *idx*. Jika *idx* = *n*+1 (yang berarti *x* ditemukan pada elemen sentinel), maka hal itu berarti bahwa *x* tidak terdapat di dalam larik *L* semula (sebelum penambahan sentinel). Keuntungannya, elemen sentinel otomatis sudah menjadi elemen yang ditambahkan ke dalam larik. Sebaliknya, jika *idx* < *n* +1 (yang berarti *x* ditemukan sebelum sentinel), maka hal itu berarti bahwa *x* sudah ada di dalam larik *L* semula.

Contoh 1.7 : Pencarian beruntun dengan menggunakan sentinel

(a) $x = 18$

13	16	14	21	76	15	18	← sentinel
1	2	3	4	5	$n = 6$	7	

18 ditemukan pada elemen ke-*n*+1. Sentinel otomatis sudah ditambahkan ke dalam larik. Ukuran larik sekarang = 7.

(b) $x = 21$

13	16	14	21	76	15	21	← sentinel
1	2	3	4	5	$N = 6$	7	

21 ditemukan pada elemen ke-5. Sentinel batal menjadi elemen yang ditambahkan ke dalam larik. Ukuran larik tetap 6.

Algoritma pencarian beruntun dengan sentinel kita tuliskan sebagai berikut:

```
procedure SeqSearchWithSentinel(input L : LarikInt, input n : integer,
                                input x : integer, output idx : integer)

{ Mencari x di dalam larik L[1..n] dengan menggunakan sentinel}
{ K.Awal: x dan elemen-elemen larik L[1..N] sudah terdefinisi nilainya.}
{ K.Akhir: idx berisi indeks larik L yang berisi nilai x.
  Jika x tidak ditemukan, maka idx diisi dengan nilai –1. }

DEKLARASI
   i : integer   { pencatat indeks larik }

ALGORITMA:
   L[n + 1] ← x    { sentinel }
   i ← 1
   while (L[i] ≠ x) do
      i ← i + 1
   endwhile
   { L[i] = x }   { Pencarian selalu berhasil menemukan x }

   { Kita harus menyimpulkan apakah x ditemukan pada elemen
     sentinel atau bukan }

   if idx = n + 1 then  { x ditemukan pada elemen sentinel }
      idx ← -1   { berarti x belum ada pada larik L semula }
   else    { x ditemukan pada indeks < n + 1 }
      idx ← i
   endif
```

Algoritma 1.15 Pencarian beruntun dengan sentinel

Program utama yang memanggil prosedur `SeqSearchWithSentinel` kira-kira algoritmanya sebagai berikut:

```
ALGORITMA:
   ...          { instruksi-instruksi sebelumnya }
   read(x)
   SeqSearchWithSentinel(L, n, x, idx)

   if idx ≠ -1 then
      write(x, ' sudah terdapat di dalam larik')

   else {x belum terdapat di dalam larik L semula, sentinel
         otomatis menjadi elemen yang ke-n+1. }
      n ← n + 1   { Naikkan ukuran efektif larik }
   endif
```

Algoritma 1.16 Program utama yang memanggil pencarian beruntun dengan sentinel

1.4 Metode Pencarian Bagidua

Pencarian pada data yang terurut menunjukkan kinerja yang lebih baik daripada pencarian pada data yang belum terurut. Hal ini sudah kita bicarakan pada metode pencarian beruntun untuk data yang sudah terurut. Data yang terurut banyak ditemukan di dalam kehidupan sehari-hari. Data nomor telepon di dalam buku telepon misalnya, sudah terurut berdasarkan nama/instansi pelanggan telepon dari A sampai Z. Data karyawan diurut berdasarkan nomor induknya dari nomor kecik ke nomor besar. Data mahasiswa diurut berdasarkan NIM (Nomor Induk Mahasiswa), kata-kata (*entry*) di dalam kamus bahasa Inggris/Indonesia telah diurut dari A sampai Z, dan sebagainya.

Terdapat metode pencarian pada data terurut yang paling mangkus (*efficient*), yaitu metode **pencarian bagidua** atau **pencarian biner** (*binary search*). Metode ini digunakan untuk kebutuhan pencarian dengan waktu yang cepat. Sebenarnya, dalam kehidupan sehari-hari kita sering menerapkan pencarian bagidua. Untuk mencari arti kata tertentu di dalam kamus (misalnya kamus Bahasa Inggris), kita tidak membuka kamus itu dari halaman awal sampai halaman akhir satu per satu, namun kita mencarinya dengan cara membelah atau membagi dua buku itu. Jika kata yang dicari tidak terletak di halaman pertengahan itu, kita mencari lagi di belahan bagian kiri atau belahan bagian kanan dengan cara membagi dua belahan yang dimaksud. Begitu seterusnya sampai kata yang dicari ditemukan. Hal ini hanya bisa dilakukan jika kata-kata di dalam kamus sudah terurut.

Prinsip pencarian dengan membagi data atas dua bagian mengilhami metode pencarian bagidua. Data yang disimpan di dalam larik harus sudah terurut. Untuk memudahkan pembahasan, selanjutnya kita misalkan elemen larik sudah terurut menurun. Dalam proses pencarian, kita memerlukan dua buah indeks larik, yaitu indeks terkecil dan indeks terbesar. Kita menyebut indeks terkecil sebagai indek ujung kiri larik dan indeks terbesar sebagai indeks ujung kanan larik. Istilah "kiri" dan "kanan" dinyatakan dengan membayangkan elemen larik terentang horizontal.
Misalkan indeks kiri adalah *i* dan indeks kanan adalah *j*. Pada mulanya, kita inisialisasi *i* dengan 1 dan *j* dengan *n*.

Langkah 1: Bagi dua elemen larik pada elemen tengah. Elemen tengah adalah elemen dengan indeks $k = (i + j)$ div 2.

(Elemen tengah, $L[k]$, membagi larik menjadi dua bagian, yaitu bagian kiri $L[i..j]$ dan bagian kanan $L[k+1..j]$)

Langkah 2:Periksa apakah $L[k] = x$. Jika $L[k] = x$, pencarian selesai sebab x sudah ditemukan. Tetapi, jika $L[k] \neq x$, harus ditentukan apakah pencarian akan dilakukan di larik bagian kiri atau di bagian kanan. Jika $L[k] < x$, maka pencarian dilakukan lagi pada larik bagian kiri. Sebaliknya, jika $L[k] > x$, pencarian dilakukan lagi pada larik bagian kanan.

Langkah 3:Ulangi Langkah 1 hingga x ditemukan atau $i > j$ (yaitu, ukuran larik sudah nol!)

Contoh 1.8 : Pencarian elemen dengan metode bagidua

Ilustrasi pencarian bagidua: Misalkan diberikan larik L dengan delapan buah elemen yang sudah terurut menurun seperti di bawah ini:

81	76	21	18	16	13	10	7
i =1	2	3	4	5	6	7	8=j

(i) Misalkan elemen yang dicari adalah x = 18.

Langkah 1:

$i = 1$ dan $j = 8$
Indeks elemen tengah $k = (1 + 8)$ div $2 = 4$ (elemen yang diarsir)

81	76	21	18	16	13	10	7
1	2	3	4	5	6	7	8

kiri kanan

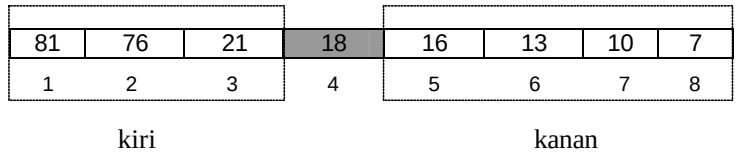
Langkah 2:

Pembandingan: $L[4] = 18$? Ya! (x ditemukan, proses pencarian selesai)

(ii) Misalkan elemen yang dicari adalah x = 16.

Langkah 1:

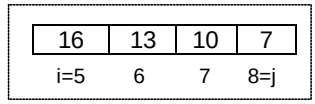
$i = 1$ dan $j = 8$
Indeks elemen tengah $k = (1 + 8)$ div $2 = 4$ (elemen yang diarsir)



Langkah 2:

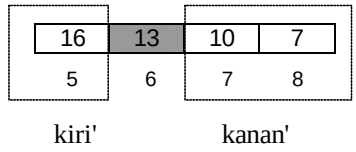
Pembandingan: $L[4] = 16$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

Pembandingan: $L[4] > 16$? Ya! Lakukan pencarian pada larik bagian kanan dengan $i = k + 1 = 5$ dan $j = 8$ (tetap)



Langkah 1':

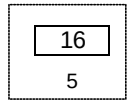
$i = 5$ dan $j = 8$
Indeks elemen tengah $k = (5 + 8)$ div $2 = 6$ (elemen yang diarsir)



Langkah 2':

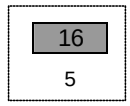
Pembandingan: $L[6] = 16$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

Pembandingan: $L[6] > 16$? Tidak! Lakukan pencarian pada larik bagian kiri dengan $i = 5$ (tetap) dan $j = k - 1 = 5$



Langkah 1":

$i = 5$ dan $j = 5$
Indeks elemen tengah $k = (5 + 5)$ div $2 = 5$ (elemen yang diarsir)



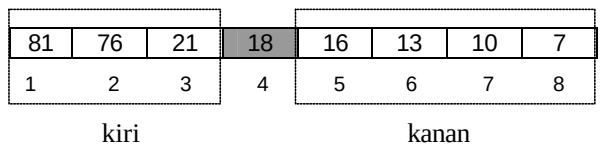
Langkah 2":

L[5] = 16? Ya! (*x* ditemukan, proses pencarian selesai)

(iii) Misalkan elemen yang dicari adalah *x* = 100

Langkah 1:

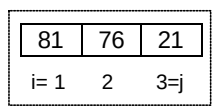
i = 1 dan *j* = 8
Indeks elemen tengah *k* = (1 + 8) div 2 = 4 (elemen yang diarsir)



Langkah 2:

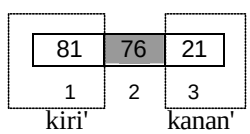
Pembandingan: *L*[4] = 100? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

Pembandingan: *L*[4] > 100? Tidak! Lakukan pencarian pada larik bagian kiri dengan *i* = 1 (tetap) dan *j* = *k* – 1 = 3



Langkah 1':

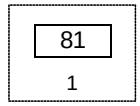
i = 1 dan *j* = 3
Indeks elemen tengah *k* = (1 + 3) div 2 = 2 (elemen yang diarsir)



Langkah 2':

Pembandingan: $L[2] = 100$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

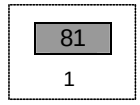
Pembandingan: $L[2] > 100$? Tidak! Lakukan pencarian pada larik bagian kiri dengan $i = 1$ dan $j = k - 1 = 1$



Langkah 1":

$i = 1$ dan $j = 1$

Indeks elemen tengah $k = (1 + 1)$ div $2 = 1$ (elemen yang diarsir)



Langkah 2":

Pembandingan: $L[1] = 100$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

Pembandingan: $L[1] > 100$? Tidak! Lakukan pencarian pada larik bagian kiri dengan $i = 1$ dan $j = k - 1 = 0$

Karena $i > j$, maka tidak ada lagi bagian larik yang tersisa. Dengan demikian, x tidak ditemukan di dalam larik. Proses pencarian dihentikan.

Algoritma pencarian bagidua pada larik *integer* yang sudah terurut menurun kita tuliskan di bawah ini, masing-masing sebagai prosedur dan sebagai fungsi.

(i) Prosedur pencarian bagidua:

```
procedure BinarySearch1(input L : LarikInt, input n : integer,
                        input x : integer, output idx : integer)

{ Mencari  x di dalam larik L[1..n] yang sudah terurut menurun dengan
  metode pencarian bagidua. Keluaran prosedur ini adalah indeks elemen
  larik yang mengandung nilai x; idx diisi 0 jika x tidak ditemukan. }
{ K.Awal : L[1..n] sudah berisi data yang sudah terurut menurun, dan x
  adalah nilai yang akan dicari.}
{ K.Akhir: idx berisi indeks elemen larik yang mengandung nilai x; tetapi
  bila x tidak ditemukan, maka idx diisi dengan -1 }

DEKLARASI
  i, j : integer      { indeks kiri dan indek kanan larik }
  k : integer         { indeks elemen tengah}
  ketemu : boolean    { flag untuk menentukan apakah X ditemukan }

ALGORITMA:
   i ← 1      { ujung kiri larik }
   j ← n      { ujung kanan larik }
   ketemu ← false   { asumsikan x belum ditemukan }
   while (not ketemu) and (i ≤ j) do
      k ←(i + j) div 2  {bagidua larik L pada posisi k}
      if (L[k] = x) then
         ketemu ← true
      else  { L[k] ≠ x }
         if (L[k] > x) then
           { Lakukan pencarian pada larik bagian kanan, set indeks ujung
             kiri larik yang baru }
            i ← k + 1
         else
           { Lakukan pencarian pada larik bagian kiri, set indeks ujung
             kanan larik yang baru }
            j ← k - 1
         endif
      endif
   endwhile
   { ketemu = true or i > j}

   if ketemu then  { x ditemukan }
      idx ← k
   else            { x tidak ditemukan di dalam larik  }
      idx ← -1
   endif
```

Algoritma 1.17 Prosedur pencarian bagidua pada larik yang terurut menurun

(ii) Fungsi pencarian bagidua:

```
function BinarySearch1(input L : LarikInt, input n : integer,
                       input x : integer) → integer

{ Mengembalikan indeks elemen larik L[1..n] yang mengandung nilai x;
  bila x tidak ditemukan, maka indek yang dikembalikan adalah -1 }

DEKLARASI
  i, j : integer      { indeks kiri dan indek kanan larik }
  k : integer         { indeks elemen tengah}
  ketemu : boolean    { flag untuk menentukan apakah X ditemukan }

ALGORITMA:
   i ← 1      { ujung kiri larik }
   j ← n      { ujung kanan larik }
   ketemu ← false   { asumsikan x belum ditemukan }
   while (not ketemu) and (i ≤ j) do
      k ←(i + j) div 2  {bagidua larik L pada posisi k}
      if (L[k] = x) then
         ketemu ← true
      else  { L[k] ≠ x }
         if (L[k] > x) then
           { Lakukan pencarian pada larik bagian kanan, set indeks ujung
             kiri larik yang baru }
            i ← k + 1
         else
           { Lakukan pencarian pada larik bagian kiri, set indeks ujung
             kanan larik yang baru }
            j ← k - 1
         endif
      endif
   endwhile
   { ketemu = true or i > j}

   if ketemu then  { x ditemukan }
      return k
   else            { x tidak ditemukan di dalam larik  }
      return -1
   endif
```

Algoritma 1.18 Prosedur pencarian bagidua pada larik yang terurut menurun

Untuk algoritma pencarian bagidua pada larik yang sudah terurut menaik, kita hanya perlu mengganti pernyataan `if (L[k] > x)` dengan `if L[k] < x)`.

Kinerja Metode Pencarian Bagidua

Pembaca dapat melihat bahwa pada setiap kali pencarian, larik dibagidua menjadi dua bagian yang berukuran hampir sama. Pada setiap pembagian, elemen tengah dibandingkan apakah sama dengan *x* (`if (L[k] = x)`). Pada kasus terburuk, yaitu pada kasus *x* tidak terdapat di dalam larik atau *x* ditemukan setelah ukuran larik tinggal 1 elemen, larik akan dibagi sebanyak $^2\log(n)$ kali, sehingga jumlah pembandingan yang dibutuhkan adalah sebanyak $^2\log(n)$ kali. Kita katakan bahwa waktu pencarian sebanding dengan $^2\log(n)$ [WIR76]. Untuk n = 256 elemen misalnya, kasus terburuk menghasilkan pembagian larik sebanyak $^2\log(256) = 8$ kali,

yang berarti jumlah pembandingan elemen adalah 8 kali (bandingkan dengan metode pencarian beruntun yang pada kasus terburuk melakukan pembandingan sebanyak 256 kali). Jadi, untuk larik yang terurut, algoritma pencarian bagidua jauh lebih cepat daripada algoritma pencarian beruntun.

1.5 Pencarian pada Larik Terstruktur

Metode pencarian yang dibahas sebelum ini menggunakan larik dengan elemen-elemen bertipe sederhana. Pada kebanyakan kasus, elemen larik sering bertipe terstruktur. Contoh, misalkan *M* adalah sebuah larik yang elemennya menyatakan nilai ujian seorang mahasiswa untuk suatu mata kuliah (*MK*) yang ia ambil. Data setiap mahasiswa adalah *NIM* (Nomor Induk Mahasiswa), nama mahasiswa, mata kuliah yang ia ambil, dan nilai mata kuliah tersebut. Deklarasi nama dan tipe adalah sebagai berikut:

```
DEKLARASI
  const Nmaks = 100
  type Mahasiswa : record <NIM : integer,  { Nomor Induk Mahasiswa }
                           NamaMhs : string,  { nama mahasiswa }
                           KodeMK   : string,  { kode mata kuliah }
                           Nilai: char  { indeks nilai MK (A/B/C/D/E) }
                          >

  type TabMhs : array[1..Nmaks] of Mahasiswa

  M : TabMhs
```

Algoritma 1.19 Pendeklarasian larik terstruktur

Struktur lojik larik *M* ditunjukkan pada Gambar 1.2.

TabMhs

	NIM	NamaMhs	KodeMK	Nilai
1	29801	Heru Satrio	MA211	A
2	29804	Amirullah Satya	FI351	B
3				
.				
.				
100	29887	Yanti Siregar	TL321	C

Gambar 1.2 Larik *M* dengan 100 elemen. Setiap elemen larik bertipe terstruktur (*record*). Tiap record terdiri atas field *NIM*, field *Nama*, field *MK*, dan field *Nilai*.

Misalkan pencarian data didasarkan pada *NIM*, maka proses pembandingan dilakukan terhadap *field NIM* saja. Algoritma pencarian beruntun dan algoritma pencarian bagidua untuk larik tertstruktur diberikan di bawah ini (untuk algoritma pencarian beruntun, kita gunakan algoritma Versi 2 dengan hasil pencarian adalah

indeks larik, sedangkan untuk algoritma pencarian bagidua kita gunakan pencarian pada larik yang terurut menaik).

(a) Algoritma Pencarian Beruntun

```
procedure CariNIM_1(input M : TabMhs, input n : integer,
                    input NIMmhs : integer, output idx : integer)

{ Mencari keberadaan NIMmhs di dalam larik M[1..n] dengan metode
  pencarian beruntun. }
{ K.Awal : nilai NIMmhs, n, dan elemen larik M[1..n] sudah terdefinisi.}
{ K.Akhir: idx berisi indeks larik tempat NIMmhs ditemukan,
  idx = -1 jika NIMmhs tidak  ditemukan }

DEKLARASI
   i : integer    { pencatat indeks larik }
   ketemu : boolean

ALGORITMA:
   i ← 1
   ketemu ← false
   while (i ≤ n ) and (not ketemu) do
     if M[i].NIM = NIMmhs then
       ketemu ← true
     else
       i ← i + 1
     endif
   endwhile
    { i > n or ketemu }

    if ketemu then { NIMmhs ditemukan }
      idx ← i
    else   { NIMmhs tidak ditemukan }
      idx ← -1
    endif
```

Algoritma 1.20 Pencarian beruntun pada larik terstruktur

(b) Algoritma Pencarian Bagidua

```
procedure CariNIM_2(input M : TabMhs, input n : integer,
                    input NIMmhs : integer, output IDX : integer)
```

```
{ Mencari NIMmhs di dalam larik M[1..n] yang sudah terurut menaik
  berdasarkan NIM dengan metode pencarian bagidua. }
{ K.Awal : Larik M[1..n] sudah berisi data yang sudah terurut menaik, dan
  NIMmhs adalah nilai yang akan dicari }
{ K.Akhir: idx berisi indeks larik tempat NIMmhs ditemukan, idx = -1 jika
  NIMmhs tidak  ditemukan }

DEKLARASI
  i, j : integer          { indeks kiri dan indeks kanan }
  k : integer             { indeks elemen tengah }
  ketemu : boolean        { flag untuk menentukan ketemu atau tidak }

ALGORITMA:
   i ← 1      { ujung kiri larik }
   j ← n      { ujung kanan larik }
   ketemu ← false   { asumsikan x belum ditemukan }

   while (not ketemu) and (i ≤ j) do

      k ← (i + j) div 2  { bagidua larik L pada posisi k }
      if (M[k].NIM = NIMmhs) then
        ketemu ← true
      else  { M[k].NIM ¹ NIMmhs }

         if  (M[k].NIM < NIMmhs) then
           { Lakukan pencarian pada larik bagian kanan, set indeks
             ujung kiri larik yang baru }
            i ← k + 1
         else
           { Lakukan pencarian pada larik bagian kiri, set indeks
             ujung kanan larik yang baru }
            j ← k - 1
         endif
      endif
   endwhile
   { ketemu = true or i > j}

   if (ketemu) then  { NIMmhs ditemukan }
      idx ← k
   else              { NIMmhs tidak ditemukan di dalam larik  }
      idx ← -1
   endif
```

Algoritma 1.21 Pencarian bagidua pada larik terstruktur

1.6 Menggunakan Metode Pencarian Beruntun atau Metode Pencarian Bagidua?

Kedua metode pencarian ini mempunyai kelebihan dan kekurangan masing-masing. Metode pencarian beruntun dapat digunakan baik untuk data yang belum terurut maupun untuk data yang sudah terurut. Sebaliknya, metode pencarian bagidua hanya dapat digunakan untuk data yang sudah terurut saja.

Ditinjau dari kinerja pencarian, kita sudah mengetahui bahwa untuk kasus terburuk (yaiyu jika pencarian gagal menemukan x), algoritma pencarian beruntun memerlukan waktu yang sebanding dengan n (banyaknya data), sedangkan algoritma pencarian membutuhkan waktu yang sebanding dengan $^2\log(n)$. Karena $^2\log(n) < n$ untuk $n > 1$, maka jika n semakin besar waktu pencarian dengan algoritma bagidua jauh lebih sedikit daripada waktu pencarian dengan algoritma beruntun. Karena itulah, algoritma pencarian bagidua lebih baik untuk mencari data pada sekumpulan nilai yang sudah terurut ketimbang algoritma pencarian beruntun.

Sebagai perbandingan antara algoritma pencarian beruntun dengan pencarian bagidua, tinjaulah kasus di mana x tidak ditemukan di dalam larik yang sudah terurut. Misalkan,

(a) untuk larik yang berukuran $n = 256$ elemen

- algoritma pencarian beruntun melakukan pembandingan elemen larik sebanyak 256 kali,
- algoritma pencarian bagidua melakukan pembandingan sebanyak $^2\log(256) = 8$ kali,

(b) untuk larik yang berukuran $n = 1024$ elemen

- algoritma pencarian beruntun melakukan pembandingan elemen larik sebanyak 1024 kali,
- algoritma pencarian bagidua melakukan pembandingan sebanyak $^2\log(1024) = 10$ kali.

1.7 Pencarian pada Larik yang Tidak Bertipe Numerik

Meskipun algoritma-algoritma pencarian yang sudah dikemukakan di atas diterapkan pada larik *integer*, mereka tetap benar untuk larik data yang bertipe bukan numerik misalnya data bertipe karakter, maupun *string*. Aculah kembali dari Buku 1 bahwa operasi perbandingan (dengan operator $<, >, \neq$) juga berlaku pada tipe data karakter maupun *string*. Jadi, pembandingan karakter seperti 'a' < 'b' atau pembandingan *string* seperti 'budi' < 'iwan' adalah ekspresi relasional yang sah.

1.8 Algoritma Pencarian Beruntun dan Pencarian Bagidua dalam Bahasa PASCAL dan Bahasa C

Dengan asumsi bahwa pembaca buku ini sudah memahami konversi dari notasi algoritmik ke notasi Bahasa Pascal dan Bahasa C (baca Buku 1), di bawah ini disajikan algoritma pencarian yang telah dikemukakan di atas dalam kedua bahasa tersebut.

ALGORITMIK

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

PASCAL

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

C

```
/* DEKLARASI */
    #define Nmaks  100                /* jumlah elemen larik */
    typedef int LarikInt [Nmaks+1];   /* indeks larik dimulai dari 0
                sampai Nmaks, tetapi elemen ke-0 tidak akan digunakan*/
```

1. Algoritma Pencarian Beruntun

ALGORITMIK

```
procedure SeqSearch2(input L : LarikInt, 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 –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 }
     idx ← i
   else
     idx ← -1
   endif
```

PASCAL

```
procedure SeqSearch2(L : LarikInt; n : integer; x : integer;
                     var 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 –1. }

var
   i : integer;    { pencatat indeks larik }

begin
   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 := -1;       { x tidak ditemukan }
   {endif}
end;
```

C

```
void SeqSearch2(LarikInt L, int n, int x, int *idx)
/* Mencari keberadaan nilai X di dalam larik L[1..N]. */
/* K.Awal:  nilai x dan elemen-elemen larik L[1..N] sudah terdefinisi. */
/* K.Akhir: idx berisi indeks larik L yang berisi x. Jika X tidak
   ditemukan, idx diisi dengan nilai -1. */

{
   int i;      /* pencatat indeks larik */

   i = 1;
   while (i < n && L[i] != x)
       i++;
   /* endwhile */
   /* i == n or L[i] == x */

   if (L[i] == x)    /* x ditemukan */
     *idx = i;
   else             /* x tidak ditemukan */
     *idx = -1;
   /*endif*/

}
```

2. Algoritma Pencarian Bagidua

ALGORITMIK

```
procedure BinarySearch2(input L : LarikInt, input n : integer,
                        input x : integer, output idx : integer)

{ Mencari  x di dalam larik L[1..n] yang sudah terurut menurun dengan
  metode pencarian bagidua. Keluaran prosedur ini adalah indeks elemen
  berisi nilai x.  }
{ K.Awal : Larik L[1..n] sudah berisi data yang sudah terurut menurun,
  dan x adalah harga yang akan dicari. }
{ K.Akhir: idx berisi indeks larik tempat x ditemukan, idx = -1 jika x
  tidak ditemukan }

DEKLARASI
  i,j : integer       { indeks kiri dan indek kanan larik }
  k : integer         { indeks elemen tengah}
  ketemu : boolean    { pertanda untuk menentukan apakah x ditemukan }

ALGORITMA:
   i ← 1    { ujung kiri larik }
   j ← n    { ujung kanan larik }
   ketemu ← false   { asumsikan x belum ditemukan }
   while (not ketemu) and (i ≤ j) do
      k ← (i + j) div 2  { bagidua larik L pada posisi k }
      if (L[k] = x) then
        ketemu ← true
      else  { L[k] ¹ x }
        if (L[k] > x) then
           { Lakukan pencarian pada larik bagian kanan, set indeks
             ujung kiri larik yang baru }
```

```
            i ← k + 1
         else
           { Lakukan pencarian pada larik bagian kiri, set indeks
             ujung kanan larik yang baru }
            j ← k - 1
         endif
      endif
   endwhile
   { ketemu = true or i > j}

   if (ketemu) then  { x ditemukan }
      idx ← k
   else             { X tidak ditemukan di dalam larik  }
      idx ← -1
   endif
```

PASCAL

```
procedure BinarySearch2(L : LarikInt; n : integer; x : integer;
                        var idx : integer);

{ Mencari x di dalam larik L[1..n] yang sudah terurut menurun dengan
  metode pencarian bagidua. Keluaran prosedur ini adalah indeks elemen
  berisi nilai x.  }
{ K.Awal : Larik L[1..N] sudah berisi data yang sudah terurut menurun,
  dan x adalah harga yang akan dicari. }
{ K.Akhir: idx berisi indeks larik tempat x ditemukan; idx = -1 jika x
  tidak ditemukan. }

var
  i,j : integer;      { indeks kiri dan indek kanan larik }
  k : integer;        { indeks elemen tengah}
  ketemu : boolean;   { flag untuk menentukan apakah X ditemukan }

begin
   i := 1;   { ujung kiri larik }
   j := n;   { ujung kanan larik }
   ketemu := false;  { asumsikan x belum ditemkan }
   while (not ketemu) and (i <= j) do
    begin
      k:=(i + j) div 2;  { bagidua larik L pada posisi k }
      if (L[k] = x) then
        ketemu:= true
      else  { L[k] <> x }
         if  (L[k] > x) then
           { Lakukan pencarian pada larik bagian kanan, set indeks
             ujung kiri larik yang baru }
            i := k + 1
         else
           { Lakukan pencarian pada larik bagian kiri, set indeks
             ujung kanan larik yang baru }
            j := k – 1;
         {endif}
      {endif}
   end{while};
   { ketemu = true or i > j}

   if (ketemu) then  { x ditemukan }
```

```
    idx := k
  else              { x tidak ditemukan di dalam larik  }
    idx := -1;
  {endif}
end;
```

C

```
void BinarySearch2(LarikInt L, int n, int X, int *idx)

/* Mencari x di dalam larik L[1..n] yang sudah terurut menurun dengan
   metode pencarian bagidua. Keluaran prosedur ini adalah indeks elemen
   berisi nilai x.  */
/* K.Awal : larik L[1..n] sudah berisi data yang sudah terurut menurun,
   dan x adalah harga yang akan dicari. */
/* K.Akhir: idx berisi indeks larik tempat X ditemukan; idx = -1 jika x
   tidak ditemukan */

{
  typedef enum{true = 1, false = 0} boolean; /* deklarasi tipe boolean */

  int i,j;          /* indeks kiri dan indeks kanan larik */
  int k;            /* indeks elemen tengah */
  boolean ketemu;    /* flag untuk menentukan apakah X ditemukan */

   i = 1;           /* ujung kiri larik */
   j = n;           /* ujung kanan larik */
   ketemu = false;  /* asumsikan x belum ditemukan */
   while (!ketemu && i <= j)
    {
      k = (i + j)/2;  /* bagidua larik L pada posisi k */
      if (L[k] == x)
        ketemu = true;
      else  /* L[k] != x */

         if  (L[k] > x)
           /*  akan  dilakukan  pencarian  pada  larik  bagian  kanan,  set
              indeks ujung kiri larik yang baru */
            i = k + 1;
         else
           /* akan dilakukan pencarian pada larik bagian kiri,
              set indeks ujung kanan larik yang baru */
            j = k - 1;
         /*endif*/
    /*endif*/
   }
   /* endwhile*/
   /* ketemu = true || i > j */

   if (ketemu)   /* x ditemukan */
      *idx = k;
   else         /* x tidak ditemukan di dalam larik  */
      *idx = -1;
   /*endif*/
}
```

Soal Latihan

1. Tulislah kembali algoritma pencarian beruntun yang memberikan hasil indeks elemen larik yang mengandung *x*, tetapi pencarian dimulai dari elemen terakhir.

2. [TEN86] Algoritma *SeqSearch8* berikut hanya membutuhkan satu ekspresi perbandingan pada awal kalang *while-do*. Realisasikan algoritma tersebut ke dalam Bahasa *Pascal* dan Bahasa *C*.

```
procedure SeqSearch8(input L : LarikInt, 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
  nilainya.}
{ K.Akhir: idx berisi indeks larik L yang berisi x; jika x tidak
  ditemukan, idx diisi dengan nilai -1. }

DEKLARASI
   i   : integer       { pencatat indeks larik }
   pos : integer

ALGORITMA:
    i ← 1
    pos ← n + 1
    while (i ≠ pos) do
      if x = L[i] then
        pos ← i
      else
        i ← i + 1
      endif
    endwhile
    { i = pos }

    if i > n then     { x tidak ditemukan }
      idx ← -1
    else              { x ditemukan }
      idx ← pos
    endif
```

3. Masalah apa yang terjadi jika algoritma pencarian beruntun kita tulis seperti berikut ini:

```
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          { x tidak ditemukan }
    idx ← -1
endif
```

4. Nyatakan algoritma pencarian bagidua berupa fungsi yang mengembalikan indeks elemen larik yang berisi nilai *x*.

5. [PAR95] [TEN86] Metode **pencarian interpolasi** (*interpolation search*) merupakan varian dari metode pencarian bagidua, di mana elemen berikutnya yang akan dibandingkan didasarkan pada nilai indek *i* dan *j*. Misalnya jika *x* "diperkirakan" terletak pada posisi bagian antara *L*[*i*] dan *L*[*j*], maka kita set *k* dengan nilai bagian antara *i* dan *j*. Memperkirakan posisi *x* di dalam larik disebut menginterpolasi. Rumus untuk menghitung *k* kita tulis sebagai berikut:

```
k ← i + (j – i)*(x – L[i]) div (L[j] – L[i])
```

(Metode pencarian interpolasi mengasumsikan data di dalam larik terdistrubusi *uniform*. Metode ini pada dasarnya sama dengan metode pencarian bagidua, kecuali dalam penentuan elemen yang diperiksa berikutnya). Tulislah algoritma pencarian interpolasi.