BAB V

Rekursi

Tujuan

1. Memahami rekursi sebagai konsep yang dapat digunakan untuk merumuskan solusi sederhana dalam sebuah permasalahan yang sulit untuk diselesaikan secara iteratif dengan menggunakan loop `for, while do`.
2. Membantu pembaca bagaimana "berpikir secara rekursif"
3. Dapat menyelesaikan suatu permasalahan dengan konsep rekursi

Rekursi adalah konsep pengulangan yang penting dalam ilmu komputer. Konsep ini dapat digunakan untuk merumuskan solusi sederhana dalam sebuah permasalahan yang sulit untuk diselesaikan secara iteratif dengan menggunakan loop `for, while do`. Pada saat tertentu konsep ini dapat digunakan untuk mendefinisikan permasalahan dengan konsisten dan sederhana. Pada saat yang lain, rekursi dapat membantu untuk mengekspresikan algoritma dalam sebuah rumusan yang menjadikan tampilan algoritma tersebut mudah untuk dianalisa.

5.1 Rekursi Dasar

Rekursi mempunyai arti suatu proses yang bias memanggil dirinya sendiri. Dalam sebuah rekursi sebenarnya tekandung pengertian sebuah prosedur atau fungsi. Perbedaannya adalah bahwa rekursi bisa memanggil dirinya sendiri, kalau prosedur atau fungsi harus diipanggil melalui pemanggil prosedur atau fungsi.

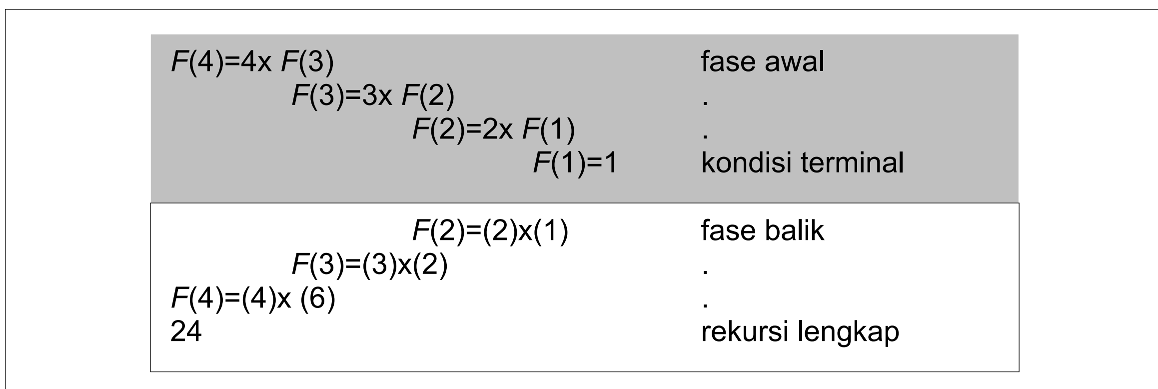
Untuk memulai bahasan rekursi, kita membahas sebuah masalah sederhana yang kemungkinan kita tidak berpikir untuk menyelesaikan dengan cara rekursif. Yaitu permasalahan faktorial, yang mana kita menghitung hasil faktorial dari sebuah bilangan, yaitu n. Faktorial dari n (ditulis n!), adalah hasil kali dari bilangan tersebut dengan bilangan di bawahnya, di bawahnya hingga bilangan 1. Sebagai contoh, 4! = (4)(3)(2)(1). Salah satu cara untuk menghitung adalah dengan menggunakan loop, yang mengalikan masing-masing bilangan dengan hasil sebelumnya. Penyelesaian dengan cara ini dinamakan iteratif, yang mana secara umum dapat didefinisikan sebagai berikut:

$n! = (n)(n-1)(n-2) \ldots$ (1)

Cara lain untuk menyelesaikan permasalahan di atas adalah dengan cara rekursi, dimana *n!* adalah hasil kali dari n dengan *(n-1)!*. Untuk menyelesaikan *(n-1)!* adalah sama dengan *n!*, sehingga *(n-1)!* adalah *n-1*dikalikan dengan *(n-2)!*, dan *(n-2)!* adalah *n-2* dikalikan dengan *(n-3)!* dan seterusnya sampai dengan $n = 1$, kita menghentikan penghitungan *n!* Cara rekursif untuk permasalahan ini, secara umum dapat kita detailkan sebagai berikut:

$$F(n) = \begin{cases} 1 & \text{jika n=0, n=1} \\ nF(n-1) & \text{jika n>1} \end{cases}$$

Pada Gambar 5.1 dibawah ini digambarkan penghitungan 4! dengan menerapkan konsep rekursi. Dalam gambar tersebut juga digambarkan dua fase dasar dari sebuah proses rekursi: fase awal dan fase balik. Dalam fase awal, masing-masing proses memanggil dirinya sendiri. Fase awal ini berhenti ketika pemanggilan telah mencapai kondisi terminal. Sebuah kondisi teminate adalah kondisi dimana sebuah fungsi rekursi kembali dari pemanggilan, artinya fungsi tersebut sudah tidak memanggil dirinya sendiri dan kembali pada sebuah nilai. Sebagai contoh, dalam penghitungan faktorial dari n, kondisi terminal adalah $n = 1, n = 0$. Untuk setiap fungsi rekursi, minimal harus ada satu kondisi terminal. Setelah fase awal selesai, kemudian proses melanjutkan pada fase balik, dimana fungsi sebelumnya akan dikunjungi lagi dalam fase balik ini. Fase ini berlanjut sampai pemanggilan awal, hingga secara lengkap proses telah berjalan.



Gambar 5.1 Proses Komputasi Secara Rekursif dari 4!

Dalam Program 5.1 ditampilkan sebuah fungsi dalam C, *fact_rec*, dengan parameter sebuah bilangan n dan menghitung faktorial secara rekursi. Fungsi tersebut bekerja sebagai berikut. Jika n kurang dari 0, maka fungsi kembali ke 0, menunjukkan kesalahan. Jika n = 0 atau 1, fungsi kembali ke 1 karena 0! dan 1! hasilnya 1. Dan ini adalah kondisi terminal. Selainnya itu, fungsi kembali dengan hasil *n* kali factorial dari *n-1*. Faktorial dari n-1 adalah penghitungan kembali secara rekursif dengan memanggil *fact* lagi. Perhatikan persamaan dari definisi rekursi dengan implementasi di bawah ini.

```
int fact_rec(int n)
{
/*************************************************************
*       Menghitung sebuah faktorial secara rekursif          *
*************************************************************/
if (n < 0)
    return 0;
else if (n == 0)
    return 1;
else if (n == 1)
    return 1;
else
    return n * fact(n-1);
}
```

Program 5.1 Fungsi Implementasi dari Proses Komputasi Faktorial secara Rekursif

Jika kita perhatikan dari fungsi rekursi di atas, maka kita lihat adanya kondisi terminal yang kita cantumkan dalam program tersebut, yaitu jika n = 0 atau n = 1. Hal ini harus kita lakukan dalam setiap fungsi rekursif. Jika tidak, maka eksekusi tidak akan pernah berhenti. Akibatnya memori tumpukan akan habis dan komputer akan hang.

Selain itu kita dapat membandingkan proses rekursi di atas dengan proses iteratif, seperti di bawah ini:

```
int fact_it (int n)
{
    int temp;
/*********************************************************
*      Menghitung sebuah faktorial dengan proses looping    *
*********************************************************/
temp = 1;
if (n < 0)
    return 0;
else if (n == 0)
    return 1;
else if (n == 1)
    return 1;
else
    for (i=2; i<=n; ++i)
       temp = temp * i;
    return (temp);
}
```

Program 5.2 Fungsi Implementasi dari Proses Komputasi Faktorial secara Iteratif

5.2 Rekursi Tail

Sebuah proses rekursi dikatakan rekursi tail jika pernyataan terakhir yang akan dieksekusi berada dalam tubuh fungsi dan hasil yang kembali pada fungsi tersebut bukanlah bagian dari fungsi tersebut. Ciri fungsi rekursi tail adalah fungsi tersebut tidak

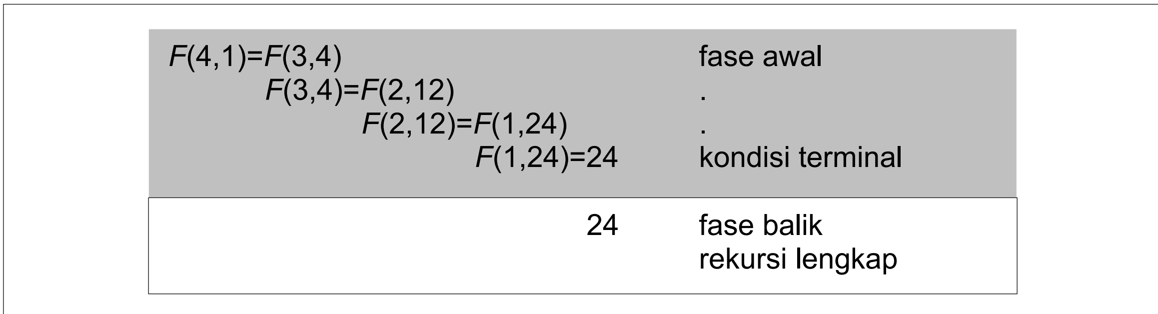
memiliki aktivitas selama fase balik. Ciri ini penting, karena sebagian besar kompiler modern secara otomatis membangun kode untuk menuju pada fase tersebut.

Ketika kompiler mendeteksi sebuah pemanggilan yang mana adalah rekursi tail, maka kompiler menulis aktivitas yang ada sebagai sebuah record yang dimasukkan ke dalam stack. Kompiler dapat melakukan hal tersebut karena pemanggilan rekursi adalah pernyataan terakhir yang dieksekusi dalam aktivitas yang sedang berlangsung, sehingga tidak ada aktivitas yang harus dikerjakan pada saat pemanggilan kembali.

Untuk memahami bagaimana sebuah rekursi tail bekerja, kita akan kembali membahas tentang penghitungan komputasi dari faktorial secara rekursi. Sebagai langkah awal, akan sangat membantu jika kita memahami alasan dalam definisi awal yang bukan rekursi tail. Dimana dalam definisi tersebut, penghitungan n! adalah dengan mengalikan n dengan (n-1)! dalam setiap aktivitas, yang mana hal tersebut terus diulang dari n = n –1 sampai n = 1. Definisi ini bukanlah rekursi tail karena nilai yang dikembalikan dalam setiap aktivitas bergantung pada perkalian n dengan nilai yang dikembalikan oleh aktivitas sesudahnya. Oleh karena itu, pencatatan aktivitas dari masing-masing pemanggilan harus diingat dalam stack sampai nilai-nilai yang dikembalikan dalam aktivitas-aktivitas sesudahnya telah terdefinisi. Sekarang marilah kita melihat bagaimana rekursi tail didefinisikan untuk menghitung n!, yang secara formal kita definisikan sebagai berikut:

$$F(n,a) = \begin{cases} a & \text{jika } n=0,\ n=1 \\ F(n-1,na) & \text{jika } n>1 \end{cases}$$

Pada definisi ini digunakan parameter kedua, yaitu a, yang mana merupakan nilai utama dari penghitungan faktorial secara rekursif. Hal ini mencegah kita untuk mengalikan nilai yang dikembalikan dalam setiap aktivitas dengan n. Dalam masing-masing pemanggilan rekursi kita mendefinisikan a = na dan n = n – 1. Kita melanjutkan sampai n = 1, sebagai kondisi terminal. Dalam gambar 5.2 menggambarkan proses komputasi 4! dengan menggunakan rekursi tail. Perhatikan bahwa di sana tidak ada aktivitas selama fase balik.



Gambar 5.2 Proses Komputasi 4! dengan Rekursi Tail

Program 5.3 mempresentasikan sebuah fungsi dalam bahasa C, yang menerima sebuah bilangan n dan menghitung faktorialnya dengan menggunakan rekursi tail. Fungsi ini juga menerima parameter tambahan, yaitu a, untuk initial pertama diset 1. Fungsi ini hampir sama dengan fact pada Program 5.1, kecuali dia menggunakan a untuk mengelola lebih jauh nilai dari komputasi faktorial dalam proses rekursi.

```
/*************************************************************
*                   Fungsi Rekursi Tail                      *
*************************************************************/
int facttail(int n, int a)
{
/*************************************************************
*     Menghitung sebuah faktorial dengan rekursi tail        *
*************************************************************/
if (n < 0)
    return 0;
else if (n == 0)
    return 1;
else if (n == 1)
    return a;
else
```

```
        return facttail(n-1,n*a);
    }
```

Program 5.3 Fungsi Implementasi Proses Komputasi Faktorial 4! dengan Rekursi Tail

5.3 Kesimpulan

1. Rekursi adalah kemampuan suatu rutin untuk memanggil dirinya sendiri.
2. Penggunaan sebuah rekursi harus mendefinisikan kondisi terminal, jika tidak eksekusi program tidak pernah berhenti, sehingga penggunaan memori tumpukan habis dan komputer akan hang.
3. Dalam beberapa situasi, pemecahan secara rekursif mempunyai keuntungan dan kekurangan yang dapat diperbandingkan dengan cara iteratif.

5.4 Latihan

1. Tuliskan fungsi penghitung bilangan Fibonacci menggunakan pengulangan, bukan menggunakan rekursi. Setelah itu bandingkan kecepatannya dengan fungsi yang menggunakan rekursi untuk n yang cukup besar, misalnya untuk n = 30.
2. Buatlah sebuah fungsi yang menulis angka dari n ke 0 dengan menggunakan proses rekursi.
3. Tulis sebuah fungsi untuk menulis anka dari 0 ke n dengan menggunakan proses rekursi.
4. Tuliskan fungsi rekursi untuk membalik suatu kalimat. Sebagai contoh, kalimat 'Struktur Data' dibalik menjadi 'ataD rutkurtS'.

   Fungsi rekursi ini menerima parameter bertipe string dan mengembalikan string hasil pembalikan.
5. Tulis sebuah fungsi yang melakukan pengecekan apakah sebuah angka merupakan bilangan prima atau bukan (n bukan bilangan prima jika dapat dibagi dengan angka kurang dari n)
6. Tuliskan program untuk menampilkan semua permutasi dari N karakter, 'A', 'B', 'C', dan seterusnya dimana 2 <= N <= 10. Sebagai contoh, untuk N =2, program menampilkan:

   ```
   A B
   B A
   ```

   Contoh lain, untuk N = 3, program menampilkan:

   ```
   A B C
   ```

```
A C B
B A C
B C A
C A B
C B A
```

Secara umum, banyaknya permutasi dari N buah karakter adalah N faktorial. Dalam contoh di atas N = 3, sehingga banyaknya permutasi adalah 3!=6.

Proses penyusunan permutasinya adalah sebagai berikut:

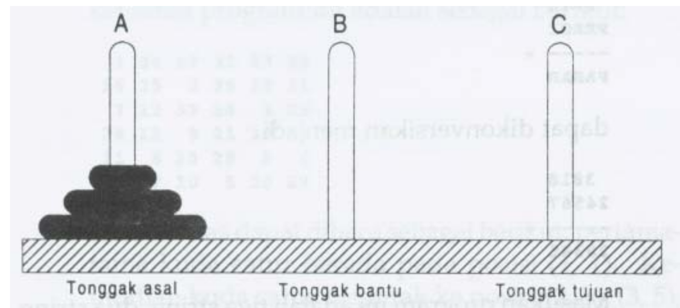
- cetak elemen ke-1, dan cetak permutasi elemen ke-2 sampai ke-N (permutasi dengan N-1 elemen)
- cetak elemen ke-2, dan cetak permutasi elemen ke-1, elemen ke-3 sampai ke-N (permutasi dengan N-1 elemen)
- cetak elemen ke-3, dan cetak permutasi elemen ke-1, elemen ke-2, elemen ke-4 sampai ke-N (permutasi dengan N-1 elemen)
- dan seterusnya sampai langkah terakhir adalah cetak elemen ke-N, dan cetak permutasi elemen ke-1 sampai elemen ke (N-1) (permutasi dengan N-1 elemen)

Proses diatas diulang terus sampai dicetak permutasi dengan 1 elemen. Sebagai contoh, untuk N = 3, caranya adalah sebagai berikut:

- cetak 'A', dan cetak permutasi ('B' 'C')
- cetak 'B', dan cetak permutasi ('A' 'C')
- cetak 'C', dan cetak permutasi ('A' 'B')

Dari proses diatas nampak bahwa banyaknya elemen yang dipermutasikan makin lama makin sedikit sampai sama dengan 1. Kondisi ini yang kita pakai sebagai batas proses.

7. Tuliskan program untuk menyelesaikan permainan Menara Hanoi. Permainan Menara Hanoi dimainkan menggunakan tiga tonggak, yaitu tonggak asal, tonggak tujuan dan tonggak bantu. Tujuan dari permainan ini adalah memindahkan piringan-piringan dari tonggak asal ke tonggak tujuan menggunakan tonggak bantu. Syarat pemindahan adalah jumlah piringan yang boleh dipindah dalam satu waktu hanya satu dan piringan yang kecil harus berada di piringan yang besar. Dengan kata lain, piringan besar tidak boleh berada di atas piringan kecil.



Gambar 5.3 Permainan Menara Hanoi

Masukan dari program ini adalah banyaknya piringan yang harus dipindah, dimana 3 <= piringan <=9.

Keluaran dari program ini adalah langkah yang harus dilakukan untuk memindahkan piringan-piringan tersebut, misalnya:

```
A pindah ke C
A pindah ke B
C pindah ke B
A pindah ke C
B pindah ke A
B pindah ke C
A pindah ke C
```