BAB I

SUBALGORITMA REKURSIF

1.1. Pendahuluan

Dalam mata kuliah Algoritma dan Pemrograman C++ Dasar telah dibahas mengenai subalgoritma yaitu algoritma yang berdiri sendiri, dapat menerima data, memroses data dan mengembalikan hasil proses, lalu mengembalikan kontrol program ke instruksi algoritma yang memanggilnya. Dalam bahasa pemrograman C++ subalgoritma dikenal dengan nama fungsi (function). Suatu subalgoritma dapat dipanggil oleh atau memanggil subalgoritma lain seperti contoh berikut.

```
FUNGSI V(T)
Fungsi untuk menghitung tegangan kapasitor pada waktu T.
Parameter T dan VT berjenis real.
{
1. [Hitung tegangan kapasitor]
   VT = (T + 0.1) * EXP(SQRT(T))
2. [Mengembalikan hasil ke program yang memanggil]
   Return (VT)
}
```

Di dalam fungsi V(T) di atas, untuk menghitung tegangan kapasitor dipanggil fungsi EXP() yang lalu memanggil fungsi SQRT(). Dalam hal ini hasil dari satu fungsi digunakan oleh fungsi yang lain. Secara umum, proses yang terjadi di dalam subalgoritma adalah menerima data melalui parameter fungsi, memproses data, lalu mengembalikan hasil proses ke instruksi algoritma yang memanggilnya.

Pada subalgoritma rekursif, terjadi proses yang berbeda. Ketika subalgoritma rekursif dipanggil maka subalgoritma ini kemudian akan memanggil dirinya sendiri secara rekursif (berulang-ulang) sampai mendapatkan hasil lalu mengembalikan hasil proses ke instruksi algoritma yang memanggilnya. Jadi subalgoritma rekursif adalah subalgoritma yang memanggil dirinya sendiri ketika memroses data yang diberikan kepadanya. Dengan demikian, masalah yang dapat

diselesaikan dengan sub-algoritma rekursif ini terbatas pada masalah yang menggunakan struktur pengulangan dalam proses penyelesaiannya.

1.2. Konsep Rekursif

Cara yang paling mudah untuk menjelaskan konsep rekursif ini adalah dengan menggunakan contoh masalah, lalu diselesaikan dengan menggunakan sub-algoritma yang telah dipelajari dan dengan subalgoritma rekursif sehingga dapat dibandingkan antara keduanya. Sebagai contoh yang pertama adalah membuat fungsi untuk menuliskan kalimat "Ini adalah fungsi iteratif" sebanyak n kali dengan menggunakan perintah pengulangan yang telah dipelajari dan menuliskan "Ini adalah fungsi rekursif" dengan menggunakan konsep rekursif.

```
Void Pesan_Iteratif(n)
{
1.  For (a = 1; a <= n; a ++)
1.a.    {Write ("Ini adalah fungsi iteratif")}
}
```

Hasil dari fungsi di atas adalah menampilkan tulisan "Ini adalah fungsi iteratif" sebanyak n kali sesuai dengan nilai parameter fungsi yang juga menjadi variabel pengontrol pengulangan (lihat statement For pada langkah 1 di atas). Untuk masalah yang sama dapat dibuat fungsi dengan menggunakan konsep rekursif seperti pada fungsi berikut ini.

```
Void Pesan_Rekursif(n)
{
1.      if (n >=1)
        {
1.a.            Write ("Ini adalah fungsi rekursif")
1.b.            Pesan_Rekursif(n-1)
        }
}
```

Hasil dari fungsi di atas juga menampilkan tulisan "Ini adalah fungsi rekursif" sebanyak n kali tetapi dengan cara yang berbeda. Perhatikan, pada fungsi di atas tidak terdapat statement FOR yang mengontrol pengulangan tetapi terdapat pemanggilan fungsi **Pesan_Rekursif (n-1)** pada langkah 1.b. Hal inilah yang

disebut sebagai rekursif, yaitu fungsi yang memanggil dirinya sendiri. Jika diperhatikan ada perbedaan antara parameter fungsi yang di dalam (lihat statement no 1.b.) dengan paramenter fungsi pada definisi fungsi. Parameter fungsi yang di dalam, range-nya yang lebih kecil dari pada range parameter fungsi yang di luar, yaitu **(n – 1)**. Hal inilah yang akan membuat pemanggilan fungsi rekursif berhenti karena setiap kali terjadi pemanggilan fungsi, *range* akan makin kecil. Pada contoh di atas, pemanggilan fungsi secara rekursif akan berhenti pada saat **n < 1**. Kondisi ini disebut ***base case***. *Base case* harus selalu ada dalam subalgoritma rekursif, kalau tidak maka pemanggilan subalgoritma tidak dapat berhenti.

Sebagai contoh yang kedua, akan dibuat fungsi untuk menjumlahkan kuadrat bilangan m sampai n, dengan harga awal m ≤ n sesuai persamaan berikut:

$$Sumsq(m,n) = m^2 + (m+1)^2 + (m+2)^2 + \ldots + n^2$$

Masalah ini akan diselesaikan dengan 2 cara yaitu:

1. Dengan menggunakan subalgoritma yang berisi struktur pengulangan yang telah dipelajari sebelumnya, dalam hal ini disebut solusi dengan cara iteratif:

```
Fungsi SUMSQ_ITERATIF(M,N)
Fungsi untuk menjumlahkan kuadrat bilangan M sampai N secara
iteratif dengan menggunakan struktur pengulangan. M, N, i dan
sum adalah variabel integer.
{
1. [inisialisasi variabel sum]
   sum = 0
2. [membuat struktur pengulangan]
   For ( i = M ; i <= N ; i ++ )
2.a.  {      sum = sum + i * i
      }
3. [mengembalikan hasil]
   Return (sum)
}
```

Untuk penyelesaian secara iteratif ini, solusi akhir dibangun secara bertahap dengan menggunakan proses pengulangan (lihat langkah ke 2) sehingga sedikit demi sedikit mendekati solusi akhir. Untuk jelasnya dapat dilihat pada tabel 1.1 tabel telusur untuk M = 5 dan N = 10 di halaman berikut.

Tabel 1.1 Tabel telusur untuk pemanggilan fungsi SUMSQ_ITERATIF(5,10)

Langkah	M	N	i	sum
1	5	10	?	0
2	5	10	5	0
2.a	5	10	5	25
2	5	10	6	25
2.a	5	10	6	61
2	5	10	7	61
2.a	5	10	7	110
2	5	10	8	110
2.a	5	10	8	174
2	5	10	9	174
2.a	5	10	9	225
2	5	10	10	225
2.a	5	10	10	355
2	5	10	11	355
3	kembali ke algoritma utama			

2. Dengan menggunakan sub-algoritma rekursif:

   Untuk penyelesaian secara rekursif, dapat dimulai dengan mencari base case, yaitu dengan mencari pada rumus di atas bagian mana yang tidak memerlukan proses pengulangan untuk mendapatkan solusi. Dalam hal ini adalah jika nilai m == n, sehingga langsung didapat solusi: Sumsq(m,m) = m^2 . Dengan demikian *base case*-nya adalah m == n. Selanjutnya, tentukan cara untuk mencapai base case jika n > m. Caranya adalah dengan menambahkan nilai m dengan 1 beberapa kali sehingga akhirnya menjadi sama dengan n. Dengan demikian, range untuk memanggil fungsi secara rekursif dapat ditulis menjadi Sumsq(m+1, n). Solusi dengan menggunakan subalgoritma rekursif adalah:

```
Fungsi SUMSQ_REC(M,N)
Fungsi untuk menjumlahkan kuadrat bilangan M sampai N secara
rekursif. M dan N adalah variabel integer.
{
1.  If  ( M < N )
1.a.    { return (M*M + SUMSQ_REC(M+1, N)) }
    else
1.b.    { return (M*M) }
```

Dari fungsi di atas, dapat dilihat bahwa solusi akhir didapat dengan memecahkan masalah menjadi beberapa submasalah yang lebih kecil. Dalam contoh ini, ditunjukkan dengan mengecilnya range dari m ke n menjadi dari

m+1 ke n, dan seterusnya. Lalu solusi dari masing-masing submasalah tersebut kemudian digabungkan menjadi solusi akhir. Untuk jelasnya lihat langkah 1.a pada fungsi di atas, yang menunjukkan pemanggilan fungsi secara rekursif, yaitu fungsi memanggil dirinya sendiri tetapi dengan parameter yang lebih mendekati nilai akhir (dalam hal ini: M + 1).

Statement pada langkah 1.b., menunjukkan instruksi yang tidak memanggil instruksi (fungsi) lain atau disebut ***Base Case***, solusi langsung didapat tanpa perlu memanggil fungsi. *Base case* disebut juga *termination condition* yaitu kondisi yang mengakhiri proses pemanggilan sub-algoritma rekursif. Pada contoh ini, *base case*-nya adalah jika nilai M == N maka pemanggilan fungsi berhenti dan langsung didapat solusi untuk fungsi sumsq_rec ini yaitu M * M.

Berbeda dengan fungsi iteratif yang menggunakan tabel telusur untuk menelusuri pemanggilan fungsi. Pada subalgoritma rekursif, proses pemanggilannya ditelusuri dengan 2 cara yaitu dengan *Call trace* dan *Call tree*. *Call trace* adalah penelusuran pemanggilan fungsi sedang *call tree* adalah gambar pohon pemanggilan fungsi. Untuk jelasnya, kedua cara tersebut akan digunakan untuk menulusuri pemanggilan fungsi SUMSQ_REC untuk M = 5 dan N = 10.

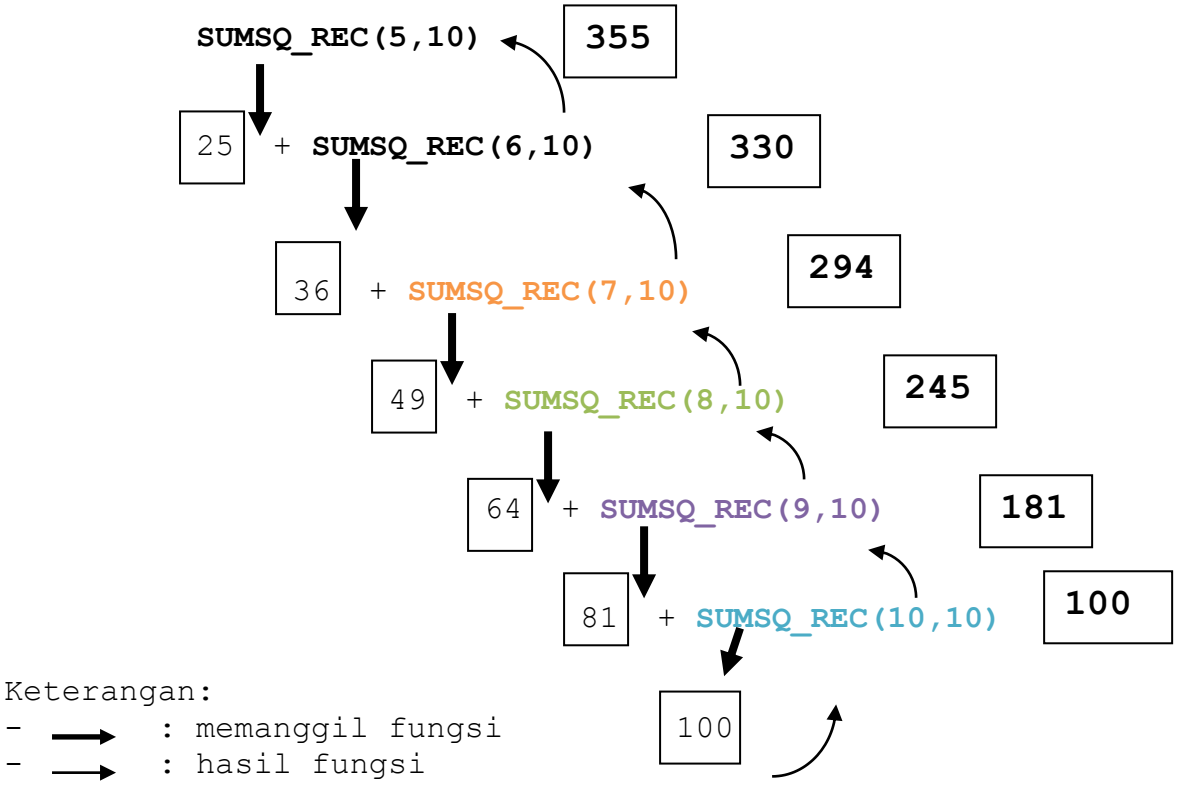
Call trace untuk fungsi SUMSQ_REC(5,10):

```
SUMSQ_REC(5,10)=
        (25 + SUMSQ_REC(6,10))
      = (25 + (36 + SUMSQ_REC(7,10)))
      = (25 + (36 + (49 + SUMSQ_REC(8,10))))
      = (25 + (36 + (49 + (64 + SUMSQ_REC(9,10)))))
      = (25 + (36 + (49 + (64 + (81 + SUMSQ_REC(10,10))))))
      = (25 + (36 + (49 + (64 + (81 + 100)))))
      = (25 + (36 + (49 + (64 + 181))))
      = (25 + (36 + (49 + 245)))
      = (25 + (36 + 294))
      = (25 + 330)
      = 355
```

Pada *call trace* di atas, dapat dilihat bahwa, terjadi pemanggilan fungsi dengan argumen m yang nilainya makin mendekati nilai akhir (m = 6, 7, ...), sampai pada saat argumen m mencapai nilai 10 atau m == n, tidak lagi terjadi pemanggilan fungsi tetapi langsung didapat hasilnya (dalam hal ini bagian *else* dari fungsi yang

dieksekusi). Kondisi inilah yang mengakhiri pemanggilan fungsi rekursif (*base case*) sehingga hasil fungsi dapat langsung dihitung.

Call tree untuk SUMSQ_REC(5,10) dapat dilihat pada gambar 1.1 di bawah ini. Dalam gambar call tree ini lebih jelas terlihat proses pemanggilan fungsi dan proses pengembalian hasil fungsi. Angka yang dicetak tebal adalah hasil pemanggilan fungsi.



Gambar 1.1 Call tree untuk SUMSQ_REC(5,10)

1.3 Macam-macam Proses Rekursif

Berdasarkan cara untuk mencapai base case, ada 3 cara untuk membuat subalgoritma rekursif, yaitu rekursif naik (*going up recursion*), rekursif turun (*going down recursion*) dan rekursif bagi-dua (*division in half recursion*). Rekursif naik adalah subalgoritma rekursif yang solusinya dibangun mulai dari awal *range* lalu ke *subrange* sampai ke akhir *range*. Fungsi SUMSQ_REC(M,N) di atas adalah contoh dari rekursif naik. Nilai M di *increment* terus sehingga mencapai nilai N. Sebaliknya yang terjadi untuk rekursif turun. Untuk menjelaskan proses rekursif ini, akan dibahas contoh-contoh solusi dengan sub algoritma rekursif berikut ini.

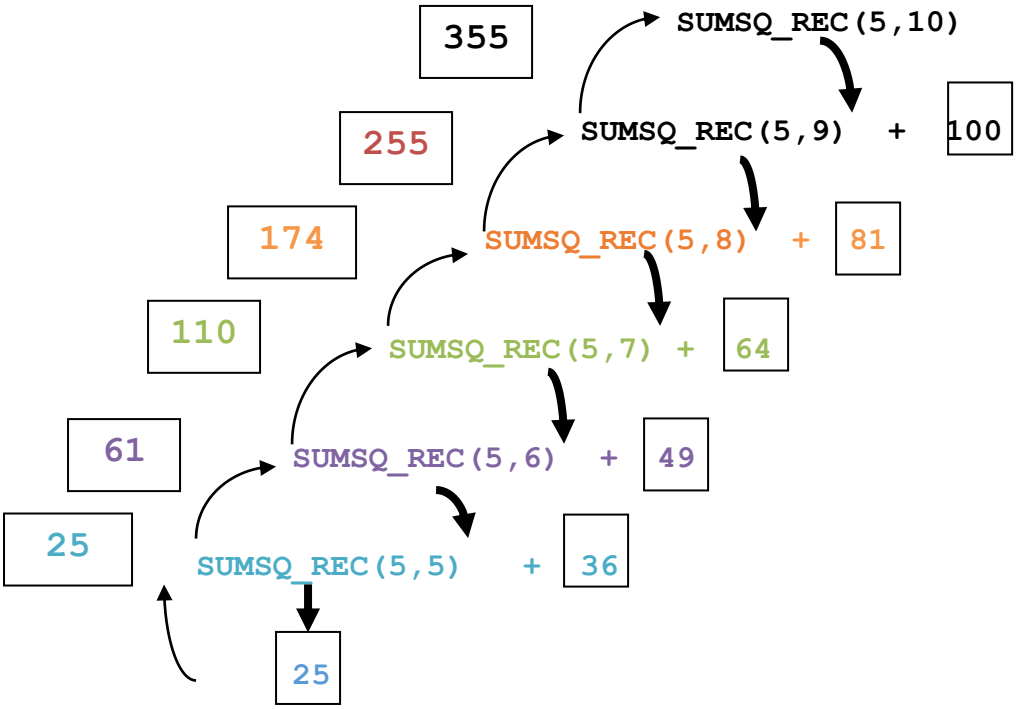
Contoh solusi secara rekursif turun:

```
Fungsi SUMSQ_REC_2(M,N)
Fungsi untuk menjumlahkan kuadrat bilangan M sampai N secara
rekursif turun. M dan N adalah variabel integer.
1.  If  ( M < N )
      { return (SUMSQ_REC_2(M, N-1) + N*N) }
    else { return (N*N) }
```

Pada subalgoritma rekursif di atas, solusi diperoleh mulai dari akhir *range* (N) lalu ke *subrange* (M .. N-1), (M .. N-2),…, (M .. M). Atau dengan kata lain, nilai N di *decrement* sampai mencapai M. Perhatikan, *base case* untuk sub-algoritma rekursif SUMSQ_REC_2 ini adalah jika M == N maka langsung didapat: N*N. Bandingkan dengan *base case* pada fungsi SUMSQ_REC yang sebelumnya.

Untuk melihat perbedaan lain antara rekursif naik dengan rekursif turun, dapat dilihat pada *call trace* dari fungsi SUMSQ_REC_2 untuk M = 5 dan N =10 berikut ini. Dan *call tree* dari pemanggilan fungsi ini dapat dilihat pada gambar 1.2 di halaman berikut.

```
SUMSQ_REC_2(5,10) =
        (SUMSQ_REC_2(5,9) + 100)
      = ((SUMSQ_REC_2(5,8) + 81) + 100)
      = (((SUMSQ_REC_2(5,7) + 64) + 81) + 100)
      = ((((SUMSQ_REC_2(5,6) + 49) + 64) + 81) + 100)
      = (((((SUMSQ_REC_2(5,5) + 36) + 49) + 64) + 81) + 100)
      = (((((25 + 36) + 49) + 64) + 81) + 100)
      = ((((61 + 49) + 64) + 81) + 100)
      = (((110 + 64) + 81) + 100)
      = ((174) + 81) + 100)
      = (255 + 100)
      = 355
```



Keterangan:
- ⟶ : memanggil fungsi
- ⟶ : hasil fungsi

Gambar 1.2 Call tree untuk SUMSQ_REC_2(5,10)

Contoh solusi secara rekursif bagi-dua:

```
Fungsi SUMSQ_REC_3(M,N)
Fungsi untuk menjumlahkan kuadrat bilangan M sampai N secara
rekursif bagi-dua. M, N dan mid adalah variabel integer.
{
1.  If  ( M == N )
    { Return (M*M) }
    else
    { mid = (M+N) div 2
      return(SUMSQ_REC_3(M,mid)+SUMSQ_REC_3(mid+1,N)) }
 }
```

Pada sub algoritma rekursif di atas, solusi diperoleh dengan membagi 2 *range* yaitu mulai dari awal sampai ke tengah *range* (M .. mid) dan dari tengah *range* ke akhir *range* (mid + 1 .. N), lalu masing-masing *range* dibagi lagi menjadi *subrange* (M .. mid1) dan (mid1 + 1 .. mid) serta *subrange* (mid +1 .. mid2) dan (mid2 +1 .. N), demikian seterusnya sampai bertemu dengan *base case*. Perhatikan,

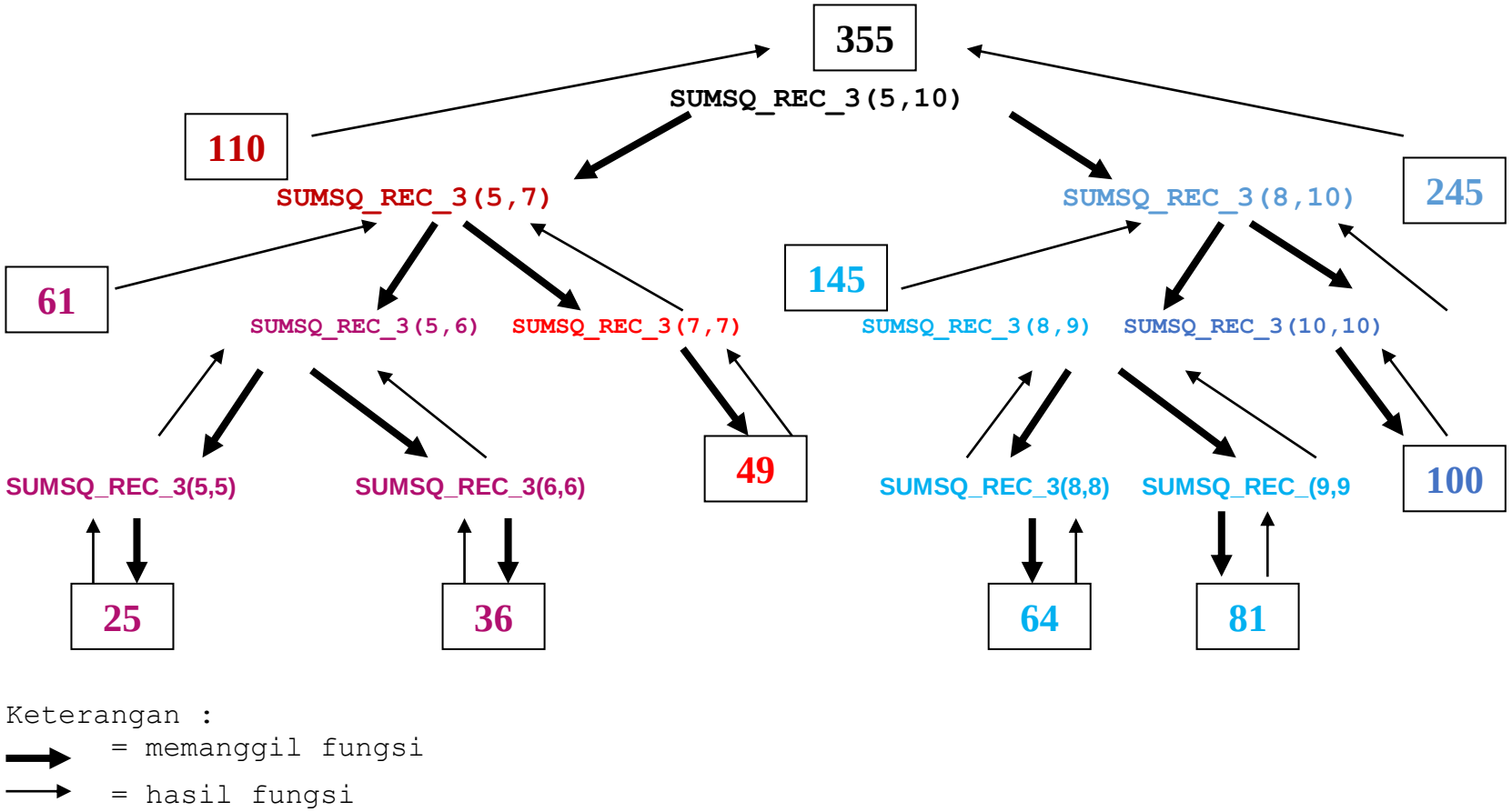
base case untuk sub algoritma rekursif SUMSQ_REC_3 ini adalah jika M == N atau jika awal *range* sama dengan akhir *range* sehingga langsung didapat solusinya yaitu : M*M.

Untuk melihat perbedaannya dengan proses rekursif yang lain, dapat dilihat pada *call trace* berikut ini dan gambar 1.3 untuk *call tree* dari fungsi SUMSQ_REC_3 untuk M = 5 dan N =10.

```
SUMSQ_REC_3(5,10) =
       SUMSQ_REC_3(5,7) + SUMSQ_REC_3(8,10)
     = (SUMSQ_REC_3(5,6) + SUMSQ_REC_3(7,7)) +
          (SUMSQ_REC_3(8,9) + SUMSQ_REC_3(10,10))
     = ( (SUMSQ_REC_3(5,5) + SUMSQ_REC_3(6,6)) + 49)
          +( (SUMSQ_REC_3(8,8) + SUMSQ_REC_3(9,9)) + 100)
     = ((25 + 36) + 49) + ((64 + 81) + 100)
     = ((61 + 49) + (145 + 100))
     = 110 + 245
     = 355
```

Bila gambar 1.3 dibandingkan dengan gambar 1.1 dan 1.2, dapat dilihat bahwa *call tree* pada gambar 1.3 lebih “pendek” dari pada call tree lainnya. Tetapi banyaknya pemanggilan fungsi untuk mencapai *base case* lebih banyak, yaitu 10 kali pada gambar 1.3 dan hanya 5 kali pada gambar 1.1 dan 1.2. Sebagai catatan, tidak semua algoritma rekursif dapat menggunakan ketiga proses ini.

Dari ketiga cara rekursif di atas, dapat dilihat bahwa ketiganya mempunyai base case yang sama yaitu m == n. Dan pemanggilan fungsi akan berhenti ketika nilai parameter sudah mencapai base case.



Gambar 1.3 Call tree untuk fungsi SUMSQ_REC_3(5,10)

1.4. Contoh-contoh Masalah

Seperti telah dijelaskan di awal bab, masalah yang dapat dipecahkan dengan menggunakan subalgoritma rekursif terbatas pada masalah yang menggunakan struktur pengulangan (*loop*) dalam proses penyelesaiannya. Dan tidak semua masalah dapat diselesaikan dengan ke tiga cara rekursif seperti telah dijelaskan sebelumnya. Ada 2 contoh masalah yang akan dibahas disini yaitu mencari faktorial dan mencari deret bilangan Fibonacci.

1.4.1. Mencari faktorial dari suatu bilangan

Faktorial dari suatu bilangan (n) didefinisikan sebagai berikut:

$$Faktorial(n) = n \times (n-1) \times \ldots \times 2 \times 1$$

Penyelesaian secara iteratif dengan menggunakan struktur pengulangan dapat dilihat pada fungsi Faktorial_Iteratif berikut ini.

```
Fungsi FAKTORIAL_ITERATIF(N)
Fungsi untuk menghitung faktorial dari bilangan N. F adalah
variabel lokal untuk menyimpan hasil perhitungan. I adalah
variabel lokal untuk counter. Semua variabel berjenis integer.
{
1. [inisialisasi variabel]
   F = 1
2. [membuat struktur pengulangan untuk menghitung faktorial]
   For ( i = 1 ; i <= N ; i ++ )
   {   F = F * i     }
3. [mengembalikan hasil]
   Return (F)
}
```

Untuk penyelesaian secara rekursif, tentukan lebih dulu base case-nya. Dari fungsi di atas, dapat dilihat jika n == 1, maka faktorialnya dapat langsung diketahui tanpa melalui proses. Dengan demikian, n==1 dapat digunakan sebagai *base case*.

```
Fungsi FAKTORIAL_REK(N)
Fungsi untuk menghitung faktorial dari bilangan N secara
rekursif. Semua variabel berjenis integer.
{
1. If ( N == 1 )
   { return (1) }
   else
   { return(N * FAKTORIAL_REK(N-1)) }
   }
```

Dibandingkan dengan penyelesaian secara iteratif, maka fungsi FAKTORIAL_REK ini tidak memerlukan variabel lokal sama sekali. Pemanggilan fiungsi secara rekursif menyebabkan hasil fungsi tidak dapat diketahui sampai bertemu dengan base case. Untuk jelasnya lihat proses pemanggilan fungsi rekursif (*call trace*) dari fungsi FAKTORIAL_REK untuk N = 5 berikut ini:

```
Faktorial_Rek(5)=(5*Faktorial_Rek(4))
                =(5*(4*Faktorial_Rek(3)))
                =(5*(4*(3*Faktorial_Rek(2))))
                =(5*(4*(3*(2*Faktorial_Rek(1)))))
                =(5*(4*(3*(2*1))))
                =(5*(4*(3*2)))
                =(5*(4*6))
                =(5*24)
                = 120
```

Untuk masalah Faktorial ini, tidak dapat dibuat proses rekursif naik atau bagi-dua seperti contoh sebelumnya karena *base case*-nya terletak pada N == 1.

1.4.2. Mencari deret bilangan FIBONACCI

Deret bilangan Fibonacci dari suatu bilangan (N) didefinisikan sebagai berikut:

$$\mathbf{FIBONACCI(N)} = \begin{cases} \mathbf{FIBONACCI(N-1)+FIBONACCI(N-2)} & \mathbf{;jika\ N > 2} \\ \mathbf{1} & \mathbf{;\ jika\ N = 2} \\ \mathbf{1} & \mathbf{;\ jika\ N = 1} \end{cases}$$

Dari definisi di atas dapat disimpulkan bahwa nilai suatu suku dari deret bilangan Fibonacci ditentukan oleh dua suku sebelumnya.

Penyelesaian secara iteratif dengan menggunakan struktur pengulangan dan variabel lokal untuk menyimpan hasil adalah sebagai berikut:

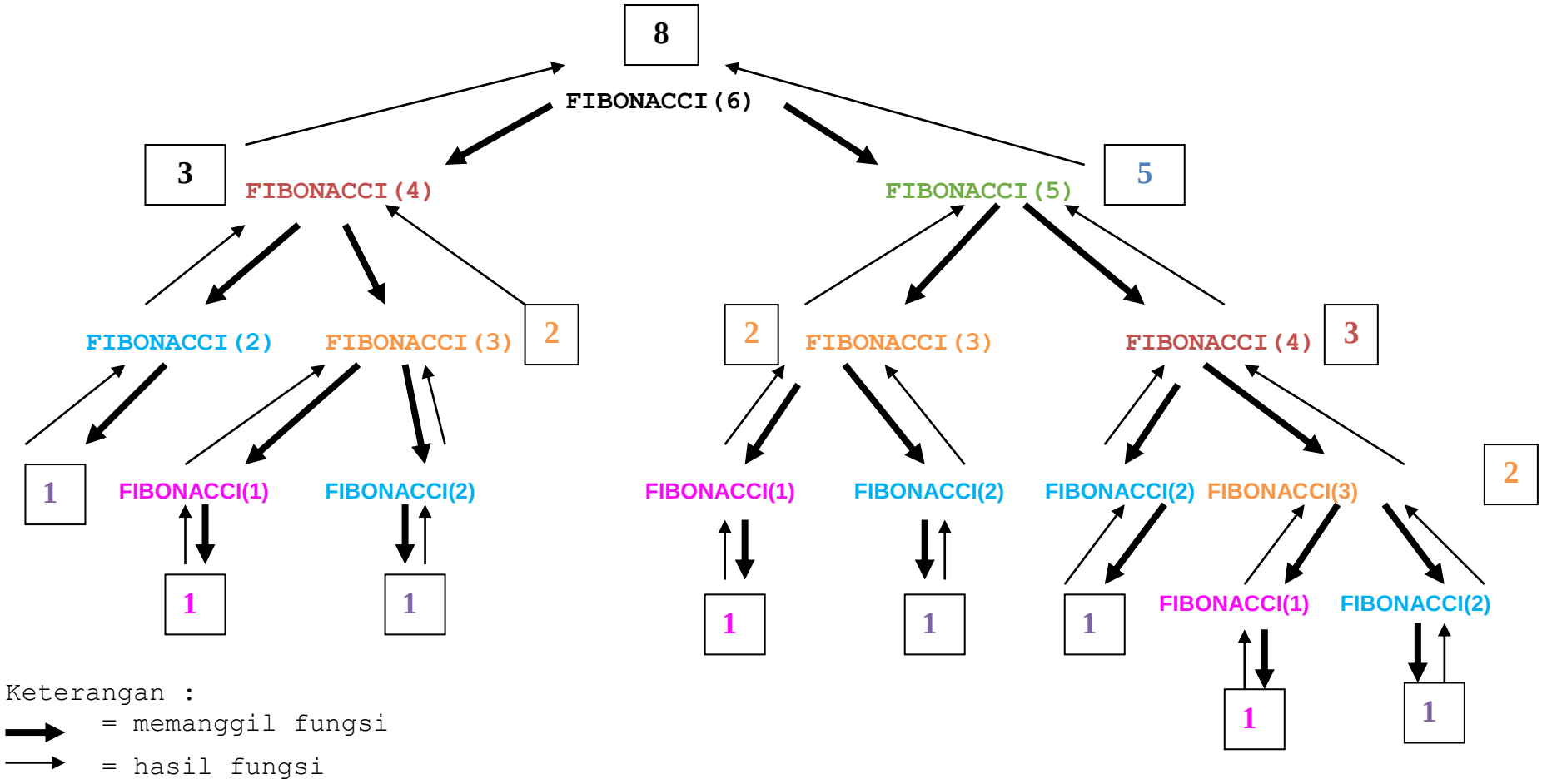
```
Fungsi FIBONACCI_ITERATIF(F, N)
Fungsi untuk menghitung deret Fibonacci dari bilangan N. F, FI, F2
adalah variabel lokal untuk menyimpan hasil perhitungan. i adalah
variabel lokal untuk counter. Semua variabel berjenis integer.
{
1.  If ( N == 1 )
    {       return (1) }
    else
    {       if ( N ==2 )
```

```
        {     Return(1)          }
        else
        {     F1 = 1
              F2 = 1
              For ( i = 3 ; i <= N ; i ++ )
              {   F = F1 + F2
                  F1 = F2
                  F2 = F
              }
              Return(F)
        }
    }
}
```

Untuk penyelesaian secara rekursif, fungsi Fibonacci membutuhkan 2 base case yaitu untuk N == 1 dan N == 2. Keduanya akan menghasilkan angka Fibonacci tanpa melalui proses apapun. Lihat fungsi rekursif berikut ini.

```
Fungsi FIBONACCI(N)
Fungsi untuk menghitung deret Fibonacci dari bilangan N secara
rekursif. Semua variabel berjenis integer.
{
1.  If  ( N == 1 )
    {     Return (1) }
    else
    {     If  ( N == 2 )
          {     Return (1) }
          else
          {     Return (FIBONACCI(N-1) + FIBONACCI(N-2))    }
    }
    }
```

Dari kedua fungsi di atas, dapat dilihat bahwa solusi dengan menggunakan subalgoritma rekursif lebih singkat dan hemat tempat karena tidak memerlukan struktur pengulangan. Untuk memperjelas proses pemanggilan fungsi secara rekursif dapat dilihat gambar 1.4 untuk N = 6 di halaman berikut. Dalam gambar dapat dilihat bahwa terjadi pemanggilan fungsi yang sama berulang-ulang, contohnya pada Fibonacci(2) yang dipanggil sebanyak 5 kali. Kejadian ini disebut sebagai redundancy, yaitu hal yang sama dikerjakan berulang kali.



Gambar 1.4 Call tree untuk fungsi FIBONACCI(6)

1.5. Infinite Regression

Fungsi Fibonacci(n) di atas menunjukkan bahwa solusi dengan menggunakan konsep rekursif lebih sederhana dan singkat disbanding solusi dengan menggunakan konsep iteratif. Tetapi dari penelusuran fungsi Fibonacci(n) ini diketahui bahwa terjadi banyak proses yang berulang sehingga dari segi waktu proses, sub algoritma rekursif membutuhkan waktu yang lebih lama dibandingkan dengan waktu proses dari sub algoritma iteratif. Hal inilah yang merupakan salah satu kekurangan dari sub algoritma rekursif.

Hal yang lainnya adalah pada sub algoritma rekursif, dapat terjadi *Infinite regression* yaitu jika subalgoritma rekursif memanggil dirinya sendiri terus menerus, tidak berhenti dan tidak pernah mencapai *base case*. Biasanya setelah beberapa waktu program akan berhenti dengan sendirinya (*computer hang*). *Infinite regression* terjadi karena :

1. Tidak ada *base case* yang menghentikan pemanggilan rekursif
2. *Base case* ada, tapi tidak pernah dipanggil

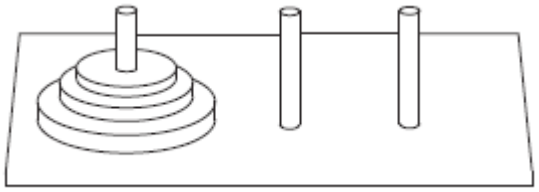
Sebagai contoh akan dicari bilangan Faktorial(0) dengan menggunakan fungsi Faktorial_Rek(N) yang telah dibahas sebelumnya. *Call trace* dari fungsi FAKTORIAL(0) adalah sebagai berikut:

```
Faktorial_Rek(0) =(0*Faktorial_Rek(-1))
                 =(0*(-1)*Faktorial_Rek(-2))
                 =(0*(-1)*(-2)*Faktorial_Rek(-3))
                 =(0*(-1)*(-2)*(-3)*Faktorial_Rek(-4))
                 = ... dan seterusnya
```

1.6. Menara Hanoi

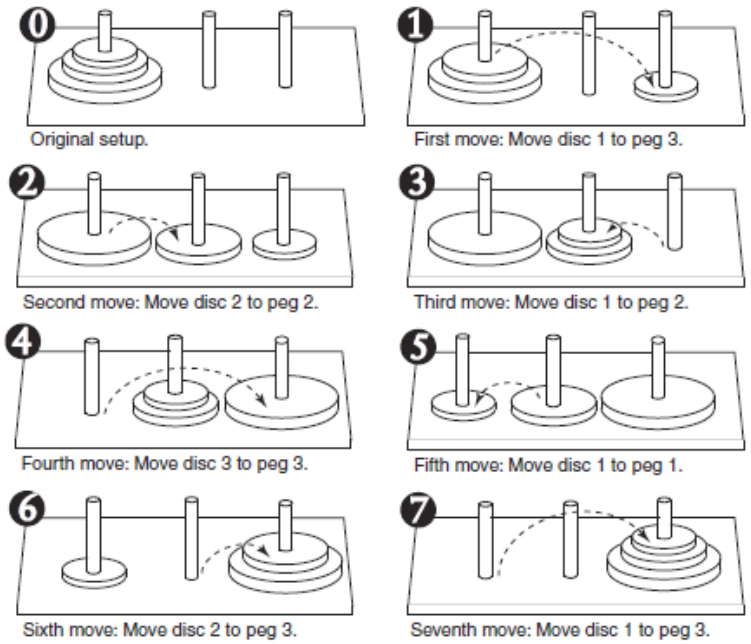
Menara Hanoi adalah permainan matematika yang sering digunakan untuk mengilustrasikan konsep rekursif. Permainan ini memerlukan 3 buah tiang dan beberapa piringan yang berlubang. Piringan tersebut berbeda-beda ukurannya, dan ditumpuk mulai dari yang paling besar ke paling kecil pada satu tiang (lihat gambar 1.5). Tujuan dari permainan ini adalah memindahkan tumpukan piring dari 1 tiang ke tiang yang lainnya dengan ketentuan:

1. Tiap kali hanya 1 piringan yang dapat dipindahkan
2. Piringan yang lebih besar tidak boleh berada di atas piringan yang lebih kecil
3. Semua piringan harus berada pada tiang kecuali piringan yang sedang dipindahkan.



Gambar 1.5 Permainan Menara Hanoi

Contoh solusi untuk 3 piringan Menara Hanoi dapat dilihat pada gambar 1.6 di bawah ini.



Gambar 1.6 Proses memindahkan 3 piringan Menara Hanoi

Secara umum, solusi untuk memindahkan n piringan adalah:

1. Pindahkan (n-1) piringan dari tiang A ke tiang B dengan perantaraan tiang C.
2. Pindahkan piringan terakhir dari tiang A ke tiang C
3. Pindahkan n-1 piringan dari tiang B ke tiang C dengan menggunakan tiang A sebagai perantara

Solusi lengkapnya adalah:

```
Void Pindah (J, A, B, C)
Fungsi untuk memindahkan J piringan dari tiang A ke tiang B dengan tiang C sebagai perantara
{
        If (J > 0)
        {   Pindah( J-1, A, C, B)
            Write ("Pindahkan piringan dari tiang ", A, " ke tiang ", C)
            Pindah( J-1, C, B, A)
        }
}
```

Latihan soal

1. Buat fungsi iteratif dan rekursif untuk menyelesaikan masalah perkalian dengan menggunakan operator + (tambah). Contoh:

   Kali(2,6) = 2 * 6 = 2 + 2 + 2 + 2 + 2 + 2

   Lalu buatlah tabel telusur, call tree dan call trace untuk Kali(2,6).

2. Buatlah fungsi iteratif dan rekursif untuk mencetak semua bilangan genap di antara a dan b. Sebagai contoh:

   Genap(5,15) = 6, 8, 10, 12, 14

   Buatlah tabel telusur, call trace atau call tree untuk Genap(5,15)

3. Buatlah fungsi iteratif dan rekursif untuk menjumlahkan angka mulai dari 1 sampai n. Sebagai contoh:

   Jumlah(5) = 1 + 2 + 3 + 4 + 5 = 15

   Buatlah tabel telusur, call trace atau call tree untuk Jumlah(8)

4. Diketahui A[n] adalah array linier. Buatlah fungsi iteratif dan rekursif untuk:
   a. Mencari elemen yang nilainya maksimum dalam array A
   b. Mencari jumlah dari elemen dalam array A
   c. Mencari nilai rata-rata elemen array A

5. Buatlah fungsi PANGKAT(x,n) untuk menghitung x^n dengan x adalah bilangan real dan n adalah bilangan integer positif. Diketahui:

   PANGKAT(x,0) = 1.0

   PANGKAT(x,n) = x * PANGKAT(x, n-1) ; untuk $n \geq 1$

   Buat fungsi iteratif untuk menghitung fungsi pangkat ini.

   Buat trace table, call trace dan call tree untuk PANGKAT(7,4)

6. Diketahui fungsi koefisien Binomial C(n,k) yang didefinisikan sebagai berikut:

   C(n,0) = 1 untuk n ≥ 0

   C(n,n) = 1 untuk n ≥ 0

   C(n,k) = C(n-1, k) + C(n-1, k-1) untuk n > k > 0

   Buat fungsi iteratif dan r ekursif untuk menghitung koefisien Binomial ini.

   Buat trace table, call trace dan call tree untuk C(5,3)

7. Diketahui fungsi untuk menghitung nilai pembagi terbesar (gcd – greatest common divisor) sebagai gerikut:

   gcd(x,y) = y ; y <= x dan x mod y = 0

   gcd(x,y) = gcd(y,x) ; x < y

   gcd(x,y) = gcd(y, x mod y) ; selain di atas

   Buat fungsi iteratif dan rekursif untuk fungsi di atas.

   Buat trace tabel, call trace dan call tree untuk gcd(254, 16)

8. Diketahui array yang berisi sederetan bilangan. Buatlah fungsi rekursif dan iteratif untuk membalikkan urutan dari deretan bilangan tadi. Contoh:

   Input: 1, 2, 3, 4, 5, 6

   Output: 6, 5, 4, 3, 2, 1

9. Diketahui fungsi Ackermann sebagai berikut:
   - Jika m = 0 maka hasilnya n+1
   - Jika n = 0 maka hasilnya A(m-1,1)
   - Selain itu, hasilnya A(m-1, A(m, n-1))

   Buatlah fungsi rekursif untuk fungsi Ackermann ini.