IKI10400 • Struktur Data & Algoritma: Rekursif

Fakultas Ilmu Komputer • Universitas Indonesia

Slide acknowledgments:
Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung

Outline

- Dasar-dasar Rekursif
  - Apa itu recursion/rekursif?
  - Aturan Rekursif
  - Induksi Matematik
- Rekursif Lanjutan
  - Divide and Conquer
  - Mengulang : *Maximum Contiguous Subsequence Sum*
  - Dynamic Programming
  - Backtracking
  - Latihan Rekursif

Dasar-dasar Rekursif

Apa itu Rekursif?

- Method yang memanggil dirinya sendiri baik secara langsung maupun secara tidak langsung.
  - $f(0) = 0; f(x) = 2\ f(x-1) + x^2$
    - $f(1) = 1; f(2) = 6; f(3) = 21; f(4) = 58$
  - $fib(n) = fib(n - 1) + fib(n - 2)$

```
public static int f (int x)
{
    if (x == 0) return 0;
    return 2 * f (x - 1) + x * x;
}
```

Method/Fungsi Recursion

- Fungsi yang memanggil dirinya, secara langsung atau lewat fungsi lain, disebut fungsi rekursif
- Proses pemanggilan diri itu disebut rekursi (*recursion*).
- Contoh:
  - Memangkatkan bilangan real tak nol dengan suatu pangkat bilangan bulat

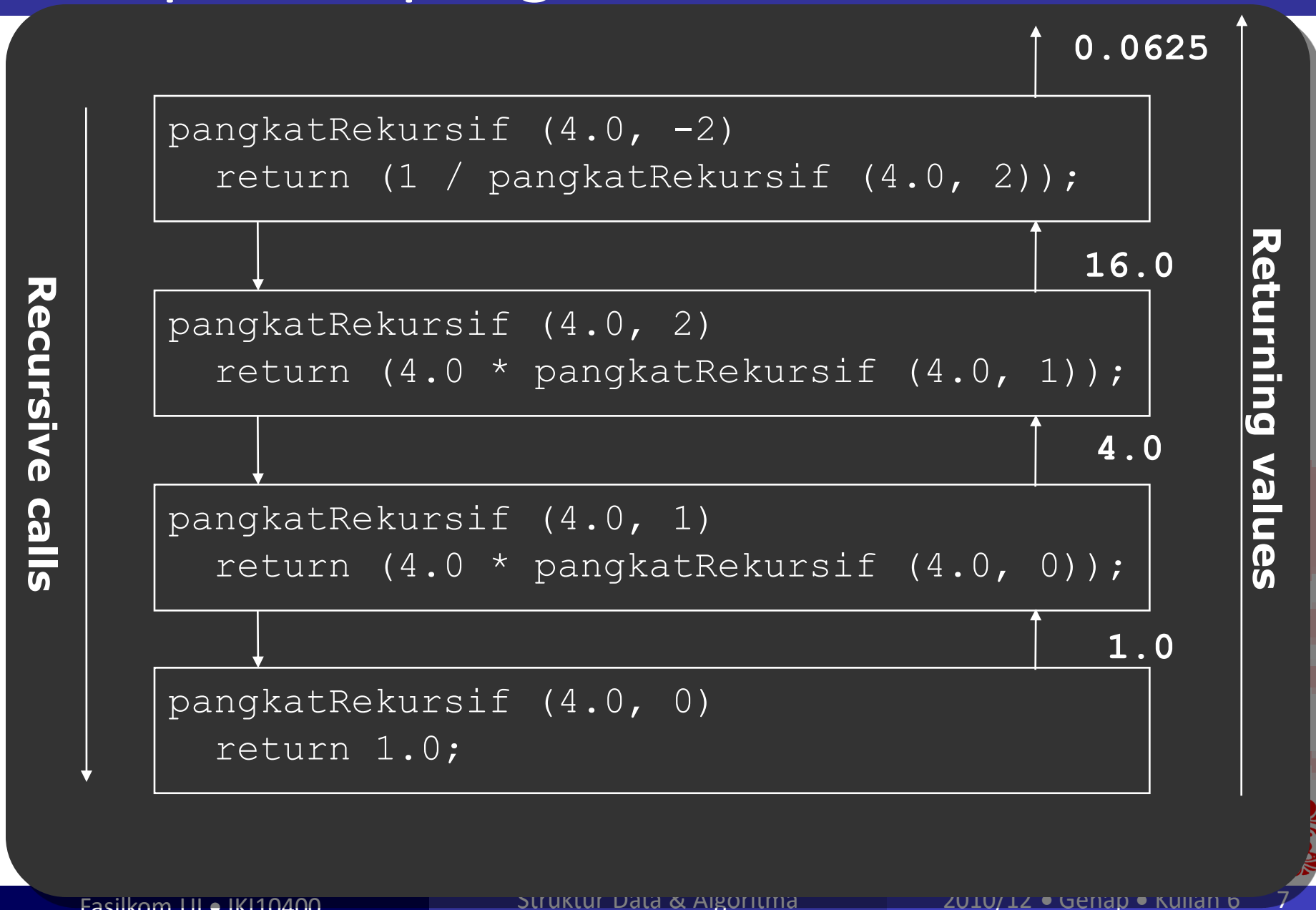
$$x^n = \begin{cases} 1 & jika\ n = 0 \\ x.x^{n-1} & jika\ n > 0 \\ \dfrac{1}{x^{-n}} & jika\ n < 0 \end{cases}$$

```
/**
    Menghitung pangkat sebuah bilangan real
    (versi rekursif).
    @param x bilangan yang dipangkatkan (x != 0)
    @param n pangkatnya
*/
public static double pangkatRekursif (double x, int n)
{
    if (n == 0) {
        return 1.0;
    } else if (n > 0) {
        return (x * pangkatRekursif (x, n - 1));
    } else {
        return (1 / pangkatRekursif (x, -n));
    }
}
```

Berapa nilai pangkat 4^{-2}?

Last in First Out

bisa pake stack, kalo udh bse case bisa langsung pop aja



Algoritme Rekursif

- Ciri masalah yang dapat diselesaikan secara rekursif adalah masalah itu dapat di-reduksi menjadi satu atau lebih masalah-masalah serupa yang lebih kecil
- Secara umum, algoritme rekursif selalu mengandung dua macam kasus:
  - **kasus induksi: satu atau lebih kasus yang pemecahan masalahnya dilakukan dengan menyelesaikan masalah serupa yang lebih sederhana (yaitu menggunakan *recursive calls*)**
  - **kasus dasar atau kasus penyetop (*base case*): satu atau lebih kasus yang sudah sederhana sehingga pemecahan masalahnya tidak perlu lagi menggunakan *recursive-calls*.**
- Supaya tidak terjadi rekursi yang tak berhingga, setiap langkah rekursif haruslah mengarah ke kasus penyetop (*base case*).

Aturan Rekursif

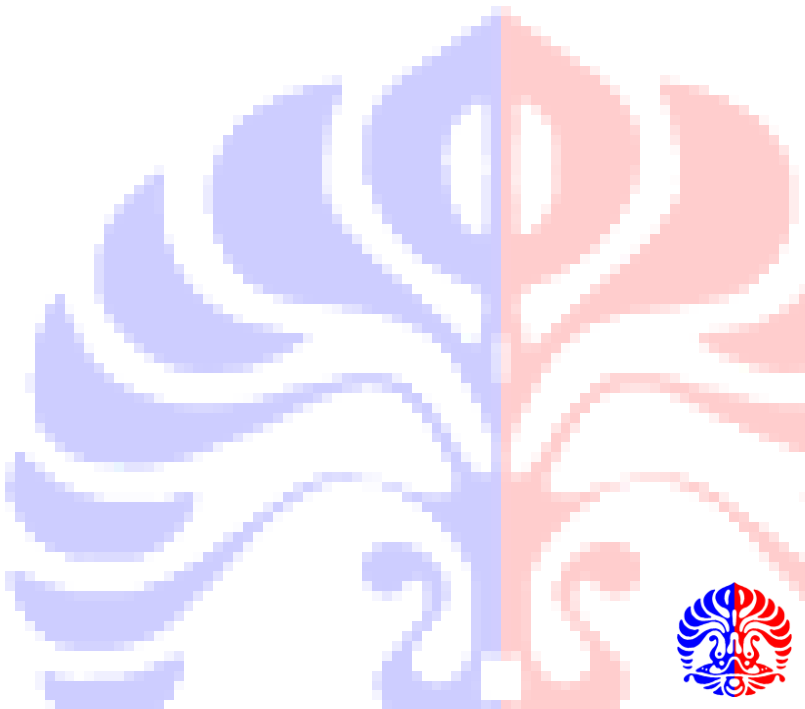
1. Punya kasus dasar
   - Kasus yang sangat sederhana yang dapat memproses input tanpa perlu melakukan rekursif (memanggil method) lagi
2. Rekursif mengarah ke kasus dasar
3. "You gotta believe". Asumsikan rekursif bekerja benar.
   Pada proses pemanggilan rekursif, asumsikan bahwa pemanggilan rekursif (untuk problem yang lebih kecil) adalah benar.
   - Contoh: `pangkatRekursif (x, n)`
     - Asumsikan: `pangkatRekursif (x, n - 1)` menghasilkan nilai yang benar.
     - Nilai tersebut harus diapakan sehingga menghasilkan nilai `pangkatRekursif (x, n)` yang benar?
     - Jawabannya: dikalikan dengan `x`
4. **Aturan penggabungan: Hindari duplikasi pemanggilan rekursif untuk sub-problem yang sama.**

Infinite Recursion

```
public static int bad (int n)
{
    if (n == 0) return 0;
    return bad (n * 3 - 1) + n - 1;
}
```

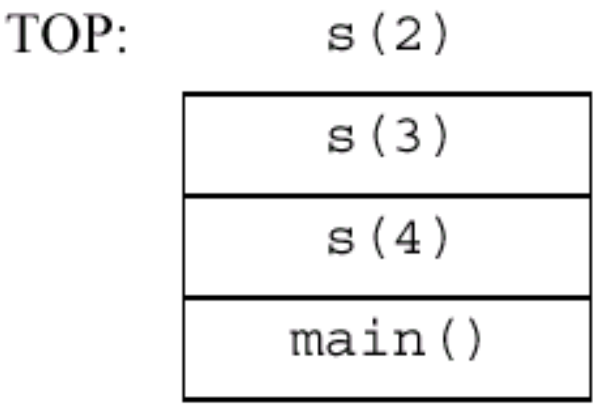
How it works?

- Java VM menggunakan *internal stack of activation records*
- Activation record dapat dilihat sebagai kertas yang berisi informasi tentang method
  - nilai parameter
  - variabel lokal
  - program counter (PC)



How it works?

- Ketika suatu method *G* dipanggil, sebuah activation record untuk *G* dibuat dan di-push ke dalam stack; saat ini *G* adalah method yang sedang aktif
- Ketika method *G* selesai (return), stack di-pop; method dibawah *G* yang dipanggil.



Stack of activation records

Too Much Recursion

```
public static long s (int n){
    if (n == 1) {
        return 1;
    } else {
        return s (n - 1) + n;
    }
}
```

- Di sebuah system, *n* >= 9410 tidak dapat dieksekusi

Pembuktian dgn Induksi

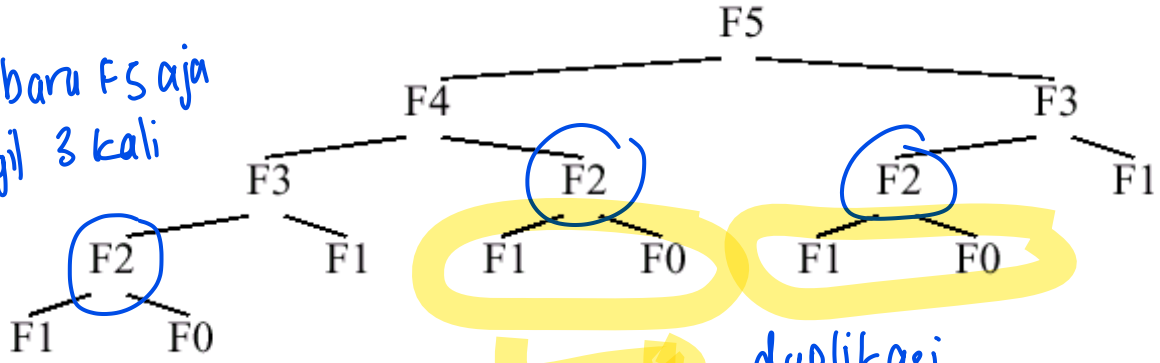
- Contoh kasus: **`pangkatRekursif (x,n)`**
- Buktikan bahwa *base case* benar.
  - **`pangkatRekursif (x,0) = 1`**
- Buktikan bahwa *inductive case* benar
  - Perhitungan/proses untuk input yang lebih kecil dapat diasumsikan memberikan jawaban yang benar atau melakukan proses dengan benar.
    - asumsikan bahwa **`pangkatRekursif (x, n-1)`** memberikan nilai x^{n-1}
  - apakah **`pangkatRekursif (x, n)`** mengembalikan nilai yang benar?
    - **`pangkatRekursif (x, n) = pangkatRekursif (x, n-1) * x`**
    - $x^n = x^{n-1} * x$

Bilangan Fibonacci

- $F_0 = 0, F_1 = 1, F_N = F_{N-1} + F_{N-2}$
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

duplikasi. baru F5 aja F2 dipanggil 3 kali

ada pengulangan jadinya g efektif




```
public static int fib1 (int n)
{
    if (n <= 1) return n;
    return fib1 (n - 1) + fib1 (n - 2);
}
```

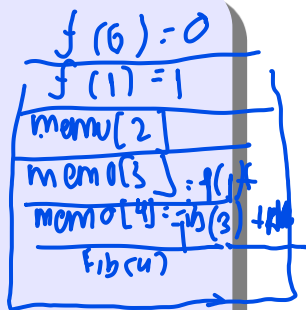
Bilangan Fibonacci

- Untuk N = 40, F_N melakukan lebih dari 300 juta pemanggilan rekursif. F_{40} = 102.334.155
  - Analisa algoritme, *Growth rate*: exponential!!!
- Aturan: Jangan membiarkan ada duplikasi proses yang mengerjakan input yang sama pada pemanggilan rekursif yang berbeda. (Aturan ke-4)

- Ide: simpan nilai fibonacci yang sudah dihitung dalam sebuah array (Teknik **memoisasi**)
  - Catatan: memoisasi sering disebut pendekatan *Dynamic Programming* secara top-down)

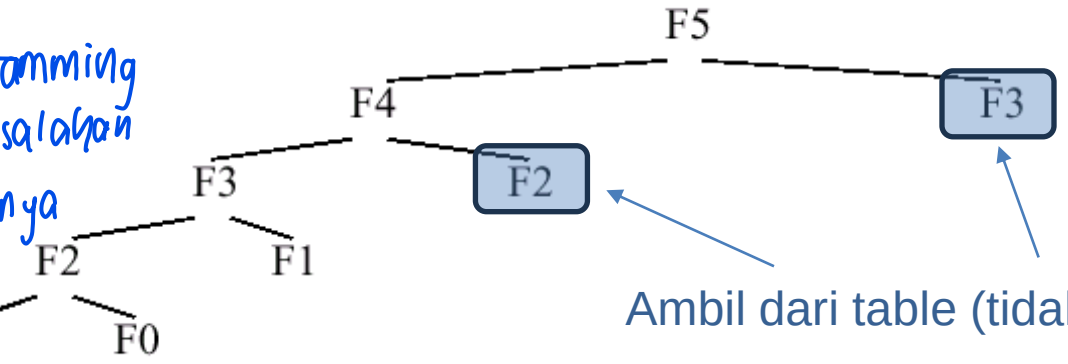
Bilangan Fibonacci dengan Memoisasi

```
//memo: table memoisasi
public static int fib1m(int n){
    if (n <= 1) return n;
    if (memo[n] == -1)
        memo[n] = fib1m(n-1) + fib1m(n-2);
    return memo[n];
}
```



setiap elemen secara default diset -1

dynamic programming
sub permasalahan disimpan hasilnya biar gaperlu hitung ulang



Ambil dari table (tidak dihitung lagi)

Bilangan Fibonacci

- *Dynamic Programming* menyelesaikan sub-permasalahan dengan menyimpan hasil sebelumnya (jadi secara bottom-up).
  - komputasi untuk semua harga n sementara pada memoisasi hanya yang diperlukan saja.

```
public static int fib2 (int n){
    if (n <= 1) return n;
    int result[] = new int[n + 1];
    result[0] = 0;
    result[1] = 1;
    for (int ii = 2; ii <= n; ii++) {
        result[ii] = result[ii - 2]
            + result[ii - 1];
    }
    return result[n];
}
```

Bilangan Fibonacci

- Hanya menyimpan dua hasil sebelumnya saja (tapi tidak berlaku umum untuk semua masalah DP).

```
public static int fib3 (int n){
    if (n <= 1) return n;
    int fib1 = 0;
    int fib2 = 1;
    int result;
    for (int ii = 2; ii <= n; ii++) {
         result = fib2 + fib1;
         fib1 = fib2;
         fib2 = result;
    }
    return result;
}
```

Tail Recursive

-> recursivenya di returnnya

- Optimisasi di level compiler/interpreter dengan tidak menyimpan activation record jika tidak ada lagi komputasi setelah method call.

```
int proc1(int n){
    …
    return proc1(100*n-1); // tail rec.
}
int proc2(int n){
    …
    return proc2(n-1) + proc2(n-2); // bukan tail. rec.
}
int proc3(int n){
    …
    int a = proc3(n-1); // bukan tail. rec.
    return a;
}
```

Bilangan Fibonacci dengan Tail Rec.

- Implementasi Fibonacci bisa secara recursif tapi tidak menyimpan activation record dan return langsung ke pemanggil fib4(n) pertama kali.

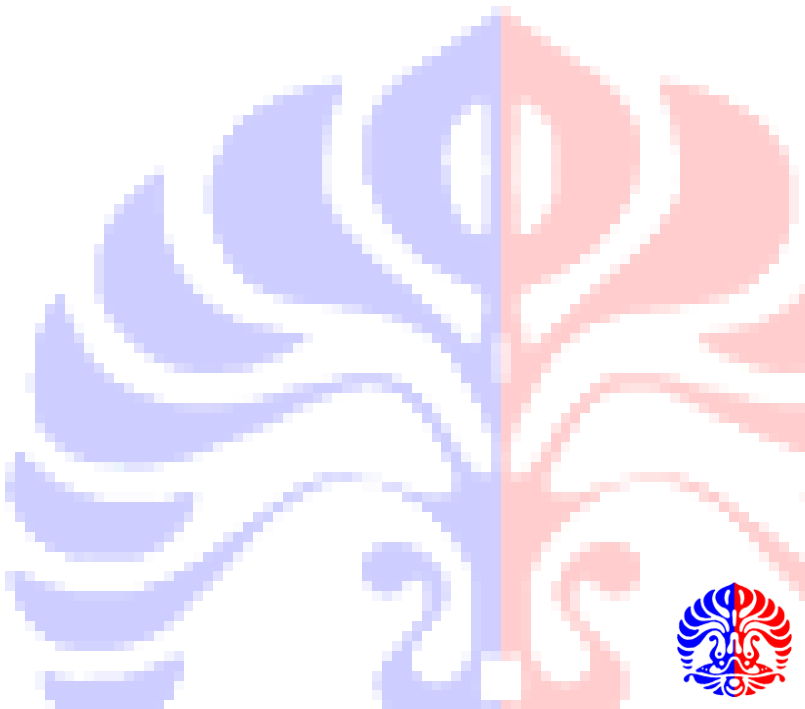
```
public static long fib4 (int n){
     return fiboHelp(0,1,n);
}

static long fiboHelp(long x, long y, int n){

     if (n==0) return x;
     else if (n==1) return y;
     else return fiboHelp(y, x+y, n-1);
}
```

Kesalahan Umum

- Base case terlalu kompleks
- Progress tidak menuju base case
- Duplikasi proses untuk nilai input yang sama dalam recursive call yang terpisah. Tidak efisien.



Pemanfaatan Rekursif dalam Problem Solving

- Memandang masalah-masalah tertentu secara rekursif dapat memudahkan formulasi pemecahan masalah tsb secara lebih elegan dan sistematis/terstruktur. Bagian berikut membahas sbb.
- Pendekatan divide-and-conquer
  - Maximum subsequence sum (lagi!)
  - Binary Search
- Problem Kombinatorik (dengan pendekatan DP)
  - Coin change problem
- Pendekatan backtracking
  - Maze Runner
- Problem-problem eksponensial
  - eight queens problem
  - Tower of Hanoi

Divide and Conquer

- Algoritma:
  - Membagi (*divide*) permasalahan ke dalam bagian yang lebih kecil.
  - Menyelesaikan (*conquer*) masalah per bagian secara recursive
  - Menggabung penyelesaian per bagian menjadi solusi masalah awal

Studi Kasus

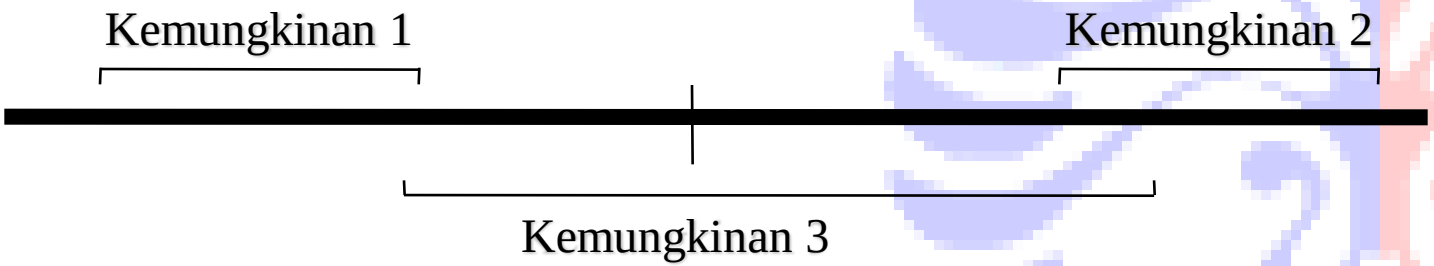
- Masalah *Maximum Contiguous Subsequence Sum*
  - Diberikan (angka integer negatif dimungkinkan) $A_1, A_2, \ldots, A_N$, cari nilai maksimum dari $(A_i + A_{i+1} + \ldots + A_j)$.
- *maximum contiguous subsequence sum* adalah nol jika semua integer adalah negatif.
- Contoh (maximum subsequences digarisbawahi)
  - -2, **11, -4, 13**, -4, 2
  - 1, -3, **4, -2, -1, 6**

Penerapan Pada Studi kasus

- Membagi permasalahan menjadi lebih kecil.
  - Deretan bilangan input di bagi dua menjadi dua bagian yang masing-masing lebih kecil dari input awal.
  - Identifikasi kemungkinan yang dapat terjadi.
- Menyelesaikan (conquer) masalah per bagian secara recursive
  - Lakukan pemanggilan rekursif kepada tiap-tiap bagian.
- Menggabungkan penyelesaian tiap bagian menjadi penyelesaian awal.
  - Bandingkan hasil tiap kemungkinan, termasuk hasil dari gabungan kedua bagian (kemungkinan tiga).

Penerapan Pada Studi kasus

- Urutan dengan nilai jumlah terbesar kemungkinan berada pada:
  - terletak di setengah input awal.
  - terletak di setengah input akhir.
  - berawal disetengah input awal dan berakhir di setengah input akhir.
- Hitung ketiga kemungkinan tersebut. Cari yang lebih besar.
- Kedua kemungkinan pertama (1, 2) merupakan permasalahan yang sama tapi dengan input lebih kecil maka dapat dihitung secara rekursif dengan input baru.



Menghitung kemungkinan ketiga

- dapat dilakukan dengan dua iterasi; lihat program
- kemungkinan ketiga berasal dari penjumlahan dua bagian:
  - Bagian pertama berawal pada setengah bagian input pertama berakhir di tengah.
  - Bagian kedua berasal dari urutan index setengah +1 hingga setengah bagian input akhir.
- Untuk bagian pertama gunakan iterasi dari kanan-ke-kiri (*right-to-left*) mulai dari element terakhir pada setengah input awal.
- Untuk bagian kedua, gunakan iterasi dari kiri-ke-kanan, (*left-to-right*) mulai dari awal setengah input akhir.

Versi Rekursif

```
private int maxSumRec (int[] a, int left, int right)
{
     int maxLeftBorderSum = 0, maxRightBorderSum = 0;
     int leftBorderSum = 0, rightBorderSum = 0;
     int center = (left + right) / 2;

    if(left == right) { // Base case
        return a[left] > 0 ? a[left] : 0;
    }
    int maxLeftSum = maxSumRec (a, left, center);
    int maxRightSum = maxSumRec (a, center+1, right);

    for(int ii = center; ii >= left; ii--) {
        leftBorderSum += a[ii];
        if(leftBorderSum > maxLeftBorderSum)
            maxLeftBorderSum = leftBorderSum;
    }
    ...
```

Versi Rekursif (lanj.)

```
    ...
    for(int jj = center + 1; jj <= right; jj++) {
        rightBorderSum += a[jj];
        if(rightBorderSum > maxRightBorderSum)
            maxRightBorderSum = rightBorderSum;
    }

    return max3 (maxLeftSum, maxRightSum,
          maxLeftBorderSum + maxRightBorderSum);
}

public int maxSubSum (int [] a)
{
    return maxSumRec (a, 0, a.length-1);
}
```

Catatan: method max3 (tidak ditunjukkan implementasinya), method tersebut mengembalikan nilai maksimum dari ketiga nilai.

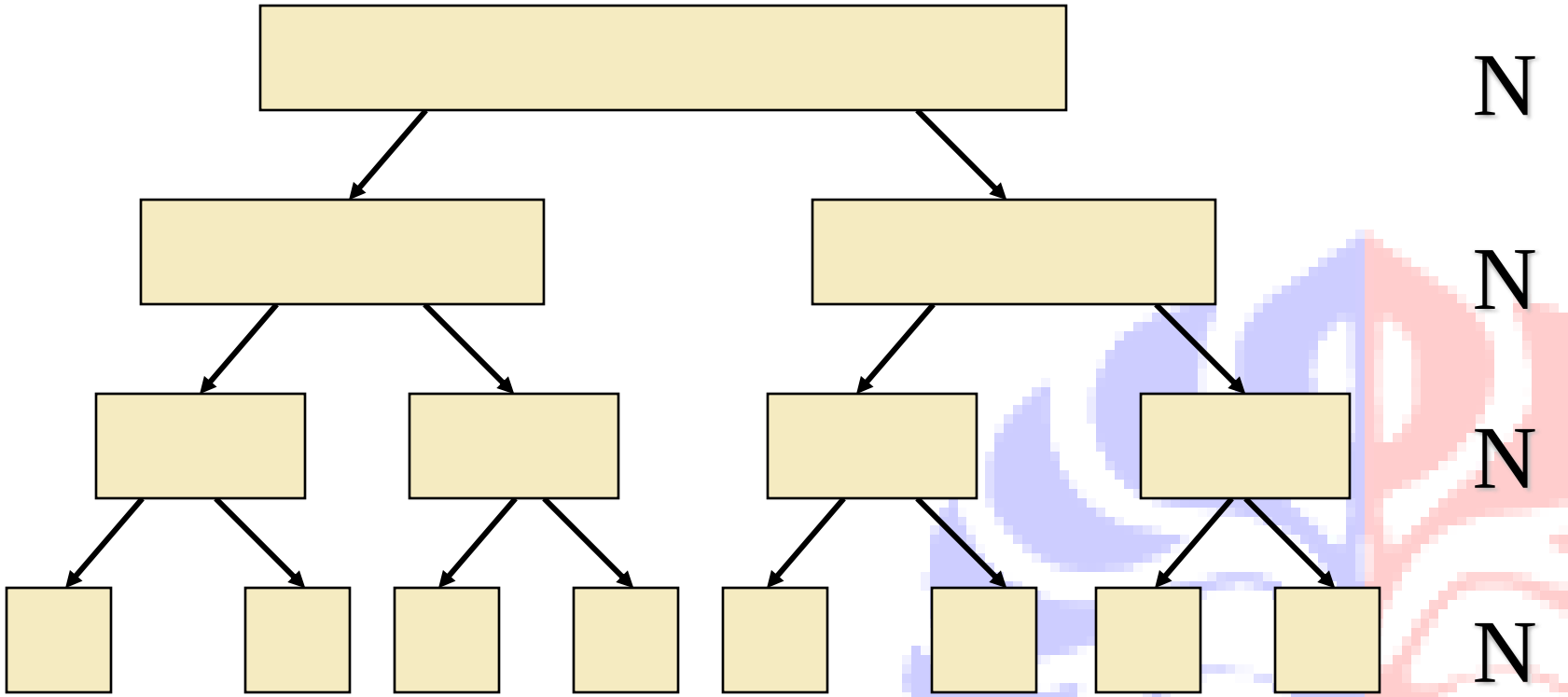
Detil Program

- Pastikan dalam rekursif program anda ada *base case*.
- Gunakan method "driver" yang **public** (method rekursif dibuat **private**)
- Aturan Rekursif :
  - Memiliki *base case*
  - Membuat *progress* menuju ke *base case*
  - Asumsikan bahwa panggilan rekursif bekerja dengan baik.
  - Hindari menghitung sebuah penyelesaian dua kali.

Analisa

- Misalkan $T(N)$ adalah waktu untuk menyelesaikan masalah dengan ukuran input N.
- maka $T(1) = 1$ (1 adalah quantum time unit ketika memproses *base case*; ingat konstanta tidak terlalu penting.).
- $T(N) = 2\ T(N / 2) + N$
  - Dua buah pemanggilan rekursif, masing-masing berukuran $N / 2$. Waktu yang dibutuhkan untuk menyelesaikan masing-masing-nya adalah $T(N / 2)$
  - Kasus ketiga membutuhkan $O(N)$.

Running Time



$O(N \log N)$

Bottom Line

$T(1) = 1 = 1 * 1$

$T(2) = 2 * T(1) + 2 = 4 = 2 * 2$

$T(4) = 2 * T(2) + 4 = 12 = 4 * 3$

$T(8) = 2 * T(4) + 8 = 32 = 8 * 4$

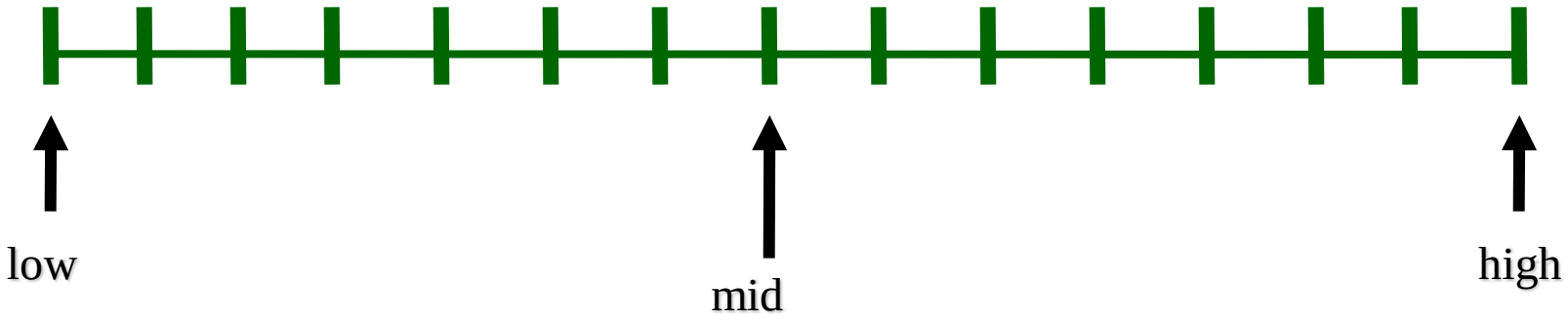
$T(16) = 2 * T(8) + 16 = 80 = 16 * 5$

$T(32) = 2 * T(16) + 32 = 192 = 32 * 6$

$T(64) = 2 * T(32) + 64 = 448 = 64 * 7$

$T(N) = N(1 + \log N) = N + N \log N = O(N \log N)$

Contoh: Binary Search

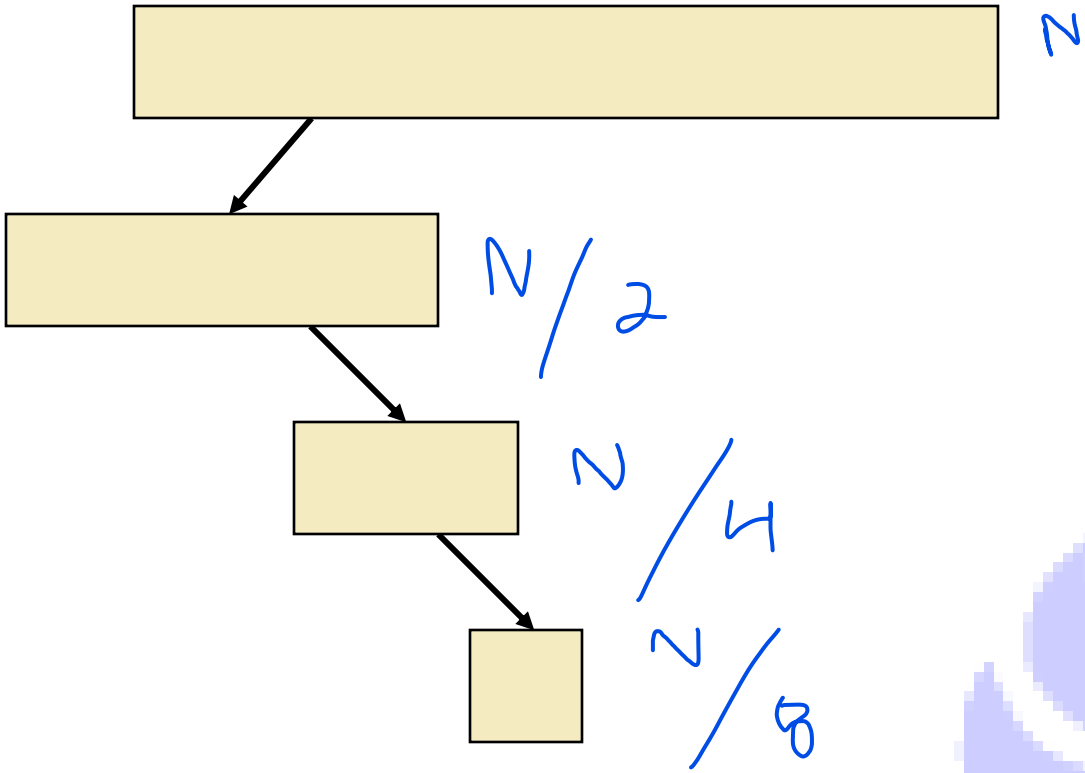



```
int binsearch(data[ ], n, low, high) {
    mid = (low+high) / 2;
    if( data[mid] == n )
                return mid;
    else if ( n < data[mid] )
                return binsearch(data[ ], n, low, mid-1 );
    else return binsearch(data[ ], n, mid+1, high);
}
```

Base case

Recursive case

Running Time of Binary Search



O (log N)

Pada penerapan divide and conquer apakah permasalahannya selalu diperkecil dengan cara dibagi dua?

Problem Change-making (Kombinatorik)

- Dengan coin yang tersedia C_1, C_2, .., C_N (cents) tentukan jumlah minimum coin yang diperlukan untuk “kembalian” sejumlah K cents.

Solusi dg Pendekatan Algo. Greedy

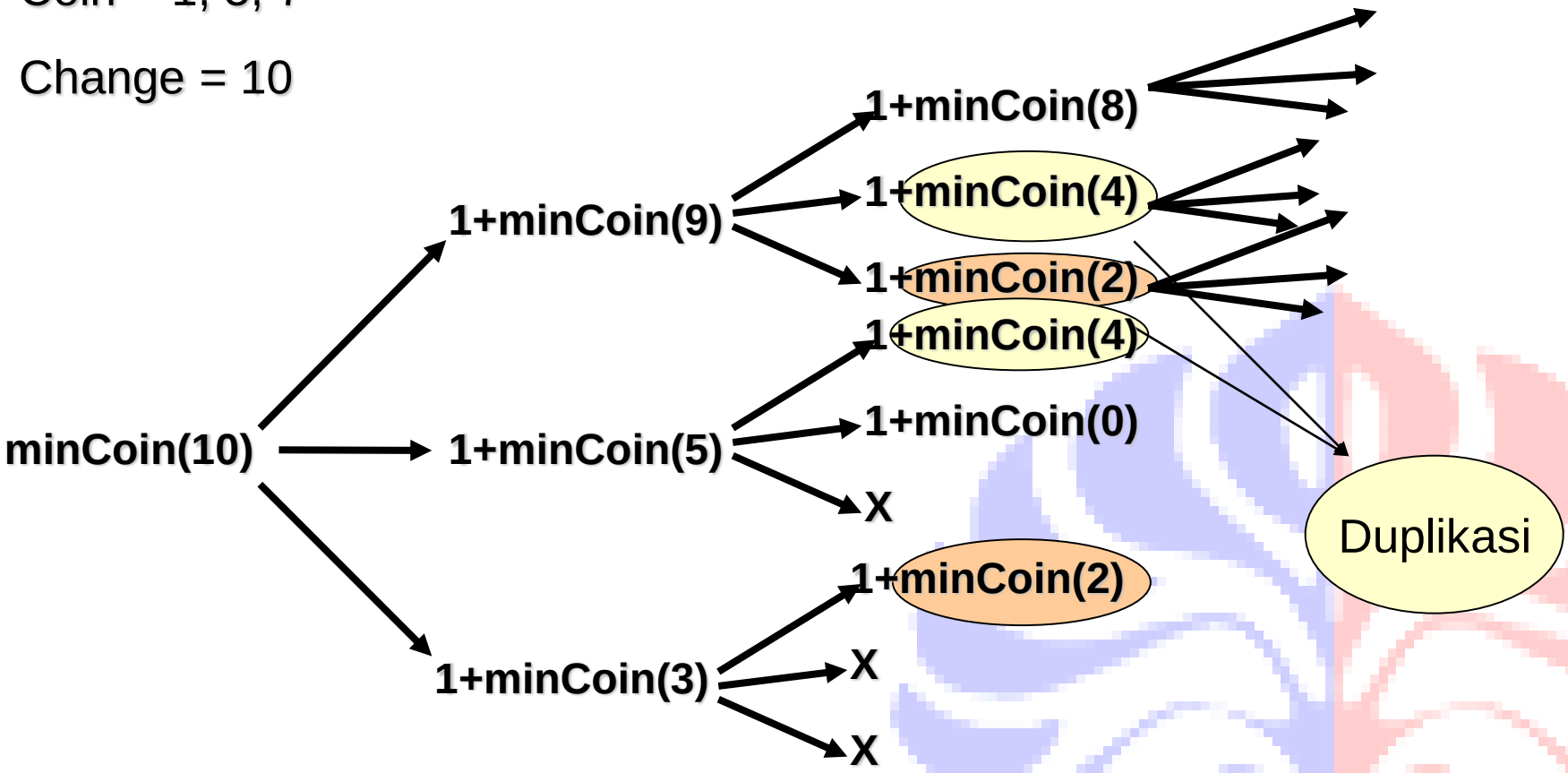
- Ambil koin terbesar yang tersedia secara berulang-ulang
- Dengan coin: 1, 5, 10, dan 25 cent, kembalian 63 cent: 25, 25, 10, 1, 1, 1.
- Masalah: solusi dengan Algoritm Greedy **tidak selalu yang terbaik**, hanya optimum lokal
- Misalkan ada tambahan coin bernilai 21, maka hasil algoritma Greedy tetap seperti di atas, padahal ada solusi yang lebih optimum yaitu 3 coin (3 buah coin 21)

Algoritma Rekursif

Misalkan yang ada hanya 3 coin saja

Coin = 1, 5, 7

Change = 10



Algoritma Rekursif

```
int makeChange (int[] coins, int change) {
    int minCoins = change;

    for (int i=0; i<coins.length; i++)
        if (coins[i] == change)
            return 1;

    for (int i=0; i<coins.length && coins[i]<change; i++) {
        int thisCoins = makeChange(coins, coins[i] ) +
                        makeChange(coins, change - coins[i] );
        if(thisCoins < minCoins)
            minCoins = thisCoins
    }
    return minCoins;
}
```

max # of coins

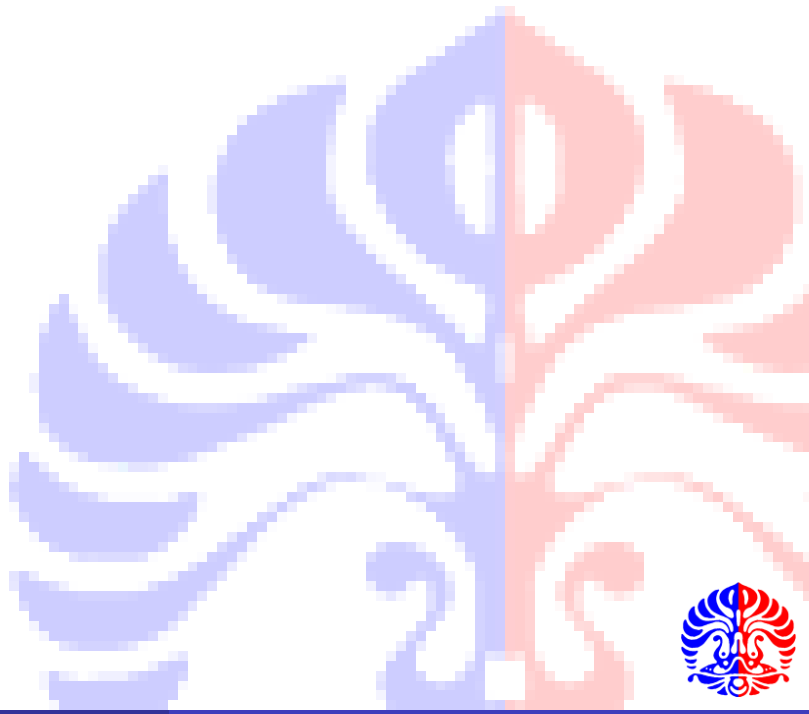
ada coin yg = change

Apakah solusi ini benar dan efficient?

Apakah bisa digeneralisasi untuk berbagai koin?

Solusi dengan Memoisasi

- Hasil perhitungan disimpan dalam cache untuk kemudian dipakai lagi untuk permasalahan yang sama.



Algoritma Rekursif (memoisasi)

```
int makeChange (int[] coins, int change) {
    if (memo[change] == -1) {
        int minCoins = change;

        for (int i=0; i<coins.length; i++)
            if (coins[i] == change)
                return 1;

        for (int i=0; i<coins.length && coins[i]<change; i++){
            int thisCoins = makeChange(coins, coins[i] ) +
                  makeChange(coins, change - coins[i] );
            if(thisCoins < minCoins)
                minCoins = thisCoins;
        memo[change] = minCoins;
    }
    return memo[change];
}
```

Versi Iterasi (DP)

```
int makeChange (int[] coins, int change, int[] coinsUsed, int[] lastCoin)
{
   coinUsed[0]=0; lastCoin[0]=0;

   for(int cents = 1; cents <= change; cents++) {
    int minCoins = cents;
    int newCoin = 1;
    for(int j = 0; j < coins.length; j++) {
        if(coins[ j ] > cents)
            continue;
        if(coinsUsed[ cents - coins[ j ] ] + 1 < minCoins) {
            minCoins = coinsUsed[cents - coins[j]] + 1;
            newCoin = coins[ j ];
        }
    }
    coinsUsed[ cents ] = minCoins;
    lastCoin[ cents ] = newCoin;
   }
}
```

Coin terakhir yg digunakan untuk membuat optimal change

newCoin => coin terakhir awal adalah 1 cent

j = index coin

Jml coin yg dibutuhkan:
1+solusi minimum untuk sisa

Menyimpan Hasil Perhitungan

- Menyimpan hasil perhitungan optimal setiap tahap ke dalam bentuk array/tabel. (Contoh Coin: 1, 5, 10, 21)

Kembalian	Jml Coin
1	1
2	2
.	.
5	1
6	2
.	.
10	1
11	2
.	.
23	3
.	.
63	3

Menyimpan Hasil Perhitungan

- Menyimpan hasil perhitungan optimal setiap tahap ke dalam bentuk array/tabel. Contoh Coin: 1, 5 10, 12, 25

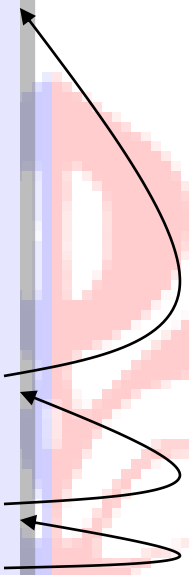
Change(38)

Kembalian	Jml Coin
0	0
1	1
.	.
5	1
…	.
12	1
25	1
…	.
37	2
38	3

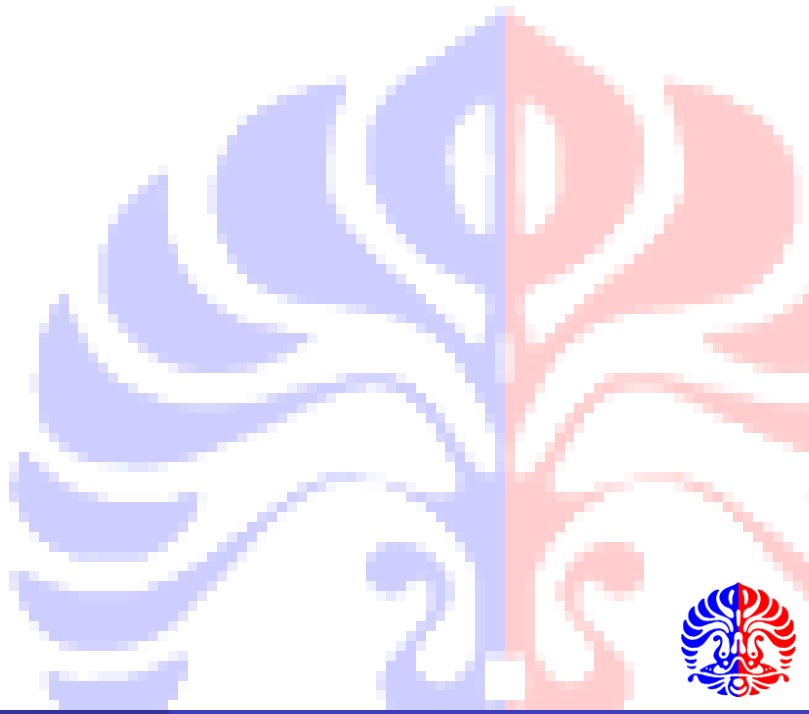
coinUsed[ ]

Kembalian	Last Coin
0	0
1	1
…	.
5	5
…	.
12	12
25	25
…	.
37	12
38	1

lastCoin[ ]

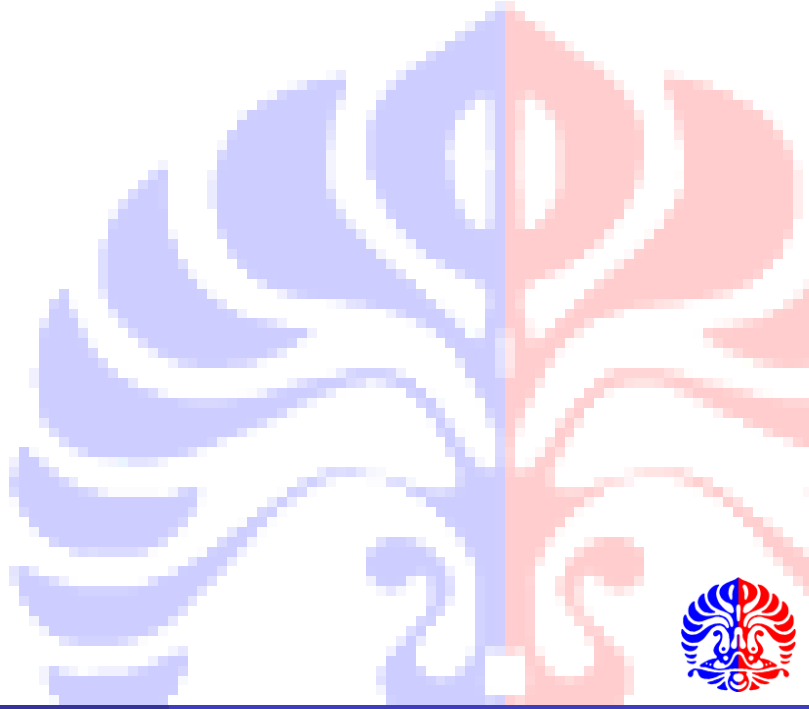


Backtracking



Backtracking Algorithm

- **Coba semua kemungkinan penyelesaian, jika hasilnya tidak memuaskan kembali ke tahap sebelumnya dan coba kemungkinan lain.**
- **Proses berhenti jika hasilnya memuaskan.**
- ***Pruning*: mengeliminir suatu set kemungkinan supaya proses lebih efisien.**



Contoh : Maze Runner

- Spesifikasi input:
  - Diberikan sebuah *maze* ukuran *N x N*.
  - Diberikan dua buah titik. Titik *s* menyatakan *start* (mulai). Titik *f* menyatakan *finish* (akhir).
- Spesifikasi output:
  - Jalani *maze* tersebut dan tandai titik-titik yang pernah dikunjungi. (Termasuk titik-titik yang dilalui tapi menuju jalan buntu.
  - Jalur yang diambil tidak harus jalur terpendek.
- Asumsi:
  - *Maze* yang diberikan valid, selalu ada jalan dari *s* ke *f*.

Contoh input: Maze Runner

Jumlah baris dalam maze file

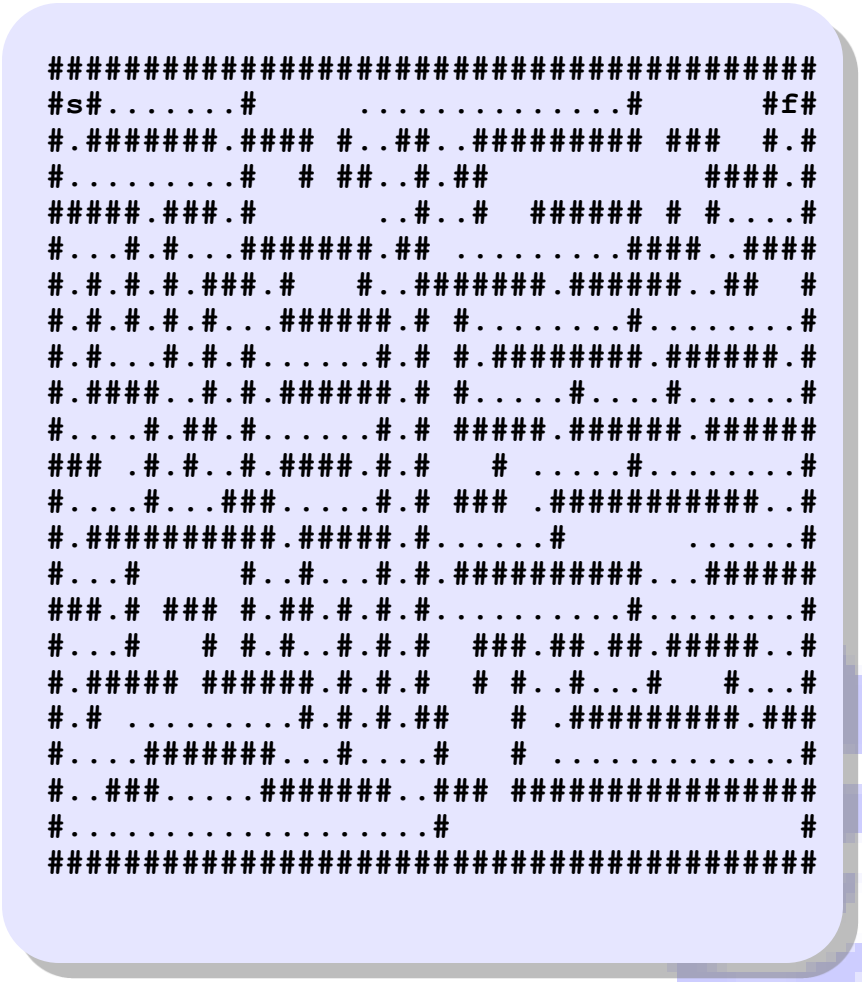
```
23
########################################
#s#       #                    #      #f#
# ####### #### #  ##  ######## ###  # #
#         #  # ##  # ##           #### #
##### ### #        #  #  ###### # #    #
#   # #   ####### ##           ####  ####
# # # # ### #   #  ####### ######  ##  #
# # # # #   ###### # #         #        #
# #   # # #      # # # ######## ###### #
# ####  # # ###### # #     #    #       #
#    # ## #      # # ##### ###### ######
###  # #  # #### # #   #      #         #
#    #   ###     # # ###  ##########  #
# ########## ##### #      #           #
#   #      #  #   # # ##########   ######
### # ### # ## # # #           #        #
#   #   # # #  # # #  ### ## ## #####  #
# ##### ###### # # #  # #  #   #   #   #
# #           # # # ##   #  ######### ###
#    #######   #    #   #              #
#  ###     #######  ### ################
#                    #                  #
########################################
```

start

finish

MAZE

Contoh output: Maze Runner



Algoritma Maze Runner: Rekursif

```
mazeRunner (titik m)
{
   if  (!m.isVisited) {
       if  (m == finish) {
           // cetak seluruh titik yang telah dikunjungi
       } else {
           // kunjungi titik tersebut, dan tandai sudah
           // dikunjungi.
           tandai titik m sebagai visited;
           // tambahkan titik-titik sekitar m yang
           // valid kedalam stack
           mazeRunner (north);
           mazeRunner (west);
           mazeRunner (south);
           mazeRunner (east);
       }
   }
}
```

Masalah Activation Record

- Kembali masalah pada algoritma rekursif ini adalah activation record (karena tidak bisa dibuat tail-recursive!).
- Jika maze berukuran besar sehingga lintasan dari s ke f melalui banyak sekali sel maka sebanyak itu activation record harus disimpan.
- Terdapat Teknik untuk mengkonversi rekursif ke iterasi dengan bantuan stack yang mensimulasikan mekanisme activation record tanpa harus menyimpan banyak data dalam stack.

Algoritma Maze Runner: Iterasi + Stack

```
stack.push(start);
while (!stack.isEmpty()) {
       m = stack.pop ();
       if  (!m.isVisited()) {
              if (m == finish) {
                     // cetak seluruh titik yang telah dikunjungi
              } else {
                     // kunjungi titik tersebut, dan tandai sebagai sudah
dikunjungi.
                     tandai titik m sebagai visited;
                     // tambahkan titik-titik sekitar m yang valid kedalam
stack
                     stack.push (north);
                     stack.push (west);
                     stack.push (south);
                     stack.push (east);
              }
       }
}
```

Latihan

- Algoritme yang mencetak seluruh titik yang pernah dikunjungi termasuk titik yang menuju jalan buntu.
  - Bagaimana agar titik-titik yang menuju jalan buntu tak perlu dicetak?
- Algoritme yang diberikan tidak memberikan jaminan bahwa jalur yang diambil adalah jalur terpendek.
  - Bagaimana agar titik-titik tersebut merupakan jalur terpendek dari *s* ke *f* ?

Contoh Lain: Eight Queen Problem

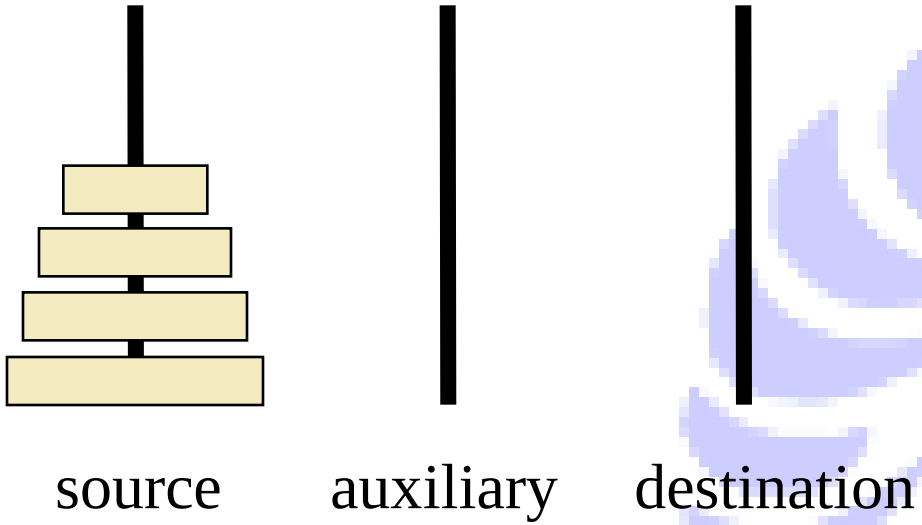
- Permasalahan meletakkan sebanyak 8 Ratu pada papan catur ukuran 8 x 8, dengan syarat tiap Ratu tidak saling mengancam.
- Biasa digeneralisasi menjadi N-Queen Problem
- Ilustrasi menarik mengenai penerapan backtracking dapat dilihat di:

`http://www.animatedrecursion.com/advanced/the_eight_queens_problem.html`

Latihan Rekursif: Tower of Hanoi

Tower of Hanoi (Lucas, 1883)

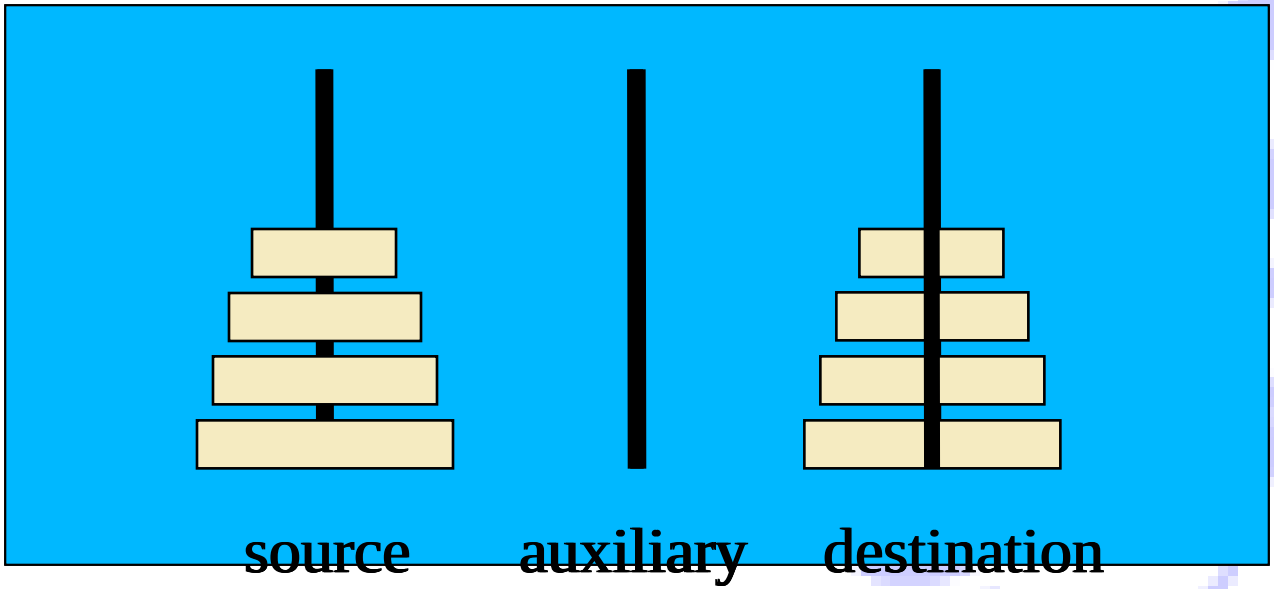
- Pindahkan tumpukan *disc* dari *source* ke *dest* dengan bantuan *auxiliary*
- Tumpukan *disc* yang besar harus selalu berada dibawah yang kecil.



- Bagaimana pembagian permasalahan menjadi lebih kecil?
- Kemungkinan-kemungkinan apa saja yang bisa terjadi?
- Bagaimana cara menggabungkan hasil pemanggilan rekursif?
- Apa base case-nya?
- Apakah pemanggilan rekursif akan selalu menuju base case?

Ide Penyelesaian Secara Rekursif

1. Pindahkan *N-1* disk teratas dari *source*, ke *auxiliary* menggunakan destination sebagai tempat sementara
2. Pindahkan disk terbawah dari *source* ke *destination*
3. Pindahkan seluruh disk dari *auxiliary* ke *destination* menggunakan *source* sebagai tempat sementara.
4. Bila *N = 0* pemanggilan rekursif berhenti.



Latihan

- Bagaimana menyatakan ide-ide tersebut dalam program (pseudo code) ?
  - Saat ini tak perlu memikirkan animasi,
  - cukup mencoba menghitung berapa langkah yang diperlukan untuk memindahkan *N disk*.

Contoh Soal Ujian Mengenai Rekursif

```
public static void main(String[] args)
   {
      int array[] = { 5, 7, 8, 20, 43, 55 };
      final int START_INDEX = 0;
      System.out.println(mystery(array, 55, START_INDEX, array.length -
1));
      System.out.println(mystery(array, 6, START_INDEX, array.length - 1));
      System.out.println(mystery(array, 15, START_INDEX, array.length -
1));
      System.out.println(mystery(array, 35, START_INDEX, array.length -
1));
      System.out.println(mystery(array, 0, START_INDEX, array.length - 1));
   }

   public static int mystery(int[] a, int key, int first, int last)
   {
      int mid = (first + last) / 2;
      if (key == a[mid])
         return a[mid];
      else if (first == last)
         return a[first];
      else if (key < a[mid])
         return mystery(a, key, first, mid);
      else
         return mystery(a, key, mid + 1, last);
   }
```

- Apakah output eksekusi program diatas
- Berapa kompleksitas method mystery?

Ringkasan

- Method rekursif adalah method yang memanggil dirinya sendiri baik secara langsung maupun secara tidak langsung.
- Aturan Rekursif
  - Definisikan *base case*: yang dapat memproses input tanpa perlu recursive lagi
  - Pada bagian rekursif pastikan akan bergerak menuju base case.
  - Asumsikan bahwa pemanggilan rekursif terhadap sub problem berjalan benar.
  - hindari duplikasi proses untuk nilai input yang sama dalam recursive call yang terpisah.
- Bila memungkinkan lakukan *tail recursive*.