# Pemrograman Dinamis

**Harry Budiman**

Teknik Informatika, STEI, Institut Teknologi Bandung
overlordpriere@yahoo.com

## Abstrak

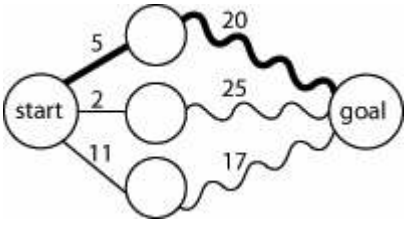
**Dalam ilmu komputer, pemrograman dinamis adalah metode menyelesaikan masalah dengan menampilkan atribut dari menumpuk submasalah (*overlapping subproblems*) dan substruktur optimal (*optimal substructure*) yang akan dideskripsikan di bawah. Metode ini mempunyai tingkat kemangkusan yang lebih baik dibandingkan dengan metode naif.**

**Istilah ini pertama kali digunakan pada tahun 1940 oleh Richard Bellman untuk mendeskripsikan proses penyelesaian masalah dimana keputusan terbaik diambil untuk setiap langkah. Pada tahun 1953, beliau memperbaikinya menjadi istilah modern yang kita kenal. Bidang ini didirikan sebagai topik analisis sistem dan engineering yang diakui oleh IEEE. Kontribusi Bellman diingat dengan nama persamaan Bellman.**

**Kata "pemrograman" dalam "pemrograman dinamis" tidak memiliki hubungan langsung dengan pemrograman komputer. Program adalah serangkaian aksi yang terencana. Misalnya penjadwalan dan eksibisi kadang disebut program. Pemrograman dalam konsep ini, adalah menemukan rencana aksi yang dapat diterima.**

**Makalah ini akan menjelaskan lebih jauh contoh implementasi Pemrograman Dinamis pada persoalan deret fibonacci dan *checkerboard* sederhana.**

## 1. Pendahuluan



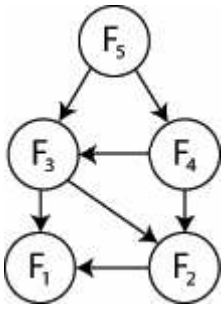

**Gambar 1. Menemukan jalan terpendek pada graf dengan menggunakan *optimal substructure.***

Garis lurus pada gambar 1 menjelaskan hubungan antara 1 simpul dengan tepat 1 simpul (sisi, *edge*); garis bergelombang menjelaskan jalan terpendek antara simpul-simpul yang dihubungkannya (simpul – simpul lain tidak digambarkan); Garis tebal menyatakan jalur terpendek dari start sampai goal secara keseluruhan.

*Optimal substructure* berarti solusi optimal untuk submasalah yang bisa digunakan untuk mencari solusi optimal keseluruhan masalah. Sebagai contoh, jalan terpendek dari start ke goal bisa ditemukan dengan pertama-tama menghitung jalan terpendek dari goal ke semua simpul yang berhubungan, lalu menggunakan hasilnya untuk memilih jalan terpendek secara keseluruhan, seperti ditunjukkan di gambar 1. Secara umum, kita dapat menyelesaikan masalah dengan substruktur optimal memakai tiga langkah proses:

1. Pecah masalah menjadi submasalah yang lebih kecil.
2. Selesaikan masalah-masalah ini menggunakan tiga langkah proses secara rekursif.
3. Gunakan solusi optimal per submasalah untuk menentukan solusi optimal untuk masalah semula.

Tiap submasalah dipecahkan dengan cara memecahkannya mencajadi submasalah yang lebih kecil, dan seterusnya hingga suatu submasalah mejadi cukup sederhana untuk dapat diselesaikan.




**Gambar 2. Graf submasalah untuk deret Fibonacci.**

Graf ini bukan berupa pohon, namun DAG (*directed acyclic graph*). Hal ini mengindikasikan *overlapping subproblems*.

Mengatakan bahwa sebuah masalah memiliki *overlapping subproblem* sama dengam mengatakan bahwa *subproblem* yang sama digunakan untuk menyelesaikan persoalan-

persoalan yang lebih besar. Sebagai contoh, dalam deret Fibonacci, $F_3 = F_1 + F_2$ dan $F_4 = F_2 + F_3$ — menghitung tiap angka berarti menghitung juga $F_2$. Karena $F_3$ dan $F_4$ diperlukan untuk menghitung $F_5$, pendekatan naif untuk menghitung mungkin akan berakhir dengan menghitung $F_2$ 2 kali atau lebih. Ini semakin mempertegas kenyataan bahwa *overlapping subproblem* ada di persoalan ini: pendekatan naif mungkin akan menghabiskan waktu menghitung ulang solusi optimal untuk submasalah yang sebelumnya sudah diselesaikan.

Untuk menghindari masalah ini, kita menyimpan solusi terbaik untuk tiap submasalah yang sudah kita hitung. Lalu ketika kita memerlukannya untuk menyelsaikan masalah yang asma nanti, kita bisa menggunakan ulang solusi yang sudah terhitung tersebut. Pendekatan ini disebut *memoization* (bukan *memorization,* walaupun dari contoh di atas, kedua istilah tersebut sama) Jika kita yakin kita tidak memerlukan solusi tertentu lagi, kita bisa membuangnya untuk menghemat ruang. Pada kasus-kasus tertentu, kita bahkan dapat menghitung solusi untuk tiap submasalah yang akan kita perlukan lebih dulu.

Dengan kata lain, pemrograman dinamis memanfaatkan:

- *Overlapping subproblems*
- *Optimal substructure*
- *Memoization*

## 2. Metode

Pemrograman dinamis biasanya memakai satu dari dua pendekatan beriku:

- **Pendekatan top-down**: Suatu masalah dipecah menjadi submasalah, dan tiap submasalah ini dipecahkan, kemudian solusinya disimpan, jika suatu waktu diperlukan lagi. Ini adalah rekursi dan memoisasi digabungkan menjadi satu.
- **Pendekatan bottom-up**: Semua submasalah yang mungkin diperlukan diselesaikan terlebih dahulu lalu digunakan untuk membangun solusi untuk masalah yang lebih besar. Pendekatan ini sedikit lebih baik dalam penyimpanan stack dan jumlah pemanggilan fungsi, namun kadang tidak intuitif untuk memperkirakan semua submasalah yang diperlukan untuk menyelesaikan masalah.

Beberapa bahasa pemrograman dengan ekstensi khusus [1] bisa secara otomatis mememoisasi hasil dari pemanggilan fungsi dengan argumen tertentu, untuk mempercepat evaluasi *call-by-name* (mekanisme ini disebut juga call-by-need). Beberapa bahasa (seperti Maple) mempunyai memoisasi yang sudag terintegrasi dalam sistem. Hal ini hanya mungkin untuk fungsi yang tidak mempunyai efek samping, yang selalu true dalam bahasa pemanggilan murni tapi kadang bersifat false dalam bahasa imperatif.

Langkah-langkah untuk program dinamis antara lain:

- Inisialisasi- Langkah pertama adalah menentukan deret alignment .

  *Set up a matrix filling in the first row and column with zeros since it's assumed there're no gap penalties*
- Pengisian matriks – mengisi matrik memerlukan nilai untuk tiap deret alignment .

  *To do this we look at the left, top,and above and diagonally left to reach the score of the alignment.*

  *If there's a match, +1, if there's a mismatch, 0, and if there's a gap, -1.*

  *Do this for all alignments until the matrix is filled up*

- Traceback – Langkah ini menentukan alignment sebenarnya yang berujung pada skor maksimu

  *We do this by starting at the lower right corner and taking the upper box, the box to the left, and the box diagonally up.*

  *We pick the box that gives us the best score. Repeat this till we get to the top right corner, this is our maximum alignment.* Dynamic programming tutorial [3]

### 2.1. Deret Fibonacci

Implementasi naif dari fungsi menemukan anggota ke-n dari deret Fibonacci, berdasarakan definisi matematis secara langsung :

```
function fib(n)
    if n = 0 or n = 1
       return 1
    else
       return fib(n - 1) + fib(n - 2)
```

Perhatikan bahwa jika kita memanggil, contoh `fib(5)`, kita memproduksi pohon pemanggilan yang berfungsi menghasilkan nilai yang sama berulang kali :

1. `fib(5)`
2. `fib(4) + fib(3)`

3. `(fib(3) + fib(2)) + (fib(2) + fib(1))`
4. `((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))`
5. `(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))`

Pada contoh di atas, `fib(2)` dihitung dua kali dari awal. Pada contoh yang lebih besari, banyak nilai fib yang sama, atau submasalah, dihitung berulang-ulang, menghasilkan algoritma yang memiliki kompleksitas waktu eksponensial.

Sekarang, semisal kita memiliki objek "peta (*map*)" sederhana yang memetakan tiap nilai dari fib yang sudah dihitung, dan kita memodifikasi fungsi kita hanya berbatas pada menggunakan dan mengupdatenya. Fungsi tersebut hanya memerlukan kompleksitas waktu linear O(n) dibandingkan dengan eksponensial :

```
var m := map(0 → 1, 1 → 1)
function fib(n)
   if map m does not contain key n
      m[n] := fib(n − 1) + fib(n − 2)
      return m[n]
```

Teknik menympan nilai yang sudah dihitung ini disebut memoisasi; ini adalah pendekatan top-down, karena kita harus memecah masalah mejadi submasalah lalu kita menghitung dan menyimpan nilainya.

Dalam pendekatan bottom-up, kita menghitung nilai-nilai fib yang lebih kecil terlebih dahulu, lalu kita buat nilai yang lebih besar dari hasilnya. Metode ini juga memiliki kompleksitas O(n) karena hanya mengandung loop repeat yang dilakukan n-1 kali :

```
function fib(n)
    var    previousFib := 0,
           currentFib := 1
    repeat n − 1 times
      var newFib := previousFib
                  + currentFib
        previousFib := currentFib
        currentFib  := newFib
    return currentFib
```

Pada kedua contoh diatas kita hanya menghitung fib(s) satu kali, lalu menggunakannya untuk menghitung fib(4) dan fib(3), dibandingkan dengan menghitung mereka tiap kali diperlukan.

## 2.2. Checkerboard

Misalkan sebuah (*checkerboard*, bisa merepresentasi berbagai hal) dengan ukuran *n x n* dan fungsi harga *c(i,j)* yang mengembalikan nilai pada papan dengan *i* merepresentasikan baris dan *j* sebagai kolom. Contoh papan berukuran 5 x 5,

| | | | | | |
|---|---|---|---|---|---|
| **5** | 6 | 7 | 4 | 7 | 8 |
| **4** | 7 | 6 | 1 | 1 | 4 |
| **3** | 3 | 5 | 7 | 8 | 2 |
| **2** | 2 | 6 | 7 | 0 | 2 |
| **1** | 7 | 3 | 5 | 6 | 1 |
| | **1** | **2** | **3** | **4** | **5** |

Maka $c(1, 3) = 5$

Misalkan kita memiliki sebuah *checker* (pemeriksa, oleh sebab itu, matriks ini diberi nama *checkerboard*) yang akan mulai di kotak manapun di tingkat terbawah (baris) dan kita ingin tahu mana jalan terpendek (jumlah dari nilai-nilai yang dilewati) untuk mencapai tingkat terakhir (pada kasus ini, 5) dengan mengasumsikan checker hanya dapat bergerak vertical, diagonal kiri atas, diagonal kanan atas, masing-masing satu langkah. Contoh, checker berada di posisi (1,3) bisa bergerak (2,2), (2,3) atau (2,4).

| | | | | | |
|---|---|---|---|---|---|
| **5** | | | | | |
| **4** | | | | | |
| **3** | | | | | |
| **2** | | x | x | x | |
| **1** | | | o | | |
| | **1** | **2** | **3** | **4** | **5** |

Masalah ini menunjukkan substruktur optimal. Yaitu solusi untuk persoalan bergantuk pada solusi bada submasalah. Mari kita definisikan fungsi $q(i, j)$ dengan $q(i, j)$ = harga minimum untuk mencapai kotak $(i, j)$.

Jika kita bisa menemukan nilai untuk semua kotak di tingkat ke-n, kita bisa mengikuti jalan dan mengikuti jalan tersebut secara mundur untuk mendapat jalan terpendek.

Mudah untuk dilihat bahwa $q(i, j)$ sama dengan jumlah minimal yang diperlukan untuk mencapai yang manapun dari ketiga kotak dibawahnya (karena hanya tiga kotak itu yang dapat mencapainya) ditambah dengan $c(i, j)$. Contoh :

| | | | | | |
|---|---|---|---|---|---|
| **5** | | | | | |
| **4** | | | A | | |
| **3** | | B | C | D | |
| **2** | | | | | |
| **1** | | | | | |
| | **1** | **2** | **3** | **4** | **5** |

$$q(A) = \min(q(B),\ q(C),\ q(D))\ +\ c(A)$$

Sekarang mari kita definisikan yang lebih umum:

$$q(i,j) = \begin{cases} \infty & j < 1 \text{ or } j > n \\ c(i,j) & i = 1 \\ \min(q(i-1,j-1), q(i-1,j), q(i-1,j+1)) + c(i,j) & \text{otherwise.} \end{cases}$$

Persamaan ini sangat lempang. Baris pertama ditambahkan hanya untuk membuat atribut rekursif lebih sederhana (ketika kita berurusan dengan sisi, kita hanya memerlukan satu rekursi saja). Baris kedua, menjelaskan apa yang terjadi pada tingkat pertama, sebagai titik mulai rekursi (basis). Baris ketiga, rekursinya itu sendiri, adalah yang terpenting. Pada dasarnya, sama dengan contoh A,B,C,D. Dari definisi kita bisa membuat rekursi lempang untuk *q*(*i*, *j*). Pada pseudocode berikut, *n* adalah ukuran papan, *c(i,j)* adalah fungsi harga, dan *min()* adalah fungsi yang mengembalikan nilai minimal:

```
function minCost(i, j)
    if j < 1 or j > n
        return infinity
    else if i = 1
        return c(i, j)
    else
        return min( minCost(i-1, j-1),
minCost(i-1, j), minCost(i-1, j+1) ) +
c(i, j)
```

Perlu dicatat bahwa fungsi ini hanya menghitung harganya saja, dan bukan jalannya. Kita akan segera mendefinisikannya. Ini, seperti contoh pada deret Fibonacci, luar biasa lambat karena menhabiskan waktu untuk menghitung jalan yang sama berulang-ulang. Namun, kita bisa menghitungnya dengan lebih cepat menggunakan pendekatan bottom-up jika kita menggunakan array dua dimensi `q[i, j]` daripada fungsi. Mengapa kita perlu melakukannya? Hal ini disebabkan karena ketika kita menggunakan fungsi, kita menghitung ulang jalan yang sama lebih dari satu kali, jika menggunakan matriks, kita bisa memilih apa yang ingin kita hitung.

Selain harga dari jalan yang kita tempuh, kita juga perlu mengetahui jalannya itu sendiri. Permasalah jalan bisa diselesaikan dengan bantuan array lain `p[i, j]`, *predecessor array*. Array ini pada dasarnya memberitahu darimana jalan tertentu berasal. Misal kode di bawah:

```
procedure computeShortestPathArrays()
    for x from 1 to n
        q[1, x] := c(1, x)
    for y from 1 to n
        q[y, 0]     := infinity
        q[y, n + 1] := infinity
    for y from 2 to n
        for x from 1 to n
            m := min(q[y-1, x-1],
q[y-1, x], q[y-1, x+1])
            q[y, x] := m + c(y, x)
            if m = q[y-1, x-1]
                p[y, x] := -1
            else if m = q[y-1, x]
                p[y, x] :=  0
            else
                p[y, x] :=  1
```

Sisanya adalah menghitung nilai minimum dan mencetaknya saja.

```
procedure computeShortestPath()
    computeShortestPathArrays()
    minIndex := 1
    min := q[n, 1]
    for i from 2 to n
        if q[n, i] < min
            minIndex := i
            min := q[n, i]
    printPath(n, minIndex)
procedure printPath(y, x)
    print(x)
    print("<-")
    if y = 2
        print(x + p[y, x])
    else
        printPath(y-1, x + p[y, x])
```

## 3. Kesimpulan

Pemrograman dinamis biasanya menggunakan salah satu dari dua pendekatan, yaitu top-down dan bottom-up dimana kedua pendekatan tersebut mengandung tiga faktor penting yaitu *Overlapping subproblems, Optimal substructure, Memoization*

## REFERENSI


[1] King, Ian, 2002 (1987), "A Simple Introduction to Dynamic Programming in Macroeconomic Models." An introduction to dynamic programming as an important tool in economic theory. http://www.business.auckland.ac.nz/Departments/econ/workingpapers/full/Text230.pdf, tanggal akses 21 Mei 2007 21:02

[2] Wagner, David B., 1995, "Dynamic Programming." An introductory article on dynamic programming in Mathematica. http://citeseer.ist.psu.edu/268391.html,

tanggal akses 21 Mei 2007, 21:12

[3] Dynamic Programming Tutorial, *http://www.sbc.su.se/%7Epjk/molbioinfo2001/dynprog/dynamic.html*, tanggal akses 21 Mei 2007, 20 : 55

[4] Eitan M. Gurari, Ohio State University: CIS 680: class notes on dynamic programming, http://www.cse.ohio-state.edu/%7Egurari/course/cis680/cis680Ch21.html, tanggal akses 21 Mei 2007, 21:32

[5] Bertsekas, D. P., 2000. Dynamic Programming and optimal Control, Vols. 1 & 2, 2nd ed. Athena Scientific. ISBN 1-886529-09-4.

[6] Munir Rinaldi, M.T, Ir, "Diktat Kuliah IF2251 Strategi Algoritmik", Institut Teknologi Bandung, 2007