Dynamic Programming

Tim Olimpiade Komputer Indonesia

Pendahuluan

Melalui dokumen ini, kalian akan:

- Memahami konsep **dynamic programming** (DP).
- Menyelesaikan contoh persoalan DP sederhana.

Motivasi

- Diberikan M jenis koin, masing-masing jenis bernilai $a_1, a_2, ..., a_M$ rupiah.
- Asumsikan banyaknya koin untuk setiap nominal yang ada tak terbatas.
- Tentukan banyaknya koin paling sedikit untuk membayar tepat sebesar N rupiah!
- (Persoalan ini biasa disebut dengan *coin change*.)

Solusi Greedy

- Mari kita coba menyelesaikan masalah ini secara *greedy*.
- Salah satu algoritma *greedy* yang mungkin adalah dengan selalu menggunakan koin terbesar yang $\leq$ sisa uang yang harus dibayar.

Solusi Greedy (lanj.)

- Misalkan kita memiliki nominal koin 1 rupiah, 6 rupiah, dan 10 rupiah dan ingin membayar 12 rupiah.
- Dengan algoritma sebelumnya, kita akan menggunakan koin 10 rupiah terlebih dahulu.
- Karena tersisa 2 rupiah, berikutnya kita akan menggunakan 2 koin 1 rupiah, sehingga totalnya kita menggunakan 3 koin.
- Namun, ada solusi lebih baik: 2 koin 6 rupiah.
- Algoritma *greedy* ini tidak memberikan solusi terbaik.

Observasi

- Anggaplah kita membayar N rupiah dengan koin-koin satu-persatu.
- Pastilah terdapat koin pertama yang kita bayarkan.
- Jika nilai koin itu adalah a_k, maka sisa uang yang perlu kita bayar adalah $N - a_k$.
- Dalam kasus ini, terdapat M pilihan koin untuk a_k.

Observasi (lanj.)

- Perhatikan bahwa penukaran $N - a_k$ merupakan suatu subpersoalan yang serupa dengan persoalan awalnya.
- Artinya, cara yang sama untuk menyelesaikan subpersoalan dapat digunakan.
- Kita akan menggunakan strategi penyelesaian secara rekursif.

Solusi Rekursif

- Definisikan sebuah fungsi $f(x)$ sebagai banyaknya koin minimum yang dibutuhkan untuk membayar tepat x rupiah.
- Kita dapat mencoba-coba satu koin yang ingin kita gunakan.
- Jika suatu koin a_k digunakan, maka kita membutuhkan $f(x - a_k)$ koin ditambah satu koin a_k.
- Atau dapat ditulis $f(x) = f(x - a_k) + 1$
- Pencarian nilai $f(x - a_k)$ dilakukan secara rekursif, kita kembali mencoba-coba koin yang ingin digunakan.

Solusi Rekursif (lanj.)

- Dari semua kemungkinan a_k, mana pilihan yang terbaik?
- Pilihan yang terbaik akan memberikan nilai $f(x - a_k) + 1$ sekecil mungkin.
- Jadi kita cukup mencoba semua kemungkinan a_k, dan ambil yang hasil $f(x - a_k) + 1$ terkecil.

Solusi Rekursif (lanj.)

- Jika $f(x)$ dihitung secara rekursif, apa yang menjadi **base case**?
- Kasus terkecilnya adalah $f(0)$, yang artinya kita hendak membayar 0 rupiah.
- Membayar 0 rupiah tidak membutuhkan satu pun koin, sehinga $f(0) = 0$.

Solusi Rekursif (lanj.)

Secara matematis, hubungan rekursif ini dituliskan:

$$f(x) = \begin{cases} 0, & x = 0 \\ \min_{1 \le k \le M, a_k \le x} f(x - a_k) + 1, & x > 0 \end{cases}$$

Implementasi Solusi Rekursif

Kita implementasikan $f(x)$ sebagai fungsi $\textsc{solve}(x)$:

$\textsc{solve}(x)$

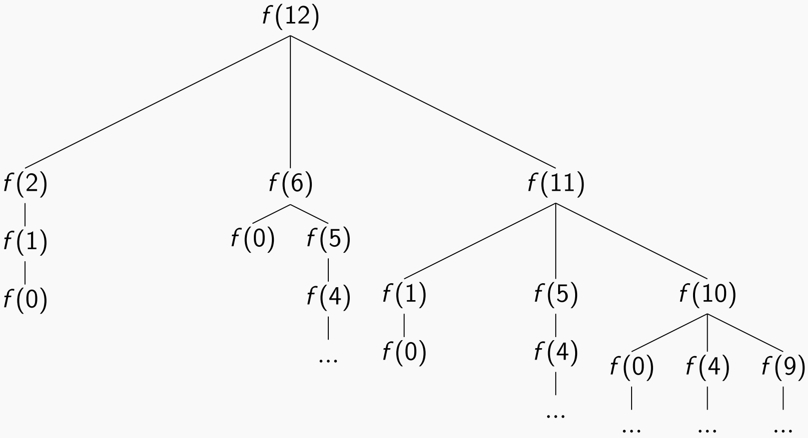
```
if (x == 0)
    return 0

best = ∞
for k = 1 to M
    if (a_k ≤ x)
        best = min(best, SOLVE(x − a_k) + 1)
return best
```

Jawaban akhirnya ada pada $\textsc{solve}(N)$.

Solusi Rekursif (lanj.)

Mari kita lihat pohon rekursi yang dihasilkan oleh fungsi f.
Berikut untuk $f(12)$ dengan nominal koin 1, 6, dan 10 rupiah.



Solusi Rekursif (lanj.)

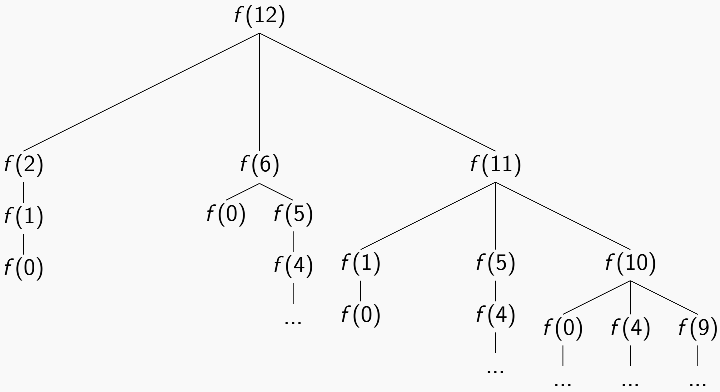
- Jika diperhatikan pada pohon rekursi, terdapat $O(M)$ cabang untuk setiap pemanggilan f.
- Untuk menghitung nilai $f(N)$, kita akan memiliki pohon rekursi yang kira-kira sedalam $O(N)$.
- Berarti kira-kira dilakukan $O(M^N)$ pemanggilan fungsi.
- Karena itu, solusi ini membutuhkan $O(M^N)$ operasi, yang mana banyaknya operasi ini eksponensial.

Solusi Rekursif (lanj.)

- Biasanya solusi eksponensial berjalan sangat lambat.
- Cobalah Anda hitung nilai $O(M^N)$ dengan $M = 3$ dan $N = 100$, untuk menyadari betapa lambatnya solusi ini!
- Kita tidak ingin memiliki solusi eksponensial pada pemrograman kompetitif, kecuali pada soal-soal tertentu yang tidak memiliki solusi polinomial.
- Karena itu, kita harus melakukan sebuah optimisasi.

Optimisasi

Jika diperhatikan, ternyata banyak $f(x)$ yang dihitung berkali-kali. Sebagai contoh, $f(5)$ dan $f(4)$.



Optimisasi (lanj.)

- Perhatikan bahwa hanya ada $N + 1$ kemungkinan x untuk $f(x)$, yaitu 0 sampai N.
- Kita dapat melakukan **memoisasi**, yaitu mencatat hasil perhitungan $f(x)$ setelah menghitungnya.
- Jika suatu ketika kita kembali memerlukan nilai $f(x)$, kita tidak perlu menghitungnya kembali.

Solusi Rekursif dengan Memoisasi

```
SOLVE(x)
if (x == 0)
    return 0
if computed[x]
    return memo[x] // Langsung kembalikan

best = ∞
for k = 1 to M
    if (a_k ≤ x)
        best = min(best, SOLVE(x − a_k) + 1)
computed[x] = true // Tandai bahwa sudah pernah dihitung
memo[x] = best
return best
```

Solusi Rekursif dengan Memoisasi (lanj.)

- Untuk menghitung suatu nilai $f(x)$, kita membutuhkan $O(M)$ iterasi.
- Sehingga untuk menghitung nilai $f(x)$ untuk seluruh x, kita membutuhkan $O(NM)$ operasi.
- Banyaknya operasi ini polinomial terhadap N dan M, dan jauh lebih cepat daripada solusi rekursif sebelumnya.

Dynamic Programming

- Merupakan metode penyelesaian persoalan yang melibatkan pengambilan keputusan dengan memanfaatkan informasi dari penyelesaian subpersoalan yang sama namun lebih kecil.
- Solusi subpersoalan tersebut hanya dihitung satu kali dan disimpan di memori.
- Jika sebuah persoalan adalah masalah optimisasi, maka biasanya kita mencoba semua kemungkinan solusi sub-problem yang dihasilkan, dan mengambil yang hasilnya paling optimal.

Dynamic Programming

Terdapat dua cara mengimplementasikan DP

- **top-down**: diimplementasikan secara rekursif sambil mencatat nilai yang sudah ditemukan (memoisasi).
- **bottom-up**: diimplementasikan secara iteratif dengan menghitung mulai dari kasus yang kecil ke besar.

Top-Down

- Cara yang sebelumnya kita gunakan adalah *top-down*.
- Kata memoisasi berasal dari "memo", yang artinya catatan.
- Pada *top-down*, penyelesaian masalah dimulai dari kasus yang besar.
- Untuk menyelesaikan kasus yang besar, dibutuhkan solusi dari kasus yang lebih kecil.
- Karena solusi kasus yang lebih kecil belum ada, maka kita akan mencarinya terlebih dahulu, lalu mencatat hasilnya.
- Hal ini dilakukan secara rekursif.

Bottom Up

- Pada *bottom-up*, penyelesaian masalah dimulai dari kasus yang kecil.
- Ketika merumuskan formula rekursif, kita mengetahui jawaban kasus yang paling kecil, yaitu *base case*.
- Informasi ini digunakan untuk menyelesaikan kasus yang lebih besar.
- Biasanya dianalogikan dengan pengisian "tabel DP".
- Hal ini dilakukan secara iteratif.

Solusi dengan Bottom-Up

Secara *bottom-up*, kita hitung semua nilai $f(x)$ untuk semua nilai x dari 0 sampai N secara menaik.
Nilai dari $f(x)$ disimpan dalam *array* $f[x]$.

SOLVE()

$f[0] = 0$
for $x = 1$ **to** N
    $best = \infty$
    for $k = 1$ **to** M
        if $(a_k \leq x)$
            $best = \min(best, f[x - a_k] + 1)$
    $f[x] = best$
return $f[N]$

Kompleksitas?

- Dengan mudah Anda dapat memperhatikan bahwa kompleksitas solusi dengan *bottom-up* adalah $O(NM)$.
- Kompleksitas ini sama seperti dengan cara *top-down*.
- Kenyataannya, sebenarnya keduanya merupakan algoritma yang sama, hanya berbeda di arah pencarian jawaban.

Mengisi "Tabel"

Cara *bottom-up* yang dijelaskan sebelumnya terkesan seperti "mengisi tabel".

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$													

Mengisi "Tabel" (lanj.)

Awalnya, diisi $f(0) = 0$.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0												

Mengisi "Tabel" (lanj.)

Berikutnya, diisi $f(1)$.
Satu-satunya pilihan adalah menukarkan dengan koin 1, karena kita tidak bisa menggunakan koin 6 atau 10. Jadi:
$f(1) = f(1-1) + 1 = f(0) + 1$

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1											

Masuk akal, untuk membayar 1 memang kita membutuhkan 1 koin.

Mengisi "Tabel" (lanj.)

Hal serupa terjadi ketika kita mengisi $f(2), f(3), f(4), dan f(5)$. Satu-satunya pilihan adalah menukarkan dengan koin 1, karena kita tidak bisa menggunakan koin 6 atau 10.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5							

Mengisi "Tabel" (lanj.)

Berikutnya adalah mengisi $f(6)$.
Terdapat pilihan untuk menggunakan koin 1 atau 6 terlebih dahulu, sehingga:

$$\begin{aligned} f(6) &= \min(f(6-1)+1, f(6-6)+1) \\ &= \min(f(5)+1, f(0)+1) \\ &= \min(5+1, 0+1) \\ &= \min(6, 1) \\ &= 1 \end{aligned}$$

Mengisi "Tabel" (lanj.)

Memang benar, untuk membayar 6 kita hanya membutuhkan 1 koin.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5	1						

Mengisi "Tabel" (lanj.)

Lakukan hal serupa untuk $x = 7$ sampai $x = 9$.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5	1	2	3	4			

Mengisi "Tabel" (lanj.)

Berikutnya adalah mengisi $f(10)$.
Terdapat pilihan untuk menggunakan koin 1, 6, atau 10 terlebih dahulu, sehingga:

$$\begin{aligned}
f(10) &= \min(f(10-1)+1,\ f(10-6)+1,\ f(10-10)+1) \\
&= \min(f(9)+1,\ f(4)+1,\ f(0)+1) \\
&= \min(4+1,\ 4+1,\ 0+1) \\
&= \min(5, 5, 1) \\
&= 1
\end{aligned}$$

Mengisi "Tabel" (lanj.)

Kembali, memang benar bahwa untuk membayar 10 kita hanya membutuhkan 1 koin, yaitu langsung menggunakan koin 10 (tanpa 1 dan 6).

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5	1	2	3	4	1		

Mengisi "Tabel" (lanj.)

Berikutnya adalah mengisi $f(11)$.

$$\begin{aligned} f(11) &= \min(f(11-1)+1,\ f(11-6)+1,\ f(11-10)+1) \\ &= \min(f(10)+1,\ f(5)+1,\ f(1)+1) \\ &= \min(1+1,\ 5+1,\ 1+1) \\ &= \min(2,6,2) \\ &= 2 \end{aligned}$$

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5	1	2	3	4	1	2	

Mengisi "Tabel" (lanj.)

Terakhir, isi $f(12)$.

$$\begin{aligned} f(12) &= \min(f(12-1)+1,\ f(12-6)+1,\ f(12-10)+1) \\ &= \min(f(11)+1,\ f(6)+1,\ f(2)+1) \\ &= \min(2+1,\ 1+1,\ 2+1) \\ &= \min(3, 2, 3) \\ &= 2 \end{aligned}$$

Dari sini, terlihat bahwa menggunakan koin 10 terlebih dahulu (pilihan paling kanan) mengakibatkan banyaknya koin yang dibutuhkan adalah 3.

Sementara menggunakan koin 6 terlebih dahulu (pilihan tengah) mengakibatkan banyaknya koin yang dibutuhkan adalah 2.

Mengisi "Tabel" (lanj.)

Jadi kita selesai mengisi "tabel" DP.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5	1	2	3	4	1	2	2

- Jika Anda menggunakan *top-down*, pada akhirnya tabel *memo* juga akan berisi nilai-nilai ini.
- Coba implementasikan secara *top-down* dan *bottom-up* dan lihat hasilnya!

Top-Down dan Bottom-Up

Top-down

- Sebuah transformasi natural dari formula rekursif, biasanya mudah diimplementasikan.
- Urutan pengisian tabel tidak penting.
- Hanya menghitung nilai dari fungsi jika hanya diperlukan.
- Ketika seluruh tabel memo pada akhirnya terisi, bisa saja lebih lambat karena adanya *overhead* pemanggilan fungsi.

Top-Down dan Bottom-Up (lanj.)

Bottom-up

- Tidak mengalami perlambatan dari *overhead* pemanggilan fungsi.
- Memungkinkan penggunaan teknik DP lanjutan seperti *flying table*, kombinasi dengan struktur data *tree*, dsb.
- Harus memikirkan urutan pengisian nilai tabel.
- Semua tabel harus diisi nilainya walaupun tidak dibutuhkan akhirnya.

Top-Down dan Bottom-Up (lanj.)

- Beberapa orang lebih alami untuk menggunakan *top-down*, sementara sisanya lebih terbiasa dengan *bottom-up*.
- Bergantung dari cara berpikir Anda, salah satunya mungkin lebih mudah Anda pelajari.
- Untuk orang yang telah berpengalaman, penggunaan *bottom-up* dan *top-down* dapat disesuaikan dengan soal yang dihadapi.

Penutup

- Terdapat dua versi DP, yaitu *top-down* dan *bottom-up*.
- Keduanya memiliki keuntungan dan kerugian, pilih yang tepat sesuai dengan kebutuhan soal.
- Kunci dari mengerjakan soal DP adalah mengidentifikasi pilihan keputusan yang bisa diambil, dan merumuskannya menjadi rumus rekursif.
- Anda perlu banyak latihan soal DP untuk menjadi terbiasa dengan melakukan formulasi rekursif ini.