13 - Pemrograman dinamis (*dynamic programming*)

[KOMS120403]

Desain dan Analisis Algoritma (2023/2024)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 12 (May 2024)

Daftar isi

- Prinsip Pemrograman Dinamis
- Contoh sederhana
- Penerapan lebih lanjut dari pemrograman dinamis
  - Permasalahan knapsack dan fungsi memori

Figure: Richard Ernest Bellman (1920-1984), penemu pemrograman dinamis

Prinsip pemrograman dinamis (*dynamic programming*) (1)

- Kata pemrograman tidak mengacu pada pemrograman komputer. Dalam hal ini, pemrograman berarti "perencanaan".
- Dinamis digunakan untuk merepresentasikan aspek waktu yang bervariasi dari masalah.

Pemrograman dinamis adalah metode pemecahan masalah dengan menguraikan solusi ke dalam serangkaian *tahapan*, sehingga solusi dari masalah dapat dilihat sebagai **sebuah rangkaian keputusan yang terurut yang terkait satu sama lain**.

Prinsip pemrograman dinamis (2)

- Biasanya digunakan untuk *optimization problems* (maksimasi/minimasi).
- Biasanya, submasalah ini muncul dari **rekurensi** yang menghubungkan solusi masalah yang diberikan dengan solusi dari submasalahnya yang lebih kecil.
- Dibandingkan dengan menyelesaikan submasalah yang tumpang tindih berulang kali, pemrograman dinamis mengarahkan untuk menyelesaikan setiap submasalah yang lebih kecil hanya sekali dan mencatat hasilnya dalam tabel dimana solusinya untuk masalah semula kemudian dapat diperoleh.

Bagian 1. Pemrograman dinamis pada barisan Fibonacci

Bilangan Fibonacci

Ingatlah bahwa deret Fibonacci didefinisikan sebagai berikut.

$$F(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F(n-1) + F(n-2), & n \geq 3 \end{cases}$$

Barisan Fibonacci: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Algoritma rekursif:

Algorithm 1 Fibonacci sequence recursively

1: **procedure** $\textsc{Fib}(n)$
2: **if** $n \leq 2$ **then** **return** 1
3: **end if**
4: **return** $(\textsc{Fib}(n-1) + \textsc{Fib}(n-2))$
5: **end procedure**

Diagram pembentukan barisan Fibonacci

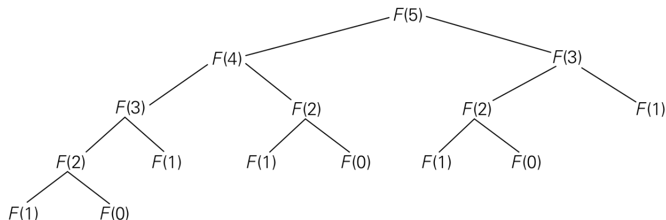


Figure: Pohon rekursi untuk menghitung angka Fibonacci ke-5

Algoritma Fibonacci rekursif memiliki kompleksitas eksponensial.

Tumpang tindih pada konstruksi bilangan Fibonacci

Cara menangani **perhitungan yang tumpang tindih**?

- Buat array 1-dimensi, dan isi dengan $(n+1)$ nilai berurutan dari $F(n)$, mulai dari $F(0)$, dan elemen terakhir adalah $F(n)$.

Algorithm 2 Fibonacci sequence iteratively

1: **procedure** FIB2(n)
2: $F[0] \leftarrow 0$; $F[1] \leftarrow 1$
3: **for** $i \leftarrow 2$ **to** n **do**
4: $F[i] \leftarrow F[i-1] + F[i-2]$
5: **end for**
6: **return** $F[n]$
7: **end procedure**

Kompleksitas waktu DP Fibonacci

Konsep:

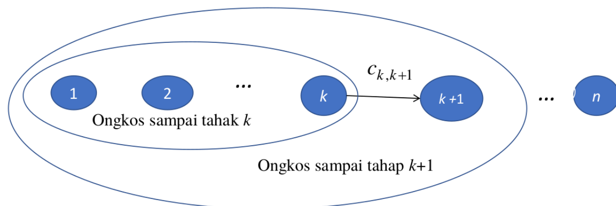
- Rekursi untuk $F[i]$ hanya dilakukan satu kali (saat pertama kali dipanggil).
- Untuk memanggil suatu nilai yang tersimpan, biayanya adalah $\Theta(1)$.
- Banyaknya panggilan dari fungsi non-memori (i.e. rekursi) adalah n, yaitu untuk $F[1], F[2], \ldots, F[n]$; yang masing-masing dilakukan dengan biaya $\Theta(1)$.
- Sehingga kompleksitas waktu adalah $\Theta(n)$

Prinsip optimalitas

Suatu masalah dikatakan memenuhi Prinsip Optimalitas **jika solusi optimal untuk setiap instance dari masalah optimisasi terdiri dari solusi optimal untuk sub-instance-nya**; yaitu jika solusi totalnya optimal, maka bagian dari solusi hingga langkah k juga optimal.

- **Implikasi:** jika kita bekerja dari langkah k ke langkah $k+1$, kita dapat menggunakan solusi optimal hingga langkah k, tanpa kembali ke keadaan awal.

$$\text{Biaya pada langkah } k+1 \quad = \quad (\text{biaya pada langkah } k) + (\text{biaya dari langkah } k \text{ ke langkah } k+1)$$



Kondisi Pemrograman Dinamis

Secara umum, berikut adalah beberapa persyaratan untuk menerapkan pemrograman dinamis pada suatu masalah
[*sumber:* Algorithm Design, by Kleinberg and Tardos]

1. Solusi untuk masalah asli dapat dihitung dari solusi submasalah (independen).
2. Banyaknya submasalah polinomial.
3. Ada pengurutan submasalah sedemikian rupa sehingga solusi submasalah hanya bergantung pada solusi submasalah yang mendahuluinya dalam urutan ini.

Tahapan pemrograman dinamis

1. **Definisikan sub-masalah:** hal ini merupakan langkah kritis. Biasanya struktur perulangan mengikuti secara alami setelah mendefinisikan submasalah.
2. **Rekurensi:** untuk membuat algoritma, nyatakan solusi untuk submasalah dalam bentuk formula rekursi dari solusi untuk submasalah yang lebih kecil.
3. **Kebenaran:** buktikan bahwa perulangannya benar, biasanya dengan teknik pembuktian *induksi*.
4. **Kompleksitas:** untuk menganalisis kompleksitas waktu, biasanya dengan formula berikut:

$$runtime = \#\ subproblems \times timeToSolveEachSubproblem$$

Pendekatan

1. Pendekatan **forward/top-down** (tabulation)

   Perhitungan dilakukan dari tahap $1, 2, \ldots, n-1, n$.
   Urutan variabel keputusan: $x_1, x_2, \ldots, x_n$.

2. Pendekatan **backward/bottom-up** (memoization)

   Perhitungan dilakukan dari tahap $n, n-1, \ldots, 2, 1$.
   Urutan variabel keputusan: $x_n, x_{n-1}, \ldots, x_1$.

Bagian 2. Contoh sederhana penerapan pemrograman dinamis

1. Masalah barisan koin (*coin row*)
2. Masalah penukaran (*coin exchange*)
3. Masalah pengumpulan koin (*coin collection*)

1. Masalah barisan koin (*coin row*)

1. Masalah barisan koin (1)

Permasalahan

Diberikan deretan n koin yang nilainya berupa bilangan bulat positif $c_1, c_2, \ldots, c_n$*, yang belum tentu berbeda. Tujuannya adalah untuk mengambil jumlah uang* *maksimum* *dengan batasan bahwa tidak ada dua koin yang berdekatan di baris awal yang dapat diambil.*

1. Masalah barisan koin (1)

Permasalahan

Diberikan deretan n koin yang nilainya berupa bilangan bulat positif $c_1, c_2, \ldots, c_n$, yang belum tentu berbeda. Tujuannya adalah untuk mengambil jumlah uang maksimum dengan batasan bahwa tidak ada dua koin yang berdekatan di baris awal yang dapat diambil.

Strategi: Misalkan $F(n)$ adalah jumlah maksimum yang dapat diambil dari deretan n koin. Bagaimanakah cara menurunkan rumus rekursif untuk $F(n)$?

- Dua partisi pilihan koin yang diizinkan:
  1. Grup yang *menyertakan koin terakhir*
  2. Grup yang *tidak menyertakan koin terakhir*

c_1 c_1 c_1 $\cdots$ c_{n-2} c_{n-1} c_n $\qquad c_n + F(n-2)$

c_1 c_1 c_1 $\cdots$ c_{n-2} c_{n-1} c_n $\qquad F(n-1)$

1. Masalah barisan koin (2)

Fungsi rekursif:

$$\begin{cases} F(n) &= \max\{c_n + F(n-2),\ F(n-1)\} \text{ for } n > 1 \\ F(0) &= 0,\ \ F(1) = c_1 \end{cases}$$

Jadi, $F(n)$ dapat dihitung seperti pada deret Fibonacci.

Algorithm 3 Coin row

```
1: procedure CoinRow(C[1..n])
2:     F[0] ← 0;  F[1] ← C[1]
3:     for i ← 2 to n do
4:         F(i) ← max{C[i] + F(i − 2), F(i − 1)}
5:     end for
6:     return F[n]
7: end procedure
```

1. Masalah barisan koin (3)

$F[0] = 0, F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Figure: Memecahkan masalah baris koin dengan pemrograman dinamis untuk baris koin 5, 1, 2, 10, 6, 2, dengan solusi optimal $\{c_1, c_4, c_6\}$ dan nilai optimal 17.

1. Kompleksitas waktu "Coin row"

Hitunglah kompleksitas waktu algoritma di atas.

Kompleksitas waktu dari algoritma DP ini tergantung pada jumlah submasalah yang harus diselesaikan dan waktu yang diperlukan untuk menyelesaikan setiap submasalah. Dalam hal ini, kita memiliki n submasalah (dimana n adalah panjang array A), dan waktu yang diperlukan untuk menyelesaikan setiap submasalah adalah konstan, yaitu $\mathcal{O}(1)$, karena hanya melibatkan operasi perbandingan dan penambahan.

Jadi, kompleksitas waktu dari algoritma DP untuk Coin-row problem adalah $\mathcal{O}(n)$, di mana n adalah panjang array A.

2. Masalah penukaran (*coin exchange*)

2. Masalah penukaran (1)

Permasalahan

Diberikan cek sejumlah n, yang akan ditukar dengan koin nilai $d_1 < d_2 < \cdots < d_m$ dimana $d_1 = 1$. Asumsikan bahwa kita memiliki jumlah koin yang tidak terbatas untuk setiap koin $d_1, d_2, \cdots, d_m$. Tentukan jumlah koin minimum yang dibutuhkan.

2. Masalah penukaran (1)

Permasalahan

Diberikan cek sejumlah n, yang akan ditukar dengan koin nilai $d_1 < d_2 < \cdots < d_m$ dimana $d_1 = 1$. Asumsikan bahwa kita memiliki jumlah koin yang tidak terbatas untuk setiap koin $d_1, d_2, \cdots, d_m$. Tentukan jumlah koin minimum yang dibutuhkan.

Strategi: Misalkan $F(n)$: jumlah minimum koin yang nilainya berjumlah n, dan tentukan $F(0) = 0$. Bagaimanakah cara menurunkan rumus rekursif untuk $F(n)$?

- Jumlah n hanya dapat diperoleh dengan menambahkan satu koin nominal d_j ke jumlah $n - d_j$ untuk $j = 1, 2, \ldots, m$ sehingga $n \geq d_j$.
- Minimalkan $F(n - d_j) + 1$.

Fungsi rekursif:

$$\begin{cases} F(n) & = \min_{j; n \geq d_j} F(n - d_j) + 1 \text{ for } n > 0 \\ F(0) & = 0 \end{cases}$$

2. Masalah penukaran (2)

Algorithm 4 ChangeMaking

```
 1: procedure MinCoinChange(D[1..m], n)
 2:     input: positive integer n, and array D[1..m] of increasing positive integers indi-
        cating the coin denominations where D[1] = 1
 3:     output: the minimum number of coins that add up to n
 4:     F[0] ← 0
 5:     for i ← 1 to n do
 6:         temp ← ∞;  j ← 1
 7:         while j ≤ m and i ≥ D[j] do
 8:             temp ← min(F[i − D[j]], temp)
 9:             j ← j + 1
10:         end while
11:         F[i] ← temp + 1
12:     end for
13:     return F[n]
14: end procedure
```

2. Masalah penukaran (3)

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

Figure: Penerapan algoritma MinCoinChange untuk nilai $n = 6$ dan satuan koin 1, 3, dan 4.

2. Kompleksitas waktu "Coin exchange"

Hitunglah kompleksitas waktu algoritma di atas.

Kompleksitas waktu dari algoritma DP ini tergantung pada jumlah submasalah yang harus diselesaikan dan waktu yang diperlukan untuk menyelesaikan setiap submasalah. Dalam hal ini, kita memiliki n submasalah (yaitu, setiap nilai dari 1 hingga n), dan waktu yang diperlukan untuk menyelesaikan setiap submasalah tergantung pada jumlah koin yang tersedia, yaitu m.

Jadi, kompleksitas waktu dari algoritma DP untuk Change-making problem adalah $\mathcal{O}(nm)$, di mana n adalah jumlah uang yang perlu ditukarkan, dan m adalah jumlah jenis koin yang tersedia.

3. Masalah pengumpulan koin (*coin collection*)

3. Masalah pengumpulan koin (1)

Permasalahan

Beberapa koin ditempatkan dalam papan berukuran $n \times m$, tidak lebih dari satu koin per sel. Sebuah robot yang ditempatkan di sel kiri-atas papan, bertujuan mengumpulkan koin sebanyak mungkin dan membawanya ke sel kanan-bawah.

- *Pada setiap langkah, robot dapat memindahkan satu sel ke kanan atau satu sel ke bawah dari lokasinya saat ini.*
- *Ketika robot mengunjungi sel dengan koin, ia selalu mengambil koin itu.*

Rancang algoritma untuk menemukan jumlah koin maksimum yang dapat dikumpulkan robot dan jalur yang harus diikuti untuk melakukannya.

3. Masalah pengumpulan koin (1)

Permasalahan

Beberapa koin ditempatkan dalam papan berukuran $n \times m$, tidak lebih dari satu koin per sel. Sebuah robot yang ditempatkan di sel kiri-atas papan, bertujuan mengumpulkan koin sebanyak mungkin dan membawanya ke sel kanan-bawah.

- *Pada setiap langkah, robot dapat memindahkan satu sel ke kanan atau satu sel ke bawah dari lokasinya saat ini.*
- *Ketika robot mengunjungi sel dengan koin, ia selalu mengambil koin itu.*

Rancang algoritma untuk menemukan jumlah koin maksimum yang dapat dikumpulkan robot dan jalur yang harus diikuti untuk melakukannya.

Strategi: Misalkan $F(i,j)$ adalah jumlah koin terbesar yang dapat dikumpulkan dan dibawa robot ke sel (i,j) di baris ke-i dan kolom ke-j di papan.

- (i,j) dapat dijangkau dari sel $(i-1,j)$ (atas) atau sel $(i,j-1)$ (kiri).
- Untuk sel di baris pertama (atau kolom pertama), asumsikan bahwa $F(i-1,j) = 0$ (atau $F(i,j-1) = 0$).

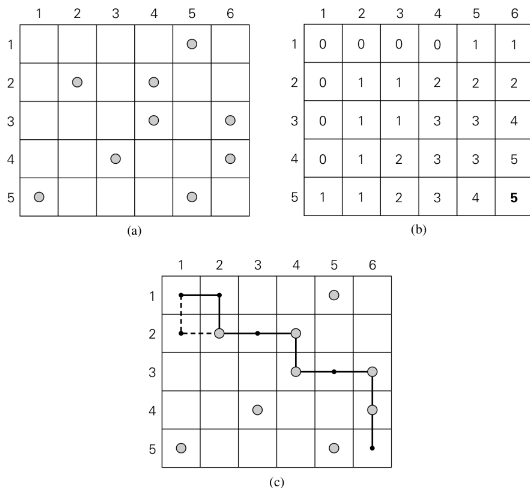
3. Masalah pengumpulan koin (2)

$$\begin{cases} F(i,j) &= \max\{F(i-1,j), F(i,j-1)\} + c_{ij} \quad \textit{untuk} \;\; 1 \leq i < n, \;\; 1 \leq j \leq m \\ F(0,j) &= 0 \; \textit{for} \; 1 \leq j \leq m \; \textit{and} \; F(i,0) \quad \textit{untuk} \;\; 1 \leq i \leq n \end{cases}$$

Algorithm 5 Robot coin collection

```
 1: procedure RobotCoinCollection(C[1..n, 1..m])
 2:     F[1,1] ← C[1,1]
 3:     for j ← 2 to m do
 4:         F[1,j] ← F[1,j − 1] + C[1,j]
 5:     end for
 6:     for i ← 2 to n do
 7:         F[i,1] ← F[i − 1,1] + C[i,1]
 8:         for j ← 2 to m do
 9:             F[i,j] ← max{F[i − 1,j], F[i,j − 1]} + C[i,j]
10:         end for
11:     end for
12:     return F[n,m]
13: end procedure
```

3. Masalah pengumpulan koin (3)



	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

Figure: (a) Koin untuk dikumpulkan. (b) Hasil algoritma pemrograman dinamis. (c) Dua jalur untuk mengumpulkan 5 koin, jumlah koin maksimum yang mungkin.

3. Kompleksitas waktu "Coin collection"

Hitunglah kompleksitas waktu algoritma di atas.

Untuk menghitung kompleksitas waktu dari algoritma DP ini, kita perlu mempertimbangkan berapa banyak submasalah yang harus diselesaikan dan berapa banyak operasi yang diperlukan untuk menyelesaikan setiap submasalah.

Karena kita harus mengunjungi setiap sel di grid sekali dan melakukan operasi konstan untuk menghitung nilai-nilai DP, kompleksitas waktu dari algoritma DP untuk Coin collection problem adalah $\mathcal{O}(nm)$, di mana n adalah jumlah baris dan m adalah jumlah kolom dalam grid.

Bagian 3. Pemrograman dinamis pada permasalahan knapsack

Knapsack problem (1)

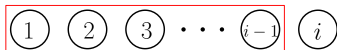
Permasalahan

Diberikan n objek dengan bobot $w_1, \ldots, w_n$ dan nilai $v_1, \ldots, v_n$, serta sebuah ransel berkapasitas W. Temukan sub-himpunan objek dengan nilai tertinggi dan memenuhi kapasitas ransel.

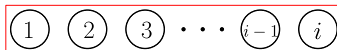
Strategi: Misalkan $F(i, j)$ adalah nilai solusi optimal untuk instance yang ditentukan oleh i item pertama $1 \leq i \leq n$,

- dengan bobot $w_1, \ldots, w_i$ dan nilai $v_1, \ldots, v_i$
- kapasitas ransel j, dengan $1 \leq j \leq W$.

Kita ingin memutuskan apakah objek i disertakan atau tidak.



$F(i-1, j)$



$v_i + F(i-1, j-w_i)$

Knapsack problem (2)

$$F(i,j) = \begin{cases} \max\{F(i-1,j), v_i + F(i-1,j-w_i)\} & \textit{jika } j - w_i \geq 0 \\ F(i-1,j), & \textit{jika } j - w_i < 0 \end{cases}$$

dengan kondisi awal:

$$F(0,j) = 0 \text{ untuk } j \geq 0 \quad \text{dan} \quad F(i,0) = 0 \text{ untuk } i \geq 0$$

Tujuan: untuk menemukan $F(n, W)$, yakni nilai maksimal subset dari n item yang diberikan yang sesuai dengan ransel kapasitas W; beserta subset yang memenuhi nilai optimal tersebut.

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Figure: Tabel untuk memecahkan masalah knapsack dengan pemrograman dinamis.

Knapsack problem (3)

Contoh

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Figure: Contoh penyelesaian instance masalah knapsack dengan algoritma pemrograman dinamis.

Kompleksitas waktu

Misalkan n adalah jumlah item dan W adalah kapasitas knapsack.

- Terdapat $(n+1) \times W$ sel dalam tabel DP yang harus diisi.
- Untuk mengisi setiap sel, kita melakukan satu operasi perbandingan dan satu operasi penjumlahan.
- Jadi, kompleksitas waktu total adalah $\mathcal{O}(n \times W)$.

Dengan demikian, kompleksitas waktu dari algoritma DP untuk Knapsack problem adalah $\mathcal{O}(n \times W)$.

Bagian 4. *Memoization*

(studi kasus pada Knapsack problem)

Prinsip *memoization*

Pada dasarnya, pemrograman dinamis memecahkan masalah yang memiliki hubungan perulangan.

- Menggunakan perulangan secara langsung dalam algoritma rekursif memiliki kelemahan, yaitu kemungkinan untuk memecahkan sub-masalah umum berkali-kali, menghasilkan kompleksitas yang mencapai eksponensial.
- Teknik pemrograman dinamis memiliki kelemahan bahwa beberapa sub-masalah mungkin tidak perlu dipecahkan.

Tujuan: untuk mendapatkan solusi terbaik (optimal) dari kedua pendekatan, dimana semua sub-masalah yang diperlukan diselesaikan hanya sekali.

Prinsip *memoization*

- Teknik ini menggunakan pendekatan top-down, algoritma rekursif, dengan tabel solusi sub-masalah.
- Sebelum menentukan solusi secara rekursif, algoritma memeriksa apakah sub-masalah sudah diselesaikan dengan memeriksa tabel.
- Jika tabel memiliki nilai yang valid, maka algoritma menggunakan nilai tabel yang lain, dilanjutkan dengan solusi rekursif.

Ingat kembali definisi $F(i, j)$, yakni nilai solusi optimal untuk instance yang ditentukan oleh i objek pertama dengan kapasitas knapsack j.

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \textit{jika } j - w_i \geq 0 \\ F(i-1, j), & \textit{jika } j - w_i < 0 \end{cases}$$

Algoritma *memoization*

Algorithm 6 MF Knapsack

```
1:  procedure MFK(i, j)
2:      input: i ∈ ℤ≥0 menunjukkan jumlah item pertama yang telah diperhitungkan
        dan j ∈ ℤ≥0 menunjukkan kapasitas ransel
3:      output: Nilai subset layak optimal dari item i pertama
4:      variables: variabel global array berbobot Wt[1..n], nilai V[1..n], dan tabel
        F[0..n, 0..W] yang entri-entrinya diinisialisasi dengan -1 kecuali untuk baris 0 dan
        kolom 0 yang diinisialisasi dengan 0.
5:      if F[i, j] < 0 then
6:          if j < Wt[i] then
7:              value ← MFK(i − 1, j)
8:          else
9:              value ← max{MFK(i − 1, j), V[i] + MFK(i − 1, j − Wt[i])}
10:         end if
11:         F[i, j] ← value
12:     end if
13:     return F[i, j]
14: end procedure
```

Contoh *memoization*

Mari terapkan prosedur MFK pada contoh sebelumnya:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$

	i	capacity j 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

Figure: Contoh penyelesaian instance masalah knapsack dengan algoritma *memoization*.

Contoh *memoization*

Implementasi algoritma langkah demi langkah adalah sebagai berikut:

- $N = 4, W = 5, w_1 = 2, w_2 = 1, w_3 = 3, w_4 = 2, v_1 = 12, v_2 = 10, v_3 = 20, v_4 = 15$
- $V[4,5] = \max\{V[3,5], 15 + V[3,3]\}$ (since $v_4 = 15, w_4 = 2, j = 15, w_4 < j \rightarrow$ apply 1st case)
- $V[3,5] = \max\{V[2,5], 20 + V[2,2]\}$
- $V[3,3] = \max\{V[2,3], 20 + V[2,0]\}$
- $V[2,5] = \max\{V[1,5], 10 + V[1,4]\}$
- $V[2,2] = \max\{V[1,2], 10 + V[1,1]\}$
- $V[2,3] = \max\{V[1,3], 10 + V[1,2]\}$
- $V[2,0] = V[1,0] = 0$ (since $w_2 = 1, j = 0, w_2 > j \rightarrow$ apply 2nd case)
- $V[1,5] = \max\{V[0,5], 12 + V[0,3]\} = \max\{0, 12 + 0\} = 12$
- $V[1,4] = \max\{V[0,4], 12 + V[0,2]\} = \max\{0, 12 + 0\} = 12$
- $V[1,2] = \max\{V[0,2], 12 + V[0,0]\} = \max\{0, 12 + 0\} = 12$
- $V[1,1] = V[0,1] = 0$
- $V[1,3] = \max\{V[0,3], 12 + V[0,1]\} = \max\{0, 12 + 0\} = 12$

Selanjutnya, lakukan substitusi mundur:

- $V[2,3] = \max\{V[1,3], 10 + V[1,2]\} = \max\{12, 10 + 12\} = 22$
- $V[2,2] = \max\{V[1,2], 10 + V[1,1]\} = \max\{12, 10 + 0\} = 12$
- $V[2,5] = \max\{V[1,5], 10 + V[1,4]\} = \max\{12, 10 + 12\} = 22$
- $V[3,3] = \max\{V[2,3], 20 + V[2,0]\} = \max\{22, 20 + 0\} = 22$
- $V[3,5] = \max\{V[2,5], 20 + V[2,2]\} = \max\{22, 20 + 12\} = 32$
- $V[4,5] = \max\{V[3,5], 15 + V[3,3]\} = \max\{32, 15 + 22\} = 37$

Dynamic Programming vs. Divide-and-Conquer

Basic Idea:

- DP: Menyimpan hasil dari submasalah yang sama untuk menghindari pengulangan perhitungan.
- Divide-and-Conquer: Memecah masalah menjadi submasalah yang lebih kecil, menyelesaikan masing-masing secara rekursif, dan kemudian menggabungkan solusinya.

Subproblems:

- DP: Masalah dibagi menjadi submasalah, yang kemudian diselesaikan secara independen.
- Divide-and-Conquer: Masalah dibagi menjadi submasalah yang tidak saling bergantung satu sama lain.

Dynamic Programming vs. Divide-and-Conquer (2)

Overlapping Subproblems:

- DP: Memiliki sifat overlapping subproblems, artinya beberapa submasalah akan diselesaikan berulang kali.
- Divide-and-Conquer: Submasalah yang dipecahkan bersifat unik dan tidak tumpang tindih.

Memorization:

- DP: Biasanya menggunakan tabel atau array untuk menyimpan hasil submasalah yang telah diselesaikan untuk digunakan kembali.
- Divide-and-Conquer: Tidak memerlukan tahap memorisasi, karena setiap submasalah diselesaikan hanya satu kali.

end of slide...