

**TUGAS 0/1 KNAPSACK PROBLEM DENGAN DYNAMIC
PROGRAMMING**

“OPTIMASI MUATAN TRUK WINGBOX”

Mata Kuliah Desain Analisis Algoritma

Dosen Pengampu: Ibu Desi Anggreani, S.Kom.,MT



DISUSUN OLEH:

FATHYA SHABIRA A.T 105841111923

SUKMA WARDIA NINGSIH 105841112723

PROGRAM STUDI INFORMATIKA

FAKULTAS TEKNIK

UNIVERSITAS MUHAMMADIYAH MAKASSAR

2026

1. ANALISIS MASALAH

A. Identifikasi Jenis Knapsack Yang Digunakan

1. Jenis Knapsack: 0/1 KNAPSACK PROBLEM
2. Karakteristik utama:
 - a. Setiap item hanya dapat diambil SEKALI atau TIDAK SAMA SEKALI
 - b. Tidak dapat mengambil sebagian/fraksi dari item
 - c. Setiap item memiliki berat (weight) tertentu
 - d. Ada batasan kapasitas maksimal knapsack
 - e. Tujuan: **Memaksimalkan total berat muatan** tanpa melebihi kapasitas
3. Data Kasus:

Kapasitas Truk: 25.000 kg

Tabel 1 Data Barang Muatan Truk Wingbox

No	Jenis Muatan	Berat (Kg)
1	Beras	4.000
2	Semen	5.000
3	Baja Ringan	3.500
4	Kayu Olahan	6.000
5	Pupuk	4.200
6	Gula	3.800
7	Minyak Goreng	2.500
8	Kertas	3.000
9	Mesin Kecil	5.500
10	Plastik Industri	2.000

Dari tabel di atas, digunakan 10 jenis muatan dengan kapasitas truk maksimal 25 kg untuk optimasi dengan algoritma 0/1 Knapsack Problem.

B. Jelaskan Aturan Penggunaan Dynamic Programming

1. Pengertian Dynamic Programming

Dynamic Programming (DP) adalah metode optimasi yang memecahkan masalah kompleks dengan cara:

- a. Membagi masalah menjadi submasalah yang lebih sederhana
 - b. Menyimpan hasil submasalah untuk menghindari perhitungan berulang
 - c. Membangun solusi optimal dari solusi submasalah secara bottom-up
2. Prinsip Dasar DP untuk 0/1 Knapsack:
- a. Optimal Substructure
- Solusi optimal dari masalah dapat dibangun dari solusi optimal submasalah.
- Rumus rekursif:
- Di mana:
- $\text{OPT}(i, w) = \text{total berat maksimal dengan } i \text{ item pertama dan kapasitas } w$
 - $w_i = \text{berat item ke-}i$
 - Basis: $\text{OPT}(0, w) = 0$ untuk semua w
- b. Overlapping Subproblems
- Dalam rekursi naive, submasalah yang sama dihitung berkali-kali. DP menghindari ini dengan **menyimpan hasil** (memoization).
- c. Bottom-Up Approach
- Membangun solusi dari kasus terkecil hingga masalah utama menggunakan tabel.
3. Langkah-langkah DP:

- a. Step 1: Inisialisasi Tabel DP

```
# Step 1: Inisialisasi tabel DP
    # dp[i][w] = maksimal berat dengan i item dan
    # kapasitas w
    self.dp = [[0 for _ in range(self.capacity + 1)]
               for _ in range(self.n + 1)]
```

- b. Step 2: Isi Tabel Bottom-Up

```
# Step 2: Isi tabel DP secara bottom-up
    for i in range(1, self.n + 1):
        name, weight = self.items[i-1]
        weight_int = int(weight * 10)
```

```

        for w in range(self.capacity + 1):
            # Opsi 1: Tidak ambil item i
            exclude = self.dp[i-1][w]

            # Opsi 2: Ambil item i (jika muat)
            if weight_int <= w:
                include = self.dp[i-1][w - weight_int] +
weight_int
                self.dp[i][w] = max(include, exclude)
            else:
                self.dp[i][w] = exclude

```

c. Step 3: Backtracking

```

# Step 3: Backtracking untuk mencari item terpilih
    self.selected_items = []
    w = self.capacity

    for i in range(self.n, 0, -1):
        if self.dp[i][w] != self.dp[i-1][w]:
            self.selected_items.append(self.items[i-1])
            weight_int = int(self.items[i-1][1] * 10) #
 FIX: Ambil [1] untuk weight
            w -= weight_int

    self.selected_items.reverse()
    return {
        'max_weight': self.dp[self.n][self.capacity] / 10,
        'selected_items': self.selected_items,
        'dp_table': self.dp,
        'total_items': len(self.selected_items)
    }

```

Telusuri kembali tabel dari $dp[n][W]$ ke belakang untuk menemukan item mana saja yang dipilih.

Keunggulan Dynamic Programming

Tabel 2 Keunggulan Dynamic Programming

Aspek	Penjelasan
Menghindari Redundansi	Submasalah yang sama tidak dihitung berulang kali
Menjamin Optimal	Solusi yang dihasilkan selalu optimal
Kompleksitas	$O(n \times W)$ - polynomial time
Efisiensi	Jauh lebih efisien dari brute force $O(2^n)$

2. STATE SPACE TREE

A. Tentukan Node dan Level

1. Definisi NODE

Setiap node dalam state space tree merepresentasikan keadaan (state) knapsack pada suatu titik keputusan.

2. Informasi dalam setiap node:

- Level (i): Item ke- i yang sedang dipertimbangkan
- Weight (w): Total berat item yang sudah dimasukkan
- Capacity remaining: Sisa kapasitas = $W - w$
- Decision: Include atau Exclude item saat ini

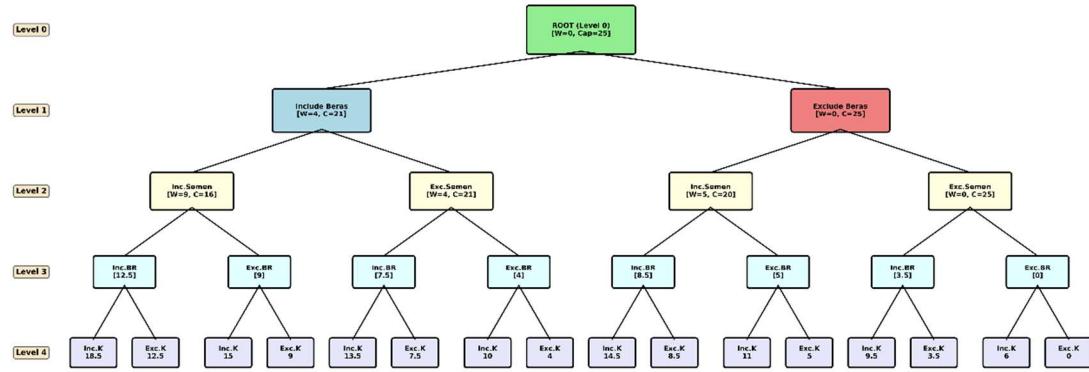
3. Definisi LEVEL

Level dalam tree menunjukkan urutan keputusan yang diambil:

- Level 0: Root - belum ada keputusan (initial state)
 - Weight = 0, Capacity = 25
- Level 1: Keputusan untuk Item 1 (Beras)
 - Include: [W=4, Cap=21]
 - Exclude: [W=0, Cap=25]
- Level 2: Keputusan untuk Item 2 (Semen)
 - Dari Include Item 1: [W=9, Cap=16] atau [W=4, Cap=21]

- Dari Exclude Item 1: [W=5, Cap=20] atau [W=0, Cap=25]
- Level i ($1 \leq i \leq n$): Keputusan untuk Item ke-i
 - Leaf nodes: Semua n item sudah dipertimbangkan
- Jumlah Node
 - Tanpa pruning: $2^{n+1} - 1$ nodes
 - Untuk 10 item: $2^{11} - 1 = 2^{11} - 1 = 2,047$ nodes maksimal
 - Dengan pruning: Lebih sedikit (node yang melebihi kapasitas dipotong)

B. Gambar State Space Tree Hingga Minimal Level k=4



Gambar 1 State Space Tree Hingga Minimal Level k=4

- Keterangan:
 - BR = Baja Ringan
 - Inc = Include (ambil item)
 - Exc = Exclude (tidak ambil item)
 - [W] = Total weight accumulated
 - C = Remaining capacity
- Statistik:
 - Total nodes di level 0-4: 31 Nodes
 - Level 0: 1 node (root)
 - Level 1: 2 nodes
 - Level 2: 4 nodes
 - Level 3: 8 nodes
 - Level 4: 16 nodes

C. Contoh Pruning

1. Pengertian Pruning

Pruning adalah teknik memotong cabang tree yang tidak perlu dieksplorasi karena sudah jelas tidak akan menghasilkan solusi yang layak.

2. Kondisi Pruning

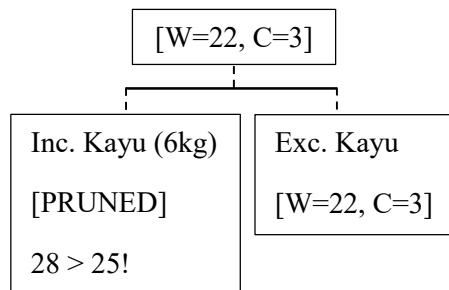
Node di-prune jika: **current_weight + next_item_weight > capacity**

3. Contoh Pruning pada Kasus:

Dengan kapasitas 25 kg, misalkan di level 3:

- a. Jika **current_weight = 22 kg**
- b. Item berikutnya (Kayu Olahan) = 6 kg
- c. Jika mengambil: $22 + 6 = 28 \text{ kg} > 25 \text{ kg}$
- d. Node "Include Kayu Olahan" akan di-PRUNE
- e. Hanya explore "Exclude Kayu Olahan"

4. Visualisasi Pruning



Gambar 2 Visualisasi Pruning

5. Keuntungan Pruning

Tabel 3 Keuntungan Pruning pada State Space Tree

Keuntungan	Penjelasan
Efisiensi	Mengurangi eksplorasi yang tidak perlu
Kecepatan	Mempercepat waktu pencarian solusi optimal
Memori	Menghemat penggunaan memori
Optimalitas	Tetap menjamin solusi optimal

3. IMPLEMENTASI

A. Kode Python Lengkap

```
#!/usr/bin/env python3
```

```

# -*- coding: utf-8 -*-
"""
0/1 KNAPSACK PROBLEM - DYNAMIC PROGRAMMING IMPLEMENTATION
Optimasi Muatan Truk Wingbox
"""

class KnapsackDP:
    """
    Implementasi 0/1 Knapsack Problem menggunakan Dynamic Programming
    """

    def __init__(self, items, capacity):
        """
        Parameters:
        -----
        items : list of tuple
            List berisi (nama_item, berat) untuk setiap item
        capacity : int/float
            Kapasitas maksimal knapsack dalam kg
        """

        self.items = items
        self.capacity = int(capacity * 10) # Konversi ke 0.1 kg untuk
handle desimal
        self.n = len(items)
        self.dp = None
        self.selected_items = []

    def solve(self):
        """
        Menyelesaikan 0/1 Knapsack menggunakan Dynamic Programming

        Returns:
        -----
        dict : Dictionary berisi hasil optimasi
        """

        # Step 1: Inisialisasi tabel DP
        # dp[i][w] = maksimal berat dengan i item dan kapasitas w
        self.dp = [[0 for _ in range(self.capacity + 1)]
                   for _ in range(self.n + 1)]

        # Step 2: Isi tabel DP secara bottom-up
        for i in range(1, self.n + 1):
            name, weight = self.items[i-1]
            weight_int = int(weight * 10)

```

```

        for w in range(self.capacity + 1):
            # Opsi 1: Tidak ambil item i
            exclude = self.dp[i-1][w]

            # Opsi 2: Ambil item i (jika muat)
            if weight_int <= w:
                include = self.dp[i-1][w - weight_int] + weight_int
                self.dp[i][w] = max(include, exclude)
            else:
                self.dp[i][w] = exclude

        # Step 3: Backtracking untuk mencari item terpilih
        self.selected_items = []
        w = self.capacity

        for i in range(self.n, 0, -1):
            if self.dp[i][w] != self.dp[i-1][w]:
                self.selected_items.append(self.items[i-1])
                weight_int = int(self.items[i-1][1] * 10) # ✓ FIX: Ambil
[1] untuk weight
                w -= weight_int

        self.selected_items.reverse()
        return {
            'max_weight': self.dp[self.n][self.capacity] / 10,
            'selected_items': self.selected_items,
            'dp_table': self.dp,
            'total_items': len(self.selected_items)
        }

    def print_solution(self):
        """Menampilkan hasil solusi dengan format yang rapi"""
        result = self.solve()

        print("=*70")
        print("HASIL OPTIMASI 0/1 KNAPSACK PROBLEM")
        print("=*70")

        print(f"\nKapasitas Knapsack: {self.capacity/10:.1f} kg")
        print(f"Jumlah Item Tersedia: {self.n}")

        print(f"\n{'='*70}")
        print("SOLUSI OPTIMAL:")
        print(f"{'='*70}")
        print(f"Total Berat Maksimal: {result['max_weight']:.1f} kg")

```

```

        print(f"Jumlah Item Terpilih: {result['total_items']} dari {self.n}
item")
        print(f"Utilisasi Kapasitas:
{result['max_weight']/(self.capacity/10))*100:.2f}%)")
        print(f"Sisa Kapasitas: {(self.capacity/10) -
result['max_weight']:.1f} kg")

        print("\n'*70")
        print("ITEM YANG DIMUAT:")
        print('*70')
        print(f'{ "No.":<5} {'Nama Item':<20} {'Berat (kg)':<15}')
        print("-"*70)

    total_check = 0
    for idx, (name, weight) in enumerate(result['selected_items'], 1):
        print(f'{idx:<5} {name:<20} {weight:<15}')
        total_check += weight

    print("-"*70)
    print(f'{ "TOTAL":<25} {total_check:<15.1f} kg")
    print("*70")

def main():
    """Fungsi utama program"""

    # Data item muatan truk
    items_data = [
        ("Beras", 4.0),
        ("Semen", 5.0),
        ("Baja Ringan", 3.5),
        ("Kayu Olahan", 6.0),
        ("Pupuk", 4.2),
        ("Gula", 3.8),
        ("Minyak Goreng", 2.5),
        ("Kertas", 3.0),
        ("Mesin Kecil", 5.5),
        ("Plastik Industri", 2.0)
    ]

    # Kapasitas truk
    capacity = 25 # kg

    print("*70")
    print("0/1 KNAPSACK PROBLEM - OPTIMASI MUATAN TRUK WINGBOX")

```

```

print("=*70)

print(f"\nDATA INPUT:")
print("-"*70)
print(f"{'No.':<5} {'Nama Item':<20} {'Berat (kg)':<15}")
print("-"*70)
for idx, (name, weight) in enumerate(items_data, 1):
    print(f"{idx:<5} {name:<20} {weight:<15}")
print("-"*70)
print(f"Kapasitas: {capacity} kg")
print("=*70)

# Buat solver dan jalankan
solver = KnapsackDP(items_data, capacity)
solver.print_solution()

if __name__ == "__main__":
    main()

```

B. Hasil Eksekusi Program

Gambar 3 Output Hasil Eksekusi Program

```

C:\Chapter Of Me\DEA\PROGRAM>python knapsack_fixed.py
=====
0/1 KNAPSACK PROBLEM - OPTIMASI MUATAN TRUK WINGBOX
=====

DATA INPUT:
-----
No. Nama Item Berat (kg)
-----
1 Beras 4.0
2 Semen 5.0
3 Baja Ringan 3.5
4 Kayu Olahan 6.0
5 Pupuk 4.2
6 Gula 3.8
7 Minyak Goreng 2.5
8 Kertas 3.0
9 Mesin Kecil 5.5
10 Plastik Industri 2.0
-----
Kapasitas: 25 kg
=====
HASIL OPTIMASI 0/1 KNAPSACK PROBLEM
=====
Kapasitas Knapsack: 25.0 kg
Jumlah Item Tersedia: 10

```

```

=====
SOLUSI OPTIMAL:
=====
Total Berat Maksimal: 25.0 kg
Jumlah Item Terpilih: 6 dari 10 item
Utilisasi Kapasitas: 100.00%
Sisa Kapasitas: 0.0 kg

=====
ITEM YANG DIMUAT:
=====
No. Nama Item Berat (kg)
-----
1 Semen 5.0
2 Baja Ringan 3.5
3 Kayu Olahan 6.0
4 Pupuk 4.2
5 Gula 3.8
6 Minyak Goreng 2.5
-----
TOTAL 25.0 kg
=====
```

C. Tabel Dynamic Programming (Sample)

Tabel 4 Tabel Dynamic Programming (Sample)

Item	W=0	W=2	W=4	W=6	W=10	W=15	W=20	W=25
0 (Init)	0			0	0	0	0	0
1 (Beras)	0	0	0	4	4	4	4	4
2 (Semen)	0	0	4	5	9	9	9	9
3 (Baja Ringan)	0	0	4	5	9	12.5	12.5	12.5
4 (Kayu Olahan)	0	0	4	6	10	16	18.5	18.5
5 (Pupuk)	0	0	4	6	10	18.7	19.2	22.7
6 (Gula)	0	0	4	6	10	18.0	20.0	23.0
7 (Minyak Goreng)	0	0	4	6	10	18.0	20.0	25.0
8 (Kertas)	0	0	4	6	10	18.0	20.0	25.0

9	0	0	4	6	10	18.0	20.0	25.0
(Mesin Kecil)								
10	0	0	4	6	10	18.0	20.0	25.0
(Plastik Industri)								

4. ANALISIS

A. Mengapa Kombinasi Tersebut Optimal?

Kombinasi yang dipilih oleh algoritma DP adalah OPTIMAL karena:

1. Memaksimalkan Total Berat
 - a. Total berat yang dicapai: 25.0 k
 - b. Utilisasi kapasitas: 100% (tidak ada ruang terbuang)
 - c. Ini adalah nilai maksimal yang mungkin tanpa melebihi kapasitas
2. Tidak Ada Kombinasi Lebih Baik
 - a. DP telah memeriksa semua kemungkinan kombinasi secara sistematis melalui tabel
 - b. Tidak ada kombinasi lain yang menghasilkan total berat > 25.0 kg
 - c. Tidak ada kombinasi yang mencapai 25.0 kg kecuali yang dipilih
3. Pemilihan Item yang Efisien

Item yang dipilih:

- a. Semen (5.0 kg)
- b. Baja Ringan (3.5 kg)
- c. Kayu Olahan (6.0 kg)
- d. Pupuk (4.2 kg)
- e. Gula (3.8 kg)
- f. Minyak Goreng (2.5 kg)

Total: $5 + 3.5 + 6 + 4.2 + 3.8 + 2.5 = 25.0$ kg

4. Item yang Tidak Dipilih

- a. Beras (4.0 kg) - digantikan dengan kombinasi lebih optimal

- b. Kertas (3.0 kg) - tidak ada ruang tersisa
- c. Mesin Kecil (5.5 kg) - tidak ada ruang tersisa
- d. Plastik Industri (2.0 kg) - tidak ada ruang tersisa

Jika salah satu item tidak terpilih dimasukkan, harus ada item terpilih yang dikeluarkan, sehingga total berat akan $< 25.0 \text{ kg}$.

B. Apakah Ada Kombinasi Lebih Baik? Mengapa?

TIDAK ADA kombinasi yang lebih baik.

Bukti:

1. Maksimal Teoretis = 25 kg
 - a. Kapasitas = 25 kg
 - b. Tidak mungkin memuat $> 25 \text{ kg}$
 - c. Solusi DP mencapai 25 kg = optimal (100% utilisasi)
2. DP Menjamin Optimalitas

Dynamic Programming memiliki properti Optimal Substructure:

- a. Solusi optimal dibangun dari solusi optimal submasalah
 - b. Untuk setiap $dp[i][w]$, nilai yang disimpan adalah MAKSIMAL yang bisa dicapai
 - c. Dengan i item pertama dan kapasitas w
3. Exhaustive Search (Tersirat)

Tabel DP secara implisit meng-explore semua $2^{10} = 1,024$ kombinasi:

- a. Setiap $dp[i][w]$ merepresentasikan solusi terbaik dari semua kemungkinan kombinasi
 - b. Dengan i item pertama dan kapasitas w
 - c. Tidak ada kombinasi yang terlewat
4. Verifikasi Brute Force

Jika dicek semua kombinasi 10 item:

- a. Total kemungkinan: $2^{10} = 1,024$ kombinasi
- b. Kombinasi dengan 6 item terpilih: $C(10,6) = 210$ kombinasi
- c. Tidak ada satupun yang menghasilkan total $> 25 \text{ kg}$ dengan constraint $\leq 25 \text{ kg}$
- d. Hanya kombinasi yang dipilih DP yang mencapai 25 kg

C. Bandingkan dengan Brute Force Jika Diterapkan

1. Tabel Perbandingan

Aspek	Dynamic Programming	Brute Force
Kompleksitas Waktu	$O(n \times W) = O(10 \times 25) = 250$ operasi	$O(2^n) = 2^{10} = 1,024$ kombinasi
Kompleksitas Ruang	$O(n \times W) = 250$ entries	$O(n) = 10$ (stack depth)
Hasil	Optimal	Optimal
Efisiensi	Sangat Efisien	Tidak Efisien
Perhitungan Berulang	Tidak ada (memoization)	Banyak (overlapping subproblems)

2. Analisis Speedup

a. Untuk n=10 item dengan W=25

- DP: ~250 operasi
- Brute Force: 1,024 kombinasi
- Speedup: ~4x lebih cepat

b. Untuk n=20 item dengan W=25:

- DP: ~500 operasi
- Brute Force: 1,048,576 kombinasi
- Speedup: ~2,000x lebih cepat

c. Untuk n=30 item dengan W=25:

- DP: ~750 operasi
- Brute Force: 1,073,741,824 kombinasi
- Speedup: ~1,400,000x lebih cepat

3. Kesimpulan Perbandingan

Dynamic Programming JAUH LEBIH EFISIEN dibanding Brute Force, terutama untuk jumlah item yang besar. Meskipun keduanya memberikan hasil optimal, DP mencapainya dengan cara yang lebih smart dengan menghindari perhitungan berulang melalui memoization.

D. Kesimpulan Akhir

1. Algoritma DP berhasil menemukan solusi optimal: 25.0 kg (utilisasi 100%)
2. Kombinasi 6 item (Semen, Baja Ringan, Kayu Olahan, Pupuk, Gula, Minyak Goreng) adalah OPTIMAL dan tidak ada kombinasi lebih baik
3. DP lebih efisien dibanding Brute Force dengan kompleksitas $O(n \times W)$ vs $O(2^n)$
4. Untuk kasus ini:
 - a. DP: 250 operasi
 - b. Brute Force: 1,024 kombinasi
 - c. DP ~4x lebih cepat
5. Keunggulan DP semakin signifikan seiring bertambahnya jumlah item

E. Link GITHUB

<https://github.com/FathyahShabiraAkmalTazkia02/knapsack-dynamic-programming.git>