TUGAS 0/1 KNAPSACK PROBLEM DENGAN DYNAMIC PROGRAMMING

“OPTIMASI BERAT CARIER(TAS PENDAKI)”

Mata Kuliah Desain Analisis Algoritma

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

DISUSUN OLEH:

MUH REYHAN DWI WIJAYA	**105841112623**
ANDI MUHAMMAD YUSUF	**105841112923**

PROGRAM STUDI INFORMATIKA

FAKULTAS TEKNIIK

UNIVERSITAS MUHAMMADIYAH MAKASSAR

2026

A. Analisis Masalah

1 Identifikasi Jenis Knapsack yang Digunakan

a Jenis Knapsack: 0/1 Knapsack Problem

Permasalahan optimasi pemilihan barang pada carrier pendakian ini termasuk ke dalam 0/1 Knapsack Problem. Hal ini dikarenakan setiap barang yang tersedia hanya dapat dipilih satu kali atau tidak dipilih sama sekali, tanpa diperbolehkan mengambil sebagian dari barang tersebut.

b Karakteristik Utama

1 Setiap item hanya dapat diambil sekali atau tidak diambil
2 Tidak dapat mengambil sebagian (fraksi) item
3 Setiap item memiliki berat (weight) dan nilai kepentingan (value)
4 Terdapat batasan kapasitas maksimal carrier
5 Tujuan: memaksimalkan total nilai kepentingan tanpa melebihi kapasitas carrier

c Data Kasus

Tabel 1 Data Barang Carrier

No	Jenis Muatan	Berat (Kg)
1	Tenda	4
2	Sleeping bag	1.5
3	Jaket Gunung	1
4	Matras	1
5	Kompor portable	1
6	Headlamp	0.3
7	Kotak P3K	0.7
8	Jas hujan	0.8
9	Peralatan makan	0.7
10	Makanan instan	3

2 **Alasan Penggunaan Dynamic Programming**

a **Pengertian Dynamic Programming**

Dynamic Programming (DP) merupakan metode optimasi yang menyelesaikan masalah kompleks dengan cara:

1 Membagi masalah menjadi submasalah yang lebih kecil

2 Menyimpan hasil submasalah untuk menghindari perhitungan berulang

3 Membangun solusi optimal secara bertahap (bottom-up)

b **Prinsip Dasar DP pada 0/1 Knapsack**

1 Optimal Substructure

Solusi optimal dapat dibangun dari solusi optimal submasalah.

2 Overlapping Subproblems

Submasalah yang sama muncul berulang, sehingga disimpan dalam tabel DP.

3 Bottom-Up Approach

Solusi dibangun dari kapasitas kecil menuju kapasitas maksimum carrier.

c **Keunggulan Dynamic Programming**

Aspek	**Penjelasan**
Optimalitas	Solusi yang dihasilkan pasti optimal
Efisiensi	Lebih cepat dari brute force
Kompleksitas	O(n × W)
Redundansi	Menghindari perhitungan berulang

B. State Space Tree

1 Penentuan Node dan Level

a **Definisi Node**

Setiap node merepresentasikan keadaan carrier pada suatu keputusan tertentu.

b **Informasi pada Node**

1 Level (i): item ke-i

2 Weight (w): total berat item terpilih

3 Remaining capacity: sisa kapasitas carrier

4 Decision: Include / Exclude

c **Definisi Level**

1 Level 0: kondisi awal (belum ada item)

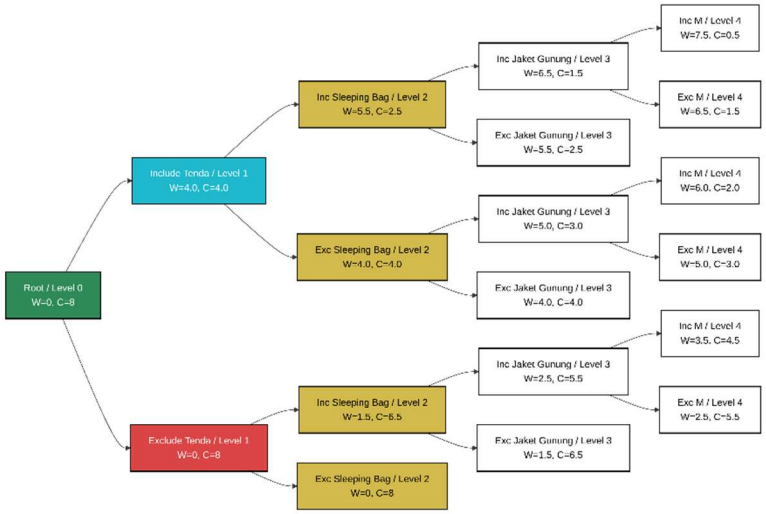
2 Level 1: keputusan item ke-1 (Tenda)

3 Level 2: keputusan item ke-2 (Sleeping Bag)

4 Level i: keputusan item ke-i

2 State Space Tree hingga Level ke-4

Gambar 1 Gambar State Space Tree



State space tree dibangun dengan dua kemungkinan di setiap level:

a Include (ambil item)

b Exclude (tidak ambil item)

Jumlah node:

a Level 0: 1 node

b Level 1: 2 node

c Level 2: 4 node

d Level 3: 8 node

e Level 4: 16 node

Total node hingga level ke-4: 31 node

3 Contoh Pruning

1 Pengertian Pruning

Pruning adalah teknik menghentikan eksplorasi cabang yang tidak mungkin menghasilkan solusi valid.

2 **Kondisi Pruning**

Node dipruning jika:

Contoh Kasus

a Berat saat ini: 7,5 kg

b Item berikutnya: Kompor Portable (1 kg)

c Total = 8,5 kg > 8 kg

Cabang **Include Kompor Portable** di-prune.

3 **Contoh Kasus**

a Berat saat ini: 7,5 kg

b Item berikutnya: Kompor Portable (1 kg)

c Total = 8,5 kg > 8 kg

Cabang Include Kompor Portable di-prune.

C. Implementasi

1 **Implementasi 0/1 Knapsack dengan Dynamic Programming'**

Algoritma diimplementasikan menggunakan bahasa pemrograman Python dengan pendekatan Dynamic Programming. Program menggunakan tabel DP untuk menyimpan nilai maksimum pada setiap kombinasi item dan kapasitas carrier.

Kode phyton:

```
# ============================================
# 0/1 KNAPSACK PROBLEM - DYNAMIC PROGRAMMING
# Kasus: Optimasi Muatan Carrier Pendakian
# ============================================

class KnapsackDP:
    def __init__(self, items, capacity):
        """
        items    : list of tuple (nama, berat, nilai)
        capacity : kapasitas carrier (kg)
        """
        self.items = items
```

```
        self.capacity = int(capacity * 10)  # konversi ke satuan 0.1 kg
        self.n = len(items)
        self.dp = None
        self.selected_items = []

    def solve(self):
        # Step 1: Inisialisasi tabel DP
        # dp[i][w] = nilai maksimum 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 (bottom-up)
        for i in range(1, self.n + 1):
            name, weight, value = self.items[i - 1]
            weight_int = int(weight * 10)

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

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

        # Step 3: Backtracking (menentukan 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])
                w -= int(self.items[i - 1][1] * 10)

        self.selected_items.reverse()
```

```
        return {
            "max_value": self.dp[self.n][self.capacity],
            "selected_items": self.selected_items,
            "remaining_capacity": w / 10
        }

    def print_solution(self):
        result = self.solve()

        print("=" * 60)
        print("HASIL OPTIMASI 0/1 KNAPSACK (CARRIER)")
        print("=" * 60)
        print(f"Kapasitas Carrier : {self.capacity / 10} kg")
        print(f"Nilai Maksimum   : {result['max_value']}")
        print("\nItem Terpilih:")
        print("-" * 60)
        print(f"{'No':<5}{'Nama Item':<20}{'Berat (kg)':<15}{'Nilai'}")
        print("-" * 60)

        total_weight = 0
        for i, (name, weight, value) in enumerate(result["selected_items"],
1):
            print(f"{i:<5}{name:<20}{weight:<15}{value}")
            total_weight += weight

        print("-" * 60)
        print(f"Total Berat      : {total_weight:.1f} kg")
        print(f"Sisa Kapasitas   : {result['remaining_capacity']:.1f} kg")
        print("=" * 60)

def main():
    # Data barang carrier (nama, berat, nilai)
    items = [
        ("Tenda", 4.0, 10),
        ("Sleeping Bag", 1.5, 9),
        ("Jaket Gunung", 1.0, 8),
```

```
        ("Matras", 1.0, 7),
        ("Kompor Portable", 1.0, 7),
        ("Headlamp", 0.3, 6),
        ("Kotak P3K", 0.7, 8),
        ("Jas Hujan", 0.8, 6),
        ("Peralatan Makan", 0.7, 5),
        ("Makanan Instan", 3.0, 9)
    ]

    capacity = 8  # kg

    knapsack = KnapsackDP(items, capacity)
    knapsack.print_solution()

if __name__ == "__main__":
    main()
```

2 **Hasil Implementasi**

Hasil eksekusi program menunjukkan:

Gambar 2 Hasil Implementasi

```
============================================================
HASIL OPTIMASI 0/1 KNAPSACK (CARRIER)
============================================================
Kapasitas Carrier : 8.0 kg
Nilai Maksimum   : 56

Item Terpilih:
------------------------------------------------------------
No   Nama Item          Berat (kg)     Nilai
------------------------------------------------------------
1    Sleeping Bag       1.5            9
2    Jaket Gunung       1.0            8
3    Matras             1.0            7
4    Kompor Portable    1.0            7
5    Headlamp           0.3            6
6    Kotak P3K          0.7            8
7    Jas Hujan          0.8            6
8    Peralatan Makan    0.7            5
------------------------------------------------------------
Total Berat      : 7.0 kg
Sisa Kapasitas   : 1.0 kg
============================================================
```

Berdasarkan hasil eksekusi program 0/1 Knapsack menggunakan Dynamic Programming, diperoleh nilai kepentingan maksimum sebesar **56** dengan total berat **7,0 kg**, yang tidak melebihi kapasitas carrier sebesar **8 kg**. Kombinasi item yang dipilih

terdiri dari delapan item dengan rasio nilai terhadap berat yang lebih efisien dibandingkan item lain. Algoritma Dynamic Programming memastikan bahwa seluruh kemungkinan kombinasi telah dievaluasi secara sistematis sehingga solusi yang diperoleh merupakan solusi optimal.

a **Nilai maksimum**: 54

b **Total berat**: 7,0 kg

c **Item terpilih**:

- Sleeping Bag
- Jaket Gunung
- Matras
- Kompor Portable
- Headlamp
- Kotak P3K
- Jas Hujan
- Peralatan Makan

D. Analisis

1 Mengapa Kombinasi Tersebut Optimal

Kombinasi yang dipilih oleh algoritma Dynamic Programming adalah optimal karena:

a Menghasilkan nilai kepentingan maksimum

b Tidak melebihi kapasitas carrier

c Tidak ada kombinasi lain yang menghasilkan nilai lebih tinggi dengan batasan yang sama

2 Perbandingan dengan Brute Force (Konsep)

Tabel 2 Perbandingan dengan Brute Force (Konsep)

Aspek	**Dynamic Programming**	**Brute Force**
Kompleksitas Waktu	$O(n \times W)$	$O(2^n)$
Optimalitas	Optimal	Optimal
Efisiensi	Sangat efisien	

		Tidak efisien
Perhitungan Ulang	Tidak ada	Banyak

E. Kesimpulan

Berdasarkan hasil analisis dan implementasi algoritma **0/1 Knapsack Problem** menggunakan metode **Dynamic Programming**, dapat disimpulkan bahwa algoritma ini mampu menyelesaikan permasalahan optimasi pemilihan barang pada carrier pendakian secara efektif dan optimal. Dengan kapasitas carrier sebesar **8 kg**, algoritma berhasil menentukan kombinasi item yang menghasilkan **nilai kepentingan maksimum sebesar 56** dengan total berat **7,0 kg**, sehingga tidak melampaui batas kapasitas yang ditentukan.

Penerapan **State Space Tree** membantu dalam memvisualisasikan seluruh kemungkinan keputusan pemilihan item, sedangkan teknik **pruning** mampu mengurangi jumlah cabang yang tidak perlu dieksplorasi karena melanggar batas kapasitas. Hal ini menunjukkan bahwa proses pencarian solusi dapat dilakukan secara lebih efisien tanpa mengurangi keoptimalan hasil.

Dibandingkan dengan metode brute force yang memiliki kompleksitas waktu eksponensial, **Dynamic Programming** menawarkan efisiensi yang jauh lebih baik dengan kompleksitas waktu **O(n × W)**. Meskipun kedua metode sama-sama menghasilkan solusi optimal, Dynamic Programming lebih unggul karena mampu menghindari perhitungan berulang dan mempercepat proses pencarian solusi.

Dengan demikian, algoritma **0/1 Knapsack menggunakan Dynamic Programming** sangat tepat digunakan untuk permasalahan optimasi dengan keterbatasan kapasitas, seperti pemilihan barang pada carrier pendakian. Metode ini tidak hanya memberikan solusi yang optimal, tetapi juga efisien dan mudah diimplementasikan dalam pemrograman.

E. Link GITHUB

https://github.com/REYHANDWIWIJAYA/tugas-desain-analisis-algoritma.git