Optimasi 0-1 Knapsack Dengan Dynamic Programming

Ubaidillah Ariq Prathama - 13520085
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13520085@std.stei.itb.ac.id

***Abstract*—Knapsack merupakan sebuah permasalahan klasik yang merupakan generalisasi dari masalah-masalah lainnya. S. Terdapat beberapa cara untuk mengimplementasikan knapsack diantaranya adalah brute force dan dynamic programming. Tentunya program yang dibuat harus efektif, terutama untuk kasus yang besar. Oleh karena itu, algoritma ini perlu dianalisis kompleksitasnya.**

***Keywords*—Knapsack, Kompleksitas Waktu, Kompleksitas Ruang, Dynamic Programming**

I. PENDAHULUAN

Knapsack adalah permasalahan klasik yang dapat membantu menyelesaikan permasalahan lain. Persoalan Knapsack merupakan sebuah persoalan optimasi. Permasalahannya didefinisikan dengan diberikan suatu himpunan barang, setiap barang dalam himpunan tersebut memiliki keuntungan dan bobot, Kemudian ditentukan kombinasi barang yang dapat dimasukkan ke dalam kontainer (knapsack) sehingga menghasilkan nilai maksimum dan bobotnya tidak melebihi kapasitas container. Setiap permasalahan pasti memiliki berbagai macam solusi dan harus dipilih solusi yang paling tepat.

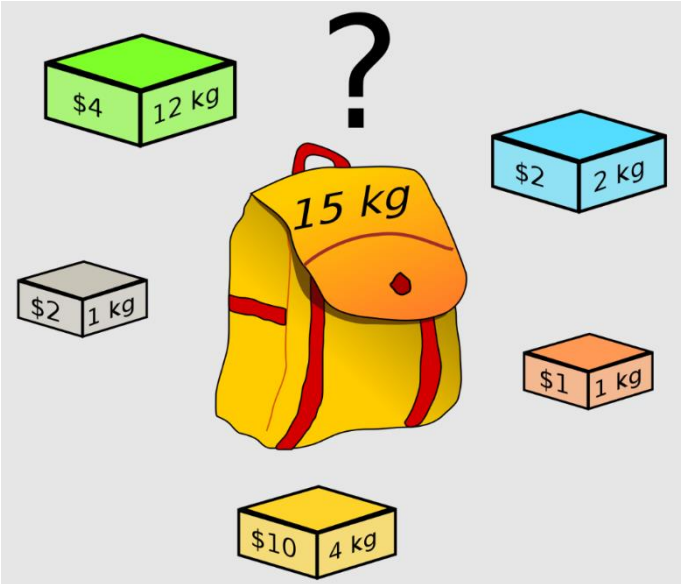
Dapat dilihat bahwa banyak sekali aplikasi dari knapsack. Oleh sebab itu, diperlukan algoritma knapsack yang optimal. Terdapat banyak cara mengoptimasi sebuah program, tetapi yang paling signifikan adalah kompleksitas waktu dan ruang. Tentunya permasalahan yang dapat diselesaikan dengan knapsack ini bisa saja berukuran besar. Jika program tidak efisien bisa saja melewati constraint memori dan waktu.

Terdapat dua pendekaran yang umum digunakan untuk menyelesaikan permasalahan knapsack, yaitu strategi brute force dan dynamic programming. Kedua algoritma ini sebenarnya sederhana dan akan dibahas kemudian. Akan tetapi, implementasi kedua algoritma ini bisa dilakukan dengan berbagai cara dan struktur data yang berbeda. Perbedaan ini menyebabkan adanya perbedaan kompleksitas yang harus kita analisis mana yang lebih efektif.

II. TEORI DASAR

A. Knapsack

Persoalan Knapsack merupakan sebuah persoalan optimasi. Permasalahannya didefinisikan dengan diberikan suatu himpunan barang, setiap barang dalam himpunan tersebut memiliki keuntungan dan bobot, Kemudian ditentukan kombinasi barang yang dapat dimasukkan ke dalam kontainer (knapsack) sehingga menghasilkan nilai maksimum dan bobotnya tidak melebihi kapasitas container.



Persoalan knapsack diselesaikan dengan maksimalisasi nilai serta Batasan pada bobot yang dapat diambil. Secara matematis dapat dituliskan sebagai berikut:

$$maksimasi : \sum_{i=1}^{n} v_i x_i$$

$$batasan : \sum_{i=1}^{n} w_i x_i \leq W$$

Dengan v_i adalah profit dari barang ke-i, w_i adalah bobot dari barang ke-i, x_i adalah jumlah barang yang diambil pada tahap ke-i, dan W merupakan kapasitas dari kontainer.

Terdapat beberapa varian dari persoalan knapsack, yaitu :

1. Persoalan 0/1 Knapsack
   Pada persoalan knapsack jenis ini, jumlah barang maksimum yang dapat diambil untuk setiap jenis barang yang ada di dalam himpunan adalah satu.

2. Persoalan Knapsack Terbatas (bounded knapsack problem)
   Pada persoalan knapsack jenis ini, jumlah barang maksimum yang dapat diambil untuk setiap jenis barang yang ada adalah sebesar c. Dengan c adalah sebuah bilangan bulat yang ditentukan oleh pengguna.
3. Persoalan Knapsack Tanpa Batas (unbounded knapsack problem)
   Pada persoalan knapsack jenis ini, tidak terdapat jumlah maksimum yang dapat diambil untuk setiap jenis barang yang terdapat di himpunan.

Perbedaan dari ketiga varian tersebut dapat dituliskan secara matematis sebagai berikut:

$$0/1\ knapsack: 0 \leq x_i \leq 1$$
$$bounded\ knapsack\ :\ 0 \leq x_i \leq c, c \in R$$
$$unbounded\ knapsack\ :\ 0 \leq x_i$$

B. Brute Force

Algoritma brute force adalah algoritma yang polos, terang-terang, sudah jelas atau straightforward. Algoritma ini mencoba segala kemungkinan solusi yang mungkin dari suatu permasalahan, membandingkan setiap solusi yang ada dan mendapatkan jawabannya dari hasil perbandingan itu.

Contoh sederhana pemakaian brute force didalam perhitungan pangkat suatu bilangan misalnya,

$$a^n = a \times a \times \cdots \times a$$

Algoritma brute force akan membutuhkan n-1 kali proses perkalian dalam menyelesaikan masalah ini.

Contoh lain adalah penggunaanya didalam mencari suatu nilai terbesar didalam suatu array. Untuk mencari nilai terbesar, algoritma brute force akan melakukan pengecekan satu persatu disetiap elemen array yang ada dan untuk setiap kali pengecekan dilakukan pembandingan nilai terbesar, hingga seluruh elemen array dicek. Brute force merupakan algoritma yang apa adanya dan tidak membutuhkan hal-hal khusus untuk mengerti dan mempelajarinnya.

Keuntungan dari penggunaan brute force ini adalah semua masalah dapat dipecahkan dengan algoritma ini, algoritma yang digunakan dan ditulis sangat mudah dimengerti dan simpel, serta berguna untuk memecahkan masalah yang kecil. Kekurangan algoritma ini di bandingkan dengan algoritma lain adalah kebutuhan waktu pemrosesan yang akan semakin bertambah seiring dengan bertambahnya data yang diproses atau dapat dikatakan bahwa kompleksitas algoritma brute force sangat tinggi.

C. Exhaustive Search

Algoritma brute force adalah algoritma sederhana yang memandang persoalan algoritmik sesuai dengan deskripsi persoalan tersebut. Algoritma exhaustive search merupakan subset dari algoritma brute force yang menyelesaikan persoalan pencarian dan optimasi. Sebuah algoritma exhaustive search akan melakukan enumerasi semua kandidat solusi dari persoalan, dan dalam setiap enumerasi, memeriksa apakah kandidat tersebut benar-benar merupakan solusi dari persoalan tersebut (untuk persoalan pencarian, apakah kandidat memenuhi constraint, dan untuk persoalan optimasi, apakah kandidat memenuhi constraint dan memiliki nilai evaluasi maksimum atau minimum).

Karakteristik dari algoritma exhaustive search adalah sederhana dan mudah diimplementasi, sehingga algoritma pertama yang muncul untuk menyelesaikan suatu persoalan pada umumnya adalah algoritma exhaustive search.

Akan tetapi, algoritma exhaustive search memiliki kekurangan. yaitu masalah efisiensi. Untuk mendapatkan solusi, algoritma exhaustive search harus melakukan enumerasi untuk setiap anggota dari himpunan kandidat solusi dari persoalan, dan ukuran himpunan kandidat solusi berkontribusi terhadap kompleksitas algoritma exhaustive search. Sebagai contoh, untuk persoalan mencari pasangan bilangan dengan jumlah terkecil dari sebuah himpunan bilangan berukuran n, maka ukuran himpunan kandidat solusi adalah $\frac{n(n-1)}{2}$ dan kompleksitas algoritma exhaustive search untuk persoalan ini adalah $O(n)$. Jika nilai n sangat 2 besar, maka ukuran himpunan kandidat solusi akan jauh lebih besar, terutama jika ukuran himpunan kandidat solusi meningkat terhadap n secara non-polinomial (Contohnya pada Travelling Salesperson Problem dengan ukuran graf n, ukuran himpunan kandidat solusi adalah $n!$, dengan anggotanya adalah semua kemungkinan permutasi dari semua simpul di graf).

Oleh karena itu, algoritma exhaustive search digunakan untuk persoalan dengan ukuran input yang kecil, atau untuk persoalan yang belum ditemukan algoritma yang lebih baik.

Adapun metode exhaustive search memiliki langkah-langkah utama sebagai berikut:

1. Enumerasi setiap solusi dengan sistematis.
2. Evaluasi tiap solusi satu per satu.
3. Umumkan solusi terbaik.

D. Dynamic Programming

Program Dinamis (dynamic programming) merupakan salah satu strategi atau metode untuk menyelesaikan suatu permasalahan. Istilah program dinamis muncul karena perhitungan solusi menggunakan tabel. Kata “program” pada program dinamis tidak berhubungan dengan source code. Yang dimaksud dengan “program” dalam program dinamis adalah perencanaan. Strategi atau metode dalam memecahkan masalah dengan menggunakan program dinamis adalah menguraikan solusi menjadi sekumpulan tahap (stage) sedemikian sehingga solusi dari permasalahan tersebut dapat dipandang dari serangkaian keputusan yang saling berkaitan. Penyelesaian suatu persoalan dengan program dinamis memiliki kemiripan dengan algoritma greedy, hanya saja pada algoritma greedy hanya terdapat satu solusi, sementara pada program dinamis dapat dihasilkan lebih dari satu kemungkinan solusi.

Karakteristik dari penyelesaian persoalan dengan menggunakan program dinamis adalah :

1. Terdapat sejumlah berhingga pilihan yang mungkin.
2. Solusi pada setiap tahap dibangun dari hasil solusi tahap sebelumnya.
3. Digunakan persyaratan optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap.

Pada program dinamis, rangkaian keputusan yang optimal dibuat dengan menggunakan prinsip optimalitas. Prinsip optimalitas yang dimaksud adalah jika solusi total merupakan solusi optimal, maka bagian solusi sampai tahap ke-k juga optimal. Dengan prinsip optimalitas dapat dijamin bahwa penagmbilan keputusan pada suatu tahap adalah keputusan yang benar untuk tahap- tahap selanjutnya. Oleh karena itu, pada program dinamis, tidak perlu kembali ke tahap awal karena digunakan hasil optimal dari tahap sebelumnya.

Karakteristik persoalan program dinamis adalah :

1. Persoalan dapat dibagi menjadi beberapa tahap (stage), yang dalam hal ini setiap tahap hanya diambil satu keputusan.
2. Masing- masing tahap terdiri dari sejumlah status (state) yang berhubungan dengan tahap tersebut. Status yang dimaksud adalah berbagai macam kemungkinan masukan yang ada pada tahap tersebut. Masing- masing tahap ini dapat digambarkan dengan menggunakan graf multitahap (multistage graph), yang dalam hal ini tiap simpul pada graf tersebut merepresentasikan status, sedangkan busur pada graf tersebut menyatakan tahap.
3. Hasil dari keputusan yang diambil pada setiap tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap berikutnya.
4. Ongkos (cost) pada suatu tahap meningkat secara teratur (steadily) dengan bertambahnya jumlah tahapan.
5. Ongkos pada suatu tahap bergantung pada ongkos pada tahap yang sudah berjalan sebelumnya dan ongkos pada tahap tersebut.
6. Keputusan terbaik pada suatu tahap bersifat independen terhadap keputusan yang dilakukan pada tahap sebelumnya.
7. Adanya hubungan rekursif yang mengidentifikasikan keputusan terbaik untuk setiap status pada tahap k sehingga dapat diambil keputusan terbaik untuk setiap status pada tahap k + 1.
8. Prinsip optimalitas berlaku pada persoalan tersebut

Terdapat dua pendekatan yang digunakan dalam program dinamis, yaitu maju (forward dan up-down) dan mundur (backward atau bottom-up). Misalkan $x_1, x_2, \cdots, x_n$ yang menyatakan peubah (variable) keputusan yang harus dibuat masing- masing pada tahap 1, 2, 3, …, n. Untuk program dinamis mundur, program dinamis akan bergerak mulai dari tahap 1, 2, 3, dan seterusnya sampai tahap n. Runutan peubah (variable) keputusan adalah $x_1, x_2, \cdots, x_n$. Hal sebaliknya berlaku untuk program dinamis mundur. Program dinamis mundur akan bergerak mulai dari tahap n, n – 1, n – 2, dan seterusnya sampai tahap 1. Runutan peubah keputusan pada program dinamis mundur adalah $x_n, x_{n-1}, \cdots, x_1$. Pada program dinamis maju, prinsip optimalitasnya adalah ongkos pada tahap k + 1 = (ongkos yang dihasilkan pada tahap k) + (ongkos dari tahap k ke tahap k + 1). Sementara prinsip optimalitas pada program dinamis mundur adalah ongkos pada tahap k = (ongkos yang dihasilkan pada tahap k + 1) + (ongkos dari tahap k+1 ke tahap k). Secara umum, terdapat empat langkah yang dilakukan dalam mengembangkan algoritma program dinamis.

Langkah- langkah tersebut adalah :

1. Karakteristikkan struktur solusi optimal.
2. Definisikan secara rekursif nilai solusi optimal.
3. Hitung nilai solusi optimal secara maju atau mundur.
4. Konstruksi solusi optimal

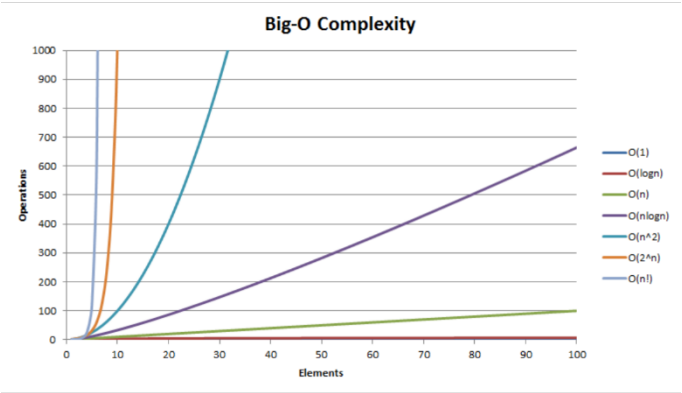
E. Kompleksitas Algoritma

Kompleksitas algoritma dinilai dari beberapa tinjauan. Tinjauan kompleksitas yang mungkin dilakukan terhadap suatu algoritma adalah kompleksitas ruang dan kompleksitas waktu . Kommpleksitas waktu $T(n)$ merupakan pengurukran terhadap jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan n. Kompleksitas ruang $S(n)$ dapat diukur melalui memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan n.

Kompleksitas waktu dihitung berdasarkan berapa kali suatu operasi dilaksanakan dalam sebuah algoritma sebagai fungsi dengan ukuran masukan n. Operasi yang diperhitungkan dalam kompleksitas waktu suatu algoritma hanya operasi-operasi khas yang mendasari algoritma tersebut. Adapun operasi-operasi tersebut adalah operasi aritmatika sederhana dan operasi perbadingan.

Kompleksitas waktu digolongkan menjadi 3 macam, yaitu $T_{max}(n)$ atau kompleksitas waktu untuk kasus terburuk, $T_{avg}(n)$ atau kompleksitas waktu untuk kasus rata-rata, dan $T_{min}(n)$ atau kompleksitas waktu untuk kasus terbaik. Nilai masing-masing besaran tersebut sangat beragam untuk tiap-tiap algoritma. Sebagai contoh, pada algoritma sequential search, $T_{min}(n) = 1$, $T_{max}(n) = n$, dan $T_{avg}(n) = 0.5(n+1)$.

Notasi untuk kompleksitas waktu asimptotik adalah “O besar” (Big-O). Dalam hal ini “$T(n) = O(f(n))$” bermakna bahwa orde maksimum $T(n)$ adalah $f(n)$, atau dengan ekspresi lain dapat dinyatakan sebagai $T(n) \leq C(f(n))$ untuk $n \geq n_0$. Dalam hal ini $f(n)$ adalah batas lebih atas dari $T(n)$ untuk n yang besar. Urutan spektrum kompleksitas waktu algoritma dari kecil ke besar adalah sebagai berikut.



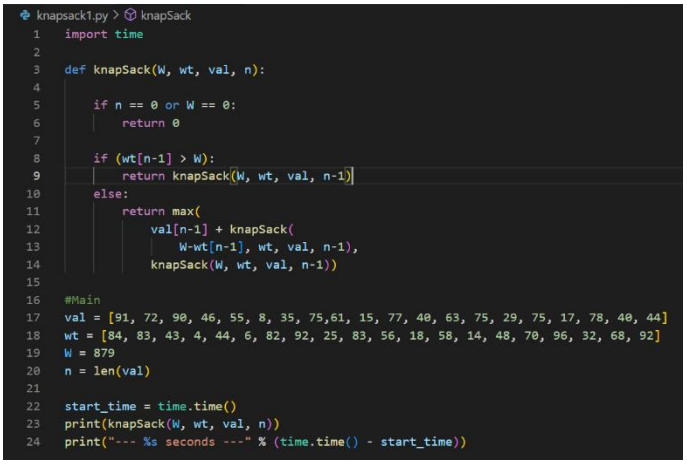
No.	Kelompok Algoritma	Nama
1.	$O(1)$	Konstan
2.	$O(\log n)$	Logaritmik
3.	$O(n)$	Linear
4.	$O(n \log n)$	N log N
5.	$O(n^2)$	Kuadratik
6.	$O(n^3)$	Kubik
7.	$O(a^n)$	Eksponensial
8.	$O(n!)$	Faktorial

Tabel 1. Kelompok Kompleksitas

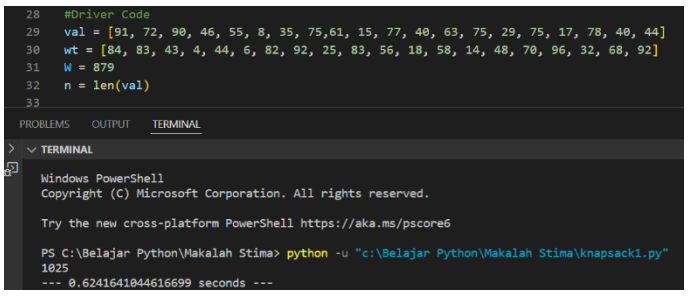
III. ALGORITMA BRUTE FORCE

Persoalan 0-1 Knapsack dapat diselesaikan dengan brute force dengan cara exhaustive search menggunakan rekursif. Solusinya adalah dengan mengecek weight dan value semua himpunan bagian dari objek yang dapat dipilih. Cukup perhatikan himpunan bagian yang memiliki total weight kurang dari kapasitas. Setiap objek memiliki dua kemungkinan yaitu masuk ke dalam himpunan solusi atau tidak dimasukkan. Oleh karena itu nilai maksimum dari value bisa didapatkan melalui 2 kemungkinan yaitu:

1. Nilai penjumlahan value dari n-1 objek dengan kapasitas W (objek ke-n tidak dimasukkan himpunan solusi)
2. Nilai penjumlahan value ke-n dan value maksimum dari n-1 objek dengan kapasitas W-weight[n]



Pengujian dilakukan dengan test case:
val = [91, 72, 90, 46, 55, 8, 35, 75,61, 15, 77, 40, 63, 75, 29, 75, 17, 78, 40, 44]
wt = [84, 83, 43, 4, 44, 6, 82, 92, 25, 83, 56, 18, 58, 14, 48, 70, 96, 32, 68, 92]
W = 879
Output :
1025
Waktu Eksekusi : 0.62 detik



Algoritma ini kurang efektif karena terdapat sub-problem yang diselesaikan berulang kali. Hal ini sesuai dengan klasifikasi permasalahan yang cocok diselesaikan dengan dynamic programming. Kompleksitas waktu dari algoritma ini adalah $O(2^n)$ karena mengecek semua himpunan bagian dari n objek. Kompleksitas ruang dari algoritma ini adalah $O(1)$ karena solusi ini menggunakan rekursif dan tidak ada stuktur data tambahan.

IV. ALGORITMA DYNAMIC PROGRAMMING

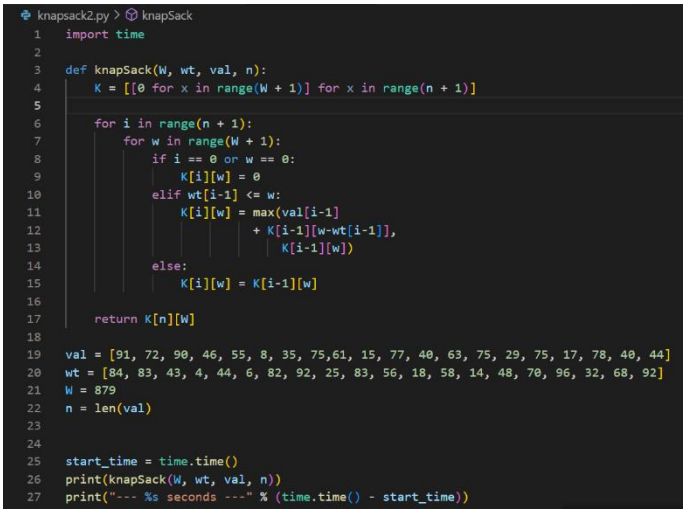
Persoalan 0-1 Knapsack dapat diselesaikan dengan dynamic programming. Seperti permasalahan dynamic programming pada umumnya, subproblem yang berulang dapat dihilangkan dengan membuat array sementara dp[][] dengan cara bottom up.

Tabel dp dapat diimplementasikan dengan kolom berisi semua weight yang mungkin dari 1 hingga kapasitas maksimum dan baris berupa weights yang bisa disimpan. Nilai dari dp[i][j] akan berisi value maksimum dengan weight = j dengan memerhatikan semua value dari 1 sampai i. Terdapat dua kemungkinan untuk weight ke-i, yaitu :

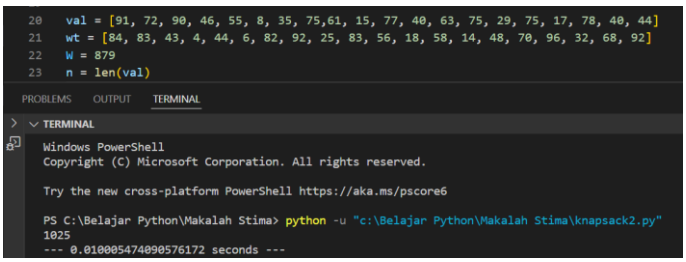
1. Isi weight ke-i pada kolom
2. Tidak mengisi weight ke-i pada kolom

Oleh karena itu, nilai tabel dp dapat diisi dengan nilai maksimum dari dua nilai di bawah ini :

1. dp[i][j] = dp[i-1][w-wt[i-1]] + val[i-1]
2. dp[i][j] = dp[i-1][j]



Pengujian dilakukan dengan test case:
val = [91, 72, 90, 46, 55, 8, 35, 75,61, 15, 77, 40, 63, 75, 29, 75, 17, 78, 40, 44]
wt = [84, 83, 43, 4, 44, 6, 82, 92, 25, 83, 56, 18, 58, 14, 48, 70, 96, 32, 68, 92]
W = 879
Output :
1025
Waktu Eksekusi : 0.01 detik



Algoritma ini cukup efektif karena tidak ada sub-problem yang diselesaikan berulang kali. Kompleksitas waktu dari algoritma ini adalah $O(NW)$ dengan N adalah banyaknya objek dan W adalah kapasitas. Setiap objek akan dicek semua kapasitas yang mungkin dari 1 sampai w sehingga didapat

kompleksitas $O(NW)$. Kompleksitas ruang dari algoritma ini adalah $O(NW)$ karena solusi ini menggunakan array 2 dimensi yang berukuran N kali W.

V. Algoritma Memoisasi pada DP

Persoalan 0-1 Knapsack dapat diselesaikan dengan dynamic programming dengan memoisasi. Metode ini sebenarnya adalah pengembangan dari rekursif dengan menghindari pengulangan subproblem yang sudah dihitung. Strategi solusi ini adalah membuat array 2 dimensi yang menyimpan suatu state (n,w) yang sudah pernah dikomputasi sehingga tidak perlu dikomputasi ulang.

knapsack3.py > ...

```python
val = [91, 72, 90, 46, 55, 8, 35, 75,61, 15, 77, 40, 63, 75, 29, 75, 17, 78, 40, 44]
wt = [84, 83, 43, 4, 44, 6, 82, 92, 25, 83, 56, 18, 58, 14, 48, 70, 96, 32, 68, 92]
W = 879
n = len(val)

t = [[-1 for i in range(W + 1)] for j in range(n + 1)]

def knapsack(wt, val, W, n):

    # base conditions
    if n == 0 or W == 0:
        return 0
    if t[n][W] != -1:
        return t[n][W]

    # choice diagram code
    if wt[n-1] <= W:
        t[n][W] = max(
            val[n-1] + knapsack(
                wt, val, W-wt[n-1], n-1),
            knapsack(wt, val, W, n-1))
        return t[n][W]
    elif wt[n-1] > W:
        t[n][W] = knapsack(wt, val, W, n-1)
        return t[n][W]


start_time = time.time()
print(knapsack(wt, val, W, n))
print("--- %s seconds ---" % (time.time() - start_time))
```

Pengujian dilakukan dengan test case:
val = [91, 72, 90, 46, 55, 8, 35, 75,61, 15, 77, 40, 63, 75, 29, 75, 17, 78, 40, 44]
wt = [84, 83, 43, 4, 44, 6, 82, 92, 25, 83, 56, 18, 58, 14, 48, 70, 96, 32, 68, 92]
W = 879
Output :
1025
Waktu Eksekusi : 0.006 detik

```
val = [91, 72, 90, 46, 55, 8, 35, 75,61, 15, 77, 40, 63, 75, 29, 75, 17, 78, 40, 44]
wt = [84, 83, 43, 4, 44, 6, 82, 92, 25, 83, 56, 18, 58, 14, 48, 70, 96, 32, 68, 92]
W = 879
n = len(val)

PROBLEMS  OUTPUT  TERMINAL
TERMINAL
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Belajar Python\Makalah Stima> python -u "c:\Belajar Python\Makalah Stima\knapsack3.py"
1025
--- 0.005999088287353516 seconds ---
```

Algoritma ini cukup efektif karena tidak ada sub-problem yang diselesaikan berulang kali. Kompleksitas waktu dari algoritma ini adalah $O(NW)$ dengan N adalah banyaknya objek dan W adalah kapasitas. Kompleksitas ini didapatkan karena tidak ada pengulangan. Kompleksitas ruang dari algoritma ini adalah $O(NW)$ karena solusi ini menggunakan array 2 dimensi yang berukuran N kali W.

VI. Optimasi Memori pada DP

Persoalan 0-1 Knapsack dapat diselesaikan dengan dynamic programming. Seperti permasalahan dynamic programming pada umumnya, subproblem yang berulang dapat dihilangkan dengan membuat array sementara dp[] dengan cara bottom up. Perbedaan dengan algoritma pada bagian IV hanyalah pada bagian ini diimplementasikan menggunakan array 1 dimensi.

knapsack4.py > ...

```python
import time

def knapSack(W, wt, val, n):
    dp = [0 for i in range(W+1)]

    for i in range(1, n+1):
        for w in range(W, 0, -1):
            if wt[i-1] <= w:
                dp[w] = max(dp[w], dp[w-wt[i-1]]+val[i-1])

    return dp[W]

val = [91, 72, 90, 46, 55, 8, 35, 75,61, 15, 77, 40, 63, 75, 29, 75, 17, 78, 40, 44]
wt = [84, 83, 43, 4, 44, 6, 82, 92, 25, 83, 56, 18, 58, 14, 48, 70, 96, 32, 68, 92]
W = 879
n = len(val)


start_time = time.time()
print(knapSack(W, wt, val, n))
print("--- %s seconds ---" % (time.time() - start_time))
```

Pengujian dilakukan dengan test case:
val = [91, 72, 90, 46, 55, 8, 35, 75,61, 15, 77, 40, 63, 75, 29, 75, 17, 78, 40, 44]
wt = [84, 83, 43, 4, 44, 6, 82, 92, 25, 83, 56, 18, 58, 14, 48, 70, 96, 32, 68, 92]
W = 879
Output :
1025
Waktu Eksekusi : 0.006 detik

```
val = [91, 72, 90, 46, 55, 8, 35, 75,61, 15, 77, 40, 63, 75, 29, 75, 17, 78, 40, 44]
wt = [84, 83, 43, 4, 44, 6, 82, 92, 25, 83, 56, 18, 58, 14, 48, 70, 96, 32, 68, 92]
W = 879
n = len(val)

PROBLEMS  OUTPUT  TERMINAL
TERMINAL
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Belajar Python\Makalah Stima> python -u "c:\Belajar Python\Makalah Stima\knapsack4.py"
1025
--- 0.005986928939819336 seconds ---
```

Algoritma ini cukup efektif karena tidak ada sub-problem yang diselesaikan berulang kali. Kompleksitas waktu dari algoritma ini adalah $O(NW)$ dengan N adalah banyaknya objek dan W adalah kapasitas. Kompleksitas ini didapatkan karena tidak ada pengulangan. Kompleksitas ruang dari algoritma ini adalah $O(W)$ karena solusi ini menggunakan array 1 dimensi W, bukan array 2 dimensi.

VII. Kesimpulan

Keempat algoritma knapsack ini pada umumnya dapat diimplementasikan pada berbagai persoalan yang ada. Algoritma Brute Force memiliki kompleksitas waktu $O(2^n)$ dan kompleksitas ruang $O(1)$. Penyelesaian Knapsack menggunakan Dynamic Programming cukup membantu secara signifikan karena berhasil mengubah kompleksitas waktu menjadi $O(NW)$ dan kompleksitas ruang menjadi $O(NW)$. Sebenarnya terjadi penurunan efektifitas pada memori, tetapi

tradeoff ini worth it karena penurunan kompleksitas waktu cukup signifikan dari yang sebelumnya eksponensial. Pada algoritma memoisasi pada DP didapat kompleksitas waktu $O(NW)$ dan kompleksitas ruang $O(NW)$. Hal ini sama dengan strategi DP sebelumnya karena pada dasarnya algoritma ini sama dengan synamic programming pada umumnya, hanya memberikan pendekatan yang lebih intuitif. Pada algoritma optimasi memori pada DP didapat kompleksitas waktu $O(NW)$ dan kompleksitas ruang $O(W)$. Perbedannya terdapat pada array yang digunakan adalah array 1 dimensi.

Keempat algoritma ini menggunakan kasus uji yang sama dengan jumlah objek yang dipilih terdapat sebanyak 20. Keempat algoritma ini juga memberikan output yang sama yaitu 1025 yang mana benar. Maka dapat disimpulkan, keempat algoritma ini dapat menyelesaikan persoalan Knapsack. Output yang berbeda hanyalah waktu eksekusi. Algoritma Brute Force memiliki waktu eksekusi 0.62 detik yang mana sesuai dengan klaim kita mengenai kompleksitas waktu. Terdapat sedikit perbedaan pada algoritma 2 dengan algoritma 3 dan 4. Secara teoritis seharusnya ketiga algoritma ini memiliki waktu eksekusi yang sama. Akan tetapi, algoritma kedua memiliki waktu eksekusi 0.1 detik sedangkan dua algoritma lainnya memiliki waktu eksekusi 0.06 detik. Perbedaan ini mungkin saja dipengaruhi oleh performa mesin dikarenakan perbedaannya tidak terlalu signifikan. Untuk memori, tidak terlalu terlihat perbedaannya karena test case yang dipakai cukup kecil.

Asumsi yang umum digunakan adalah komputer dapat melakukan 10^8 komputasi dalam 1 detik. Jika algoritma brute force digunakan untuk menyelesaikan knapsack yang berukuran 30 objek bisa memakan waktu hingga 10 detik, jika terdapat 50 objek bisa memakan waktu berhari-hari. Hal ini jelas tidak efektif. Dynamic programming membantu kita untuk membuat program lebih efisien. Jika terdapat 50 objek dan kapasitas maksimum 20000 bahkan secara teoritis hanya memerlukan waktu 0.01 detik. Walaupun mengorbankan sedikit memori tapi hal ini worth untuk dicoba. Dengan asumsi integer memiliki ukuran 4 bytes. Jika terdapat 50 objek dan 20000, akan memakan memori sebanyak 4MB yang dapat dibilang kecil untuk ukuran memori sekarang. Bahkan jika digunakan optimasi memori hanya diperlukan 20KB memori.

VIII. Ucapan Terima Kasih

Pertama penulis ingin mengucapkan puji syukur kepada Tuhan Yang Maha Esa karena dengan rahmat dan karunai-Nya penulis dapat menyelesaikan makalah dengan judul “Optimasi 0-1 Knapsack Dengan Dynamic Programming” ini dengan baik. Penulis juga berterima kasih kepada dosen mata kuliah IF2211 Strategi Algoritma, Dr. Ir. Rinaldi Munir, M.T., Dr. Masayu Leylia Khodra, S.T, M.T., dan Dr. Nur Ulfa Maulidevi, S.T, M.Sc. atas bimbingan beliau selama ini dalam mengajar dan memberikan ilmu pada mata kuliah matematika diskrit sehingga penulis mampu membuat makalah ini. Penulis juga berterima kasih kepada rekan-rekan yang telah memberikan semangat dan dorongan kepada penulis.

Referensi

[1] R. Munir, *Matematika Diskrit*. Bandung : Penerbit Informatika, Palasari

[2] W. Gozali & A.F.Aji, Pemrograman Kompetitif Dasar, Jakarta : Nulis Buku Jendela Dunia, 2015.

[3] https://www.ics.uci.edu/~eppstein/161/960206.html , diakses pada 21 Mei 2022 pukul 21.37

[4] https://www.javatpoint.com/0-1-knapsack-problem, diakses pada 21 Mei 2022, pukul 21.37

[5] https://www.programiz.com/dsa/dynamic-programming, diakses pada 21 Mei 2022 pukul 21.37

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/stima21-22.htm , diakses pada 21 Mei 2022, pukul 22.00

[7] Silde Perkuliahan Strategi Algoritma, Teknik Informatika, Institut Teknologi Bandung

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 Mei 2022

Ubaidillah Ariq Prathama - 13520085