Perbandingan Algoritma *Greedy* dan *Dynamic Programming* pada *0-1 Knapsack Problem*

Wisnu Aditya Samiadji 13519093
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
1359093@std.stei.itb.ac.id

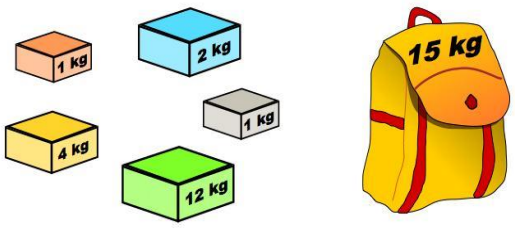
Abstrak*—Algoritma pastilah bukan hal yang asing dalam kehidupan manusia, terutama bagi mereka yang menekuni bidang komputasi. Algoritma digunakan sebagai solusi dari berbagai permasalahan Dalam bidang komputasi, ada berbagai jenis cara dalam implementasi algoritma, diantaranya adalah *Greedy Algorithm* dan *Dynamic Programming*. Tentu cara implementasi ini memiliki kompleksitas waktu yang berbeda, bergantung pada persoalan yang akan dipecahkan. Pada makalah ini, penulis akan membandingkan kedua cara implementasi yang telah disebutkan pada *0-1 Knapsack Problem.

Kata Kunci*— Algoritma, *Knapsack Problem*, Kompleksitas, *Greedy Algorithm* , *Dynamic Programming

I. PENDAHULUAN

Algoritma yang baik adalah algoritma yang mangkus dan sangkil. Dengan kata lain, algoritma yang diimplementasikan haruslah memakan waktu dan ruang memori seminimal mungkin. Dengan demikian, algoritma dapat digunakan untuk permasalahan dengan nilai input yang sangat besar.

Knapsack Problem adalah permasalahan dalam optimisasi kombinatorial. Permasalahan ini telah dipelajari selama lebih dari 1 abad, bermula dari pemikiran matematikawan Tobias Dantzig dalam memecahkan masalah dalam mengangkut barang dengan nilai tertinggi tanpa melebihi kapasitas dari wadah penyimpanan yang digunakan.



Gambar 1. Ilustrasi Knapsack Problem (Sumber : https://askgif.com/blog/58/how-to-solve-knapsack-problem-using-dynamic-programming)

Knapsack Problem berbunyi sebagai berikut : Diberikan kumpulan barang yang memiliki berat dan nilai tertentu, tentukanlah nilai terbesar yang dapat diangkut dalam wadah penyimpanan dengan kapasitas beban tertentu.

Knapsack Problem sering muncul dalam alokasi sumber daya dengan keuangan sebagai batasan. *Problem* ini dipelajari dalam bidang kombinatorika, bidang komputasi, kriptografi, teori kompleksitas, matematika terapan, dan lain sebagainya.

Ada berbagai tipe *Knapsack Problem*, tetapi yang akan dibahas pada masalah ini adalah yang paling dasarnya, dikenal dengan nama *0-1 Knapsack Problem.* Penulis akan mencoba *Greedy Algorithm* dan *Dynamic Programming* yang ditulis dalam Bahasa C++ untuk memecahkan *0-1 Knapsack Problem* ini.

II. DASAR TEORI

A. *Kompleksitas Algoritma*

Menurut Kamus Besar Bahasa Indonesia (KBBI), kompleksitas berarti kerumitan/keruwetan. Secara bahasa, kompleksitas algoritma dapat diartikan sebagai tingkat kerumitan algoritma. Kompleksitas algoritma menunjukkan seberapa banyak waktu dan seberapa besar memori yang dibutuhkan bagi computer untuk menjalankan algoritma yang dimaksud. Namun, kedua aspek ini bergantung pada arsitektur komputer serta jenis *compiler* yang digunakan Ketika mengeksekusi algoritma tersebut. Sehingga ruang dan waktu yang digunakan menjadi parameter yang bersifat relatif. Oleh karena itu, pengukuran kompleksitas algoritma tidak memandang aspek-aspek yang telah disebutkan, melainkan menggunakan model abstrak yang bersifat independen terhadap aspek-aspek luar.

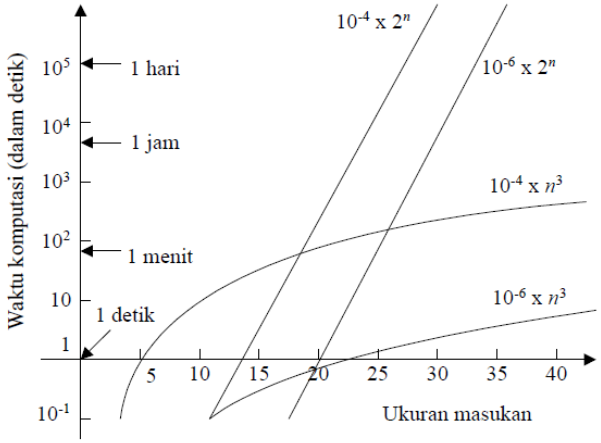
Kompleksitas algoritma biasa dinotasikan dalam bentuk fungsi yang menunjukkan relasi antara ukuran masukan dengan banyaknya operasi fundamental yang akan dilakukan algoritma.

Kompleksitas yang demikian disebut dengan kompleksitas waktu atau *time complexity*. Selain waktu, kompleksitas algoritma juga dapat dapat diteliti dari banyaknya storage/ruang yang dibutuhkan dalam menjalankan algoritma tersebut. Kompleksitas ini disebut *space complexity*.

Kompleksitas waktu biasa dinotasikan sebagai **T(n). T(n)** menunjukkan berapa banyak waktu yang dibutuhkan dalam mengolah masukan dengan besar *n*. Kompleksitas waktu terbagi menjadi tiga, yaitu:

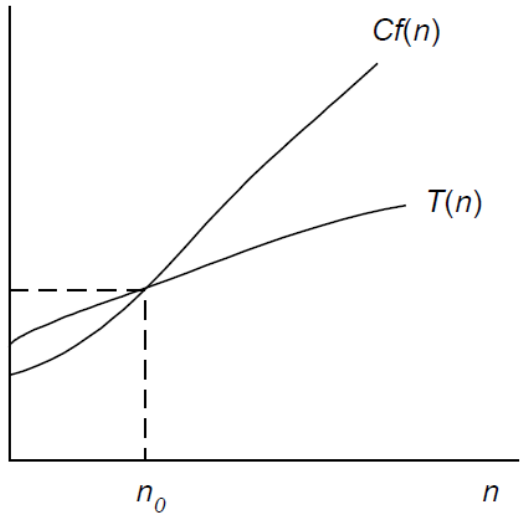
1. $\boldsymbol{T_{max}(n)}$ merupakan kompleksitas waktu algoritma untuk kasus terburuk (*worst case*).
2. $\boldsymbol{T_{min}(n)}$, merupakan kompleksitas waktu algoritma untuk kasus terbaik (*best case*).
3. $\boldsymbol{T_{avg}(n)}$, merupakan kompleksitas waktu algoritma untuk semua kasus secara rata-rata (*average case*).

Fungsi kompleksitas waktu **T(n)** biasa dinyatakan dengan notasi asimptotik (*Asymptotic Notations*). Misalkan, kita memiliki algoritma dengan $T_1(n) = n^3 + 18n^2 + 10$. Jika dibandingkan dengan algoritma lain yang memiliki $T_2(n) = n^3 + 10n + 1$, dapat dikatakan bahwa baik $T_1(n)$ maupun $T_2(n)$ memiliki laju pertumbuhan yang sangat mirip.



Gambar 2. Grafik Perbandingan $T(n)$ (Sumber : http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf)

Notasi asimptotik biasa dikenal sebagai Big-Oh Notation (O), yang dapat didefinisikan sebagai $T(n) = O(f(n))$. Definisi tersebut dibaca : "$T(n)$ adalah $O(f(n))$ atau $T(n)$ berorde paling besar $f(n)$ bila terdapat konstanta C dan n_0 sedemikian sehingga $T(n) \leq C(f(n))$ untuk $n \geq n_0$. $f(n)$ adalah batas atas (*upper bound*) dari $T(n)$ untuk nilai n yang besar.



Gambar 3. Grafik Hubungan $C(f(n))$ dengan $T(n)$ (*Sumber* : http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf)

Berikut merupakan contoh notasi Big-Oh yang umum dijumpai terurut mulai dari waktu membesar yang terkecil:

Kelompok Algoritma	*Nama*
O(1)	Konstan
O(*log* n)	Logaritmik
O(n)	Linier
O(n *log* n)	*n log n*
$O(n^2)$	Kuadratik
$O(n^3)$	Kubik
$O(n^k)$	Polinomial
$O(2^n)$ (atau $O(k^n)$)	Eksponensial
O(n!)	Faktorial

Gambar 4. Big-Oh yang Umum (*Sumber* : http://courses.cs.washington.edu/courses/cse373/14wi/lecture4.pdf)

B. *Greedy Algorithm*

Greedy Algorithm, secara bahasa berarti "algoritma serakah", adalah algoritma sederhana dan intuitif yang biasa digunakan dalam masalah optimasi. Melalui algoritma ini, programmer akan selalu membuat pilihan yang tampaknya terbaik saat itu. Artinya,algoritma ini membuat pilihan yang optimal secara lokal dengan harapan pilihan yang dibuat akan menghasilkan solusi yang optimal untuk semua kasus/secara global.

Asumsikan bahwa Anda memiliki fungsi objektif yang perlu dioptimalkan (baik dimaksimalkan maupun diminimalkan) pada titik tertentu. *Greedy Algorithm* membuat pilihan serakah di setiap langkah untuk memastikan bahwa fungsi tujuan dioptimalkan. *Greedy Algorithm* hanya memiliki satu kesempatan untuk menghitung solusi optimal sehingga tidak pernah mundur dan membalikkan keputusan.

```
Misal Anda memiliki 6 aktivitas yang terurutkan berdasarkan waktu selesai
    mulai[]  =  {1, 3, 0, 5, 8, 5};
    selesai[] =  {2, 4, 6, 7, 9, 9};
Satu orang maksimal hanya memiliki waktu t untuk menyelesaikan aktivitas.
Banyaknya aktivitas maksimal yang dapat dilakukan orang tersebut adalah:
```

Gambar 5. Contoh permasalahan untuk *Greedy Algorithm.* (*Sumber: milik pribadi*)

Persoalan diatas dapat dengan mudah diselesaikan dengan *Greedy Algorithm*. Hal yang perlu Anda lakukan adalah memilih aktivitas dengan selisih waktu selesai dan mulai yang terkecil.

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int mulai[6]  =  {1, 3, 0, 5, 8, 5};
    int selesai[6] =  {2, 4, 6, 7, 9, 9};
    int t, aktivitas = 0;   cin >> t;
    int selisih[6];
    for(int i = 0; i < 6; i++)  selisih[i] = selesai[i] - mulai[i];
    sort(selisih, selisih + 6);
    for(int i = 0; i < 6; i++){
        if(t - selisih[i] >= 0){
            aktivitas++;
            t -= selisih[i];
        }
    }
    cout << "Banyak aktivitas yang dapat dilakukan adalah " << aktivitas << endl;
}
D:\Kuliah\Semester 3\IF2120 - Matematika Diskrit\Bahan Makalah\Activity.exe
7
Banyak aktivitas yang dapat dilakukan adalah 4
```

Gambar 6. Penyelesaian dari masalah pada Gambar 5 dalam bahasa C++. (*Sumber: milik pribadi*)

Greedy Algorithm memiliki kelebihan dan kekurangan. Beberapa kelebihannya antara lain:

1. Relatif lebih mudah untuk dibayangkan
2. Menganalisis kompleksitas waktu *Greedy Algorithm* umumnya akan jauh lebih mudah daripada teknik lain (seperti *Divide & Conquer*). Untuk teknik *Divide & Conquer*, tidak jelas apakah teknik tersebut cepat atau lambat. Hal ini karena pada setiap tingkat rekursi ukuran semakin kecil dan jumlah *subproblem* bertambah.

Beberapa kekurangannya antara lain:

1. Perlu bekerja ekstra untuk membuktikan kebenaran dari algoritma. Bahkan jikalau algoritma Anda benar, membuktikan kebenaran *Greedy Algorithm* tetaplah sulit.
2. Rawan terkena *corner case*

C. *Dynamic Programming*

Dynamic Programming secara bahasa berarti memrogram secara dinamis. *Dynamic Programming*, atau biasa disebut DP, adalah teknik optimasi yang dapat kita gunakan untuk memecahkan masalah dimana pekerjaan yang sama diulang terus menerus. Anda tentu tidak asing dengan istilah *cache*, bukan? Penggunaan *cache* pada dasarnya adalah aplikasi dari DP.

Akan tetapi, DP tidak dapat digunakan pada semua permasalahan. Jika permasalahan tidak membutuhkan pekerjaan yang dilakukan secara berulang, maka penggunaan DP tidak dibutuhkan. Ada beberapa kriteria bagi sebuah permasalahan agar dapat diselesaikan dengan DP, yaitu:

1. Memiliki substruktur yang optimal
2. Memiliki *subproblem* yang tumpang-tindih / berulang (overlap)

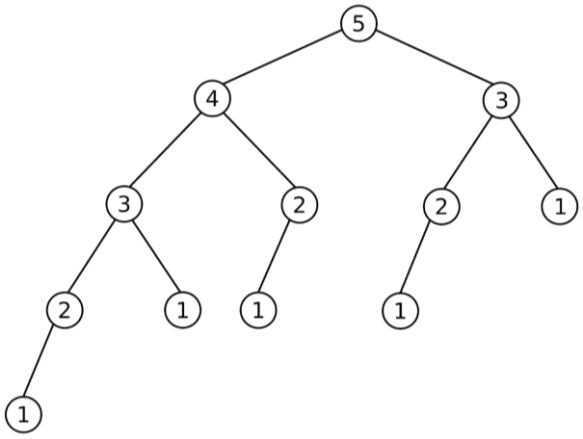
Substruktur yang optimal merupakan karakteristik paling penting dalam DP, bahkan rekursi secara umum. Jika sebuah permasalahan dapat dipecahkan dengan rekursi, besar kemungkinan terdapat substruktur yang optimal di dalamnya.

Subproblem yang berulang karakteristik kedua terpenting dalam DP. Jika permasalahan memiliki *subproblem* yang berulang, maka komputer akan memecahkan permasalahan yang sama lebih dari sekali, dan tentu ini memakan waktu.

Contoh permasalahan sederhana yang bisa diselesaikan dengan DP adalah mencari bilangan Fibonacci. Definisi bilangan Fibonacci adalah bilangan yang dihasilkan dari penjumlahan dua bilangan Fibonacci sebelumnya, dengan 0 sebagai bilangan Fibonacci ke-0 dan 1 sebagai bilangan Fibonacci ke-1. Secara matematis dituliskan sebagai berikut:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \; (n > 1)$$

Jika rekursi dari bilangan Fibonacci ini digambarkan dalam tree (untuk n = 5), maka akan terlihat seperti ini:



Gambar 7. Tree Rekursi Bilangan Fibonacci untuk n = 5 (*Sumber : https://simpleprogrammer.com/guide-dynamic-programming/*)

Dapat dilihat bahwa program akan menghitung F_3 dan F_2 lebih dari sekali. Kompeksitas waktu untuk cara rekursi ini adalah $\boldsymbol{O(2^n)}$. Angka yang dihasilkan tentu akan sangat besar jika n membesar. Bandingkan dengan implementasi DP berikut:

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
ll fib[100001];
void generate(int n){
    if(n > 100000) cout << "Melebihi kapasitas array\n";
    fib[0] = 0;
    fib[1] = 1;
    for(int i = 2; i <= n; i++) fib[i] = fib[i - 1] + fib[i - 2];
}
```

Gambar 8. Implementasi DP untuk Bilangan Fibonacci dalam bahasa C++. (*Sumber: milik pribadi*)

Pada gambar 8, array fib digunakan sebagai "cache" untuk program, sehingga komputer tidak perlu menghitung F_n yang sama berulang kali. Alhasil, kompleksitas waktu untuk implementasi DP pada pencarian bilangan Fibonacci adalah $\boldsymbol{O(n)}$. Tentu jauh lebih cepat jika dibandingkan dengan cara rekursif. DP tentu memiliki kelebihan dan kekurangan. Kelebihan DP diantaranya:

1. Menghemat waktu dalam penulisan dan kompilasi.
2. Lebih cepat secara general.

Sedangkan kekurangannya yaitu:

1. Memiliki risiko *runtime error* yang besar.
2. Boros memori
3. Tidak memiliki formasi yang tetap. Dengan kata lain, setiap permasalahan memiliki cara implementasi yang unik.

III. Pembahasan dan Implementasi

A. Penyelesaian 0-1 Knapsack Problem dengan Greedy Algorithm

Seperti yang telah dijelaskan sebelumnya, *Greedy Algorithm* adalah tentang memilih secara serakah. Sebagai contoh, kita memiliki daftar barang berikut:

Barang	0	1	2	3
Harga	24	18	18	10
Berat	24	10	10	7

Tabel 1. Contoh data barang beserta harga dan beratnya.

Misalkan kapasitas knapsack adalah W. Ada 2 cara “serakah” untuk menyelesaikannya, yaitu serakah secara harga (selanjutnya akan disebut cara 1) dan serakah secara rasio (selanjutnya akan disebut cara 2).

Untuk serakah secara harga, langkah-langkahnya yaitu:

1. Deklarasikan sebuah variabel (misal res) yang bertipe integer untuk menyimpan harga maksimal dari isi *knapsack*. Deklarasikan nilainya dengan 0.
2. Mengurutkan *array* dari harga barang .
3. Iterasikan array mulai dari indeks paling belakang jika diurut menaik atau dari paling depan jika sebaliknya. Periksa apakah W – harga[i] >= 0. Jika ya, maka ubah nilai W = W – harga[i] dan tambahkan nilai res menjadi res + harga[i]. Jika tidak, maka skip barang pada indeks tersebut.
4. Outputkan res

Untuk serakah secara rasio, langkah-langkahnya yaitu:

1. Sama seperti langkah no 1 pada cara sebelumnya.
2. Buat array yang akan diisikan rasio harga/berat barang. Perhatikan bahwa pembagian berpotensi menghasilkan bilangan real.
3. Urutkan array rasio.
4. Sama seperti langkah no 3 pada cara sebelumnya
5. Sama seperti langkah no 4 pada cara sebelumnya

Perhatikan bahwa cara 1 efektif jika W cukup besar. Misalkan nilai W adalah 50. Dengan cara 1, hasil yang didapatkan adalah 60 dan merupakan jawaban yang optimal. Angka ini didapat dengan memilih barang 0, 1, dan 2. Bagaimana jika nilai W diperkecil, misal 24? Dengan cara ini, hasil yang didapatkan adalah 24. Namun, 24 bukanlah hasil optimal, karena jawaban yang optimal adalah 36 (memilih barang 1 dan 2).

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<pair<double, int> > vp;
int knapsackGreedyValue(int W, int wt[], int val[], int n){
    if(n == 0 || W == 0)    return 0;
    int result = 0;
    double ratio;
    for(int i = 0; i < n ; i++){
        vp.push_back(make_pair(val[i], i));
    }
    sort(vp.begin(), vp.end()); //sorting berdasarkan harga (value) secara menaik
    for(int i = vp.size() - 1; i >= 0; i--){
        int temp = vp[i].second;
        if(W - wt[temp] >= 0){
            cout << "Barang yang terpilih adalah barang no-" << temp << endl;
            result += val[temp];
            W -= wt[temp];
        }
        else    continue;
    }
    return result;
}
```

```
Masukkan banyak benda: 4
Masukkan harga dan berat unuk barang ke-0: 24 24
Masukkan harga dan berat unuk barang ke-1: 18 10
Masukkan harga dan berat unuk barang ke-2: 18 10
Masukkan harga dan berat unuk barang ke-3: 10 7
Masukkan kapasitas knapsack: 50
Barang yang terpilih adalah barang no-0
Barang yang terpilih adalah barang no-2
Barang yang terpilih adalah barang no-1
60
```

```
Masukkan banyak benda: 4
Masukkan harga dan berat unuk barang ke-0: 24 24
Masukkan harga dan berat unuk barang ke-1: 18 10
Masukkan harga dan berat unuk barang ke-2: 18 10
Masukkan harga dan berat unuk barang ke-3: 10 7
Masukkan kapasitas knapsack: 25
Barang yang terpilih adalah barang no-0
24
--------------------------------
```

Gambar 9. Implementasi Cara 1 dengan C++ dan Hasil untuk Kedua *Dummy Test Case.* (*Sumber: milik pribadi*)

Barang	0	1	2	3
Harga	24	18	18	10
Berat	24	10	10	7
Rasio	1.0	1.8	1.8	1,4

Tabel 2. Tabel 1 Ditambah Baris untuk Rasio Harga dan Berat Masing-Masing Barang.

Bagaimana jika kita menggunakan cara 2? Jika W = 25, cara ini akan menghasilkan 36, yang merupakan jawaban optimal. Namun, jika W = 50, cara ini akan menghasilkan 46 yang didapat dengan memilih barang 1,2, dan 3. Angka ini tentu berbeda dengan jawaban optimalnya, yaitu 60.

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<pair<double, int> > vp;
int knapsackGreedyRatio(int W, int wt[], int val[], int n){
    if(n == 0 || W == 0)    return 0;
    int result = 0;
    double ratio;
    for(int i = 0; i < n ; i++){
        ratio = (double)val[i] / (double)wt[i]; //rasio harga/berat
        vp.push_back(make_pair(ratio, i));
    }
    sort(vp.begin(), vp.end()); //sorting ratio
    for(int i = vp.size() - 1; i >= 0; i--){
        int temp = vp[i].second;
        //cout << vp[i].first << endl;
        if(W - wt[temp] >= 0){
            cout << "Barang yang terpilih adalah barang no-" << temp << endl;
            result += val[temp];
            W -= wt[temp];
        }
        else    continue;
    }
    return result;
}
```

```
Masukkan banyak benda: 4
Masukkan harga dan berat unuk barang ke-0: 24 24
Masukkan harga dan berat unuk barang ke-1: 18 10
Masukkan harga dan berat unuk barang ke-2: 18 10
Masukkan harga dan berat unuk barang ke-3: 10 7
Masukkan kapasitas knapsack: 25
Barang yang terpilih adalah barang no-2
Barang yang terpilih adalah barang no-1
36
--------------------------------
```

```
Masukkan banyak benda: 4
Masukkan harga dan berat unuk barang ke-0: 24 24
Masukkan harga dan berat unuk barang ke-1: 18 10
Masukkan harga dan berat unuk barang ke-2: 18 10
Masukkan harga dan berat unuk barang ke-3: 10 7
Masukkan kapasitas knapsack: 50
Barang yang terpilih adalah barang no-2
Barang yang terpilih adalah barang no-1
Barang yang terpilih adalah barang no-3
46
--------------------------------
```

Gambar 10. Implementasi Cara 2 dengan C++ dan Hasil untuk Kedua *Dummy Test Case.* (*Sumber: milik pribadi*)

Kedua cara memiliki kompleksitas waktu yang sama yaitu $\boldsymbol{O(nlogn)}$. Namun, pendekatan menggunakan *Greedy Algorithm* pada *0-1 Knapsack Problem* ternyata tidak menghasikan solusi yang optimal secara terus menerus.

B. Penyelesaian 0-1 Knapsack Problem dengan Dynamic Programming

Pada bagian ini, penulis akan mencoba pendekatan dengan DP untuk menyelesaikan *0-1 Knapsack Problem.* Seperti yang telah dijelaskan pada bab II, sebuah permasalahan membutuhkan substruktur yang optimal dan *subproblem* yang berulang.

Dengan *test case* yang sama, akan dicoba penggunaan cara rekursif terlebih dahulu untuk melihat apakah kedua syarat yang

telah disebutkan ada pada permasalahan ini. Secara sederhana, Pertimbangkan setiap *subset* barang dan hitung total berat dan harga semua *subset*. Anda hanya perlu mepertimbangkan subset yang totalnya lebih kecil dari atau sama dengan W. Hanya ada dua kasus saat menentukannya, yaitu jika barang diambil atau tidak. Kemudian, dapat dicari harga maksimum dari dua nilai berikut:

1. Harga maksimum yang diperoleh dari n-1 barang dan bobot W.
2. Harga item ke-n ditambah harga maksimum yang didapat dari n-1 barang, dengan nilai W dikurangi berat barang ke-n.

```cpp
#include <bits/stdc++.h>
using namespace std;
int maxx(int a, int b){
    return (a > b) ? a : b;
}
int knapsackRec(int W, int wt[], int val[], int n){
    if (n == 0 || W == 0)   return 0;
    if (wt[n - 1] > W)
        return knapsackRec(W, wt, val, n - 1);
    // Cari maksimum dari kedua kasus:
    // (0) Barang ke-n tidak dimasukkan
    // (1) Barang ke-n dimasukkan
    else
        return maxx(val[n - 1] + knapsackRec(W - wt[n - 1], wt, val, n - 1), knapsackRec(W, wt, val, n - 1));
}
```

```
Masukkan banyak benda: 4
24 24
18 10
18 10
10 7
Masukkan kapasitas knapsack: 25
36
--------------------------------
Masukkan banyak benda: 4
24 24
18 10
18 10
10 7
Masukkan kapasitas knapsack: 50
60
```

Gambar 11. Implementasi Cara Rekursif dengan C++ dan Hasil untuk Kedua *Dummy Test Case.* (*Sumber: milik pribadi*)

Mengacu pada solusi optimal yang telah dibahas pada bagian sebelumnya, cara rekursif mampu mendapatkan solusi optimal dari kedua kasus. Namun, seperti pada kasus Fibonacci, cara rekursif ini akan memiliki kompleksitas waktu $\boldsymbol{O(2^n)}$ dan kompleksitas ruang $\boldsymbol{O(1)}$. Hal ini terjadi karena cara rekursif akan selalu memanggil dirinya sendiri, tidak peduli apakah dia sudah pernah mengerjakan tugas tersebut. Karena cara rekursif sama sekali tidak menyimpan apa yang telah dikerjakan, ia hampir tidak memakan memori sama sekali. Selanjutnya mari kita coba cara DP.

Penulis akan menambahkan sebuah *array* dua dimensi yang akan menyimpan solusi terbaik untuk berat W dan n barang. *Array* ini kemudian akan menggantikan pemanggilan rekursi pada cara sebelumnya.

```cpp
#include <bits/stdc++.h>
using namespace std;
int maxx(int a, int b){
    return (a > b) ? a : b;
}
int knapsackDP(int W, int wt[], int val[], int n) {
    int w, knapsack[n + 1][W + 1];
    for (int i = 0; i <= n; i++){
        for (w = 0; w <= W; w++){
            if (i == 0 || w == 0)   knapsack[i][w] = 0;
            //menentukan apakah lebih baik barang diambil atau tidak ditentukan dibawah ini
            else if (wt[i - 1] <= w){
                knapsack[i][w] = maxx(val[i - 1] + knapsack[i - 1][w - wt[i - 1]], knapsack[i - 1][w]);
            }
            else    knapsack[i][w] = knapsack[i - 1][w];
        }
    }
    return knapsack[n][W];
}
```

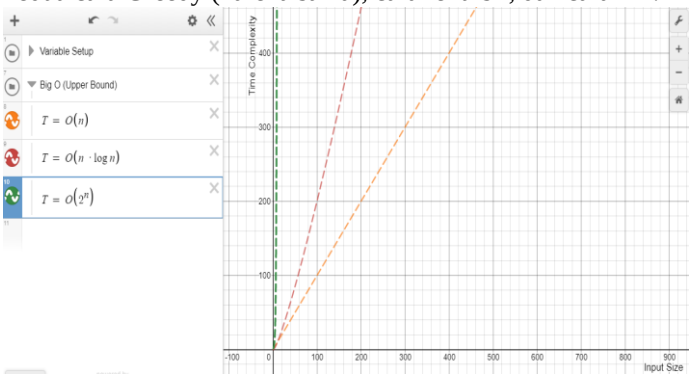
```
Masukkan banyak benda: 4
24 24
18 10
18 10
10 7
Masukkan kapasitas knapsack: 25
36
```

```
Masukkan banyak benda: 4
24 24
18 10
18 10
10 7
Masukkan kapasitas knapsack: 50
60
--------------------------------
```

Gambar 12. Implementasi Cara DP dan Hasil untuk Kedua *Dummy Test Case.* (*Sumber: milik pribadi*)

Dapat dilihat pada Gambar 12, cara DP pun menghasilkan solusi optimal. Kompleksitas waktu dan kompleksitas ruang untuk cara ini adalah $\boldsymbol{O(n * W)}$, dengan n adalah banyak barang dan W adalah kapasitas *knapsack*.

Berikut adalah grafik perbandingan kompleksitas waktu dari kedua cara Greedy (karena sama), cara rekursif, dan cara DP:



Gambar 13. Grafik Perbandingan Kompleksitas Waktu (*Sumber : Dibuat di website* https://www.desmos.com)

IV. KESIMPULAN

Mengacu pada hasil analisis dan impelementasi *Greedy Algorithm* dan *Dynamic Programming* dengan bahasa C++, dapat disimpulkan bahwa pada *0-1 Knapsack Problem*, impelementasi menggunakan *Dynamic Programming* adalah yang terbaik untuk saat ini. Hal ini didasarkan pada konsistensinya dalam menghasilkan solusi yang optimal. Selain itu, cara ini pun memiliki kompleksitas waktu polinomial dan dinilai sebagai kompleksitas waktu terbaik. Dapat dilihat pula pada gambar 13, pertumbuhan grafik kompleksitas waktu

Greedy Algorithm ($\boldsymbol{O(n \log n)}$) lebih besar jika dibandingkan $\boldsymbol{O(n * W)}$ milik cara DP. Cara rekursif yang sempat dibahas pada bagian DP dapat menjadi solusi jika nilai input sangat kecil, namun penulis tetap menyarankan untuk menggunakan algoritma DP, sekali lagi karena selalu menghasilkan solusi yang optimal dan cepat.

VI. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa, karena berkat kehadiran-Nya, penulis dapat menyelesaikan makalah ini. Tak lupa penulis berterima kasih kepada kedua orang tua penulis, serta Bapak Dr. Ir. Rinaldi, MT sebagai dosen mata kuliah Matematika Diskrit Terakhir, penulis ingin berterima kasih kepada sahabat penulis yang bersedia menemani dan menyumbangkan ide selama pembuatan makalah ini.

Referensi

[1] https://askgif.com/blog/58/how-to-solve-knapsack-problem-using-dynamic-programming (Diakses pada 09 Desember 2020)
[2] https://www.c-sharpcorner.com/article/01-knapsack-problem-by-using-greedy-method/ (Diakses pada 09 Desember 2020)
[3] https://courses.cs.washington.edu/courses/cse373/14wi/lecture4.pdf (Diakses pada 09 Desember 2020)
[4] http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf (Diakses pada 09 Desember 2020)
[5] http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf (Diakses pada 09 Desember 2020)
[6] https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/ (Diakses pada 10 Desember 2020)
[7] https://www.geeksforgeeks.org/greedy-algorithms/ (Diakses pada 10 Desember 2020)
[8] https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_01_knapsack.htm#:~:text=In%200-1%20Knapsack%2C%20items,whole%20or%20should%20leave%20it.&text=0-1%20Knapsack%20cannot%20be,may%20give%20an%20optimal%20solution (Diakses pada 10 Desember 2020)
[9] https://simpleprogrammer.com/guide-dynamic-programming/ (Diakses pada 10 Desember 2020)

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.

Cirebon, 11 Desember 2020

Wisnu Aditya Samiadji - 13519093