Perbandingan Performa Algoritma *Greedy* dan *Dynamic Programming*

Pratamamia Agung Prihatmaja (NIM 13515142)
Program Studi Teknik Informatikan
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha No. 10 Bandung 40132, Indonesia
apratamamia@gmail.com

Abstrak*—Algoritma *greedy* dan *dynamic programming* merupakan dua algoritma yang cukup dikenal luas di kalangan komunitas *programmer* dan banyak digunakan dalam pemecahan masalah komputasi. Kedua algoritma mempunyai keunggulan dan kelemahan masing-masing dan pada beberapa kasus dapat saling menggantikan satu sama lain, bergantung pada kebutuhan dan spesifikasi sistem yang akan dikembangkan. Oleh karena itu, butuh pemahaman yang mendalam mengenai performa masing-masing algoritma sebelum mulai menggunakan algoritma tersebut. Komponen performa yang cukup penting untuk diperhatikan adalah kompleksitas waktu, kompleksitas ruang, dan solusi yang dihasilkan. Pada makalah ini, penulis akan memaparkan hasil pengujian performa algoritma *greedy* dan *dynamic programming* dalam penyelesaian *1/0 knapsack problem.

Kata kunci—greedy; dynamic programming; knapsack problem; performa algoritma

I. Pendahuluan

Seiring dengan semakin berkembangnya ilmu dan teori komputasi, berbagai macam algoritma bermunculan untuk dapat memecahkan suatu persoalan komputasi. Berbagai algoritma ini mempunyai keuntungan dan kekurangan masing-masing. Oleh karena itu, penggunaan suatu algoritma sangat menentukan performa dari sistem yang dikembangkan.

Dengan semakin banyaknya algoritma yang ditemukan, maka pengembang sistem (*system developer*) perlu untuk lebih pintar dalam memilih algoritma yang cocok untuk digunakan dalam sistem yang akan dibuat. Pengembang perlu memperhatikan spesifikasi sistem sebelum menentukan algoritma yang akan digunakan sehingga performa sistem dapat mencapai maksimal.

Di antara berbagai macam algoritma yang ada, algoritma *greedy* dan *dynamic programming* merupakan dua algoritma yang cukup sering dipakai. Kedua algoritma ini mempunyai hubungan yang cukup erat dan dalam implemetasinya sering dipakai secara bergantian, tergantung pada hasil yang ingin dicapai. Keduanya juga banyak digunakan dalam memecahkan berbagai persoalan komputasi, seperti *0/1 knapsack problem*, *job allocation problem*, dan berbagai persoalan komputasi lain.

II. Dasar Teori

A. *Algoritma* Greedy

Algoritma *greedy* merupakan salah satu algoritma yang cukup populer dalam memecahkan persoalan optimasi. Algoritma ini menggunakan pendekatan iteratif dalam implementasinya. Pada setiap langkah, dipilih solusi optimum lokal, dengan harapan dapat mencapai solusi optimum global.

Algoritma ini tidak menjamin dapat tercapai solusi optimum. Namun, dengan pemilihan strategi *greedy* yang baik, dapat diperoleh solusi yang mendekati optimum atau bahkan dapat dibuktikan solusinya selalu optimum. Keunggulan utama dari algoritma ini adalah dapat menyelesaikan persoalan dalam kompleksitas waktu yang cukup baik.

Dalam implementasinya, terdapat lima elemen utama dari algoritma *greedy*, yaitu:

a) Himpunan kandidat

Merupakan himpunan yang merepresentasikan elemen yang dapat diambil pada suatu langkah.

b) Himpunan solusi

Merupakan himpunan penyelesaian yang telah diambil hingga suatu langkah tertentu.

c) Fungsi seleksi

Fungsi yang digunakan untuk mengambil elemen himpunan kandidat di setiap langkahnya.

d) Fungsi kelayakan

Fungsi yang digunakan untuk melakukan pengecekan apakah himpunan solusi masih valid, yaitu tidak melanggar *constraint* yang ditentukan oleh persoalan (kapasitas *knapsack*, dsb.).

e) Fungsi objektif

Fungsi yang diharapkan dari solusi persoalan (mencapai *value* maksimum, dsb.).

B. *Algoritma* Dynamic Programming

Algoritma *dynamic programming* merupakan metode pemecahan persoalan komputasi dimana persoalan dibagi-bagi ke dalam sekumpulan langkah atau tahapan yang saling

berkaitan, dimana dalam setiap tahapan dihasilkan suatu keputusan yang merupakan solusi optimal sampai tahap tersebut.

Pada *dynamic programming*, digunakan prinsip optimalitas. Prinsip ini menyatakan bahwa jika solusi total optimal, maka solusi hingga tahap ke-*k* juga optimal. Oleh karena itu, dengan menggunakan prinsip optimalitas ini, untuk membangun solusi optimum tahap *k+1*, dapat digunakan solusi optimum tahap sebelumnya tanpa harus kembali ke tahap awal.

Perbedaan utama algoritma *greedy* dan *dynamic programming* adalah pada banyak rangkaian keputusan yang dapat dihasilkan selama eksekusi program. Pada algoritma *greedy*, hanya satu rangkaian keputusan yang dihasilkan, sementara pada algoritma *dynamic programming* rangkaian keputusan yang dihasilkan dapat lebih dari satu. Rangkaian keputusan yang dijadikan solusi adalah rangkaian keputusan yang menghasilkan solusi optimum.

C. 0/1 Knapsack Problem

Knapsack problem atau *rucksack problem* merupakan suatu persoalan dalam *combinatorial optimization*. Persoalan ini pertama kali muncul pada tahun 1957, melalui suatu publikasi oleh George Dantzig. Beliau adalah seorang ilmuwan matematika asal Amerika yang cukup penting kontribusinya dalam bidang riset operasi, ilmu komputer, dan statistik. *Knapsack problem* merupakan salah satu persoalan yang tergolog dalam *NP* (*nondeterministic polynomial)—complete problems.*

Knapsack problem menyatakan bahwa jika diberikan serangkain barang yang diketahui berat dan nilainya, serta suatu wadah yang mempunyai kapasitas berat maksimum, akan ditentukan himpunan barang yang dapat dibawa sedemikian sehingga barang-barang tersebut tidak melebihi kapasitas wadah, serta nilai dari barang-barang tersebut maksimum. Setiap barang tmerupakan satu kesatuan, sehingga tidak dapat dibagi ke dalam sub-bagian yang lebih kecil. Oleh karena itu, untuk setiap barang, hanya ada kemungkinan untuk membawanya atau meninggalkannya. Secara matematis, persoalan *knapsack* ini dapat dinyatakan sebagai berikut.

$$maximize\left(\sum_{i=1}^{n} V_i X_i\right), where \sum_{i=1}^{n} W_i X_i < C$$

dimana V_i merupakan nilai dari barang ke-*i*, W_i merupakan berat barang ke-*i*, X_i adalah keputusan apakah barang ke-*i* dibawa (nilai 1) atau tidak dibawa (nilai 0), serta *C* merupakan kapasitas maksimum dari *knapsack*.

Tabel 1: Contoh Knapsack Problem

Item	1	2	3	4	5	6
Nilai	12	6	14	23	8	15
Berat	8	12	23	10	8	12

Knapsack problem merupakan salah satu persoalan yang sangat populer dan pengaplikasiannya muncul dalam kehidupan nyata dalam berbagai bentuk. Menurut Stony Brook University Algorithm Repository (1998), *knapsack problem* merupakan persoalan komputasi ke-18 paling populer dan ke-4 paling dibutuhkan. Persoalan ini mempunyai banyak implementasi, di antaranya adalah untuk *resource allocation*, *budget control*, *network flow*, dan sebagainya.

III. Implemetasi Algoritma *Greedy* pada *0/1 Knapsack Problem*

Pada bagian ini, akan dipaparkan bagaimana penerapan algoritma *greedy* dalam pemecahan persoalan *0/1 knapsack problem.* Terdapat berbagai pendekatan dan strategi untuk menyelesaikan persoalan ini menggunakan algoritma *greedy*, dan masing-masing mempunyai efektifitas tersendiri, terutama dalam hal kemampuan untuk mencapai solusi optimum.

A. Strategi Greedy

Seperti telah dinyatakan sebelumnya, terdapat berbagai strategi penyelesaian *1/0 knapsack problem* menggunakan algoritma *greedy.* Perbedaan utama dari satu strategi dengan strategi lain adalah pada fungsi seleksi yang digunakan untuk mengambil elemen di setiap tahapnya. Pada penelitian ini, akan digunakan tiga macam strategi, yaitu *greedy by value*, *greedy by weight*, dan *greedy by density.* Dari masing-masing strategi akan dilihat strategi mana yang menghasilkan solusi lebih optimum.

Greedy by value adalah suatu strategi dimana pada setiap tahap, dipilih barang yang nilainya paling besar. Strategi ini mencoba untuk mencapai keuntungan maksimum dengan memasukkan barang yang paling menguntungkan terlebih dahulu.

Greedy by weight adalah strategi *greedy* yang pada tiap tahapnya memasukkan barang yang teringan. Strategi ini mencoba mencapai keuntungan maksimum dengan memasukkan sebanyak-banyaknya barang ke dalam *knapsack.*

Greedy by density merupakan suatu strategi yang pada tiap langkahnya mencoba memasukkan barang dengan densitas keuntungan (*value/weight)* terbesar. Strategi ini mencoba untuk mencapai keuntungan maksimum dengan memasukkan barang yang keuntungan per unit berat-nya terbesar terlebih dahulu.

B. Implementasi

Implementasi algoritma *greedy* dalam pemecahan *1/0 knapsack problem* dapat dilihat seperti pada *pseudo-code* berikut.

Listing 1: Algoritma Greedy

```
Greedy algorithm for 1/0 knapsack problem
Input:
1. Array of struct item {
    value : int;
    weight : int;
    density : float; }
2. Knapsack capacity.
```

Algoritma:

1. (opsional: untuk *greedy by density*) Hitung densitas keuntungan tiap barang.
2. Urutkan array item berdasarkan strategi *greedy* yang digunakan.
   - Untuk *greedy by value*, urutkan berdasarkan *value* secara menurun.
   - Untuk *greedy by weight*, urutkan berdarkan *weight* secara menaik.
   - Untuk *greedy by density*, urutkan berdasarkan *density* secara menurun.
3. Untuk masing-masing barang dalam array, ambil barang tanpa melebihi kapasitas *knapsack* secara berurutan dari elemen pertama array terurut. Jika barang diambil, tambahkan nilai barang ke nilai dari *knapsack*, dan kurangkan kapasitas *knapsack* dengan berat barang.
4. Tampilkan *value* dari *knapsack*.

C. Performa Algoritma Greedy

Dalam hal kompleksitas waktu, algoritma *greedy* yang diimplementasikan mempunyai kompleksitas yang terdiri atas:

1. Kompleksitas dalam pengurutan array menggunakan *quick sort*, yaitu *O(n log n)*.
2. Kompleksitas untuk memilih barang yang akan dimasukkan ke dalam *knapsack*, yaitu sebesar *O(n)*.

Dari poin 1 dan 2 di atas, dapat disimpulkan kompleksitas waktu total untuk algoritma *greedy* untuk penyelesaian *1/0 knapsack problem* ini adalah *O(n log n) + O(n) = O(n log n)*.

Dalam hal kompleksitas ruang, algoritma ini membutuhkan memori yang digunakan untuk menyimpan informasi setiap barang sebanyak *n* buah. Memori yang digunakan ini masih cukup proporsional, terutama untuk sistem dengan memori yang terbatas.

IV. Implementasi Algoritma *Dynamic Programming* Pada *1/0 Knapsack Problem*

Pada bagian ini, akan dipaparkan bagaimana penerapan algoritma *dynamic programming* dalam penyelesaian *1/0 knapsack problem.*

A. Strategi

Pada algoritma *dynamic programming* untuk penyelesaian *1/0 knapsack problem* ini, persoalan dibagi ke dalam tahap-tahap. Pada masing-masing tahap, akan ditentukan pada tahap tersebut keuntungan maksimum dapat dicapai dengan mengambil atau tidak mengambil barang ke-*i*. Solusi optimal untuk setiap tahap dibangun secara rekursif dengan mengacu pada solusi dari tahap sebelumnya.

Secara matematis, solusi dari *1/0 knapsack problem* menggunakan algoritma *dynamic programming* adalah sebagai berikut.

Basis:

$$f_0(y) = 0, y \geq 0$$

$$f_k(y) = -\infty, y < 0$$

Rekurens:

$$f_k(y) = \max(f_{k-1}(y), v_k + f_{k-1}(y - w_k), k > 0$$

Pada ekspresi matematika di atas, $f_k(y)$ menyatakan keuntungan optimum pada tahap ke-*k* untuk kapsaitas *knapsack* sebesar *y* satuan berat, v_k menyatakan nilai dari barang ke-*k*, dan w_k adalah berat barang ke-*k*.

B. Implementasi

Implementasi algoritma *dynamic programming* untuk pemecahan *1/0 knapsack problem* dapat dilihat pada *pseudo-code* berikut.

Listing 2: Algoritma Dynamic Programming

```
DP algorithm for 1/0 knapsack problem
Input:
1. Array value (v).
2. Array weight (w).
3. Banyak item barang (n).
4. Kapasitas knapsack (C)
Algoritma:
1. Inisialisasi basis
   for i = 0 to W do
      m[0,i] = 0
   end for
2. Rekurens
   for i = 1 to n do
     for j = 0 to C do
       if w[i] ≤ j then
          m[i,j]=max(m[i-1,j],m[i-1,j- w[i]]
          + v[i])
       else
          m[i,j] = m[i-1,j]
       end if
     end for
   end for
3. Tampilkan value maksimum
```

Pada implementasi seperti pada *pseudo-code* di atas, digunakan suatu matriks status untuk menyimpan solusi dari masing-masing tahap. Penggunaan matriks tersebut diilustrasikan sebagai berikut.

Tabel 2: Contoh Persolan Knapsack Untuk Algoritma DP

Barang	1	2	3	4
Nilai	10	40	30	50
Berat	5	4	6	3
Kapasitas	10			

Tabel 3: Matriks Status pada Dynamic Programing

V(i,w)	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	0	40	40	40	40	40	50	50
3	0	0	0	40	40	40	40	40	50	70
4	0	0	50	50	50	50	90	90	90	90

C. Performa Algoritma Dynamic Programming

Untuk penyelesaian *1/0 knapsack problem*, algoritma *dynamic programming* mempunyai kompleksitas waktu terburuk sebesar *O(W.n)* dengan *W* adalah kapasitas *knapsack* dan *n* adalah banyak item barang yang dapat dimasukkan ke dalam *knapsack*. Walaupun kompleksitas waktu tersebut secara kasat mata terlihat seperti kompleksitas waktu polinomial, namum pada kenyataannya *O(W.n)* termasuk dalam *pseudo-polynomial*, sehingga *knapsack problem* ini masih tergolong dalam *NP-problem.*

Dalam hal kompleksitas ruang, algoritma *dynamic programming* ini membutuhkan kapasitas memori minimal untuk membuat matriks status sebesar *W.n*. Kapasitas memori minimal ini tergolong cukup besar dan perlu pertimbangan lebih terutama untuk penerapan pasa sistem dengan kapasitas memori terbatas.

V. Hasil Eksperimen

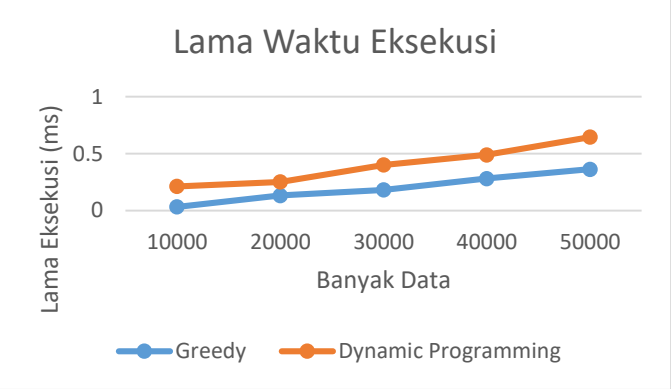
Pada bagian ini, akan dipaparkan hasil implementasi algoritma *greedy* dan *dynamic programming* dalam penyelesaian *1/0 knapsack problem*. Eksperimen dilakukan pada lingkungan sistem operasi Windows® 10 64-bit, CPU Intel® Core-i7 4720HQ, dan RAM 12 GB. Program dikembangkan dalam bahasa C++ dan menggunakan *compiler* GCC versi 5.1.0.

A. Waktu eksekusi

Pengukuran lama waktu eksekusi untuk tiap algoritma dilakukan dengan menyiapkan *testcase* yang mempunyai banyak data barang tertentu dan kemudian menggunakan *testcase* tersebut dalam eksekusi masing-masing algoritma. Waktu eksekusi yang dibutuhkan oleh masing-masing algoritma dapat dilihat pada ilustrasi berikut. Waktu eksekusi dinyatakan dalam satuan milidetik.

Tabel 4: Lama Eksekusi

Banyak Data	***Greedy***	***Dynamic Programming***
10000	0.03125	0.2125
20000	0.1325	0.2513
30000	0.1815	0.4012
40000	0.2812	0.4893
50000	0.3625	0.6452



Gambar 1. Grafik Lama Waktu Eksekusi Algoritma

B. Optimalitas Solusi

Pengujian solusi hasil eksekusi algoritma *greedy* dan *dynamic programming* dilakukan dengan cara yang sama dengan pengukuran waktu eksekusi kedua algoritma tersebut, yaitu dengan cara menyiapkan *testcase* dengan banyak data tertentu, kemudian menguji menggunakan *testcase* tersebut dalam eksekusi masing-masing algoritma. Berikut adalah hasil yang didapatkan dari pengujian tersebut.

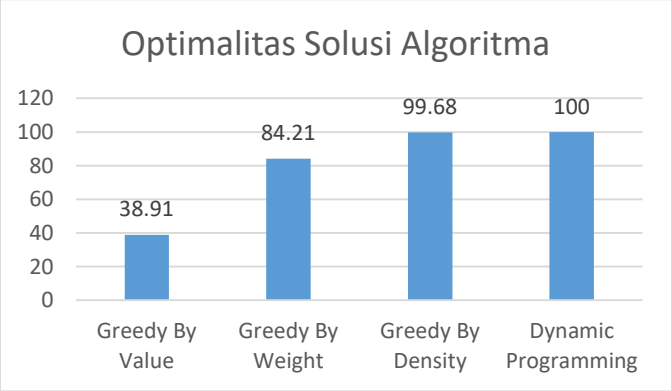
Tabel 5: Solusi yang Diberikan Masing-Masing Algoritma

Banyak Data	***Greedy by Value***	***Greedy by Weight***	***Greedy by Density***	***Dynamic Programming***
100	180	313	372	375
200	314	515	607	607
300	316	769	871	873
400	396	838	1089	1095
500	320	1255	1424	1424

Dari data tersebut, dapat ditarik kesimpulan bahwa optimalitas solusi yang didapat dari masing-masing algoritma adalah sebagai berikut.

Tabel 6: Optimalitas Solusi Masing-Masing Algoritma

Banyak Data	*Greedy by Value*	*Greedy by Weight*	*Greedy by Density*	*Dynamic Programming*
100	48%	83.47%	99.2%	100%
200	51.73%	84.84%	100%	100%
300	36.2%	88.09%	99.77%	100%
400	36.16%	76.53%	99.45%	100%
500	22.47%	88.13%	100%	100%



Gambar 2. Grafik Rata-Rata Optimalitas Algoritma

VI. Analisis

Pada bagian ini, akan dilakukan analisis dari data yang diperoleh dari hasil eksperimen mengenai performa algoritma *greedy* dan *dynamic programming* dalam pemecahan *1/0 knapsack problem.* Analisis yang dilakukan terutama mengenai lama waktu eksekusi, penggunaan sumber daya memori, dan optimalitas solusi yang dihasilkan.

Dalam hal waktu eksekusi, dapat dilihat dari Gambar 1 bahwa algoritma *dynamic programming* mempunyai waktu eksekusi yang lebih lama dibandingkan dengan algoritma *greedy.* Secara teoritis, hal ini dapat dijelaskan sebagai akibat dari kompleksitas waktu dari kedua algoritma. Kompleksitas waktu dari algoritma *dynamic programming* adalah $O(W.n)$ sedangkan untuk algoritma *greedy* adalah $O(n \log n)$.

Untuk penggunaan sumber daya memori, algoritma *greedy* membutuhkan memori untuk menyimpan data barang yang dapat dimasukkan ke dalam *knapsack*. Maka kebutuhan memori untuk algoritma ini cukup "murah" dan dapat sangat bermanfaat untuk sistem dengan kapasitas memori yang terbatas.

Pada algoritma *dynamic programming*, selain membutuhkan memori untuk menyimpan data barang, algoritma ini juga memerlukan memori untuk menyimpan matriks status di tiap tahapnya. Matriks ini memerlukan memori yang cukup besar, karena harus menyimpan data sebesar *W.n*, dengan *W* adalah kapasitas *knapsack* dan *n* adalah banyak barang yang dapat dimasukkan. Besar memori ini tentu cukup "mahal", apalagi untuk sistem dengan kapasitas memori yang terbatas.

Pada eksperimen yang dilakukan untuk menguji lama eksekusi algoritma *dynamic programming* pada bagian V di atas, digunakan data yang cukup besar (di atas sepuluh ribu data). Untuk membangun matriks status, tidak cukup jika hanya mengandalkan memori yang dialokasikan untuk *stack program.* Pada kenyataannya, pada lingkungan pengerjaan yang digunakan pada eksperimen ini, untuk alokasi *array integer* dengan besar lebih dari satu juta elemen tidak mampu lagi ditampung oleh *stack program.* Oleh karena itu, memori yang digunakan adalah memori pada *heap*.

Salah satu solusi untuk permasalahan besarnya kebutuhan memori pada algoritma *dynamic programming* adalah dengan cara dealokasi memori yang sudah tidak dibutuhkan secara berkala. Sebagai contoh, pada algoritma *dynamic programming* yang digunakan dalam penyelesaian *knapsack problem* seperti dipaparkan di atas, saat telah mencapai tahap ke-*k*, hanya dibutuhkan solusi dari tahap *k-1*. Oleh karena itu, matriks status untuk tahap 1 sampai *k-2* dapat di-dealokasi. Hal ini dapat meghemat penggunaan memori, karena memori yang telah di-dealokasi tersebut dapat digunakan untuk kebutuhan pada tahap selanjutnya.

Untuk optimalitas hasil eksekusi, dapat dilihat dari Gambar 2 bahwa algoritma *dynamic programming* selalu menghasilkan solusi optimal, sementara algoritma *greedy* tidak selalu menghasilkan solusi optimal. Namun, dapat dilihat juga bahwa beberapa strategi *greedy* yang berbeda juga menghasilkan optimalitas solusi yang berbeda pula.

Algoritma *greedy by value* tidak menghasilkan solusi yang cukup optimal, dengan rata-rata optimalitas 38.91%. Algoritma *greedy by weight* cukup optimal dengan rata-rata optimalitas mencapai 84.21%, sementara algoritma *greedy by density* paling optimal di antara strategi *greedy* lain, dengan optimalitas rata-rata 99.68%. Oleh karena itu, pemilihan strategi *greedy* yang sesuai sangat mempengaruhi solusi yang dihasilkan.

VII. Kesimpulan

Dari hasil analisis eksperimen performa algoritma *greedy* dan *dynamic programming*, diperoleh fakta bahwa masing-masing algoritma mempunyai keunggulan dan kelemahan masing-masing. Dari segi kompleksitas waktu dan penggunaan sumber daya memori, algoritma *greedy* lebih unggul, sedangkan dari sisi optimalitas solusi, algoritma *dynamic programming* lebih unggul.

Oleh karena itu, dalam mengaplikasikan salah satu dari dua algoritma tersebut ke suatu sistem,perlu dilihat secara cermat mengenai spesifikasi dan kebutuhan sistem. Jika sistem memiliki kapasitas memori yang terbatas atau hanya membutuhkan solusi pendekatan, maka algoritma *greedy* cocok untuk digunakan. Sebaliknya, apabila sistem membutuhkan solusi yang optimum, maka algoritma *dynamic programming* dapat lebih dipertimbangkan. Dalam implementasi algoritma *dynamic programming*, manajemen memori perlu untuk mendapat perhatian lebih mengingat besarnya memori yang

diperlukan dalam eksekusi algoritma ini, terutama ketika mengolah data yang cukup besar.

UCAPAN TERIMA KASIH

Pertama-tama, penulis mengucap syukur kepada Tuhan Yang Maha Esa karena atas berkat dan rahmat-Nya penulis dapat menyelesaikan makalah ini dengan baik. Kemudian penulis juga mengucapkan terima kasih kepada orang tua penulis yangtelah memberikan dukungan selama proses penulisan makalah ini. Penulis juga turut mengucapkan terima kasih kepada Dr. Masayu Leylia Khodra, S.T., M.T., Dr. Nur Ulfa Maulidevi, S.T., M.Sc., dan Dr. Ir. Rinaldi Munir, M.T. selaku dosen mata kuliah IF2211 Strategi Algoritma yang telah membimbing dan memberi materi kepada penulis selama proses pembelajaran mata perkuliahan Strategi Algoritma.

REFERENSI

[1] Munir, Rinaldi. *Diktat Kuliah Strategi Algoritma.* unpublished.

[2] A. Shaheen, A. Sleit, "Comparing between different approaches to solve the 0/1 knapsack problem", IJCSNS International Journal of Computer Science and Network Security, Vol. 16 No.7, Juli 2016.

[3] Brown, Kevin Q. *Dynamic Programming in Computer Science*. Carnegie Mellon University Research Showcase. 1979.

[4] Park, Jaehyun. *Dynamic Programming.* Stanford University. 2015.

[5] Erickson, Jeff. *Lecture in Dynamic Programming*. Illinois State University. 2014.

[6] A. Malik, A. Sharma, V. Saroha, "Greedy Algorithm", International Journal of Scientific and Research Publications, Vol. 3 Issue 8. Agustus 2013.

[7] Agarwal, Udit. *Algorithms Design and Analysis.* Dhanpat Rai and Co. 2017.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. *Introduction to Algorithm.* MIT Press: 2009. Edisi Ketiga.

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, 17 Mei 2017

Pratamamia Agung P
13515142