Pemanfaatan *Dynamic Programming* Pada Kompleksitas Algoritma

Vincent Tanjaya 13518133[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13518133@std.stei.itb.ac.id

***Abstract*—Dynamic Programming adalah suatu metode untuk menyelesaikan suatu permasalahan kompleks dengan memecahkannya menjadi bagian-bagian masalah kecil dan menyelesaikan masalah kecil tersebut hanya sekali, dan menyimpan solusi tersebut ke dalam struktur data contohnya array.**

***Keywords*— Dynamic Programming, kompleksitas algoritma, *Big O Notation, knapsack*.**

I. Pendahuluan

Dynamic Programming adalah suatu metode untuk menyelesaikan suatu permasalahan kompleks dengan memecahkannya menjadi bagian-bagian masalah kecil dan menyelesaikan masalah kecil tersebut hanya sekali, dan menyimpan solusi tersebut ke dalam struktur data contohnya array.

Pada saat ini, mengingat bahwa tingkat efisiensi suatu program sangat penting, dimana pada saat ini mengolah suatu data yang besari jika tidak dilakukan menggunakan algoritma yang efisien, maka akan mengakibatkan waktu pengolahan data berhari-hari, bahkan bertahun-tahun. Dengan mengetahui bahwa kompleksitas algoritma sangat penting, banyak orang berlomba-lomba untuk menemukan algoritma yang paling efisiensi untuk menyelesaikan suatu masalah.

Kompleksitas algoritma adalah besaran waktu dan ruang yang dibutuhkan untuk menyelesaikan suatu permasalahan. Semakin tinggi besaran waktu dan ruang yang diperlukan, maka semakin tinggi nilai kompleksitas suatu algoritma.

Dynamic Programming yang akan dibahas pada makalah ini adalah menyederhanakan suatu kompleksitas algoritma yang simpel, seperti fibonacci dan algoritma yang bersifat O(2^n) yang dimana kompleksitas algoritmanya bersifat eksponensial. Serta akan dibahas penyelesaian persoalan knapsack pada makalah ini.

II. Landasan Teori

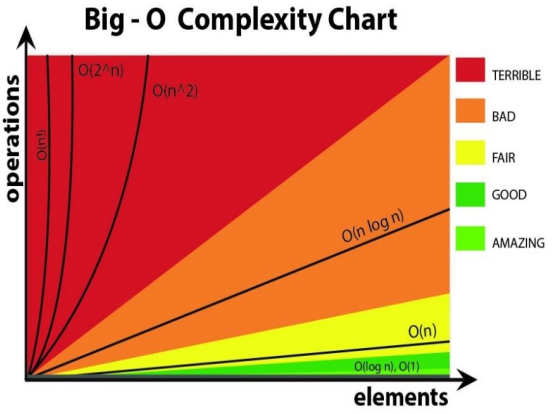
A. Kompleksitas Algoritma.

Kompleksitas algoritma adalah besaran waktu dan ruang yang dibutuhkan untuk menyelesaikan suatu permasalahan. Semakin tinggi besaran waktu dan ruang yang diperlukan, maka semakin tinggi nilai kompleksitas suatu algoritma. Besaran yang paling mempengaruhi nilai kompleksitas algoritma adalah besaran waktu, karena besaran ruang pada saat ini dapat diselesaikan dengan mudah. Besaran kompleksitas algoritma biasanya dinyatakan dengan *Big O Notation* (O(n)), dimana n adalah nilai dari besaran masukan. Contoh dimana suatu algoritma menerima input *array of integer* dimana berupa [1,2,8,6] maka nilai n adalah sebesar 4.

1. Big O Notation

Big O Notation adalah merupakan nilai kelajuan waktu dari suatu algoritma. Contoh-contoh nilai *Big O Notation* yang sering ditemukan

Notasi	Nama
$O(1)$	*Constant Time*
$O(N)$	*Linear Time*
$O(logN)$	*Logarithmic Time*
$O(NlogN)$	*NlogN Time*
$O(N^c)$ c>1	*Polynomial Time*
$O(C^N)$ c>1	*Exponential Time*
$O(N!)$	*Factorial Time*



Gambar 1. Grafik Kompleksitas Algoritma

Pada gambar diatas dapat dilihat bahwa kompleksitas algoritma yang diatas $O(NlogN)$ dikategorikan menjadi algoritma yang tidak bagus, dikarenakan pengolahan data yang bersifat diatas polynomial membutuhkan waktu yang sangat lama.

Contoh masing-masing algoritma yang menyatakan notasi pada tabel diatas:

1. *Constant Time*

   Contoh algoritma yang memiliki kompleksitas algoritma *Constant Time* adalah sebagai berikut:

```
def Sum1toN (n)

        return n*(n+1)/2
```

Gambar 2. Algoritma *Constant Time*

Algoritma diatas merupakan algoritma yang memiliki kompleksitas *Constant Time* karena, berapapun nilai n yang dimasukkan hanya terjadi 1 kali komputasi

2. *Linear Time*

```
def SumOfArray (arr):
    sum = 0
    for i in range(len(arr)):
        sum+= arr[i]
```

Gambar 3. Algoritma *Linear Time*

Algoritma diatas merupakan algoritma yang memiliki kompleksitas *Linear Time* karena, jumlah operasi tambah dilakukan sepanjang array yaitu n kali.

3. *Logarithmic Time*

```
function binary_search(A, n, T):
    L := 0
    R := n - 1
    while L <= R:
        m := floor((L + R) / 2)
        if A[m] < T:
            L := m + 1
        else if A[m] > T:
            R := m - 1
        else:
            return m
    return unsuccessful
```

Gambar 4. Algoritma *Logarithmic Time*

Algoritma diatas merupakan algoritma yang memiliki kompleksitas *Logarithmic Time*, karena *average case* diatas adalah 2log(n) dimana didalam *Big O Notation* dapat dinyatakan sama dengan $O(NlogN)$.

4. *NlogN Time*

```
procedure quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
                        partition    =
partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end procedure
```

Gambar 5. Algoritma *Quick Sort*

Algoritma diatas adalah salah satu algoritma yang memiliki kompleksitas algoritma O(NlogN). Algoritma diatas sering disebut dengan quick sort.

5. *Polynomial Time*

```
def SumOfArray (arr):
    for i in range(len(arr)-1):

for j in range(i+1,len(arr)):
            if(arr[i]>arr[j]):
                temp = arr[j]
                arr[j] = arr[i]
                arr[i] = temp
```

Gambar 6. Algoritma *Polynomial Time*

Algoritma diatas adalah salah satu contoh sort yang paling umum dan dikenal tidak efektif karena memiliki kompleksitas O(n^2) yaitu bubble sort.

6. *Exponential Time*

```
def fibonacci(n):
    if(n == 0):
        return 1
    else if (n == 1):
        return 1
    else

return
fibonacci(n-1)+fibonacci(n-2)
```

Gambar 7. Algoritma *Exponential Time*

Algoritma diatas memiliki kompleksitas algoritma eksponensial karena pada algoritma ini memecahkan subproblem tetapi subproblem yang sama dipecahkan lebih dari sekali. Kompleksitas algoritma ini adalah O(2^n). Algoritma ini dapat menggunakan *Dynamic Programming* untuk mengubah kompleksitasnya menjadi O(n) yang akan dibahas pada bagian selanjutnya.

7. *Factorial Time*

```
def factorial(n):
    for each in range(n):

        print(n)

        factorial(n-1)
```

Gambar 8. Algoritma *Factorial Time*

Algoritma diatas memiliki kompleksitas algoritma *Factorial Time.*

B. *Dynamic Programming*

Dynamic Programming adalah suatu metode untuk menyelesaikan suatu permasalahan kompleks dengan memecahkannya menjadi bagian-bagian masalah kecil dan menyelesaikan masalah kecil tersebut hanya sekali, dan menyimpan solusi tersebut ke dalam struktur data contohnya array.

Contoh-contoh penyederhanaan kompleksitas algoritma pada kasus umumnya:

Algoritma	Tidak menggunakan *Dynamic Programming*	Menggunakan *Dynamic Programming*
Fibonacci	$O(2^N)$	$O(N)$
Traveling Salesman Problem	$O(N!)$	$O(2^N * N^2)$
Knapsack Problem	$O(2^N)$	$O(NK)$
Longest Common Subsequence	$O(2^N)$	$O(NK)$

Dapat dilihat di tabel atas bahwa menggunakan *Dynamic Programming* untuk menyelesaikan persoalan dapat menghasilkan algoritma yang lebih efisien, contoh seperti fibonacci, untuk menghitung fibonacci(10) jika tidak menggunakan *Dynamic Programming* melakukan sekitar 1024 komputasi sedangkan menggunakan *Dynamic Programming* hanya melakukan 10 komputasi saja.

Untuk menyelesaikan masalah dengan *Dynamic Programming* ada 2 cara yaitu dengan *memoize* dan *Bottom-Up*. Pada contoh dibawah ini akan menampilkan 3 cara algoritma fibonacci yaitu dengan cara rekursif, *memoize*, dan *Bottom-Up*

```
def fibonacci(n):
    if(n == 0):
        return 1
    else if (n == 1):
        return 1
    else

return
fibonacci(n-1)+fibonacci(n-2)
```

Gambar 9. Algoritma *Fibonacci* dengan Rekursif

```
def fibonacci(n,memo):
    if(memo[n]!=Nil):
        return memo[n]
```

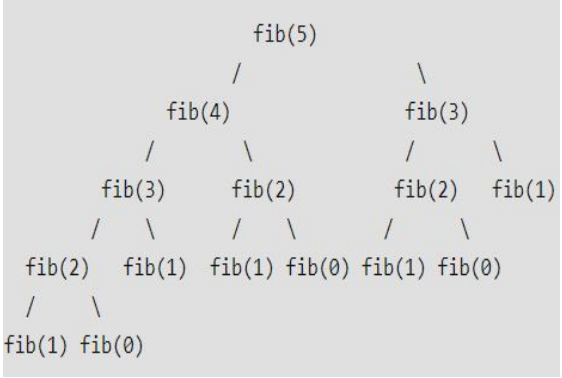
```
    else if (n == 1 or n == 2):
        return 1
    else

result=fibonacci(n-1)+fibonacci(n-
2)
        memo[n] = result

    return result
```

Gambar 10. Algoritma *Fibonacci* dengan *memoize*

```
def fibonacci(n):

    arr = []*n
    arr[0] = 1
    arr[1] = 1

    for i in range(2,n+1):

arr[i] = arr[i-1]+arr[i-2]
    return arr[n]
```

Gambar 11. Algoritma *Bottom-Up*

Dapat dilihat pada gambar diatas untuk algoritma fibonacci dengan rekursif pengulangan subproblem dilakukan lebih dari sekali. Ilustrasi ada pada gambar dibawah;



Gambar 12. Ilustrasi Fibonacci dengan Rekursif

Dari gambar atas terlihat bahwa subproblem fib(3) dihitung oleh komputer sebanyak 2 kali, dan dapat terlihat pada gambar diatas juga kompleksitas algoritma fibonacci dengan rekursif adalah O(2^n).

Untuk algoritma *memoize* dapat terlihat bahwa komputasi yang telah dilakukan disimpan ke dalam *array* kemudian nilai tersebut diakses jika nilainya memang sudah pernah dihitung.

Untuk solusi *Bottom-Up*, sepertinya namanya dari bawah ke atas, algoritma yang menggunakan *Bottom-Up* ini menghitung fibonacci yang dari awal hingga ke fibonacci yang diinginkan, penyelesaian *Bottom-Up* ada karena, komputer memiliki nilai maksimum pemanggilan rekursif pada *memoize*, sehingga pada saat menghitung fibonacci(1000), akan terjadi *error* pada solusi *memoize* dan pada solusi *Bottom-Up* tidak akan terjadi *error*.

III. Studi Kasus Pada Persoalan Knapsack

Knapsack Problem adalah salah satu persoalan yang lumayan sering diperbincangkan dalam mempelajari *Dynamic Programming*. Secara singkat pada persoalan ini adalah, anda dapat mengambil suatu berat tertentu untuk berjualan, terdapat daftar berat barang, dan keuntungan masing-masing barang, anda hanya dapat mengambil masing-masing hanya 1 barang, barang-barang apa saja yang diambil sehingga mendapatkan profit yang maksimum. Contoh Persoalan *Knapsack Problem:*

```
value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;
```

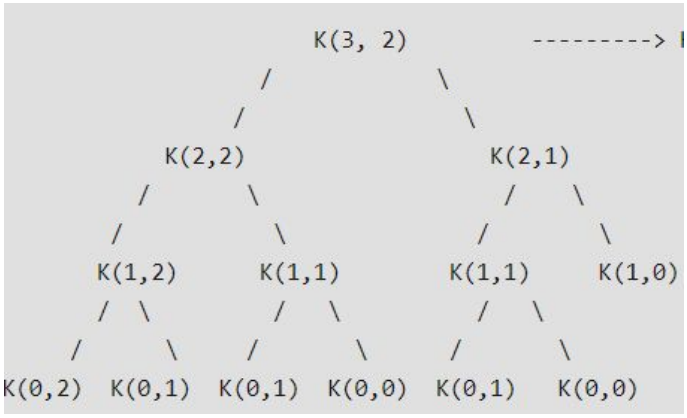
Gambar 13. Contoh Persoalan *Knapsack*

Seperti pada umumnya persoalan ini dapat diselesaikan dengan rekursif biasa dan menggunakan *Dynamic Programming*. Berikut adalah contoh penyelesaian *Knapsack Problem.*

```
def knapSack(W , wt , val , n):
    if n == 0 or W == 0 :
        return 0
    if (wt[n-1] > W):

       return knapSack(W,wt,val,n-1)

    else:

        return max(val[n-1]
        +knapSack(W-wt[n-1],
```

```
        wt , val , n-1),
        knapSack(W,wt,val,n-1))
```

Gambar 14. *Knapsack* menggunakan Rekursif



Gambar 15. Ilustrasi *Knapsack* menggunakan Rekursif

Pada Ilustrasi diatas dapat dilihat bahwa kompleksitas algoritma diatas adalah O(2^n), yang dimana masih tergolong kurang baik. Dengan menggunakan *Dynamic Programming, Knapsack Problem* dapat diselesaikan dengan O(NK).

```
def knapSack(W, wt, val, n):

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

          else:
             K[i][w] = K[i-1][w]

    return K[n][W]
```

Gambar 16. *Knapsack* menggunakan *Dynamic Programming*

Pada contoh diatas terlihat bahwa komputasi yang pernah dilakukan disimpan ke dalam matriks, sehingga tidak terjadi pengulangan komputasi. Kompleksitas Algoritma ini adalah O(NK). Algoritma diatas adalah menggunakan *memoize approach.*

```
def knapsack(value, weight, capacity):

    n = len(value) - 1
    m = [[-1]*(capacity + 1)
    for _ in range(n + 1)]
    for w in range(capacity + 1):
        m[0][w] = 0
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weight[i] > w:
                m[i][w] = m[i - 1][w]
            else:

                m[i][w] =
                max(m[i - 1][w -
                weight[i]] + value[i],
                 m[i - 1][w])
    return m[n][capacity]
```

Gambar 17. *Knapsack* dengan *Bottom-Up*

Algoritma diatas juga memiliki kompleksitas O(NK), dapat terlihat diatas bahwa algoritma tersebut menghitung nilai dari bawah ke atas untuk mendapatkan solusi.

IV. Kesimpulan

Dynamic Programming sangat diperlukan untuk menyelesaikan algoritma yang memiliki kompleksitas tinggi, dan kebanyakan algoritma dengan kompleksitas tinggi adalah algoritma yang menggunakan rekursif atau pemanggilan fungsi dia sendiri. Konsep dari *Dynamic Programming* adalah untuk menghindari terjadinya pengulangan perhitungan pada sub masalah yang sudah diselesaikan dengan menyimpan nilai tersebut ke sebuah struktur data dan dipanggil kembali jika diperlukan. Struktur data yang digunakan biasanya berupa

V. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa karena hanya atas berkat-nya makalah berjudul "Pemanfaatan *Dynamic Programming* Pada Kompleksitas Algoritma" dapat selesai. Penulis juga berterima kasih kepada

dosen mata kuliah IF2120 Matematika Diskrit, pada Semester I Tahun 2019/2020 atas ilmu yang telah diberikan kepada penulis.

Daftar Pustaka

[1] https://blog.usejournal.com/top-50-dynamic-programming-practice-problems-4208fed71aa3, diakses pada 5 Desember 2019 pada pukul 17:30

[2] https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html, diakses pada 5 Desember 2019 pada pukul 17:40

[3] https://www.geeksforgeeks.org/quick-sort/, diakses pada 5 Desember 2019 pada pukul 18:00

[4] https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm, diakses pada 5 Desember 2019 pada pukul 18:10

[5] https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/, diakses pada 5 Desember 2019 pada pukul 18:30

[6] https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/, diakses pada 5 Desember 2019 pada pukul 17:00

[7] http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf, diakses pada 5 Desember 2019 pada pukul 17:05

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, 5 Desember 2019

Vincent Tanjaya, 13518133