Pendekatan Algoritma Divide & Conquer dan Dynamic Programming pada *Binomial Coefficient Problem*

Wisnu Aditya Samiadji 13519093
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13519093@std.stei.itb.ac.id

***Abstrak*— Algoritma bukanlah 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 *Divide & Conquer (DnC)* dan *Dynamic Programming (DP)*. Tentu cara implementasi ini memiliki kompleksitas waktu yang berbeda, bergantung pada persoalan yang akan dipecahkan. *Binomial Coefficient* sendiri merupakan topik yang cukup dikenal di kalangan ilmuwan. Binomial Coefficient merupakan turunan dari penggunaan Ekspansi Binomial. Pada karya tulis ini, penulis akan mencoba melakukan pendekatan DnC dan DP untuk menghitung *Binomial Coefficient* ini dan membandingkannya. Perbandingan akan dilakukan dengan tingkat kompleksitas masing-masing algoritma.**

Kata Kunci—Dynamic Programming, Divide & Conquer, DP, DnC, Binomial Coefficient, Kompleksitas

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 dengan ukuran sangat besar.

Binomial Coefficient pada hakikatnya adalah bilangan yang menunjukkan banyaknya cara untuk mengambil k objek dari sekumpulan n benda. Binomial Coefficient juga sering dikenal sebagai kombinasi dan biasa ditulis ${}_nC_k$. Bilangan ini merupakan bagian dari penjabaran ekpansi Binomial.

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + \ldots + \binom{n}{n}a^0 b^n$$

$$\text{where } \binom{n}{r} = {}^nC_r = \frac{n!}{r!(n-r)!}$$

Gambar 1. Ekspansi Binomial (Sumber : https://www.onlinemathlearning.com/terms-binomial-expansion.html

Binomial Coefficient memiliki beberapa kegunaan, diantaranya adalah menentukan perkiraan nilai dari aljabar tertentu dan jumlah dari beberapa deret tak hingga.

Penulis berasumsi bahwa pendekatan dengan Dynamic Programming akan lebih optimal. Untuk mengetesnya, penulis akan mengimplementasikan kedua pendekatan untuk Binomial Coefficient dengan bahasa C++ yang akan ditampilkan pada bagian lain karya tulis ini.

II. LANDASAN 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 perangkat 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.

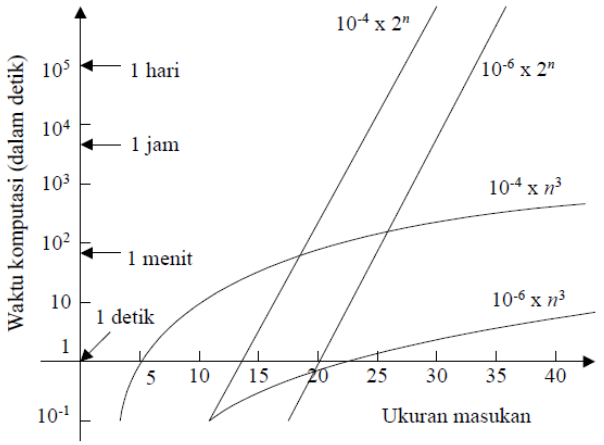
Kompleksitas algoritma biasa dinotasikan dalam bentuk fungsi yang menunjukkan relasi antara ukuran masukan dengan banyaknya operasi fundamental yang akan dilakukan algoritma.

Kompleksitas yang demikian disebut dengan kompleksitas waktu atau *time complexity*. Selain waktu, kompleksitas algoritma juga dapat dapat diteliti dari banyaknya ruang yang dibutuhkan untuk 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:

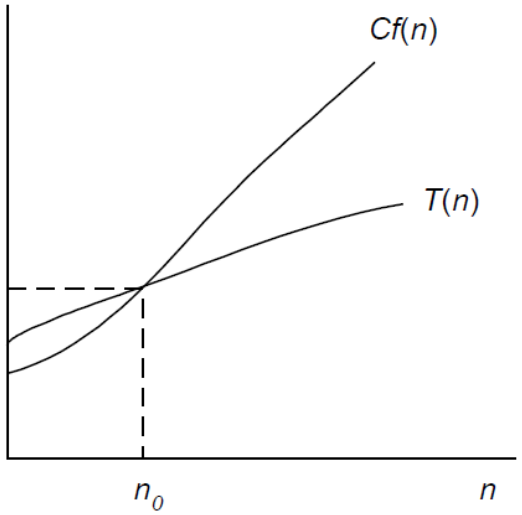
A. $\boldsymbol{T_{max}(n)}$ merupakan kompleksitas waktu algoritma untuk kasus terburuk (*worst case*).
B. $\boldsymbol{T_{min}(n)}$, merupakan kompleksitas waktu algoritma untuk kasus terbaik (*best case*).
C. $\boldsymbol{T_{avg}(n)}$, merupakan kompleksitas waktu algoritma untuk semua kasus secara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

Fungsi kompleksitas waktu **T(n)** biasa dinyatakan dengan notasi asimptotik (*Asymptotic Notations*). Misalkan, kita memiliki algoritma dengan $T_1(n) = n^3 + 2n^2 + 4$. Jika dibandingkan dengan algoritma lain yang memiliki $T_2(n) = n^3 + 5n + 3$, 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)

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

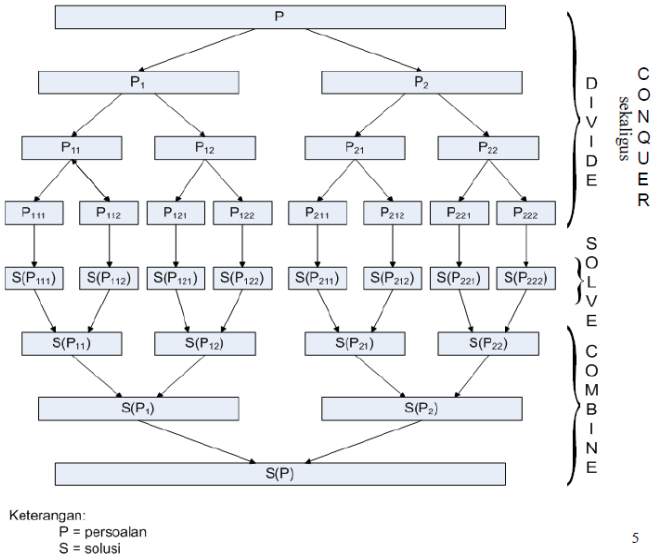
Gambar 4. *Big-Oh Umum (Sumber : http://courses.cs.washington.edu/courses/cse373/14wi/lecture4.pdf)*

B. *Divide & Conquer*

Divide & Conquer dahulunya merupakan nama dari sebuah strategi militer, yaitu *devide et impera*. Strategi tersebut merupakan fundamental dari *Divide & Conquer*. *Divide & Conquer* secara bahasa terdiri atas kata *divide* (bagi) dan *conquer* (taklukkan). Ada 3 bagian pada Divide & Conquer, yaitu:

- *Divide*: Membagi persoalan menjadi beberapa sub-masalah yang memiliki kemiripan dengan persoalan semula namun berukuran lebih kecil (idealnya berukuran hampir sama)

- *Conquer (solve)*: Menyelesaikan masing-masing sub-masalah (secara langsung atau secara rekursif).
- *Combine* : Menggabungkan solusi masing-masing sub-masalah sehingga membentuk solusi persoalan semula.

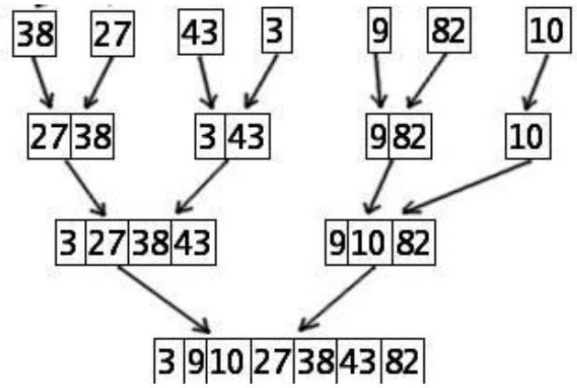


Gambar 5. Skema Divide & Conquer (Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Divide-and-Conquer-(2018).pdf)

Objek persoalan yang dapat di-*divide* biasanya merupakan masukan berukuran n yang berupa larik, tabel, matriks, eksponen, dan lainnya. Secara luas, algoritma Divide & Conquer diungkapkan sebagai algoritma rekursif.

Contoh permasalahan yang dapat menggunakan algoritma DnC adalah Merge Sort pada sebuah tabel. Misalkan ada sebuah larik berukuran n yang berisi bilangan bulat. Langkah DnC dalam Merge Sort tersebut adalah :

1. Jika n = 1, maka tabel sudah terurut.
2. Jika n > 1, maka bagi tabel Amenjadi dua bagian, bagian kiri dan bagian kanan, masing-masing bagian berukuran n/2 elemen [DIVIDE].
3. Secara rekursif, terapkan DnC pada masing-masing bagian [CONQUER].
4. Gabung hasil pengurutan kedua bagian sehingga diperoleh tabel yang terurut [COMBINE].



Gamar 6. Combine pada Merge Sort (Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Divide-and-Conquer-(2018).pdf)

Kompleksitas waktu Merge Sort ini adalah $T(n) = n^2 \log n$.

C. *Dynamic Programming (DP)*

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)

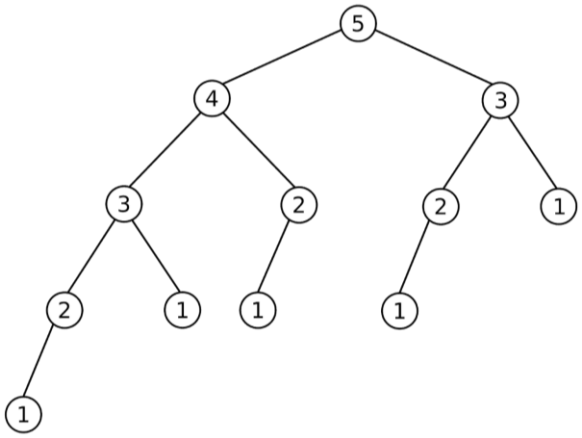
Substruktur yang optimal merupakan karakteristik paling penting dalam DP, bahkan rekursi secara umum. Jika sebuah permasalahan dapat dipecahkan dengan rekursi, besar kemungkinan terdapat substruktur yang optimal di dalamnya.

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

A. *Penyelesaian Binomial Coefficient dengan Divide & Conquer (DnC)*

Pada dasarnya, persoalan *Binomial Coefficient* dengan Divide & Conquer dipecahkan dengan memanfaatkan karakteristik uniknya. Untuk setiap 0 <= k < n, berlaku :

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Gambar 9. Karakteristik Unik Binomial Coefficient 1 (Sumber : milik pribadi)

Selain itu, untuk mengoptimalkan eksekusi, penulis juga memanfaatkan satu karakteristik lagi, yaitu untuk setiap 0<=k<n, berlaku:

$$\binom{n}{k} = \binom{n}{n-k}$$

Gambar 10. Karakteristik Unik Binomial Coefficient 2 (Sumber : milik pribadi)

Implementasi algoritmanya adalah sebagai berikut:

```
ull solveDnC(ull n, ull k){
        if(n - k < k){
                return solveDnC(n, n - k); //Optimisasi
        }
        else if (k == 0 || n == k){
                return 1; //C(n, 0) = 1, //C(n, n) = 1
        }
        else{
          return solveDnC (n - 1, k - 1) + solveDnC (n - 1, k);
        }
}
```

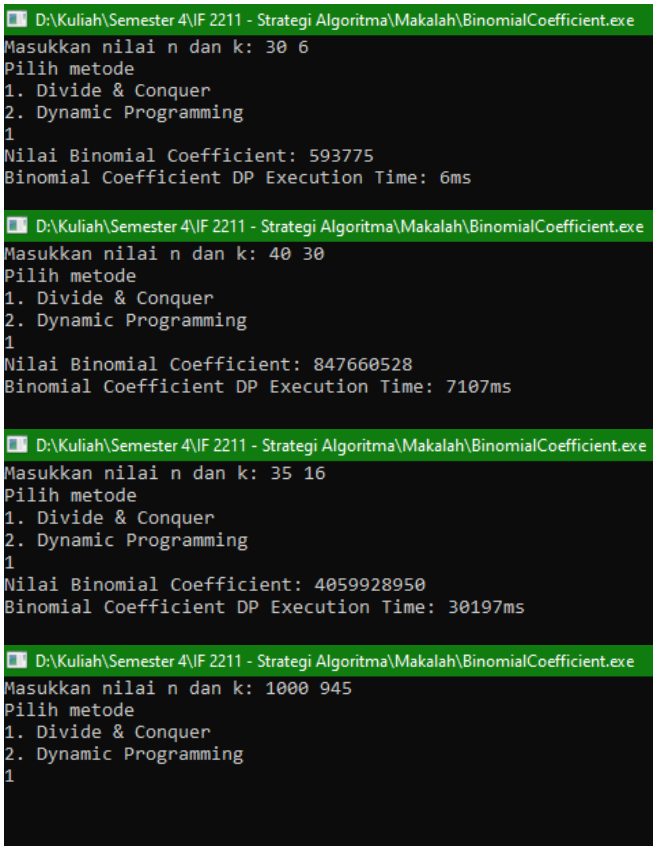
Pada implementasi, karakteristik yang ada pada Gambar 6

```
if(method == 1){
        clock_t time = clock();
        cout << "Nilai Binomial Coefficient: " << solveDnC(n, k) << endl;
        time = clock() - time;
        cout << "Binomial Coefficient DP Execution Time: " << time << "ms\n";
                return 0;
```

digunakan saat n – k < k. Dengan optimisasi ini, jumlah operasi rekursif yang perlu dieksekusi oleh program dapat berkurang. Main program yang digunakan pada kedua algoritma akan menghitung *execution time* dari fungsi ketika dijalankan. Pada karya tulis ini, penulis akan menggunakan 3 test case yang sama untuk kedua algoritma.

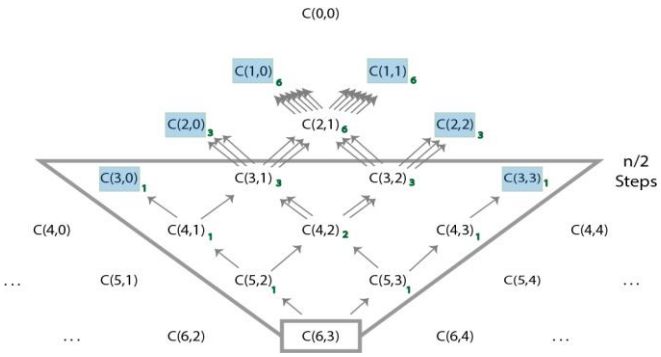
n	k	Execution Time(ms)
30	6	6
40	30	7107
35	16	30197
1000	945	>300000

Tabel 3.1 Tabel n, k, dan *execution time* untuk Divide & Conquer



Gambar 11. Eksekusi Test Case DnC(Sumber : milik pribadi)

Pada eksekusi test case ketiga, program tidak berhasil mengeluarkan output apapun setelah lebih dari 5 menit. Jika dianalisis dari kompleksitasnya, T(n) dari pendekatan DnC adalah: $T(n) \approx 2^{n/2}$.



Gambar 12. Ilustrasi Langkah Divide & Conquer pada *Binomial Coefficient* (Sumber : milik pribadi)

Hal ini terjadi karena saat setiap kali eksekusi, program akan memanggil dirinya sendiri sembari mengurangi nilai n, hingga n = 1. Lalu, dengan optimasi, program hanya perlu memanggil hingga n = n / 2. Setaip kali n dikurangi, program memanggil 2 kali fungsi tersebut. Sehingga, jumlah operasi yang dieksekusi adalah maksimum $2^{n/2}$. Jika dihitung dengan menimbang bahwa C++ dapat menjalankan 2×10^8 operasi dalam 1 sekon, maka program ini membutuhkan 1.63×10^{142} sekon (sangat-sangat lama).

B. Penyelesaian Binomial Coefficient dengan Dynamic Programming (DP)

Cara dasar dalam pendekatan DP ini hampir sama dengan yang digunakan pada pendekatan DnC. Bedanya tentu terletak pada memoisasi. Pada pendekatan DP, terdapat sebuah larik yang menyimpan nilai dari n dan k yang telah dieksekusi oleh program.

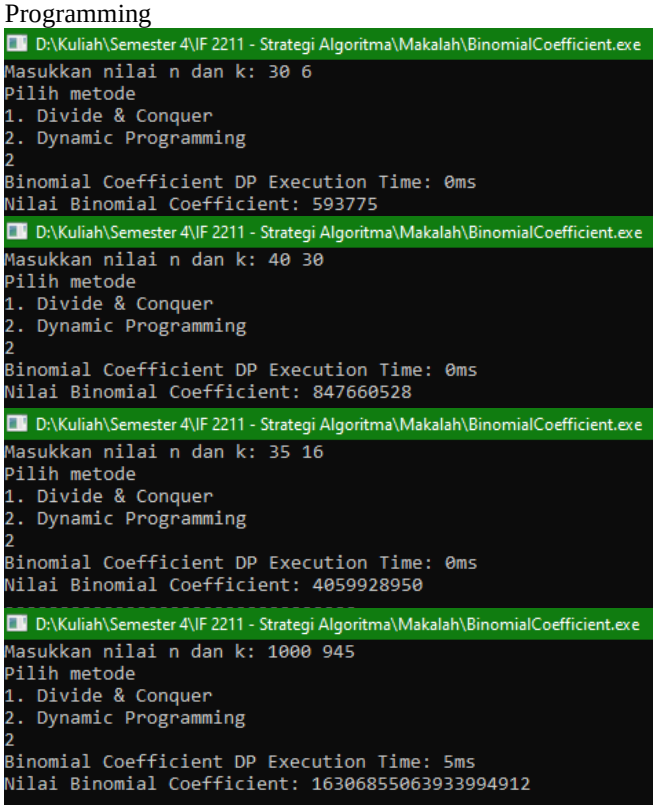
Implementasi algoritmanya adalah sebagai berikut:

```
ull binocoeffdp[5001][5001];
//2. Dynamic Programming
void solveDP(ull n, ull k){
        ios_base::sync_with_stdio(0);
cin.tie(0); cout.tie(0);
        clock_t time = clock();
        if(n - k < k)       solveDP(n, n - k);
        //Optimisasi
        else{
                for(int i = 1; i <= n; i++){
        binocoeffdp[i][0] = 1;        //C(n, 0) =
1               binocoeffdp[i][i] = 1;
        //C(n, n) = 1
        binocoeffdp[i][1] = i;        //C(n, 1) =
n
}
                for(int i = 1; i <= n; i++){
                        for(int j = 1; j <
i; j++){
        binocoeffdp[i][j] = binocoeffdp[i -
        1][j - 1] + binocoeffdp[i - 1][j];
                        }
                }
                cout << "Binomial Coefficient
DP Execution Time: " << clock() - time << "ms\n";
        }
        return;
}
```

```
if(method == 2){
                solveDP(n, k);
                cout << "Nilai Binomial
Coefficient: " << binocoeffdp[n][k];
                return 0;
        }
DP Execution Time: " << time << "ms\n";
                return 0;
        }
```

n	k	Execution Time(ms)
30	6	0
40	30	0
35	16	0
1000	945	5

Tabel 3.2 Tabel n, k, dan *execution time* untuk Dynamic Programming



Gambar 13. Eksekusi Test Case DP(Sumber :milik pribadi)

Dari data pada tabel 3.1 dan 3.2 serta Gambar 11 dan 13, dapat dilihat bahwa algoritma DP mengungguli DnC dalam menyelesaikan *Binomial Coefficient*. Pada saat n membesar atau dan k mendekati n / 2, kecepatan eksekusi algoritma DnC menurun drastis, sedangkan algoritma DP stabil. Jika dianalisis dari kompleksitas waktunya, algoritma DP memiliki $T(n) = n \times k$ dan kompleksitas ruang $O(k)$.

IV. KESIMPULAN

Dari pembahasan pada bab sebelumnya, hipotesis penulis benar bahwa algoritma DP akan lebih efektif untuk masalah ini. Meskipun algoritma DP memakan memori, penulis merasa jika memori yang digunakan kurang berarti jika pengguna akan menggunakan algoritma ini dalam skala besar, yang biasa dilakukan pada computer-komputer *high-level.* Perlu diingat bahwa apa yang terjadi pada tes juga dibatasi oleh kemampuan komputer milik penulis.

VIDEO LINK AT YOUTUBE

https://youtu.be/l8jqLc1UeLw

V. 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 Ibu Nur Ulfa Maulidevi sebagai dosen pengampu Mata Kuliah Strategi Algoritma. Terakhir, penulis ingin berterima kasih kepada tim pembuat soal UAS Strategi Algoritma yang membantu saya mendapat ide untuk membuat makalah ini.

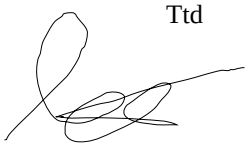
REFERENSI

[1] https://mathworld.wolfram.com/BinomialCoefficient.html (Diakses pada Mei 2021)
[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Divide-and-Conquer-(2018).pdf (Diakses pada Mei 2021)
[3] *https://simpleprogrammer.com/guide-dynamic-programming/* (Diakses pada Mei 2021)
[4] http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf (Diakses pada Mei 2021)
[5] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian1.pdf (Diakses pada Mei 2021)
[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian2.pdf (Diakses pada Mei 2021)
[7] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian3.pdf (Diakses pada Mei 2021)
[8] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian4.pdf (Diakses pada Mei 2021)
[9] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf (Diakses pada Mei 2021)
[10] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian2.pdf (Diakses pada Mei 2021)
[11] https://www.wolframalpha.com/ (Diakses pada Mei 2021)
[12] https://kullabs.com/class-12/mathematics-1/binomial-theorem/application-of-binomial-theorem-for-approximation?type= (Diakses pada Mei 2021)

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 Mei 2021
Ttd



Wisnu Aditya Samiadji 13519093