Analisa Kompleksitas Penyelesaian *Nim Game* Menggunakan Sprague-Grundy Theorem dan Dynamic Programming

Frederik Imanuel Louis - 13520163[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13520163@std.stei.itb.ac.id

***Abstrak*—Nim Game adalah permainan kombinatorial dimana pemain mengambil objek dari tumpukan secara bergantian dan pemain yang tidak dapat mengambil objek dinyatakan kalah. Pemenang dari Nim Game dapat ditentukan dari kondisi awal permainan menggunakan berbagai metode, misalnya brute force, Nim-Sum, atau Sprague-Grundy Theorem. Aplikasi Sprague-Grundy Theorem dalam menyelesaikan Nim Game menggunakan Dynamic Programming ditemukan dapat menyelesaikan Nim Game dengan kompleksitas waktu asimptotik yang berbanding lurus pada banyaknya tumpukan dan logaritmik pada banyak kemungkinan objek yang diambil pemain dikali ukuran tumpukan terbesar.**

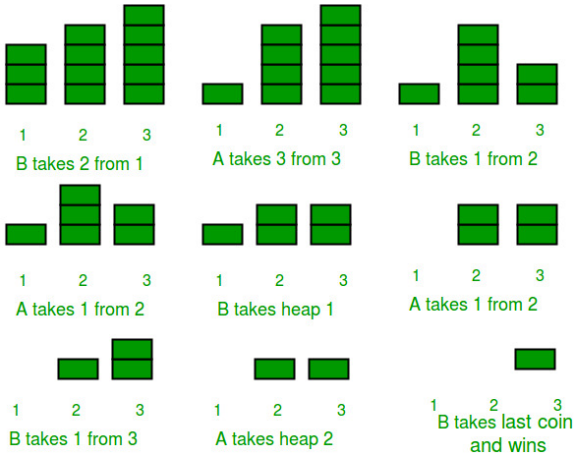
***Kata Kunci*—Algoritma, Kompleksitas, Memoisasi, Mex**

I. PENDAHULUAN

Perkembangan komputer telah membuat kemajuan besar pada bidang ilmu Matematika. Kekuatan komputasi komputer yang sekarang ini jauh melebihi kemampuan manusia membuka peluang yang besar bagi simulasi permasalahan matematis dalam komputer. Salah satu permasalahan yang sering disimulasikan komputer adalah permasalahan Teori Permainan Kombinatorial.

Teori Permainan Kombinatorial adalah subbagian dari bidang ilmu Kombinatorika yang mempelajari permainan yang dimainkan dua orang dengan informasi sempurna, dimana tiap pemain mengetahui seluruh kondisi dan keadaan permainan. Permainan kombinatorial juga tidak memiliki elemen keberuntungan, yang berarti semua langkah yang dilakukan pemain bersifat deterministik dan pasti, misalnya tanpa lemparan koin atau dadu. Selain itu, permainan kombinatorial akan berhenti setelah berhingga langkah, dimana salah satu pemain dinyatakan menang atau permainan dinyatakan seri. Oleh karena itu, pemenang permainan kombinatorial yang dimainkan dengan strategi optimal dapat ditentukan dari kondisi awal permainan. Hal tersebut dapat dilakukan salah satunya dengan mensimulasikan permainan dengan komputer dengan mengecek semua keadaan permainan yang mungkin dan menentukan strategi optimal tiap pemain.

Nim Game merupakan salah satu contoh permainan kombinatorial, dimana tiap pemain secara bergiliran mengambil sejumlah objek dari tumpukan, dan pemain yang tidak dapat mengambil objek dinyatakan kalah.



Gambar 1.1 Contoh Permainan Nim
Sumber: geeksforgeeks.org

Seperti permainan kombinatorial lainnya, pemenang Nim Game juga dapat ditentukan dari kondisi awal permainan. Terdapat berbagai metode untuk menentukan hal tersebut, misalnya brute force, Nim-Sum Theory, dan Sprague-Grundy Theorem. Pada makalah ini, akan dibahas aplikasi Sprague-Grundy Theorem menggunakan Dynamic Programming dalam menyelesaikan Nim Game, beserta analisis kompleksitasnya.

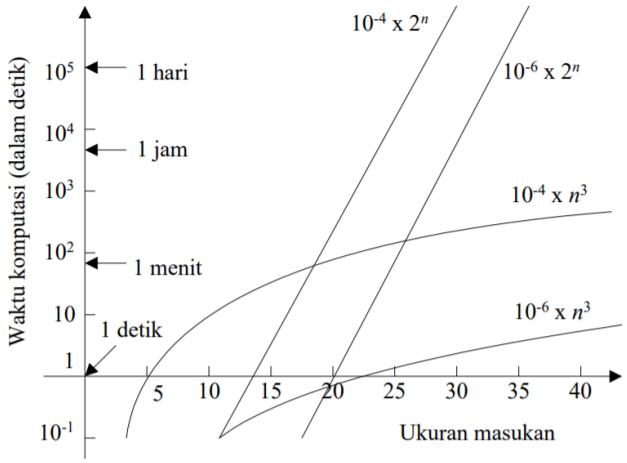
II. LANDASAN TEORI

A. Kompleksitas Algoritma

Kompleksitas Algoritma adalah suatu hal yang mengukur seberapa efisien suatu algoritma bekerja dalam menyelesaikan suatu permasalahan, dari segi memori dan/atau waktu relatif terhadap besarnya masukan. Algoritma dengan kompleksitas yang lebih rendah berarti algoritma tersebut membutuhkan sumber daya yang lebih kecil saat algoritma dijalankan.

Kompleksitas waktu diukur dari banyaknya operasi komputasi primtif yang dilakukan suatu algoritma. Operasi tersebut diantaranya operasi baca (input), operasi tulis (output), operasi aritmetika, operasi pengisian nilai (asignment), operasi perbandingan, operasi pengaksesan elemen, pemanggilan fungsi, alokasi memori, dan sejenisnya. Komputer modern dapat melakukan sekitar 10^9 operasi primitif dalam satu detik.

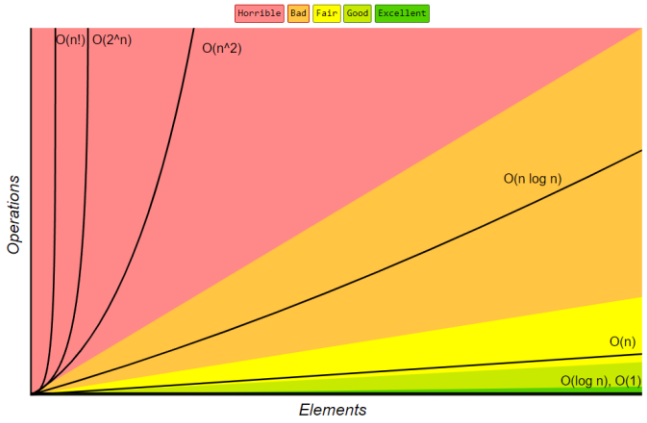
Kompleksitas memori suatu algoritma adalah banyaknya memori yang diperlukan algoritma relatif terhadap besarnya masukan input. Algoritma yang membutuhkan alokasi memori yang lebih sedikit, misalnya hanya menggunakan satu atau dua variabel, akan memiliki kompleksitas memori yang lebih rendah dari algoritma yang membutuhkan alokasi memori besar, misalnya array yang menyimpan semua input user. Batas memori yang dapat digunakan algoritma dibatasi oleh hardware komputer, yaitu besarnya memori yang dimiliki komputer.



Gambar 2.1 Grafik perbandingan ukuran masukan terhadap waktu komputasi algoritma
Sumber: Slide Materi Kuliah

Kompleksitas waktu asimptotik meninjau bagaimana kompleksitas waktu tumbuh jika ukuran input meningkat. Kompleksitas asimptotik inilah yang biasanya digunakan untuk mengukur efisiensi algoritma, sebab kita pada umumnya tidak membutuhkan informasi mengenai jumlah persis operasi yang dilakukan algoritma. Dengan komputer yang dapat melakukan hingga 10^9 operasi dalam satu detik, perbedaan satu atau dua operasi tidak akan terasa pada waktu jalan algoritma. Namun, untuk masukan yang besar, kita ingin meninjau apakah suatu algoritma masih dapat digunakan. Hal tersebut dapat kita perkirakan mengunakan kompleksitas waktu asimptotik. Salah satu notasi yang digunakan untuk menyatakan kompleksitas waktu asimptotik adalah notasi *O-Besar* atau *Big-O*. Secara sederhana, suatu algoritma yang memiliki kompleksitas waktu asimptotik $O(f(N))$ dapat berjalan dalam waktu berorde paling besar $f(N)$, dengan N adalah besarnya masukan.

Notasi tersebut memungkinkan kita dengan mudah menentukan apakah suatu algoritma dapat dijalankan untuk N tertentu. Misalnya algoritma Floyd-Warshall yang memiliki kompleksitas waktu asimptotik $O(N^3)$ dengan besar input $N = 10^5$ akan membutuhkan hingga $(10^5)^3 = 10^{15}$ operasi. Meningat bahwa komputer dapat melakukan 10^9 operasi dalam satu detik, kita dapat melihat bahwa algoritma tersebut dapat membutuhkan hingga $10^{15-9} = 10^6$ detik ≈ 12 hari sehingga algoritma tersebut kurang cocok digunakan. Untuk algoritma Dijkstra yang memiliki kompleksitas waktu asimptotik $O(N^2)$, operasi yang dilakukan dapat mencapai $(10^5)^2 = 10^{10}$ operasi, yang membutuhkan waktu sekitar 10 detik saja. Hal tersebut menunjukkan bahwa algoritma Dijkstra lebih cocok digunakan dalam kasus tersebut.



Gambar 2.2 Grafik perbandingan efisiensi beberapa notasi Big-O
Sumber: bigocheatsheet.com

Secara umum, algoritma yang memiliki kompleksitas waktu asimptotik dengan orde yang lebih kecil merupakan algoritma yang lebih efisien. Algoritma yang paling efisien akan memiliki kompleksitas waktu asimptotik $O(1)$ yang berarti lama jalan algoritma tidak dipengaruhi besarnya masukan. Beberapa kompleksitas waktu asimptotik yang umum ditemui pada skema algoritma dasar dapat dilihat pada tabel berikut.

Tabel 2.1 Kompleksitas beberapa algoritma umum

Kompleksitas	Algoritma
$O(1)$	Push/Pop Stack
$O(\log(N))$	Binary search, Heap insert/delete
$O(sqrt(N))$	Cek keprimaan suatu bilangan
$O(N)$	Linear search, DFS, BFS
$O(N\log(N))$	Merge sort, Heap sort
$O(N^2)$	Bubble sort, Selection sort, Dijkstra
$O(N^3)$	Floyd-Warshall, Perkalian Matriks
$O(2^N)$	Fibonacci tanpa memoisasi
$O(N!)$	Enumerasi semua permutasi

B. Nim Game

Permainan Kombinatorial adalah permainan yang dimainkan oleh dua pemain dimana tiap pemain bermain secara bergantian. Tiap pemain memiliki informasi sempurna, yang berarti tiap pemain mengetahui seluruh keadaan permainan. Permainan tersebut juga memiliki properti deterministik, yang berarti tidak ada elemen keberuntungan (misalnya pelemparan koin atau dadu), dan permainan akan selesai dalam berhingga langkah. Beberapa contoh permainan kombinatorial adalah Catur, Othello, dan Nim Game.

Nim Game merupakan permainan dimana dua orang secara bergantian mengambil objek dari satu atau lebih tumpukan. Dalam tiap giliran, pemain dapat mengambil objek sebanyak yang telah ditentukan sebelumnya, dan pemain yang tidak dapat mengambil objek dari tumpakan dinyatakan kalah. Salah satu varian Nim Game klasik adalah Nim dengan satu tumpukan dimana tiap pemain dapat mengambil sebanyak minimal satu dan maksimal N objek dari tumpukan, dan pemain yang tidak

dapat mengambil objek dinyatakan kalah. Bagi varian ini, terdapat solusi umum bahwa jika tumpukan awal merupakan kelipatan $N + 1$, maka pemain kedua (mulai pada giliran kedua) akan menang, dan pemain pertama akan menang jika tumpukan awal bukan kelipatan $N + 1$. Hal ini mudah dibuktikan karena tiap tumpukan bukan kelipatan $N + 1$, pemain yang sedang bermain dapat membuat tumpukan menjadi kelipatan $N + 1$, dan seterusnya sampai tumpukan habis (perhatikan bahwa hanya pemain tersebut yang dapat membuat tumpukan habis).

Pada makalah ini, akan dibahas varian Nim yang lebih umum, dimana terdapat T tumpukan, dan tiap pemain dapat mengambil sejumlah $s_1, s_2, s_3, \ldots,$ atau s_N buah objek dari satu tumpukan, serta pemain yang tidak dapat mengambil objek dari tumpukan dinyatakan kalah.

C. Sprague-Grundy Theorem

Sprague-Grundy Theorem menyatakan bahwa semua permainan kombinatorial dimana kedua pemain bermain secara optimal ekuivalen dengan Nim Game yang dimainkan dengan satu tumpukan, dimana pemenang permainan dapat ditentukan menggunakan Grundy Number.

Grundy number atau Nimber adalah suatu bilangan yang merepresentasikan state game terkecil yang tidak bisa dicapai dari suatu state. Pada Nim Game, Nimber suatu state adalah minimum excludent (mex) atau bilangan bulat nonnegatif terkecil yang tidak dapat dicapai dari state tersebut.

Misalkan kita memiliki permainan Nim satu tumpukan dimana tiap pemain dapat mengambil sebanyak 1,4, atau 9 objek dari tumpukan, dengan tumpukan awal berisi 12 objek. Nimber dari state awal permainan dihitung dengan mencari mex dari Nimber state (12-1), (12-4), dan (12-9), yaitu mex(N(11), N(8), N(3)) dimana N(i) menyatakan Nimber state i. Pencarian Nimber state yang lain kemudian dilakukan secara rekursif dengan base case N(0)=0.

```cpp
#include<bits/stdc++.h>
using namespace std;

int N=12;
int take[3]={1,4,9};

int mex(int n){
    if(n==0)return 0;
    vector<int> values;
    for(auto i: take){
        if(n-i>=0)values.push_back(mex(n-i));
    }
    int ret=0;
    sort(values.begin(),values.end());
    for(auto i: values)if(ret==i)ret++;
    return ret;
}

int main(){
    for(int i=1;i<=N;i++)
        printf("Nimber %d adalah: %d\n",i,mex(i));
}
```

Gambar 2.3 Implementasi Mex dalam C++
Sumber: Dokumen Pribadi

Pemenang dari Nim Game tersebut dapat diketahui dari Nimber yang kita peroleh. Sprague-Grundy Theorem menyatakan jika Nimber adalah nol, maka pemain kedua akan menang, dan jika Nimber bukan nol, maka pemain pertama akan menang.

```
Nimber 1 adalah: 1
Nimber 2 adalah: 0
Nimber 3 adalah: 1
Nimber 4 adalah: 2
Nimber 5 adalah: 0
Nimber 6 adalah: 1
Nimber 7 adalah: 0
Nimber 8 adalah: 1
Nimber 9 adalah: 2
Nimber 10 adalah: 0
Nimber 11 adalah: 1
Nimber 12 adalah: 0
```

Gambar 2.4 Contoh Perhitungan Nimber
Sumber: Dokumen Pribadi

Untuk permainan Nim dengan banyak tumpukan, Sprague-Grundy Theorem menyatakan bahwa kita dapat mengetahui Nimber dari keseluruhan permainan dengan melakukan exclusive or (xor) pada semua nilai Nimber masing-masing tumpukan. Misalnya kita memiliki permainan yang sama, tetapi dengan 3 tumpukan dengan ukurang 12, 8, dan 9. Maka, Nimber seluruh permainan adalah N(12) xor N(8) xor N(9) yaitu 0 xor 1 xor 2 = 3. Maka, karena Nimber permainan bukan nol, kita mendapatkan bahwa pemain pertama dapat menang dengan strategi optimal. Perhatikan pula bahwa hal tersebut berarti semua permainan Nim dengan dua tumpukan yang memiliki banyak objek yang sama akan selalu memiliki Nimber nol. Pemain kedua dapat selalu menang dengan mencerminkan gerakan pemain pertama dari satu tumpukan ke tumpukan lainnya. Dengan itu, saat pemain pertama hendak mengambil objek dari tumpukan, kedua tumpukan selalu memiliki jumlah objek yang sama sehingga hanya ada dua kemungkinan, yaitu ia dapat mengambil objek dari kedua tumpukan, atau ia tidak dapat mengambil objek dari kedua tumpukan. Jika ia dapat mengambil objek dari kedua tumpukan, maka pemain kedua selalu dapat mencerminkan langkah pemain pertama pada tumpukan yang lain, dan jika ia tidak dapat mengambil objek lagi, maka pemain pertama kalah. Hal tersebut mendemonstrasikan salah satu strategi yang mungkin untuk dua tumpukan yang sama, dan Sprague-Grundy Theorem menjamin adanya strategi serupa bagi pemain pertama jika Nimber permainan tidak nol, dan bagi pemain kedua jika Nimber permainan nol.

D. Dynamic Programming

Dynamic Programming (DP) adalah teknik algoritma yang sering digunakan untuk menyelesaikan masalah optimisasi atau perhitungan. Teknik DP pada umumnya tidak menciptakan suatu solusi baru bagi suatu permasalahan, melainkan menggunakan metode yang sudah ada, misalnya backtracking atau complete search, dan mengoptimasi kompleksitas waktu algoritma agar solusi tersebut dapat digunakan untuk menyelesaikan permasalahan dengan masukan yang relatif besar. Beberapa permasalahan klasik yang dapat diselesaikan dengan DP adalah General Coin Change Problem, Max Range Sum, Longest Increasing Subsequence, Knapsack, Travelling Salesman Problem, dan Fibonacci dengan Memoisasi.

Dynamic Programming bekerja dengan cara membagi permasalahan menjadi submasalah yang lebih kecil, dan menyelesaikan masing-masing submasalah tersebut. Setelah

submasalah diselesaikan, jawaban submasalah akan disimpan dalam suatu tabel memoisasi. Hal tersebut sangat penting karena pada umumnya, banyak submasalah dari suatu masalah yang berhimpitan sehingga memoisasi menghilangkan keperluan perhitungan ulang submasalah tersebut. Tabel memoisasilah yang memungkinkan optimasi masif bagi solusi yang sudah ada.

Teknik DP terbagi menjadi dua cabang, yaitu Top-Up DP dan Bottom-Up DP. Top-Up DP pada umumnya menggunakan fungsi yang dipanggil secara rekursif yang mengisi tabel memoisasi sesuai kebutuhan. Hal tersebut memungkinkan ada elemen tabel memoisasi yang kosong saat pemanggilan rekursif selesai. Bottom-Up DP mengisi tabel memoisasi secara keseluruhan, mulai dari submasalah yang paling kecil hingga masalah yang kita ingin selesaikan terisi di tabel memoisasi. Dari pengisian tabel memoisasi, terlihat bahwa Top-Up DP pada umumnya lebih cepat karena tidak menghitung nilai semua elemen tabel memoisasi, tetapi beberapa permasalahan tertentu hanya dapat diselesaikan dengan Bottom-Up DP.

Untuk mengillustrasikan cara kerja DP, kita akan meninjau aplikasi Top-Down DP pada penyelesaian masalah berikut: "Carilah banyak cara melempar dadu 6 sisi satu kali atau lebih sehingga jumlah mata dadu yang muncul pada semua pelemparan berjumlah n modulo $10^9 + 7$ dengan $n \leq 10^5$ ". Permasalahan dapat diselesaikan secara rekursif dengan mengecek banyak cara memperoleh $n-1, n-2, n-3, n-4, n-5$ dan $n-6$ mata dadu dan menjumlahkannya.

```cpp
#include<bits/stdc++.h>
using namespace std;

const int Mod=1e9+7;

long long count(long long n){
    if(n<0)return 0;
    if(n==0)return 1;
    long long ret=0;
    for(int i=1;i<7;i++){
        ret+=count(n-i);
        ret%=Mod;
    }
    return ret;
}

int main(){
    long long n; cin>>n;
    cout<<count(n)<<endl;
}
```

Gambar 2.5 Penyelesaian Rekursif Murni dalam C++
Sumber: Dokumen Pribadi

Namun, solusi rekursif murni tersebut menciptakan dua submasalah untuk setiap submasalah, sehingga program memiliki kompleksitas $O(2^n)$. Kompleksitas tersebut tidak dapat digunakan untuk menyelesaikan permasalahan dimana n mendekati 10^5. Teknik DP dapat mengambil solusi tersebut dan menyelesaikannya dalam $O(N)$.

```cpp
#include<bits/stdc++.h>
using namespace std;

const int Mod=1e9+7;
long long dp[100003];

long long count(long long n){
    if(n<0)return 0;
    if(n==0)return 1;
    if(dp[n]!=-1)return dp[n];
    long long &ret=dp[n];
    ret=0;
    for(int i=1;i<7;i++){
        ret+=count(n-i);
        ret%=Mod;
    }
    return ret;
}

int main(){
    memset(dp,-1,sizeof(dp));
    long long n; cin>>n;
    cout<<count(n)<<endl;
}
```

Gambar 2.6 Top-Down DP dalam C++
Sumber: Dokumen Pribadi

Terlihat bahwa solusi Top-Down DP dan solusi rekursif sebenarnya tidak berbeda jauh, cukup ditambahkan memoisasi pada array dp dan pengecekan memo saat pemanggilan fungsi. Namun, perubahan sederhana tersebut mampu mengubah kompleksitas secara drastis. Terlihat bahwa tiap isi tabel memoisasi hanya dihitung maksimal satu kali, sehingga kompleksitas waktu solusi DP tersebut adalah $O(N)$. Optimasi tersebut cukup untuk menyelesaikan permasalahan dengan nilai N mendekati 10^5.

III. Analisa Kompleksitas Penyelesaian Nim Game Dengan Sprague-Grundy Theorem dan Dynamic Programming

A. Implementasi Penyelesaian Nim Game dengan Sprague-Grundy Theorem dan Dynamic Programming

Pertama-tama, untuk menyelesaikan Nim Game dengan N tumpukan, kita cukup memodifikasi implementasi pencarian nilai Nimber dengan mencari nilai exclusive or (xor) dari Nimber setiap tumpukan. Kemudian, kita cukup mengecek apakah Nimber seluruh permainan adalah nol atau bukan. Jika Nimber adalah nol, maka pemain kedua menang, dan pemain pertama menang jika sebaliknya.

```cpp
#include<bits/stdc++.h>
using namespace std;

// State awal Nim Game
const int N=6;
int heap[N]={11,15,14,10,23,8};
const int M=5;
int take[M]={1,4,9,12,16};

// Perhitungan nimber secara rekursif
int mex(int n){
    if(n==0)return 0;
    vector<int> values;
    for(auto i: take){
        if(n-i>=0)values.push_back(mex(n-i));
    }
    int ret=0;
    sort(values.begin(),values.end());
    for(auto i: values)if(ret==i)ret++;
    return ret;
}

int main(){
    int ans=0;
    // xor semua nimber
    for(int i=0;i<N;i++){
        int temp=mex(heap[i]);
        ans=ans^temp;
        printf("Nimber tumpukan %d objek: %d",heap[i],temp);
    }
    if(ans)cout<<"Pemain pertama menang";
    else cout<<"Pemain kedua menang";
    cout<<endl;
}
```

Gambar 3.1 Implementasi Sprague-Grundy Theorem dalam menyelesaikan Nim Game dalam C++
Sumber: Dokumen Pribadi

Kemudian, untuk mengoptimasi algoritma, kita bisa menerapkan teknik top-down dynamic programming pada fungsi mex dengan melakukan memoisasi pada tiap pemanggilan fungsi mex.

```cpp
#include<bits/stdc++.h>
using namespace std;

// State awal Nim Game
const int N=6;
int heap[N]={11,15,14,10,23,8};
const int M=5;
int take[M]={1,4,9,12,16};

//Tabel memoisasi
int nimber[100];

// Perhitungan Nimber dengan teknik DP
int mex(int n){
    if(n==0)return 0;
    // Pengecekan tabel memoisasi
    if(nimber[n]!=-1)return nimber[n];
    vector<int> values;
    for(auto i: take){
        if(n-i>=0)values.push_back(mex(n-i));
    }
    // Pengisian tabel memoisasi
    nimber[n]=0;
    sort(values.begin(),values.end());
    for(auto i: values)if(nimber[n]==i)nimber[n]++;
    return nimber[n];
}

int main(){
    memset(nimber,-1,sizeof(nimber));
    int ans=0;
    // xor semua tumpukan
    for(int i=0;i<N;i++){
        int temp=mex(heap[i]);
        ans=ans^temp;
        printf("Nimber tumpukan %d objek: %d\n",heap[i],temp);
    }
    if(ans)cout<<"Pemain pertama menang";
    else cout<<"Pemain kedua menang";
    cout<<endl;
}
```

Gambar 3.2 Penambahan teknik DP pada solusi
Sumber: Dokumen Pribadi

```
Nimber tumpukan 11 objek: 1
Nimber tumpukan 15 objek: 0
Nimber tumpukan 14 objek: 1
Nimber tumpukan 10 objek: 0
Nimber tumpukan 23 objek: 3
Nimber tumpukan 8 objek: 1
Pemain pertama menang
```

Gambar 3.3 Output program solusi Nim Game
Sumber Dokumen Pribadi

B. Kompleksitas Waktu Penyelesaian Nim Game dengan Sprague-Grundy Theorem dan Dynamic Programming

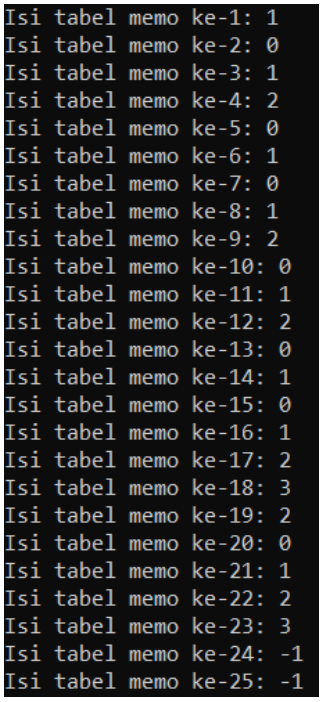
Untuk menganalisa kompleksitas waktu implementasi pada Gambar 3.2, kita akan menelusuri alur program dari fungsi main. Pertama, program mengisi tabel memoisasi dengan nilai -1. Jika tabel memoisasi memiliki ukuran T, maka akan dilakukan T buah proses penulisan pada tabel. Proses tersebut memerlukan kompleksitas waktu asimptotik $O(T)$.

Kemudian, program akan mencari nilai Nimber tiap tumpukan dengan memanggil fungsi mex. Pada fungsi mex, jika banyak objek di tumpukan sudah ada di tabel memoisasi, maka fungsi akan langsung mengeluarkan nilai, sehingga hanya terjadi satu proses. Jika nilai belum ada di tabel memoisasi, maka fungsi mex akan memanggil dirinya secara rekursif untuk mencari Nimber dari tiap state yang mungkin dicapai. Pemanggilan mex akan dilakukan sebanyak M kali dimana M adalah banyaknya kemungkinan objek yang mungkin diambil pemain dari satu tumpukan. Setelah itu, fungsi mex akan melakukan sorting pada semua nilai Nimber yang mungkin dicapai dan mencari Nimber dari tumpukan tersebut. Sorting yang dilakukan oleh C++ STL memerlukan kompleksitas waktu asimptotik $O(M \log(M))$ dengan asumsi worst-case, yaitu ada M nilai pada array values. Maka, kompleksitas waktu asimptotik dari satu perhitungan Nimber adalah $O(M \log(M))$ ditambah M kali pemanggilan fungsi mex lainnya.

Namun, karena tiap nilai Nimber maksimum dicari satu kali (untuk mengisi tabel memoisasi), maka kita dapat mengasumsikan pada kasus worst-case dimana semua nilai Nimber perlu dicari, akan terjadi secara total K buah perhitungan Nimber dimana K adalah banyak objek maksimum dari satu tumpukan. Hal tersebut benar karena nilai Nimber tertinggi yang akan kita cari adalah dari tumpukan terbesar. Maka, total kompleksitas waktu asimptotik dari perhitungan semua Nimber adalah $O(KM \log(M))$.

Pada fungsi main, dilakukan pula exclusive xor pada semua Nimber yang sudah diperoleh. Karena xor merupakan operasi primitif (bitwise), maka kompleksitas waktu asimptotik yang diperlukan adalah $O(N)$ dengan N adalah banyaknya tumpukan.

Dengan demikian, kita memperoleh total kompleksitas waktu asimptotik yang diperlukan program penyelesaian Nim Game adalah $O(T + KM \log(M) + N)$ dimana T adalah ukuran tabel memoisasi, K adalah banyak objek pada tumpukan terbesar, M adalah banyaknya kemungkinan objek yang dapat diambil oleh pemain, dan N adalah banyaknya tumpukan dalam keadaan awal permainan.



Gambar 3.3 Isi tabel memoisasi pada solusi DP
Sumber: Dokumen Pribadi

Namun, perhatikan bahwa tabel memoisasi yang digunakan maksimum hanya sampai K, yaitu banyak objek pada tumpukan terbesar. Hal ini berarti jika nilai K sudah pasti, atau jika tabel memoisasi dialokasi dapat secara dinamis, kompleksitas waktu dapat dioptimasi menjadi $O(K + KM\log(M) + N)$ atau dapat disederhanakan menjadi $O(KM\log(M) + N)$. Namun, tidak semua kasus input memungkinkan hal tersebut (optimasi bersifat spesifik), sehingga implementasinya tidak akan dibahas pada makalah ini.

C. Kompleksitas Memori Penyelesaian Nim Game dengan Sprague-Grundy Theorem dan Dynamic Programming

Untuk menganalisa kompleksitas memori implementasi pada Gambar 3.2, kita cukup mengecek ukuran tiap variabel yang kita gunakan. Variabel heap yang menyimpan besar tiap tumpukan memiliki ukuran N dimana N adalah banyak tumpukan. Variabel take yang menyimpan semua kemungkinan banyak objek yang dapat diambil oleh pemain memiliki ukuran M dimana M adalah banyak kemungkinan objek yang mungkin diambil pemain. Tabel memoisasi memiliki ukuran T. Dengan itu, kompleksitas memori asimptotik yang diperlukan adalah $O(N + M + T)$.

Namun, seperti pada analisa kompleksitas waktu, ukuran tabel memoisasi juga dapat dioptimasi untuk menghemat memori apabila ukuran tumpukan terbesar sudah pasti, atau jika tabel memoisasi dialokasi secara dinamis. Dengan itu, kompleksitas memori asimptotik menjadi $O(N + M + K)$ dimana K adalah banyak objek pada tumpukan terbesar. Namun, tidak semua kasus input memungkinkan hal tersebut (optimasi bersifat spesifik), sehingga implementasinya tidak akan dibahas pada makalah ini.

IV. KESIMPULAN

Kompleksitas waktu asimptotik dari penyelesaian Nim Game menggunakan Sprague-Grundy Theorem dan Dynamic Programming adalah $O(T + KM\log(M) + N)$ dimana T adalah ukuran tabel memoisasi, K adalah banyak objek pada tumpukan terbesar, M adalah banyaknya kemungkinan objek yang dapat diambil oleh pemain, dan N adalah banyaknya tumpukan dalam keadaan awal permainan. Pada keadaan tertentu, algoritma solusi dapat dioptimasi menjadi $O(KM\log(M) + N)$. Kompleksitas memori asimptotik dari solusi yang sama adalah $O(N + M + T)$, dan pada keadaan tertentu, dapat dioptimasi menjadi $O(N + M + T)$.

V. UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada:

1. Tuhan Yang Maha Esa,
2. orang tua penulis,
3. Bapak dan Ibu dosen pengampu mata kuliah Matematika Diskrit IF2120, dan
4. teman-teman penulis

yang telah mendukung penulis selama proses penyusunan makalah ini.

REFERENSI

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf. Diakses pada 1 Desember 2021,
[2] https://www.bigocheatsheet.com/. Diakses pada 1 Desember 2021
[3] https://devopedia.org/algorithmic-complexity. Diakses pada 1 Desember 2021
[4] Sriram, Pranav A. (2014), Olympiad Combinatorics.
[5] https://www.geeksforgeeks.org/introduction-to-combinatorial-game-theory. Diakses pada 1 Desember 2021.
[6] https://www.geeksforgeeks.org/combinatorial-game-theory-set-2-game-nim/. Diakses pada 1 Desember 2021
[7] Dutta, Prajit K. (1999), *Strategies and games: theory and practice*, MIT Press, ISBN 978-0-262-04169-0.
[8] https://www.geeksforgeeks.org/combinatorial-game-theory-set-4-sprague-grundy-theorem/. Diakses pada 1 Desember 2021
[9] http://www.cut-the-knot.org/Curriculum/Games/Grundy.shtml. Diakses pada 2 Desember 2021
[10] Halim, S., Halim, F., Skiena, S. S., & Revilla, M. A. (2013). *Competitive programming 3*. Lulu Independent Publish.
[11] Wengrow, J. (2020). *A Common-Sense Guide to Data Structures and Algorithms*. Pragmatic Bookshelf.

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, 10 Desember 2021

Frederik Imanuel Louis, 13520163