Analisis Kompleksitas Metode Rekursi: Perbandingan Tail dan Direct Recursion pada Bahasa Pemrograman Prolog

Edward Terrance Lie - 13525127
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: edwardtl18k@gmail.com , 13525127@std.stei.itb.ac.id

***Abstract*—PROLOG merupakan salah satu bahasa yang menggunakan paradigma deklaratif dengan sub *logical programming* berbasis fakta dan aturan (*knowledge based*). Dalam pengaplikasiannya, bahasa ini tidak memiliki struktur kontrol prosedural seperti perulangan sehingga harus sepenuhnya bergantung pada mekanisme rekursi. Namun, penggunaan rekursi konvensional (*direct* recursion) sering menghadapi masalah inefisiensi pada penggunaan memori sehingga dapat menyebab *error stack overflow*. Penelitian ini bertujuan untuk mengevaluasi dan melakukan analisis komparatif terhadap rekursi konvensional (*direct recursion*) dan *tail recursion* untuk mendapatkan metode terbaik dalam mengimplementasikan rekursi di dalam bahasa PROLOG. Penelitian ini dilakukan dengan melakukan pengujian terhadap rekursi konvensional (*direct* recursion) dan *tail recursion* dengan tiga pengujian, yaitu *test case* aritmatika dan manipulasi list secara *single reversal* maupun *double reversal*. Hasil pengujian membuktikan bahwa metode *tail recursion* memiliki keunggulan baik efisiensi waktu eksekusi maupun efisiensi memori sehingga metode ini dapat diandalkan dalam mengelola data berskala besar. Di sisi lain, *direct recursion* memiliki permasalahan pada *stack overflow* saat menangani input list dengan elemen sebanyak $N = 10^7$ akibat pengalokasian *local stack frame* secara terus menerus tanpa dapat didealokasikan sebelum mencapai kasus basis.**

***Keywords*—Prolog, Tail Recursion, Direct Recursion, Tail Cal Optimization**

I. Pendahuluan

Dalam dunia pemrograman, paradigma pemrograman dapat memberikan analisis dan cara pandang khusus dalam menyelesaikan masalah tertentu. Salah satu paradigma pemrograman adalah paradigma deklaratif. Paradigma ini fokus dalam menyelesaikan masalah dan hal yang perlu dilakukan, bukan bagaimana cara komputer menyelesaikannya. Paradigma ini dijalankan dengan mengekespresikankan logika, fungsi, maupun basis data program untuk menyelesaikan tugas tertentu. Salah satu cabang dari pemrograman dengan paradigma deklaratif adalah *logical programming. Logical Programming* merupakan paradigma yang berdasaran fakta dan aturan (*knowledge based*) untuk menyelesaikan masalah melalui inferensi logika dan *reasoning* [1].

Salah satu bahasa pemrograman yang menggunakan paradigma ini adalah bahasa PROLOG. Dalam PROLOG, penyelesaian masalah tidak dilakukan secara sekuensial, melainkan dengan memberikan *query* kemudian dievaluasi dengan *query resolution*. *Query resolution* dilakukan dengan inferensi logika terhadap deklarasi sejumlah fakta serta sejumlah aturan mengenai objek dan relasi antarpredikat yang telah terdefinisi sebelumnya. Selain itu, proses ini hanya dapat melakukan operasi aritmatik dan logika beserta pemanggilan aturan berdasarkan fakta yang diberikan.

Karena PROLOG merupakan bahasa pemrograman deklaratif, bahasa ini tidak memiliki struktur kontrol seperti perulangan dalam menyelesaikan suatu masalah. Oleh karena itu, rekursi merupakan satu satunya mekanisme yang dapat digunakan dalam melakukan perulangan, iterasi, serta struktur data yang kompleks. Rekursi adalah proses mendefinisikan objek dalam terminologinya sendiri [2]. Dalam penggunaannya predikat rekursi didefinisikan menjadi dua bagian, yaitu Basis dan Langkah Rekursi (memanggil dirinya sendiri). Basis merupakan bagian di dalam predikat rekursi yang berperan sebagai bagian yang memberikan sebuah nilai yang terdefinisi secara eksplisit pada predikat rekursif sekaligus sebagai batas akhir dalam sebuah predikat. Selanjutnya, langkah rekursi merupakan proses melaksanakan langkah untuk memanggil dirinya sendiri untuk menemukan nilai predikat pada suatu input dari nilai-nilai lainnya pada input yang lebih kecil.

Meskipun rekursi menjadi mekanisme fundamental dalam bahasa PROLOG, rekursi standar (*direct recursion*) memiliki keterbatasan dalam efisiensi memori. Dalam penerapannya, *direct recursion* pada rekursi linear memerlukan memori sebesar $O(n)$. Pemakaian memori ini didasarkan pada konvensi pemanggilan predikat/fungsi pada pemrograman. Pada pemanggilan tersebut, setiap langkah rekursif akan membentuk *stack frame* baru di dalam *local stack* untuk menyimpan variabel yang yang belum selesai dievaluasi. Alokasi ini akan terus diakumulasi hingga mencapai kasus basis. Oleh karena itu, saat rekursi dilakukan dengan kedalaman rekursi yang sangat besar, seperti $N > 100.000$, rekursi dapat menyebabkan kegagalan fatal berupa

stack overflow karena penggunaan memori telah melebihi batas yang telah dialokasikan oleh *compiler*.

Dari permasalahan tersebut, terdapat satu teknik alternatif dalam menggunakan rekursi, yaitu dengan menggunakan *tail recursion*. Teknik ini tetap menggunakan mekanisme dasar rekursi, yaitu memanggil dirinya sendiri. Namun, dalam penggunaanya *tail recursion* umumnya memiliki tambahan sebuah variabel yang ditugaskan sebagai sebuah variabel akumulator jika operasi yang digunakan adalah operasi aritmatika atau struktur data abstrak maupun kompleks list. Dengan menggunakan *Tail Recursion*, jika compiler memiliki fitur berupa TCO (*Tail Call Optimization*), efisiensi memori dapat berubah dari sebesar $O(n)$ menjadi $O(1)$. Dengan demikian, permasalahan seperti *stack overflow* dapat dihindari karena *compiler* melakukan optimisasi dengan melewati proses pembuatan *local stack frame* baru dan hanya mengubah *stack frame* saat ini dengan argumen yang baru sehingga dapat dianggap sebagai *looping* secara kode mesin. Namun, optimasi ruang ini tetap dapat membawa *trade-off* berupa kompleksitas waktu asimtotik program yang tetap berjalan secara linear $O(n)$. Di sisi lain, *tail recursion* juga memberikan komputasi tambahan akibat perilaku list dalam PROLOG karena program hanya dapat mengambil komponen *head*. Oleh karena itu, program memerlukan komputasi tambahan untuk membalik urutan elemen dalam akumulator ke kondisi semula.

Makalah ini bertujuan untuk melakukan analisis komparatif terhadap perilaku rekursi terutama rekursi linear dalam bentuk *direct recursion* dan *tail recursion* di dalam bahasa pemrograman PROLOG. Selanjutnya, penelitian ini juga diarahkan untuk menguji batas skalabilitas sistem rekursi terhadap risiko *stack overflow* yang mungkin terjadi saat masukan data berskala besar, sekaligus melakukan validasi efisiensi TCO (*Tail Call Optimization)* berdasarkan waktu eksekusi dan kebutuhan alokasi memori *stack frame* lokal.

II. Dasar Teori

A. Bahasa Pemrograman PROLOG

PROLOG adalah singkatan dari PROgramming in LOGic. Bahasa ini merupakan salah satu bahasa yang menggunakan paradigma deklaratif dengan subparadigma *logical programming*. Bahasa ini dikembangkan berdasarkan Kalkulus Predikat Orde Pertama (*Relational Logic*) [3].

Pemrograman dalam prolog meliputi tiga hal, yaitu :

- Deklarasi sejumlah fakta mengenai suatu objek dan relasinya
- Deklarasi sejumlah aturan mengenai suatu objek dan relasinya
- Pertanyaan (*query*) mengenai suatu objek dan relasinya.

Fakta dalam PROLOG merupakan sebuah predikat yang menggambarkan sebuah hubungan atau properti suatu objek yang dianggap benar. Argumen dalam fakta harus bersifat konstanta. Selanjutnya, aturan (*rules*) dalam PROLOG merupakan sebuah sistem untuk mendeduksi fakta baru yang tidak terdaftar di dalam basis data secara eksplisit [3]. Aturan dalam prolog dibentuk dalam bentuk implikasi sebagai contoh, “A :- B,C,D.” Pada contoh ini, A bernilai true ketika B dan C dan D juga bernilai true. Kemudian, *query* dalam PROLOG merupakan tujuan (*goals*) yang ingin dicapai oleh pengguna dengan melakukan request untuk melakukan *query resolution* atau inferensi untuk menentukan apakah *query* tersebut bernilai *true* atau *false*.

Di luar tipe data primitif, PROLOG memiliki struktur data kompleks berupa list. List dalam PROLOG direpresentasikan sebagai *singly linked list* karena list diakses dengan memecah komponen menjadi *Head* dan *Tail* (sisa list setelah *head*). Karakteristik pengaksesan yang bersifat *head-only* ini menyebabkan operasi penambahan elemen di depan bersifat konstan $O(1)$, sedangkan operasi di ujung belakang (*append*) memerlukan penelusuran seluruh elemen list sehingga bersifat linear $O(n)$.

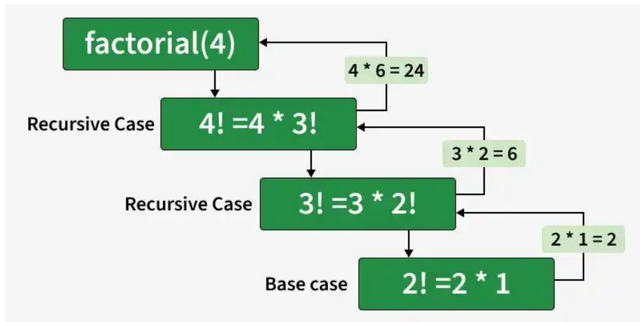
Karena PROLOG merupakan bahasa pemrograman deklratif, bahasa ini tidak memiliki sistem kontrol perulangan sehingga setiap aturan yang ingin dikerjakan secara berulang harus dideduksi secara rekursif. Selain itu, bahasa PROLOG juga memiliki sebuah sistem tambahan dalam membantu rekursi, yaitu *cut*. *Cut* digunakan untuk memberikan kontrol atas backtracking yang dilakukan di dalam rekursi. *Cut* umumnya digunakan agar program lebih efisien serta membatasi pencarian solusi lebih dari satu.

B. Rekursi Linear dan Metode Tail Recursion

Rekursi adalah proses mendefinisikan objek dalam terminologinya sendiri [2]. Rekursi umumnya digunakan dalam memecahkan masalah kompleks menjadi masalah yang lebih kecil [4]. Dalam rekursi, terdapat dua bagian, yaitu basis dan langkah rekursi. Basis merupakan bagian dalam predikat rekursi yang berperan sebagai pemberi nilai eksplisit sekaligus batas akhir sebuah predikat dalam melakukan langkah rekursi. Selanjutnya, langkah rekursi adalah bagian dari predikat yang digunakan untuk memecah masalah kompleks menjadi masalah yang lebih kecil dengan cara memanggil dirinya sendiri untuk mendapatkan nilai predikat dari suatu *input* yang lebih kecil. Penggunaan kedua komponen ini wajib saling berhubungan karena saat langkah rekursi tidak menemukan kasus basis, rekursi akan terus terjadi dan dapat menyebabkan *error* berupa *stack overflow*.

Dalam PROLOG, rekursi digunakan sebagai satu-satunya teknik dalam melakukan iterasi matematis, iterasi logis, maupun manipulasi struktur data kompleks seperti list. Pembahasan di dalam penelitian ini dibatasi secara khusus pada lingkup rekursi linear (*linear recursion*), yaitu struktur rekursi yang hanya menghasilkan maksimal satu cabang pemanggilan dirinya sendiri di setiap langkah komputasi. Dalam implementasinya, terdapat dua pendekatan rekursi dalam *direct recursion* yang memiliki dampak berbeda terhadap penggunaan memori, yaitu *non-tail recursion* dan *tail rercursion.* Namun, demi menyeleraskan dengan variabel yang diuji dan konvensi penulisan pada judul makalah ini,

penggunaan istilah *direct recursion* merujuk pada *non-tail recursion* (rekursi standar) dan istilah *tail recursion* tetap pada padanannya.



Gambar. 1. Visualisasi Rekursi
Sumber : https://www.geeksforgeeks.org/python/recursion-in-python/

Direct Recursion (rekursi standar) memiliki operasi tambahan seperti operasi aritmatika atau unifikasi (melakukan instantiasi pada variabel yang belum diketahui) setelah langkah rekursi selesai dieksekusi. Pada metode ini, program akan terus membentuk *stack frame* baru di dalam *local stack* unuk menyimpan variabel serta operasi yang belum dieksekusi. Oleh karena itu, *compiler* memerlukan *local stack frame* dalam jumlah yang sangat besar sehingga dapat berakibat pada *error stack overflow* karena penggunaan memori telah melebihi batas yang telah dialokasikan oleh *compiler*. Berikut adalah contoh penerapan *direct recursion* dalam menentukan panjang sebuah list :

```
len([],0):-!.
len([H|T], Hasil):- len(T,HasilSebelum),
Hasil is HasilSebelum+1.
```

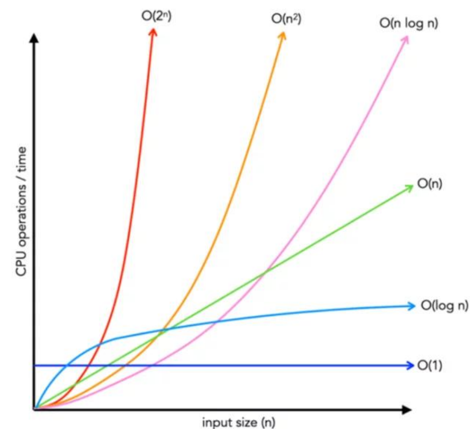
Di sisi lain, *Tail Recursion* dalam PROLOG adalah tipe pemanggilan rekursi yang tidak memiliki operasi tambahan yang dilakukan setelah pemanggilan rekursi (langkah rekursi). Pada metode ini, beberapa *compiler* PROLOG dapat melakukan *Tail Call Optimization* (TCO). Fitur ini akan mengoptimasi bentuk rekursi menjadi bentuk iteratif atau biasa disebut *looping* yang memiliki efiesiensi memori yang lebih tinggi karena tidak membutuhkan *local stack frame* tambahan. Namun, dalam beberapa kasus, *tail recursion* memang membutuhkan variabel tambahan yang berfungsi sebagai akumulator untuk menyimpan hasil sementara di dalam predikat tersebut. Berikut ini adalah contoh penerapan *tail recursion* dalam menentukan panjang sebuah list :

```
% Format predikat
len_tail(List,Akumulator,PanjangList).
% Pemanggilan :
len_tail(List,0,PanjangList).
% Deklarasi Rule :
len_tail([],PanjangList,PanjangList):-!.
len_tail([_|T],Akumulator,PanjangList):-
Next is Akumulator + 1,
len_tail(T,Next,PanjangList).
```

C. *Kompleksitas Algoritma*

Kompleksitas Algoritma adalah besaran yang digunakan untuk menerangkan model abstrak dalam pengukuran waktu atau ruang [5]. Kompleksitas algoritma dibagi menjadi dua bagian, yaitu komplesitas waktu (*time complexity*) dan kompleksitas ruang (*space complexity*). Algoritma yang efisien dinilai dari seberapa minimum waktu yang diperlukan dan memori yang dibutuhkan dalam menjalankan algoritma tersebut [5].

Pada kompleksitas algoritma, kompleksitas waktu diukur berdasarkan jumlah tahapan komputasi yang dilakukan di dalam algoritma sebagai fungsi dari ukuran masukan N. Sedangkan kompleksitas ruang, diukur dari penggunaan memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi atau predikat dari ukuran masukan N. Konsep ini digunakan dalam menentukan seberapa cepat waktu komputasi meningkat ketika masukan (n) mulai membesar. Konsep ini diperlukan untuk memudahkan para *programmer* dalam memahami dan melakukan generalisasi algoritma baru terhadap ukuran masukan data yang besar. Salah satu yang sering digunakan adalah Notasi Big-O yang berfungsi sebagai salah satu kompleksitas waktu asimtotik untuk kasus terburuk.



Gambar. 2. Grafik Kompleksitas Algoritma
Sumber : https://msgprogramator.sk/en/big-o-notation/

Pada pemrosesan predikat rekursif, kompleksitas ruang tidak hanya ditentukan oleh ukuran struktur data, tetapi juga alokasi memori di dalam *local stack*. Pada metode rekursif konvensional (*direct rercursion* dengan *non-tail recursion*), setiap pemanggilan rekursif akan membentuk *local stack frame* baru sehingga menghasilkan kompleksitas ruang secara linear, yaitu $O(n)$. Namun, saat menerapkan metode rekursi *Tail Recursion* pada compiler yang mendukung fitur TCO (*Tail Call Optimization*), rekursi dapat dioptimasi menjadi stack frame yang iteratif sehingga memangkas kompleksitas ruang menjadi konstan $O(1)$.

III. HASIL DAN PEMBAHASAN

A. Spesifikasi Lingkungan Eksperimen Rekursi

Pengujian dilakukan dengan menggunakan konfigurasi perangkat keras berupa Laptop ROG Strix G16JU dengan CPU Intel Core I5-13450HX 4,6 GHz dengan 10 Core dan 16 Thread dengan RAM 16 GB 4800 MHz DDR5. Selanjutnya, pengujian dilaksanakan pada OS Linux Mint 22.3 dengan versi kernel Ubuntu 24.0.4 dan menggunakan compiler SWI-Prolog dengan versi 10.0.2.

Dalam pengujian, terdapat predikat bawaan SWI Prolog yang digunakan untuk melakukan pengukuran. Predikat tersebut adalah statistics/2 untuk mendapatkan jumlah alokasi memori pada stack yang digunakan (dalam satuan KiloBytes) dan mendapatkan waktu yang digunakan (dalam satuan milisekon) untuk menyelesaikan komputasi.

B. Media Pengujian Komputasi Rekursi

Dalam pengujian ini, terdapat beberapa unit tes dalam Bahasa Prolog yang digunakan sebagai media pembanding dalam menguji kompleksitas waktu dan ruang *pada direct recursion dan tail recursion*. Berikut adalah potongan kode utama dalam penerapan metode *Direct Recursion dan Tail Recursion*

- *Test Case* 1: Sumasi (aritmatika murni)

Direct Recursion

```
sum_dir(0,0):-!.
sum_dir(N, Hasil) :-
    N > 0, Next is N - 1,
    sum_dir(Next, Resultafter),
    Hasil is Resultafter + N.
```

Tail Recursion

```
sum_tail(N,Hasil):-
    sum_tails(N,0,Hasil).

sum_tails(0,Hasil,Hasil):- !.

sum_tails(N,Acc,Hasil):-
    Next is N-1, NewAcc is Acc + N,
    sum_tails(Next,NewAcc,Hasil).
```

- *Test Case* 2 : Manipulasi list dan unifikasi list dengan mengalikan setiap anggota list dengan angka 2 (*Single Reversal*)

Direct Recursion

```
list_double_direct([], []).
list_double_direct([Head|Tail],
FinalList) :-
    list_double_direct(Tail, Res),
    R is Head*2, FinalList =
[R|Res].
```

Tail Recursion

```
list_double_tail(List, Res) :-
list_double_tailhelp(List, [],
Res).

list_double_tailhelp([], Acc,
Final) :- reverse(Acc, Final).

list_double_tailhelp([Head|Tail],
Acc, Res) :-
    R is Head * 2, NextAcc
=[R|Acc],
    list_double_tail(Tail, NextAcc,
Res).
```

- *Test Case* 3 : Manipulasi List dan Unifikasi List dengan melakukan pengelompokan angka ganjil genap ke dalam list angka ganjil atau genap (*Double Reversal*)

Direct Recursion:

```
filter_direct([], [], [], 0).
filter_direct([H|T], Evens, Odds,
TotalCount) :-

filter_direct(T, Even, Odd,
CountBefore),

TotalCount is CountBefore + 1,
((H mod 2)=:=0
    ->  Evens = [H|Even], Odds =
Odd
    ;   Odds = [H|Odd], Evens =
Even
    ).
```

Tail Recursion :

```
filter_tail(List, Evens, Odds) :-
    filter_tail(List, [], [],
Evens, Odds).

filter_tail([], AccEven, AccOdd,
FinalEven, FinalOdd) :-
reverse(AccEven, FinalEven),
reverse(AccOdd, FinalOdd).

filter_tail([H|T], AccEven, AccOdd,
Evens, Odds) :- ((H mod 2)=:=0),
!,NextAccEven = [H|AccEven],
filter_tail(T, NextAccEven, AccOdd,
Evens, Odds).
```

```
filter_tail([H|T], AccEven, AccOdd,
Evens, Odds) :-
    ((H mod 2)=:=1),NextAccOdd =
[H|AccOdd]
            filter_tail(T, AccEven,
NextAccOdd, Evens, Odds).
```

C. *Hasil Pengujian*

Pada pengujian ini, penggunaan kode sumber untuk setiap *test case* telah dimodifikasi untuk mendapatkan data dalam proses pengujian. Beberapa variabel yang ingin diuji adalah waktu eksekusi dan alokasi memori untuk *local stack*. Pengujian dilakukan dengan melakukan 5 kali *query* terhadap *compiler* SWI-Prolog. Kemudian data yang didapat akan dihitung rata-ratanya dan ditampilkan pada tabel di bawah ini.

Tabel 1 (*Test Case* 1 (Sumasi: Aritmatika Murni))

N	Alokasi Stack Lokal Direct (KB)	Waktu Eksekusi Direct (ms)	Alokasi Stack Lokal Tail (KB)	Waktu Eksekusi Tail(ms)
1.000	93,921	0,489	0,1797	0,313
10.000	937,671	6,03	0,1797	2,579
100.000	9375,171	31,259	0,1797	10,743
1.000.000	93750,171	279,034	0,1797	67,176
10.000.000	937500,171	3286,766	0,1797	528,928

Tabel 2 (*Test Case* 2 (Manipulasi List dan Unifikasi List dengan mengalikan setiap anggota list dengan angka 2))

N	Alokasi Stack Lokal Direct (KB)	Waktu Eksekusi Direct (ms)	Alokasi Stack Lokal Tail (KB)	Waktu Eksekusi Tail(ms)
1.000	109,476	1,5	0,1094	0,457
10.000	1093,851	5,325	0,1094	2,662
100.000	10937,601	48,81	0,1094	8,939
1.000.000	109375,101	706,049	0,1094	93,83
10.000.000	Gagal (Stack Overflow)	Gagal (Stack Overflow)	0,1094	1145,889

Tabel 3 (*Test Case* 3 (Manipulasi List dan Unifikasi List) melakukan pengelompokan angka ganjil genap ke dalam list angka ganjil atau genap))

N	Alokasi Stack Lokal Direct (KB)	Waktu Eksekusi Direct (ms)	Alokasi Stack Lokal Tail (KB)	Waktu Eksekusi Tail(ms)
1.000	140,742	1,336	0,125	0,61
10.000	1406,367	6,626	0,125	2,356
100.000	14062,617	65,522	0,125	14,953
1.000.000	140625,117	780,429	0,125	123,469
10.000.000	Gagal (StackOverflow)	Gagal (StackOverflow)	0,125	1625,492

D. *Pembahasan Hasil Pengujian*

- Tren umum hasil pengujian

Berdasarkan data pengujian, ditemukan perbedaan performa yang sangat signifikan. Pada *direct recursion*, alokasi stack lokal terus meningkat bersamaan dengan meningkatnya jumlah input yang diproses. Dari pola pada hasil pengujian di atas, alokasi stack lokal pada direct recursion memiliki kompleksitas ruang sebesar $O(n)$ karena setiap langkah rekursi yang dilakukan akan membentuk *stack frame* lokal yang baru. Dalam *direct recursion*, setiap *stack frame* lokal perlu tetap dialokasikan karena terdapat operasi yang belum dijalankan setelah langkah rekursi dipanggil sehingga *compiler* harus tetap menyimpan *stack frame* lokal sebanyak N. Berbeda dengan *direct recursion*, *tail recursion* memiliki alokasi stack lokal dengan kompleksitas ruang sebesar $O(1)$. Hal ini dibuktikan dengan tidak adanya pertumbuhan pada alokasi stack lokal meski jumlah masukan terus meningkat. Kondisi ini terjadi karena compiler SWI-Prolog memiliki optimisasi TCO (*Tail Call Optimization*). Optimisasi ini memungkinkan compiler untuk menggunakan *stack frame* lokal yang sama seperti pemanggilan predikat sebelumnya karena pemanggilan predikat rekursi berada di paling akhir sehingga tidak ada operasi data yang diperlukan kembali. Oleh karena itu, data lokal dari predikat sebelumnya tidak diperlukan kembali sehingga memungkinkan *compiler* menggunakan *stack frame* yang sama.

Selain dari sisi kompleksitas ruang, terdapat pula perbedaan signifikan dalam segi kompleksitas waktu. Berdasarkan data pengujian *test case* 1 (sumasi), secara umum fungsi `sum_dir` dan `sum_tail` memiliki kompleksitas waktu $O(n)$ karena operasi pertambahan dilakukan sebanyak n kali. Namun, hasil pengujian menunjukkan bahwa pertumbuhan waktu yang terjadi tidak menghasilkan koefisien $T(n)$ yang konsisten secara sempurna (fluktuasi). Hal ini disebabkan oleh keterbatasan perangkat keras. Saat data komputasi melebihi cache L1 dan L2, CPU akan melanjutkan penggunaan penyimpanan alternatif menuju *cache* L3 dan L4 atau bahkan RAM. Namun, penyimpanan alternatif seperti *cache* L3, L4, bahkan RAM memiliki perbedaan kecepatan yang signifikan dibanding *cache* L1 dan L2. Selain itu, hasil pengujian juga konsisten membuktikan bahwa waktu eksekusi *tail recursion* selalu lebih cepat dibanding *direct recursion*. Hal ini dapat terjadi karena setiap operasi di dalam bahasa mesin memiliki biaya komputasi per elemen yang berbeda. Sebagai contoh, operasi *push* maupun *pop* pada *stack frame* lokal memiliki biaya komputasi yang lebih besar dibanding operasi aritmatika dasar.

- Tren Khusus pada *Direct Recursion*

Pada hasil pengujian dalam *test case* 2 dan 3 dengan input list sebesar $N = 10.000.000$, terjadi sebuah *error* berupa *stack overflow* dengan pesan *error* `Stack Limit (1.0 Gb) Exceeded`. Hal ini terjadi karena langkah rekursi terus dilakukan hingga alokasi memori lokal yang disediakan (1 GB dalam pengujian ini), telah terlewati. Kegagalan ini terjadi karena karakteristik pertumbuhan kompleksitas ruang yang linear sebesar $O(n)$. Berdasarkan tren $N = 1.000.000$, alokasi memori untuk *stack frame* lokal adalah 109,37 MB (*test case 2*) dan 140,625 MB (*test case 3*). Jika dihitung secara teoritis, peningkatan linear sebesar 10 kali lipat dalam jumlah N akan mengakibatkan peningkatan ukuran *stack*

frame lokal menjadi $> 1\ GB$. Karena batas alokasi yang diberikan hanya 1.0 GB, sistem mengalami kehabisan memori untuk menyimpan langkah rekursi berikutnya sebelum mencapai kasus basis sehingga menyebabkan *error stack overflow*. Hal ini menjadi bukti bahwa *direct recursion* memiliki keterbatasan dalam menangani pemrosesan data dalam jumlah besar.

IV. KESIMPULAN

Di dalam bahasa PROLOG khususnya compiler SWI-Prolog, metode *tail recursion* memiliki kelebihan dalam hal kompleksitas algoritma, baik kompleksitas waktu maupun kompleksitas ruang. Implementasi TCO (*Tail Call Optimization*) pada SWI-Prolog berhasil mengubah kompleksitas ruang rekursi linear pada umumnya, yaitu $O(n)$ menjadi $O(1)$. Hal ini terjadi karena *compiler* kembali menggunakan *local stack frame* yang sama akibat ketiadaan operasi yang tertinggal. Di sisi lain, metode *direct recursion* memiliki banyak kekurangan terutama dalam menangani data berskala besar. Hal ini terjadi karena *direct recursion* terus membuat *local stack frame* baru setiap kali terjadi langkah rekursi. Namun, setiap langkah rekursi tidak dapat dioptimasi karena masih ada operasi yang belum dieksekusi sebelumnya sehingga program harus terus menumpuk *stack frame* hingga kasus basis sehingga memiliki kompleksitas ruang $O(n)$. Selain itu, metode *tail recursion* dengan TCO juga selalu memiliki kecepatan eksekusi yang lebih cepat meskipun keduanya memiliki kompleksitas waktu $O(n)$. Hal ini terjadi karena biaya komputasi yang berbeda untuk operasi *push* dan *pop* pada *direct recursion* yang menyebabkan peningkatan waktu eksekusi secara signifikan pada data dengan skala yang lebih besar.

V. LAMPIRAN

Berikut merupakan seluruh kode sumber dan hasil pengetesan yang telah penulis buat

- Spreadsheet hasil pengetesan : https://tinyurl.com/hasiltesdirecttailrecursion26
- Kode Sumber Pengetesan : https://github.com/EdwTorch/Prolog-DirectnTail-Recursion-Test

VI. UCAPAN TERIMAKASIH

Penulis ingin mengucapkan terimakasih kepada Tuhan Yang Maha Esa, orang tua, serta Bapak Ir. Rinaldi Munir, S.T., M.T., dan teman-teman yang telah mendukung saya dalam pembuatan makalah ini sehingga saya dapat menyelesaikannya dengan tepat waktu. Saya juga berterimakasih terkhusus kepada Bapak Rinadi selaku dosen mata kuliah IF1220 Matematika Diskrit Institut Teknologi Bandung atas bimbingannya dan ilmu yang telah diberikan selama satu semester terkait teori dan dasar utama dalam dunia pemrograman mulai dari logika hingga kompleksitas algoritma. Semoga ilmu yang didapatkan dapat terserap dan dapat diimplementasikan dengan baik dalam kehidupan perkuliahan dan pemrograman kedepannya.

REFERENCES

[1] B. Rhani "Introduction of Programming Paradigms", GeeksforGeeks, 2026, https://www.geeksforgeeks.org/system-design/introduction-of-programming-paradigms/ (diakses 17 Juni 2026)

[2] R. Munir, "Barisan, sumasi, rekursi, dan relasi rekurens (Bagian 1)" Homepage Rinaldi Munir, 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/13-Barisan,%20rekursi-dan-relasi-rekurens-(Bagian1)-2026.pdf (diakses 17 Juni 2026)

[3] IF1221 Computational Logic, "Introduction to Prolog", Edunex 2026

[4] Kartik, "Introduction to Recursion", GeeksforGeeks, 2026, https://www.geeksforgeeks.org/dsa/introduction-to-recursion-2/ (diakses pada 18 Juni 2026).

[5] R.Munir "Kompleksitas Algoritma Bagian 1", Homepage Rinaldi Munir 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/25-Kompleksitas-Algoritma-Bagian1-2026.pdf (diakses pada 18 Juni 2026).

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.

Jakarta, 19 Juni 2025

Edward Terrance Lie (13525127)