Studi Perbandingan Algoritma Iteratif dan Rekursif dalam Pengecekan Bilangan Sempurna

Disusun oleh:

Muhammad Rio Adrian	**103012400023**
Naufal Nabiel Hadi	**103012400227**
Sigit Bimantoro	**103012400163**

DAFTAR ISI

BAB I
PENDAHULUAN

1.1 Latar Belakang

Bilangan sempurna (Perfect Number) dalam teori bilangan didefinisikan sebagai bilangan bulat positif yang nilainya sama dengan jumlah pembagi-pembagi positifnya (proper divisors). Definisi ini tidak mencakup bilangan itu sendiri.

1.2 Rumusan Masalah

Studi kasus ini meminta untuk menjawab pertanyaan berikut:

1. Bagaimana fungsi kompleksitas asimtotik T(n) dan Θ(n) untuk kedua algoritma iteratif dan rekursif bekerja dalam pengecekan bilangan sempurna?
2. Bagaimana visualisasi perbedaan waktu eksekusi metode iteratif dan rekursif dengan menggunakan domain input yang sama?

1.3 Manfaat Penelitian

Studi kasus ini memiliki manfaat sebagai berikut:

1. Mendeskripsikan cara kerja fungsi kompleksitas asimtotik T(n) dan Θ(n) bagi kedua algoritma iteratif dan rekursif bekerja dalam pengecekan,
2. Memaparkan visualisasi perbedaan waktu eksekusi metode iteratif dan rekursif dengan menggunakan domain input yang sama.

1.4 Batasan Masalah

Pada laporan ini, para peneliti hanya berfokus pada perbandingan algoritma iteratif dan rekursif dalam kasus pengecekan bilangan sempurna. Algoritna yang digunakan akan ditulis dalam bahas C++ dan memanfaatkan aplikasi bawaan, yaitu Command Prompt. Domain bilangan akan dibatasi pada interval [1,60000].

BAB II
PEMBAHASAN

2.1 Bilangan Sempurna

Suatu bilangan asli dikatakan bilangan sempurna jika dan hanya jika jumlah semua pembagi positif dari n selain n adalan n (Affaf, 2016). Contohnya, 6 dikatakan bilangan sempurna karena jumlah semua faktor atau pembagi habis positif dari 6 selain dirinya sendiri (1,2,3) adalah (1+2+3) = 6 (enam); alias dirinya sendiri. Sebaliknya, 9 bukan merupakan contoh bilangan sempurna karena jumlah semua faktornya, selain dirinya sendiri, (1+3) ≠ 9 (sembilan). Selain itu, 28 adalah bilangan sempurna karena jumlah semua faktornya, selain dirinya sendiri, (1+2+4+7+14) = 28; alias sama dengan dirinya sendiri.

2.2 Deskripsi Algoritma

Kami akan membagi pembahasan deskripsi algoritma menjadi dua bagian, yaitu yang iteratif dan rekursif. Keduanya akan menggunakan algoritma main() yang sama. Berikut adalah algoritma main() yang digunakan,

```cpp
int main() {
    int n1;

    cout << "--------------------------------------" << endl;
    cout << "PENGECEK BILANGAN SEMPURNA (ITERATIF)" << endl;
    cout << "--------------------------------------" << endl;
    cout << "Masukkan bilangan sebagai batas atas (Domain >= 1): ";

    n1 = 60000; // Bilangan yang dipilih

    // Mulai timing
    auto start = std::chrono::high_resolution_clock::now();

    if (bilanganSempurna(n1, n1/2) == n1) {
        cout << n1 << " adalah bilangan sempurna!" << endl;
    } else {
        cout << n1 << " BUKAN bilangan sempurna!" << endl;
    }

    // Akhiri timing
    auto stop = std::chrono::high_resolution_clock::now();
    auto duration =
        std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

    cout << "Waktu eksekusi: " << duration.count()
         << " microseconds" << endl;

    return 0;
}
```

Gambar 1. Fungsi main() iteratif pengecek bilangan sempurna.

main.cpp | rekursif.cpp | rekursif.h | main.cpp | iteratif.cpp | iteratif.h

```cpp
#include "rekursif.h"
#include <iostream>
#include <chrono>

using namespace std;

int main()
{
    int n1;
    cout << "--------------------------------------" << endl;
    cout << "PENGECEK BILANGAN SEMPURNA (REKURSIF)" << endl;
    cout << "--------------------------------------" << endl;
    cout << "Masukkan bilangan sebagai batas atas (Domain >= 1): ";

    n1 = 10000;

    auto startTime = std::chrono::high_resolution_clock::now();

    if (bilanganSempurna(n1, n1/2) == n1) {
        cout << n1 << " adalah bilangan sempurna!" << endl;
    } else {
        cout << n1 << " BUKAN bilangan sempurna!" << endl;
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration =
        std::chrono::duration_cast<std::chrono::microseconds>(endTime - startTime);

    cout << "Waktu eksekusi: " << duration.count()
         << " microseconds" << endl;

    return 0;
}
```

Gambar 2. Fungsi main() rekursif pengecek bilangan sempurna.

2.2.1 Algoritma Iteratif

Algoritma iteratif pengecekan bilangan sempurna hanya terdiri dari satu algoritma looping dan satu algoritma percabangan.

```c
#include "iteratif.h"

int bilanganSempurna(int n1,int n2){

    int sum = 0;
    int i;
    for(i=n2;i>=1;i--){
        if(n1%i==0){
            sum += i;
        }
    }
    return sum;
}
```

Gambar 3. Algoritma iteratif pengecek bilangan sempurna.

Berdasarkan algoritma iteratif di atas, kita akan mendefinisikan variabel *sum* dan menginisialisasikan nilai *sum* dengan 0. Variabel *n1* merupakan variabel bilangan input yang ingin di cek, apakah *n1* merupakan bilangan sempurna atau tidak. Semetara itu, variabel *n2* akan digunakan sebagai variabel temporal yang menampung nilai yang lebih kecil sama dengan ($\leq$ (n1)/2). Dengan begitu, kita dapat melakukan pengecekan terhadap semua bilangan yang membagi habis *n1* melalui variabel *n2*. Setiap looping, akan dilakukan pengecekan, apakah *n2* habis membagi *n1*? Apabila iya, *n2* akan ditambahkan ke *sum*, tetapi, apabila tidak, maka *n2* tidak akan ditambahkan. Apabila sampai selesai looping, nilai *sum* $\neq$ *n1*, maka bilangan n1 bukan bilangan sempurna. Namun, Apabila sebaliknya, *sum* = *n1*, maka n1 merupakan bilangan sempurna.

Nilai i sama dengan nilai n2 awal. Nilai n2 awal adalah setengah dari nilai n1 (n2 = (n1)/2) karena tidak ada nilai n2 yang memenuhi (0 < n2 > (n1)/2) dan ((n1)/(n2) = 0).

2.2.2 Algotitma Rekursif

Algoritma rekursif pengecekan bilangan sempurna hanya terdiri dari empat percabangan dan dua rekursi fungsi.

```c
#include "rekursif.h"

int bilanganSempurna(int n1,int n2){

    if(n2 == 0){
        return 0;
    } else if (n2 == 1){
        return n2;
    } else if(n1%n2 == 0){
        return n2 + bilanganSempurna(n1,n2-1);
    } else {
        return bilanganSempurna(n1,n2-1);
    }
}
```

Gambar 4. Algoritma rekursif pengecek bilangan sempurna.

Berdasarkan algoritma rekursif di atas, kita diberikan input nilai, *n1*, yang ingin dicek, apakah nilai input, *n1*, merupakan bilangan sempurna atau

tidak; dan nilai *n2* yang akan dimanfaatkan untuk mencari dan menghitung nilai-nilai yang habis membagi *n1*. Terdapat empat kondisi pada fungsi rekursif yang digunakan. Kondisi pertama/awal akan memeriksa apakah n1 = 1? Apabila iya, maka fungsi akan mengembalikan 0 yang membuat $n1 \neq n2$ sehingga nilai input n1 dinyatakan bukan bilangan sempurna karena 1 bukan bilangan sempurna. Setelah itu, kondisi kedua akan memeriksa apakah *n2* = 1. Kondisi kedua digunakan sebagai rekursi akhir atau akhir dari proses rekursif karena satu pasti membagi habis semua bilangan. Kondisi ketiga memeriksa apakah bilangan *n2* yang direkursif membagi habis bilangan input, *n1*? Apabila iya, maka nilai *n2* akan di return dan diakumulasikan dengan nilai *n2* lain yang berbeda dan membagi habis *n1*. Kondisi terakhit, else, akan mengembalikan fungsi rekursifnya atau merekursi saja yang menandakan bahwa bilangan *n2* yang diperiksa tidak membagi habis bilangan input, *n1*.

Sama seperti algoritma iteratif sebelumnya, pada algoritma rekursif ini, nilai *n2* yang digunakan untuk menampung nilai yang membagi nilai input, *n1*, akan menampung nilai awal (*n1*)/2 dengan alasan yang sama, tidak ada nilai *n2* yang memenuhi (0 < *n2* > (n1)/2) dan ((*n1*)/(*n2*) = 0).

2.3 Kompleksitas Algoritma

Untuk pemaparan kompleksitas Algoritma, kami akan membagi pemaparan kompleksitas algoritma menjadi dua bagian, yaitu yang iteratif dan rekursif. **Operasi dasar yang didefinisikan adalah <u>perbandingan</u>**.

2.3.1 Kompleksitas Algoritma Iteratif

Berikut adalah fungsi kompleksitas algoritma iteratif,

$$T_{iteratif}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} 1$$

Atau

$$(\lfloor n/2 \rfloor - 1 + 1)$$

Rumus ini menyatakan banyaknya operasi dasar perbandingan yang dilakukan sampai selesai looping oleh algoritma. Setiap looping, hanya ada satu operasi perbandingan yang dilakukan. Karena T(n) = ⌊n/2⌋, maka kelas kompleksitas dari algoritma iteratif di atas adalah **Θ(n)** atau **linear**.

2.3.2 Kompleksitas Algoritma Rekursif

Berikut adalah fungsi kompleksitas algoritma rekursif,

$$T_{\text{rekursif}}(n) = \begin{cases} 1, & n = 0 \\ 2, & n = 1 \\ T_{\text{rekursif}}(n-1) + 3, & n > 1 \end{cases}$$

T(n) = T(n-1) + 3

T(n) = (T(n-2) + 3) + 3

T(n) = T(n-2) + 6

T(n) = (T(n-3) + 3) + 6

T(n) = T(n-3) + 9

T(n) = T(n-i) + i*3

T(n) = T(1) + (n-1)*3

T(n) = 2 + (n-1)*3

T(n) = 3*n - 1

Rumus di atas dapat disimplifikasikan menjadi,

$$T_{\text{rekursif}}(n) = \begin{cases} 1, & n = 0 \\ 3n - 1, & n \geq 1 \end{cases}$$

Rumus di atas menyatakan banyaknya operasi dasar perbandingan yang dilakukan sampai selesai rekursi atau pemanggilan ulang fungsi rekursif. Apabila nilai input, n, lebih dari 1, maka terdapat tiga operasi perbandingan yang dilakukan. Apabila n = 1, maka terdapat dua operasi perbandingan yang dilakukan. Namun, apabila n = 0, maka hanya ada satu operasi perbandingan yang dilakukan. Karena T(n) = 3n-1, maka kelas kompleksitas dari algoritma iteratif di atas adalah **Θ(n)** atau **linear**.

2.4 Simulasi

Berikut adalah beberapa simulasi dari program algoritma iteratif dan rekursif pengecek bilangan sempurna.

```cpp
#include "iteratif.h"
#include <chrono>

using namespace std;

int main() {
    int n1;

    cout << "-------------------------------------" << endl;
    cout << "PENGECEK BILANGAN SEMPURNA (ITERATIF)" << endl;
    cout << "-------------------------------------" << endl;
    cout << "Masukkan bilangan sebagai batas atas (Domain >= 1): ";

    n1 = 8128; // Bilangan yang dipilih

    // Mulai timing
    auto start = std::chrono::high_resolution_clock::now();

    if (bilanganSempurna(n1, n1/2) == n1) {
        cout << n1 << " adalah bilangan sempurna!" << endl;
    } else {
        cout << n1 << " BUKAN bilangan sempurna!" << endl;
    }

    // Akhiri timing
    auto stop = std::chrono::high_resolution_clock::now();
    auto duration =
        std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

    cout << "Waktu eksekusi: " << duration.count()
         << " microseconds" << endl;

    return 0;
}
```

Gambar 5. Input 8128 pada algoritma iteratif.

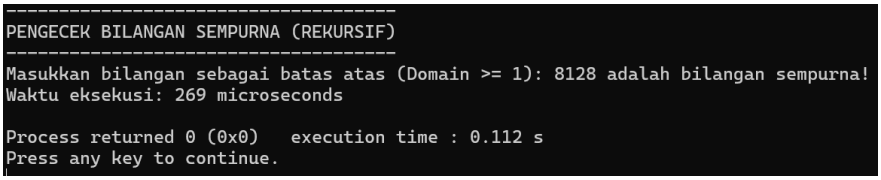
```
-------------------------------------
PENGECEK BILANGAN SEMPURNA (ITERATIF)
-------------------------------------
Masukkan bilangan sebagai batas atas (Domain >= 1): 8128 adalah bilangan sempurna!
Waktu eksekusi: 217 microseconds

Process returned 0 (0x0)   execution time : 0.113 s
Press any key to continue.
```

Gambar 6. Output dari Input 8128 pada algoritma iteratif.

main.cpp | rekursif.cpp | rekursif.h | main.cpp | iteratif.cpp | iteratif.h

```cpp
#include "rekursif.h"
#include <iostream>
#include <chrono>

using namespace std;

int main()
{
    int n1;
    cout << "-------------------------------------" << endl;
    cout << "PENGECEK BILANGAN SEMPURNA (REKURSIF)" << endl;
    cout << "-------------------------------------" << endl;
    cout << "Masukkan bilangan sebagai batas atas (Domain >= 1): ";

    n1 = 8128;

    auto startTime = std::chrono::high_resolution_clock::now();

    if (bilanganSempurna(n1, n1/2) == n1) {
        cout << n1 << " adalah bilangan sempurna!" << endl;
    } else {
        cout << n1 << " BUKAN bilangan sempurna!" << endl;
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration =
        std::chrono::duration_cast<std::chrono::microseconds>(endTime - startTime);

    cout << "Waktu eksekusi: " << duration.count()
         << " microseconds" << endl;

    return 0;
}
```

Gambar 7. Input 8128 pada algoritma rekursif.



Gambar 8. Output dari Input 8128 pada algoritma rekursif.

Algoritma Iteratif				
Input(X)	Running time rata-rata	Running Time 1	Running Time 2	Running Time 3
1	53,33333333	55	51	54
10	57,66666667	61	53	59
50	58,66666667	55	60	61
100	63,33333333	65	55	70
300	64,33333333	64	77	52
500	67	90	52	59
700	75	66	71	88
1000	75,33333333	98	55	73
1500	79	69	90	78
2000	81,66666667	91	73	81
2500	86,33333333	97	93	69
3000	92	120	74	82
3500	93,66666667	119	82	80
4000	99	120	94	83
4500	99	112	102	83
5000	107,6666667	124	100	99
6000	112	111	115	110
7000	119,6666667	109	130	120
8000	120	101	149	110
9000	127	132	120	129
10000	135	116	111	178
15000	139	120	145	152
20000	109,6666667	112	104	113
25000	125	102	170	103
30000	144,6666667	152	170	112
35000	157,3333333	181	102	189
40000	136	136	102	170
45000	105,3333333	99	107	110
50000	199	119	306	172
55000	208	144	308	172
60000	250,3333333	210	424	117

Tabel 1. Tabel *Running Time* Algoritma Iteratif

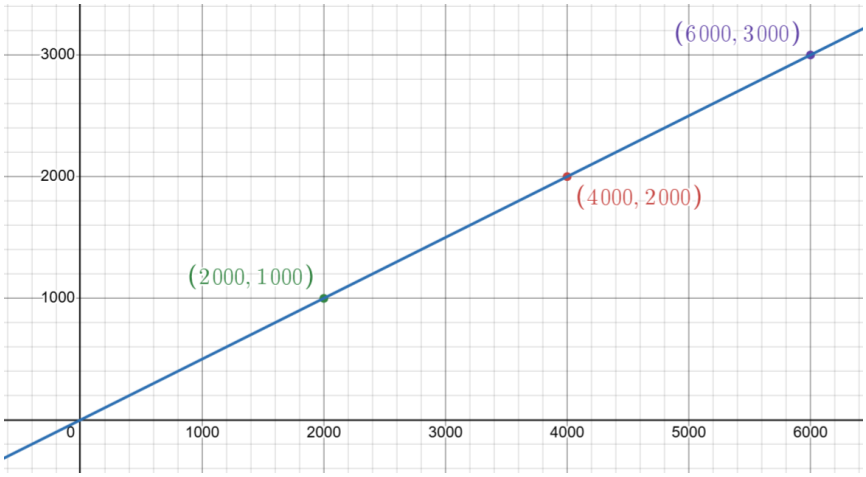
Algoritma Rekursif				
Input(X)	Running time rata-rata	Running Time 1	Running Time 2	Running Time 3
1	70	91	59	60
10	71	69	64	80
50	73,66666667	85	71	65
100	75	101	63	61
300	77,66666667	67	100	66
500	82,66666667	64	93	91
700	108,6666667	113	82	131
1000	109,3333333	122	79	127
1500	136,3333333	157	107	145
2000	137,6666667	139	154	120
2500	144	154	129	149
3000	155,6666667	160	157	150
3500	173,6666667	201	174	146
4000	197,6666667	142	177	274
4500	209,6666667	217	199	213
5000	215	249	224	172
6000	239	226	233	258
7000	285	229	194	432
8000	325	424	357	194
9000	328	383	306	295
10000	340,6666667	376	301	345
15000	405,6666667	358	303	556
20000	522	558	346	662
25000	558	570	549	555
30000	977	1087	803	1041
35000	777	625	661	1045
40000	1071	718	885	1610
45000	861,3333333	959	736	889
50000	891,6666667	975	802	898
55000	1015	1094	1062	889
60000	1108	1281	846	1197

Tabel 2. Tabel *Running Time* Algoritma Rekursif

2.5 Grafik

2.5.1 Grafik Algoritma Iteratif

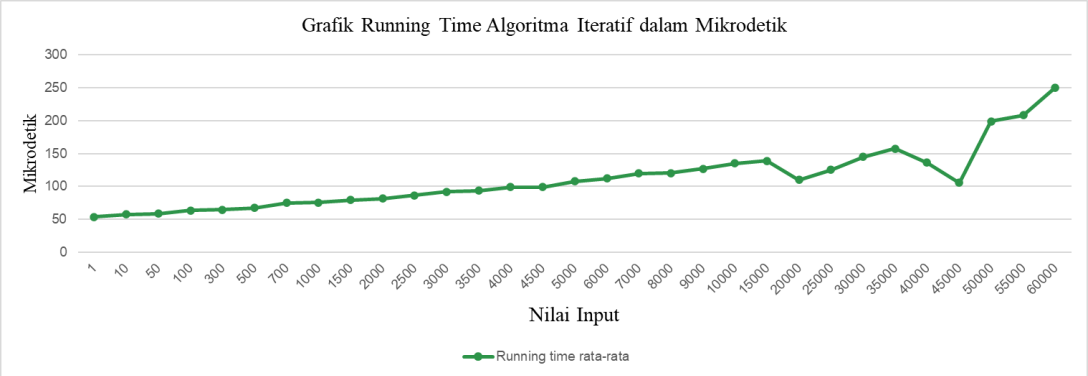
Rumus kompleksitas algoritma iteratif T(n) = ⌊n/2⌋. Namun, untuk memudahkan perbandingan, akan diambil T(n) ≈ n/2. Maka dari itu, berikut adalah rumus kompleksitas algoritma iteratif n/2.



Gambar 9. Grafik kompleksitas algoritma iteratif.

2.5.2 Grafik Running Time Algoritma Iteratif

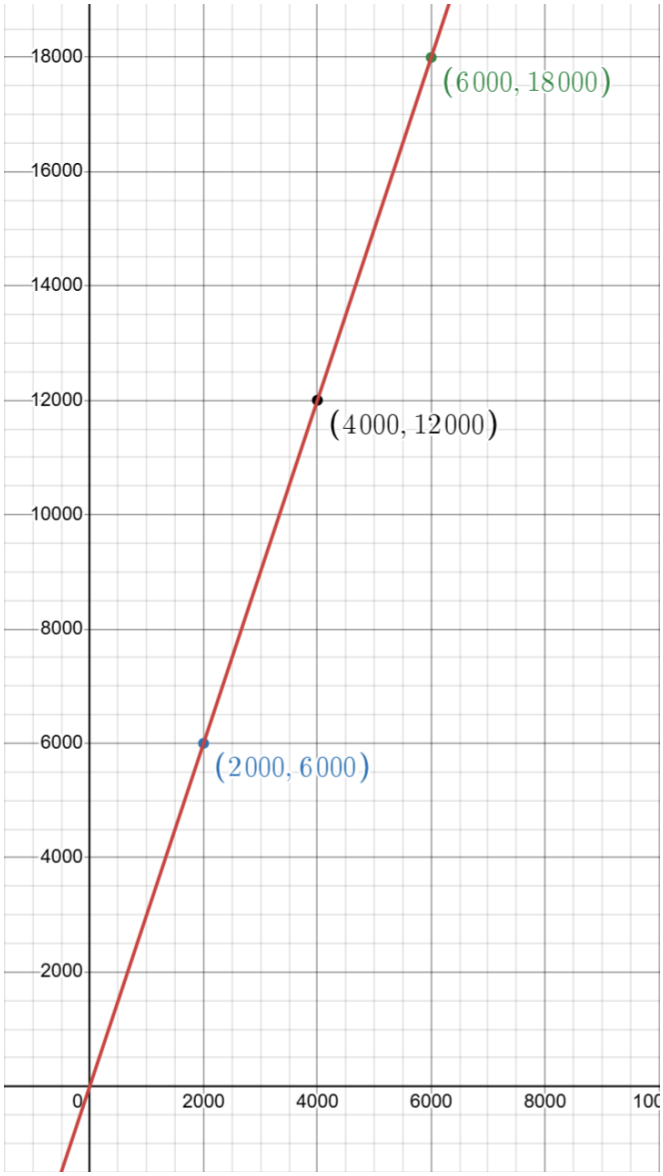
Berikut adalah grafik running time yang didapatkan dari tabel *running time* algoritma iteratif.



Gambar 10. Grafik running time algoritma iteratif.

2.5.3 Grafik Algoritma Rekursif

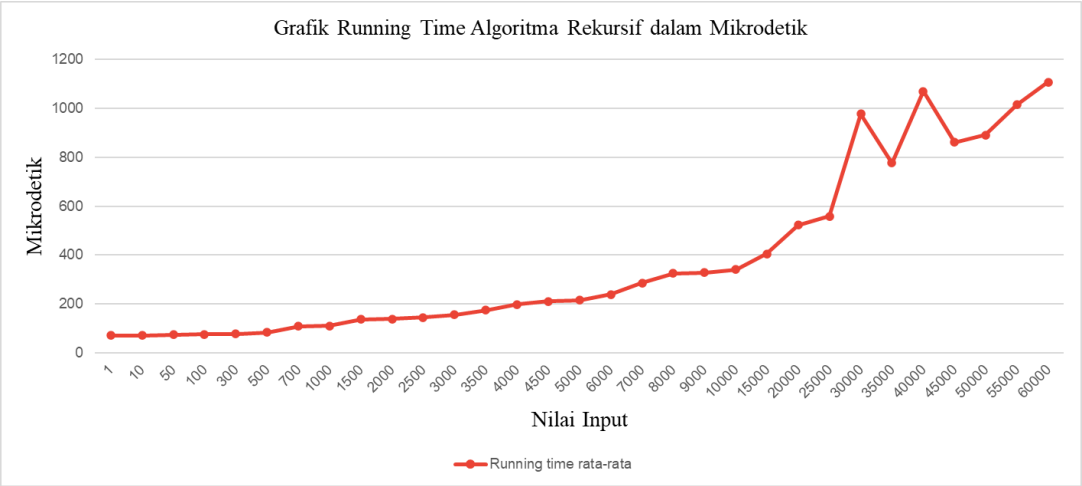
Berikut adalah grafik kompleksitas algoritma rekursif dengan rumus kompleksitas T(n) = 3n-1.



Gambar 11. Grafik kompleksitas algoritma rekursif.

2.5.4 Grafik Running Time Algoritma Rekursif

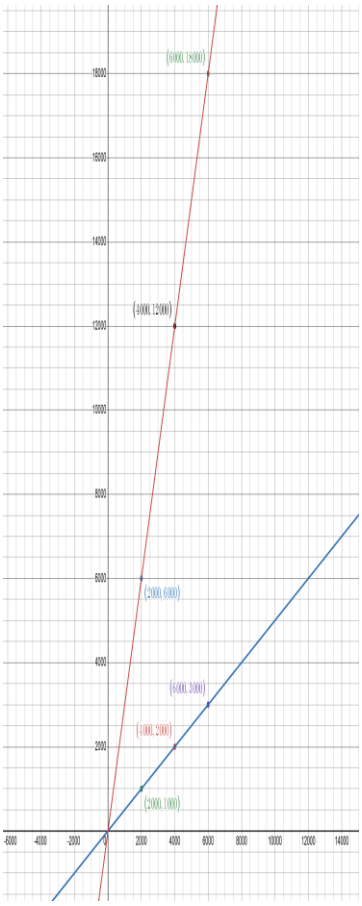
Berikut adalah grafik *running time* yang didapatkan dari tabel running time algoritma rekursif.



Gambar 12. Grafik *running time* algoritma Rekursif.

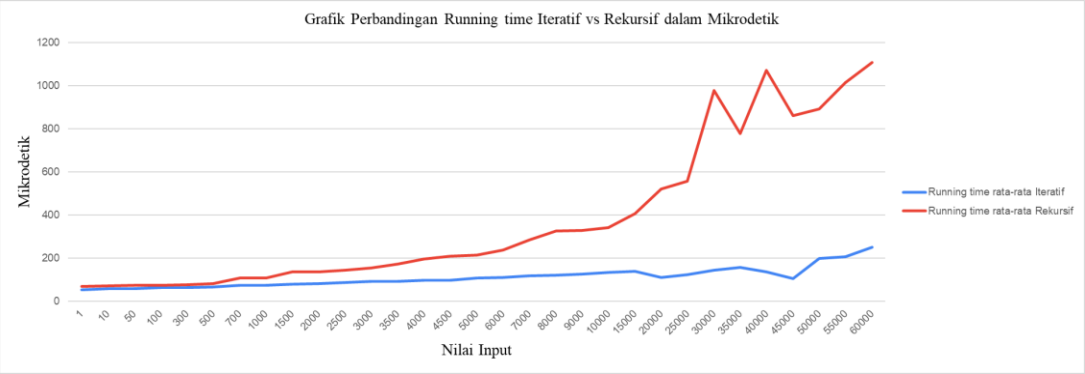
2.6 Perbandingan

Berikut adalah perbandingan grafik kompleksitas algoritma iteratif (biru) dengan algoritma rekursif (merah).



Gambar 13. Grafik perbandingan kompleksitas algoritma Iteratif dengan algoritma Rekursif.

Berikut adalah perbandingan grafik *running time* algoritma iteratif (biru) dengan algoritma rekursif (merah).



Gambar 14. Grafik perbandingan *running time* algoritma Iteratif dengan algoritma Rekursif.

BAB III
PENUTUP DAN DAFTAR PUSTAKA

3.1 Kesimpulan

Kedua algoritma iteratif ($T(n) = \lfloor n/2 \rfloor$) dan algoritma rekursif ($T(n) = 3n-1$) di atas sama-sama melakukan pengecekan input untuk memverifikasi sifat bilangan sempurna dari input dan memiliki kelas kompleksitas yang sama, yaitu linear atau $\Theta(n)$. Namun, fungsi kompleksitas algoritma rekursif itu lebih kompleks dibandingkan dengan fungsi kompleksitas algoritma iteratif; dan algoritma rekursif mengeksekusi lebih banyak operasi dasar daripada algoritma iteratif. Hal itu terbukti pada bagian perbandingan grafik kompleksitas dan grafik *running time* yang disajikan. Maka dari itu, algoritma iteratif di atas dinilai lebih efektif dan efisien dalam melakukan pengecekan bilangan sempurna dibandingkan dengan algoritma rekursif yang jauh lebih kompleks.

3.2 Daftar Pustaka

Affaf, Moh. (2016, Juli). Bilangan Sempurna Genap dan Keprimaan bilangan Mersenne. *APOTEMA: Jurnal Pendidikan Matematika*, Vol. 2, No. 2, hlm. 63–75. STKIP PGRI Bangkalan. Diakses dari https://stkippgri-bkl.ac.id/wp-content/uploads/2017/04/Affaf.pdf

---

i