# Analisis Algoritma Rekursif untuk Mencari Element Maksimum sebuah List

Fadil Fauzani - 13520032[1]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*[1]13520032@std.stei.itb.ac.id*

***Abstract*—Algoritma adalah kumpulan langkah-langkah yang dibuat untuk mensimulasikan permasalahan. Algoritma ada yang baik dan buruk, bergantung dengan alur dan kompleksitas algoritma itu sendiri. Rekursifitas memiliki kelebihan cenderung lebih simpel dalam kode tapi belum tentu dalam kompleksitasnya. Untuk membedahnya kita perlu Graf untuk mereprsentasikan alur algoritma.**

***Keywords*—Algoritma, Alur, Rekursi, Komplesitas,.**

## I. PENDAHULUAN

Dalam dunia keinformatikaan, terdapat banyak masalah. Masalah-masalah ini bervariasi mulai dari simpel hingga rumit. Mulai dari masalah yang bisa kita langsung pikirkan solusinya hingga masalah yang harus menggunakan diagram bantu untuk merepresentasikan masalah tersebut. Didalam kehidupan saya selama perkuliahan juga begitu, banyak persoalan mulai dari Mata kuliah Algortima dan Struktur Data, Matematika Diskrit, hingga dari kehidupan pergaulan itu sendiri. Mulai dari persoalan tentang bilangan, logika, struktur, komputer, koding, hingga persoalan kehidupan pergaulan. Diantara permasalahan itu, ada salah satu permasalahan yang mudah tetapi menarik untuk dibahas. Persoalan ini berhubungan dengan himpunan angka. Yaitu persoalan tentang mencari nilai maksimal dari suatu himpunan.

Kita terutama saya sendiri, sering menemui persoalan untuk menemukan element maksimal pada sebuah List. Masalah ini dapat memiliki banyak solusi yang berbeda beda. Dari yang rumit hingga yang simpel, dan juga pasti memiliki efektifitas yang berbeda-beda. Mulai dari pendekatan prosedural, hingga rekursif.

Kita bisa menganalisis salah satu solusi tersebut dengan melihat kodenya, panjang kodenya, kemudahan untuk memperoleh kode tersebut, alur kode, kompleksitas.

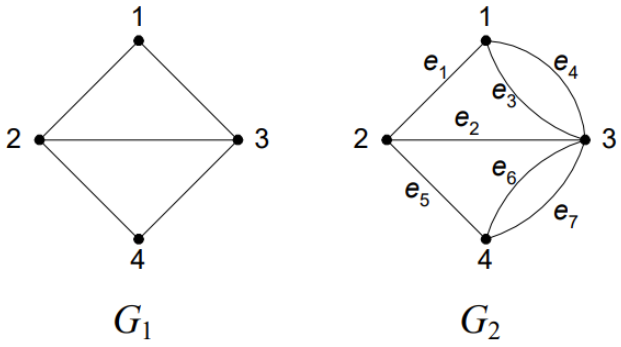
Dengan Teori Graf kita bisa merepresentasikan alur kode sehingga dapat lebih mudah untuk dilihat. Dan dengan melihat kodenya kita bisa mementukan kompleksitas kode tersebut dengan melihat banyaknya langkah langkah yang terjadi,

Kali ini saya akan membahas tentang pendekatan rekursif, dari bagaimana landasan berfikirnya, langkah langkah pembuatan, analisis alur yang terjadi pada algoritmanya, hingga kompleksitasnya dan membandingkannya dengan solusi lain.

## II. LANDASAN TEORI

### A. Graf

Graf adalah struktur diskrit yang digambarkan dengan kumpulan simpul (*Nodes)* dan sisi (*Edges)*. Graf ditulis dengan Notasi $G = (V, E)$, *V* adalah himpunan tidak kosong yang berisi kumpulan simpul yang ada pada *G* dan *E* himpunan yang berisi kumpulan sisi yang menyambungkan antar simpul di G (boleh kosong).



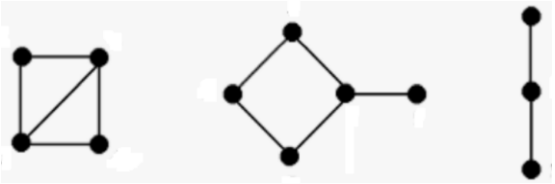

Gambar 2.1 Contoh Graf

Graf G1 dan G2 pada gambar diatas dapat dituliskan dengan Notasi,

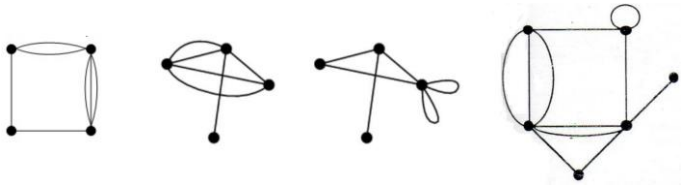
$$G_1 = (\{1,2,3,4\}, \{(1,2),(1,3),(2,3),(2,3),(3,4)\})$$

$$G_2 = (\{1,2,3,4\}, (e_1, e_2, e_3, e_4, e_5, e_6, e_7))$$

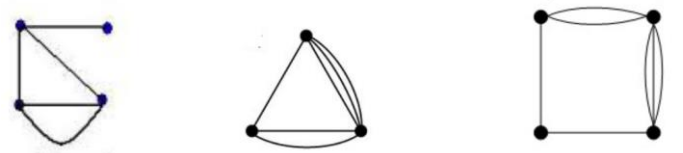
Graf dapat dibedakan menjadi 2 yaitu Graf sederhana, Graf yang tidak memiliki sisi ganda dan gelang, dan Graf tak-sederhana, Graf yang memiliki sisi ganda atau gelang. Graf tak-sederhanda dapat dibedakan lagi menjadi 2 yaitu Graf ganda, Graf yang memiliki sisi ganda, dan Graf semu, Graf yang memiliki gelang. Berdasarkan orientasi sisi, Graf dibagi menjadi Graf tak-berarah dan Graf berarah.



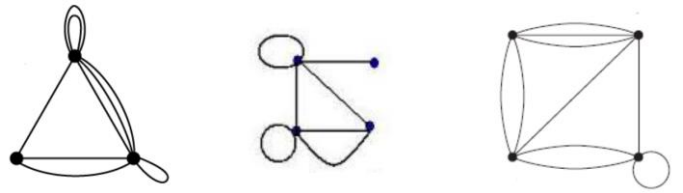
*Gambar 2.2 Graf sederhana*



*Gambar 2.3 Graf tak-sederhana*

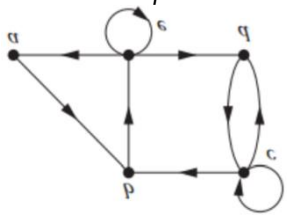


*Gambar 2.4 Graf ganda*



*Gambar 2.5 Graf semu*

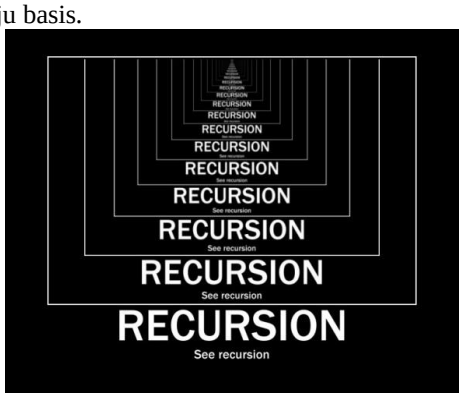
*Gambar 2.6 Graf tak-berarah*



*Gambar 2.7 Graf berarah*

## B. Rekursi

Secara simpel, Rekursi adalah sesuatu yang dalam pendefinisiannya terdapat terminologi dirinya sendiri. Algoritma rekursif adalah Algoritma yang terdapat rekursi didalam pendefinisiannya. Di dalam Algoritma rekursif terdapat basis dan rekurens. Basis adalah bagian paling dasar dimana tidak ada bagian rekursif lagi, di sini adalah bagian yang membuat algoritma rekursif mempunyai akhir. Rekurens adalah bagian dimana terdapat pemanggilan diri sendiri didalamnya, biasanya pemanggilannya dibedakan sedikit dan perubahannya akan menuju basis.




*Gambar 2.8 Gambar yang bersifar rekursif*

Faktorial dapat diekspresikan dalam bentuk rekursif yaitu,

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0, \end{cases}$$

berdasarkan rumus diatas, diketahui bahwa basis dalam faktorial adalah ketika n = 0, dengan nilai basis 0. Jika kita ingin mencari 5! maka prosesnya akan seperti berikut,

$$5! = 5 * 4!$$
$$5! = 5 * 4 * 3!$$
$$5! = 5 * 4 * 3 * 2!$$
$$5! = 5 * 4 * 3 * 2 * 1!$$
$$5! = 5 * 4 * 3 * 2 * 1 * 1$$
$$5! = 120$$

dengan proses diatas kita dapatkan $5! = 120$.

Jika perhitungan bilangan faktorial dibuat kedalam algoritma rekursif maka kodenya akan menjadi seperti ini

```
{ contoh Algoritma O(log n)
function fibrecc(N: integer)
      if (N <= 0) then
            -> 1
      else
            -> N * fibrecc(N-1)
```

## C. Komplesitas Algoritma

Komplesitas Algoritma adalah besaran / kuantitas yang dipakai untuk waktu atau ruang yang dipakai algoritma untuk berjalan. Umumnya semakin kompleks suatu algoritma maka semakin buruk juga kinerja algoritma tersebut dalam waktu atau ruang. Kompleksitas algoritma yang bagus adalah yang nilainya kecil karena berdampak pada keberjalananya algoritma. Pengukuran komplesitas dibagi menjadi 2, yaitu

1. Kompleksitas Waktu
   Komplesitas waktu adalah pengukuran waktu yang dibutuhkan algoritma untuk berjalan. Kompleksitas waktu umumnya dihitung dengan mencari banyaknya operasi khas yang dilakukan pada algoritma. Operasi-operasi khas ini adalah bagian dari:
   - Aritmatika
   - Perbandingan
   - Perubahan variabel
   - Penambahan variabel
   - dll
2. Komplesitas Ruang
   Komplesitas Ruang adalah pengukur memori yang dipakai algoritma ketika berjalan. Biasanya diukur dengan mencari maksimal berapa variabel yang sedang dipakai pada algoritma pada waktu tertentu

Dalam penggunannya, Kompleksitas algoritma biasanya memakai Notasi Big-O (Big-O Notation), namun tidak hanya ada Big-O tetapi juga ada Big-Omega dan Big-Theta.

1. Big-O
   Big-O, adalah pengukuran T(n), sehingga $T(n) \leq O(f(n))$. Artinya, Jika $T(n) = O(f(n))$ ada $C$ dan $n_0$ sehingga $T(n) \leq C * f(n)$ untuk $n \geq n_0$.
2. Big-Omega
   Big-Omega, adalah pengukuran T(n), sehingga $T(n) \geq \Omega(f(n))$. Artinya, Jika $T(n) = \Omega(f(n))$ ada $C$ dan $n_0$ sehingga $T(n) \geq C * f(n)$ untuk $n \geq n_0$.

3. Big-Theta
   Big-Theta hanya ada jika berlaku $T(n) = O(f(n))$ dan $T(n) = \Omega(f(n))$ sehingga $T(n) = O(f(n)) = \Omega(f(n)) = \Theta(f(n))$.

Kompleksitas Algoritma dapat dikelompokan berdasarkan jenis fungsinya yaitu.

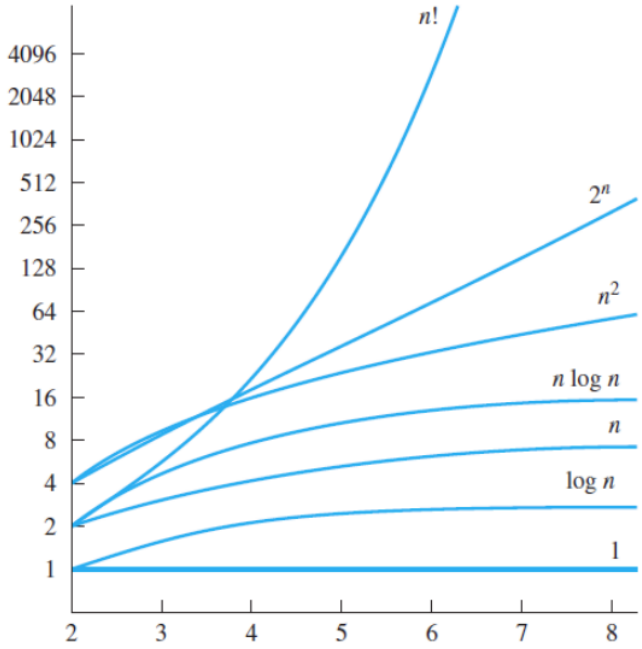
| $O(f(n))$ | Nama | Kelompok |
|---|---|---|
| $O(1)$ | Konstan | Polinomial |
| $O(\log n)$ | Logaritmik | Polinomial |
| $O(n)$ | Linear | Polinomial |
| $O(n * \log n)$ | Linear Logaritmik | Polinomial |
| $O(n^2)$ | Kuadratik | Polinomial |
| $O(n^3)$ | Kubik | Polinomial |
| $O(2^n)$ | Eksponensial | Eksponensial |
| $O(n!)$ | Faktorial | Eksponensial |

*Tabel 2.1 Pengelompokan Jenis Kompleksitas Algoritma*

Dari jenis jenis Kompkesitas Algoritma di tabel di atas, urutan yang dari yang paling baik adalah

$$1 < \log n < n < n * \log n < n^2 < n^3 < 2^n < n!$$

dimana komplesitas yang kecil lebih baik daripada kompleksitas yang besar.




*Gambar 2.9 Grafik Pertumbuhan berbagai kompleksitas*

1. $O(1)$
   Komplesitas konstan, artinya keberjalanan algoritma tidak bergantung pada ukuran data yang diproses. Umumnya terdapat pada algoritma yang tidak mempunya pengulangan dalam instruksi-instruksinya. Contoh : Pertukaran nilai variabel.

```
{ contoh pertukaran nilai
variabel }
temp <- a
a <- b
b <- temp
{hasil akhir anew = b, bnew = a  }
```

2. $O(\log n)$
   Kompleksitas logaritmik, artinya laju perubahan keberjalanan algoritma melaju lebih lambat dibandingkan dengan laju pertumbuhan ukuran data yang diproses. Umumnya terdapat pada algoritma yang memecahkan persoalan ke persoalan-persoalan kecil yang berukuran sama. Contoh, Binary search.

```
{ contoh Algoritma O(log n)
procedure printLogN(N: integer)
      if (N = 0) then
            output('\n')
      else
            output('L')
            printLogN(N div 2)
```

3. $O(n)$
   Komplesitas Linear, artinya laju perubahan keberjalanan algortima melaju sama cepatnya dengan laju pertumbuhan data yang diproses. Umumnya terdapat pada algoritma yang melakukan pencari yang melewati semua data. Contoh, Sequential Search.

```
{ contoh Algoritma O(n) }
procedure printN(N: integer)
      while (N > 0) do
            output('L')
            N <- N - 1
      output('\n')
```

4. $O(n * \log n)$
   Mulai dari sini, laju perubahan keberjalanan algoritma melaju lebih cepat dibandingkan dengan laju pertumbuhan ukuran data yang diproses. Umumnya ditemukan pada algoritma yang menggunakan prinsip divide & conquer.

```
{ contoh Algoritma O(n log n) }
procedure printNLogN(N: integer)
      i traversal [1..N]
            j traversal [1..N div 2]
                  output('L')
      output('\n')
```

5. $O(n^2)$
   Untuk data yang besar akan sangat jauh perbedaannya karena berubah kuadratik. Umumnya ditemukan pada algoritma yang terdapat nested loop pada ukuran data.

```
{ contoh Algoritma O(n*n) }
procedure printNxN(N: integer)
      i traversal [1..N]
            j traversal [1..N]
                  output('L')
      output('\n')
```

6. $O(n^3)$
   Ditemukan pada algoritma yang terdapat 3 kali loop bersarang yang mengulang sesuai ukuran data.

```
{ contoh Algoritma O(n*n*n) }
procedure printNxNxN(N: integer)
      i traversal [1..N]
            j traversal [1..N]
                  k traversal [1..N]
                        output('L')
      output('\n')
```

7. $O(2^n)$

Komplesitas ini sangat tidak efisien karena laju pertumbuhannya eksponensial. Biasanya digunakan jika ingin mencoba segala kemungkinan dari sesuatu yang tidak diketahui (password / sandi) atau dinamakan juga dengan Brute Force. Contoh, Algoritma mencari sircuit hamilton.

```
{ contoh Algoritma O(2^n) }
procedure print2powN(N, i: integer)
      if (i = N) then
            output('\n')
      j traversal[1..2]
            print('A')
            print2powN(N, i + 1)
```

8. $O(n!)$

Algoritma jenis ini memproses setiap masukan dan menghubungkannya dengan n – 1 masukan lainnya. Contohnya dalam Algoritma pedangang keliling.

```
{ contoh Algoritma O(n!) }
procedure printNf(N:integer)
      if (N = 0) then
            output('\n')
      i tranversal [1..N]
            output('L')
            printNf(N-1)
```

### *D. List*

List adalah sebuah struktur data yang menyimpan kumpulan element yang sama. Dalam beberapa bahasa pemrograman, List juga dinamakan sebagai Array. List disini akan dipakai sebagai struktur data dalam Algoritma yang kita buat. Dalam pengaksesakan List terdapat beberapa selektor. Yaitu

1. Head(L) = mengembalikan element pertama di list
2. Next(L) = mengembalikan list yang berisi element tanpa element pertama
3. L[i] = mengambil element ke i di list
4. L.Length = mengembalikan panjang list.

## III. ANALISIS ALGORITMA REKURSIF

### *A. Pembuatan Algoritma*

Algoritma Rekursif membutuhkan 2 bagian penting, yaitu bagian basis dan bagian rekursif. Bagian basis yang kita ambil adalah ketika list tersebut hanya memiliki satu element. Misal kita memiliki selektor Head(L) untuk mengambil element pertama List dan Next(L) untuk mengambil List yang berisi semua element List kecuali yang pertama. Kita juga perlu fungsi isOneElement yang berguna untuk menandakan basis. Maka Algoritma basis nya akan sebagai berikut.

```
if (isOneElement(L)) then
      -> Head(L)
```

Jika kita menemukan List yang memiliki satu element kita bisa pastikan bahwa element maksimal dari List tersebut adalah element pertamanya.

Setelah membuat basis, kita perlu membuat bagian rekursifnya. Disinilah kita perlu membuat supaya bagian rekursinya benar dan menuju basis. Untuk mempersimpel masalah misal kita mempunyai List dengan 2 element, maka untuk mencari maksimalnya kita hanya perlu membandingkan element pertama dan element kedua, mudah bukan? Sekarang bagaimana jika 3 element, 4 element, hingga N element? Disini kita perlu mengambil element pertama List dan membandingkannya dengan element maksimal dari List Next(L), dengan begitu secara prinsip kita hanya mengambil nilai maksimal dari keduanya. Maka Algoritma rekursifnya akan sebagai berikut.

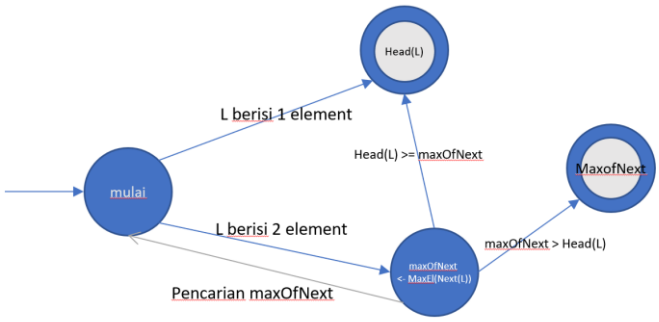
```
if (not(isOneElement(L))) then
      maxOfNext <- MaxEl(Next(L))
      if (Head(L) >= maxOfNext) then
            -> Head(L)
      else
            -> maxOfNext
```

Maka dengan menggabungkan bagian basis dan rekurens kita bisa membuat Algoritma utuh kita yaitu.

```
function MaxEl( L : List) -> real
      if (isOneElement(L)) then
            -> Head(L)
      else
            maxOfNext <- MaxEl(Next(L))
            if (Head(L) >= maxOfNext) then
                  -> Head(L)
            else
                  -> maxOfNext
```

### *B. Representasi Algoritma sebagai Graf*

Untuk lebih memahami Algoritma ini, kita bisa merepresentasikan alur Algoritma kita sebagai Graf berarah, dengan simpul merepresentasikan proses algortima dan sisi merepresentasikan kombinasi keadaan di dalam algoritma.




*Gambar 3.7 Graf Alur Algoritma*

Simpul yang bewarna putih di tengah menandakan hasil return algoritma. Sisi bewarna abu menandakan alur yang ditempuh pemanggilan fungsi. Algoritma ini bisa dikatakan sangat simpel karena dapat dibuat Graf dengan 4 simpul dan 5 sisi, yang merupakan nilai yang sedikit.

Dari sini terlihat jelas bahwa akan terjadi looping sampai ke basis yaitu di element terakir L. setelah itu akan dibandingkan mana yang paling besar hingga ke element pertama. Jika sudah maka hasil akan direturn berdasarkan yang mana yang paling besar.

Misal kita mempunyai List dengan isi sebagai berikut : [1,2,3,4], maka Algoritmanya akan berjalan seperti ini,

Mulai: { L = [1,5,3,4,2] }
maxOfNext <- MaxEl(Next(L)) { 5 }
Mulai: { L1 = [5,3,4,2] }
maxOfNext <- MaxEl(Next(L)) { 4 }
Mulai : { L2 = [3,4,2] }
maxOfNext <- MaxEl(Next(L)) { 4 }
Mulai : { L3 = [4,2] }

```
maxOfNext <- MaxEl(Next(L)) { 2 }
Mulai : { L3 = [5] }
        -> 2
        -> 4
        -> 4
        -> 5
        -> 5
```

*Warna menandakan scope yang berbeda.*

### *B. Komplesitas Algoritma*

Untuk Mencari Komplesitas waktu Algoritma, kita perlu melihat sekali lagi kode yang kita buat

```
function MaxEl( L : List) -> real
      if (isOneElement(L)) then
            -> Head(L)
      else
            maxOfNext <- MaxEl(Next(L))
            if (Head(L) >= maxOfNext) then
                  -> Head(L)
            else
                  -> maxOfNext
```

Bisa dilihat bahwa untuk T(1) tidak ada perbandingan langusng mengembalikan nilai Head(L), T(1) = 0. untuk N > 1, ada satu perbandingan dan langsung memanggil diri sendiri maka,

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

Jika kita turunkan sampai N =1 satu maka akan seperti ini,

$$T(n) = 1 + T(n-1) = 1 + 1 + T(n-2) = 1 + 1 + \cdots + 0$$

Dari sini kita bisa simpulkan kompleksitas waktu algoritma tersebut adalah

$$T(n) = n - 1$$

atau dalam notasi Big-O

$$T(n) = n - 1 = O(n)$$

Dari sini kita simpulkan bahwa algoritma kita mempunyai komplesitas yang linear, artinya bertambahnya N akan berbanding lurus dengan bertambahnya waktu algoritma.

Bagaimana algoritma kita jika dibandingkan algoritma lain. Disini kita akan membandingkan dengan 2 algoritma lain yaitu linear search dan Sorted search.

Berikut Kode untuk linear search,

```
function MaxElLin( L : List) -> real
      max <- L[0]
      i <- 0
      while (i < L.Length) do
            if (max < L[i]) then
                  max <- L[i]
      -> max
```

Berikut Kode untuk Sorted search,

```
function MaxElSorted( L : List) -> real
      max <- L[0]
      if (max < L[L.Length - 1]) then
            max <- L[L.Length - 1]
      -> max
```

Dari sini kita bisa dapatkan bahwa kompleksitas algoritma linear search dan sorted search berturut turut adalah O(n), O(1).

Untuk membandingkan ketiganya kita akan buat Table yang merepresentasikan fitur yang mereka punya.

| Algoritma | T(n) | S(n) | Harus sorted? |
|---|---|---|---|
| Rekursif | O(n) | O(n) | Tidak |
| Linear Search | O(n) | O(1) | Tidak |
| Sorted Search | O(1) | O(1) | Ya |

Dari sini kita mengetahui bahwa Sorted Search adalah yang paling cepat tetapi List yang dimasukan harus sudah terurut sehingga itu menjadi satu satunya kelemahan dari Sorted Search. Linear Search dan rekursif memiliki kompleksitas waktu yang sama namun untuk Linear Search memiliki kompleksitas yang konstan, untu rekursi karena perhitungan maxOfNext menyebabkan kompleksitasnya menjadi linear.

## IV. KESIMPULAN

Berdasarkan analissi di bab sebelumnya kita mendapatkan bahwa Algoritma mencari element maksimal terdapat beberapa cara yang berbeda beda, dengan menggunakan Algoritma rekursif kita bisa mendapatkan Komplesitas linear. Algoritma mencari element maksimal dengan pendekatan rekursi bekerja dengan mencari element maksimal List jika element pertama dihapus dan setelah mendapatkannya element maksimal tersebut dibandingkan kembali dengan element pertama dari L, jika element pertama lebih besar dari element paling besar List L jika element pertamanya dihapus maka, element pertamanya lah yang menjadi nilai maksimum List L, sebaliknya jika element paling besar List L jika element pertamanya dihapus lebih besar dari element pertama List L, maka element paling besar dari List L jika element pertamanya dihapus lah yang menjadi element paling besar List L. Algoritma mencari element maksimal dengan rekursif sudah termasuk algoritma yang, dan dibandingkan dengan algoritma yang lain, Algoritma rekursif adalah salah satu yang terbaik baik karena kompleksitasnya waktu linear (O(n)) dan kompleksitas ruang yang dipakai juga linear (O(n)). Dari saya sendiri lebih suka untuk memakai algoritma yang rekursif karena mudah untuk dimengerti, tetapi membuat algoritma rekursif bukan hal yang mudah perlu memutar otak dahulu baru bisa mendapatkan solusi rekursif.

## V. UCAPAN TERIMAKASIH

Pertama-tama saya ingin mengucapkan puja dan puji syukur kepada Allah SWT karena sudah memudahkan, melancarkan, dan mentakdirkan selesainya makalah ini yang dibuat untuk pemenuhan tugas mata kuliah IF2120 Matematika Dikrit dengan tepat waktu. Saya juga ingin mengucapkan terimakasih pada diri saya sendiri yang Alhamdulillah dapat mengumpulkan Niat, Ide, dan Materi sehingga dapat menyelesaikan tugas makalah ini dengan tepat waktu. Saya juga ingin mengucapkan terimakasih kepada teman-teman, kakak-kakak tingkat yang saya lihat judulnya sehingga saya mendapatkan ide judul untuk makalah saya. Tak lupa saya juga ingin mengucapkan terimakasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T., selaku pengampu mata kuliah IF2120 Matematika Diskrit pada Semester I 2021/2022, karenanya saya bisa mengerjakan tugas ini, karena ilmu yang diberikannya juga saya bisa mengerjakan tugas, kuis, maupun ujian dengan lancar. Sekali lagi saya ucapkan terimakasih.

## REFRENSI


[1] https://www.geeksforgeeks.org/find-the-maximum-element-in-an-array-which-is-first-increasing-and-then-decreasing/. Diakses Pukul 22:40 11 Desember 2021

[2] Munir, Rinaldi. 2020. "Rekursi dan Relasi Rekurens (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Rekursi-dan-relasi-rekurens-(Bagian-1).pdf (Diakses Pukul 21.00 11 Desember 2021

[3] Munir, Rinaldi. 2020. "Kompleksitas Algoritma (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf (Diakses Pukul 21:50 11 Desember 2021)

[4] Munir, Rinaldi. 2020. "Kompleksitas Algoritma (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf (Diakses Pukul 21:50 11 Desember 2021)

[5] Munir, Rinaldi. 2020. "Graf (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf (Diakses Pukul 22:20 11 Desember 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.

Bandung, 11 Desember 2020

Fadil Fauzani 13520032