Analisis Perbandingan Efisiensi Penyelesaian Persoalan *Longest Increasing Subsequence* dengan Metode *Decrease & Conquer* dan *Dynamic Programming*

Primanda Adyatma Hafiz - 13520022
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13520022@std.stei.itb.ac.id

***Abstract*—Persoalan *Longest Increasing Subsequence* (LIS) merupakan persoalan yang bertujuan untuk mencari *subset* array terbesar yang masih terurut membesar. Persoalan LIS dapat diselesaikan dengan metode *dynamic programming* yang memerlukan kompleksitas waktu $O(n^2)$ sedangkan penyelesaian persoalan ini dengan metode *decrease and conquer* menghasilkan kompleksitas waktu $O(n \log n)$. Persoalan ini lebih efisien bila diselesaikan dengan metode *decrease and conquer*, akan tetapi metode ini tidak dapat memberikan array solusi LIS secara efektif.**

Keywords—Longest Increasing Subsequence, Decrease And Conquer, Dynamic Programming

I. PENDAHULUAN

Persoalan *Longest Increasing Subsequence* (LIS) merupakan salah satu jenis persoalan optimasi dalam pemrosesan array 1 dimensi. Pada persoalan LIS kita diminta untuk mencari *subset* array dengan jumlah elemen terbanyak yang terurut membesar dari indeks yang terkecil. Persoalan ini juga termasuk persoalan pemrosesan array statik karena pada saat *runtime* array yang diproses tidak dapat mengalami perubahan.

Persoalan LIS merupakan salah satu persoalan klasik di dunia pemrograman dan juga banyak dipelajari di beberapa lintas bidang antara lain yaitu pada bidang matematika dan fisika. Biasanya persoalan LIS hanya meminta nilai panjang *subset* array maksimum yang masih terurut membesar, akan tetapi terkadang untuk meningkatkan kesulitan permasalahan diminta pula setiap elemen dari *subset* array tersebut. Terdapat beberapa kandidat solusi untuk menyelesaikan persoalan LIS, antara lain adalah dengan metode *brute force*, *greedy*, *decrease & conquer*, dan *dynamic programming*.

Metode *brute force* adalah sebuah pendekatan yang sangat jelas untuk memecahkan persoalan, biasanya penyelesaian dengan metode ini hanya didasarkan pada *problem statement* dan definisi konsep yang dilibatkan. Dalam hal persoalan LIS, metode ini memanfaatkan proses *exhaustive search* untuk mengecek seluruh kemungkinan *subset* array yang memenuhi kemudian dari kandidat solusi akan dicari solusi yang paling optimum. Akan tetapi, untuk masukan array dengan jumlah elemen yang banyak metode ini tidak cukup efisien karena terdapat 2^n kemungkinan *subset* array yang perlu dievaluasi sehingga program hanya efektif untuk array dengan $n \leq 30$.

Metode *greedy* adalah pendekatan penyelesaian persoalan dimana pada setiap langkahnya akan diambil pilihan yang merupakan solusi optimum lokal dengan harapan bahwa langkah sisanya mengarah ke solusi optimum global. Dalam hal persoalan LIS, metode ini akan melakukan iterasi dari indeks terkecil pada array dengan terus mengambil elemen array yang masih memenuhi untuk dimasukkan ke akhir array solusi sehingga pada akhirnya array solusi tetap terurut membesar. Solusi dengan metode *greedy* ini tergolong cukup efektif karena hanya memerlukan $O(n)$ pemrosesan, akan tetapi metode *greedy* tidak dapat menjamin solusi yang optimum global karena terdapat kemungkinan panjang array solusi kurang dari panjang array dari solusi yang sebenarnya.

Oleh karena itu, pada makalah ini penulis akan membahas dua alternatif penyelesaian persoalan LIS lainnya yaitu dengan menggunakan *decrease & conquer* serta *dynamic programming*. Penulis juga akan menganalisis efisiensi program dengan menggunakan dua metode tersebut serta kelebihan dan kekurangan dari setiap metode yang digunakan.

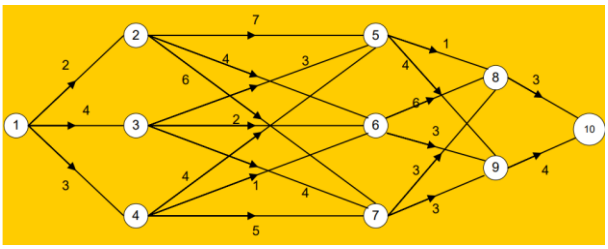
II. LANDASAN TEORI

A. Dynamic Programming

Dynamic programming adalah sebuah metode penyelesaian masalah dengan menguraikan solusi ke dalam kumpulan tahapan yang memiliki *state* tertentu. Oleh karena itu, solusi dari persoalan dengan metode *dynamic programming* dapat dipandang sebagai serangkaian keputusan yang saling berkaitan satu sama lain.

Dynamic programming biasanya digunakan untuk menyelesaikan persoalan optimasi serta persoalan yang

memiliki karakteristik rekursif. Pada kenyataannya metode *greedy* juga dapat menyelesaikan persoalan optimasi (maksimasi atau minimasi). Akan tetapi, dalam beberapa kasus strategi *greedy* tidak dapat memberikan solusi yang optimal karena strategi *greedy* hanya menghasilkan satu rangkaian keputusan saja yang bisa jadi hanya merupakan solusi optimum lokal. Sedangkan dengan *dynamic programming* dapat dihasilkan lebih dari satu rangkaian keputusan untuk nantinya dipertimbangkan sehingga dapat menjamin solusi sebagai solusi optimum global.



Gambar 1. Ilustrasi Dynamic Programming
Diambil dari
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf pada 9 Mei 2022

Dynamic programming menggunakan prinsip optimalitas yang menyatakan bila solusi optimal, maka bagian solusi sampai tahap ke k juga optimal. Selain itu, prinsip optimalitas juga berarti bahwa untuk mencari solusi optimal dari tahap k ke tahap $k+1$, kita dapat menggunakan solusi dari tahap ke k tanpa kembali ke tahap awal. Oleh karena itu, misalnya untuk mencari nilai optimal maksimum dapat digunakan skema *dynamic programming* sebagai berikut :

$$dp[i] = \max(dp[i], dp[i-1] + c[i])$$

Maka dari itu persoalan dengan penyelesaian menggunakan *dynamic programming* harus memiliki karakteristik :

- Dapat dibagi menjadi beberapa tahap, yang pada setiap tahap hanya diambil satu keputusan
- Masing-masing tahap terdiri dari sejumlah *state* yang berhubungan dengan tahap tersebut

Adapun terdapat dua pendekatan yang digunakan dalam *dynamic programming* yaitu :

1) *top-down*

   Perhitungan dilakukan dari tahap 1, 2, ..., n-1, n. Biasanya pendekatan ini menggunakan metode iteratif.

2) *bottom-up*

   Perhitungan dilakukan dari tahap n, n-1, ..., 2, 1. Biasanya pendekatan ini memanfaatkan fungsi rekursif.

Langkah-langkah dalam pengembangan algoritma *dynamic programming* adalah sebagai berikut :

1) Karakteristikkan struktur solusi optimal
2) Definisikan secara rekursif nilai solusi optimal
3) Hitung nilai solusi optimal baik secara *top-down* maupun *bottom-up*
4) Rekonstruksi solusi optimal

Dalam penerapannya sendiri *dynamic programming* digunakan untuk menyelesaikan beberapa persoalan klasik pemrograman antara lain adalah persoalan *knapsack*, *coin change*, *maximum range sum*, dan *longest increasing subsequence*.

B. *Decrease & Conquer*

Decrease and Conquer adalah strategi pennyelesaian masalah dengan mereduksi persoalan menjadi dua sub-persoalan yang lebih kecil, tetapi selanjutnya hanya memproses satu persoalan saja. Hal tersebut berbeda dengan *divide and conquer* yang memproses semua sub-persoalan dan menggabungkan solusi dari seluruh sub-persoalan.

Strategi *decrease and conquer* terdiri dari dua tahapan utama yakni :

1) *Decrease*

   Memecah persoalan menjadi sub-persoalan yang lebih kecil

2) *Conquer*

   Memproses satu sub-persoalan secara rekursif

Terdapat tiga variansi strategi *decrease and conquer* yaitu :

1) *Decrease by a constant*

   Ukuran persoalan direduksi sebesar konstanta yang sama setiap iterasi algoritma. Biasanya konstanta sebesar 1.

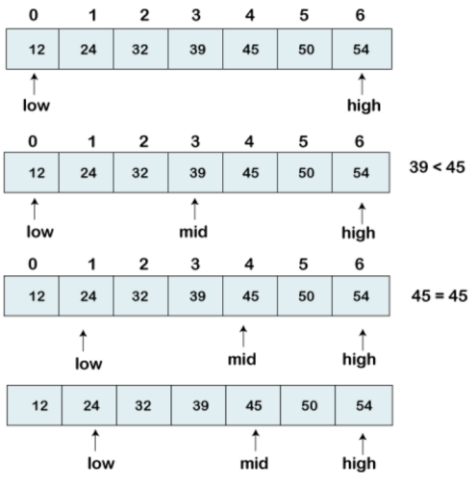
2) *Decrease by a constant factor*

   Ukuran persoalan direduksi sebesar faktor konstanta yang sama setiap iterasi algoritma. Biasanya konstanta sebesar 2.

3) *Decrease by variable size*

   Ukuran persoalan direduksi bervariasi pada setiap iterasi algoritma.

Adapun beberapa contoh algoritma *decrease and conquer* yang populer antara lain adalah *fast exponentiation* dan *binary search*.



Gambar 2. Ilustrasi Binary Search
Diambil dari
https://static.javatpoint.com/ pada 9 Mei 2022

Selain itu, terdapat pula algoritma *lower bound* yang merupakan hasil modifikasi dari algoritma *binary search*. Algoritma ini berfungsi untuk mencari lokasi dengan indeks terkecil pada array yang nilainya masih lebih besar atau sama dengan nilai masukan. Algoritma ini bekerja dengan terus melakukan *binary search* untuk kemungkinan indeks elemen terkecil yang masih memenuhi dan terus menyimpan nilai indeks yang memenuhi ke dalam sebuah variabel. Adapun implementasi algoritmanya adalah sebagai berikut :

```
int lower_bound(int *arr, int n, int val){
  int l = 0;
  int r = n-1;
  int res = n;
  while(l<=r){
    int mid = (l+r)/2;
    if(a[mid]>=val){
      res = mid;
      r = mid-1;
    }else{
      l = mid+1;
    }
  return res;
}
```

Dengan menggunakan algoritma tersebut dapat diperoleh indeks elemen terkecil yang masih lebih besar atau sama dengan val secara efisien yaitu hanya dengan $O(\log n)$.

C. *Longest Increasing Subsequence*

Longest Increasing Subsequence merupakan persoalan untuk mencari panjang terbesar dari array baru yang merupakan *subset* dari array masukan tetapi masih terurut membesar.

Pada persoalan ini akan diberikan sebuah array dengan n buah elemen lalu akan dicari LIS yang merupakan $i_1 < i_2 < \cdots < i_k, a[i_1] < a[i_2] < \cdots < a[i_k]$.

Misalnya yaitu $n = 8$ dan $A = \{-7,10,9,2,3,8,8,1\}$ memiliki LIS sepanjang 4 dengan $LIS = \{-7,2,3,8\}$. Contoh lainnya yaitu $n = 5$ dan $A = \{3,10,2,1,20\}$ memiliki LIS sepanjang 3 dan $LIS = \{3,10,20\}$

Persoalan LIS dapat diselesaikan dengan menggunakan *exhaustive search*, akan tetapi jumlah kemungkinan solusi yang ada adalah sebanyak 2^n kemungkinan sehingga tidak memungkinkan untuk nilai n yang besar.

Oleh karena itu, untuk mengatasi permasalahan ini dapat digunakan strategi *decrease and conquer*, *dynamic prrogramming*, atau menggunakan struktur data. Pada makalah ini sendiri hanya akan dibahas penyelesaian persoalan LIS dengan menggunakan *decrease and conquer* serta *dynamic programming*.

III. PEMBAHASAN

A. *Penyelesaian dengan Metode Dynamic Programming*

Penyelesaian dengan metode *dynamic programming* untuk persoalan LIS ini intinya adalah dengan mencari elemen sebelumnya yang tetap menjamin array yang terurut membesar serta panjang yang maksimum. Berikut adalah langkah-langkahnya.

1) Inisialiasi array sepanjang n dengan nilai 1
2) Array ini berisi informasi terkait panjang LIS maksimum dari suatu prefix array yang meyimpan informasi nilai LIS maksimum yang mungkin.
3) Lakukan iterasi untuk range indeks $i = [1 \ldots n-1]$ dengan terus memperbarui nilai array *dynamic programming* jika nilai elemen yang berada sebelum elemen array tersebut lebih kecil dengan mengikuti persamaan berikut :

$$dp[i] = \max(dp[i], dp[j] + 1), 0 \leq j < i$$

4) Nilai LIS merupakan nilai maksimum pada array *dynamic programming*

Berikut adalah contoh implementasi kodenya dalam bahasa cpp.

```
vector<int>dp(n,1);
for(int i=1;i<n;i++){
    for(int j=0;j<i;j++){
        tot++;
        if(a[i]<=a[j])continue;
        dp[i]=max(dp[i],dp[j]+1);
    }
}
int ans=dp[0];
for(int i=1;i<n;i++){
    ans=max(ans,dp[i]);
}
cout<<"Hasil LIS : "<<ans<<endl;
cout<<"Total Perbandingan : "<<tot<<endl;
```

Gambar 3. Kode Program LIS dengan dynamic programming
Diambil dari dokumentasi pribadi pada 19 Mei 2022

Kelebihan dari metode *dynamic programming* ini yaitu kita dapat dengan mudah memperoleh nilai dari setiap elemen array yang menyusun LIS dengan melakukan iterasi mundur. Berikut adalah langkah-langkah pencariannya:

1) Inisialiasi $cur = panjang\ LIS$ dan $last = \infty$
2) Lakukan iterasi array dari elemen terakhir ke arah depan
3) Setiap ditemukan elemen array yang nilainya cur dan kurang dari last maka *decrement* cur, ubah $last = a[i]$, dan masukkan $a[i]$ ke array jawaban sebagai elemen pertama
4) Salah satu kandidat solusi terdapat pada array jawaban

```
int cur=ans;
int last=INF;
deque<int>d;
for(int i=n-1;i>=0;i--){
    if(dp[i]==cur && a[i]<last){
        d.push_front(a[i]);
        cur--;
        last=a[i];
    }
}
for(int i=0;i<d.size();i++){
    cout<<d[i]<<" ";
}
cout<<endl;
```

Gambar 4. Kode Program LIS untuk menampilkan array hasil
Diambil dari dokumentasi pribadi pada 15 Mei 2022

Dalam menyelesaikan persoalan LIS, solusi program dengan *dynamic programming* mencari panjang LIS dengan menggunakan 2 *nested loop* dimana jumlah perulangan pada *loop* bagian dalam sebanyak jumlah elemen yang ada di posisi sebelumnya. Oleh karena itu kompleksitas dari pencarian panjang LIS adalah $O(n^2)$.

Sedangkan untuk pencarian keseluruhan elemen yang berada dalam LIS diperlukan sekali iterasi dari elemen paling terakhir sehingga nilai kompleksitasnya adalah $O(n)$.

Berikut adalah data jumlah perbandingan yang diperlukan untuk setiap kasus uji dengan nilai panjang array (n) yang divariasikan.

Tabel 1. Data Uji LIS dengan metode *dynamic programming*

Panjang array	Total perbandingan
5	10
20	190
50	1225
100	4950
200	19900

Berikut adalah salah satu hasil uji LIS dengan metode *dynamic programming* untuk $n = 5$.

```
5
134 100 135 19 162
Hasil LIS : 3
Total Perbandingan : 10
```

Gambar 5. Hasil uji LIS dengan metode dynamic programming
Diambil dari dokumentasi pribadi pada 19 Mei 2022

Sedangkan berikut adalah hasil plot data panjang array terhadap total perbandingan yang diperlukan untuk metode *dynamic programming*.

Gambar 6. Grafik Total Perbandingan Terhadap Panjang Array untuk Metode Dynamic Programming
Diambil dari dokumentasi pribadi pada 19 Mei 2022

Dari grafik di atas terlihat bahwa perubahan nilai total perbandingan terhadap panjang array meningkat secara kuadratik. Hal ini sesuai dengan besar kompleksitas dari algoritma LIS dengan *dynamic programming* yaitu $O(n^2)$.

B. Penyelesaian dengan Metode Decrease And Conquer

Selain dengan menggunakan metode *dynamic programming*, persoalan LIS juga dapat diselesaikan dengan menggunakan metode *decrease and conquer*. Dalam hal ini, algoritma *decrease and conquer* yang digunakan untuk menyelesaikan permasalahan yakni algoritma *lower bound*.

Inti penyelesaian permasalahan LIS dengan metode *decrease and conquer* ini adalah dengan terus mencari kandidat elemen terkecil untuk menggantikan elemen terkecil di array LIS yang masih lebih besar dibandingkan kandidat elemen tersebut. Kemudian bila ditemukan kandidat elemen yang lebih besar dibandingkan keseluruhan elemen di array LIS maka elemen tersebut akan dimasukkan ke array LIS. Berikut adalah langkah-langkah penyelesaiannya.

1) Inisialiasi *ans* yaitu array dinamis yang digunakan untuk menyimpan array LIS
2) Lakukan iterasi array masukan dari elemen awal
3) Pada setiap iterasi cari nilai indeks *ans* dari hasil *lower bound ans* untuk elemen array masukan yang sedang dilakukan iterasi
4) Jika indeks yang dikembalikan lebih kecil dari panjang array dinamis *ans* maka ganti elemen *ans* pada indeks tersebut dengan elemen array masukan yang sedang diiterasi. Hal ini bertujuan untuk membuat array LIS menjadi *lexicographycally minimum* dengan tetap mejamin seluruh elemen pada array LIS memiliki nilai yang saling berbeda
5) Sedangkan jika indeks yang dikembalikan nilainya sama dengan panjang array maka tambahkan elemen array masukan yang sedang diiterasi ke array dinamis *ans* dengan menjadikannya sebagai elemen terakhir. Hal ini dilakukan karena elemen array masukan yang sedang diiterasi ini nilainya lebih besar dibandingkan keseluruhan elemen pada array LIS sehingga perlu dimasukkan sebagai kandidat elemen LIS.

Berikut adalah implementasi fungsi *lower bound* pada bahasa cpp.

```
int lowerbound(vector<int>ans,int e,int *totalPerbandingan){
    int l=0;
    int r=sz(ans)-1;
    int ret=sz(ans);
    while(l<=r){
        int mid=(l+r)/2;
        (*totalPerbandingan)++;
        if(ans[mid]>=e){
            ret=mid;
            r=mid-1;
        }else{
            l=mid+1;
        }
    }
    return ret;
}
```

Gambar 7. Kode Program LIS untuk Fungsi Lower Bound
Diambil dari dokumentasi pribadi pada 19 Mei 2022

Sedangkan berikut adalah implementasi algoritma program utama pada bahasa cpp.

```
int totalPerbandingan=0;
vector<int>ans;
for(int i=0;i<n;i++){
    int idx=lowerbound(ans,a[i],&totalPerbandingan);
    totalPerbandingan++;
    if(idx==ans.size()){
        ans.push_back(a[i]);
    }else{
        ans[idx]=a[i];
    }
}
cout<<"Hasil LIS : "<<ans.size()<<endl;
cout<<"Total Perbandingan : "<<totalPerbandingan<<endl;
```

Gambar 8. Kode Program LIS dengan Decrease And Conquer
Diambil dari dokumentasi pribadi pada 19 Mei 2022

Dalam menyelesaikan persoalan LIS dengan metode *decrease and conquer* hanya diperlukan satu *loop* saja pada program utama yang digunakan untuk melakukan iterasi seluruh elemen pada array masukan. Kemudian pada setiap iterasi dilakukan pemanggilan fungsi *lower bound* yang memiliki kompleksitas waktu $O(\log n)$ sehingga total kompleksitas waktu dari keseluruhan program adalah $O(n \log n)$.

Akan tetapi, salah satu kelemahan dari metode ini yaitu tidak dimungkinkannya pencarian keseluruhan elemen LIS secara efektif seperti halnya pada metode *dynamic programming*. Hal ini dikarenakan program hanya menyimpan kandidat elemen yang paling optimal untuk LIS dan bukan kandidat yang sebenarnya dipilih pada solusi akhir.

Berikut adalah data jumlah perbandingan yang diperlukan untuk setiap kasus uji dengan nilai panjang array (n) yang divariasikan.

Tabel 2. Data Uji LIS dengan metode *decrease and conquer*

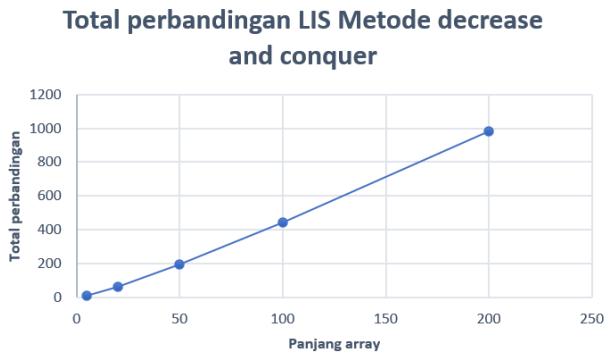
Panjang array	Total perbandingan
5	10
20	64
50	194
100	443
200	984

Berikut adalah salah satu hasil uji LIS dengan metode *decrease and conquer* untuk $n = 20$.

```
20
157 21 114 94 40 2 57 50 33 52 40 175 89 38 104 46 135 114 64 80
Hasil LIS : 7
Total Perbandingan : 64
```

Gambar 9. Hasil uji LIS dengan Metode Decrease And Conquer
Diambil dari dokumentasi pribadi pada 19 Mei 2022

Sedangkan berikut adalah hasil plot data panjang array terhadap total perbandingan yang diperlukan untuk metode *decrease and conquer*.



Gambar 10. Grafik Total Perbandingan Terhadap Panjang Array untuk Metode Decrease And Conquer
Diambil dari dokumentasi pribadi pada 19 Mei 2022

Dari grafik di atas terlihat bahwa perubahan nilai total perbandingan terhadap panjang array cenderung meningkat secara linier. Hal ini sesuai dengan besar kompleksitas waktu dari algoritma ini yang telah dikalkulasi sebelumnya yakni sebesar $O(n \log n)$ yang cenderung bersifat linear walaupun tidak sepenuhnya linear.

IV. Kesimpulan

Dalam menyelesaikan persoalan LIS dapat digunakan 2 metode yakni *dynamic programming* dan *decrease and conquer*. Berdasarkan hasil perhitungan dan pengujian diperoleh kompleksitas waktu untuk metode *dynamic programming* sebesar $O(n^2)$. Sedangkan metode *decrease and conquer* memiliki kompleksitas waktu sebesar $O(n \log n)$.

Akan tetapi, metode *decrease and conquer* memiliki kelemahan yaitu tidak bisa menghasilkan keseluruhan elemen array solusi LIS secara efektif. Sedangkan metode *dynamic programming* dapat menghasilkan array solusi LIS dengan kompleksitas waktu $O(n)$ saja.

Oleh karena itu dapat disimpulkan bahwa penggunaan metode *decrease and conquer* lebih optimal dibandingkan metode *dynamic programming* untuk menyelesaikan persoalan LIS jika tidak diperlukan array solusi LIS. Sedangkan jika diperlukan array solusi LIS maka metode *dynamic programming* terbukti lebih efisien.

V. Ucapan Terima Kasih

Puji syukur kehadirat Tuhan yang Maha Esa atas segala rahmat, karunia, serta taufik dan hidayah-Nya sehingga penulis dapat menyelesaikan makalah yang berjudul "Analisis Perbandingan Efisiensi Penyelesaian Persoalan *Longest Increasing Subsequence* dengan Metode *Decrease & Conquer* dan *Dynamic Programming*" sebagai pemenuhan tugas akhir semester II pada mata kuliah Strategi Algoritma IF2211.

Penulis juga ingin berterimakasih kepada Ibu Dr. Masayu Leylia Khodra, S.T., M.T. dan ibu Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen K1 dan K2 yang telah membagikan ilmunya kepada kami, para mahasiswa, selama satu semester ini. Penulis juga beterimakasih kepada seluruh pihak yang telah berkontribusi baik secara langsung maupun tidak langsung terhadap kelancaran penulisan makalah ini yang namanya tidak bisa saya sebutkan satu persatu. Terakhir, penulis ingin mengucapan permohonan maaf apabila terdapat kesalahan dalam penulisan makalah ini.

VI. Referensi

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf Diakses pada 9 Mei 2022
[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Decrease-and-Conquer-2021-Bagian1.pdf Diakses pada 9 Mei 2022
[3] Halim, Steven, dan Felix Halim. 2013. Competitive Programming 3. Singapore. Diakses pada 9 Mei 2022

VII. 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, 19 Mei 2022

Primanda Adyatma Hafiz (13520022)