Penyelesaian Permasalahan *Longest Increasing Subsequence* (LIS) dengan Menggunakan *Dynamic Programming* dan Struktur Data *Segment Tree*

Irfan Sofyana Putra / 13517078
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517078@std.stei.itb.ac.id

***Abstract*—Permasalahan "Longest Increasing Subsequence" atau LIS adalah salah satu permasalahan klasik di bidang *computer science*. Permasalahan LIS ini dapat diselesaikan dengan konsep *Dynamic Programming*. Akan Tetapi, dengan penggunaan konsep *dynamic programming* saja, kompleksitas waktu untuk menyelesaikan permasalahan ini masih tinggi. Terdapat alternatif lain untuk menyelesaikan masalah ini dengan kompleksitas waktu yang lebih baik, yaitu dengan menggunakan bantuan struktur data *segment tree* yang merupakan struktur data dengan paradigma *divide and conquer*. Makalah ini akan membahas bagaimana permasalahan LIS tersebut diselesaikan dengan *dynamic programming* dan bantuan struktur data *segment tree*.**

Keywords—component; Longest Increasing Subsequence, LIS, Dynamic programming, Segment tree, Divide and conquer, Struktur data.

I. PENDAHULUAN

Permasalahan *Longest Increasing Subsequence*(LIS) adalah sebuah permasalahan klasik di bidang *computer science.* Permasalahan LIS ini berupa diberikan sekumpulan bilangan $a_1, a_2, a_3, ..., a_n$. Kemudian kita harus mencari *subsequence* terpanjang yang dimana elemen-elemen yang dipilihnya harus selalu menaik. Subsequence yang dipilih tidak harus selalu berurutan dan unik.

Untuk menyelesaikan permasalahan ini, kita dapat melakukan beberapa strategi penyelesaian masalah seperti *bruteforce*. Akan tetapi, apabila kita hanya menggunakan *bruteforce*, maka kompleksitas waktu yang didapatkan menjadi sangat mahal karena kita harus mencoba semua kemungkinan elemen yang diambil. Kompleksitas yang dihasilkan adalah $O(n^2)$

Apabila kita coba lebih melihat permasalahan ini dengan seksama, maka sebenarnya kita dapat melakukan *dynamic programming*. Akan tetapi dengan strategi penyelesaian *dynamic programming,* kompleksitas yang dihasilkan masih cukup mahal yaitu $O(n^2)$. Untuk nilai n yang cukup besar, maka program akan berjalan lambat.

Apabila kita coba untuk menganalisis lebih jauh lagi, maka proses pencarian solusi dengan *dynamic programming* dapat dioptimasi lagi dengan sebuah struktur data *segment tree*. Struktur data *segment tree* sendiri adalah sebuah struktur data yang mengaplikasikan *binary tree*. Paradigma yang digunakan oleh *segment tree* sendiri adalah *divide and conquer*.

Pada makalah ini, penulis akan membahas mengenai permasalahan LIS, *segment tree*, dan *dynamic programming*, lalu bagaimana menyelesaikan permasalahan tersebut dengan menggunakan ketiga konsep tersebut.

II. TEORI DASAR

A. Permasalahan *Longest Increasing Subsequence* (LIS)

Permasalahan *longest increasing subsequence* atau biasa disingkat LIS adalah salah satu contoh permasalahan klasik yang ada di *computer science*. Permasalahan yang harus diselesaikan adalah mencari *subsequence* terpanjang dari sekumpulan barisan bilangan dimana bilangan-bilangan yang dipilih harus selalu menaik. Perlu diketahui juga bahwa bilangan-bilangan yang dipilih tidak harus selalu berurutan. Permasalahan mencari *longest increasing subsequence* (LIS) dipelajari dalam banyak multidisiplin ilmu seperti *random matrix theory, representation theory,* dan fisika.

Contoh permasalahan mencari LIS adalah sebagai berikut: Misalkan diberikan 16 barisan bilangan [1, 9, 5, 13, 3, 11, 7, 15, 2, 10, 6, 14, 4, 12, 8, 16]. Maka LIS dari barisan bilangan tersebut adalah [1, 3, 7, 10, 12, 16]. Panjang dari *subsequence* ini adalah 6. Perlu diingat juga bahwa LIS dari suatu barisan bilangan bisa saja tidak unik, sebagaimana untuk contoh ini, kita akan menemukan juga bahwa [1, 5, 7, 10, 12, 16] atau [1, 5, 7, 10, 14, 16] adalah LIS dari barisan bilangan tersebut.

Pada permasalahan ini kita akan membatasi bahwa barisan bilangan hanya memuat bilangan-bilangan bulat positif.

B. *Dynamic Programming*

Dynamic Programming atau program dinamis adalah sebuah metode penyelesaian suatu masalah dengan cara menguraikan solusi permasalahan menjadi beberapa tahapan sedemikian sehingga solusi dari persoalan dapat dipandang dari serangkaian keputusan yang saling berkaitan.

Suatu permasalahan dapat diselesaikan dengan paradigma *dynamic programming* apabila memiliki kriteria sebagai berikut:

1. Terdapat sejumlah berhingga pilihan yang mungkin
2. Solusi pada setiap tahap yang dibangun dari hasil solusi tahap sebelumnya.
3. Kita menggunakan persyaratan optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap.

Pada dasarnya paradigma penyelesaian masalah dengan *dynamic programming* tidak terlalu jauh berbeda. Cara menyelesaikan masalah dengan metode ini sama-sama membentuk solusi secara bertahap. Hanya saja, perbedaan dari kedua paradigma ini adalah dari letak pengambilan keputusannya. Dengan penyelesaian *greedy,* maka solusi yang diambil adalah solusi yang pada saat itu menghasilkan solusi optimal dengan berharap solusi global juga akan menjadi optimal. Sementara itu, dengan penyelesaian *dynamic programming*, maka solusi yang diambil adalah solusi yang menghasilkan solusi optimal tanpa harus mengambil solusi optimal pada saat tertentu.

Pada kebanyakan persoalan, algoritma greedy tidak dapat memberikan solusi yang optimal. Hal ini dikarenakan pada algoritma greedy, pengambilan keputusan pada setiap langkah tidak pernah mempertimbangkan lebih jauh apakah pilihan tersebut pada langkah-langkah selanjutnya merupakan pilihan yang juga menghasilkan solusi yang optimal. Dari sejumlah pilihan yang ada, tidak akan pernah diketahui pilihan mana yang terbaik hingga menuju proses yang lebih jauh ke depan.

Strategi penyelesaian masalah lain yang mirip dengan *dynamic programming* adalah *bruteforce*. Persamaan diantara keduanya adalah sama-sama untuk mencoba semua solusi yang dapat dicoba pada saat tertentu, hanya terdapat perbedaan yang sangat berpengaruh, yaitu *bruteforce* tidak pernah menyimpan hasil pencarian. Sementara *dynamic programming* akan selalu menyimpan hasil pencarian. Perbedaan inilah yang menjadi pembeda mengapa algoritma dengan *dynamic programming* jauh lebih cepat dibanding dengan *bruteforce.*

Karakteristik yang lebih mendetail mengenai persoalan *dynamic programming* adalah

1. Persoalan dapat dibagi menjadi beberapa tahap (stage), yang pada setiap tahap hanya diambil satu keputusan.
2. Masing-masing tahap terdiri dari sejumlah status (state) yang berhubungan dengan tahap tersebut. Secara umum, status merupakan bermacam kemungkinan masukan yang ada pada tahap tersebut.
3. Hasil dari keputusan yang diambil pada setiap tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap berikutnya.
4. Ongkos (cost) pada suatu tahap meningkat secara teratur (steadily) dengan bertambahnya jumlah tahapan
5. Ongkos pada suatu tahap bergantung pada ongkos tahap-tahap yang sudah berjalan dan ongkos pada tahap tersebut.
6. Keputusan terbaik pada suatu tahap bersifat independen terhadap keputusan yang dilakukan pada tahap sebelumnya.
7. Adanya hubungan rekursif yang mengidentifikasikan keputusan terbaik untuk setiap status pada tahap k memberikan keputusan terbaik untuk setiap status pada tahap k + 1.
8. Prinsip optimalitas berlaku pada persoalan tersebut.

Dynamic programming memiliki dua pendekatan, yaitu *dynamic programming* maju (*forward/up-down*) dan *dynamic programming* mundur(*backward* atau *bottom-up*).

Misalkan $x_1, x_2, x_3, ..., x_n$ menyatakan peubah keputusan yang harus dibuat untuk masing-masing tahap 1, 2, 3, …, n. Maka

1. *Dynamic programming* maju akan bergerak mulai dari tahap 1, terus maju untuk tahap ke 2, 3, dan seterusnya sampai tahap ke n.
2. *Dynamic programming* mundur mulai dari tahap n, terus mundur ke tahap n – 1, n – 2, dan seterusnya sampai tahap 1.

Dynamic *programming* memiliki prinsip optimalitas, yakni:

1. *Dynamic programming* maju
   cost pada tahap $k+1$ = (*cost* yang dihasilkan pada tahap k) + *cost* dari tahap k ke tahap $k+1$, untuk $k = 1, 2, ..., n-1$
2. *Dynamic programming* mundur
   cost pada tahap k = (*cost* yang dihasilkan pada tahap $k+1$) + *cost* dari tahap ($k+1$) ke tahap k. Untuk $k = n, n-1, n-2, ..., 1$

Untuk menyelesaikan persoalan dengan strategi *dynamic programming*, maka kita dapat melakukan langkah-langkah berikut ini:

1. Karakteristikkan struktur solusi optimal.
2. Definisikan secara rekursif nilai solusi optimal.
3. Hitung nilai solusi optimal secara maju atau mundur.
4. Konstruksi solusi optimal

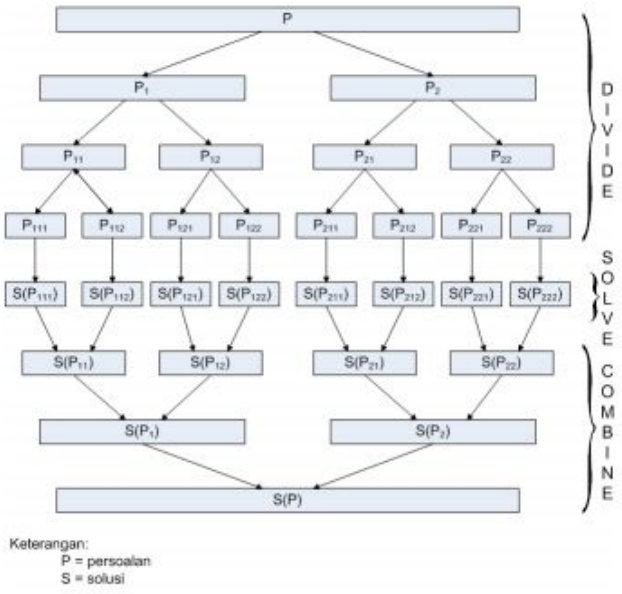
Satu hal yang harus diingat pada *dynamic programming* adalah sejatinya ia adalah suatu penyelesaian dengan *bruteforce*/*exhaustive search*. Akan tetapi, nilai hasil pencariannya akan selalu disimpan. Sehingga kita tidak perlu melakukan pencarian berkali-kali.

C. Algoritma *Divide and conquer*

Algoritma *divide and conquer* adalah suatu algoritma penyelesaian masalah selain *bruteforce, greedy,* dan *dynamic programming*. *Divide and conquer* adalah suatu penyelesaian masalah dengan membagi masalah tersebut menjadi beberapa sub-masalah yang lebih kecil. Lalu masing-masing dari sub-masalah tersebut dicari solusi nya dan kemudian digabungkan lagi untuk mendapatkan solusi seperti permasalahan awal. Metode penyelesaian masalah dengan *divide and conquer* umumnya memiliki kompleksitas yang lebih baik dibanding dengan *bruteforce/exhaustive search.*

Lebih spesifik lagi, algoritma *divide and conquer* adalah suatu algoritma, yang terdiri dari beberapa langkah, yaitu:

1. Divide
   Membagi persoalan menjadi beberapa sub-masalah yang memiliki kemiripan dengan persoalan semula namun berukuran lebih kecil (idealnya berukuran hampir sama)
2. Conquer
   memecahkan (menyelesaikan) masing-masing sub-masalah yang secara rekursif.
3. Combine
   Menggabungkan solusi masing-masing sub-masalah sehingga membentuk solusi persoalan semula.



Gambar 1. Ilustrasi *Divide and Conquer.* Sumber: Slide Kuliah "Algoritma *Divide and Conquer",* Rinaldi Munir.

Objek persoalan yang dibagi pada permasalahan *divide and conquer* biasanya berupa masukan *instance* persoalan yang berukuran *n* seperti:

1. Tabel
2. Matriks
3. Eksponen
4. Lain-lain yang bergantung pada persoalannya

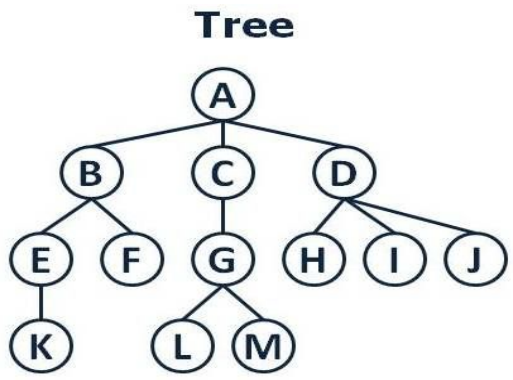
Salah satu syarat suatu permasalahan dapat diselesaikan dengan *divide and conquer* adalah permasalahan tersebut dapat dibagi menjadi beberapa sub-masalah yang memiliki karakteristik yang sama.

D. Struktur Data Tree

Struktur data tree adalah sebuah struktur data non-linier yang menggambarkan hubungan hierarkis (*one to many relationship)* antara elemen-elemen. Tree bisa digambarkan menjadi sekumpulan simpul dengan satu elemen simpul disebut *root* lalu simpul lainnya terbagi menjadi beberapa himpunan-himpunan yang tak saling berhubungan.

Beberapa istilah dalam struktur data *tree* adalah sebagai berikut:

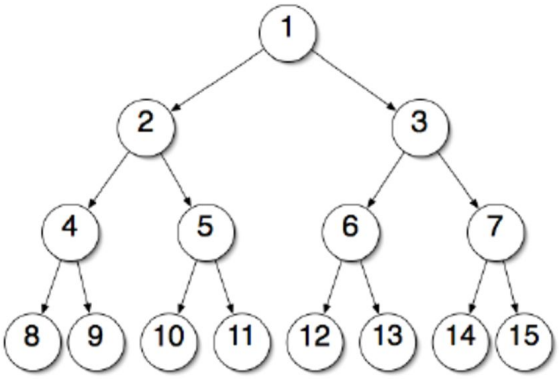
1. Simpul
   Simpul atau biasa juga disebut *node* adalah sebuah elemen pada struktur data tree yang biasanya menyimpan suatu informasi yang diperlukan. Umumnya penggambaran simpul ini adalah dengan lingkaran.
2. *Root*
   Root atau akar adalah sebuah simpul yang ada pada struktur data tree dan letaknya ada di hierarki paling atas.
3. *Size*
   Size adalah banyaknya simpul yang ada pada suatu tree.
4. *Height*
   Height adalah banyaknya tingkatan dalam suatu tree
5. *Predecessor*
   Predecessor adalah suatu istilah yang menyatakan simpul yang berada satu level di atas suatu simpul.
6. *Successor*
   Successor adalah suatu istilah yang menyatakan simpul yang berada satu level di bawah suatu simpul
7. *Ancestor*
   Seluruh simpul yang terletak sebelum simpul tertentu dan terletak pada jalur yang sama
8. *Descendant*
   Descendant adalah seluruh simpul yang terletak sesudah simpul tertentu dan terletak pada jalur yang sama.
9. *Parent*
   Predecessor satu level di atas suatu simpul.
10. *Child*
    Successor satu level di bawah suatu simpul.
11. *Sibling*
    Simpul-simpul yang memiliki parent yang sama dengan suatu simpul
12. *Subtree*
    Bagian dari tree yang berupa suatu simpul beserta descendantnya dan memiliki semua karakteristik dari tree tersebut
13. *Leaf*
    Leaf adalah semua simpul yang tidak memiliki *child*.
14. *Degree*
    Degree adalah banyaknya *child* dari sebuah simpul.



Gambar 2. Ilustrasi Struktur Data Tree.
Sumber:https://codepumpkin.com/

Umumnya tree akan dibagi menjadi dua macam, yaitu:

1. General Tree
   Sebuah tree biasa yang tidak memiliki batasan apapun.
2. Binary Tree
   Binary tree adalah sebuah tree dimana setiap simpul yang bukan merupakan *leaf* akan memiliki maksimal *child* sebanyak dua buah.



Gambar 3. Ilustrasi Binary Tree
Sumber : https://www.researchgate.net

E. Teorema Master

Dalam menghitung kompleksitas waktu suatu algoritma yang berjalan dengan konsep divide and conquer, teorema master dapat digunakan untuk mempermudah perhitungan. Teorema master memberikan hubungan antara bentuk rekurens suatu kompleksitas dengan kompleksitas waktu asimptotik. Dengan menggunakan teorema master, hanya dengan mengetahui bentuk rekurens dari $T(N)$ saja, nilai Big-O dapat ditentukan dengan mudah.

Teorema master mengacu pada bentuk rekurens:

$T(n) = aT(\frac{n}{b}) + cn^d$

Nilai a menyatakan banyaknya submasalah hasil dari pemecahan masalah utama. N merupakan ukuran dari masalah utama. Sehingga $\frac{N}{b}$ adalah ukuran submasalah hasil pemecahan masalah utama. Sedangkan cn^d adalah merepresentasikan komputasi yang diperlukan diluar dari fungsi rekursif.

Nilai Big-O yang dihasilkan oleh teorema master akan bergantung kepada nilai a, b, dan f(n) ekspresi rekurens kompleksitas waktu. Terdapat 3 kasus dalam menentukan nilai Big-O, yakni:

1. $T(n) = O(n^d)$, jika $a < b^d$
2. $T(n) = O(n^d \log n)$, jika $a = b^d$
3. $T(n) = O(n^{\log_b a})$, jika $a > b^d$

F. Struktur Data *Segment Tree*

Segment Tree adalah sebuah struktur data yang berbentuk *binary tree* yang digunakan untuk menyimpan suatu interval atau *segment*. Setiap simpul yang ada pada *binary* tersebut akan menyimpan suatu informasi tertentu untuk suatu interval.

Misalkan kita memiliki sebuah tree T dan array A[1:N], maka struktur data *segment tree* yang dimaksud akan menyimpan:

1. *Root* dari T akan merepresentasikan informasi yang dicakup oleh array A[1:N]
2. Setiap *leaf* dari T akan merepresentasikan elemen tunggal A[i] untuk setiap $1 \leq i \leq N$
3. Setiap simpul selain *root* dan *leaf* akan merepresentasikan gabungan dari interval A[i:j] sedemikian sehingga $0 \leq i < j$

Proses kerja dari struktur data *segment tree* adalah membagi rentang-rentang terus menerus hingga tidak dapat dibagi lagi. Kemudian informasi untuk setiap simpul akan digabungkan. Oleh karena itu, prinsip dari struktur data *segment tree* mengikuti paradigma *divide and conquer*.

Lebih rinci, proses kerja dari struktur data *segment tree* terhadap suatu array A[1:N] adalah sebagai berikut:

1. *Root* dari tree yang merepresentasikan *segment tree* tersebut akan mencakup seluruh informasi yang ada pada array A. Alias mencakup seluruh rentang dari 1 hingga N
2. Apabila suatu rentang hanya terdiri dari 1 elemen tunggal, maka simpul pada tree tersebut hanya mencakup informasi dari array berindeks rentang tersebut.
3. Apabila suatu rentang terdiri dari 2 elemen atau lebih (l, r), maka kita definisikan nilai median dari dua rentang tersebut $m = ((l + r) \; div \; 2)$. Selanjutnya kita akan membagi rentang tersebut menjadi rentang kiri dan kanan, sedemikian sehingga rentang kiri akan menyimpan informasi untuk A[l:m] dan rentang kiri akan menyimpan informasi untuk A[m+1:r]
4. Lakukan proses rekursif untuk masing-masing rentang kiri dan rentang kanan.
5. Gabungkan kembali hasil proses rekursif rentang kiri dan rentang kanan untuk mendapatkan informasi dari rentang semua (l, r)

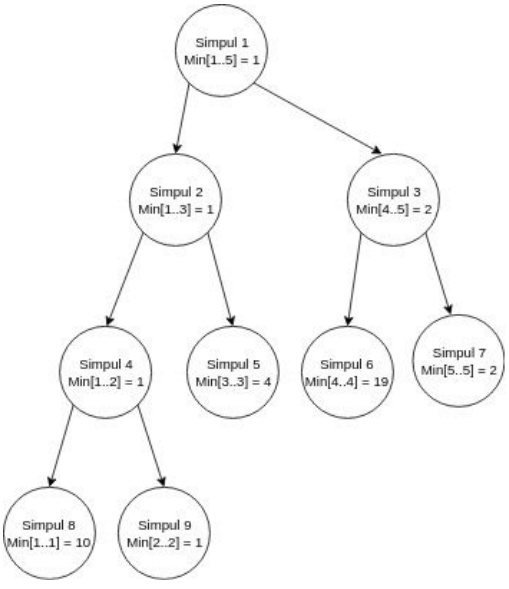
Struktur data *segment tree* biasanya digunakan ketika kita memerlukan pemrosesan *update* dan *query* secara cepat. Adapun maksudnya *update* adalah meng*update* suatu elemen array yang kemudian akan mengubah struktur data *segment tree* tersebut lalu *query* adalah operasi untuk mendapatkan suatu informasi tertentu untuk rentang tertentu.

Salah satu pengaplikasian *segment tree* adalah pada persoalan klasik *range minimum query*. Dimana kita memiliki sebuah array A[1:N] dengan batasan $1 \leq N \leq 100000$. Lalu kita memiliki Q buah *query* dengan batasan $1 \leq Q \leq 100000$. Untuk setiap *query*, kita harus menentukan berapa elemen terkecil yang ada pada array berindeks l hingga r.

Secara sekilas permasalahan ini mudah diselesaikan dengan *bruteforce*. Akan tetapi kompleksitas waktu dengan solusi *bruteforce* adalah O(n^2). Untuk n mencapai 100000, maka algoritma akan berjalan sangat lambat.

Salah satu solusi untuk menyelesaikan permasalahan ini adalah dengan menggunakan *segment tree*, dimana setiap simpul pada *tree* akan berisi nilai terkecil dari rentang yang dia simpan. Dengan menggunakan solusi *segment tree* seperti ini, pencarian solusi akan lebih efektif. Kompleksitas yang dihasilkannya adalah $O(n \log n)$.

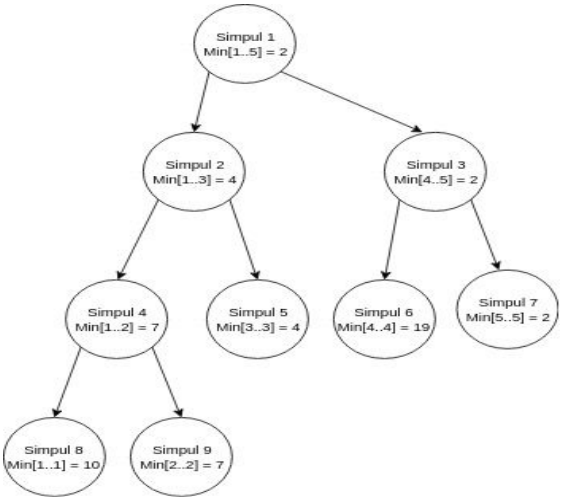
Contoh *instance* permasalahan *Range minimum query* untuk array yang berukuran 5 dan nilainya [10, 1, 4, 19, 2]. Dengan menggunakan struktur data *segment tree*, maka kita akan memiliki informasi *tree* sebagai berikut:



Gambar 4. Ilustrasi *Segment Tree*
Sumber : Dokumentasi Pribadi

Nilai min[i..j] pada setiap simpul melambangkan nilai minimal dari array yang berindeks i hingga j. Perhatikan bahwa apabila kita sudah memiliki informasi seperti yang ada pada *tree*, maka apabila kita menginginkan *query* nilai minimal dari indeks ke 1 hingga ke 4, kita tidak memerlukan iterasi satu per satu sebanyak 4 kali, melainkan cukup dengan mengetahui nilai dari simpul 2 (min[1..3]) dan simpul 6(min[4..4]) lalu membandingkannya. Untuk *array* dengan ukuran yang besar, maka proses dengan menggunakan *segment tree* akan terasa jauh lebih cepat dibanding dengan *bruteforce.*

Misalkan kita menginginkan proses *update* suatu nilai elemen *array*, maka kita hanya perlu meng*update* beberapa simpul saja. Sebagai contoh apabila kita meng*update* elemen array kedua dengan nilai 7, maka yang kita perlukan adalah meng*update* nilai min[i..j] pada simpul 1, 2, 4, 9. Sehingga *segment tree* yang kita miliki akan menjadi



Gambar 4. *Segment Tree* Setelah *update*
Sumber : Dokumentasi Pribadi

Perhatikan bahwa proses untuk membuat *tree* tersebut akan berlangsung secara linier ($O(n)$), karena kita akan mencoba membuat simpul untuk setiap elemen pada *array* dan menggabungkannya.

Sementara itu, untuk proses *query* dan *update*, proses akan berjalan dalam waktu yang cepat yaitu sekitar $\log n$. Dimana n adalah banyak elemen pada *array*. Hal ini terjadi karena pada setiap saat kita hanya ingin memproses rentang yang masih diperlukan saja dan membuang rentang yang sudah tidak diperlukan.

III. PEMBAHASAN

Seperti yang sudah dijelaskan sebelumnya, LIS atau *longest increasing subsequence* adalah suatu permasalahan dimana kita harus mencari *subsequence* terpanjang dari sebuah *sequence* sedemikian sehingga bilangan-bilangan yang dipilih pada *sequence* harus selalu menaik. Pada makalah ini persoalan dibatasi untuk *sequence* yang berisi bilangan bulat positif.

Salah satu solusi untuk menyelesaikan permasalahan ini adalah dengan menggunakan *bruteforce* atau *exhaustive search*. Solusi dengan menggunakan *bruteforce* atau *exhaustive search* akan memiliki kompleksitas $O(2^n)$ dimana

untuk setiap elemen kita akan coba untuk diambil atau tidak diambil. Solusi dengan *bruteforce* ini jelaslah sangat lambat.

Solusi lain yang bisa digunakan untuk menyelesaikan permasalah ini adalah dengan menggunakan *dynamic programming*. Pendekatan *dynamic programming* yang dilakukan adalah dengan melakukan *dynamic programming* maju. Misalkan kita mempunyai fungsi $dp(i)$ yang menyatakan panjang maksimal dari LIS apabila kita mengambil *sequence* ke-i sebagai elemen terakhir yang diambil. maka dengan konsep *dynamic programming*, kita akan mendapatkan bahwa nilai dari $dp(i) = max(dp(j) + 1))$ untuk setiap $j < i$ dan $sequence[j] < sequence[i]$ dengan nilai $dp(1) = 1$ karena *sequence* dengan satu elemen pasti memiliki LIS dengan panjang 1. LIS terpanjang yang bisa kita dapatkan adalah nilai maksimal dari seluruh dp(i) yang ada untuk setiap i dari 1 sampai n, dimana n adalah banyak *sequence*.

Sebagai contoh, kita memiliki *sequence* [5, 1, 9, 2]. Maka proses pencarian *dynamic programming* akan melalui langkah-langkah berikut ini:

1. i = 1. Ini merupakan *base case* dari persoalan, oleh karena itu dp(1) = 1
2. i = 2. Kita coba untuk setiap $j < i$, lalu mencari dp(j) yang paling maksimal dengan $sequence[j] < sequence[i]$. Akan tetapi tidak ada *sequence* yang lebih kecil dari 1. Sehingga nilai dp(2) = 1
3. i = 3. Kita coba untuk setiap $j < i$, lalu mencari dp(j) yang paling maksimal dengan $sequence[j] < sequence[i]$. Ditemukan bahwa nilai j yang memenuhi adalah 1 atau 2. Sehingga nilai dp(3) = dp(1) + 1 = 2
4. i = 4. Kita coba untuk setiap $j < i$, lalu mencari dp(j) yang paling maksimal dengan $sequence[j] < sequence[i]$ .Ditemukan bahwa nilai j yang memenuhi adalah 2. Sehingga nilai dp(4) = dp(2) + 1 = 2

Nilai maksimal dari seluruh nilai dp(i) untuk i dari 1 sampai 4 adalah 2, oleh karena LIS yang kita dapatkan memiliki panjang 2, yaitu [5, 9] atau [1,9] atau [1,2].

Solusi dengan menggunakan *dynamic programming* seperti contoh di atas akan memiliki kompleksitas $O(n^2)$. Hal ini dikarenakan untuk setiap i[1..n] kita harus melakukan pencarian sebanyak (i-1), sehingga proses yang kita perlukan adalah $0 + 1 + 2 + ... + n - 1 = \frac{(n-1)n}{2} = O(n^2)$, Untuk n yang besar, proses pencarian solusi dengan cara ini akan berjalan cukup lambat.

Perhatikan bahwa, solusi dengan *dynamic programming* seperti ini bisa kita optimasi lagi dengan menggunakan bantuan struktur data *segment tree*. Permasalahan yang membuat cara sebelumnya berjalan lambat adalah mencari nilai dp(j) maksimal dimana $sequence[j] < sequence[i]$. Apabila kita analisis, ternyata permasalahan ini sebenarnya bisa kita modelkan menjadi, mencari elemen terbesar yang ada pada suatu *array* dengan indeks $[1: sequence[i] - 1]$. Oleh karena itu kita bisa menggunakan bantuan struktur data *segment tree* agar proses menjadi lebih cepat.Adapun setiap simpul yang ada pada *Segment Tree* yang digunakan untuk persoalan ini akan menyimpan informasi indeks berisi elemen maksimal untuk suatu rentang tertentu.

Sehingga solusi kombinasi dari *dynamic programming* dan bantuan struktur data *segment tree* adalah sebagai berikut:

1. Kita memiliki sebuah fungsi dp(i) yang menyatakan panjang maksimal dari LIS apabila kita mengambil sequence ke-i sebagai elemen terakhir yang diambil.
2. Kita siapkan sebuah *array* arr, dimana arr[i] akan berisi informasi LIS terpanjang dengan akhiran bilangan i.
3. Siapkan sebuah *array* parent, dimana nilai dari *parent*[i] adalah indeks terakhir dari *sequence* yang dipilih sebelum *sequence* dengan indeks-ke i dipilih.
4. *Base case* untuk persoalan ini adalah saat i = 1. Nilai dari dp(i) adalah 1. Karena sequence dengan panjang 1 pasti memiliki LIS dengan panjang 1. Lakukan *update* nilai arr[1] dengan 1
5. Untuk setiap i dari 2 sampai n, dimana n adalah panjang *sequence*, maka kita akan mencari nilai maksimal dari arr dengan indeks 1 sampai *sequence*[i]-*1*. Jika tidak ditemukan nilai maksimalnya, maka dp(i) akan bernilai 1. Jika ditemukan maka nilai dp(i) akan menjadi nilai dari arr[*query(1, sequence[i]-1)*] + 1. Fungsi *query* adalah fungsi yang diimplementasikan oleh *segment tree* yang menghasilkan indeks dimana nilai maksimal arr dari indeks 1 hingga *sequence[i]-1* ada pada indeks tersebut. Selain itu, kita juga mengubah *parent[i]* menjadi indeks terakhir dari bilangan yang dipilih sebelum bilangan (*sequence*[i]) dipilih . Lalu Lakukan *update* nilai arr[sequence[i]] dengan dp(i). Proses *update* akan dilakukan oleh *segment tree*.
6. LIS pada *sequence* ini dapat kita cari dengan mencari nilai maksimal untuk setiap dp(i) dari 1 hingga n.
7. Untuk mengeluarkan bilangan apa saja yang dipilih pada *sequence*, maka kita cukup untuk melakukan iterasi lewat *array* parent sudah diproses.

Dengan menggunakan solusi ini, maka untuk setiap i dari 1 hingga n, dimana n adalah panjang *sequence*, maka kita akan melakukan operasi *query* dan *operasi* update yang dilakukan oleh *segment tree*. Seperti yang sudah dijelaskan sebelumnya, operasi dengan *query* atau *update* pada *segment tree* akan berjalan dalam waktu *log n*. Sehingga kompleksitas solusi untuk permasalahan ini menjadi $O(n \log n)$.

Contoh penerapan:

Misalkan terdapat *sequence* dengan panjang 6, [6, 1, 2, 9, 3, 10]. Maka penyelesaian dengan *dynamic programming* dan *segment tree* akan melalui langkah-langkah berikut ini:

1. i = 1, maka dp(1) = 1. Lalu nilai arr[6] menjadi 1. Ubah nilai parent[1] menjadi 1.
2. i = 2, lakukan *query(1, 0)*, yaitu mencari nilai arr maksimal yang berindeks 1 sampai 0. Karena tidak ada nilai arr yang memenuhi maka nilai dp(2) menjadi 1. update nilai arr[1] menjadi 1. Ubah nilai parent[2] menjadi 2
3. i = 3, lakukan *query*(1, 1), yaitu mencari nilai arr maksimal yang berindeks 1 sampai 1. Maka akan didapat nilainya yaitu 1(arr[1]). sehingga nilai dari dp(3) = 2. update nilai arr[2] menjadi 2. ubah nilai parent[3] menjadi 2.
4. i = 4, lakukan query(1, 8), yaitu mencari nilai arr maksimal yang berindeks 1 sampai 8. Maka akan didapatkan nilainya yaitu 2 (arr[2]). Sehingga nilai dari dp(4) = 3. Update nilai arr[9] menjadi 3. Ubah parent[4] menjadi 3.
5. i = 5, lakukan query(1, 2), yaitu mencari nilai arr maksimal yang berindeks 1 sampai 2. Maka akan didapatkan nilainya yaitu 2 (arr[2]). Sehingga nilai dari dp(5) = 3. Update nilai arr[3] menjadi 3. Ubah parent[5] menjadi 3.
6. i = 6, lakukan query(1, 9), yaitu mencari nilai arr maksimal yang berindeks 1 sampai 9. Maka akan didapatkan nilai yaitu 3 (arr[9]). Sehingga nilai dari dp(6) = 4. Update nilai arr[10] = 4. Ubah parent[6] = 4.

Nilai maksimal dp(i) untuk i dari 1 sampai 6 adalah 4, maka LIS dari *sequence* tersebut memiliki panjang 4. Bilangan-bilangan yang diambil adalah [1, 2, 9, 10].

Berikut adalah hasil eksperimen yang telah dilakukan oleh penulis dengan 5 buah kasus uji. Kasus uji yang ditampilkan adalah sebagian hasil saja karena hasil yang sebenarnya terlalu banyak untuk ditampilkan.

1. Kasus uji 1

   Sequence = [30, 1, 4, 6, 5, 200]

```
Masukkan banyak bilangan pada sequence: 6
Masukkan sequence tersebut: 30 1 4 6 5 200
Panjang LIS: 4
LIS = [1,4,5,200]
Time taken : 0.0077s
```

2. Kasus uji 2

   Sequence = [20, 1, 3, 5, 6, 3, 7, 9, 100,200]

```
Masukkan banyak bilangan pada sequence: 10
Masukkan sequence tersebut: 20 1 3 5 6 3 7 9 100 200
Panjang LIS: 8
LIS = [1,3,5,6,7,9,100,200]
Time taken : 0.0094s
```

3. Kasus uji 3

   Sequence di random dengan menggunakan 1000 bilangan.

```
Panjang LIS: 61
LIS = [10,26,28,29,41,43,64,91,107,119,152,154,160,164,179,190,191,202,204,221,2
67,273,276,291,316,318,335,346,380,393,395,414,432,467,484,512,552,575,608,628,6
31,654,665,682,732,737,757,763,787,789,831,863,914,930,959,963,964,969,971,974,9
82]
Time taken : 0.0065s
```

4. Kasus Uji-4

   Sequence yang digunakan adalah bilangan-bilangan terurut dari 1 hingga 10000.

```
Panjang LIS: 10000
LIS = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,2
8,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,
55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81
,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,1
06,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,1
26,127,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145,1
46,147,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,1
66,167,168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,1
86,187,188,189,190,191,192,193,194,195,196,197,198,199,200,201,202,203,204,205,2
06,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,2
26,227,228,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,2
46,247,248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,264,265,2
66,267,268,269,270,271,272,273,274,275,276,277,278,279,280,281,282,283,284,285,2
86,287,288,289,290,291,292,293,294,295,296,297,298,299,300,301,302,303,304,305,3
06,307,308,309,310,311,312,313,314,315,316,317,318,319,320,321,322,323,324,325,3
26,327,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,344,345,3
46,347,348,349,350,351,352,353,354,355,356,357,358,359,360,361,362,363,364,365,3
66,367,368,369,370,371,372,373,374,375,376,377,378,379,380,381,382,383,384,385,3
```

```
9917,9918,9919,9920,9921,9922,9923,9924,9925,9926,9927,9928,9929,9930,9931,9932,
9933,9934,9935,9936,9937,9938,9939,9940,9941,9942,9943,9944,9945,9946,9947,9948,
9949,9950,9951,9952,9953,9954,9955,9956,9957,9958,9959,9960,9961,9962,9963,9964,
9965,9966,9967,9968,9969,9970,9971,9972,9973,9974,9975,9976,9977,9978,9979,9980,
9981,9982,9983,9984,9985,9986,9987,9988,9989,9990,9991,9992,9993,9994,9995,9996,
9997,9998,9999,10000]
Time taken : 0.0180s
```

5. Kasus uji-5

   Sequence yang digunakan adalah bilangan *random* dari 1 hingga 100000.

```
Panjang LIS: 619
LIS = [188,346,433,619,676,705,863,1054,1308,1524,1526,1914,2407,2527,2567,2956,
2967,2996,3576,3677,4052,4142,4207,4517,5131,5203,5217,5471,5752,5754,5842,5855,
5878,5926,6000,6276,6354,6397,6488,6535,6880,7001,7296,7477,7478,7543,7561,7592,
7653,7873,7903,7906,8228,8344,8356,8372,8420,8429,8544,8667,8837,8961,9020,9089,
9140,9448,9477,9598,9808,9861,9921,9941,10193,10408,10515,10787,10811,11003,1102
1,11092,11105,11122,11128,11285,11343,11363,11515,11596,11601,11827,11866,11966,
```

```
9917,9918,9919,9920,9921,9922,9923,9924,9925,9926,9927,9928,9929,9930,9931,9932,
9933,9934,9935,9936,9937,9938,9939,9940,9941,9942,9943,9944,9945,9946,9947,9948,
9949,9950,9951,9952,9953,9954,9955,9956,9957,9958,9959,9960,9961,9962,9963,9964,
9965,9966,9967,9968,9969,9970,9971,9972,9973,9974,9975,9976,9977,9978,9979,9980,
9981,9982,9983,9984,9985,9986,9987,9988,9989,9990,9991,9992,9993,9994,9995,9996,
9997,9998,9999,10000]
Time taken : 0.0180s
```

KESIMPULAN

Permasalahan *longest increasing subsequence* atau biasa disingkat LIS adalah salah satu contoh permasalahan klasik yang ada di *computer science*. Permasalahan yang harus diselesaikan adalah mencari *subsequence* terpanjang dari sekumpulan barisan bilangan dimana bilangan-bilangan yang dipilih harus selalu menaik. Perlu diketahui juga bahwa bilangan-bilangan yang dipilih tidak harus selalu berurutan. Permasalahan mencari *longest increasing subsequence* (LIS) dipelajari dalam banyak multidisiplin ilmu seperti *random matrix theory, representation theory,* dan fisika.

Solusi untuk penyelesaian permasalahan LIS untuk saat yang ini yang paling efisien adalah dengan menggunakan *dynamic programming*. Solusi yang dihasilkan akan memiliki kompleksitas $O(n^2)$.

Solusi dengan *dynamic programming* ini dapat dioptimasi lagi dengan bantuan struktur data *segment tree* yang merupakan struktur data yang mengaplikasikan paradigma *divide and conquer*. Dengan menggunakan *dynamic programming* dan struktur data *segment tree*, maka permasalahan *longest increasing subsequence* (LIS) dapat diselesaikan dengan kompleksitas waktu $O(n \log n)$. Sejauh ini, kompleksitas $O(n \log n)$ merupakan kompleksitas terbaik untuk menyelesaikan permasalahan ini.

PENUTUP

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena berkat-Nya penulis dapat menyelesaikan makalah ini dengan baik. Penulis juga tak lupa mengucapkan terima kasih kepada ibu, keluarga, serta teman-teman yang terus memberikan dukungan baik secara langsung maupun tidak langsung sehingga akhirnya makalah ini dapat selesai. Penulis juga ingin mengucapkan terima kasih kepada Bapak Rinaldi Munir selaku dosen mata kuliah strategi algoritma yang telah memberikan banyak ilmu baik di dalam perkuliahan maupun di luar perkuliahan. Terakhir, penulis memohon maaf apabila di dalam penulisan makalah ini terdapat kesalahan baik yang disengaja maupun tidak disengaja. Penulis berharap makalah ini mampu berguna bagi banyak orang.

REFERENCES

[1] Munir, Rinaldi. 2009. Diktat Kuliah Strategi Algoritma. Bandung: Program Studi Teknik Informatika Institut Teknologi Bandung

[2]] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. London: MIT Press.

[3] Gozali, Wiliam, dan Alham Fikri. 2018. *"Pemrograman Kompetitif Dasar"*. Jakarta : Ikatan Alumni Tim Olimpiade Komputer Indonesia.

[4] Halim, S. (2013). Competitive Programming 3: The New Lower Bound of Programming Contests. Singapore: School of Computing, National University of Singapore.

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, 25 April 2019

Irfan Sofyana Putra / 13517078