Penerapan Konsep DSU on Tree dan Struktur Data Segment Tree Pada Rancang Algoritma: Studi Kasus SPOJ Klasik LIS and Tree

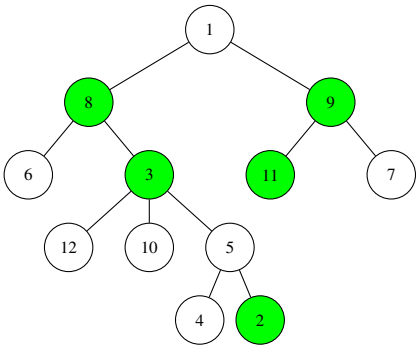
Dimas Hirda Pratama, Rully Soelaiman dan Abdul Munif
Departemen Informatika, Fakultas Teknologi Informasi dan Komunikasi, Institut Teknologi Sepuluh Nopember (ITS)
Jl. Arief Rahman Hakim, Surabaya 60111 Indonesia
e-mail: dimas14@mhs.if.its.ac.id, rully@is.its.ac.id, munif@if.its.ac.id

***Abstrak*—Permasalahan LIS and TREE merupakan sebuah permasalahan yang melibatkan sebuah struktur data tree. Dimana pada tree tersebut akan dicari LIS terpanjang dari seluruh simple path yang ada. Untuk menangani berbagai permasalahan pada permasalahan tersebut dibutuhkan struktur data yang mampu mendukung operasi-operasi tersebut dengan efisien.**
Pada Tugas Akhir ini akan dirancang penyelesaian permasalahan LIS and TREE antara lain operasi pencarian nilai LIS pada node dan subtree saat ini, operasi update nilai LIS pada node dan subtree saat ini dan menggabungkan serta memindahkan nilai pada dua subtree yang berbeda. Struktur data klasik yang biasa digunakan dalam penyelesaian permasalahan ini merupakan salah satu jenis stuktur data Tree yaitu Segment Tree dengan menggabungkan konsep Disjoint Set Union.
Pada Tugas Akhir ini digunakan struktur data Segment Tree dan konsep Disjoint Set Union untuk menyelesaikan operasi-operasi tersebut.

Kata Kunci—disjoint set union tree, segment tree, longest increasing subsequence

I. PENDAHULUAN

LIS and TREE[1] merupakan permasalahan pencarian *Longest Increasing Subsequence* pada sebuah *tree*, dimana seluruh *node* akan terhubung satu sama lainnya, dengan mengasumsikan seluruh *path* yang ada merupakan *simple path*.
Diberikan sejumlah kasus uji dimana pada setiap kasus uji yang ada akan diberikan data berupa bilangan bulat yang menyatakan jumlah *node* pada *tree* dan juga diberikan bilangan bulat a dan b untuk menggambarka bahwa *node* a dan b terhubung oleh *edge*.
Penelitian ini akan membahas bagaimana mengetahui hubungan antar *node* dengan menggunakan konsep *disjoint set union on tree* dan mencari nilai LIS pada *tree* tersebut menggunakan bantuan *segment tree*. Gambar 1 menunjukkan contoh bentuk *tree* dari masukan pada situs SPOJ. Tujuan dari penelitian ini adalah menyelesaikan permasalahan, menguji kebenaran dan performa dari algoritma



Gambar. 1. *Node* dengan warna hijau menunjukkan LIS dari contoh masukan pada situs penilaian daring SPOJ

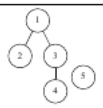
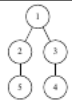
II. METODE PENYELESAIAN

A. Disjoint Set Union

Struktur data *disjoint set* adalah sebuah struktur data yang menyimpan sekumpulan himpunan atau *set*. Dua himpunan bisa disebut *disjoint* jika kedua himpunan tersebut tidak memiliki perpotongan, atau perpotongannya sama dengan *null*. Sebagai contoh himpunan $\{1, 2\}$ dan $\{3, 4\}$ merupakan himpunan yang *disjoint* karena tidak memiliki elemen yang sama diantara keduanya.
Disjoint set union sendiri merupakan suatu algoritma yang digunakan untuk menyatukan himpunan yang *disjoint* dengan himpunan lainnya. Hal ini dapat dilakukan dengan melakukan operasi $merge(a, b)$, dimana a dan b adalah dua himpunan yang saling *disjoint*.
Contoh:
Dimisalkan terdapat 5 buah himpunan bagian yang memiliki anggota berjumlah 1 yaitu: $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, dan $\{5\}$. Kemudian dilakukan operasi seperti pada Gambar 2 dan 3.

No	Operasi	Visualisasi Himpunan	Keterangan
1	Kondisi Awal		Kondisi awal himpunan.
2	$merge(2, 1)$		Kondisi setelah $merge(2, 1)$.

Gambar. 2. Visualisasi bentuk *tree* dari contoh masukan

3	$merge(4,3)$		Kondisi setelah $merge(4,3)$
4	$merge(3,1)$		Kondisi setelah $merge(3,1)$
5	$merge(5,2)$		Kondisi setelah $merge(5,2)$

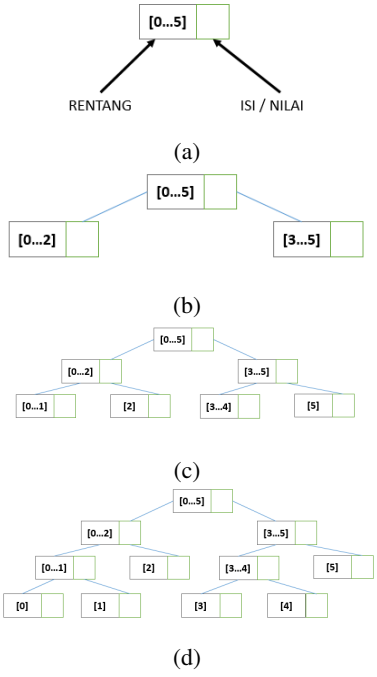
Gambar. 3. Visualisasi bentuk *tree* dari contoh masukan

B. Segment Tree

Segment Tree merupakan salah satu jenis struktur data yang merepresentasikan suatu nilai yang terhubung dengan suatu jeda atau interval tertentu di dalam *array*[3]. *Segment tree* juga merupakan *binary tree* dimana setiap *node* memiliki maksimal 2 *child*, biasa disebut *left* dan *right* child. Secara umum untuk mengakses nilai didalamnya bisa menggunakan *index* atau *pointer*. Dimisalkan terdapat sebuah struktur data *segment tree* yang menyimpan interval untuk suatu *array* $A[]$ dengan panjang n.Maka rumusan algoritma yang dapat digunakan untuk konstruksi *segment tree* sebagai berikut[4]:

1) Buat node yang berisikan seluruh elemen *array* $A[]$.
2) Jika elemen di dalam node beranggotakan lebih dari satu, maka buat 2 *child node* baru yang beranggotakan elemen $A[0, ..., n/2]$ dan $A[(n/2)+1, ..., n]$.
3) Jika elemen di dalam node hanya beranggotakan 1 elemen maka proses dihentikan.

Pada Gambar 4a menunjukkan *node* yang menyimpan nilai dari seluruh rentang yang ada menjadi *root* dari *segment tree*.



Gambar. 4. Proses pembentukan *Segment Tree*

Gambar 4b menunjukkan proses rekursi untuk membuat dua *child* baru, *child* kiri menyimpan nilai dari rentang 0 hingga 2, *child* kanan menyimpan nilai dari rentang 3 hingga 5. Gambar 4c menunjukkan lanjutan proses rekursi lagi, namun untuk *node* yang menyimpan hanya satu rentang saja, yaitu *node* yang menyimpan rentang 2 dan 5 proses dihentikan, bentuk akhir dari *segment tree* terlihat pada Gambar 4d.
Selain itu pada umumnya struktur data *segment tree* juga memiliki fungsi $query()$ untuk mengambil nilai dari suatu rentang dan juga $update()$ untuk merubah nilai pada suatu rentang.
Algoritma untuk melakukan fungsi $query()$ sebagai berikut:

1) Cek apakah nilai dari rentang yang diinginkan berada dalam rentang saat ini.
2) Jika iya ambil nilai yang berada di rentang tersebut.
3) Jika tidak lakukan rekursi ke *left child* dan *right child*.

Algoritma untuk melakukan fungsi $update()$ sebagai berikut:

1) Cek apakah nilai dari rentang yang diinginkan berada dalam rentang saat ini.
2) Jika iya ambil *update* nilai yang berada di rentang tersebut.
3) Jika tidak lakukan rekursi ke *left child* dan *right child*.

C. Heavy-Light Decomposition

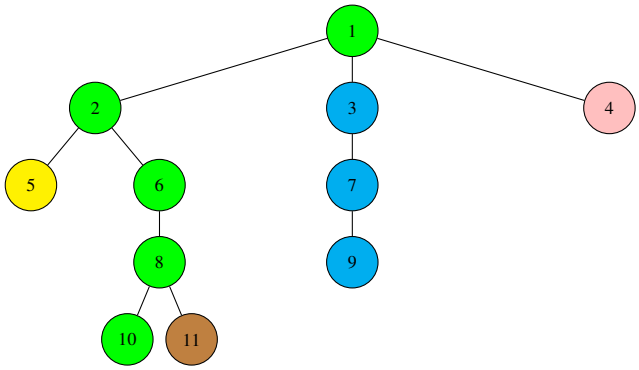
Heavy-Light Decomposition adalah teknik dekomposisi sebuah *tree* menjadi sebuah *disjoint chains* (tidak ada 2 *chain* yang memiliki *node* yang sama). Disebut *Heavy-Light* karena dekomposisi dilakukan berdasarkan kriteria yang telah ditentukan untuk membedakan apakah *chain* tersebut merupakan *node heavy* atau *light*.

Penggunaan konsep *Heavy-Light Decomposition*, yakni mengumpamakan sebuah *node* mempunyai satu *special child* yang merupakan sebuah *subtree* dengan ukuran paling besar di antara *child* lainnya. sehingga dapat diumpamakan *tree* yang telah dibuat terdiri dari beberapa *chain* dimana setiap *chain* mempunyai *head* yang bukan merupakan *special child* dari *parent child* tersebut.

Setelah membentuk *tree*, selanjutnya dicatat *edge* mana yang merupakan *heavy edge* dan *light edge*. Dengan aturan berikut:

1) *Heavy edge* adalah *edge* dengan jumlah *child* lebih besar atau sama dengan setengah jumlah *child* dari *parent child* tersebut.
2) *Light edge* adalah *edge* dengan jumlah *child* kurang dari setengah jumlah *child* dari *parent child* tersebut.

Pada Gambar 5 dapat dilihat terdapat banyak warna berbeda, setiap warna menggambarkan *chain* yang berbeda, dan *heavy chain* untuk *tree* tersebut ditandai dengan warna hijau.



Gambar. 5. Contoh penggambaran *Heavy-Light Decomposition*

D. Longest Increasing Subsequence

Dalam ilmu komputer, permasalahan *Longest Increasing Subsequence* atau yang biasa disingkat menjadi LIS, merupakan permasalahan yang bertujuan untuk menemukan *subsequence* terpanjang dari sebuah *sequence* dimana *subsequence* tersebut memiliki elemen yang sudah disortir dari nilai terkecil ke terbesar.
Untuk dapat menyelesaikannya algoritma yang dibutuhkan adalah sebagai berikut:
Dimisalkan terdapat sebuah *array* bernama $A[]$ yang memiliki panjang tertentu, dan penulis juga memiliki LIS terpanjang sementara atau disebut *active list* dengan panjang tertentu. Kemudian penulis ingin menambahkan elemen ke-*i* atau $A[i]$ maka harus memperhatikan beberapa hal sebagai berikut[2]:

1) jika $A[i]$ adalah elemen terkecil dari *active list* maka buat *active list* baru dengan panjang 1.
2) Jika $A[i]$ adalah elemen terbesar dari *active list* maka lakukan duplikasi terhadap *active list* terpanjang dan tambahkan elemen tersebut di akhir.
3) Jika $A[i]$ bukan elemen terbesar ataupun terkecil maka cari *active list* dengan elemen terakhir yang terbesar dan lebih kecil dari $A[i]$, lakukan duplikasi dan tambahkan elemen tersebut, dan hapus *list* lain dengan panjang yang sama.

Contoh
Untuk menemukan LIS dari array $A[0, 8, 4, 12, 2, 10, 14]$

1) $A[0]$ = 1, karena merupakan elemen pertama dan belum ada *active list* maka masuk ke *case* 1.

   $[0]$

2) $A[1]$ = 8, karena angka 8 lebih besar daripada 0, maka masuk ke *case* 2.

   $[0]$
   $[0, 8]$

3) $A[2]$ = 4, karena angka 4 berada di antara 0 dan 8, maka akan masuk ke *case* 3.

   $[0]$
   $[0, 4]$
   ~~$[0, 8]$~~ (dihilangkan)

4) $A[3]$ = 12, karena angka 12 lebih besar daripada 4, maka masuk ke *case* 1.

   $[0]$
   $[0, 4]$
   $[0, 4, 12]$

5) $A[4]$ = 2, karena angka 2 berada di antara 0 dan 12 maka akan masuk ke *case* 3.

   $[0]$
   $[0, 2]$
   ~~$[0, 4]$~~ (dihilangkan)
   $[0, 4, 12]$

6) $A[5]$ = 10, karena angka 10 berada di antara 0 dan 12 maka akan masuk ke *case* 3.

   $[0]$
   $[0, 2]$
   $[0, 2, 10]$
   ~~$[0, 4, 12]$~~ (dihilangkan)

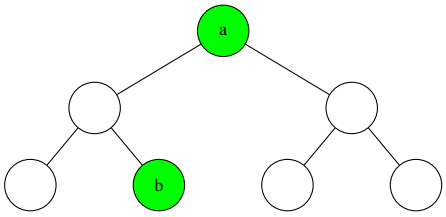
7) $A[6]$ = 14, karena angka 14 lebih besar daripada 10 maka akan masuk ke *case* 1.

   $[0]$
   $[0, 2]$
   $[0, 2, 10]$
   $[0, 2, 10, 14]$
   merupakan LIS dari array tersebut.

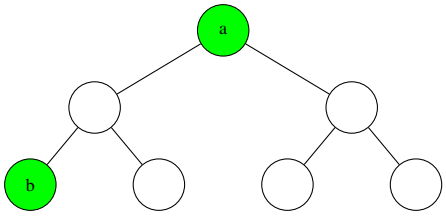
E. Strategi Penyelesaian

Dalam persoalan *LIS and TREE* LIS yang dicari merupakan sebuah *path* dari *tree* itu sendiri, dan *path* ini merupakan sebuah *simple path*. Maka untuk mendapatkan nilai LIS tersebut, secara umum terdapat beberapa *case* agar mendapatkan nilai LIS dari setiap *path* yang ada.
Case yang ada adalah sebagai berikut:

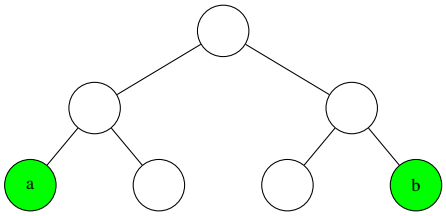
1) *Case* yang pertama adalah jika nilai *node* yang lebih besar berada di atas *node* tersebut, seperti pada Gambar 6. Menunjukkan bahwa nilai *node* $a > b$, sehingga arah *path* dari bawah menuju ke atas.
2) *Case* yang kedua adalah jika nilai *node* yang lebih besar berada di bawah *node* tersebut, seperti pada Gambar 7. Menunjukkan bahwa nilai *node* $a < b$, sehingga arah *path* dari atas menuju ke bawah.
3) *Case* yang kedua adalah jika nilai *node* yang lebih besar berada di *depth* yang sama dari *node* tersebut, seperti pada Gambar 8. Menunjukkan bahwa nilai *node* $a > b$, sehingga arah *path* dari bawah menuju ke atas kemudian menuju ke bawah.



Gambar. 6. Visualisasi *case* 1.



Gambar. 7. Visualisasi *case* 2.



Gambar. 8. Visualisasi *case* 3.

Secara umum algoritma penyelesaian yang dilakukan untuk setiap *node* adalah sebagai berikut:

1) Tentukan *child* yang menjadi *big child* atau *small child* menggunakan konsep *Heavy-light Decomposition*.
2) Lakukan rekursi ke *small child*
3) Lakukan perulangan untuk mencari LIS dan LDS dari *parent* pada seluruh *child* yang bukan merupakan *bigchild*.
4) Simpan hasil LIS dan LDS tersebut ke *Segment Tree*.
5) Cari nilai LIS dan LDS dari *child* ke *parent*.
6) Simpan hasil LIS dan LDS tersebut ke *Segment Tree*.
7) Setelah selesai, kembalikan *Segment Tree* ke kondisi semula.
8) Lakukan rekursi ke *big child*.
9) Ulangi proses ke 3 hingga 6.
10) Setelah selesai, biarkan kondisi *Segment Tree* karena akan digunakan kembali.
11) Lakukan proses hingga semua *node* terlewati.

III. UJI COBA DAN ANALISIS

A. Uji coba kebenaran

Uji coba kebenaran dilakukan dengan mengumpulkan berkas kode sumber hasil implementasi ke situs sistem penilaian daring SPOJ kali. Permasalahan yang diselesaikan adalah LIS and TREE dengan kode LISTREE. Hasil uji kebenaran dan waktu eksekusi program saat pengumpulan kasus uji pada situs SPOJ ditunjukkan pada Gambar 9 untuk teknik pengaksesan *segment tree* menggunakan *index* dan Gambar 10 untuk teknik pengaksesan *segment tree* menggunakan *pointer*.

ID	DATE	PROBLEM	RESULT	TIME	MEM	LANG
21792977	2018-06-06 22:15:59	LIS and tree	accepted edit ideone it	1.82	41M	CPP14

Gambar. 9. Hasil uji kebenaran teknik *index* dengan melakukan *submission* ke situs penilaian daring SPOJ

ID	DATE	PROBLEM	RESULT	TIME	MEM	LANG
21793114	2018-06-06 22:54:40	LIS and tree	accepted edit ideone it	0.68	42M	CPP14

Gambar. 10. Hasil uji kebenaran teknik *pointer* dengan melakukan *submission* ke situs penilaian daring SPOJ

Berikutnya adalah pengujian performa dari algoritma yang dirancang dan diimplementasi dengan melakukan uji *submission* dengan mengumpulkan berkas kode implementasi dari algoritma yang dibangun sebanyak 20 kali ke situs penilaian daring SPOJ dengan mencatat waktu eksekusi serta memori yang dibutuhkan.
Hasil dari pengujian untuk teknik pengaksesan *segment tree* menggunakan *index* dapat dilihat pada Gambar 11 dan Tabel 1.

ID	DATE	PROBLEM	RESULT	TIME	MEM	LANG
21792977	2018-06-06 22:15:59	LIS and tree	accepted edit ideone it	1.82	41M	CPP14
21792975	2018-06-06 22:15:45	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792967	2018-06-06 22:14:14	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792961	2018-06-06 22:11:47	LIS and tree	accepted edit ideone it	1.85	41M	CPP14
21792958	2018-06-06 22:11:18	LIS and tree	accepted edit ideone it	1.86	41M	CPP14
21792953	2018-06-06 22:10:43	LIS and tree	accepted edit ideone it	1.86	41M	CPP14
21792949	2018-06-06 22:10:13	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792948	2018-06-06 22:09:43	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792946	2018-06-06 22:09:09	LIS and tree	accepted edit ideone it	1.81	41M	CPP14
21792945	2018-06-06 22:08:35	LIS and tree	accepted edit ideone it	1.82	41M	CPP14
21792941	2018-06-06 22:08:02	LIS and tree	accepted edit ideone it	1.84	41M	CPP14
21792940	2018-06-06 22:07:33	LIS and tree	accepted edit ideone it	1.84	41M	CPP14
21792938	2018-06-06 22:06:48	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792936	2018-06-06 22:06:13	LIS and tree	accepted edit ideone it	1.85	41M	CPP14
21792932	2018-06-06 22:05:38	LIS and tree	accepted edit ideone it	1.81	41M	CPP14
21792928	2018-06-06 22:04:44	LIS and tree	accepted edit ideone it	1.89	41M	CPP14
21792923	2018-06-06 22:04:09	LIS and tree	accepted edit ideone it	1.87	41M	CPP14
21792919	2018-06-06 22:03:32	LIS and tree	accepted edit ideone it	1.82	41M	CPP14
21792917	2018-06-06 22:03:00	LIS and tree	accepted edit ideone it	1.84	41M	CPP14
21792915	2018-06-06 22:02:26	LIS and tree	accepted edit ideone it	1.87	41M	CPP14

Gambar. 11. Hasil uji coba teknik *index submission* ke situs penilaian daring SPOJ sebanyak 20 kali

Tabel 1. Kecepatan maksimal, minimal dan rata-rata dari hasil uji coba pengumpulan teknik *index* sebanyak 20 kali pada situs pengujian daring SPOJ

Waktu Maksimal	1.89 detik
Waktu Minimal	1.81 detik
Waktu Rata-Rata	1.85 detik
Memori Maksimal	41.0 MB
Memori Minimal	41.0 MB
Memori Rata-Rata	41.0 MB

Hasil dari pengujian untuk teknik pengaksesan *segment tree* menggunakan *pointer* dapat dilihat pada Gambar 12 dan Tabel 2.

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
21793114		2018-06-06 22:54:40	LIS and tree	accepted edit ideone it	0.68	42M	CPP14
21793113		2018-06-06 22:54:33	LIS and tree	accepted edit ideone it	0.69	41M	CPP14
21793112		2018-06-06 22:54:28	LIS and tree	accepted edit ideone it	0.67	41M	CPP14
21793109		2018-06-06 22:53:50	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793106		2018-06-06 22:53:45	LIS and tree	accepted edit ideone it	0.70	41M	CPP14
21793103		2018-06-06 22:53:05	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793101		2018-06-06 22:52:59	LIS and tree	accepted edit ideone it	0.66	41M	CPP14
21793095		2018-06-06 22:51:17	LIS and tree	accepted edit ideone it	0.69	41M	CPP14
21793093		2018-06-06 22:51:10	LIS and tree	accepted edit ideone it	0.70	42M	CPP14
21793092		2018-06-06 22:51:03	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793086		2018-06-06 22:50:04	LIS and tree	accepted edit ideone it	0.73	41M	CPP14
21793085		2018-06-06 22:49:55	LIS and tree	accepted edit ideone it	0.67	42M	CPP14
21793083		2018-06-06 22:49:48	LIS and tree	accepted edit ideone it	0.67	41M	CPP14
21793082		2018-06-06 22:49:41	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793081		2018-06-06 22:49:31	LIS and tree	accepted edit ideone it	0.68	41M	CPP14
21793079		2018-06-06 22:49:04	LIS and tree	accepted edit ideone it	0.67	42M	CPP14
21793078		2018-06-06 22:48:26	LIS and tree	accepted edit ideone it	0.70	42M	CPP14
21793077		2018-06-06 22:48:18	LIS and tree	accepted edit ideone it	0.66	42M	CPP14
21793076		2018-06-06 22:48:10	LIS and tree	accepted edit ideone it	0.70	42M	CPP14
21793000		2018-06-06 22:22:38	LIS and tree	accepted edit ideone it	0.68	41M	CPP14

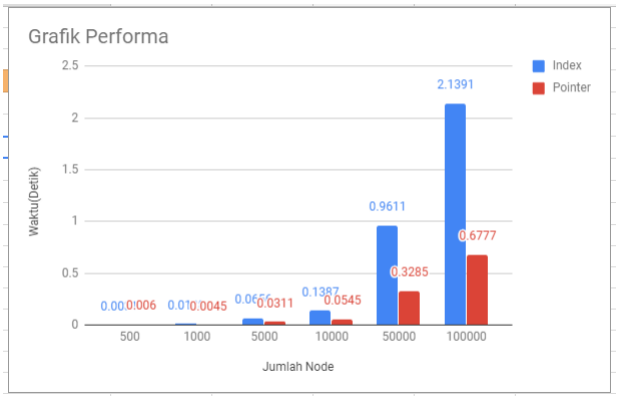
Gambar. 12. Hasil uji coba teknik *pointer submission* ke situs penilaian daring SPOJ sebanyak 20 kali

Tabel 2. Kecepatan maksimal, minimal dan rata-rata dari hasil uji coba pengumpulan teknik *pointer* sebanyak 20 kali pada situs pengujian daring SPOJ

Waktu Maksimal	0.73 detik
Waktu Minimal	0.66 detik
Waktu Rata-Rata	0.6795 detik
Memori Maksimal	41.0 MB
Memori Minimal	41.0 MB
Memori Rata-Rata	41.0 MB

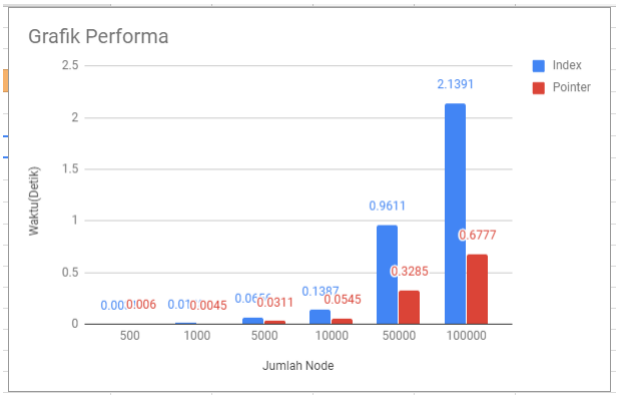
B. Uji Coba Kinerja

Pada uji coba yang ditunjukkan Gambar 13 terlihat performa program dengan input jumlah kasus uji satu, dan jumlah *node* yang bertambah hingga 100000.



Gambar. 13. Hasil uji coba program menggunakan data dari generator

Pada gambar grafik berwarna merah menunjukkan performa dari implementasi menggunakan pointer dan warna biru menunjukkan performa dari implementasi menggunakan index. Terlihat perbedaan yang cukup besar ketika masukan mencapai 100000 *node*, hal ini dikarenakan banyaknya bagian *segment tree* yang kosong yang harus dicek pada saat menggunakan implementasi index. Pada uji coba yang ditunjukkan Gambar 14 terlihat performa program dengan input jumlah kasus uji yang bertambah hingga 1000, dan jumlah *node* yang konstan yaitu 100000.



Gambar. 14. Hasil uji coba program menggunakan data dari generator

Pada gambar grafik berwarna merah menunjukkan performa dari implementasi menggunakan pointer dan warna biru menunjukkan performa dari implementasi menggunakan index. Waktu disini bertambah seiring dengan banyaknya kasus uji secara konstan.

C. Analisis Kompleksitas

Kedua teknik penyelesaian baik menggunakan *pointer* atau *index* memiliki kompleksitas yang sama yaitu $O(N \log^2 N)$. Kompleksitas ini didapatkan dari banyaknya proses memindahkan nilai dari *small child* ke *big child* yaitu sebesar $O(N \log N)$ dan proses $query()$ serta $update()$ yang dilakukan yaitu sebesar $O(\log N)$ dengan N adalah jumlah dari *node* pada *tree*.

IV. KESIMPULAN

Dari hasil uji coba yang dilakukan terhadap implementasi penyelesaian permasalahan *LIS and TREE* dapat diambil kesimpulan sebagai berikut:

1) Implementasi algoritma *Disjoint Set Union on Tree* dengan struktur data *Segment Tree* dapat menyelesaikan permasalahan *LIS and TREE* dengan benar.
2) Implementasi algoritma *DSU on TREE* pada struktur data *Segment Tree* dapat menghasilkan kecepatan proses yang berbeda dengan menggunakkan pendekatan yang berbeda yaitu (*by value & by reference*).
3) Kompleksitas waktu yang dibutuhkan untuk seluruh sistem adalah $O(n \log^2 n)$, dengan n merupakan banyaknya *node* yang ada pada *tree* tersebut.

UCAPAN TERIMA KASIH

Penulis mengucapkan puji syukur kehadirat Allah SWT atas segala rahmat dan karunia-Nya sehingga memungkinkan penulis untuk dapat menyelesaikan penelitian ini. Penulis juga mengucapkan terima kasih kepada orang tua dan keluarga penulis, juga kepada Bapak Rully Soelaiman dan Bapak Abdul Munif selaku dosen pembimbing penulis dan kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama penulis mengerjakan penelitian ini.

DAFTAR PUSTAKA

[1] Bartek, ”LIS and TREE” [Online]. Available: http://www.spoj.com/problems/LISTREE/ . [Accessed 17-May-2018].
[2] Geeksforgeeks, ”Longest Increasing Subsequence” [Online]. Avaiable: http://www.geeksforgeeks.org/longest-monotically-increasing-subsequence-size-n-log-n [Accesed 17-May-2018].
[3] **Understanding Segment Trees** - Understanding segment tree CodingHigway [Online]. Avaiable: http://codinghighway.com/2014/09/13/understanding-segment-trees/ [Accessed 17-May-2018].
[4] **Segment Tree — Set 1 (Sum of given range)** - Segment Tree — Set 1 (Sum of given range) - GeeksforGeeks [Online]. Avaiable: http://www.geeksforgeeks.org/segment-trees-set-1-sum-of-given-range/ [Accessed 17-May-2018].