Penerapan Algoritma Program Dinamis Dalam Fitur Koreksi Kata Otomatis Pada Aplikasi Pesan

Alif Raditya Rochman (13511013)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13511013@std.stei.itb.ac.id

***Abstract*—Makalah ini membahas mengenai penerapan algoritma program dinamis dalam pembangunan fitur koreksi kata otomatis pada aplikasi pesan. Dengan fitur koreksi kata otomatis, kata yang penulisannya salah yang diketik oleh pengguna aplikasi akan langsung diganti dengan kata yang dianggap benar. Untuk menentukan kata yang dianggap benar, digunakanlah sebuah nilai untuk menentukan jarak dari satu kata dengan kata lain. Nilai jarak tersebut didapatkan dengan menghitung jumlah biaya untuk menyisipkan, menghapus, atau mengganti sebuah karakter pada kata. Untuk mendapatkan nilai jarak tersebut dapat digunakan algoritma *brute force*. Akan tetapi algoritma tersebut memiliki performa yang kurang baik, sehingga digunakanlah algoritma program dinamis.**

Index Terms*—Koreksi Kata Otomatis, Aplikasi Pesan, Program Dinamis, *Levenshtein Distance

I. PENDAHULUAN

Manusia sebagai makhluk sosial selalu melakukan komunikasi dengan sesama manusia. Banyak cara yang dilakukan untuk saling berkomunikasi, diantanya yaitu melalui tatap muka, berkirim surat, dan lain sebagainya. Dengan perkembangan teknologi saat ini, manusia memiliki lebih banyak cara untuk melakukan komunikasi. Manusia dapat mengirim pesan elektronik berupa *email*, SMS, dan *instant messaging* melalui aplikasi pesan yang ada pada kakas elektronik yang dimilikinya.

Saat ini, banyak aplikasi pesan yang beredar di pasar. Banyaknya aplikasi tersebut merupakan hasil dari meningkatnya kebutuhan manusia untuk melakukan komunikasi yang mudah dan nyaman. Beberapa aplikasi pesan yang sering digunakan saat ini diantaranya adalah Skype, Line, Whatsapp, dan Kakao Talk.

Salah satu hal yang menjadi fokus dari pengembangan aplikasi pesan yaitu kenyamanan dari pengguna. Karena hal tersebut, banyak fitur yang dibangun untuk memberikan kenyamanan lebih kepada pengguna. Fitur koreksi kata otomatis merupakan salah satu fitur yang dibangun karena alasan tersebut.

Fitur koreksi kata otomatis merupakan fitur yang secara otomatis akan membenarkan kata yang terdapat kesalahan pengetikan ketika pengguna sedang mengetik. Sehinga akan menghemat waktu pengguna dengan tidak perlu membenarkan kata yang dia ketik secara manual.

Cara kerja dari fitur ini sebenarnya cukup sederhana. Saat pengguna selesai mengetikkan suatu *string* (ditandai dengan ditekannya spasi atau karakter khusus lainnya), aplikasi akan mengecek apakah *string* tersebut terdapat di dalam kamus kata. Bila tidak, maka kamus akan mencari *string* yang memiliki jarak paling kecil dengan *string* yang telah diketik. Kemudian *string* yang telah diketik akan diganti dengan *string* yang ditemukan.

Untuk membuat fitur ini, dibutuhkan kamus kata valid yang cukup lengkap sebagai data untuk membandingkan kata yang dimasukkan. Setiap bahasa dapat memiliki kamus kata sendiri. Agar kenyamanan pengguna terjama, diharapkan algoritma untuk fitur ini memiliki waktu eksekusi yang rendah.

II. TEORI DASAR

A. Program Dinamis

Program Dinamis (*Dynamic Programming*) merupakan sebuah metode untuk menyelesaikan masalah kompleks dengan menguraikan suatu masalah menjadi tahapan-tahapan masalah yang lebih sederhana[1].

Karakteristik dari persoalan yang dapat diselesaikan dengan program dinamis yaitu[2] :

1. Persoalan dapat dibagi menjadi beberapa tahapan dimana dibutuhkan sebuah keputusan disetiap tahap.
2. Setiap tahap memiliki sejumlah status yang berhubungan dengan tahap tersebut.
3. Keputusan di satu tahap akan mengubah status pada tahap berikutnya.
4. Keputusan optimal di suatu tahap tidak bergantung kepada keputusan yang telah dibuat pada tahap yang lalu.
5. Terdapat hubungan rekursif yang dapat menyelesaikan tahap bila tahapan sebelumnya telah dapat diselesaikan.
6. Tahap terakhir harus dapat diselesaikan.

Terdapat dua pendekatan yang dapat dilakukan denggan motede program dinamis : program dinamis maju (*top-down)* dan program dinamis mundur (*bottom-up)*[3]. Program dinamis maju merupakan sebuah pendekatan dimana solusi dari sebuah masalah akan

didapatkan dari solusi dari masalah yang lebih kecil secara rekursif. Bila terdapat masalah yang *overlap*, maka solusi dari masalah tersebut bisa disimpan untuk menghindari perhitungan berulang. Program dinamis mundur merupakan sebuah pendekatan dimana solusi dibangun tidak secara rekursif melainkan berawal dari solusi permasalahan yang paling kecil kemudian diiterasi untuk mendapatkan solusi permasalahan yang lebih besar sampai ke solusi permasalahan yang diinginkan[4].

Program dinamis sering digunakan untuk melakukan optimalisasi solusi permasalahan yang dapat diselesaikan dengan algoritma *brute force*. Banyak permasalahan klasik yang dapat diselesaikan oleh algoritma program dinamis. Beberapa permasalahan klasik yang dapat diselesaikan yaitu permasalahan *knapsack*, *travelling salesman problem*, *maxsum*, *longest increasing subsequence*, *longest common subsequence*, *edit distance*, dan *coin change*[5].

B. Levenshtein Distance

Levenshtein Distance merupakan cara untuk menilai bagaimana kemiripan dua buah *string* dengan menghitung jumlah operasi minimum yang diperlukan untuk mentransformasikan sebuah string ke string yang lain. Terdapat tiga jenis operasi dasar yang didefinisikan : penyisipan, penghapusan, dan penggantian[6].

Penyisipan yang dimaksudkan disini yaitu menyisipkan sebuah karakter dalam sebuah string. Misalkan terdapat string “trategy”. Bila kita melakukan penyisipan karakter “s” pada awal string tersebut, maka didapatkan string “strategy”. Dengan operasi ini, panjang string akan bertambah sebesar satu karakter.

Makna penghapusan yaitu menghapus tepat sebuah karakter yang ada pada indeks tertentu. Misalkan terdapat string “strategy”. Bila kita melakukan penghapusan pada karakter ketiga, maka didapatkan string “stategy”. Dengan operasi ini, panjang string akan berkurang sebesar satu karakter.

Makna penggantian yaitu mengganti tepat sebuah karakter yang ada pada indeks tertentu dengan karakter yang lain. Misalkan terdapat string “strategy”. Bila kita melakukan penggantian pada karakter ketiga dengan karakter “c”, maka didapatkan string “stcategy”. Dengan operasi ini, panjang string tidak berubah.

Misalkan string a yaitu “hello” dan string b yaitu “hbo”. *Levenshtein distance* dari string a dan b yaitu 3. Nilai tersebut didapatkan dari jumlah perubahan yang dilakukan terhadap string a untuk menjadi string b. Perubahan yang dilakukan yaitu penghapusan dua huruf “l” dan pengubahan huruf “e” menjadi hurub “b”.

III. ALGORITMA PENYELESAIAN

A. Brute Force

Dalam pembuatan fitur koreksi kata otomatis, yang perlu diperhatikan adalah menentukan jarak antara dua buah string, string S sebagai string masukan pada pengguna dan string K_i yaitu string yang ada pada kamus kata K. *Levenshtein distance* akan digunakan untuk menentukan jarak antara string S dengan string K_i.

Untuk menghitung *levenshtein distance* dari dua buah string dapat dilakukan secara rekursif. Formula dari algoritma rekursif untuk string a dan string b diberikan oleh $\mathrm{lev}_{a,b}(|a|,|b|)$, dimana :

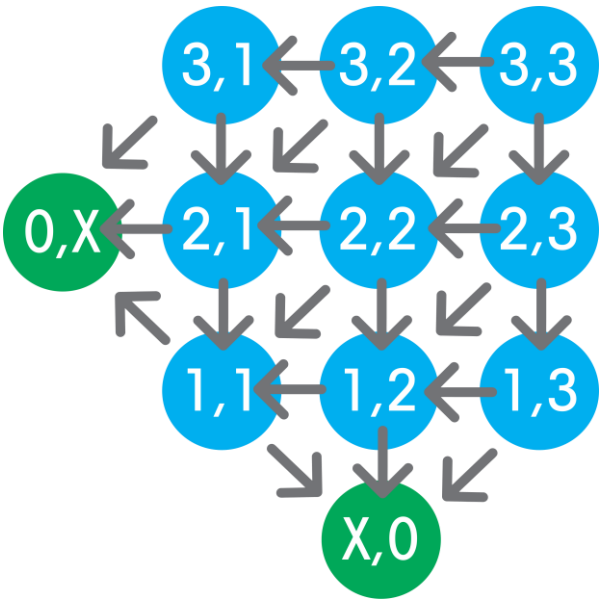
$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Gambar 3.1 Foruma Levenshtein Distance

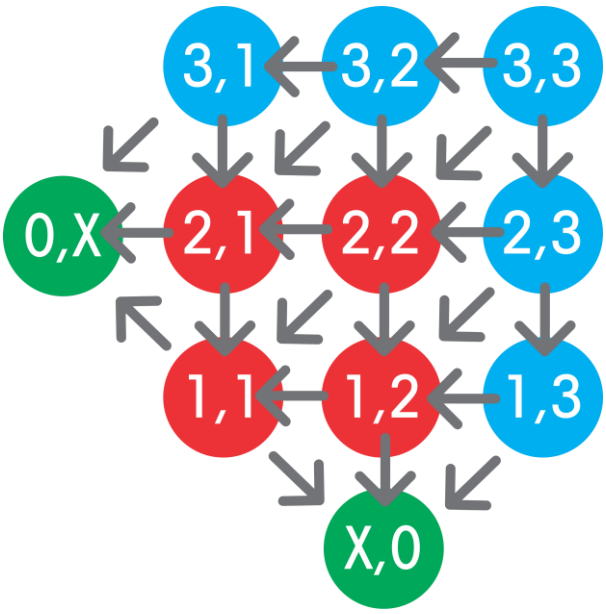
Misalkan i adalah indeks dari string a pada suatu tahap dan j ada adalah indeks dari string b. Formula yang diberikan diatas akan membandingkan string dari a dan string b dari indeks paling akhir. Untuk setiap tahap, akan diputuskan operasi mana yang lebih optimal. Untuk mengecek nilai solusi dari operasi penghapusan, maka status i akan dikurangi sejumlah satu. Untuk mengecek nilai solusi dari operasi penyisipan, maka status j akan dikurangi dengan sejumlah satu. Untuk operasi penggantian, maka status i dan status j masing-masing akan dikurangi dengan sejumlah satu. Bila karakter ke i dari string a dan karakter ke j dari string b sama, maka operasi penggantian tidak perlu diperhitungkan, tetapi status i dan status j dapat langsung dikurangi dengan sejumlah satu. Formula yang diberikan juga dapat diubah menjadi *pseudocode* berikut :

```
int lev(string a, int len_a, string b,int
                          len_b){
    if (len_a = 0) then
        → len b
    if (len_b = 0) then
        → len_a
    declare cost = 0
    if (a_len_a = b_len_b) then // MATCH checking
        cost ← 0
    else
        cost ← 1
    → minimum(
        lev (a, len_a-1, b, len_b) +1, // DELETE
        lev (a, len_a, b, len_b-1) +1, // INSERT
        lev (a, len_a-1, b, len_b-1) + cost); //
        SUBSTITUTE or CHECKING
}
```

Misalkan kita akan mengecek *levenshtein distance* dari string a “oil” dan string b “all”. Kita dapat membuat pohon solusinya sebagai berikut :



Gambar 3.2 Pohon solusi *brute force*



Gambar 3.3 Pohon solusi program dinamis maju

Pada gambar tersebut, warna biru merupakan permasalahan yang dapat dibuat tahapannya dan warna hijau adalah basisnya.

B. Program Dinamis Mundur

Kita dapat membangun solusi dengan metode program dinamis mundur setelah kita paham mengenai solusi *brute force* yang telah di bahas sebelumnya. Perhatikan bahwa persoalan yang diberikan sudah memenuhi karakteristik persoalan program dinamis. Persoalan ini dapat dibagi menjadi beberapa tahapan dan juga memiliki status yang berhubungan dengan tahap tertentu.

Bila kita perhatikan pohon solusi dari persoalan *brute force*, maka kita akan melihat bahwa terdapat persoalan dengan solusi yang overlap antara satu dengan lainnya. Dengan kata lain, terdapat solusi yang dimanfaatkan untuk menjawab lebih dari satu persoalan yang lebih besar. Dengan mengetahui hal itu, kita dapat membuat sebuah tabel untuk menyimpan solusi yang telah dikalkulasi sebelumnya untuk membangun solusi program dinamis mundur.

Untuk lebih jelasnya, kita akan melihat kembali contoh pada bagian sebelumnya. Misalkan kita akan mengecek *levenshtein distance* dari string a "oil" dan string b "all". Kita dapat membuat pohon solusinya sebagai berikut :

Bagian overlap dapat dilihat pada bagian yang berwarna merah. Dengan menggunakan program dinamis, kita dapat menghilangkan *overlap* tersebut dan membuat program menjadi lebih cepat. *Pseudocode* dari algoritma program dinamis mundur dapat dilihat sebagai berikut :

```
int lev(string a, int len a, string b,int
len b){
    if (len_a == 0) then
        → len b
    if (len_k == 0) then
        → len a
    if (dp[len_a,len_b] != -1) then// overlap solution
        return dp[len_a,len_b]
    declare cost = 0
    if (a[len_a] = b[len_b]) then// MATCH checking
        cost = 0;
    else
        cost = 1;
    dp[len_a,len_b] = minimum(
      lev (a, len_a-1, b, len_ki) +1, // DELETE
      lev (a, len_a, b, len_ki-1) +1, // INSERT
       lev (a, len_a-1, b, len_ki-1) + cost); //
      SUBSTITUTE or MATCH
    return dp[len_a,len_b];
}
```

C. Program Dinamis Maju

Untuk membangun solusi dengan metode program dinamis maju, kita akan meninjau kembali formula yang diberikan. Perhatikan bahwa kita dapat membangun tabel dengan cara terbalik. Sehingga kita akan menyelesaikan permasalahan kecil dan menyelesaikan permasalahan yang lebih besar dari solusi permasalahan yang lebih kecil tersebut. Lakukan penyelesaian secara iterative sampai mencapai permasalahan yang sesuai.

```
int lev(string a, string b)
{

  declare int d[m, n]
  // inisialisasi basis permasalahan
  for i from 1 to m
      d[i,0] ← i
```

```
  for j from 1 to n
      d0,j ← j
  // membangun solusi untuk permasalahan besar
  for j from 1 to n
      for i from 1 to m
          if ai = bj then
            di,j ← di-1,j-1 // no op req
          else
            di,j ← minimum(
              di-1,j + 1,  // a deletion
              di,j-1 + 1,  // an insertion
              di-1,j-1 + 1) // a substitution
  → dm,n
}
```

III. PENGUJIAN

A. Brute Force

Setelah membahas algoritma penyelesaian pada bab sebelumnya, kita akan melihat bagaimana performa dari algoritma brute force dalam menyelesaikan permasalahan pada fitur koreksi kata otomatis. Sebelum menguji pada fitur koreksi kata otomatis yang sebenarnya, kita akan melihat performa dari algortima brute force untuk menentukan *Levenshtein distance* (d) dari dua buah string : a dan b. Hasil pengujian yang didapat digambarkan pada tabel dibawah ini.

string a	string b	d	waktu eksekusi	jumlah iterasi
“helloworld”	“helloworld”	0	4250 ms	12146179
“helloworld”	“hllowwrld”	2	1721 ms	4976167
“he”	“helloworld”	8	1 ms	331
“strategy”	“stregy”	1	61 ms	162817
“algorithm”	“algo”	5	4 ms	8461
“algorithm”	“rithm”	4	13 ms	33544
“algorithm”	“helloworld”	8	1720 ms	4976167

Tabel 3.1 Hasil pengujian algoritma *brute force* untuk menentukan *levenshtein distance*

Untuk melakukan pengujian secara keseluruhan, kita akan menggunakan akan menggunakan kamus yang ada bersumber dari Interociter *bulletin board*. Jumlah kata yang disediakan yaitu sebanyak 110.000 kata. Hasil pengujian yang didapat digambarkan pada tabel dibawah ini.

string s	string k	levenshtein distance	waktu eksekusi
“strtegy”	Strategy	?	>12jam
“altrithm”	Algorithm	?	>12 jam
“math”	Math	?	>12 jam

Tabel 3.2 Hasil pengujian algoritma *brute force* untuk fitur koreksi kata otomatis

B. Program Dinamis Mundur

Dengan menggunakan data yang sama, kita akan melakukan pengujian terhadap algoritma program dinamis mundur untuk menyelesaikan permasalahan fitur koreksi kara otomatis. Sebelum menguji pada fitur koreksi kata otomatis yang sebenarnya, kita akan melihat performa dari algoritma program dinamis mudur untuk menentukan *levenshtein distance* (d) dari dua buah string : a dan b. Hasil pengujian yang didapat digambarkan pada tabel dibawah ini.

string a	string b	d	waktu eksekusi	jumlah iterasi
“helloworld”	“helloworld”	0	1 ms	301
“helloworld”	“hllowwrld”	2	1 ms	271
“he”	“helloworld”	8	0 ms	61
“strategy”	“stregy”	1	1 ms	145
“algorithm”	“algo”	5	0 ms	109
“algorithm”	“rithm”	4	1 ms	136
“algorithm”	“helloworld”	8	0 ms	271

Tabel 3.3 Hasil pengujian algoritma program dinamis mundur untuk menentukan *levenshtein distance*

Untuk melakukan pengujian secara keseluruhan, kita akan menggunakan kamus yang ada bersumber dari Interociter *bulletin board*. Jumlah kata yang disediakan yaitu sebanyak 110.000 kata. Hasil pengujian yang didapat digambarkan pada tabel dibawah ini.

string s	string k	levenshtein distance	waktu eksekusi
“strtegy”	Strategy	1	7301 ms
“altrithm”	Algorithm	2	8147 ms
“math”	Math	0	4316 ms

Tabel 3.4 Hasil pengujian algoritma program dinamis mundur untuk fitur koreksi kata otomatis

C. Program Dinamis Maju

Dengan menggunakan data yang sama, kita akan melakukan pengujian terhadap algoritma program dinamis maju untuk menyelesaikan permasalahan fitur koreksi kata otomatis. Sebelum menguji pada fitur koreksi kata otomatis yang sebenarnya, kita akan melihat performa dari algoritma program dinamis maju untuk menentukan *levenshtein distance* (d) dari dua buah string : a dan b. Hasil pengujian yang didapat digambarkan pada tabel dibawah ini.

string a	string b	d	waktu eksekusi	jumlah iterasi
“helloworld”	“helloworld”	0	1 ms	100
“helloworld”	“hllowwrld”	2	0 ms	90
“he”	“helloworld”	8	0 ms	20
“strategy”	“stregy”	1	0 ms	48
“algorithm”	“algo”	5	0 ms	36
“algorithm”	“rithm”	4	0 ms	45
“algorithm”	“helloworld”	8	1 ms	90

Tabel 3.5 Hasil pengujian algoritma program dinamis maju untuk menentukan *levenshtein distance*

Untuk melakukan pengujian secara keseluruhan, kita akan menggunakan kamus yang ada bersumber dari Interociter *bulletin board*. Jumlah kata yang disediakan yaitu sebanyak 110.000 kata. Hasil pengujian yang didapat digambarkan pada tabel dibawah ini.

string s	string k	levenshtein distance	waktu eksekusi
“strtegy”	Strategy	1	583 ms
“altrithm”	Algorithm	2	577 ms
“math”	Math	0	469 ms

Tabel 3.6 Hasil pengujian algoritma program dinamis maju untuk menentukan *levenshtein distance*

V. Analisis

Berdasarkan hasil pengujian, dapat dilihat bahwa algoritma *brute force* memiliki performa yang paling lambat dibandingkan dengan yang lain. Untuk membandingkan dua buah string yane mmiliki panjang 10 karakter, dibutuhkan waktu 4250 ms. Hal ini jelas tidak feasible bila diimplementasikan untuk membuat fitur koreksi kata otomatis. Untuk pengecekan satu string yang dimasukkan oleh pengguna, aplikasi perlu untuk mengecek *levenshtein distance* antara string yang dimasukkan dengan setiap kata pada kamus kata. Bila terdapat 100.000 kata pada kamus, maka diperkirakan dibutuhkan 111 jam untuk memproses satu string yang dimasukkan. Oleh karena itu, hasil pengujian keseluruhan dengan algoritma *brute force* sulit untuk didapat. Bila dianalisa lebih lanjut, hal ini dikarenakan kompleksitas dari penggunaan algoritma ini yaitu $O(N^3)$, dimana N adalah panjang string masukan.

Performa algoritma dinamis mundur jauh lebih baik daripada algoritma *brute force* dikarenakan elminasi tahap yang overlap. Algoritma ini memiliki kompleksitas waktu yang linear yaitu O(N*M) dimana N adalah panjang string masukan dan M adalah panjang string sebuah kata pada kamus. Untuk membandingkan dua buah string yane memiliki panjang 10 karakter, hanya dibutuhkan waktu 1 ms.

Performa algoritma dinamis maju lebih baik daripada performa algoritma dinamis mundur. Walaupun waktu untuk menghitung *levenshtein distance* tidak terlalu berbeda jauh, tetapi ketika kita membandingkan dengan setiap kata pada kamus, maka perbedaan waktunya menjadi signifikan. Perhatikan bahwa waktu untuk memproses sebuah kata masukan dengan menggunakan algoritma program dinamis mundur dapat mencapai 8 detik, sedangkan dengan menggunakan program dinamis maju tidak mencapai 1 detik.

VI. Kesimpulan

Dari hasil pengujian dan analisis yang diberikan, kita dapat menyimpulkan bahwa :

1. Penggunaan algoritma program dinamis dapat mempercepat pencarian string pada kamus dibandingkan dengan algoritma brute force.
2. Metode algoritma program dinamis maju memiliki performa yang lebih baik daripada algoritma program dinamis mundur
3. Untuk membangun fitur koreksi kata otomatis, lebih baik menggunakan algoritma program dinamis maju untuk menghitung jarak dua buah string.

VII. Tambahan

Untuk meningkatkan ketepatan dari fitur koreksi kata otomatis, kita dapat memanfaatkan peletakan key terhadap keyboard untuk menghitung jarak antara dua string. Selain itu dapat juga diberinkan nilai rank yang menunjukkan jumlah kata tersebut digunakan. Nilai rank tersebut akan berguna untuk memilih dua string yang memiliki *levenshtein distance* yang sama dengan string masukan.

VIII. Ucapan Terima Kasih

Selama pembuatan makalah ini, penulis ingin mengucapkan terima kasih kepada pihak-pihak berikut ini:

1. Pak Rinaldi Munir dan Ibu Masayu Leylia Khodra sebagai pengajar matakuliah IF2211 yang sudah mengajarkan metode-metode penyelesaian masalah.
2. Teman-teman Teknik Informatika 2011 yang telah mendukung penulis untuk terus berkarya.
3. Dewan Eksekutif HMIF 2013/2014 yang telah memberikan sarana dan prasarana yang dapat penulis manfaatkan untuk membuat tugas-tugas kuliah.

Referensi

[1] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. Introduction to Algorithm. MIT Press : Hill, 2001, pp 320.

[2] Michael A. Trick, “A Tutorial on Dynamic Programming”. 14 Juni 1997. http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html.

[3] S. Halim and F. Halim , *Competitive Programming 2 : This increase the lower bound of programming contest*. Singapore: NTU, 2011, pp. .56-58

[4] S. Halim and F. Halim , *Competitive Programming 2 : This increase the lower bound of programming contest*. Singapore: NTU, 2011, pp. .55

[5] S. Halim and F. Halim , *Competitive Programming 2 : This increase the lower bound of programming contest*. Singapore: NTU, 2011, pp. .56

[6] S. Halim and F. Halim , *Competitive Programming 2 : This increase the lower bound of programming contest*. Singapore: NTU, 2011, pp. .160

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, 20 Desember 2013

Alif Raditya Rochman (13511013)