

Fast Pattern Matching Algorithm on Two-dimensional String

Luqman A. Siswanto / 13513024
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
luqmanarifin@s.itb.ac.id

Abstrak — Makalah ini terfokus pada pembahasan *pattern recognizing* pada string 2 dimensi. Untuk teks berukuran $n \times n$ dan *pattern* berukuran $m \times m$, algoritma yang akan digunakan pada makalah ini dapat berjalan pada kompleksitas $O(n^2)$. Padahal menggunakan pendekatan secara naif (*brute force*), *pattern matching* dilakukan dalam $O(n^2m^2)$. Teknik yang digunakan untuk mereduksi kompleksitas pencocokan string 2-dimensi ini adalah menggunakan kombinasi antara *presum dynamic programming* dan *hashing*. Tentu saja teknik ini tidak sepenuhnya berjalan dengan benar karena penggunaan *hashing* menyebabkan resiko *collision*. Akan tetapi, hash cukup baik untuk dijalankan pada string standard. Sementara ada teknik lain untuk menghindari *collision* dengan cara hash berlapis.

Kata Kunci — dua-dimensi, *dynamic programming*, *hashing*, *pattern matching*, *pattern recognizing*, *string*.

I. PENDAHULUAN

Algoritma *pattern matching* dapat ditemukan hampir di setiap bidang dalam informatika / ilmu komputer. *Pattern matching* merupakan hal yang sangat pokok dalam berbagai masalah yang menyangkut bidang informatika seperti perbandingan dan pencarian yang menyebar secara sangat luas di seluruh cabang pokok ilmu komputer seperti *discrete mathematics*, *bio-informatics*, *security engineering*, *computer engineering*.

Pencocokan *pattern* pada teks dengan panjang n dan *pattern* dengan panjang m merupakan studi literatur informatika yang telah klasik. Banyak ilmuwan telah lama meneliti hal ini. Banyak algoritma yang sudah umum dikenal dalam penyelesaian masalah ini. Sebut saja *Knuth-Morris-Pratt (KMP) algorithm*, *Boyer-Moore (BM) algoritim*, *Rabin-Karp algorithm*, dan banyak algoritma lainnya. Pendekatan lain banyak dilakukan, baik secara naif maupun tidak. Pendekatan yang trivial contohnya adalah *brute force*.

Banyak tulisan dan makalah yang membahas mengenai string-matching pada tahun lalu dan tahun-tahun sebelumnya beserta penerapannya dalam

berbagai bidang. Ada yang membahas hubungannya dalam pencocokan DNA sequence, *regex* untuk validasi formulir, mendeteksi kanker, SMS *blocker* untuk validasi pesan, sidik jari, cap bibir, *face recognition*, deteksi *down syndrom*, dan perbandingan struktur kimia.

Tulisan ini berfokus pada pengembangan algoritma untuk pencarian *pattern* dengan algoritma berkompleksitas rendah. Penerapan *string matching* 2D tidak akan banyak dibahas secara detail. Dalam makalah ini akan dibahas secara tuntas dan jelas mengenai teknik reduksi dan ide brilian tentang mereduksi pengecekan untuk validasi yang tidak perlu, membuang operasi-operasi yang tidak dibutuhkan. Teknik ini dapat dikembangkan selanjutnya dengan ide yang sama untuk *pattern recognizing* pada dimensi berikutnya, tiga-dimensi, empat-dimensi, dan seterusnya.

Pattern matching 2D dapat diaplikasikan secara luas untuk pencocokan citra (*image*). Citra dapat dimodelkan sebagai string 2 dimensi, begitu juga dengan citra *pattern* yang akan dicocokkan. Oleh karena itu, permasalahan pencocokan citra dapat dikonversi menjadi permasalahan pencocokan string 2D.

Dengan teknik yang mirip, bahkan dimungkinkan pula dilakukan *pattern matching* untuk citra yang bergerak seperti *video* atau film. Tentunya ini tidak akan berhasil apabila algoritma yang digunakan tidak mangkus dan sangkil. Apabila algoritma yang digunakan tidak cukup cepat berjalan pada kompleksitas yang ramping, maka pencocokan citra bergerak tidak dimungkinkan berjalan dengan cepat.

II. DASAR TEORI

Pencocokan *string* dalam makalah kali ini menggunakan gabungan teknik *presum dynamic programming* dan *hashing*. Perlu diketahui bahwa *dynamic programming* (DP) tidak hanya berguna

dalam penyelesaian masalah besar yang kompleks, juga *hashing* juga tidak hanya berguna dalam persoalan *security*. Ada banyak solusi untuk satu permasalahan dan bisa jadi ada banyak permasalahan yang bisa diselesaikan dengan satu solusi. Di bawah ini dipaparkan dasar teori yang akan digunakan sebagai dasar penulis untuk menulis makalah ini.

A. String Matching Algorithm

Banyak algoritma yang dapat digunakan dalam string matching secara naif maupun secara tidak naif. Yang saat ini populer adalah *Knuth-Morris-Pratt* dan *Boyer-Moore algorithm*.

Masalah pencocokan string 1-dimensi adalah *classic problem* dalam dunia *computer science*. Didefinisikan teks dengan panjang n dan *pattern* sepanjang m karakter, keduanya *0-indexed*. Cari $0 \leq i < n$ sehingga untuk setiap $0 \leq j < m$, kondisi `teks[i + j] = pattern[j]`.

1) Brute-force

Dengan pendekatan *brute force*, dilakukan pengecekan untuk setiap indeks teks, apakah indeks tersebut merupakan awal dari *pattern* yang akan di-*match*. Algoritma ini berjalan dalam kompleksitas $O(nm)$. Berikut adalah *pseudo-code* dari pendekatan *brute-force*.

```
for each (i in text)
  bool match = false;
  for each (j in pattern)
    if(text[i + j] != pattern[j])
      match = false;
  if(match)
    print "matched at index ", i;
```

Pattern : GULA
Teks : GILAGULU GULAKU GALIGALIGU

```
GILAGAGGULA GULAKU GALIGALIGU
1 GULA
2  GULA
3   GULA
4    GULA
5     GULA
6      GULA
7       GULA
8        GULA
9         GULA
```

Brute-force tidak cukup baik dijalankan untuk ukuran n dan m yang besar. Dengan seiring berjalannya n , algoritma meningkat dengan pesat berbanding lurus dengan m karena kompleksitasnya $O(nm)$. Walaupun demikian, untuk string yang kecil ($n, m < 100$) belum

ada perbedaan yang signifikan.

2) Knuth-Morris-Pratt

Algoritma KMP adalah algoritma yang berfungsi mencari *pattern* di dalam teks secara terurut dari kiri ke kanan seperti *brute-force* namun dengan teknik yang lebih pintar. Algoritma KMP melakukan perbaikan dari pendekatan *brute-force* dengan cara mengurangi pengecekan-pengecekan yang tidak perlu. Ide utama dari algoritma KMP adalah ketika indeks i dalam teks diketahui tidak akan *match* dengan *pattern*, indeks j sebagai *pointer* dari *pattern* tidak mengulang dari 1 lagi. Dengan ide tersebut, algoritma KMP dapat berjalan di kompleksitas $O(n + m)$.

Dalam keberjalanannya, KMP membutuhkan *border function*. *Border function* ini berguna untuk memutuskan seberapa jauh *pointer j* dari *pattern* harus mundur/kembali ketika diketahui teks tidak sama dengan *pattern*. Secara *worst case*, *pointer j* akan kembali mundur ke indeks 1. Secara *best case*, *pointer j* akan mundur sebanyak satu indeks saja.

Border function KMP dapat dilakukan sebelum algoritma KMP dijalankan (*pre-processing*). *Border function* membutuhkan *array of integer* seukuran dengan ukuran *pattern* (sebesar m). Algoritma untuk melakukan *pre-process border function* berjalan dalam kompleksitas $O(m)$. *Pre-process* KMP menghasilkan *array of integer* yaitu *border function* ini.

Maksud dari *border function* dari string s pada indeks ke- i adalah berapa panjang maksimum prefiks s yang cocok dengan sufiks string dengan indeks maksimum i .

Berikut adalah contoh tabel *border function* untuk string `abaabaa`

```
INDEX   : 1 2 3 4 5 6 7
STRING  : A B A A B A A
BORDER  : 0 0 1 1 2 3 1
```

Border function pada indeks ke-6 adalah 3 karena prefiks `ABA` adalah substring terpanjang dengan panjang 3 yang cocok dengan string bersufiks $i = 6$.

Berikut adalah contoh implementasi *border function*.

```
for(int i = 1; i < n; i++) {
  int j = a[i - 1];
  while(j > 0 && s[i] != s[j]) {
    j = a[j - 1];
  }
  if(s[i] == s[j]) a[i] = j + 1;
}
return a;
```

Setelah melakukan *pre-process*, algoritma KMP baru dijalankan. Algoritma ini berjalan dalam kompleksitas $O(n)$ memanfaatkan ide yang sudah ditulis di atas dengan bantuan *border-function* yang sudah di-*preprocess*.

Berikut adalah contoh implementasi algoritma KMP.

```
int[] b = preProcessKmp(pattern);
int j = 0;
for(int i = 0; i < text.length();) {
    if(pattern[j] == text[i]) {
        i++; j++;
    } else if(j > 0) {
        j = b[j - 1];
    } else {
        i++;
    }
    if(j == pattern.length()) {
        print "Pattern found at", i;
    }
}
print "Pattern not found";
```

Bagaimanapun juga, algoritma KMP hanya berjalan dalam string 1 dimensi. Algoritma KMP bagus dijalankan dalam *pattern* yang memiliki kemiripan struktur dengan teksnya. Ini terjadi karena ketika *pattern* mirip dengan teks, maka pointer *pattern* tidak perlu mengulang dari indeks 1 kembali.

3) Boyer-Moore Algorithm

Algoritma *Boyer Moore* adalah algoritma *pattern matching* yang dilakukan secara mundur pengecekannya yakni dari belakang ke depan. Algoritma ini adalah algoritma yang paling populer untuk pencarian *plain text* dalam teks standar karena kompleksitasnya yang ramping. Sementara, *worst case*-nya algoritma ini berjalan dalam kompleksitas $O(nm)$.

Algoritma ini membutuhkan *pre-process* seperti KMP tetapi yang diproses adalah kemunculan terakhir dari setiap karakter yang muncul. Ide utama dari algoritma ini adalah, untuk karakter yang tidak muncul, maka pencocokan string tidak perlu dilakukan.

Alur bekerja algoritma ini dibagi menjadi 3 kasus :

- Teks[i] dan pattern[j] match, pengecekan mundur ke $i - 1$ dan $j - 1$.
- Teks[i] dan pattern[j] tidak match, i maju sejauh jarak karakter teks[i] muncul terakhir kali pada pattern.
- Teks[i] dan pattern[j] tidak match dan teks[i] tidak muncul pada pattern, i maju sejauh nilai j saat ini.

Berikut adalah alur cara kerja algoritma *Boyer Moore*.

```
TEKS      : ALI SUKA KELABU
PATTERN   : ELA
            ELA
            ELA
            ELA
            ELA
```

Boyer Moore berjalan pada kasus terburuk ketika :

```
Teks      = AAA..AAA
Pattern   = BAA..A
```

Dalam kasus ini *Boyer-Moore* berjalan dalam kompleksitas $O(nm)$. *Boyer Moore* tidak terlalu baik dalam pencocokan string yang teks dan *pattern*-nya memiliki struktur yang mirip. Namun algoritma ini dapat berjalan hingga $O(n / m)$ dalam *best case*. Algoritma *Boyer Moore* paling umum digunakan dalam *string matching* pada teks yang dapat dibaca manusia.

B. Dynamic Programming

Dynamic programming adalah teknik untuk menyelesaikan permasalahan yang besar dan kompleks dengan cara mereduksinya menjadi subpersoalan yang identik. Ciri khusus dari teknik ini adalah menjaga solusi untuk indeks i tetap optimal ketika dikembangkan ke indeks $i + 1$. Pembuktian bahwa *dynamic programming* adalah solusi yang benar untuk sebuah permasalahan mirip dengan pembuktian dengan induksi matematika.

Contoh persoalan *dynamic programming* adalah sebagai berikut :

Anda memiliki 10 rupiah. Anda dapat belanja sebesar 1, 2, 3 rupiah dan uang tersebut seluruhnya harus habis. Berapa banyak kemungkinan urutan belanja yang terjadi?

Masalah ini dapat direpresentasikan dalam satu *array of integer* di mana indeks menyatakan rupiah. Dan isi *array*-nya merupakan banyak kemungkinan untuk nilai rupiah tersebut.

Basis permasalahan ini adalah :

$$f[0] = 1$$

Secara intuitif, rekurens untuk permasalahan ini adalah :

$$f[i] = f[i - 1] + f[i - 2] + f[i - 3]$$

Akibatnya :

- f[1] = 1
- f[2] = 2
- f[3] = 4
- f[4] = 7
- f[5] = 13
- f[6] = 24
- f[7] = 44
- f[8] = 81
- f[9] = 149
- f[10] = 274

Dari perhitungan di atas, didapat bahwa banyak cara untuk membelanjakan uang 10 rupiah ada 274 cara berbeda.

C. Presum Dynamic Programming

Konsep dari *presum* adalah bagaimana melakukan *caching* dari beberapa data supaya ketika akses data diminta, query dapat dieksekusi dengan cepat.

Presum dapat dilakukan dengan cara menjumlahkan/mengoperasikan nilai secara satu arah sehingga ketika diminta untuk mengeluarkan hasil operasi untuk *range* tertentu, hasil dapat diambil dengan kompleksitas $O(1)$. Contoh dari permasalahan *presum dynamic programming* adalah sebagai berikut.

Anda diberikan sejumlah angka dengan indeks sebagai berikut :

INDEX : 1 2 3 4 5
 VALUE : 4 1 2 3 4

Secara cepat, diberikan angka i, j . Keluarkan jumlah angka dari indeks dengan *range* $i - j$.

Persoalan ini dapat diselesaikan menggunakan *presum*. Pertama kali bangun *pre-processing* untuk *array* sehingga tiap indeks dari *array* tersebut menyatakan jumlah nilai dari 1 s.d. indeks tersebut. Berikut adalah hasil *presum*.

INDEX : 1 2 3 4 5
 PRESUM : 4 5 7 10 14

Untuk menjawab *query* (i, j) , cukup mengembalikan $presum[j] - presum[i - 1]$. Setiap *query* dapat dijawab dengan cepat dalam kompleksitas $O(1)$.

Dalam *presum* dua dimensi, ide yang sama dapat diimplementasikan dengan cara yang sedikit berbeda. Tujuan yang akan kita capai adalah untuk setiap (i, j) pastikan isi *array* pada indeks tersebut adalah jumlah nilai dari indeks $(1, 1)$ hingga (i, j) .

Cara pembangunan *presum* dua dimensi sebagai berikut dengan induksi matematik.

Basis adalah $f(1, 1) = f(1, 1)$. Akan dicari nilai *presum* untuk indeks (i, j) . Misalkan untuk indeks $(i - 1, j)$, $(i, j - 1)$, $(i - 1, j - 1)$ masing-masing indeks menyimpan *presum* dari $(1, 1)$ menuju indeks masing-masing. Maka secara intuitif persamaan untuk $f(i, j)$ adalah sebagai berikut.

$$f(i, j) = f(i - 1, j) + f(i, j - 1)$$

Akan tetapi nilai $f(i - 1, j - 1)$ dalam persamaan tersebut terhitung dua kali akibat $f(i - 1, j)$ mengandung $f(i - 1, j - 1)$, begitu juga $f(i, j - 1)$ juga mengandung $f(i - 1, j - 1)$.

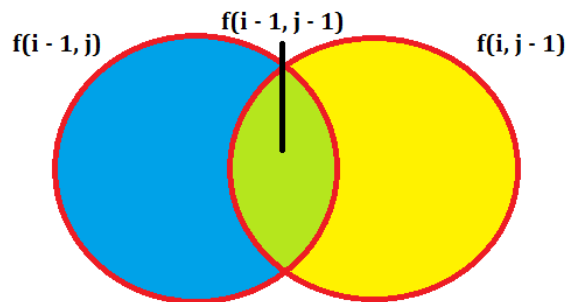


Diagram Venn rekurens *presum* DP 2D

Sehingga persamaan yang benar untuk *presum* pada indeks (i, j) adalah sebagai berikut

$$f(i, j) = f(i - 1, j) + f(i, j - 1) - f(i - 1, j - 1)$$

Untuk mengambil *range query* pada indeks (i, j) sampai (p, q) , persamaan di bawah ini berlaku

$$f(i, j, p, q) = f(p, q) - f(p, j - 1) - f(i - 1, q) + f(i - 1, j - 1)$$

Penurunan persamaan di atas didapat dengan metode yang sama dengan penurunan persamaan rekurens *presum* DP di atas.

D. Hashing

Ide utama dari hashing adalah mengganti representasi sebuah objek menjadi sebuah benda lain yang lebih sederhana sehingga lebih mudah diproses / dioperasikan. Secara umum, objek yang akan di-hash biasanya direpresentasikan menjadi sebuah nilai integer. Idealnya, satu *hashed-value* (yang dalam hal ini *integer*) adalah wujud yang *unique* dari objek yang telah di-hash. Akibatnya, dua *key* yang berbeda menandakan dua objek yang berbeda.

Implementasi *hash* biasanya melibatkan bilangan prima sebagai *key generator* dan bilangan prima untuk modulo *hashed-value*. Implementasi ini juga berkaitan erat dengan operasi aritmatik modular.

Contoh penggunaan *hash* adalah sebagai berikut. Misalkan *key generator* adalah 5, dan modulo *hashed-value* nya adalah 31.

```
Text :      A      B      A
Hash :    1*52  2*51  1*50
Hash value = (1*52 + 2*51 + 1*50) mod 31
           = 36 mod 31
           = 5
```

III. IMPLEMENTASI DAN EKSPERIMEN

Untuk menyelesaikan masalah *string matching* ini, perlu dilakukan *pre-processing* 2D untuk *hash* teks dan *hash pattern*. Perlu dibangkitkan *presum* setiap indeks *hash* teks karena pengaksesan *range query* membutuhkan kompleksitas yang cepat. Sementara *hash pattern*, cukup diketahui hasil *hash* akhirnya saja untuk dicocokkan dengan *hash* teks di masing-masing indeks.

A. Pre-processing

Sebelum melakukan *pre-process* yang lain, dilakukan pembangkitan pangkat *key generator* ($PRIME = 131$) modulo ($MOD = 100000007$). Modulo dipilih dari bilangan prima yang sangat besar untuk menghindari seminimal mungkin *collision* ketika *hash*. Di bawah ini adalah implementasi *pre-process power of PRIME*.

```
power[0] = 1;
for(int i = 1; i < 2 * N; i++) {
    power[i] = power[i - 1] * PRIME;
    power[i] %= MOD;
}
```

Kemudian dilakukan implementasi *pre-processing* *hash* teks dalam bahasa C++. Kode berikut sekaligus membangkitkan *array presumnya* untuk memudahkan

menjawab *range query* yang dibutuhkan suatu saat nanti. Berikut implementasinya.

```
for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= m; j++) {
        pre[i][j] = text[i][j];
        pre[i][j] += pre[i - 1][j] * PRIME;
        pre[i][j] += pre[i][j - 1] * PRIME;
        pre[i][j] -= pre[i-1][j-1] * power[2];
        pre[i][j] %= MOD;
        if(pre[i][j] < 0) {
            pre[i][j] += MOD;
        }
    }
}
```

Setelah itu, dilakukan *pre-processing* terhadap *pattern*. Implementasinya sebagai berikut.

```
for(int i = 1; i <= n2; i++) {
    for(int j = 1; j <= m2; j++) {
        val[i][j] = pattern[i][j];
        val[i][j] += val[i - 1][j] * PRIME;
        val[i][j] += val[i][j - 1] * PRIME;
        val[i][j] -= val[i-1][j-1] * power[2];
        val[i][j] %= MOD;
        if(val[i][j] < 0) {
            val[i][j] += MOD;
        }
    }
}
```

Hasil *hash* dari *pattern* tersimpan dalam $val[n2][m2]$. Nilai *hash* inilah yang akan dicocokkan dengan nilai *hash* dari masing-masing indeks teks.

B. Implementasi

Hal utama yang perlu diimplementasi adalah fungsi untuk mengambil nilai dari *array* 2D hasil *presum* *hash* teks. Lebih spesifiknya kita perlu mengambil *hash-value* dari *array* dengan indeks(*i, j*) dengan mengambil sebanyak (*nlen, mlen*) vertikal dan horizontal indeks.

Berikut adalah implementasinya.

```
long long compute
(int i, int j, int nlen, int mlen)
{
    // right-below corner of i-index
    int ci = i + nlen - 1;
    // right-below corner of j-index
    int cj = j + mlen - 1;

    long long ret = pre[ci][cj];
    ret -= pre[i - 1][cj] * power[nlen];
    ret -= pre[ci][j - 1] * power[mlen];
    ret += pre[i-1][j-1] * power[nlen+mlen];
    ret %= MOD;
    if(ret < 0) {
```

```

    ret += MOD;
}
return ret;
}

```

Program utama (pengecekan 2D string) diimplementasi dengan 2 *nested loop* sebagai berikut.

```

long long value = val[n2][m2];
for(int i = 1; i + n2 <= n + 1; i++) {
    for(int j = 1; j + m2 <= m + 1; j++) {
        if(compute(i, j, n2, m2) == value) {
            printf("Found at %d %d\n", i, j);
        }
    }
}
}

```

C. Eksperimen

Eksperimen pada program terdiri dari 6 *testcase*. Format masukan adalah *text* dan *pattern* yang akan di-*match*. Format *output* adalah keterangan apakah *pattern* ditemukan beserta indeksnya, atau menampilkan *pattern* tidak ditemukan dalam *text*. Berikut adalah ukuran dari masing-masing *testcase*.

Test case	n - text	m - text	n-pattern	m-pattern
1	5	5	2	2
2	1000	1000	4	4
3	1000	1000	500	500
4	1000	1000	100	100
5	1000	1000	690	799
6	1000	1000	1	10001

Eksperimen menghasilkan data sebagai berikut.

Test case	Expected	Reality	Execution time
1	Found	Found at index 3 3	1 ms
2	Found	Found at index 957 874	120 ms
3	Found	Found at index 157 346	104 ms
4	No	Not found	97 ms
5	No	Not found	102 ms
6	No	Not found	59 ms

IV. ANALISIS

A. Analisis Kompleksitas

Pre-processing memerlukan $O(n^2 + m^2)$ operasi. Algoritma utama melakukan *looping* sebanyak 2 *nested*

sehingga algoritma total berkompleksitas $O(n^2)$. Ini disebabkan karena fungsi *compute* dipanggil dengan kompleksitas $O(1)$.

Kompleksitas total algoritma *pattern matching* untuk pencocokan string 2D ini adalah $O(n^2 + m^2)$.

B. Kemungkinan Collision

Implementasi *hash* tentunya tidak lepas dari kendala *collision*. Dengan *hash* modulo 1 milyar + 7, kolisi bisa dihindari tetapi tidak menutup kemungkinan *collision* tak sengaja bisa terjadi. Tetapi untuk keperluan *hashing* teks planar atau teks random, sangat kecil kemungkinan kolisi tak sengaja bisa terjadi. Apabila *mapping* dari objek ke nilai *hash* terdistribusi normal, maka peluang terjadinya *collision* hanya 0,0000001%.

Implementasi *hashing* pada algoritma *string matching* ini sangat sederhana. Akibatnya input di bawah ini, hasil yang diinginkan keliru.

```

Input text :
ab
ca
Input pattern :
ac
ba
Expected : Not found
Reality : Found at index 1 1

```

Ini terjadi karena *hash* antara karakter b dan c memiliki *power of prime* yang sama. Akibatnya ketika karakter dipertukarkan, program mengira teks sama dengan *pattern* tersebut.

C. Menghindari Hash Collisions dengan Hash Berlapis

Implementasi *hash* tidak sepenuhnya dapat lepas dari *collisions* karena bagaimana besar prima modulo yang digunakan, ukuran *hashed-value* tetap saja ada batasnya (*limited value*). Sementara banyak konfigurasi string untuk *n* dan *m* manapun tidak ada batasnya (*infinite value*). Oleh karena itu, secara teoritis tentu saja kemungkinan *collisions* masih ada walaupun persentasenya kecil sekali.

Hash collisions dapat dihindari dengan penggunaan *hash* secara berlapis. *Hash* bisa dilakukan 4 kali di setiap rotasi text. Artinya setiap *hash* telah dilakukan, *text* di-rotate, kemudian di-*hash* lagi. Ini berakibat pengecekan *hash text* dengan *hash pattern* harus dilakukan 4 kali pula.

Penggunaan *hash* juga dapat divariasikan menggunakan nilai bilangan prima yang lain untuk PRIME ataupun modulo *hash*. Semakin variatif penggunaan *hash* dan

semakin berlapis, data yang dikompresi menggunakan hash akan semakin terjamin kesahan dan kevalidannya.

V. KESIMPULAN

Banyak teknik yang dapat digunakan untuk permasalahan *string matching*, tidak hanya algoritma yang *well-known* seperti KMP dan *Boyer-Moore*. Salah satu teknik kreatif yang dapat diimplementasikan adalah gabungan antara *presum dynamic programming* dan *hashing*. Dengan kompleksitas $O(n^2)$ algoritma ini cukup cepat untuk dijalankan pada waktu yang rasional.

Penggunaan *hash* dalam mengimplementasi sesuatu tentu tidak lepas dari resiko *collision*. Salah satu metode untuk menghindari *collision* adalah dengan memanfaatkan fungsi *hash* yang berlapis. Variasi bilangan prima juga dapat dimanfaatkan.

Pattern matching String 2D dapat diaplikasikan ke dalam pencocokan citra (*image*) di mana citra ditranslasikan menjadi string 2D, kemudian permasalahan dikonversi menjadi pencocokan string 2D.

VI. KATA PENUTUP

Makalah ini ditulis oleh Luqman A. Siswanto untuk keperluan memenuhi tugas mata kuliah IF2211 Strategi Algoritma di Institut Teknologi Bandung semester genap tahun ajaran 2014/2015.

Penulis ingin menghaturkan banyak terima kasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T. dan Bu Dra. Harlili S., M.Sc. yang telah mengampu mata kuliah IF2211 Strategi Algoritma selama satu semester ini. Tanpa bimbingan dan arahan beliau, makalah ini tidak akan dapat diselesaikan. Penulis juga menyampaikan terima kasih kepada rekan-rekan HMIF ITB (Himpunan Mahasiswa Informatika ITB) atas kesempatan dan kerjasamanya.

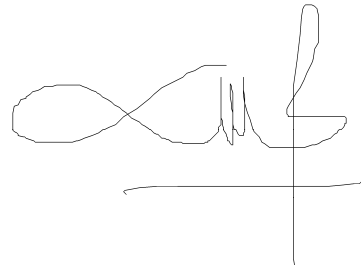
REFERENSI

- [1] Munir, Rinaldi. 2009. *Diktat Kuliah Strategi Algoritma*. Bandung : Program Studi Teknik Informatika Institut Teknologi Bandung.
- [2] Halim, Steven Halim and Felix. 2010. *Competitive Programming 3 : Increasing the Lower Bound of Programming Contests*. Singapore : National University of Singapore.
- [3] *Universidad de Valladolid Online Judge*. <http://uva.onlinejudge.org>. Diakses pada 4 Mei 2015.

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, 4 Mei 2015



Luqman A. Siswanto
13513024