KERJA PRAKTIK - EF234603

Desain Algoritma Randomisasi pada persoalan Hashing: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash

Laboratorium Algoritma dan Pemrograman Teknik Informatika ITS
Jl. Teknik Kimia, Keputih, Kec. Sukolilo, Surabaya, Jawa Timur 60111
Periode: 1 Agustus 2024 - 31 Oktober 2024

Oleh:

Alexander Weynard Samsico 5025211014

Pembimbing Departemen
Dr. Yudhi Purwananto, S.Kom., M.Kom.

Pembimbing Lapangan
Rully Soelaiman, S.Kom., M.Kom.

DEPARTEMEN TEKNIK INFORMATIKA
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya 2024

[Halaman ini sengaja dikosongkan]

KERJA PRAKTIK - EF234603

Desain Algoritma Randomisasi pada persoalan Hashing: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash

Laboratorium Algoritma dan Pemrograman Teknik Informatika ITS
Jl. Teknik Kimia, Keputih, Kec. Sukolilo, Surabaya, Jawa Timur 60111
Periode: 1 Agustus 2024 - 31 Oktober 2024

Oleh:

Alexander Weynard Samsico 5025211014

Pembimbing Departemen
Dr. Yudhi Purwananto, S.Kom., M.Kom.

Pembimbing Lapangan
Rully Soelaiman, S.Kom., M.Kom.

DEPARTEMEN TEKNIK INFORMATIKA
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya 2024

[Halaman ini sengaja dikosongkan]

LEMBAR PENGESAHAN
KERJA PRAKTIK

Desain Algoritma Randomisasi pada persoalan Hashing: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash

Oleh:

Alexander Weynard Samsico 5025211014

Disetujui oleh Pembimbing Kerja Praktik:

1. Dr. Yudhi Purwananto, S.Kom., M.Kom.
   NIP. 197007141997031002

(Pembimbing Departemen)

2. Rully Soelaiman, S.Kom., M.Kom.
   NIP. 197002131994021001

(Pembimbing Lapangan)

[Halaman ini sengaja dikosongkan]

Desain Algoritma Randomisasi pada persoalan Hashing: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash

Nama Mahasiswa	: Alexander Weynard Samsico
NRP	: 5025211014
Departemen	: Teknik Informatika FTEIC-ITS
Pembimbing Departemen	: Dr. Yudhi Purwananto, S.Kom.,M.Kom.
Pembimbing Lapangan	: Rully Soelaiman, S.Kom., M.Kom.

ABSTRAK

Hashing merupakan suatu teknik yang di mana mengubah suatu data, seperti *string*, menjadi suatu nilai *hash* dengan fungsi tertentu. Diberikan suatu fungsi *hashing* yang menerima suatu *string* untuk diubah menjadi nilai *hash* dengan parameter bilangan basis B dan bilangan modulo M. Pada persoalan *hashing* ini, diberikan pasangan B dan M sebanyak K dan meminta untuk mencari dua *string* berbeda yang memiliki nilai *hash* yang sama berdasarkan setiap pasangan B dan M yang diberikan. Teknik ini dikenal sebagai teknik *anti-hash* di mana untuk mencari dua string berdasarkan nilai hash yang sudah diketahui. Jika menggunakan pendekatan langsung berupa mencari string dari setiap kombinasi yang ada, maka tidak efisien karena membutuhkan waktu yang sangat lama berdasarkan panjang batasan *string* yang diberikan. Kerja Praktik ini melakukan desain algoritma yang efektif dan efisien untuk menyelesaikan persoalan *hashing*.

Kerja Praktik ini mengusulkan untuk mendesain algoritma randomisasi untuk menyelesaikan persoalan *hashing*. Algoritma Randomisasi merupakan suatu algoritma yang di mana membangkitkan suatu nilai secara acak dengan banyak percobaan

sehingga mendapatkan nilai yang benar dan sesuai dengan teori probabilitas. Algoritma randomisasi juga mendukung Birthday Paradox sebagai dasar teori untuk menentukan berapa banyaknya percobaan yang diperlukan agar mendapatkan hasil yang diinginkan, dalam kasus ini adalah dua *string*. Algoritma ini menggunakan teknik *Birthday Attack* dan *Composition Attack* untuk mencari dua *string* dengan nilai *hash* yang sama berdasarkan banyaknya pasang B dan M yang diberikan. Teknik *Birthday Attack* dapat mencari dua *string* yang sama pada setiap pasang B dan M serta *Composition Attack* sebagai transisi untuk setiap pasang B dan M yang ditentukan.

Metode uji coba yang dilakukan adalah melalui uji coba kebenaran dengan mengirimkan kode sumber berdasarkan desain algoritma yang dibuat pada situs penilaian *Sphere Online Judge* (SPOJ). Berdasarkan uji coba tersebut, didapatkan hasil kinerja randomisasi untuk menyelesaikan persoalan hashing studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash diraih dengan waktu minimum 1.47 detik dengan 62 MB sebagai hasil terbaik. Hasil kinerja juga menunjukkan waktu yang dibutuhkan dengan rata-rata 1.865 detik dan memori yang dibutuhkan dengan rata-rata 70.65 MB serta memori minimum 59 MB. Jadi, desain algoritma randomisasi efektif dan efisien dalam menyelesaikan persoalan *hashing.*

Kata Kunci : Hashing, Anti-Hash, Algoritma Randomisasi, Birthday Paradox, Birthday Attack, Composition Attack

KATA PENGANTAR

Penulis ucapkan puji syukur kepada Tuhan Yang Maha Esa atas rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan salah satu kewajiban penulis sebagai mahasiswa Departemen Teknik Informatika yaitu Kerja Praktik yang berjudul Desain Algoritma Randomisasi pada persoalan Hashing: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash.

Dengan segala kerendahan hati, penulis menyadari bahwa masih banyak kekurangan baik dalam melaksanakan kerja praktik maupun menyusun buku laporan kerja praktik ini. Namun penulis berharap buku laporan kerja praktik ini dapat menambah wawasan pembaca dan dapat menjadi sumber referensi

Dalam proses penyusunan buku laporan ini, penulis telah banyak mendapatkan bimbingan, bantuan, dan dukungan dari berbagai pihak. Oleh karena itu, penulis ucapkan rasa hormat dan terima kasih yang sebesar-besarnya kepada:

1. Kedua orang tua penulis.
2. Bapak Dr. Yudhi Purwananto, S.Kom., M.Kom. selaku dosen pembimbing kerja praktik dan koordinator kerja praktik.
3. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku pembimbing lapangan selama kerja praktik berlangsung.
4. Teman-teman penulis yang senantiasa memberikan semangat ketika penulis melaksanakan Kerja Praktik.

Surabaya, 31 Oktober 2024

Alexander Weynard Samsico

[Halaman ini sengaja dikosongkan]

DAFTAR ISI

LEMBAR PENGESAHAN v

KATA PENGANTAR ix

DAFTAR ISI xi

DAFTAR GAMBAR xv

DAFTAR TABEL xvii

DAFTAR KODE SEMU xix

DAFTAR KODE SUMBER xxi

BAB I PENDAHULUAN 1

1.1. Latar Belakang 1

1.2. Tujuan 3

1.3. Manfaat 3

1.4. Rumusan Masalah 3

1.5. Lokasi dan Waktu Kerja Praktik 4

1.6. Metodologi Kerja Praktik 4

1.6.1. Perumusan Masalah 4

1.6.2. Studi Literatur 5

1.6.3. Desain Algoritma 5

1.6.4. Implementasi Sistem 6

1.6.5. Pengujian dan Evaluasi 6

1.6.6. Kesimpulan dan Saran 7

1.7. Sistematika Laporan 7
1.7.1. Bab I Pendahuluan 7
1.7.2. Bab II Profil Perusahaan 7
1.7.3. Bab III Tinjauan Pustaka 7
1.7.4. Bab IV Analisis dan Perancangan Infrastruktur Sistem 8
1.7.5. Bab V Implementasi Sistem 8
1.7.6. Bab VI Pengujian dan Evaluasi 8
1.7.7. Bab VII Kesimpulan dan Saran 8
BAB II PROFIL INSTANSI 9
2.1. Profil Laboratorium Algoritma dan Pemrograman ITS 9
2.2. Logo Perusahaan 9
2.3. Visi dan Misi 10
2.3.1. Visi 10
2.3.2. Misi 11
2.4. Struktur Organisasi 11
2.5. Lokasi 12
BAB III TINJAUAN PUSTAKA 13
3.1. Deskripsi Umum Permasalahan 13
3.2. Algoritma Randomisasi 15
3.3. Birthday Attack 15
3.4. Composition Attack 16
3.5. Strategi Penyelesaian 17

3.5.1 Uji Coba pencarian k_paket sebagai strategi penyelesaian ... 18
3.6. .NET C# ... 22
BAB IV DESAIN ALGORITMA ... 27
4.1. Deskripsi Umum Permasalahan ... 27
4.2. Desain Algorima ... 30
4.2.1 Model Operasi Birthday Attack Pertama ... 30
4.2.2 Model Composition Attack ... 35
BAB V IMPLEMENTASI SISTEM ... 41
5.1. Lingkungan Implementasi ... 41
5.2. Penggunaan Fast Input Output untuk optimasi performa ... 42
5.3. Variabel Global ... 45
5.4. Fungsi ModEx ... 47
5.5. Fungsi GetHash ... 48
5.6. Fungsi GetHashReplace ... 48
5.7. Fungsi GetTotalHash ... 49
5.8. Fungsi GetTotalHashReplace ... 50
5.9. Fungsi GetRandomString ... 50
5.10. Fungsi BuildStringFirst ... 52
5.11. Fungsi ConvertAlphabet ... 53
5.12. Fungsi GetStringNext ... 53
5.13. Fungsi BuildStringContinue ... 56
5.14. Fungsi Main ... 57

BAB VI PENGUJIAN DAN EVALUASI 59
6.1. Lingkungan Uji Coba 59
6.2. Skenario Uji Coba 60
6.3. Uji Coba Kebenaran 60
6.4. Evaluasi Pengujian 65
BAB VII KESIMPULAN DAN SARAN 67
7.1. Kesimpulan 67
7.2. Saran 68
DAFTAR PUSTAKA 69
BIODATA PENULIS 71

DAFTAR GAMBAR

Gambar 2.1 Logo Laboratorium Algoritma dan Pemrograman .. 10
Gambar 2.2 Struktur Organisasi Laboratorium Algoritma dan Pemrograman 11
Gambar 3.1 Uji kebenaran dengan k_paket = 1 20
Gambar 3.2 Uji kebenaran dengan k_paket = 2 20
Gambar 3.3 Uji kebenaran dengan k_paket = 3 20
Gambar 3.4 Uji kebenaran dengan k_paket = 4 20
Gambar 3.5 Uji kebenaran dengan k_paket = 6 20
Gambar 3.6 Uji kebenaran dengan k_paket = 12 20
Gambar 3.7 Hasil distribusi pembangkitan kelas Random 25
Gambar 4.1 Diagram Alur Desain Algoritma 28
Gambar 6.1 Hasil uji coba pada situs SPOJ 60
Gambar 6.2 Peringkat hasil pada situs SPOJ 61
Gambar 6.3 Hasil 20 kali uji coba pada situs SPOJ 63
Gambar 6.4 Grafik hasil waktu uji coba pada situs SPOJ sebanyak 20 kali 64
Gambar 6.5 Grafik hasil memori uji coba pada situs SPOJ sebanyak 20 kali 64

[Halaman ini sengaja dikosongkan]

DAFTAR TABEL

Tabel 3.1 Kompleksitas waktu dan memori serta kasus terburuk pembagian pasang 18
Tabel 3.2 Properti Dictionary<TKey, TValue> C# 22
Tabel 3.3 Metode Dictionary<TKey, TValue> C# 23
Tabel 5.1 Spesifikasi Lingkungan Implementasi Program 41
Tabel 5.2 Penjelasan Variabel Global 46
Tabel 6.1 Spesifikasi Lingkungan Uji Coba 59
Tabel 6.2 Hasil uji coba 20 kali pada situs SPOJ 61

[Halaman ini sengaja dikosongkan]

DAFTAR KODE SEMU

Kode Semu 1.1: GetHash 1
Kode Semu 3.1: GetHash 13
Kode Semu 4.1: Main... 29
Kode Semu 4.2: BuildStringFirst 31
Kode Semu 4.3: GetRandomString 31
Kode Semu 4.4: GetHashReplace 33
Kode Semu 4.5: BuildStringContinue 35
Kode Semu 4.6: GetStringNext 36
Kode Semu 4.7: GetTotalHash 38
Kode Semu 4.8: GetTotalHashReplace 39

[Halaman ini sengaja dikosongkan]

DAFTAR KODE SUMBER

Kode Sumber 5.1 *Fast Input Output* untuk optimasi performa ... 44
Kode Sumber 5.2 Variabel Global ... 45
Kode Sumber 5.3 Fungsi *ModEx* ... 47
Kode Sumber 5.4 Fungsi *GetHash* ... 48
Kode Sumber 5.5 Fungsi *GetHashReplace* ... 49
Kode Sumber 5.6 Fungsi *GetTotalHash* ... 49
Kode Sumber 5.7 Fungsi GetTotalHashReplace ... 50
Kode Sumber 5.8 Fungsi *GetRandomString* ... 52
Kode Sumber 5.9 Fungsi *BuildStringFirst* ... 52
Kode Sumber 5.10 Fungsi *ConvertAlphabet* ... 53
Kode Sumber 5.11 Fungsi *GetStringNext* ... 56
Kode Sumber 5.12 Fungsi *BuildStringContinue* ... 56
Kode Sumber 5.13 Fungsi *Main* ... 58

[Halaman ini sengaja dikosongkan]

BAB I
PENDAHULUAN

1.1. Latar Belakang

Hashing merupakan salah satu cara penanganan dalam keamanan data. Data dapat berupa suatu kata atau kalimat atau *string* yang dimaksud rahasia sehingga harus diamankan. Cara kerja *hashing* adalah menyamarkan suatu *string* menjadi suatu nilai *hash* dengan suatu algoritma tertentu. Nilai *hash* ini dapat sulit untuk didapatkan nilai awal data asli karena dengan algoritma yang tidak diketahui atau banyaknya kemungkinan data atau *string* yang memiliki hasil nilai *hash* yang sama. Salah satu tantangan adalah bagaimana agar mendapatkan suatu data yang serupa dengan hanya diketahuinya nilai *hash*, hal ini disebut juga dengan *anti hash*.

Hal ini serupa dengan persoalan *hashing* Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash. Diberikan suatu fungsi *GetHash* untuk mentransformasikan sebuah *string* s menjadi suatu nilai *hash*nya dengan parameter bilangan basis B dan bilangan modulo M sebagai Kode Semu 1.1.

Kode Semu 1.1: GetHash

```
1   GetHash(string str, int B, int M)
2       hash ← 0
3       for chr in str
4           hash ← (hash * B + chr – ‘a’ + 1) % M
5       end for
6       return hash
```

String yang dimasukkan terdiri dari sebatas ‘a’ – ‘z’. Diberikan kumpulan pasangan B dan M sejumlah K, persoalan *hashing* ini meminta untuk mencari dua *string* dengan nilai *hash* yang sama berdasarkan dari setiap pasangan B dan M. Dalam kata lain, dua *string* yang ditemukan jika memproduksi nilai *hash* dengan setiap pasang B dan M maka akan mengeluarkan nilai *hash* yang sama. Syarat dua *string* tersebut adalah harus berbeda, tidak boleh kosong, dan tidak lebih dari 65536 karakter. Cara untuk mendapatkan dua *string* bisa dengan mencoba setiap kombinasi *string* yang berdasarkan dari huruf ‘a’-‘z’. Akan tetapi, jika menggunakan panjang *string* 65536, maka melakukan penelusuran *string* sebanyak 26^{65536} kombinasi. Sehingga cara ini membutuhkan kompleksitas waktu yang sangat lama dengan $O(K^* (26^{65536}))$ untuk setiap kasus uji. Oleh karena itu, cara ini kurang cocok sebagai penyelesaian persoalan ini karena kurang efisien.

Pada Kerja Praktik ini, diusulkan penerapan algoritma randomisasi sebagai penyelesaian untuk persoalan *hashing* Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash. Hasil dari Kerja Praktik ini diharapkan dapat menyelesaikan persoalan *hashing* Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash dengan algoritma randomisasi dengan efektif dan efisien.

1.2. Tujuan

Tujuan dari Kerja Prakik ini dijabarkan sebagai berikut:

1. Melakukan desain algoritma randomisasi untuk menyelesaikan persoalan hashing studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash.
2. Mengimplementasi algoritma randomisasi untuk menyelesaikan persoalan hashing studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash.
3. Mengevaluasi hasil kinerja randomisasi untuk menyelesaikan persoalan hashing studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash.

1.3. Manfaat

Manfaat yang diperoleh dari Kerja Praktik ini adalah memberikan pemahaman baru dan mendalam terkait konsep algoritma randomisasi dalam penemuan dua *string* dengan nilai *hash* yang sama berdasarkan fungsi *hashing* yang diberikan dengan efektif dan efisien.

1.4. Rumusan Masalah

Rumusan masalah dari Kerja Praktik ini dijabarkan sebagai berikut:

1. Bagaimana desain algoritma randomisasi untuk menyelesaikan studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash?
2. Bagaimana implementasi algoritma randomisasi untuk menyelesaikan studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash?

3. Bagaimana hasil kinerja randomisasi untuk menyelesaikan studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash?

1.5. Lokasi dan Waktu Kerja Praktik

Kerja Praktik ini dilakukan pada lokasi dan waktu sebagai berikut:

Lokasi : Laboratorium Algoritma dan Pemrograman
Alamat : Jl. Teknik Kimia, Keputih, Sukolilo, Surabaya
Waktu : 1 Agustus 2024 – 31 Oktober 2024
Hari Kerja : Senin – Jumat
Jam Kerja : 08.00 WIB – 16.00 WIB

Kerja Praktik ini lebih sering dilakukan secara *Work From Home* dan daring karena lebih mudah dilakukan dan tidak ada kebutuhan yang mendesak untuk dibutuhkan kerja secara luring.

1.6. Metodologi Kerja Praktik

Metodologi dalam pembuatan buku kerja praktik meliputi :

1.6.1. Perumusan Masalah

Dari latar belakang yang telah diberikan, dilakukan perumusan masalah yang akan diselesaikan. Proses ini bertujuan untuk memberikan gambaran yang lebih jelas mengenai tujuan dan manfaat dari kerja praktik yang akan dilaksanakan. Proses dan alur kerja

diharapkan dapat terarah sesuai dengan permasalahan yang diselesaikan.

1.6.2. Studi Literatur

Setelah mendapatkan rumusan masalah yang dilakukan, Tahap ini dilakukan pengumpulan sumber-sumber yang relevan terkait topik penelitian yang diselesaikan. Studi literatur yang dipelajari mencakup teori-teori yang membantu dalam penelitian ini yaitu Algoritma Randomisasi, *Birthday Attack*, *Composition Attack*, dan penggunaan bahasa .NET C# sebagai bahasa pemrograman yang digunakan. Selain itu, dilakukan analisis strategi penyelesaian dalam pengumpulan teori-teori menjadi suatu metode yang optimal untuk digunakan dalam penyelesaian persoalan *hashing*. Studi literatur ini bertujuan untuk mendapatkan pemahaman yang mendalam terkait konsep-konsep teori yang akan digunakan dalam desain algoritma randomisasi.

1.6.3. Desain Algoritma

Setelah mendapatkan konsep strategi penyelesaian dari studi literatur, dapat dilakukan perancangan desain algoritma yang akan digunakan sebagai implementasi program. Desain ini diawali dengan pembuatan fungsi utama yaitu main, kemudian memodelkan Model *Birthday Attack* Pertama dan Model *Composition Attack*. Selain itu, desain ini juga meliputi beberapa fungsi pendukung tambahan seperti *GetHashReplace* untuk mengganti nilai *hash* dari suatu karakter, *GetTotalHash* untuk menjumlahkan nilai *hash*, dan *GetTotalHashReplace* untuk mengganti nilai *hash* dari suatu total *hash*. Desain algoritma ini bertujuan untuk

dapat menghasilkan implementasi sistem yang benar sehingga dapat diterima pada situs penilaian SPOJ.

1.6.4. Implementasi Sistem

Implementasi ini merupakan tahap merealisasikan desain yang telah dibuat ke dalam kode sumber pemrograman. Kode sumber pemrograman dibuat sesuai spesifikasi desain pada bahasa pemrograman .NET C#. Implementasi dilakukan pada fungsi-fungsi yang sudah dikemukakan seperti *GetHash*, *GetHashReplace*, *GetTotalHash*, *GetTotalHashReplace*, *GetRandomString* dengan *BuildStringFirst* sebagai model *Birthday Attack* Pertama, *GetStringNext* dengan *BuildStringContinue* sebagai model *Composition Attack*. Selain itu, adanya beberapa fungsi tambahan sebagai pendukung seperti *Fast Input Output*, *ModEx* sebagai modular eksponen, dan *ConvertAlphabet* untuk konversi menjadi *string*. Setiap fungsi diimplementasikan secara terpisah dan digabungkan sebagai satu sistem. Selain itu, implementasi setiap fungsi ini juga diuji untuk memastikan bahwa setiap fungsionalitas dapat bekerja dengan baik.

1.6.5. Pengujian dan Evaluasi

Pengujian dilakukan untuk mengetes bahwa implementasi sistem yang telah dibuat sesuai spesifikasi desain dapat menyelesaikan masalah persoalan *hashing* Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash dengan efektif dan efisien. Pengujian dilakukan pada situs penilaian SPOJ untuk menguji kebenaran dan menguji hasil kinerja dalam waktu dan memori dari

implementasi solusi yang telah dibuat. Hasil pengujian tersebut akan dievaluasikan sebagai hasil kinerja implementasi program dalam menyelesaikan persoalan tersebut beserta dengan analisis berapa kinerja waktu dan memori yang dibutuhkan pada program tersebut.

1.6.6. Kesimpulan dan Saran

Hasil pengujian yang dilakukan dibentuk suatu kesimpulan yang menjawab perumusan masalah yang telah diketahui. Pengujian ini telah menunjukkan bahwa desain algoritma randomisasi dapat menyelesaikan permasalahan dengan model Birthday Attack dan Composition Attack. Implementasi algoritma randomisasi juga dilakukan dengan mengikuti alur desain algoritma yang sudah dibuat. Kemudian, hasil kinerja pengujian dibuat kesimpulan berapa waktu dan memori yang diperlukan oleh program. Setelah itu terdapat saran agar dapat meningkatkan efisiensi program pada implementasi yang dibuat.

1.7. Sistematika Laporan

1.7.1. Bab I Pendahuluan

Bab ini berisi latar belakang, tujuan, manfaat, rumusan masalah, lokasi dan waktu kerja praktik, metodologi, dan sistematika laporan.

1.7.2. Bab II Profil Perusahaan

Bab ini berisi profil Laboratorium Algoritma dan Pemrograman, visi misi, struktur organisasi, serta lokasi instansi.

1.7.3. Bab III Tinjauan Pustaka

Bab ini berisi dasar teori dari teknologi yang digunakan dalam menyelesaikan proyek kerja praktik.

1.7.4. Bab IV Analisis dan Perancangan Infrastruktur Sistem

Bab ini berisi mengenai tahap analisis sistem dalam menyelesaikan proyek kerja praktik.

1.7.5. Bab V Implementasi Sistem

Bab ini berisi uraian tahap - tahap yang dilakukan untuk proses implementasi sistem.

1.7.6. Bab VI Pengujian dan Evaluasi

Bab ini berisi hasil uji coba dan evaluasi dari sistem yang telah dikembangkan selama pelaksanaan kerja praktik.

1.7.7. Bab VII Kesimpulan dan Saran

Bab ini berisi kesimpulan dan saran yang didapat dari proses pelaksanaan kerja praktik.

BAB II
PROFIL INSTANSI

2.1. Profil Laboratorium Algoritma dan Pemrograman ITS

Laboratorium Algoritma dan Pemrograman merupakan salah satu Laboratorium di Teknik Informatika ITS yang berfokus pada bidang keahlian yang menawarkan kemampuan lulusan dalam merancang dan menganalisis algoritma dalam menyelesaikan masalah secara efektif dan efisien, mengaplikasikan model pemrograman berbagai bahasa yang ada, serta memilih bahasa pemrograman yang tepat untuk menghasilkan aplikasi yang sesuai, seperti aplikasi berbasis kerangka kerja dan aplikasi pada perangkat bergerak.

Mata kuliah yang terkait dengan Algoritma dan Pemrograman meliputi Dasar pemrograman, Struktur Data, Pemrograman Berorientasi Objek, Perancangan dan Analisis Algoritma, Pemrograman Web, serta Pemrograman Berbasis Kerangka Kerja. Sementara itu, mata kuliah bidang keahlian laboratorium ini meliputi Pengembangan Analisis Algoritma, Pemrograman Berbasis Antarmuka, Pemrograman Perangkat Bergerak, Topik Khusus Algoritma dan Pemrograman.

2.2. Logo Perusahaan

Logo dari Laboratorium Algoritma dan Pemrograman dapat dilihat pada Gambar 2.1. Logo ini mewakili berdirinya Lab Algoritma dan Pemrograman di dalam naungan Departemen Teknik Informatika, Fakultas

Teknologi Elektro dan Informatika Cerdas, Institut Teknologi Sepuluh Nopember.

Gambar 2.1 Logo Laboratorium Algoritma dan Pemrograman

2.3. Visi dan Misi

2.3.1. Visi

Sejalan dengan visi ITS yaitu menjadi perguruan tinggi dengan reputasi internasional dalam ilmu pengetahuan, teknologi, dan seni, terutama yang menunjang industri dan kelautan yang berwawasan lingkungan, maka visi Departemen Informatika adalah menjadi inovator bidang informatika yang unggul di tingkat nasional dengan reputasi internasional, serta berperan aktif dalam upaya memajukan dan menyejahterakan bangsa

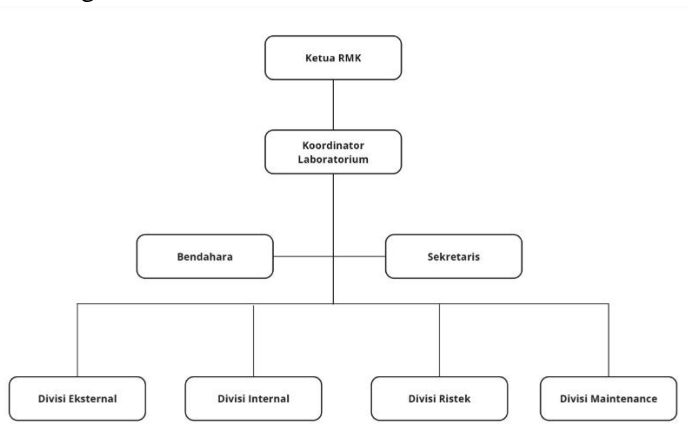
2.3.2. Misi

Departemen Informatika memiliki misi sebagai berikut:

1. Menyelenggarakan proses pembelajaran yang berkualitas, dan memenuhi standar nasional maupun internasional,
2. Melaksanakan penelitian yang inovatif, bermutu, dan bermanfaat,
3. Meningkatkan pemanfaatan teknologi informasi dan komunikasi untuk masyarakat,
4. Menjalin kemitraan dengan berbagai lembaga, baik di dalam maupun di luar negeri.

2.4. Struktur Organisasi

Struktur organisasi yang terdapat pada Laboratorium Algoritma dan Pemrograman ITS untuk masa kepengurusan 2024 dapat direpresentasikan sebagai bagan Gambar 2.2.



Gambar 2.2 Struktur Organisasi Laboratorium Algoritma dan Pemrograman

2.5. Lokasi

Laboratorium Algoritma dan Pemrograman Departemen Teknik Informatika, Jl. Teknik Kimia, Keputih, Sukolilo, Surabaya

BAB III
TINJAUAN PUSTAKA

3.1. Deskripsi Umum Permasalahan

Studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash merupakan permasalahan untuk mencari dua string dengan nilai hash-hash yang sama. Diberikan suatu *polynomial hash* yang akan mengonversi sebuah string S, terdiri dari huruf kecil a-z, dengan diberikan untuk basis B dan modulo M. Berikut adalah kode semu pembuatan nilai hash dari string.

Kode Semu 3.1: GetHash

```
1   GetHash(string str, int B, int M)
2       hash ← 0
3       for chr in str
4           hash ← (hash * B + chr – 'a' + 1) % M
5       end for
6       return hash
```

Diberikan suatu pasangan B dan M sebanyak K untuk mencari dua *string* yang berbeda yang ketika melakukan hashing dengan setiap B dan M diberikan memiliki nilai *hash* yang sama.

Berikut adalah format masukan dari permasalahan:

1. Baris pertama adalah T, sebagai jumlah uji kasus
2. Setiap uji kasus, baris berikutnya berisi K sebagai jumlah pasangan B dan M.

3. K baris berikutnya, berisi B_i dan M_i.

Format keluaran dari permasalahan adalah hasil dua string berbeda yang dipisahkan dengan sebuah spasi. Adapun beberapa batasan permasalahan sebagai berikut

1. $1 \leq T \leq 20$
2. $1 \leq K \leq 12$
3. $32 \leq B_i \leq 256$
4. $1 \leq M_i \leq 256$
5. Keluaran string tidak bisa kosong dan tidak lebih dari 65536 huruf
6. Batasan waktu: 8 detik
7. Batasan memori: 1536 MB

Permasalahan ini dibutuhkan struktur data yang cepat dalam mengelola dan penemuan nilai-nilai *hash*. Struktur data Dictionary dari .NET C#, dikenal dengan tabel hash yang cepat, dapat dimanfaatkan sebagai penyimpanan nilai-nilai hash beserta string yang akan dibandingkan dengan string yang lain. Permasalahan ini murni dalam perbandingan antara dua *string* dengan hash value yang sama sehingga tidak ditemukan teori yang spesifik untuk membuat dua string secara langsung dengan fungsi *GetHash* yang diberikan. Jika menggunakan pendekatan *brute force,* yang di mana mengecek satu per satu string yang memiliki nilai hash yang sama dengan maksimal K = 12, maka akan didapatkan kompleksitas waktu $O(26^{65536} * 12)$ secara maksimal karena diperlukan mengecek setiap huruf pada *string* dengan panjang maksimal 65536 huruf pada setiap K. Kompleksitas waktu *brute force* tidak cocok karena harus mencari dua *string* di antara 26^{65536}, terutama mencari secara berurutan. Oleh

karena itu, Algoritma randomisasi (algoritma acak) dapat dimanfaatkan sebagai pembangkitan nilai *string* secara acak untuk menemukan dua hasil *string* tersebut dengan menggunakan teorema probabilitas.

3.2. Algoritma Randomisasi

Algoritma randomisasi merupakan algoritma yang menggunakan pendekatan pembangkitan nilai secara acak. Tujuan dari algoritma acak ini adalah melakukan beberapa percobaan dengan nilai acak dan mendapatkan nilai yang benar. Algoritma ini akan lebih cepat jika dibandingkan dengan metode *brute force* karena memanfaatkan probabilitas. Algoritma dapat digunakan selama memenuhi batasan yang diberikan pada studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash. Ada beberapa pendekatan dalam algoritma randomisasi yang di mana salah satunya adalah *hash collision. Hash collision* adalah kasus di mana ditemukan dua string dengan nilai hash yang sama. Algoritma ini diikuti dengan pemahaman teori pada subbab 3.3. (Errichto, 2019)

3.3. Birthday Attack

Birthday Paradox merupakan fenomena di mana di antara 23 orang secara acak, ada 50% peluang bahwa dua orang memiliki tanggal ulang tahun yang sama. Metode ini merupakan metode yang cocok untuk studi kasus yang akan dikerjakan yaitu mencari dua *string* dengan *hash value* yang sama, atau disebut dengan *hash collision*. *Birthday Attack* biasa diimplementasikan Algoritma *randomisasi*. Misal jumlah nilai *hash* adalah B, maka dengan *Birthday Attack* hanya memerlukan sekitar $\sqrt{B}$ percobaan (Errichto, 2019). Meskipun metode ini

memperkecil pencarian *string*, dua *string* harus memenuhi *hash value* pada setiap pasangan B_i dan M_i dari setiap K. Pada kasus terburuk, banyak kombinasi nilai *hash* adalah 256^{12} dan akan membutuhkan sekitar 256^6 percobaan untuk menemukan dua *string*. Angka percobaan tersebut masih terlalu besar karena komputer hanya bisa melakukan perulangan sekitar 10^8 sampai dengan 10^9 yang setara dengan 1 detik. Sehingga diperlukan pendekatan lain untuk memperkecil populasi nilai-nilai *hash* yang didapatkan dari *string*.

3.4. Composition Attack

Composition Attack merupakan serangan untuk mencari *hash collision* yang di mana dikhususkan untuk menyelesaikan masalah untuk mencari banyak *hash* yang sama antara dua string. *Composition Attack* dapat bekerja khusus untuk *hash* ini yaitu *Polynomial Rolling Hash. Composition Attack* bekerja untuk setiap pencarian dua string masing-masing *hash*. Langkah operasi dilakukan dengan dua tahap:

1. Menemukan 2 string yang memiliki hash yang sama pada pasangan B dan M pertama.
2. Kemudian untuk pasangan B dan M berikutnya, akan menggunakan 2 string yang ditemukan pertama menjadi alphabetnya. Misalkan 2 string tersebut adalah S dan T, maka pada B dan M yang kedua, 2 hasil string dengan hash value yang sama didapatkan dari ST, SS, SST, dst. Dengan metode ini terjamin bahwa B dan M pertama untuk hash tetap berlaku. (dacin21, 2018)

Jika ada K pasangan B_i dan M_i, maka pencarian dapat dilakukan melalui sebanyak $\sqrt{M_1} + \sqrt{M_2} + \ldots + \sqrt{M_k}$ kombinasi. Namun, setiap peralihan

proses *Birthday Attack* pada *Composition Attack* akan membuat hasil *string* menjadi sangat besar sebanyak $2log_{26}\left(\sqrt{(2\ ln2)M_1}\right) * 2log_2\left(\sqrt{(2\ ln2)M_2}\right) *$ $2log_2(\sqrt{(2\ ln2)M_3})\ldots* \ 2log_2(\sqrt{(2\ ln2)M_k})$ atau setara dengan $|S|^K$ jika menggunakan panjang string yang tetap. Jika K = 12 dan setiap M_i = 256, maka panjang *string* lebih dari 65536. Sehingga cara tidak bisa digunakan sebagai penyelesaian sendiri. (dacin21, 2018)

3.5. Strategi Penyelesaian

Birthday Attack dan *Composition Attack* memiliki kelebihan dan kekurangan masing-masing. *Birthday Attack* memiliki kekurangan dalam waktu pencarian dan kelebihan dalam penggunaan memori yaitu pada panjang *string* dan sebaliknya untuk *Composition Attack* memiliki kekurangan dalam penggunaan memori yaitu pada panjang *string* dan kelebihan dalam waktu pencarian. Oleh karena itu, kedua *attack* ini dapat dicari titik tengah sehingga seimbang dalam segi waktu dan memori. *Composition Attack* tidak harus dilakukan untuk setiap pasangan B_i dan M_i secara mandiri dan bisa dirangkap menjadi per kelompok pasangan. Per kelompok pasangan dapat disebut sebagai k_paket. Sehingga setiap transisi melalui *Composition Attack* dapat dilakukan kasus terburuk sebanyak K = 12 pasangan dibagi k_paket kali. Transisi dapat pengaruh dalam perpangkatan ukuran *string*.

3.5.1 Uji Coba pencarian k_paket sebagai strategi penyelesaian

Dengan sekali *Birthday Attack* saja, komputer hanya sanggup melakukan iterasi pencarian sebanyak 10^9 kali selama satu detik. Oleh karena itu, *Composition Attack* untuk membagi-bagi pasang B dan M untuk *Birthday Attac*k. Akan tetapi, jika tiap pasang B dan M dilakukan masing-masing *Birthday Attack*, maka akan menghasilkan panjang string yang sangat besar sehingga melampaui perbatasan soal konstrain di mana *output* harus tidak lebih dari 65536 karakter. Sehingga *Compostion Attack* perlu dibagi menjadi tiap kelompok pasangan (k_paket) agar *Birthday Attack* dapat mengurus beberapa pasangan B dan M sekaligus dan transisi *Composition Attack* yang sedikit.

K_paket yang sesuai perlu dicari supaya waktu komputasi dan ukuran *string* dapat memenuhi batasan masalah persoalan *hashing*. Berikut adalah perbandingan k_paket, per kelompok pasangan B dan M, dimulai dari kelompok 1 pasangan sampai dengan kelompok 12 pasangan pada Tabel 3.1.

Tabel 3.1 Kompleksitas waktu dan memori serta kasus terburuk pembagian pasang

Pembagian Pasang (k_paket)	Waktu	Waktu terburuk	Ukuran string	Ukuran string dengan 32
1	$O(\sqrt{M_1} + \sqrt{M_2} + \ldots + \sqrt{M_{12}})$	$O(12 * \sqrt{256})$	$\lvert S \rvert^{12}$	32^{12}

2	$O(\sqrt{M_1M_2} + \sqrt{M_3M_4} + \ldots + \sqrt{M_{11}M_{12}})$	$O(6 * \sqrt{256^2})$	$\|S\|^6$	32^6
3	$O(\sqrt{M_1M_2M_3} + \sqrt{M_4M_5M_6} + \ldots + \sqrt{M_{10}M_{11}M_{12}})$	$O(4 * \sqrt{256^3})$	$\|S\|^4$	32^4
4	$O(\sqrt{M_1M_2M_3M_4} + \sqrt{M_5M_6M_7M_8} + \sqrt{M_9M_{10}M_{11}M_{12}})$	$O(3 * \sqrt{256^4})$	$\|S\|^3$	32^3
6	$O(\sqrt{M_1M_2M_3M_4M_5M_6} + \sqrt{M_7M_8M_9M_{10}M_{11}M_{12}}$	$O(2 * \sqrt{256^6})$	$\|S\|^2$	32^2
12	$O(\sqrt{M_1M_2 \ldots M_{12}})$	$O(\sqrt{256^{12}})$	$\|S\|^1$	32^1

Panjang ukuran *string* yang dipilih adalah 32 karena jika huruf berdasarkan dari 2 pilihan dalam *Composition Attack* akan mendapatkan pilihan *string* yang cukup luas sebanyak 2^{32} atau 4294967296 kombinasi. Perlu diingat bahwa teori *Birthday Paradox* yang di mana pencarian sekitar $\sqrt{M}$ hanya merupakan estimasi atau kasus rata-rata sehingga ada suatu kasus di mana pencarian string lebih dari waktu yang pada Tabel 2.1. Oleh karena itu, berdasarkan Tabel 3.1, dilihat bahwa dengan pembagian 4 pasang B dan M memiliki waktu dan ukuran *string* yang paling seimbang dibandingkan pembagian B dan M yang lainnya. Sehingga pembagian 4 yang layak untuk dipilih. Hal ini juga terbukti dari uji kebenaran pada halaman situs penilaian daring, SPOJ 40643 AHASHREV – The Revenge Of Anti Hash.

33693988	☐	2024-10-28 17:26:01	The Revenge Of Anti Hash	runtime error (SIGKILL) edit ideone it	3.32	2061M	NCSHARP

Gambar 3.1 Uji kebenaran dengan k_paket = 1

33693963	☐	2024-10-28 17:17:55	The Revenge Of Anti Hash	runtime error (SIGKILL) edit ideone it	2.52	2061M	NCSHARP

Gambar 3.2 Uji kebenaran dengan k_paket = 2

33693983	☐	2024-10-28 17:24:05	The Revenge Of Anti Hash	wrong answer edit ideone it	0.36	66M	NCSHARP

Gambar 3.3 Uji kebenaran dengan k_paket = 3

33693964	☐	2024-10-28 17:18:20	The Revenge Of Anti Hash	**accepted** edit ideone it	2.12	67M	NCSHARP

Gambar 3.4 Uji kebenaran dengan k_paket = 4

33693961	☐	2024-10-28 17:17:03	The Revenge Of Anti Hash	runtime error (SIGABRT) edit ideone it	0.08	32M	NCSHARP

Gambar 3.5 Uji kebenaran dengan k_paket = 6

33693992	☐	2024-10-28 17:26:47	The Revenge Of Anti Hash	runtime error (SIGABRT) edit ideone it	0.08	30M	NCSHARP

Gambar 3.6 Uji kebenaran dengan k_paket = 12

Gambar 3.1 dan Gambar 3.2 menunjukkan hasil mendapatkan hasil *runtime error SIGKILL (Signal Kill).* Hal ini disebabkan dari memori yang melebihi batasan soal untuk program yaitu 1536MB dengan besar 2061MB pada k_paket = 1 dan k_paket = 2. Gambar 3.3 menunjukkan hasil *wrong answer.* Hal ini disebabkan karena panjang *string* yang didapat adalah 32^4 atau 1048576 karakter pada

k_paket = 3. Panjang ini melebihi permintaan soal yaitu tidak lebih dari 65536 karakter. Gambar 3.5 dan Gambar 3.6 menunjukkan hasil *runtime error SIGABRT (Signal Abort)*. Hal ini disebabkan program yang dijalankan terlalu lama atau percobaan yang berlebihan sehingga program harus dihentikan pada k_paket = 6 dan k_paket = 12. Gambar 3.4 menunjukkan hasil *accepted* yang menunjukkan program berjalan dengan hasil yang benar pada k_paket = 4. Dari hasil-hasil ini dapat terbukti bahwa 4 merupakan pilihan yang tepat sebagai pengelompokan sebanyak 4 pasangan B dan M sehingga dilakukan transisi sebanyak 3 kali secara maksimal.

Setiap transisi pada *Composition Attack* akan melakukan pencarian setiap 4 pasang hash. Jika dengan kasus terburuk K = 12, maka akan ada $M_1M_2M_3M_4 + M_5M_6M_7M_8 + M_9M_{10}M_{11}M_{12}$ kombinasi. Pada 4 pasangan pertama, dua *string* dibuat berdasarkan dari huruf a-z. Sedangkan pasangan seterusnya akan membuat dua *string* berdasarkan hasil dua *string* sebelumnya sebagai huruf. Panjang *string* yang dapat digunakan adalah 32 karena mengandung 2^{32} = 4294967296 pada *Composition Attack* yang sesuai dengan banyaknya *hash* kombinasi setiap 4 pasangan. Dalam hal panjang *string*, maka hasil akhir akan memiliki panjang 32^3 atau setara dengan 32768 yang memenuhi batasan panjang *string* yaitu 65536. Dengan cara ini, kompleksitas waktu didapatkan $O(\sqrt{M_1M_2M_3M_4} + \sqrt{M_5M_6M_7M_8} + \sqrt{M_9M_{10}M_{11}M_{12}})$ sebagai jumlah percobaan yang dilakukan yang berarti sekitar 3 * 256^2 = 196608

percobaan yang dilakukan. Kompleksitas waktu juga bergantung dari pembangkitan acak string untuk kapan hasil dua string ditemukan.

3.6. .NET C#

C# atau dibaca C Sharp adalah bahasa pemrograman berbasis objek. Bahasa ini populer dalam pengembangan aplikasi berbasis .NET (dotnet), yang di mana aplikasi desktop pada komputer. .NET (dotnet) adalah platform aplikasi dari Microsoft yang didesain untuk mengutamakan produktivitas, performa, keamanan, dan keandalan (Microsoft, 2024). Bahasa .NET C# sendiri memiliki suatu struktur data yaitu Dictionary<TKey, TValue>.

Dictionary<TKey, TValue> merupakan struktur data yang menyimpan *collection* berupa key dan value. Dictionary menyediakan mapping dari banyak key dengan banyak value. Kelebihan dari Dictionary <TKey, TValue> adalah saat mendapatkan value dengan menggunakan key membutuhkan waktu yang sangat cepat atau linear O(1) karena Dictionary diimplementasikan sebagai Hash Table. *Key* dan *value* dapat digunakan dengan tipe data seperti integer, *string*, dan lain-lain. Berikut adalah beberapa properti yang digunakan pada Dictionary sebagai Tabel 3.2 (Microsoft, 2024)

Tabel 3.2 Properti Dictionary<TKey, TValue> C#

Properti	**Deskripsi**
Comparer	Digunakan untuk membandingkan antar *key* pada Dictionary<TKey, TValue>
Count	Mendapatkan jumlah pasang *key/value* pada Dictionary<TKey, TValue>

Item[Tkey]	Mendapatkan *value* dari *Key* yang ditentukan
Keys	Mendapatkan kumpulan *key* dari Dictionary<TKey, TValue>
Values	Mendapatkan kumpulan *value* dari Dictionary<TKey, TValue>

Selain properti pada Tabel 3.2 ada beberapa metode yang dapat digunakan Dictionary<TKey, TValue> sebagai Tabel 3.3 (Microsoft, 2024)

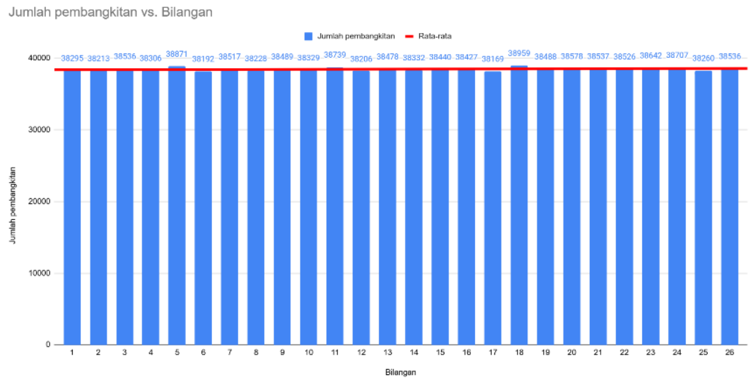
Tabel 3.3 Metode Dictionary<TKey, TValue> C#

Metode	**Deskripsi**
Add(TKey, TValue)	Menambahkan pasangan *key* dan value ke Dictionary<TKey, TValue>
Clear	Menghapus seluruh *key* dan *value* dari Dictionary<TKey, TValue>
ContainsKey(TKey)	Memeriksa apakah Dictionary<TKey, TValue> mengandung *key* yang dicari
ContainsValue(TValue)	Memeriksa apakah Dictionary<TKey, TValue> mengandung *value* yang dicari
Remove(TKey)	Menghapus *value* dari key yang ditentukan dari Dictionary<TKey, TValue>
TryGetValue(TKey, TValue)	Mendapatkan *value* dengan key yang ditentukan

Pada permasalahan studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash, Dictionary<TKey, TValue> dari .NET C# dan metode-metode pada Tabel 3.3 akan

digunakan untuk mengelola hashing dengan menyimpan hash value sebagai key dan menyimpan string sebagai value. Penggunaan struktur data Dictionary<TKey, TValue> juga dipilih karena memiliki pencarian key yang sangat efisien yaitu O(1) untuk pengecekan string dengan hash value yang sama pada setiap key yang ada pada Dictionary<TKey, TValue>.

Selain Dictionary<TKey, TValue>, diperlukan suatu *class* dan metode untuk membangkitkan bilangan acak. .NET C# telah menyediakan suatu kelas untuk membangkitkan bilangan acak yaitu kelas System.Random. gewarren dari situs Microsoft Learn (2024) menjelaskan kelas Random menggunakan *pseudo-random number generator*, algoritma di mana menghasilkan rangkaian bilangan yang memenuhi statistik tertentu. Kelas Random ini menggunakan distribusi *uniform* karena bilangan-bilangan dibangkit melalui probabilitas yang sama dari kumpulan bilangan yang terbatas. Distribusi dapat dibuktikan dengan pembangkitan bilangan antara 1-26 sebanyak 1000000 percobaan dengan gambar grafik sebagai Gambar 3.7



Gambar 3.7 Hasil distribusi pembangkitan kelas Random

Gambar 3.7 menunjukkan hasil distribusi dari pembangkitan bilangan acak antara 1 sampai dengan 26 dengan 1000000 percobaan. Sumbu horizontal menyatakan bilangan dan sumbu vertikal menyatakan jumlah pembangkitan bilangan tersebut. Setiap batang per bilangan menyatakan bahwa distribusi dilakukan secara merata untuk setiap bilangan dengan rata-rata 38462 kali dibangkitkan. Sehingga kelas random ini cocok digunakan sebagai pembangkitan nilai pada pembuatan *string* untuk persoalan *Hashing* karena distribusi dilakukan secara merata atau uniform.

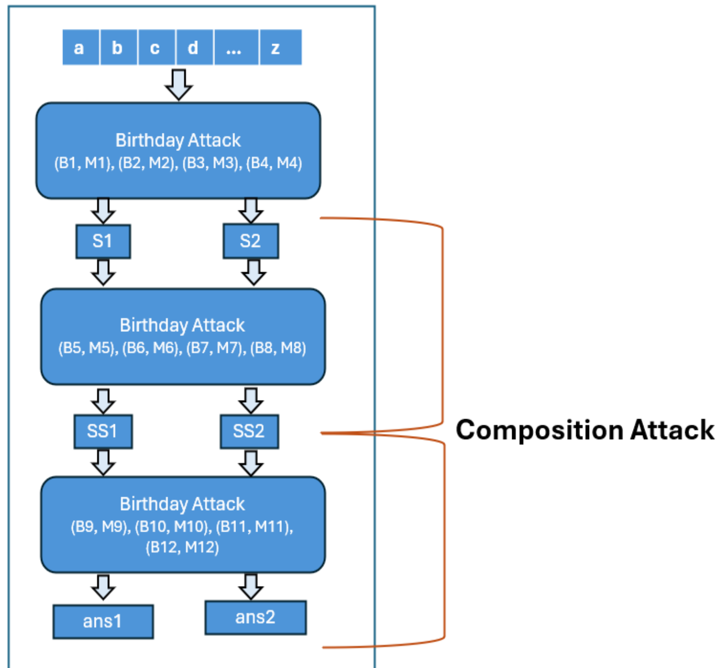
Kelas Random memiliki suatu metode yaitu Next() yang digunakan untuk mendapatkan nilai acak dengan batasan tertentu. Batasan nilai awal atau nilai akhir dapat disertakan sebagai *seed* pada parameter metode Next(). (gewarren, 2024)

[Halaman ini sengaja dikosongkan]

BAB IV
DESAIN ALGORITMA

4.1. Deskripsi Umum Permasalahan

Program dari Studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash adalah suatu program *anti-hash* yang dapat melakukan pencarian dua *string* yang memiliki nilai *hash* yang sama sesuai dengan pasangan B dan M yang diberikan. Program akan menerima masukan berupa pasangan B sebagai basis dan M sebagai modulo yang digunakan untuk fungsi *hash* dari suatu *string* sebanyak K. Kemudian program akan mengeluarkan dua buah *string* yang memiliki K nilai *hash* yang sama. Program terbagi menjadi tiga proses untuk setiap kelipatan 4 pasang B dan M sesuai dengan K yang diberikan. Pada 4 pasang pertama, program akan menjalankan *Birthday Attack* pertama terlebih dahulu untuk mencari dua *string* yang memiliki 4 nilai *hash* yang sama. Kemudian pada setiap 4 pasang berikutnya, dua *string* tersebut akan menjadi suatu alfabet sebagai bahan acak pembuat *string* yaitu *Composition Attack*. Berdasarkan Bab III Tinjauan Pustaka, algoritma yang digunakan akan mengikuti diagram alur berdasarkan Gambar 4.1



Gambar 4.1 Diagram Alur Desain Algoritma

Gambar 4.1 menunjukkan alur algoritma yang akan didesain. Desain dimulai dengan *Birthday Attack* pertama untuk membuat kedua *string* berdasarkan 4 pasang B dan M yang pertama dengan huruf ‘a’ sampai dengan ‘z’. Kemudian berlanjut pada *Composition Attack* melalui *Birthday Attack* pada tiap 4 pasang B dan M berikutnya sebanyak maksimal dua kali. Dua *string* yang didapatkan melalui *Birthday Attack* sebelumnya akan dijadikan sebagai *alphabet* atau huruf pada *Birthday Attack* berikutnya sampai menghasilkan suatu *output* dengan ans1

dan ans2 sebagai jawaban dua *string* dengan nilai *hash* yang sama dengan K pasang.

Berikut adalah kode semu fungsi utama atau main yang menunjukkan proses-proses berjalan pada Kode Semu 4.1

Kode Semu 4.1: Main

```
1   Input t
2   while t ← t – 1 > 0 do
3       Input k
4       for i = 0 to k do
5           Input b[i], m[i]
6       end for
7       Initialize k_original ← k
8       if k_original > 4 do
9           k = 4
10      end if
11      BuildStringFirst()
12      if k_original <= 4 do
13          if found do
14              Write alphabet[0] + " " + alphabet[1]
15          end if
16          Resets alphabet[0], alphabet[1], HashTable
17          found ← false
18          continue
19      end if
20      if not found
21          continue
22      end if
23      for i = 4 to k_original increment by 4 do
24          Resets HashTable
25          k ← k + 4
26          if k > k_original do
27              k ← k_original
```

```
28              BuildStringContinue(i)
29              break
30          else
31              BuildStringContinue(i)
32          end if
33      end for
34      if found do
35          Write alphabet[0] +  " " + alphabet[1]
36          Resets alphabet[0], alphabet[1], HashTable
37          found ← false
38      end if
```

Kode Semu 4.1 menjalankan program utama dengan pergabungan *Birthday Attack* dan *Composition Attack*. *Birthday Attack* Pertama akan dijalankan melalui *BuildStringFirst*() yang akan dibatasi hanya 4 pasang B dan M pertama. Kemudian *Composition Attack* akan dijalankan melalui *BuildStringContinue*() untuk setiap kelipatan 4 pasang B dan M berikutnya.

4.2. Desain Algorima

Pada program terdapat beberapa fungsi yang digunakan sebagai proses anti hash.

4.2.1 Model Operasi Birthday Attack Pertama

Operasi ini dilakukan dengan pembangkitan *string* secara acak dan akan dibandingkan nilai *hash*nya. Pada *string* pertama, *string* dibangkitkan secara acak untuk setiap hurufnya dari a sampai dengan z. Kemudian pada *string-string* berikutnya, operasi cukup melakukan pergantian satu huruf secara acak dari *string* sebelumnya karena akan menghasilkan nilai *hash* yang berbeda. Selain

itu, proses ini juga lebih cepat daripada membuat *string* dari awal lagi. Setiap *string* dan hasil *hash* setiap pasangan B_i dan M_i akan disimpan dalam tabel *hash*. Dalam C#, Dictionary digunakan sebagai tabel *hash*. Setelah membangkitkan setiap *string, string* akan dibandingkan apakah nilai *hash*nya sudah disimpan pada Dictionary sebelumnya. Jika nilai *hash* ditemukan pada Dictionary dan dua *string* tidak sama, maka dua *string* dengan *hash* yang sama berhasil ditemukan dan akan disimpan sebagai dua huruf baru untuk *Composition Attack*. Jika belum ada, maka masukan nilai *hash* dan *string* tersebut ke dalam Dictionary dan lanjutkan pencarian. Model ini digambarkan sebagai Kode Semu 4.2: BuildStringFirst

---

Kode Semu 4.2: BuildStringFirst

BuildStringFirst()*:*
1 GetRandomString(*maxLen*)

Untuk *Birthday Attack* Pertama akan membuat *string* berjumlah x dan diisi dengan parameter *maxLen* sebagai panjang *string* yang sudah ditentukan sebelumnya yang direpresentasikan melalui fungsi *GetRandomString(). GetRandomString* dituliskan sebagai Kode Semu 4.3: GetRandomString

---

Kode Semu 4.3: GetRandomString

GetRandomString(int len*):*
1 **Initialization**: *strTemp*, *key*, *finalHash*[]
2 strTemp ← GenerateRandomString(int *len*)
3 **for** *c* = 0 **to** *k*-1 **do**
4 *finalHash*[*c*] ← GetHash(*strTemp*, *b*[*c*], *m*[*c*])

```
5           key.append(finalHash[c])
6       end for
7       HashTable[key] ← strTemp
8       for i = 1 to x-1 do
9           key ← “”
10          pos ← random(0 to len)
11          old ← strTemp[pos]
12          strTemp[pos] ← random(‘a’ to ‘z’)
13          for c = 0 to k-1 do
14              finalHash[c] = GetHashReplace(len, pos, old,
            temp[pos], finalHash[c])
15              key.append(finalHash[c])
16          end for
17          if HashTable.ContainsKey(key) then
18                  if HashTable[key] == strTemp continue
19                  alphabet[0] ← HashTable[key]
20                  alphabet[1] ← strTemp
21                  found ← true
22                  break
23          end if
24          HashTable[key] ← strTemp
25      end for
```

Kode Semu 4.3: GetRandomString membuat *string* sebanyak x dan sekaligus mencari apakah dua *string* dengan *hash* yang sama ditemukan. Sesuai dengan subbab 3.4, Kode Semu 4.3: GetRandomString hanya dipakai untuk 4 pasang B_i dan M_i pertama karena pembentukan string dilakukan acak dari karakter a sampai z. Selain itu, Kode Semu 4.3: GetRandomString membuat *string* pertama dengan membuat *string* baru dari awal. Kemudian pada *string* berikutnya, akan memperbarui *string* sebelumnya dengan cara mengubah posisi secara acak dengan karakter secara acak pula. Sehingga program tidak

perlu membuat setiap *string* dari awal dengan iterasi sebanyak variabel *len* atau panjang *string* yang diatur.

Pada Kode Semu 4.3: GetRandomString, juga terlihat bahwa untuk mendapatkan *hash string* berikutnya, program menggunakan *GetHashReplace*. Yang membedakan *GetHashReplace* dengan *GetHash* pada Kode Semu 3.1: GetHash adalah bahwa *GetHashReplace* hanya menggantikan nilai *hash* dari karakter posisi tertentu. *GetHashReplace* ini memerlukan parameter posisi karakter yang diganti, karakter lama, karakter baru, dan nilai hash sebelumnya. Pendapatan fungsi ini ditemukan melalui fungsi *GetHash* dalam bentuk model matematika.

$$\text{GetHash}() = (((\text{Hash} * B + x_0) * B + x_1) * B + x_2) \ldots * B + x_n) \% M \quad (4.1)$$

Pada fungsi (4.1), x_i merupakan *chr* – ‘a’ + 1. Selanjutnya karena *Hash* awal diinisialisasi 0, maka dapat dijabarkan sehingga ditemukan persamaan (4.2)

$$\text{GetHash}() = (x_0 * B^{N-1} + x_1 * B^{N-2} + x_2 * B^{N-3} + \ldots + x_{N-1} * B^0) \% M \quad (4.2)$$

x_i menunjukkan karakter pada posisi ke-i, sehingga ketika ada perubahan huruf, hanya $x_i * B^{N-1-i}$ yang diubah. Sehingga Persamaan (4.2) mewujudkan fungsi *GetHashReplace()* direpresentasikan dengan Kode Semu 4.4: GetHashReplace

---

Kode Semu 4.4: GetHashReplace

GetHashReplace(*len*, *pos*, *oldChr*, *newChr*, *oldHash*, *B*, *M*):

1 *count* ← *len* - 1 - *pos*

2 *powerBase* ← *ModEx*(*B*, *count*, *M*)

```
3 oldSub ← (oldChr - 'a' + 1) * powerBase % M
4 newSub ← (newChr - 'a' + 1) * powerBase % M
5 newHash ← (oldHash - oldSub + newSub) % M
6 if newHash < 0 then
7    newHash ← newHash + M
8 end if
9 return newHash
```

Kode Semu 4.4: GetHashReplace berdasarkan dari persamaan (4.2). Variabel *powerBase* merupakan hasil perpangkatan B^{N-1-i} dengan menggunakan eksponensial modular karena hasil bisa sangat besar sehingga perlu modulo diperlukan. Dalam proses mendapatkan nilai *hash* baru, perlu di-modulokan juga menggunakan M agar hasil tidak *overload*. Jika hasil hash baru terkadang menjadi kurang dari 0, maka tambahkan dengan M.

Untuk setiap *string* yang telah terbuat akan dimasukkan ke Dictionary sebagai tabel *hash* yang menyimpan nilai *hash* sebagai *key* dan *string* sebagai nilai *value*. Setiap pemasukan *string* akan diperiksa apakah nilai *hash* sudah tersimpan pada Dictionary tersebut atau belum. Ketika sudah tersimpan sebelumnya dan *string value* tidak sama, maka ditemukan dua *string* dengan nilai *hash* yang sama dan didaftarkan sebagai *alphabet*[0] dan *alphabet*[1] untuk sebagai jawaban *output* atau huruf pada *Composition Attack*.

4.2.2 Model Composition Attack

Algoritma ini hanya berlaku jika ada lebih dari 4 pasangan B dan M dalam masukkan program. Keluaran dari Model Operasi *Birthday Attack* Pertama adalah hasil dua *string* yang tersimpan pada *alphabet*[0] dan *alphabet*[1]. Kedua nilai variabel ini akan digunakan sebagai *alphabet* yang akan diacak untuk *Composition Attack* sesuai dengan subbab 3.4. Sebelum memulai *Composition Attack*, nilai *hash* tiap *alphabet* perlu dihitung terlebih dahulu untuk masing-masing pasangan B dan M yang baru yang akan dikalkulasikan untuk kombinasi alfabet. Model ini dipanggil melalui fungsi *BuildStringContinue* sebagai Kode Semu 4.5

Kode Semu 4.5: BuildStringContinue

```
BuildStringContinue(start)
1   found ← false
2   for ki = start to k - 1 do
3       tempHashContinue[ki, 0] ←
        GetHash(alphabet[0], b[ki], m[ki]
4       tempHashContinue[ki. 1] ←
        GetHash(alphabet[1], b[ki], m[ki])
5   end for
6   GetStringNext(start)
```

Kode Semu 4.5: *BuildStringContinue* akan menerima suatu variabel yaitu *start*, sebagai mulainya pasangan B dan M ke berapa. Kemudian *BuildStringContinue* akan menghitung nilai *hash* dari setiap *alphabet* dan dengan setiap pasang B dan M sebanyak empat, sesuai k yang diatur dari fungsi Main().

Setelah itu, program akan memanggil fungsi *GetStringNext* untuk memulai pembuatan *string* secara acak berdasarkan dua *alphabet* sebagai Kode Semu 4.6: GetStringNext.

Kode Semu 4.6: GetStringNext

```
GetStringNext(start):
1    Initialization: fHA[], temp, sA ← "", key ← 0
2    lenAlphabet ← alphabet[0].Length
3    for ki = start to k-1 do
4        fHA[ki] ← 0
5    end for
6    for i = 0 to x-1 do
7        key ← 0
8        if i == 0 then
9            for al = 0 to maxLen-1 do
10               ch ← random(0, 1)
11               sA.append(ch)
12               idx ← sA[al]
13               for ki = start to k-1 do
14                   fHA[ki] ← GetTotalHash(lenAlphabet,
         fHA[ki], tempHashContinue[ki, idx], b[ki], m[ki])
15               end for
16            end for
17            for ki = start to k-1 do
18                key.append(fHA[ki])
19            end for
20           HashTable[key] ← sA
21       else
22           pos ← random(0 to len)
23           old ← strTemp[pos]
24           oldIdx ← old
25           if old == '0' then
```

```
26                 temp[pos] ← '1'
27             else temp[pos] ← '0'
28             for ki = start to k-1 do
29                 fHA[ki]  ← GetTotalHashReplace(maxLen,
           lenAlphabet, pos, tempHashContinue[ki, oldIdx],
           tempHashContinue[ki, idx], fHA[ki], b[ki], m[ki])

30             end for
31             for ki = start to k-1 do
32                 key.append(fHA[ki])
33             end for
34             if HashTable.ContainsKey(key) then
35                 if HashTable[key] == sA continue
36                 tA ← ConvertAlphabet(HashTable[key])
37                 tB ← ConvertAlphabet(sA)
38                 alphabet[0] ← tA
39                 alphabet[1] ← tB
40                 found ← true
41                break
42             end if
43            HashTable[key] ← sA
44        end if
```

Kode Semu 4.6: GetStringNext menjalankan pembuatan *string-string* dengan menggunakan jawaban *string* sebelumnya sebagai *alphabet*[0] dan *alphabet*[1] sehingga pengacakan huruf *string* tidak berdasarkan 'a' sampai dengan 'z', melainkan menggunakan *alphabet*[0] dan *alphabet*[1] sebagai huruf-hurufnya. *Alphabet*[0] dan *alphabet*[1] akan dikonversi pada jawaban *string* bagian akhir. *String-string* baru ini menggunakan *alphabet*[0] dan *alphabet*[1] yang masing-masing memiliki nilai *hash* sendiri yang sudah dihitung sebelumnya sehingga untuk

mendapatkan nilai *hash* dari *string* baru ini diperlukan suatu metode lain selain *GetHash* yang menghitung nilai *hash* masing-masing huruf, yaitu *GetTotalHash*. *GetTotalHash* merupakan fungsi menjumlahkan antara dua hash dari suatu *string*. Setiap huruf yang didapatkan melalui pengambilan *alphabet*[0] dan *alphabet*[1] secara acak, nilai *hash* tersebut akan ditambahkan pada nilai total *hash* sehingga mendapatkan nilai *hash* asli dari *string* yang sudah jadi didapatkan. Operasi ini dapat dituliskan sebagai Kode Semu 4.7: GetTotalHash.

Kode Semu 4.7: GetTotalHash

GetTotalHash(*len2, hash1, hash2, B, M*):
1 *powerBase* ← ModEx(*B, len2, M*)
2 *totalHash* ← (*hash1* * *powerBase* % *M* + *hash2*) %*M*
3 **return** *totalHash*

Kode Semu 4.7: GetTotalHash melakukan penjumlahan antara dua buah *hash* dengan perpindahan hash1 sebanyak panjang *string* dari *hash2 (len2)*. Perpindahan dilakukan dengan memanfaatkan perpangkatan dengan modular eksponen. Setelah didapatkan *string* pertama, langkah berikutnya adalah pembuatan *string-string* berikutnya dengan menggantikan salah satu huruf atau *alphabet* secara acak pada *string* sebelumnya. Langkah ini mirip dengan pada model *Birthday Attack* Pertama. Namun, perbedaan terletak pada pergantian suatu huruf. Pada model *Birthday Attack* Pertama, program mengganti suatu huruf dengan salah satu huruf dari ‘a’ sampai dengan ‘z’ secara acak. Sedangkan, pada model *Composition Attack* atau pada

Kode Semu 4.5: GetStringNext, program menggantikan huruf langsung dengan huruf yang lainnya karena pergantian dilakukan antara dua huruf saja yaitu *alphabet*[0] dan *alphabet*[1]. Aturan pergantian huruf adalah:

- Jika huruf sebelumnya adalah *alphabet*[0], maka digantikan *alphabet*[1] secara langsung.
- Jika huruf sebelumnya adalah *alphabet*[1], maka digantikan *alphabet*[0] secara langsung.

Dengan cara ini, program akan melakukan pencarian *string* dengan nilai *hash* yang sama lebih cepat karena tidak ada duplikasi *string* yang terjadi saat pembuatannya karena pengacakan bisa kembali lagi ke *alphabet* sebelumnya. Kemudian untuk pergantian nilai *hash* pada *alphabet* yang diganti dapat dilakukan dengan fungsi operasi *GetTotalHashReplace*. Yang membedakan dengan *GetHashReplace* adalah bahwa yang digantikan nilai *hash* dari suatu *string alphabet* bukan karakter. Sehingga operasi ini dapat dibuatkan melalui Kode Semu 4.8: GetTotalHashReplace.

Kode Semu 4.8: GetTotalHashReplace

GetTotalHashReplace(*len, len2, pos, oldHashChar, newHashChar, oldHash, B, M*):

1 *count* ← *len* - 1 - *pos*
2 *powerBase* ← *ModEx*(*B*, *count***len2*, *M*)
3 *oldSub* ← *oldHashChar* * *powerBase* % *M*
4 *newSub* ← *newHashChar* * *powerBase* % *M*
5 *newHash* ← (*oldHash* - *oldSub* + *newSub*) % *M*
6 **if** *newHash* < 0 **then**

7 *newHash* ← *newHash* + *M*
8 **end if**
9 **return** *newHash*

Kode Semu 4.8: GetTotalHashReplace mirip dengan Kode Semu 4.3: *GetHashReplace* dengan perbedaan yang terletak perkalian pada *oldSub* dan *newSub* yang langsung dilakukan dengan variabel *powerBase*. *powerBase* ini merupakan *modulo* eksponen dengan *count* * *len2*, karena memerlukan panjang total pergeseran antara jumlah huruf dari *string* asli dan jumlah huruf dari suatu *alphabet*.

Setiap *string* akan disimpan pada Dictionary sebagai tabel *hash* yang baru dengan nilai *hash* sebenarnya sebagai *key* dan *string* sebagai *value*. Jika nilai *hash* sudah tersimpan pada Dictionary tersebut sebelumnya dan kedua *string* tersebut berbeda, maka ditemukan dua *string* dengan nilai *hash* yang sama. Kedua *string* akan menggantikan *alphabet*[0] dan *alphabet*[1] sebagai jawaban *output* atau untuk proses *Composition Attack* berikutnya. Model ini akan dijalankan jika memiliki lebih dari 8 pasangan B dan M sehingga akan mendapatkan jawaban *output* yang sebenarnya.

BAB V
IMPLEMENTASI SISTEM

Bab ini akan menjelaskan terkait implementasi sistem berdasarkan dasar teori dengan desain algoritma yang telah dirancang pada bab sebelumnya.

5.1. Lingkungan Implementasi

Lingkungan implementasi yang digunakan adalah sebuah PC dengan spesifikasi perangkat keras dan perangkat lunak seperti pada Tabel 5.1

Tabel 5.1 Spesifikasi Lingkungan Implementasi Program

No	Jenis Perangkat	Spesifikasi
1	Perangkat Keras	• CPU: AMD Ryzen 7 6800H 3.20 GHz • Random Access Memory: 16 GB
2	Perangkat Lunak	• Windows 11 Home • Visual Studio Code • Menggunakan bahasa .NET C#

5.2. Penggunaan Fast Input Output untuk optimasi performa

Untuk dapat mengoptimasi performa program, diperlukan suatu fungsi *input* dan *output* yang cepat. Di sini akan menggunakan *Fast Input Output* untuk bahasa .NET C# sebagai Kode Sumber 5.1

```
public static class InputOutput
{
    private const byte _minus = (byte)'-';
    private const byte _zero = (byte)'0';
    private static System.IO.BufferedStream _readStream,
_writeStream;
    private const int BUFF_Size = 1 << 22;
    private static int _inBuffSize = BUFF_Size, _outBuffSize =
BUFF_Size;
    public static bool ThrowErrorOnEof { get; set; } = false;

    static InputOutput()
    {
        _readStream = new
System.IO.BufferedStream(System.Console.OpenStandardInpu
t(), _inBuffSize);
        _writeStream = new
System.IO.BufferedStream(System.Console.OpenStandardOut
put(), _outBuffSize);
    }

    public static int ReadInt()
    {
        byte readByte;
        while ((readByte = GetByte()) < _minus) ;
```

```
        var neg = false;
        if (readByte == _minus)
        {
            neg = true;
            readByte = GetByte();
        }
        var m = readByte - _zero;
        while (true)
        {
            readByte = GetByte();
            if (readByte < _zero) break;
            m = m * 10 + (readByte - _zero);
        }
        return neg ? -m : m;
    }

    [System.Runtime.CompilerServices.MethodImpl(System.Ru
ntime.CompilerServices.MethodImplOptions.AggressiveInlini
ng)]
    private static byte GetByte()
    {
        var b = _readStream.ReadByte();
        if (b == -1)
        {
            if (ThrowErrorOnEof) throw new
System.Exception("End Of Input");
            else return 0;
        }
        return (byte)b;
    }

    [System.Runtime.CompilerServices.MethodImpl(System.Ru
ntime.CompilerServices.MethodImplOptions.AggressiveInlini
ng)]
    private static void WriteByte(byte b)
```

```
    {
        _writeStream.WriteByte(b);
    }

    public static void WriteToBuffer(string s, bool newLine =
false)
    {
        foreach (var b in
System.Text.Encoding.ASCII.GetBytes(s)) WriteByte(b);
        if (newLine) WriteByte(10);
    }

    public static void WriteLineToBuffer(string s) =>
WriteToBuffer(s, true);

    public static void Flush()
    {
        _writeStream.Flush();
        ThrowErrorOnEof = false;
    }
}
```

Kode Sumber 5.1 *Fast Input Output* untuk optimasi performa

Kode Sumber 5.1 memanfaatkan *BufferedStream* dalam membaca dan menulis data secara cepat yang di mana juga berguna untuk mengurus banyak masukan dan keluaran. *ReadInt*() berfungsi untuk membaca atau menerima masukan input integer ke dalam *buffer* program dengan memproses tiap *byte* dari *integer* yang telah dimasukkan dengan fungsi *GetByte*() ke dalam *_readStream*. *WriteLineToBuffer*(*string s*) merupakan fungsi untuk menuliskan atau mengeluarkan suatu *string* dalam *buffer*. Tiap huruf akan diproses dengan tiap *byte* dan baris baru melalui *WriteToBuffer* dan *WriteByte* ke

dalam *buffer _writeStream. AggressiveInlining* juga digunakan untuk mempercepat penulisan *byte*. Terakhir, *Flush* digunakan untuk mengeluarkan *buffer* yang berisi keluaran yang sudah ditulis ke dalam terminal.

5.3. Variabel Global

Variabel global digunakan untuk memudahkan dalam pengaksesan data pada tiap fungsi yang ada dalam program. Implementasi variabel global dapat dilihat pada Kode Sumber 5.2

```
    const int x = 1000000000;
    static int div = 4;
    static int k;
    static int[] b = new int[13];
    static int[] m = new int[13];
    static int maxLen = 24;
    static bool found = false;
    static int[,] tempHashContinue = new int[13,2];
    static string[] alphabet = new string[2];
    static Dictionary<uint, string> hashStr = new Dictionary<uint, 
string>();

    static Random rand = new Random();
```

Kode Sumber 5.2 Variabel Global

Untuk penjelasan dari setiap variabel global dinyatakan dalam bentuk Tabel 5.2.

Tabel 5.2 Penjelasan Variabel Global

No.	Variabel	Keterangan
1	const int x	Banyak percobaan pembuatan *string* yang dilakukan untuk mendapatkan hasil
2	static int div	Pembagian kelompok pasangan B dan M (k_paket) sesuai pada subbab 3.5.1
3	static int k	Banyak pasangan B dan M yang dimasukkan dalam soal
4	static int[] b	Nilai basis yang dimasukkan tiap pasang untuk fungsi *hash*
5	static int[] m	Nilai *modulo* yang dimasukkan tiap pasang untuk fungsi *hash*
6	static int maxLen	Panjang *string* yang diatur
7	static bool found	Indikator apakah hasil sudah ditemukan atau belum
8	static int[,] tempHashContinue	Menyimpan nilai *hash* sementara setiap *string* sebagai *alphabet* dalam proses *Composition Attack*
9	static string[] alphabet	Menyimpan dua *string* sebagai *alphabet* dalam proses *Composition Attack*
10	static Dictionary<uint, string> hashStr	Suatu tabel *hash* / Dictionary untuk menyimpan string beserta nilai *hash*nya. Hash disimpan pada *key* (*uint*) dan *string* disimpan dalam *value* (*string*)
11	static Random rand	Kelas fungsi random yang akan digunakan

Variabel-variabel ini akan digunakan untuk fungsi-fungsi berikutnya yang akan digunakan.

5.4. Fungsi ModEx

Fungsi *ModEx* merupakan fungsi modular eksponen yang digunakan untuk operasi perpangkatan dengan suatu variabel modulo. Modulo digunakan agar hasil tidak melampaui batas modulo yang diberikan dan agar hasil tidak terjadi *overflow*. Fungsi ini akan digunakan terutama pada fungsi *GetHashReplace* dan *GetTotalHashReplace*. Implementasi fungsi dapat direpresentasikan pada Kode Sumber 5.3

```
static int ModEx(int b, int e, int m)
  {
    if (e == 0) return 1;
    if (b == 0 || e == 0) return 0;
    int r = 1;
    while (e > 0)
    {
      if ((e & 1) == 1) r = (r * b) % m;
      e >>= 1;
      b = (b * b) % m;
    }
    return r;
  }
```

Kode Sumber 5.3 Fungsi *ModEx*

Fungsi *ModEx* akan melakukan iterasi perkalian setiap kuadrat dari bilangan b berdasarkan eksponen yang

diberikan yaitu e sekaligus menjaga nilai hasil dengan suatu modulo m.

5.5. Fungsi GetHash

Fungsi *GetHash* merupakan fungsi untuk mendapatkan nilai *hash* dari suatu *string* yang diberikan beserta dengan bilangan basis B dan bilangan modulo M. Fungsi *GetHash* ini sudah disediakan dalam deskripsi permasalahan soal pada subbab 3.1. Fungsi ini hanya digunakan untuk model *Birthday Attack* Pertama seperti desain pada subbab 4.2.1. Implementasi dalam bahasa .NET C# dapat dilihat pada Kode Sumber 5.4.

```
static int GetHash(string str, int B, int M)
{
   int hash = 0;
   foreach (char chr in str)
      hash = (hash * B + chr - 'a' + 1) % M;
   return hash;
}
```

Kode Sumber 5.4 Fungsi *GetHash*

5.6. Fungsi GetHashReplace

Fungsi *GetHashReplac*e merupakan fungsi untuk mendapatkan nilai *hash* dengan pergantian suatu huruf pada model *Birthday Attack* Pertama sesuai yang dijelaskan pada subbab 4.2.1. Implementasi fungsi dapat direpresentasikan sebagai Kode Sumber 5.5.

```
static int GetHashReplace(int len, int pos, char oldChr, char newChr, int oldHash, int B, int M)
{
    int count = len - 1 - pos;
    int powerBase = ModEx(B, count, M);
    int oldSub = (oldChr - 'a' + 1) * powerBase % M;
    int newSub = (newChr - 'a' + 1) * powerBase % M;

    int newHash = (oldHash - oldSub + newSub) % M;
    if (newHash < 0)
    {
        newHash += M;
    }
    return newHash;
}
```

Kode Sumber 5.5 Fungsi *GetHashReplace*

5.7. Fungsi GetTotalHash

Fungsi *GetTotalHash* merupakan fungsi untuk mendapatkan hasil penjumlahan hasil *hash* dari kedua *string* untuk model *Composition Attack* sesuai dengan penjelasan pada subbab 4.2.2. Implementasi fungsi dapat direpresentasikan sebagai Kode Sumber 5.6.

```
static int GetTotalHash(int len2, int hash1, int hash2, int B, int M) {
    int powerBase = ModEx(B, len2, M);
    int totalHash = (hash1 * powerBase%M + hash2)%M;
    return totalHash;
}
```

Kode Sumber 5.6 Fungsi *GetTotalHash*

5.8. Fungsi GetTotalHashReplace

Fungsi *GetTotalHashReplace* merupakan fungsi untuk mendapatkan nilai *hash* dengan pergantian *string alphabet* pada model *Composition Attack* sesuai yang dijelaskan pada subbab 4.2.2. Implementasi fungsi dapat direpresentasikan sebagai Kode Sumber 5.7.

```
static int GetTotalHashReplace(int len, int len2, int pos, int oldHashChar, int newHashChar, int oldHash, int B, int M) {
    int count = len - 1 - pos;
    int powerBase = ModEx(B, count*len2, M);
    int oldSub = oldHashChar * powerBase % M;
    int newSub = newHashChar * powerBase % M;
    int newHash = (oldHash - oldSub + newSub)%M;

    if (newHash < 0) newHash += M;
    return newHash;
}
```

Kode Sumber 5.7 Fungsi GetTotalHashReplace

5.9. Fungsi GetRandomString

Fungsi *GetRandomString* merupakan fungsi yang akan digunakan sebagai model *Birthday Attack Pertama*. Fungsi akan membuat *string* berjumlah x secara acak dan sekaligus memeriksa apakah menemukan dua *string* dengan nilai *hash* yang sama berdasarkan maksimal 4 pasang B dan M yang pertama melalui Dictionary *hashStr*. Pada *string* pertama, akan dibuat antara 26 huruf antara ‘a’ sampai dengan ‘z’ secara acak. Pada *string-string* berikutnya, akan dilakukan pergantian huruf secara acak

dengan posisi yang acak. Fungsi ini merupakan hasil desain algoritma yang sesuai pada subbab 4.2.1. Implementasi fungsi dapat direpresentasikan sebagai Kode Sumber 5.8.

```
static void GetRandomString(int len)
  {
      char[] temp = new char[len];
      int[] finalHash = new int[4];
      string strTemp;
      for (int i = 0; i < len; i++)
      {
        temp[i] = (char)(rand.Next(26) + 97);
      }
      strTemp = new string(temp);
      uint key = 0;
      for (int c = 0; c < k; c++) {
        finalHash[c] = (int)GetHash(strTemp, b[c], m[c]);
        key |= (uint)(finalHash[c] << 8*c);
      }

      hashStr[key] = strTemp;

      for (int i = 1; i < x; i++)
      {
        key = 0;
        temp = strTemp.ToCharArray();
        int pos = rand.Next(len);
        char old = temp[pos];
        temp[pos] = (char)(rand.Next(26) + 97);
        strTemp = new string(temp);
        for (int c = 0; c < k; c++) {
            finalHash[c] = (int)GetHashReplace(len, pos, old,
temp[pos], finalHash[c], b[c], m[c]);
```

```
            key |= (uint)(finalHash[c] << 8*c);
        }

        string val = "";

        if (hashStr.TryGetValue(key, out val)) {
            if (val == strTemp) continue;
            alphabet[0] = val;
            alphabet[1] = strTemp;
            found = true;
            break;
        }
        hashStr[key] = strTemp;
    }

}
```

Kode Sumber 5.8 Fungsi *GetRandomString*

Pada akhir fungsi, dua hasil *string* akan disimpan dalam *alphabet*[0] dan *alphabet*[1].

5.10. Fungsi BuildStringFirst

Fungsi *BuildStringFirst* merupakan fungsi utama dari model *Birthday Attack* yang mendaftarkan panjang *string* dan dipanggil pada fungsi utama. Implementasi fungsi dapat direpresentasikan sebagai Kode Sumber 5.9.

```
static void BuildStringFirst()
{
    GetRandomString(maxLen);
}
```

Kode Sumber 5.9 Fungsi *BuildStringFirst*

5.11. Fungsi ConvertAlphabet

Fungsi *ConvertAlphabet* merupakan fungsi yang digunakan mentransformasi suatu *alphabet* (*string* dari hasil *Birthday Attack*) yang direpresentasikan dengan '0' dan '1' menjadi *string* yang sebenarnya. Implementasi fungsi dapat direpresentasikan sebagai Kode Sumber 5.10.

```
static string ConvertAlphabet(char[] a) {
   string ret = "";
   for (int i = 0; i < a.Length; i++) {
      int id = a[i] - 48;
      ret += alphabet[id];
   }
   return ret;
}
```

Kode Sumber 5.10 Fungsi *ConvertAlphabet*

5.12. Fungsi GetStringNext

Fungsi *GetStringNext* merupakan fungsi *Birthday Attack* pada model *Composition Attack*. Fungsi ini akan membuat *string* berdasarkan dua *string* hasil sebelumnya sebagai huruf (*alphabet*) berjumlah x secara acak. Pada *string* pertama, akan dibuat antara 2 huruf (*alphabet*) secara acak. Pada *string-string* berikutnya, akan dilakukan pergantian huruf dengan yang lainnya dengan posisi yang acak. Fungsi sekaligus mencari dua *string* dengan nilai *hash* pada tiap 4 pasang B dan M yang ditentukan. Kemudian hasil akan diubah menjadi *string* yang sebenarnya dengan fungsi *ConvertAlphabet* dan menggantikannya sebagai *alphabet*[0] dan *alphabet*[1] yang baru. Fungsi ini merupakan hasil desain algoritma

yang sesuai pada subbab 4.2.2. Implementasi fungsi dapat direpresentasikan sebagai Kode Sumber 5.11.

```
static void GetStringNext(int start) {
        int lenAlphabet = alphabet[0].Length;
        int []fHA = new int[13];
        char[] temp = new char[maxLen];
        for (int ki = start; ki < k; ki++) {
            fHA[ki] = 0;
        }
        string sA = "";
        uint key;
        for (int i = 0; i < x; i++) {
            key = 0;

          if (i == 0) {
                for (int al = 0; al < maxLen; al++) {
                    char ch = (char)((char)rand.Next(2)+'0');
                    sA += ch;
                    int idx = sA[al]-'0';
                    for (int ki = start; ki < k; ki++) {
                         fHA[ki] = GetTotalHash(lenAlphabet, fHA[ki],
tempHashContinue[ki, idx], b[ki], m[ki]);
                    }
                }

                int c = 0;

                for (int ki = start; ki < k; ki++) {
                    key |= (uint)(fHA[ki] << 8 * c);
                    c++;
                }
                hashStr[key] = sA;
            }
```

```
else {
    temp = sA.ToCharArray();
    int pos = rand.Next(maxLen);
    char old = temp[pos];
    int oldIdx = old - '0';
    if (old == '0') temp[pos] = '1';
    else temp[pos] = '0';
    sA = new string(temp);
    int idx = sA[pos] - '0';
    for (int ki = start; ki < k; ki++) {
            fHA[ki] = GetTotalHashReplace(maxLen,
lenAlphabet, pos, tempHashContinue[ki, oldIdx],
tempHashContinue[ki, idx], (int)fHA[ki], b[ki], m[ki]);
    }

    int c = 0;

    for (int ki = start; ki < k; ki++) {
        key |= (uint)(fHA[ki] << 8 * c);
        c++;
    }

    string val = "";
    if (hashStr.TryGetValue(key, out val)) {
        if (val == sA) continue;
        string tA = ConvertAlphabet(val.ToCharArray());
        string tB = ConvertAlphabet(sA.ToCharArray());
        alphabet[0] = tA;
        alphabet[1] = tB;
        found = true;
        break;
    }
    hashStr[key] = sA;
}
```

```
    }
}
```

Kode Sumber 5.11 Fungsi *GetStringNext*

5.13. Fungsi BuildStringContinue

Fungsi *BuildStringContinue* merupakan fungsi utama pada model *Composition* Attack untuk memanggil fungsi *GetStringNext*. Sebelum memasuki fungsi *GetStringNext*, fungsi ini melakukan pencarian nilai hash untuk setiap alphabet yang sebelumnya dimasukkan dua string dalam variabel *tempHashContinue* untuk tiap pasang B dan M. Desain fungsi *BuildStringContinue* sesuai pada alur dari subbab 4.2.2. Implementasi fungsi *BuildStringContinue* dapat direpresentasikan sebagai Kode Sumber 5.12.

```
static void BuildStringContinue(int start)
{
    found = false;
    for (int ki = start; ki < k; ki++) {
        tempHashContinue[ki,0] = GetHash(alphabet[0], b[ki],
m[ki]);
        tempHashContinue[ki,1] = GetHash(alphabet[1], b[ki],
m[ki]);
    }
    GetStringNext(start);
}
```

Kode Sumber 5.12 Fungsi *BuildStringContinue*

5.14. Fungsi Main

Fungsi *Main* merupakan fungsi utama berjalannya program yang menerima masukkan, memanggil fungsi *BuildStringFirst* dan fungsi *BuildStringContinue*, dan menghasilkan keluaran dua string yang diminta. Untuk proses membaca masukkan dan menulis keluaran menggunakan *Fast Input Output* sesuai pada subbab 5.2 dan akan melakukan *Flush* untuk mengeluarkan seluruh keluaran pada *terminal*. Fungsi ini juga melibatkan pembersihan ulang Dictionary *hashStr* pada setiap transisi antara *BuildStringFirst* dan *BuildStringContinue*. Selain itu, variabel *found* akan diisi kembali sebagai false dan variabel *alphabet*[0] dan *alphabet*[1] dikosongkan untuk setiap kasus uji coba. Fungsi ini merupakan implementasi desain dari subbab 4.1. Implementasi fungsi *main* dapat direpresentasikan sebagai Kode Sumber 5.13.

```
static void Main()
  {
    int t = InputOutput.ReadInt();
    while (t-- > 0)
    {
      k = InputOutput.ReadInt();
      int ka;
      for (int i = 0; i < k; i++)
      {
        b[i] = InputOutput.ReadInt();
        m[i] = InputOutput.ReadInt();
      }
      ka = k;
      if (ka > div) k = div;
      BuildStringFirst();
      if (ka <= div) {
        if (found) {
```

```
                InputOutput.WriteLineToBuffer(alphabet[0] +  " "
+ alphabet[1]);
            }
            alphabet[0] = "";
            alphabet[1] = "";
            hashStr.Clear();
            found = false;
            continue;
         }
         if (!found) continue;
         for (int i = div; i < ka; i+=div) {
            hashStr.Clear();
            k += div;
            if (k > ka) {
               k = ka;
               BuildStringContinue(i);
               break;
            } else BuildStringContinue(i);
         }
         if (found) {
             InputOutput.WriteLineToBuffer(alphabet[0] +  " " +
alphabet[1]);
            alphabet[0] = "";
            alphabet[1] = "";
            found = false;
            hashStr.Clear();
         }
      }
      InputOutput.Flush();
   }
```

Kode Sumber 5.13 Fungsi *Main*

BAB VI
PENGUJIAN DAN EVALUASI

Bab ini menjelaskan tahapan uji coba dan evaluasi dari hasil implementasi program yang sudah dilakukan dalam penyelesaian persoalan *hashing*: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash.

6.1. Lingkungan Uji Coba

Lingkungan uji coba yang digunakan untuk uji coba kebenaran pada situs penilaian daring SPOJ sebagai Tabel 6.1.

Tabel 6.1 Spesifikasi Lingkungan Uji Coba

No	Jenis Perangkat	Spesifikasi
1	Perangkat Keras	• CPU: AMD Ryzen 7 6800H 3.20 GHz • Random Access Memory: 16 GB
2	Perangkat Lunak	• Windows 11 Home • Visual Studio Code • Menggunakan bahasa .NET C#

6.2. Skenario Uji Coba

Pada subbab ini akan dijelaskan skenario uji coba yang akan digunakan dalam menguji implementasi penyelesaian persoalan *hashing*: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash yang telah dibuat sebelumya. Skenario uji coba akan dilakukan dengan menggunakan situs penilaian *Sphere Online Judge* (SPOJ) untuk menguji kebenaran dan kinerja program implementasi yang telah dibuat.

6.3. Uji Coba Kebenaran

Uji coba kebenaran dilakukan dengan mengirimkan kode sumber implementasi ke situs penilaian *Sphere Online Judge* (SPOJ) . Setelah melakukan setiap pengiriman kode sumber implementasi, SPOJ akan memberikan umpan balik mengenai kebenaran kode sumber dalam penyelesaian masalah tersebut. Hasil uji coba pada situs SPOJ ditunjukkan pada Gambar 6.1 dan hasil peringkat situs SPOJ pada Gambar 6.2.

Gambar 6.1 Hasil uji coba pada situs SPOJ

RANK	DATE	USER	RESULT	TIME	MEM	LANG
1	2023-05-03 23:16:57	Viplov Jain	accepted	0.03	5.3M	CPP14
2	2024-10-08 17:39:41	jakubciecielag	accepted	0.31	29M	NCSHARP
3	2024-06-09 16:48:41	yovnsh	accepted	0.35	28M	NCSHARP
4	2024-06-13 22:28:19	SiogunJH	accepted	0.37	29M	NCSHARP
5	2024-06-20 16:03:41	kkm	accepted	0.39	29M	NCSHARP
6	2021-09-03 07:22:26	[Rampage] Blue.Mary	accepted	0.43	6.6M	CPP
7	2024-06-14 10:12:13	prz3m3k86	accepted	0.46	29M	NCSHARP
8	2024-06-14 16:23:19	dawid14723	accepted	0.53	29M	NCSHARP
9	2022-11-06 13:17:29	peterklimenko	accepted	0.74	27M	PYTHON3
10	2024-10-03 17:39:07	014_Alexander Weynard S.	accepted	1.47	62M	NCSHARP
11	2024-10-07 22:39:20	Rully Soelaiman	accepted	1.58	75M	NCSHARP
12	2024-10-13 07:18:10	afiq	accepted	1.90	59M	NCSHARP
13	2023-12-18 09:08:16	Oleg	accepted	6.91	29M	CPP14

Gambar 6.2 Peringkat hasil pada situs SPOJ

Dari hasil uji coba yang telah dilakukan, kode sumber program mendapatkan umpan balik berupa *Accepted* yang berarti solusi kode sumber yang dikirimkan sudah benar dan dapat diterima. Waktu yang dibutuhkan adalah 1.47 detik dan memori yang dibutuhkan adalah 62MB. Selain itu, dilakukan pengujian serupa sebanyak 20 kali untuk melihat variasi waktu dan memori yang dibutuhkan program. Hasil uji coba sebanyak 20 kali dinyatakan dalam Tabel 6.2 dan Gambar 6.3.

Tabel 6.2 Hasil uji coba 20 kali pada situs SPOJ

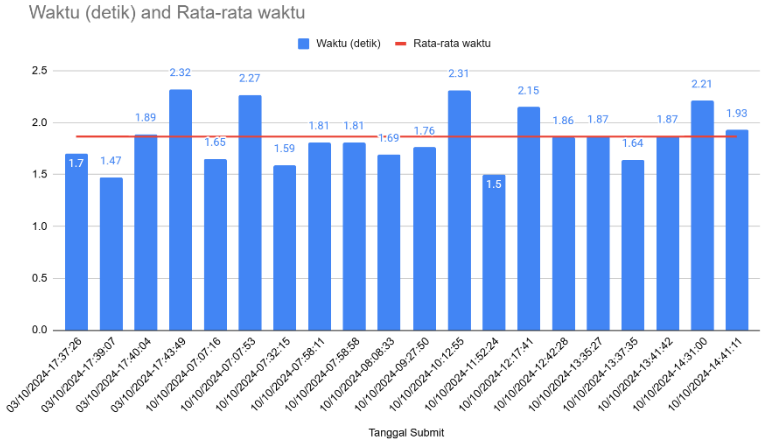
No.	Hasil	Waktu (detik)	Memori (MB)
1	Accepted	1.7	71
2	Accepted	1.47	62
3	Accepted	1.89	69
4	Accepted	2.32	90
5	Accepted	1.65	60
6	Accepted	2.27	79
7	Accepted	1.59	78

8	Accepted	1.81	61
9	Accepted	1.81	68
10	Accepted	1.69	68
11	Accepted	1.76	64
12	Accepted	2.31	67
13	Accepted	1.5	59
14	Accepted	2.15	79
15	Accepted	1.86	75
16	Accepted	1.87	81
17	Accepted	1.64	69
18	Accepted	1.87	74
19	Accepted	2.21	75
20	Accepted	1.93	64

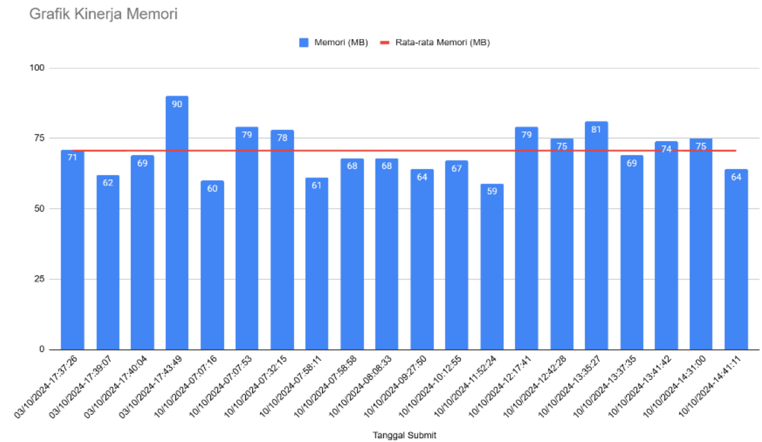
33604675	2024-10-10 07:58:58	The Revenge Of Anti Hash	accepted edit ideone it	1.81	68M	NCSHARP
33604671	2024-10-10 07:58:11	The Revenge Of Anti Hash	accepted edit ideone it	1.81	61M	NCSHARP
33604567	2024-10-10 07:32:15	The Revenge Of Anti Hash	accepted edit ideone it	1.59	78M	NCSHARP
33604428	2024-10-10 07:07:53	The Revenge Of Anti Hash	accepted edit ideone it	2.27	79M	NCSHARP
33604427	2024-10-10 07:07:16	The Revenge Of Anti Hash	accepted edit ideone it	1.65	60M	NCSHARP
33575515	2024-10-03 17:43:49	The Revenge Of Anti Hash	accepted edit ideone it	2.32	90M	NCSHARP
33575494	2024-10-03 17:40:04	The Revenge Of Anti Hash	accepted edit ideone it	1.89	69M	NCSHARP
33575488	2024-10-03 17:39:07	The Revenge Of Anti Hash	accepted edit ideone it	1.47	62M	NCSHARP
33575476	2024-10-03 17:37:26	The Revenge Of Anti Hash	accepted edit ideone it	1.70	71M	NCSHARP
33573362	2024-10-02 14:37:26	The Revenge Of Anti Hash	accepted edit ideone it	1.67	60M	NCSHARP
33607388	2024-10-10 14:31:00	The Revenge Of Anti Hash	accepted edit ideone it	2.21	75M	NCSHARP
33607189	2024-10-10 13:41:42	The Revenge Of Anti Hash	accepted edit ideone it	1.87	74M	NCSHARP
33607166	2024-10-10 13:37:35	The Revenge Of Anti Hash	accepted edit ideone it	1.64	69M	NCSHARP
33607148	2024-10-10 13:35:27	The Revenge Of Anti Hash	accepted edit ideone it	1.87	81M	NCSHARP
33606883	2024-10-10 12:42:28	The Revenge Of Anti Hash	accepted edit ideone it	1.86	75M	NCSHARP
33606715	2024-10-10 12:17:41	The Revenge Of Anti Hash	accepted edit ideone it	2.15	79M	NCSHARP
33606560	2024-10-10 11:52:24	The Revenge Of Anti Hash	accepted edit ideone it	1.50	59M	NCSHARP
33605898	2024-10-10 10:12:55	The Revenge Of Anti Hash	accepted edit ideone it	2.31	67M	NCSHARP
33605232	2024-10-10 09:27:50	The Revenge Of Anti Hash	accepted edit ideone it	1.76	64M	NCSHARP
33604704	2024-10-10 08:08:33	The Revenge Of Anti Hash	accepted edit ideone it	1.69	68M	NCSHARP

Gambar 6.3 Hasil 20 kali uji coba pada situs SPOJ

Berdasarkan hasil 20 kali uji coba, digambarkan sebuah grafik untuk waktu dan memori sebagai Gambar 6.4 untuk waktu dan Gambar 6.5 untuk memori.



Gambar 6.4 Grafik hasil waktu uji coba pada situs SPOJ sebanyak 20 kali



Gambar 6.5 Grafik hasil memori uji coba pada situs SPOJ sebanyak 20 kali

Berdasarkan dari hasil grafik kinerja waktu Gambar 6.4, kode sumber program mendapatkan kinerja waktu dengan rata-rata 1.865 detik dan dengan waktu minimum 1.47 detik. Kemudian pada hasil grafik kinerja memori Gambar 6.5, kode sumber program mendapatkan kinerja memori dengan rata-rata 70.65 MB dan dengan memori minimum 59 MB.

6.4. Evaluasi Pengujian

Hasil pengujian dilakukan dengan mengirimkan kode sumber implementasi program ke situs penilaian SPOJ. Pengujian dilakukan sebanyak 20 kali untuk melihat variasi waktu dan memori yang dibutuhkan oleh program. Dari hasil pengiriman kode sumber sebanyak 20 kali didapatkan hasil waktu dengan rata-rata 1.865 detik dan dengan waktu minimum 1.47 detik. Selain waktu, didapatkan juga memori dengan rata-rata 70.65 MB dan dengan memori minimum 59 MB. Semua hasil pengujian mendapatkan umpan balik dari situs SPOJ yaitu *Accepted*, yang menandakan bahwa kode sumber program dapat diterima dan benar.

Pengujian ini menunjukkan bahwa implementasi program pada kode sumber terbukti berhasil menyelesaikan persoalan *hashing*: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash dengan cukup efisien. Program juga menunjukkan bahwa waktu dan memori berjalan di bawah batasan persoalan. Pada grafik gambar 6.3 dan 6.4 menunjukkan bahwa hasil waktu dan memori yang diberikan bervariatif. Hal ini disebabkan karena algoritma yang digunakan adalah pembangkitan nilai secara acak sehingga berapa lama waktu yang dibutuhkan agar mendapatkan hasil keluaran yang benar

bervariatif. Namun, pengiriman program mendapatkan waktu rata-rata 1.865 detik dan memori rata-rata 70.65 MB dalam sebanyak 20 kali pengujian serta mendapatkan hasil *Accepted*. Oleh karena itu, desain algoritma yang digunakan dalam implementasi sudah tepat dan optimal dalam menyelesaikan persoalan *Hashing*: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash.

BAB VII
KESIMPULAN DAN SARAN

7.1. Kesimpulan

Berdasarkan hasil uji coba yang telah dilakukan dengan implementasi desain Algoritma Randomisasi pada penyelesaian kode sumber pada persoalan *hashing*: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash untuk mendapatkan dua *string* dengan nilai *hash* yang sama, didapat beberapa kesimpulan terkait Kerja Praktik ini sebagai berikut:

a. Desain algoritma pada persoalan hashing pada studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash dapat diselesaikan dengan algoritma randomisasi dengan menggunakan teorema *Birthday Attack* dan *Composition Attack* dengan kompleksitas waktu $O(\sqrt{M_1 M_2 M_3 M_4} + \sqrt{M_5 M_6 M_7 M_8} + \sqrt{M_9 M_{10} M_{11} M_{12}})$.
b. Implementasi algoritma randomisasi untuk menyelesaikan persoalan hashing studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash dengan desain algoritma yang telah. Desain dimulai dengan pembuatan string secara acak dengan menggunakan model metode *Birthday Attack* dan *Composition Attack*. Selain itu, dibuat juga beberapa fungsi tambahan seperti *GetHashReplace, GetTotalHash, dan GetTotalHashReplace* sebagai fungsi pendukung dalam implementasi program.
c. Hasil kinerja randomisasi untuk menyelesaikan persoalan hashing studi kasus SPOJ 40643 AHASHREV – The Revenge Of Anti Hash diraih dengan waktu minimum 1.47 detik dengan 62 MB pada pengiriman yang sama sebagai hasil terbaik.

Hasil kinerja juga menunjukan waktu yang dibutuhkan dengan rata-rata 1.865 detik dan memori yang dibutuhkan dengan rata-rata 70.65 MB serta memori minimum 59 MB.

7.2. Saran

Setelah mendapatkan kesimpulan dari hasil pengujian, berikut adalah saran dalam desain algoritma randomisasi desain Algoritma Randomisasi pada penyelesaian kode sumber pada persoalan *hashing*: Studi Kasus SPOJ 40643 AHASHREV - The Revenge Of Anti Hash:

a. Implementasi kode sumber program yang telah dibuat diharapkan dapat ditingkatkan efisiensinya dalam segi waktu dan memori.

DAFTAR PUSTAKA

dacin21. (2018). *On the mathematics behind rolling hashes and anti-hash tests*. Diambil dari Codeforces: https://codeforces.com/blog/entry/60442

Errichto. (2019). *Algoritma randomisasis lecture, part 1 & 2*. Diambil dari Codeforces: https://codeforces.com/blog/entry/71097

gewarren. (2024, 8 1). *System.Random class*. Diambil dari Microsoft Learn: https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-random

Microsoft. (2024). *Dictionary<TKey,TValue> Class*. Diambil dari Microsoft Learn: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-8.0

Microsoft. (2024, 10 1). *Introduction to .NET*. Diambil dari Microsoft Learn: https://learn.microsoft.com/en-us/dotnet/core/introduction?WT.mc_id=dotnet-35129-website

[Halaman ini sengaja dikosongkan]

BIODATA PENULIS

Nama	: Alexander Weynard Samsico
Tempat, Tanggal Lahir	: Jakarta, 12 Maret 2003
Jenis Kelamin	: Laki-laki
Telepon	: 082233441578
Email	: samsicoweynard@gmail.com

AKADEMIS

Kuliah	: Departemen Teknik Informatika – FTEIC , ITS
Angkatan	: 2021
Semester	: 7 (Tujuh)