DAT470/DIT065 Assignment 3

Xinyan Liu
liuxinya@chalmers.se

Leah Wanja Ndirangu
leahw@chalmers.se

2025-05-11

Problem 1

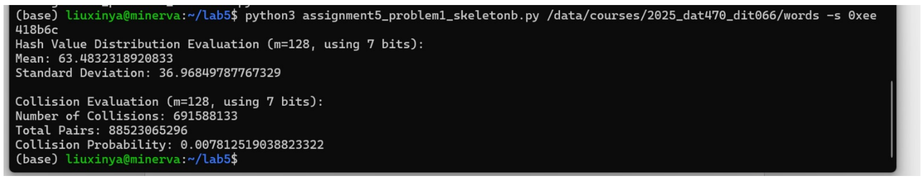
(a) Using assignment5_problem1_skeleton.py as starting point, imple- ment the Murmur3_32 algorithm, following the pseudocode on Wikipedia. Table 1 lists some correct hash values which you can use to evaluate the correctness of your implementation. Note that Python strings consist of Unicode characters, but the hashing algorithm assumes 8-bit characters. To this end, you should encode all strings into UTF-8 byte sequences before hashing (this should be done by the hash function itself, the inputs are assumed to be Unicode strings). Also, you must use bitwise operations on integers. For performance reasons, you must not convert integers into strings of binary characters or arrays or lists of ones and zeros, but you must operate on 32-bit words.

Below is the implementation of the `map` function in Python:

```
#!/usr/bin/env python3

import argparse,struct

def rol32(x,k):
    """Auxiliary function (left rotation for 32-bit words)
        """
    return ((x << k) | (x >> (32-k))) & 0xffffffff

def murmur3_32(key, seed):
    """Computes the 32-bit murmur3 hash"""
    k1 = 0xcc9e2d51
    k2 = 0x1b873593
    hash1 =seed & 0xffffffff
    key = bytearray(key.encode('utf-8'))
    length_key = len(key)
    number_of_blocks = length_key //4

    for start_block in range(0,number_of_blocks *4,4):
        key1 = struct.unpack_from('<I',key,start_block)[0]
        key1 = (key1 * k1) & 0xffffffff
        key1 = rol32(key1,15)
        key1 = (key1 * k2) & 0xffffffff

        hash1 ^=key1
        hash1 = rol32(hash1,13)

        hash1 =(hash1 * 5 + 0xe6546b64) & 0xffffffff

    tail = key[number_of_blocks *4:]
    key1 = 0
    if len(tail) >=3:
        key1 ^= tail[2] << 16

    if len(tail) >= 2:
        key1 ^= tail[1] << 8
    if len(tail) >= 1:
        key1 ^= tail[0]
        key1 = (key1 * k1) & 0xffffffff
        key1 = rol32(key1, 15)
        key1 = (key1 * k2) & 0xffffffff
        hash1 ^= key1

    # Finalization
    hash1 ^= length_key
    hash1 ^= (hash1 >> 16)
    hash1 = (hash1 * 0x85ebca6b) & 0xffffffff
    hash1 ^= (hash1 >> 13)
    hash1 = (hash1 * 0xc2b2ae35) & 0xffffffff
    hash1 ^= (hash1 >> 16)

    return hash1

def auto_int(x):
    """Auxiliary function to help convert e.g. hex integers
       """
    return int(x,0)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Computes MurMurHash3 for the keys.'
    )
    parser.add_argument('key',nargs='*',help='key(s) to be
       hashed',type=str)
    parser.add_argument('-s','--seed',type=auto_int,default
       =0,help='seed value')
    args = parser.parse_args()
    keys = []
    with open(args.key[0], 'r', encoding='utf-8') as f:
        for line in f:
            key = line.strip()
            if key:
                keys.append(key)
    seed = args.seed
    for key in keys:
        h = murmur3_32(key,seed)
        print(f'{h:#010x}\t{key}')
```

(b) Evaluate the quality of your hash function. It is probably infeasible to extensively evaluate the full 32-bit value space, so instead, we'll fix a pa- rameter m that is a power of two, and evaluate the log2 m least significant bits of the hash value. This is because this is exactly what we are going to do to select the registers in the second problem

The following graph 1 shows arithmetic mean and the standard deviation of the hash values, as well as collision probability and the number of collisions



```
(base) liuxinya@minerva:~/lab5$ python3 assignment5_problem1_skeletonb.py /data/courses/2025_dat470_dit066/words -s 0xee
418b6c
Hash Value Distribution Evaluation (m=128, using 7 bits):
Mean: 63.4832318920833
Standard Deviation: 36.96849787767329

Collision Evaluation (m=128, using 7 bits):
Number of Collisions: 691588133
Total Pairs: 88523065296
Collision Probability: 0.007812519038823322
(base) liuxinya@minerva:~/lab5$
```

Figure 1: Results for b.

Here are the code

```
#!/usr/bin/env python3
import datetime

import matplotlib.pyplot as plt
import numpy as np
from collections import Counter
import argparse,struct
import math
def rol32(x,k):
    """Auxiliary function (left rotation for 32-bit words)
        """
    return ((x << k) | (x >> (32-k))) & 0xffffffff

def murmur3_32(key, seed):
    """Computes the 32-bit murmur3 hash"""
    k1 = 0xcc9e2d51
    k2 = 0x1b873593
    hash1 =seed & 0xffffffff
    key = bytearray(key.encode('utf-8'))
    length_key = len(key)
    number_of_blocks = length_key //4

    for start_block in range(0,number_of_blocks *4,4):
        key1 = struct.unpack_from('<I',key,start_block)[0]
        key1 = (key1 * k1) & 0xffffffff
        key1 = rol32(key1,15)
        key1 = (key1 * k2) & 0xffffffff

        hash1 ^=key1
        hash1 = rol32(hash1,13)

        hash1 =(hash1 * 5 + 0xe6546b64) & 0xffffffff

    tail = key[number_of_blocks *4:]
    key1 = 0
    if len(tail) >=3:
        key1 ^= tail[2] << 16

    if len(tail) >= 2:
        key1 ^= tail[1] << 8
    if len(tail) >= 1:
        key1 ^= tail[0]
        key1 = (key1 * k1) & 0xffffffff
        key1 = rol32(key1, 15)
        key1 = (key1 * k2) & 0xffffffff
        hash1 ^= key1

    # Finalization
    hash1 ^= length_key
    hash1 ^= (hash1 >> 16)
    hash1 = (hash1 * 0x85ebca6b) & 0xffffffff
    hash1 ^= (hash1 >> 13)
    hash1 = (hash1 * 0xc2b2ae35) & 0xffffffff
    hash1 ^= (hash1 >> 16)

    return hash1

def auto_int(x):
    """Auxiliary function to help convert e.g. hex integers
       """
    return int(x,0)

def evaluate_hash_distribution(hashes, m):
    mask = m - 1
    reduced_hashes = [h & mask for h in hashes]

    freq = Counter(reduced_hashes)
    values = list(freq.keys())
    counts = list(freq.values())

    plt.bar(values, counts)
    plt.xlabel('Hash Value (Least Significant 7 Bits)')
    plt.ylabel('Frequency')
    plt.title('Hash Value Distribution Histogram (Least
        Significant 7 Bits)')
    current_date = datetime.datetime.now()
    plt.savefig(f"{current_date}.png")
    mean_val = np.mean(reduced_hashes)
    std_val = np.std(reduced_hashes)
    return mean_val, std_val, freq

def evaluate_collisions(hashes, m):
    mask = m - 1
    reduced_hashes = [h & mask for h in hashes]
    freq = Counter(reduced_hashes)

    collisions = sum(math.comb(count, 2) for count in freq.
        values() if count > 1)
    total_pairs = math.comb(len(hashes), 2)
    collision_prob = collisions / total_pairs
    return collisions, total_pairs, collision_prob


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Computes MurMurHash3 for the keys.'
    )
    parser.add_argument('key',nargs='*',help='key(s) to be
        hashed',type=str)
    parser.add_argument('-s','--seed',type=auto_int,default
        =0,help='seed value')
    parser.add_argument('-m', type=int, default=128)

    args = parser.parse_args()
    keys = []
    with open(args.key[0], 'r', encoding='utf-8') as f:
        for line in f:
            key = line.strip()
            if key:
                keys.append(key)

    seed = args.seed
    for key in keys:
        h = murmur3_32(key,seed)
        # print(f'{h:#010x}\t{key}')

    hashes = [murmur3_32(key, seed) for key in keys]

    mean_val, std_val, freq = evaluate_hash_distribution(
        hashes, args.m)
    print(f"Hash Value Distribution Evaluation (m={args.m},
        using 7 bits):")
    print(f"Mean: {mean_val}")
    print(f"Standard Deviation: {std_val}")

    collisions, total_pair, collision_prob =
        evaluate_collisions(hashes, args.m)
    print(f"\nCollision Evaluation (m={args.m}, using 7 bits
        ):")
    print(f"Number of Collisions: {collisions}")
    print(f"Total Pairs: {total_pair}")
    print(f"Collision Probability: {collision_prob}")
```

The figure 2 below shows a frequency distribution of the distribution of hash values

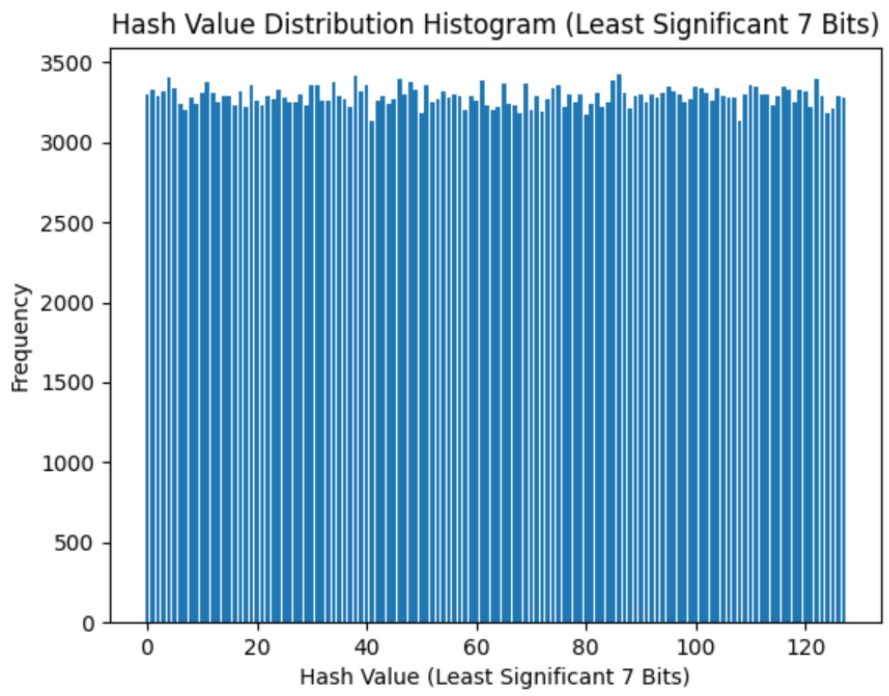


Figure 2: frequency distribution of the distribution of hash values

(c) Based on your evaluation, would you say the hash function appears to be a good one or not? Why or why not?

It is a good one. Hash values should be evenly distributed across the entire possible output range, without any obvious clustering phenomena.

Problem 2

(a) Using assignment5_problem2_skeleton.py as your starting point, im- plement the function compute_jr. The function takes as input a string x, a seed value S, and log2 m, the base-2 logarithm of the number of registers, that is, how many bits are required to represent m. It then computes y ← h(x, S) using murmur3_32 from Problem 1, sets j to be the log2 m least significant bits of y, and computes r ← (y) defined as the 1-based bit position of the leftmost (most significant) 1-bit in y. That is, for example, (cb65a3e7) = 1, (6d93092b) = 2, (35e38af 6) = 3, and so on. We define (0) = 0. The return value of the function should be (j, r). Table 2 lists some example outputs with m = 128.

Here is our code

```
#!/usr/bin/env python3

import argparse
import sys
import argparse,struct

def rol32(x,k):
    """Auxiliary function (left rotation for 32-bit words)
        """
    return ((x << k) | (x >> (32-k))) & 0xffffffff

def murmur3_32(key, seed):
    """Computes the 32-bit murmur3 hash"""
    # use the implementation from Problem 1
    k1 = 0xcc9e2d51
    k2 = 0x1b873593
    hash1 = seed & 0xffffffff
    key = bytearray(key.encode('utf-8'))
    length_key = len(key)
    number_of_blocks = length_key // 4

    for start_block in range(0, number_of_blocks * 4, 4):
        key1 = struct.unpack_from('<I', key, start_block)[0]
        key1 = (key1 * k1) & 0xffffffff
        key1 = rol32(key1, 15)
        key1 = (key1 * k2) & 0xffffffff

        hash1 ^= key1
        hash1 = rol32(hash1, 13)

        hash1 = (hash1 * 5 + 0xe6546b64) & 0xffffffff

    tail = key[number_of_blocks * 4:]
    key1 = 0
    if len(tail) >= 3:
        key1 ^= tail[2] << 16

    if len(tail) >= 2:
        key1 ^= tail[1] << 8
    if len(tail) >= 1:
        key1 ^= tail[0]
        key1 = (key1 * k1) & 0xffffffff
        key1 = rol32(key1, 15)
        key1 = (key1 * k2) & 0xffffffff
        hash1 ^= key1

    # Finalization
    hash1 ^= length_key
    hash1 ^= (hash1 >> 16)
    hash1 = (hash1 * 0x85ebca6b) & 0xffffffff
    hash1 ^= (hash1 >> 13)
    hash1 = (hash1 * 0xc2b2ae35) & 0xffffffff
    hash1 ^= (hash1 >> 16)

    return hash1

def auto_int(x):
    """Auxiliary function to help convert e.g. hex integers
       """
    return int(x,0)

def dlog2(n):
    """Auxiliary function to compute discrete base2
       logarithm"""
    print(f"dlog2 {n}")
    return n.bit_length() - 1

def rho(n):
    """Given a 32-bit number n, return the 1-based position
       of the first
    1-bit"""
    print(f"hex {h:08x}")
    if n==0:
        return 0
    binary_str = bin(n)[2:].zfill(32)
    binary_list = list(binary_str)
    print(binary_list)
    for i in range(len(binary_list)):
        if binary_list[i] == '1':
            return i+1


def compute_jr(key,seed,log2m):
    """hash the string key with murmur3_32, using the given
       seed
    then take the **least significant** log2(m) bits as j
    then compute the rho value **from the left**

    E.g., if m = 1024 and we compute hash value 0x70ffec73
    or 0b01110000111111111110110001110011
    then j = 0b0001110011 = 115
         r = 2
         since the 2nd digit of 0111000011111111111011 is
             the first 1

    Return a tuple (j,r) of integers
    """
    h = murmur3_32(key,seed)
    print(f'h {h:08x}')
    j = ~(0xffffffff << log2m) & h
    r = rho(h)
    return j, r


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Computes (j,r) pairs for input integers
            .'
    )
    parser.add_argument('key',nargs='*',help='key(s) to be
        hashed',type=str)
    parser.add_argument('-s','--seed',type=auto_int,default
        =0,help='seed value')
    parser.add_argument('-m','--num-registers',type=int,
        default=2**58,
                        help=('Number of registers (must
                             be a power of two)'))
    args = parser.parse_args()

    seed = args.seed
    m = args.num_registers
    if m <= 0 or (m&(m-1)) != 0:
        sys.stderr.write(f'{sys.argv[0]}: m must be a
            positive power of 2\n')
        quit(1)

    log2m = dlog2(m)

    # keys = []
    # with open(args.key[0], 'r', encoding='utf-8') as f:
    #     for line in f:
    #         key = line.strip()
    #         if key:
    #             keys.append(key)

    for key in args.key:
        h = murmur3_32(key,seed)

        j, r = compute_jr(key,seed,log2m)

        print(f'{key}\t{j}\t{r}')
```