1 Implementeringsprojekt

I denne implementeringsopgave skal I implementere to forskellige algoritmer til at analysere en stor strøm af data. Mere præcists vil I blive udsat for en strøm af par $(x_1, d_1), \ldots, (x_n, d_n)$ hvor hver x_i kommer fra et univers $U = [2^{64}]$ af 64-bit nøgler og hvor $d_i \in \mathbb{Z}$ er et heltal. Bemærk at d_i kan være både positivt og negativt og at den samme nøgle x kan forekomme flere gange i strømmen. For en nøgle x definerer vi $s(x) = \sum_{i \in [n]: x_i = x} d_i$. Vi kan tænke på $s(x)$ som antallet af gange x forekommer i strømmen, men vægtet med d_i'erne. I denne opgave vil vi interessere os for summen $S = \sum_{x \in U} s(x)^2$, som er et klassisk mål for hvor meget varians der er i datastrømmen.

I første del af opgaven skal I implementere og bruge en hashtabel med chaining, der kan bruges til at beregne S præcist. I anden del skal I implementere og bruge den såkaldte Count-Sketch algoritme, til at estimere kvadratsummen S mere effektivt og med meget mindre plads. I skal sammenligne de to algoritmers pladsforbrug og køretid. Anden del vil blive offentliggjort efter forelæsningen om second moment estimation. Det vil være muligt (og klogt) at have første del implementeret og testet inden. Implementeringsprojektet skal afleveres den 3. juni.

Bemærkning om programmeringssprog. Som udgangspunkt skal implementeringsprojektet laves i enten C# eller F#. Hvis I ønsker at programmere i andre sprog skal I hurtigts muligt kontakte jeres instruktor som vil undersøge mulighederne for dette. Det vil være en fordel at arbejde i F# eller C# da I vil få brug for at teste jeres kode på faktiske strømme af data og funktionerne til at generere disse strømme er skrevet i disse sprog.

I får brug for følgende ved implementering af jeres hashfunktioner:
Right-Shift: For at lave et right-shift a x med l bits bruges `x>>l` i C# og `x>>>l` i F#.
Bitwise AND: For at lave det bitvise AND mellem to strenge x og y bruges `x&y` i C# og `x&&&y` i F#.
Store multiplikationer: I får desuden brug for at multiplicere store heltal (i multiply-shift, 64-bit multiplikation og i multiply-mod-prime, 128-bit multiplication). For at forsimple opgaven er I velkomne til at benytte typerne `bigint` i F# og `BigInteger` i C# der kan rumme så store heltal og som understøtter multiplikation og de bitvise operationer ovenfor.

1.1 Del 1: Hashing med chaining

I første del, beskrevet her, skal I implementere og bruge en almindelig hash tabel med chaining til at bestemme kvadratsummen S for forskellige datastrømme. I vil arbejde med 64-bit nøgler og skal implementere både multiply-mod-prime med 64-bit nøgler og $p = 2^{89} - 1$ (se hashingnoterne `https://arxiv.org/pdf/1504.06804.pdf` sektion 2.2.1) og multiply-shift (sektion 3 i hashingnoterne). I vil blive udsat for strømme der er lige lange (altså med samme n), men med meget forskellige antal forskellige nøgler. Da pladsforbruget ved hashing med chaining afhænger af antallet af forskellige nøgler vil de derfor kræve meget forskelligt lagerforbrug. I skal rapportere køretiderne og se hvordan de afhænger af lagerforbruget. Specielt burde det blive klart, hvor meget langsommere algoritmen kører når lagerforbruget er stort. I skal også se, hvordan valget af hashfunktion påvirker køretiden.

Opgave 1. Implementering af hashfunktioner. Målet med denne opgave er at implementere multiply-shift og multiply-mod-prime hashing. For at gøre tingene simplere og koden hurtigere vil I kun skulle implementere hashfunktionerne hvor størrelsen af billedmængden er en toerpotens. (Man kan godt implementere hashfunktionerne effektivt selv om størrelsen af billedmængden ikke er en toerpotens, men det vil vi ikke komme ind på i dette kursus.)

(a) Implementer multiply-shift hashing som er parametriseret af a og l, hvor a er et ulige 64-bit tal, og hvor l er et positivt heltal mindre end 64. Mere præcist skal hashfunk-

tionen være

$$h(x) = (a \cdot x) \texttt{>>} (64 - l),$$

hvor a er et tilfældigt ulige 64-bit heltal. I kan bruge `www.random.org/bytes` til at generere a, f.eks. ved at generere 8 bytes og sætte sidste bit i sidste byte til at være 1.

(b) Implementer multiply-mod-prime hashing, hvor $p = 2^{89} - 1$, og som er parametriseret af a, b og l, hvor a og b er heltal skarpt mindre end p og hvor l er et positivt heltal mindre end 64. Mere præcist skal hashfunktionen være

$$h(x) = ((a \cdot x + b) \mod p) \mod 2^l,$$

hvor a og b er uafhængige og uniformt tilfældige i $[p] = \{0, 1, \ldots, p-1\}$. I skal bruge observationerne i Exercise 2.7 og 2.8 i hashingnoterne til effektivt at implementere udregningen af $a \cdot x + b \mod p$. Igen kan I benytte `www.random.org/bytes` til at generere a og b ved at generere to tilfældige bitstrenge af længde 89 og smide resultatet væk i det *ekstremt* usandsynlige tilfælde at en af dem består af lutter 1-taller.

(c) I skal nu teste køretiderne af de to hashfunktioner I har implementeret. Brug generatoren i 1.2 til at generere en strøm, $x_1, \ldots, x_n$, af nøgler og udskriv summen af deres hashværdier $\sum_{i=1}^{n} h(x_i)$, både for jeres multiply-mod-prime og multiply-shift hashfunktion. Denne sum er ikke interessant i sig selv men nogle optimeringskompilere smider overflødig kode væk, så I skal udskrive summen for at sikre at jeres computer beregner alle hashværdierne. Vælg selv n tilstrækkeligt stor til at I kan se forskellene, men lille nok til at det ikke tager for lang tid at køre koden. Rapporter køretiderne og kommenter på de forskelle I ser.

Opgave 2. Implementering af hash tabel med chaining. Vi vil igen simplificere problemet og antage at størrelsen af hashtabellen er en toerpotens. Målet er at implementere en hashtabel ved brug af chaining, som er parametriseret ved en hashfunktion, h, og et positivt heltal l, hvor billedmængden for h er $[2^l]$. Hashtabellen skal håndtere følgende operationer.

(a) `get`(x): Skal returnere den værdi, der tilhører nøglen x. Hvis x ikke er i tabellen skal der returneres 0.

(b) `set`(x, v): Skal sætte nøglen x til at have værdien v. Hvis x ikke allerede er i tabellen så tilføjes den til tabellen med værdien v.

(c) `increment`(x, d): Skal lægge d til værdien tilhørende x. Hvis x ikke er i tabellen, skal x tilføjes til tabellen med værdien d.

Opgave 3. Udregning af kvadratsummer. Implementer en funktion der givet en strøm af par $(x_1, d_1), \ldots, (x_n, d_n)$ beregner kvadratsummen $S = \sum_{x \in U} s(x)^2$. I skal bruge hashtabellen som I har implementeret i opgave 2 til at gemme værdierne for hvert x i strømmen.

I skal køre funktionen med forskellige strømme og se hvordan køretiden afhænger af antallet af forskellige nøgler blandt $x_1, ..., x_n$. I skal også teste hvor stor en effekt det har om jeres hashtabel bruger multiply-shift eller multiply-mod-prime hashing.

Strømmene kan I generere med koden i 1.2 hvor I selv kan specificere n samt antallet af forskellige nøgler, 2^l. Med 2^l forskellige nøgler skal I sætte størrelsen af billedmængden for jeres hashfunktioner til 2^l. Kør jeres kode med større og større l men med det samme n (bemærk at dette kræver at I vælger n tilstrækkelig stor til at starte med), og fortsæt indtil koden enten tager for lang tid at køre eller I løber tør for plads. Præsenter resultaterne for hhv. multiply-shift og multiply-mod-prime og de forskellige værdier af l i en tabel. I skulle gerne observere at jo større l er, jo mindre betydning har det hvilken af de to hashfunktioner I bruger. Kommenter på dette.

1.2 Strømme til at teste jeres datastrukturer

I kan med fordel bruge følgende funktion, skrevet i F# og C#, til at generere strømme til at teste jeres datastrukturer på. Funktionen tager et heltal n, som er antallet af elementer i strømmen, og et heltal l, som beskriver hvor mange forskellige nøgler der skal være i strømmen. Mere specifikt så vil der være 2^l forskellige nøgler i strømmen og strømmen vil have længde n.

Følgende kode genererer en strøm i F#:

```
let createStream (n : int) (l : int) : seq<uint64 * int> =
    seq {
        // We generate a random uint64 number.
        let rnd = System.Random()
        let mutable a = 0UL
        let b : byte [] = Array.zeroCreate 8
        rnd.NextBytes(b)
        let mutable x : uint64 = 0UL
        for i = 0 to 7 do
            a <- (a <<< 8) + uint64(b.[i])

        // We demand that our random number has 30 zeros on
           the least
        // significant bits and then a one.
        a <- (a ||| ((1UL <<< 31) - 1UL)) ^^^ ((1UL <<< 30)
           - 1UL)

        let mutable x = 0UL
        for i = 1 to (n/3) do
            x <- x + a
            yield (x &&& (((1UL <<< l) - 1UL) <<< 30), 1)

        for i = 1 to ((n + 1)/3) do
            x <- x + a
            yield (x &&& (((1UL <<< l) - 1UL) <<< 30), -1)

        for i = 1 to (n + 2)/3 do
            x <- x + a
            yield (x &&& (((1UL <<< l) - 1UL) <<< 30), 1)
    }
```

Følgende kode genererer en strøm i C#:

```
public static IEnumerable<Tuple<ulong, int>> CreateStream(
    int n, int l) {
    // We generate a random uint64 number.
    Random rnd = new System.Random();
    ulong a = 0UL;
    Byte[] b = new Byte[8];
    rnd.NextBytes(b);
    for(int i = 0; i < 8; ++i) {
        a = (a << 8) + (ulong)b[i];
    }

    // We demand that our random number has 30 zeros on the
       least
    // significant bits and then a one.
    a = (a | ((1UL << 31) - 1UL)) ^ ((1UL << 30) - 1UL);

    ulong x = 0UL;
    for(int i = 0; i < n/3; ++i) {
        x = x + a;
        yield return Tuple.Create(x & (((1UL << l) - 1UL) <<
            30), 1);
    }

    for(int i = 0; i < (n + 1)/3; ++i) {
        x = x + a;
        yield return Tuple.Create(x & (((1UL << l) - 1UL) <<
            30), -1);
    }

    for(int i = 0; i < (n + 2)/3; ++i) {
        x = x + a;
        yield return Tuple.Create(x & (((1UL << l) - 1UL) <<
            30), 1);
    }
}
```