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. Implementeringsprojektet skal afleveres den 9. 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 af 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, skal I gange et 89-bit tal med et 64-bit tal så her rækker 128-bit multiplikation ikke engang). 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 hashtabel 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, at algoritmen kører meget langsommere 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

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

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. Bemærk at I her skal benytte 64-bit multiplikation hvor overflow over det 64'te bit smides væk. I må således *ikke* benytte `bigint` eller `BigInteger` ved implementering af multiply-shift.

(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 sektion 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 hashtabel 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 sektion 1.2 hvor I selv kan specificere n samt antallet af forskellige nøgler, 2^l. Det er vigtigt at I altid sørger for at $2^l \leq n$, da antallet af forskellige nøgler aldrig kan overstige det totale antal nøgler i strømmen. 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 krævet at I vælger n tilstrækkelig stor til at starte med så I i alle eksperimenter har at $2^\ell \leq n$. 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. Rapporter også det første l hvor algoritmen ikke terminerede fordi det tog for lang tid. 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. Specifikt 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);
    }
}
```

1.3 Del 2: Count-Sketch

I del 2 skal I ved brug af såkaldte count-sketches analysere kvadratsummen S for de samme datastrømme mere effektivt og med meget mindre plads. I modsætning til jeres løsning med hashing med chaining, der kan bruges til at udregne S præcist er count-sketches approksimative: jo mere plads, jo bedre præcision. I denne del vil vi kun fokusere på en enkelt strøm. I skal se hvordan tid og præcision udvikler sig med flere tællere og sammenligne med den langsommere løsning fra Del 1 hvor det lykkedes at beregne S præcist.

Vi genkalder kort hvordan Count-Sketch fungerer. For mere udførlige detaljer refereres I til kursusnoterne. Vi vælger først to hashfunktioner $h : U \to [m]$ og $s : U \to \{-1, 1\}$ for et givet[1] m (der afhænger af den præcision vi ønsker i vores estimat af S). Vi initialiserer dernæst et array $C \leftarrow 0^m$. For et element (x, d) i vores datastrøm opdaterer vi $C[h(x)] \leftarrow C[h(x)] + s(x)d$. Efter at vi har processeret alle elementer i strømmen sætter vi $X = \sum_{y \in [m]} C[y]^2$. I den pæne teoretiske del af noterne om second moment estimation

[1] Bemærk at noterne kalder antallet af counters k og ikke m

(sektion 3) bliver det vist at hvis Count-Sketch implementeres ved at benytte uafhængige 4-universelle hashfunktioner h og s, vil

$$\mathbb{E}[X] = S \quad \text{og} \quad \text{Var}[X] \leq 2S^2/m$$

Bemærk at S har det andet navn F_2 i noterne. Det er imidlertid ikke ligetil at implementere sådanne 4-universelle hashfunktioner effektivt. I stedet skal I bruge konstruktionen forklaret i sektion 4 i noterne om second moment estimation. I vil blive guidet igennem denne konstruktion i Opgave 4 og 5. Med denne konstruktion er h og s hverken helt uafhængige eller helt 4-universelle. Imidlertid er det vist i noterne at med denne (meget effektive) implementering af hashfunktionerne vil

$$S \leq \mathbb{E}[X] \leq (1 + u/p^2)S, \quad \text{og} \quad \text{Var}[X] \leq (1 + 3u/p^2)2S^2/m$$

Da $p = 2^{89} - 1$ og $u = 2^{64}$ er disse estimater *meget* tæt på dem vi får med rigtige 4-universelle hashfunktioner.

Opgave 4. Implementering af 4-universel hashfunktion. Implementer hashfunktionen $g : U \to [p]$, parametriseret af a_0, a_1,a_2,a_3, og defineret som

$$g(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \mod p.$$

Her er $p = 2^{89} - 1$, og a_0, a_1,a_2 og a_3 uafhængige og uniformt tilfældige i $[p] = \{0, \ldots, p-1\}$. I kan bruge `www.random.org/bytes` til at generere a_0, a_1,a_2 og a_3 på samme måde som i opgave **1**(b). Til implementeringsdetaljerne skal i bruge Algoritme 1 (med $k = 4$) i noterne om second moment estimation.

Opgave 5. Implementering af hashfunktioner til Count-Sketch Lad $m = 2^t \leq 2^{64}$ være en toerpotens. Implementer en funktion der som input tager t samt en hashfunktion $g : U \to [p]$ og kan evaluere de to hashfunktioner $h : U \to [m]$ og $s : U \to \{-1, 1\}$ defineret ved

$$h(x) = g(x) \mod m, \quad og$$
$$s(x) = 1 - 2\left\lfloor \frac{g(x)}{2^{b-1}} \right\rfloor.$$

Her er $p = 2^b - 1 = 2^{89} - 1$, så $b = 89$. Med andre ord består $h(x)$ af de $\log_2(m) = t$ mindst betydende bits af $g(x)$ og $s(x)$ er enten -1 eller 1 afhængigt af værdien af den mest betydende bit i $g(x)$. Til implementeringsdetaljerne skal I bruge Algoritme 2 i noterne om second moment estimation.

I skal nu implementere Count-Sketch.

Opgave 6. Implementering af Count-Sketch Implementer Count-Sketch, der er parametriseret ved hashfunktioner $h : U \to [m]$ og $s : U \to \{-1, 1\}$ hvor $m = 2^t$ er en toerpotens. I skal desuden implementere en funktion, der givet jeres sketch $C[0, \ldots, m-1]$ udregner estimatet $X = \sum_{y \in [m]} C[y]^2$ for S.

I skal nu bruge generatoren fra sektion 1.2 til at generere en strøm af nøgler og teste kvaliteten af de estimater I får for S ved brug af Count-Sketch. Mere præcist skal I løse følgende opgave.

Opgave 7. Eksperimenter med jeres implementering Generer en strøm af nøgler, σ ved brug af koden i sektion 1.2. Vælg antallet af forskellige nøgler lige under den grænse I rapporterede i Opgave 3 hvor jeres datastruktur ikke kunne klare det mere. Brug jeres implementering af hashing med chaining fra del 1 til at beregne den eksakte værdi af S for strømmen σ. Gentag nu følgende eksperiment 100 gange:

- Beregn Count-Sketch af datastrømmen σ. Hashfunktionerne h og s skal I vælge som I opgave 5 hvor I bruger hashfunktionen, g, fra opgave 4 som input.
- Beregn estimateren $X = \sum_{y \in [m]} C[y]^2$.

Da I bruger den samme strøm af data til hvert eksperiment er det meget vigtigt at I bruger nye tilfældige bits til at implementere jeres hashfunktioner i hvert eksperiment — ellers vil I altid få den samme værdi af X. Lad $X_{(1)}, \ldots, X_{(100)}$ være jeres estimater sorteret således at $X_{(1)} \leq \cdots \leq X_{(100)}$. Præsenter resultaterne af jeres eksperimenter som 100 punkter $(1, X_{(1)}), \ldots, (100, X_{(100)})$ i et koordinatsystem. Indsæt også en horisontal linje ved den faktiske værdi af S (som I har beregnet præcist ved brug af hashing med chaining). Beregn og rapporter også mean square error over jeres 100 eksperimenter $\sum_{i=1}^{100}(X_{(i)} - S)^2/100$ som I forventning er den rigtige varians. Diskuter om jeres resultater stemmer overens med udregningerne af middelværdi og varians for estimateren X,

$$\mathbb{E}[X] = S \quad \text{og} \quad \mathrm{Var}[X] \approx 2S^2/m.$$

Betragt nu de (usorterede) resultater af jeres eksperimenter $X_1, \ldots, X_{100}$. Del dem i 9 grupper af størrelse 11, $G_1 = (X_1, \ldots, X_{11}), \ldots, G_9 = (X_{89}, \ldots, X_{99})$. I kommer altså til at få X_{100} til overs. Beregn nu $M_i = \mathsf{median}(G_i)$ for $i = 1, \ldots, 9$ og sorter værdierne $M_{(1)} \leq \cdots \leq M_{(9)}$. Præsenter dem som 9 punkter i et koordinatsystem $(1, M_{(1)}), \ldots, (9, M_{(9)})$ og indlæg igen en horisontal linje ved den faktiske værdi af S. Diskuter hvad I ser. I skulle gerne observere at jeres punkter ligger væsentligt tættere på den horisontale linje der repræsenterer den faktiske værdi af S.

Opgave 8. Betydning af m og for kvaliteten af jeres estimat og for køretid Gentag de 100 eksperimenter fra opgave 7 med 2-3 forskellige værdier af m (men med den samme strøm af data σ) og præsenter resultaterne i et tilsvarende format. Rapporter igen mean square error. Stemmer den overens med hvad I forventer? Rapporter også køretiden af Count-Sketch for de forskellige værdier af m. Sammenlign med køretiden I fik når I brugte hashing med chaining. Hvor god en approksimation kan I få af S og stadig have en markant lavere køretid end med hashing med chaining?