UnsignedBigInt

Toteutusdokumentti

1. Ohjelman rakenne

Ohjelma on luokkarakenteeltaan yksinkertainen. UnsignedBigInt -luokka toteuttaa yleisimmät aritmeettiset operaatiot ja tarjoaa Javan omaa BigInteger-luokkaa muistuttavan rajapinnan, jonka avulla lukuja voidaan käsitellä. BitArray -luokkaa käytetään sisäisesti säilyttämään lukujen bittiesitys long-kokonaislukumuuttujien avulla. Tämän lisäksi BitArray toteuttaa muutaman matalan tason bittioperaation.

DiffieHellman -luokka on esimerkki kirjaston käytöstä ja se suorittaa Diffie-Hellman-protokollan mukaisen avaimenvaihdon. Lisäksi ohjelmasta löytyvät testiluokat UnsignedBigIntTest ja RandomizedTest, joista ensimmäinen koostuu perinteisistä yksikkötesteistä ja jälkimmäinen on tarkoitettu kirjaston satunnaistestaukseen Javan omaa BigInteger-luokkaa vasten.

2. Aika- ja tilavaativuus

Alla on esitetty muutaman kirjastossa toteutetun algoritmin aika- ja tilavaativuudet pseudokoodiesityksen avulla:

Yhteenlasku

```
function sum(a, b) {
        result = 0
        c = 0
        // laskettaessa kaksi n-bittistä lukua yhteen, suoritetaan for-silmukka n.bits + 1 kertaa
        // eli aikavaativuus on luokkaa O(n) * yhden toistokerran aikavaativuus
        for(i = 0; i <= max(a.bits, b.bits); i++) {
                // on helppo nähdä, että kaikkien lohkon sijoitusoperaatioiden ja ehtolauseen
                // ehdon tarkistuksen aikavaativuus on O(1), eli yhden toistokerran aikavaativuus
                // on luokkaa O(1)
                s = a[i] + b[i] + c
                c = 0
                if(s >= 2) {
                        c = 1
                        s = s - 2
                }
                result[i] = s
        }
        return result
}
```

Silmukka suoritetaan n kertaa ja yksi toistokerta voidaan suorittaa vakioajassa eli algoritmin aikavaativuus on luokkaa O(n). Apumuuttujat i, s ja c vievät kaikki vakiomäärän tilaa, sillä ne voidaan esittää esimerkiksi 32-bittisinä kokonaislukumuuttujina. Sen sijaan result on mielivaltaisen kokoinen taulukko. Toistolohkoa tarkastelemalla huomaamme, että sijoituslause result[i] = s sijoittaa taulukon result indeksiin i muuttujan s arvon. Tiedämme, että i saa suurimman arvonsa for-silmukan viimeisellä toistokerralla, jolloin i = max(a.bits, b.bits) + 1 eli i = n + 1. Tällöin taulukon koko on suurimmillaan n + 1 eli tilavaativuus on O(n).

Vähennyslaskun aika- ja tilavaativuus osoitetaan vastaavalla tavalla.

Kertolasku

```
function product(a, b) {
	result = 0

	// silmukka suoritetaan b.bits = n kertaa eli algoritmin aikavaativuus on luokkaa
	// O(n) * toistokerran aikavaativuus
	for(i = 0; i < b.bits; i++) {
		if(b[i] == 1) {
			temp = clone(a)			// n-bittisen luvun kopiointi O(n)
			shift temp left i times		// O(n/2) = O(n)
			result = sum(result, temp)	// sum-funktion aikavaativuus O(n)
		}
	}

	return result
}
```

Näemme, että yhden toistokerran aikavaativuus on O(3n) = O(n), joten kun silmukka suoritetaan n kertaa on algoritmin aikavaativuus tällöin O(n^2). Tilavaativuuden määrittävät apumuuttujat result ja temp, joiden tilavaativuus on suurimmillaan silmukan viimeisen suorituskerran aikana O(n^2).

Jakolasku

```
function divide(a, b) {
	quotient = 0
	remainder = 0

	// silmukka suoritetaan n kertaa
	for(i = n - 1; i >= 0; i--) {
		shift remainder left once		// O(n)
		result[0] = a[i]			// O(1)
		if(remainder >= b) {			// O(n)
			remainder = remainder - b;	// O(n)
			quotient[i] = 1			// O(1)
		}
	}

	return result
}
```

Yhden toistokerran aikavaativuus on luokkaa O(n) eli kun silmukkaa toistetaan n kertaa, on algoritmin aikavaativuus O(n^2). Tilavaativuutta dominoi muuttuja remainder, jolle pätee aina ehtolauseen vuoksi aina remainder < 2b. Tällöin tilavaativuus on luokkaa O(n+1) = O(n).

3. Puutteet ja parannusehdotukset

Bittioperaatioista vasemmalle ja oikealle siirto eivät toimi optimaalisella tavalla, kun siirto on yli yhden bitin suuruinen. Nykyinen toteutus toimii käytännössä siten, että siirrettäessä esimerkiksi viisi bittiä vasemmalle, suoritetaan viisi yhden bitin suuruista siirtoa. Koska siirtoa ei ole mahdollista suorittaa mielivaltaisen suurelle kokonaisluvulle yhdellä konekäskyllä (vaan siirrot täytyy tehdä määrätty yksikkö, esimerkiksi 64-bittiä kerrallaan), tulee siirtojen aikavaativuudeksi O(n * m). Tämä johtaa siihen, että esimerkiksi nykyisellä toteutuksella kertolaskun aikavaativuus ei mahdu luokan O(n^2) sisään.

Ajanpuutteen vuoksi kirjasto ei määrittelydokumentista poiketen käytä kertolaskuun Karatsuba-algoritmia, vaan kertolasku lasketaan aina ”naiivin” basecase-algoritmin avulla. Alustava toteutus algoritmista löytyy kuitenkin karatsuba -metodin sisältä UnsignedBigInt-luokasta. Tämänhetkisen toteutuksen ongelmana pinon ylivuoto, joka tapahtuu suurilla luvuilla laskettaessa. Yksi mahdollinen vaihtoehto ongelman ratkaisemiseksi olisi esimerkiksi algoritmin muuttaminen rekursiivisesta iteratiiviseksi.

Parannusehdotuksena mainittakoon esimerkiksi se, että kirjaston tehokkuutta voisi selkeästi optimoida aiempien puutteiden korjaamisen lisäksi nostamalla käytettävää kantalukua. Tällä hetkellä kaikki operaatiot suoritetaan bitti kerrallaan (kantaluku 2^1), mutta kantaluvuksi voitaisiin valita esimerkiksi 2^8, jolloin jokainen operaatio suoritettaisiin tavu kerrallaan tai jopa 2^32, jolloin saavutettaisiin merkittäviä nopeusetuja hyödyntämällä laitteiston ja suoritusympäristön tarjoamia optimointeja.

4. Testaus

UnsignedBigInt -luokkaa testataan yksikkötestein UnsignedBigIntTest -luokan ja satunnaistestauksen avulla RandomizedTest -luokan kautta. Näistä UnsignedBigIntTest sisältää yksikkötestit luokan jokaiselle metodille, kun taas RandomizedTest testaa vain muutaman keskeisimmän aritmeettisen operaation toimintaa.

Ohjelmaa on testattu runsaasti myös käsin, mutta varsinaista testidataa ei ole saatavilla. Kokemusten perusteella kirjaston operaatiot toimivat virheettömästi, mutta ovat todella hitaita muihin vastaaviin kirjastoihin (esimerkiksi GMP) tai edes Javan BigInteger-luokkaan verrattuna. Tämä oli jokseenkin odotettavaa. Vastaavaa suorituskykyä on todella hankala saavuttaa, mutta kohdassa 3. esitetyin muutoksin voitaisiin luultavasti saavuttaa hyvinkin merkittäviä nopeusetuja nykyiseen toteutukseen verrattuna.