Algoritmer og Datastrukturer E2024

Simone Ingvild Lebech

Afleveringsdato: 29. Oktober 2024

Underviser: Ole Dolriis

Contents

1 Opgave 1

Opgave: Der skal skrives en rekursiv algoritme, som har et naturligt tal som parameter og returnerer summen af de ulige tals kvadrater fra 1 til N.

Rekursion virker ved at lave en stack af funktionskald. Når funktionen kalder sig selv oprettes en ny instans af funktionen, som lægges på stakken. Denne process fortsætter, indtil en base case nås, som er en betingelse, der stopper rekursionen. Når base casen er nået, begynder funktionskaldene at blive fjernet fra stakken og returnere deres resultater.

1.1 Forklaring af kode

Koden beregner summen af kvadraterne af alle ulige tal fra 1 op til en given n-værdi ved hjælp af rekursion. Hvis n er mindre end 1 betyder det, at der ikke er flere ulige tal at summere over, og metoden returnerer 0. Hvis n er et ulige tal, tilføjes kvadratet af n til summen, og metoden kaldes rekursivt med n-2. Dette springer det næste lige tal over, og fortsætter med at summere de resterende ulige tal. Hvis n derimod er lige, kaldes metoden rekursivt med n-1, her er det ikke nødvendigt at inkludere n i summen for det er lige, og det ulige tal før n vil allerede være inkluderet i det tidligere rekursive kald.

Koden ses vedlagt som javafilen ex1.

2 Opgave 2

Opgave: Find store-O tidskompleksisteten for den givne algoritme og begrund.

```
public static int myMethod( int N )
{
    int x = 0; int y = 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            for (int k = 0; k < N*Math.sqrt(N); k++)
            {
                x++;
            }
            j *= 2;
        }
        i += i;
    } for (int i = 0; i < N*N; i++)
```

```
            y++;
        return x+y;
}
```

Tidskompleksiteten er et mål for, hvor hurtigt en algoritme kan udføre en opgave ift. størrelsen af inputdata (altså en vækstrate). For at bestemme tidskompleksiteten af den givne algoritme, analyseres hver af de nestede loops.

2.1 Løkke-analyse

Første løkke:

- Yderste løkke: itererer N gange.
- Miderste løkke: itererer også N gange for hver iteration af den yderste løkke.
- Inderste løkke: itererer $N \cdot \sqrt(N)$ gange for hver iteration af de to ydre løkker.

Linjen $x++$ udfører altså $N \cdot N \cdot N \cdot \sqrt(N)$ gange, mens linjerne $j = 2$ og $i+ = 1$ i de inderste loops ikke påvirker den asymptotiske tidskompleksitet, fordi de udføres et konstant antal gange for hver iteration.

Anden løkke:

- Løkken itererer $N \cdot N$ gange, og $y++$ udføres lige så mange gange.

2.2 Tidskompleksitet

I den givne kode er det det første sæt af løkker som er tidsdominerende, og herunder er det den inderste løkke. Det samlede antal iterationer i den inderste løkke bliver $N \cdot N \cdot N \cdot sqrt(N) = O(N^{2.5})$. Da vi kun kigger på worst case scenariet ser vi bort fra lavere ordens led, og tidskompleksiteten bliver da $O(N^{2.5})$. Det betyder at køretiden for algoritmen vokser proportionalt med inputstørrelsen, så jo større N, jo længere køretid. Den er altså ret ineffektiv ved store N værdier.

3 Opgave 3

Opgave: Der skal skrives en rekursiv algoritme med signaturen: bool Additive(String s), og parameteren skal indeholde en streng bestående af cifre. Det skal tjekkes om der er en sekvens af tre tal som følger hinanden, hvor det tredje ciffer er lig summen af de to foregående.

3.1 Forklaring af kode

Hvis strengen er mindre end 3 lang, returneres false med det samme, da der i så fald ikke kan være en sekvens af 3 cifre. Herefter konverteres de første tre tegn til tal ved at trække ASCII-værdien for 0 fra. Herunder tjekkes for sum-egenskaben, altså om det tredje tal er lig med summen af de to første, hvis dette er tilfældet returneres true. Skulle sum-egenskaben ikke være gældende, fjernes det første tegn, og funktionen kaldes rekursivt på den resterende sekvens.

Kode ses vedlagt som javafilen AdditiveString.

4 Opgave 4

Opgave: Der skal skrives en algoritme, der har et array af usorterede, entydige naturlige tal som input, og så skal der findes de tre tal i arrayet, hvis sum er tættest på en potens af 2. For den skrevne algoritme skal store-O tidskompleksiteten findes, og muligheder for yderligere optimering diskuteres.

4.1 Forklaring af kode

Der er valgt en brute force tilgang, hvor alle de mulige kombinationer af tre tal i arrayet genereres, og der derefter findes den nærmeste potens af 2 for hver kombination.

nearestPowerOfTwo-metoden finder den nærmeste potens af 2 til et givet tal. Variablen power sættes til 0, der køres en iterativ løkken hvor, så længe $2^{(power)}$ er mindre end det givne tal, så øges power med 1. Metoden returnerer så $2^{(power-1)}$ som er den nærmeste potens af 2.

findClosest-metoden finder de tre tal i listen, hvis sum er tættest på en potens af 2. Der itererers over alle tripletkombinationer, for hver kombination regnes så summen af de tre tal, og den nærmeste potens af 2 findes ved ovenstående metode. Nu beregnes forskellen mellem summen og den nærmeste potens, og skulle denne forskel være mindre end den nuværende mindste forskel, er den nuværende kombination bedre end den foregående. Efter alle de mulige kombinationer er gennemløbet returnerer metoden et array med de tre tal og den nærmeste 2'er potens.

4.2 Tidskompleksitet

Man ender med en kubisk tidskompleksitet $O(N^3)$ da der bruges tre indlejrede løkker.

4.3 Optimering af kode

Der er flere muligheder for optimering af koden, eksempelvis kunne man sortere listen ved start og dermed muligvis reducere antallet af kombinationer som skal gennemløbes. Man kunne også bruge hashing og finde komplementære tal til lagrede delsummer.

Kode ses vedlagt som javafilen ClosestToPowerOfTwo.

5 Opgave 5

Opgave: Find store-O tidskompleksisteten for den givne metode og begrund.

```
int myMethod( int N )
{
    int x = 0;
    for (int i = 1; i <= Math.sqrt(N); i++)
    {
            for(int j = 1; j <= N; j++)
                for (int k = 1; k < N;)
                    {
                        x++;
                        k = k * 2;
                    }
    }
    return x;
}
```

5.1 Tidskompleksitet

Metoden indeholder 3 nested loops, som itererer over forskellige intervaller afhængigt af inputværdien N. For at bestemme tidskompleksiteten laves en løkkeanalyse.

5.2 Løkke-analyse

Første løkke:

- Yderst: Itererer fra 1 til $\sqrt(N)$. Antallet af interationer er altså proportionalt med kvadratroden af N, som kan skrives $O(\sqrt{N})$.
- Mellemst: Itererer fra 1 til N for hver iteration af yderste løkke. Antallet af interationer er proportional med N, hvilket giver et bidrag på $O(N)$ for hver iteration af yderste løkke.

- Inderst: Antallet af iterationer er lidt mere besværligt her, fordi k dobles i hver iteration. Fordi k starter ved 1 og går op til N-1, vil antallet af interationer højest være logaritmisk ift. N. Dette skrives $O(log(N)$.

For at finde den samlede tidskompleksitet ganges bidragene fra hver løkke: $O(\sqrt{N} \cdot N \cdot log(N))$. Ved at bruge regnereglerne for O-notationen kan det skrives: $O(N^{\frac{3}{2}} \cdot log(N))$. Det betyder at køretiden for algoritmen vokser lidt hurtigere end kubisk ift. inputsstørrelsen af N.

6 Opgave 6

Opgave: Der skal skrives en rekursiv algoritme med følgende signatur: int sumDivisibleBy3(int N). Algoritmen skal returnere summen af heltal > 0 og $\leq$ N, som er dividerbar med 3. Undgå overflødige kald.

6.1 Forklaring af kode

Vi ønsker at finde summen af alle tal fra 1 til N, som kan deles på 3. Dette gøres ved en rekursiv funktion og memoization. Der oprettes et array til at huske resultaterne af tidligere beregninger, hvert indeks i arrayet tilsvarer en værdi af N, og værdien på det indeks er summen af alle tal op til den tilsvarende N der kan deles på 3. Når funktionen kaldes med en bestemt N, tjekkes først om resultatet allerede er beregnet og gemt i memo, hvis dette er tilfældet returneres det, hvis det ikke er gemt udføres der beregninger. Hvis $N \leq 3$ er der ingen tal > 0 og $\leq N$ som kan deles på 3, og der returneres 0.

Der er to rekursive tilfælde; hvis N kan deles med 3, lægges N til summen og funktionen kaldes rekursivt med N-3. Hvis N ikke kan deles med 3, akldes funktionen rekursivt med N-1.

Kode ses vedlagt som javafilen SumDivisibleBy3

7 Opgave 7

Opgave: Der skal findes den største heltallige værdi af X og Y hvor $X^Y = Z$, under forudsætning af, at $X > 2$ og $Y > 2$. Findes flere løsninger vælges den med størst X-værdi.

7.1 Forklaring af kode

Der tjekkes om Z er indenfor de tilladte grænser, og hvis ikke, returneres en fejl. Derefter initialiseres maxX og maxY variablerne til -1, for at indikere at der ikke er en løsning endnu. Nu itereres over alle de mulige X fra 3 til $\sqrt{Z}$, dette er en optimering da Y altid vil være $\leq \sqrt{Z}$. For hver værdi af X laves en binærsøgning for at finde tilsvarende Y, det gøres ved at søge efter den største

Y-værdi hvor $X^Y \leq Z$. Hvis der findes en løsning og den aktuelle X $> maxX$ opdateres maxX og maxY med nye værdier. Skulle der ikke være en løsning returneres et array med $-1, 1$.

Kode ses vedlagt som javafilen PowerDecomposition.

8 Opgave 8

Opgave: Der skal indsættes 3 nye elementer i den givne hashtabel ved quadratic probing.

Quadratic probing er en teknik til at løse kollisioner i hashtabeller. Når der opstår en kollision, prøver vi at placere elementet et nyt sted, ved at tilføje et kvadratisk tal til det oprindelige indeks.

- Indsæt Q: Q skal indsættes på 7, som er ledigt. Q placeres på 7.
- Indsæt C: C skal indsættes på 8, som er optaget af E. Derfor prøves $8 + 1^2(9)$, som er ledig. C placeres på 9.
- Indsæt H: H skal indsættes på 2, som er optaget. Derfor prøves $2 + 1^2(3)$ som er optaget af R. Der fortsættes med $2 + 2^2(6)$ som er optaget af P. Der fortsættes med $2 + 3^2(11)$ som ligger udenfor tabellen, og der startes fra indeks 0 som er tomt. H placeres på 0.

Den nye tabel ses:

0	H
1	
2	V
3	R
4	
5	
6	P
7	Q
8	E
9	C
10	F

9 Opgave 9

Opgave: Store-O tidskompleksiteten for den givne metode skal findes.

```
public static long myMethod( int N )
{
    if (n <= 1)
        return 1;
    else
        return myMethod(n-1) + myMethod(n-2);
}
```

Det ses at den givne metode beregner et tal baseret på Fibonacci-sekvensen, da den rekursivt kalder sig selv med n-1 og n-2. Dette skaber et træ af rekursive kald, hvor antallet af noder vokser eksponentielt med n.

9.1 Tidskompleksitet

Hvert kald til myMethod udfører et konstant antal operationer (sammenligning og addition), og antallet af kald vokser eksponentielt med n. Problemet ligger i den overlappende beregning. For eksempel, når vi beregner myMethod(5), vil vi også beregne myMethod(4) og myMethod(3). Men når vi beregner myMethod(4), vil vi igen beregne myMethod(3). På den måde er der mange beregninger som udføres gentane gange, og det er altså her den eksponentielle vækst kommer fra. Derfor bliver tidskompleksititen også $N(2^n)$ sådan at køretiden stiger med en faktor 2 for hver stigning i n.

10 Opgave 10

Opgave: Der skal skrives en rekursiv metode som beregner den binære logaritme (base 2) af et tal N, under forudsætning af at N er en potens af 2.

10.1 Forklaring af kode

Metoden log2(int N) beregner den binære logaritme af et givet tal N. Hvis N er lig med 1, er logartimen til basis 2 af 1 lig med 0. Dette er basisfilfældet i rekursionen. Der er også et rekursivt tilfælde, hvis N ikke er lig med 1, deles N med 2, og log2 kaldes rekursivt på resultatet. Derefter tilføjes 1 til resultatet af det rekursive kald - dette skyldes at man ved hver division med 2 går et niveau ned i det binære system.

Eksempel: log2(8) kaldes:
Da 8 ikke er lig 1 divideres med 2. Der kaldes rekursivt log2(4).

Da 4 ikke er lig 1, divideres med 2. Der kaldes rekursivt log2(2).

Da 2 ikke er lig 1, divideres med 2. Der kaldes rekursivt log2(1)

Da 1 er lig 1, returneres 0.

log2(2) = 1 + 0 = 1

log2(4) = 1 + 1 = 2

log2(8) = 1 + 2 = 3

Altså vil den binære logaritme af 8 være 3.

Koden ses vedlagt i javafilen log2.

11 Opgave 11

Opgave: Der skal skrives en algoritme som kaldt med tabellen kan afgøre om en kandidat ved et valg fik mere end 50% af stemmerne, i så fald returneres kandidaten nummer, ellers -1.

11.1 Forklaring af kode

Der bruges et HashMap til at tælle stemmer for hver kandidat, og Map bruges til at definere en generel dataStruktur til at lagre stemmerne. Der findes en vinder baseret på en liste over stemmer og der returneres enten kandidaten ID eller -1.

Indenfor findWinnder metoden ses det at der oprettes et HashMap til at lagre stemmerne for hver kandidat (kandidatens ID bruges som nøgle og stemmeantal er værdien). totalVotes tæller det samlede antal stemmer, mens winningThreshold beregner tærsklen for at vinde (over halvdelen af stemmerne, altså mere end 50%. Der itereres gennem listen af stemmer, og for hver stemme øges stemmeantallet for den givne kandidat, hvis kanidaten ikke har modtaget andre stemmer endnu sættes værdien til 0. For at finde den endelige vinder kigges alle kandiaterne og stemmeantal igennem, hvis kandidatens stemmeantal er højere end tærsklen vinder de og ellers returneres -1 for at vise der ikke er en klar vinder.

11.2 Tidskompleksitet

Tidskompleksiteten er domineret af de to for-løkker. Derfor laves en løkkenalyse

Første løkke itererer over listen af stemmer, og hver gang udføres et konstant antal operationer, der hentes en stemme fra listen, der søges efter stemmen og stemmeantallet øges for kandidaten. Antallet af iterationer er lig med længden på listen af stemmer, som kaldes n.

Anden løkke itererer gennem antallet af stemmer per kanidat og der udføres igen et konstant antal operationer, der hentes en kandiat og stemmeantal og der sammenlignes med vindertærsklen. Antallet af iterationer er så i værste tilfælde lig med antallet af kandidater i listen. Hvis alle stemmer forskelligt vil dette være op til n.

Tidskompleksiteten er $O(n)$ i værste tilfælde og dermed stiger køretiden proportionalt med antal stemmer (n).