IDMA: Aflevering 1

Jeppe Bonde Bakkensen | slr105

20/12/2024

Indhold

Jeppe Bonde Bakkensen

Opgave 1

Consider the following pseudocode where A and B are arrays indexed from 1 to n:

```
for i := 1 upto n {
    B[i] := 1
    for j := 1 upto i {
        B[i] := B[i] * A[j]
    }
}
```

1.a Explanation in Plain Language

Explain in plain language what the algorithm above does. In particular, what is the meaning of the entries B[i] after the algorithm has terminated?

Algoritmen udregner produktet af elementer i arrayen A og gemmer resultatet i $B[i]$.

- For hvert index i fra 1 til n, sætter algoritmen $B[i]$ til værdien 1
- Derefter itererer algoritmen gennem det indre loop for hvert index $A[j]$ fra 1 til i
- Produktet af det i´te element i A udregnes og gemmes i $B[i]$

Dvs. når algoritmen har gennemløbet alle elementer indeholder $B[i]$ de kumulerede produktet af $A[i]$. Svarende til:

$$B[i] = \prod_{j=1}^{i} A[j]$$

1.b Time Complexity Analysis

Provide an asymptotic analysis of the running time as a function of the array size n.

Algoritmen består af et ydre loop, som kører fra 1 til n gange. For hvert ydre loop-kald udførers i iterationer i det indre loop. I worst case betyder det, den samlet køretid svarer til:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Ved at ignorere de konstante faktorer har algoritmen en upperbound køretide svarende til: $O(n^2)$. Dvs. at algoritmens køretid vokser kvadratisk med n.

1.c Optimized Algorithm and Asymptotic Optimality

Can you improve the code to run faster while retaining the same functionality? How much faster can you get the algorithm to run? Analyse the time complexity of your new algorithm. Can you prove that it is asymptotically optimal?

Jeg har forbedret algoritmen ved at fjerne det ydre for loop, hvor $B[i]$ altid er 1 og erstattet det med en konstanten $B[1] = A[1]$.

My optimized pseudocode:

```
B[1] := A[1]
for i := 2 upto n
    B[i] := B[i-1] * A[i]
```

Algoritmen består nu af et loop, som udregner den kumulative produkt af en Array A med n elementer. For hvert element i $A[i]$ udfører algoritmen en multiplikationsoperation og tildeler $B[i]$ den nye værdi. Dette svarer til konstant tid $O(1)$. Da algoritmen itererer gennem alle n elementer en gang, er den samlet køretid $O(n)$.

Eftersom at algoritmen skal udregne det kumulative produkt, skal algoritmen løbe igennem alle elementer mindst en gang. Derfor må $O(n)$ den asymptotiske optimal.

Opgave 2

Consider the following pseudocode where A is an array indexed from 1 to n:

```
j := n
good := TRUE
while (j > 1 and good)
    i := j - 1
    while (i >= 1 and good)
        if (A[i] > A[j])
            good := FALSE
        i := i - 1
    j := j - 1
if (good)
    return "success"
else
    return "failure"
```

2.a Explanation in Plain Language

Explain in plain language what the algorithm above does. In particular, what do we know about the array A when `"success"` or `"failure"` is returned, respectively?

- Algoritmen starter med at sætte j til n og variablen `good` til `TRUE`.
- For hvert indeks j fra n ned til 2, udføres følgende:
  - Et loop, som sætter i til $j - 1$.
  - Et indre loop, som itererer baglæns gennem arrayet A for hvert indeks i fra $j - 1$ ned til 1.
  - Hvis et tallet på indekset før j er større end j sættes `good` til `FALSE`, og begge loops afbrydes.
- Hvis `good` stadig er `TRUE` efter alle iterationer, returneres `"success"`, ellers returneres `"failure"`.

Algoritmen kontrollerer om tallet er sorteret fra lavest til højest. Hvis listen er sorteret fra lavest til højest retuneres *"success"*. Hvis bare et element i listen har en højere værdi end det næste element retuneres *"failure"*.

2.b Time Complexity Analysis

Provide an asymptotic analysis of the running time as a function of the array size n.

Tidskompleksitet for denne algoritme bygger på de samme principper som algoritmen fra opg. 1. I denne algoritme har du igen et ydre loop som kører n - 1 gang, for hver ydre loop-kald udføres n - 1 iterationer i det indre loop. I worst case betyder det, at den smalet køretid svare til:

$$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

Ved at ignorere de konstante faktorer har algoritmen en upperbound køretid svarende til $O(n^2)$. Dvs. at algoritmens køretid vokser kvadratisk med n.

Jeppe Bonde Bakkensen

2.c Optimized Algorithm and Asymptotic Optimality

Can you improve the code to run faster while retaining the same functionality? How much faster can you get the algorithm to run? Analyze the time complexity of your new algorithm. Can you prove that it is asymptotically optimal?

Jeg har optimeret koden ved at gentænke, hvordan man sammenligner om indekset før er større end det nuværende index. Dette har jeg gjort ved at lave et enkelt for loop, som sammenligner det tidligere indeks med det nuværende indeks, hvilket bibeholder funktionaliteten af den oprindelige kode.

```
for j = 2 upto n
    if (A[j-1] > A[j])
        return "failure"
return "succes"
```

Da algoritmen, skal kontrollerer om det forrige indeks i arrayen er mindre end det nuværende indeks skal algoritmen itererer gennem alle n elementer en gang, hvilket giver en samlet køretid på $O(n)$. Ligesom i opgave 1 må $O(n)$ være den asymptotiske optimale køretid, da algoritmen i worst case skal iterere igennem alle indekser.

Opgave 3

In the following snippet of code, A is an array indexed from 1 to n that contains integers, and B is an auxiliary array, also indexed from 1 to n, that is meant to contain Boolean values.

```
for i := 1 upto n {
    if (A[i] < 1 or A[i] > n)
        return "failure"
}
i := 1
found := -1
while (i <= n and found < 0) {
    for j := 1 upto n {
        B[j] := false
    }
    j := i
    while (B[j] == false) {
        B[j] := true
        j := A[j]
    }
    if (A[A[j]] == j)
        found := j
    i := i + 1
}
return found
```

3.a Explain in plain language what the algorithm above does

Explain in plain language what the algorithm above does In particular, when does it return a positive value, and, if it does, what is the meaning of this value?

- Algoritmen starter med et **for-loop**, som itererer fra 1 til n. Dette loop sikrer, at hver værdi i arrayen A opfylder betingelserne om, at værdien er et positivt tal mellem 1 og længden af arrayen.
- Hvis alle tal i arrayen tilfredsstiller det første **for-loop** forsætter algortimen til et **while-loop**, som kører så længde $i \leq n$ og der ikke er fundet nogen cyklus (dvs. **found = -1**)
- Inde i det ydre while-loop kører der et indre for-loop fra 1 til n, som sætter alle værdierne **hjælpearrayen** B til `false`. Denne array anvendes til at holde styr på, hvilke indekser, som cyklussen allerede har stødt på i den nuværende iteration.
- Herefter sættes variablen j til i som starter med værdien 1.
- Algortimen kører nu et **indre while-loop**, hvor den looper gennem værdierne i A indtil den opdager en gentagelse:
  - Hvis $B[j]$ er **false** markeres værdien nu til **true** og j sættes nu til denne værdi af $A[j]$. Herefter "hopper"algoritmen til det næste indeks af baseret på værdien i af $A[j]$.
  - Når algortimen rammer en værdi, som allerede er sat til **true** i B, betyder det at der er fundet en **cyklus**.

- Når en cyklus er fundet, evalueres den værdi af j som indikeret en cyklus om den svarer til det samme indeks som værdien selv. Dvs. at hvis algoritmen fandt en cyklus når $j = 5$ og der på index 5 plads også befandt sig et femtal vil algoritmen gemme værdien i found, hvorefter den vil retunerer værdien af j.
- Hvis dette ikke er tilfældet vil i stige med 1 og algortimen vil kører igen. Når algortimen har kørt igennem alle indekser i arrayen uden at finde et match retuneres -1, som indikerer at algortimen ikke fandt en cyklus, hvor overstående betingelse gælder.

Det vil sige at algoritmen retunerer en positiv værdi, når der er fundet en cyklus og værdien af j svarer til værdien af indekset svarende til j. Dvs. at hvis algoritmen fandt en cyklus når $j = 5$ og der på index 5 plads også befandt sig et femtal vil algoritmen gemme værdien i found, hvorefter den vil retunerer værdien af j.

3b Time Complexity Analysis
Provide an asymptotic analysis of the running time as a function of the array size n.

I det første for loop itererers der gennem alle n elementer fra i til n. Hvilket svarer til til $O(n)$.

Derefter er der et while loop hvor det ydre loop kører fra i til n. Inde i while loopet er der to indre loops.

- Et for-loop som itererer gennem alle n elementer fra i til n. Hvilket svarer til en tidskompleksitet på $O(n)$.
- Et while-loop, som værste tilfælde kører gennem alle n elementer fra i til n. Hvilket svarer til en tidskompleksitet på $O(n)$.

Den samlede tidskompleksitet af de indre loops svarer til:

$$O(n) + O(n) = O(n)$$

da det ydre while-loop selv kører har en upperbound kørertid på $O(n)$, bliver den samlede tidskompleksetet af while-loopet med de indre loops
$O(n) \cdot O(n) = O(n^2)$
samlet set ser køretiden for algoritmen således ud:

$$O(n) + O(n^2)$$

Da $O(n^2)$ vokser hurtigere end $O(n)$ er algoritmens upperbound køretid $O(n^2)$. Dvs. at algoritmens køretid vokser kvadratisk med n.

3c Optimized Algorithm and Asymptotic Optimality
Can you improve the code to run faster while retaining the same functionality? How much faster can you get the algorithm to run? Analyse the time complexity of your new algorithm. Can you prove that it is asymptotically optimal?

```
for i := 1 upto n {
    if (A[i] < 1 or A[i] > n)
        return "failure"
}
for i := 1 upto n {
    B[i] := false
}
```

```
j := 1
while not B[j] {
    B[j] := true
    j := A[j]
}
if A[A[j]] == j {
    return j
}
return -1
```

Jeg har optimeret koden ved at fjerne det ydre while loop, og dermed fjernet et O(n), da loops som kører

Opgave 4

In the following snippet of code, A is an array indexed from 1 to $n \geq 2$ containing integers.

```
search (A, lo, hi)
    if (A[lo] >= A[hi])
        return "failure"
    else if (lo + 1 == hi)
        return lo
    else
        mid = floor((lo + hi) / 2)
        if (A[mid] > A[lo])
            search (A, lo, mid)
        else
            search (A, mid, hi)
```

The first call to the algorithm is `search(A, 1, n)`, where n is whatever size (at least 2) the array has.

4.a Explain in plain language what the algorithm above does

Explain in plain language what the algorithm above does. If the algorithm returns something other than "failure", then what is the meaning of the value returned?

- Algoritmen er en rekursive søge algoritme med to base cases.
  - Den første er hvis værdien $A[lo]$ er større eller lig $A[hi]$. Returnerer algoritmen *"failure"*.
  - Den anden base case er, hvis **lo + 1** er lig **hi** retuneres *lo*
- Ellers opdeles søgningen rekursivt baseret på, om A[`mid`] er større eller mindre end A[`lo`].

Hvis der på intet tidspunkt opstår en situation hvor $A[lo] >= A[hi]$, vil algortimen til sidst nå et punkt, hvor der kun er to værdier tilbage. I dette tilfælde retuneres *lo*, hvilket er indekset for den mindste værdi i arrayen.

4.b Time Complexity Analysis

Provide an asymptotic analysis of the running time as a function of the array size n

Algoritmen bliver rekursivt delt op i 2 halvdele, hvorefter algortimen kun kigger på den halvdel som opfylder betingelsen fra if-statementet. Det sker R gange indtil det er reduceret til 1, hvilken kan udtrykkes således $T(\frac{n}{2^R}) = 1$. Det svarer til at

$$n \geq 2^R$$

For at finde upperbound af køretiden issulerers R i ovenstående udtryk.

Jeppe Bonde Bakkensen

$$R \leq log_2(n)$$

Dvs. den rekursive søgealgoritme har en upperbound køretid på $O(log_2(n))$.

4.c)Recursive calls and additional guarantees

Could it be the case that recursive calls of the algorithm also return "failure", or would it be sufficient to check just once before making the first recursive call?

Det er ikke tilstrækkeligt kun at kontrollere betingelsen én gang før det første rekursive kald, da lo og hi ændrer sig efter hvert kald. Dermed kan en situation, hvor $A[lo] \geq A[hi]$, opstå i et senere rekursivt kald, selv hvis den ikke var gældende i det første kald.

Hvis betingelsen ikke kontrolleres efter hvert rekursivt kald, risikerer algoritmen at forsætte på et ugyldigt interval. Dermed skal algoritmen kontroller efter hvert kald om $A[lo] \geq A[hi]$.

If we get the additional guarantee that all elements in the array are distinct, could we remove the "failure"check completely, since we would be guaranteed to never have this answer returned anyway?

Selvom arrayen har unikke værdier vil der stadig være tilfælde, hvor $A[lo] >= A[hi]$. Hvis man havde en array med 2 elementer $[4, 1]$, så selvom det er 2 unikke vil der i dette tilfælde opstå en fejl da $4 > 1$. Dermed kan man ikke "bare"fjerne failure-check, da der gennem arrayen kan være steder, hvor $A[lo] >= A[hi]$.

What about if we get the additional guarantee that the array is sorted in increasing order?

Dette delspørgsmål har jeg forstået, som om, at alle tal ikke er unikke, men at alle tal er sorteret fra lavest til højest. Ud fra betingelse om at alle værdier er sorteret kan man ikke kunne fjerne failure-check, da flere af værdierne kan være de samme, og dermed vil der være tilfælde, hvor f.eks. $A[lo] = 2$ og $A[hi] = 2$ og dermed vil betingelsen om at $A[lo] >= A[hi]$ gælde og dermed returnerer failure.

What if both of these extra guarantees apply?

Hvis arrayen er sorteret fra lavest til størst, og alle elementer er unikke vil det altid gælde at $A[lo] < A[hi]$, hvilket medfører, at situationen, hvor $A[lo] >= A[hi]$ aldrig kunne forekomme, og algoritmen aldrig vil returnerer "failure", og dermed vil bliver "failure-chekket være overflødigt.