Københavns Universitet
Introduktion til diskret matematik og algoritmer - Problem set 2

Victor Vangkilde Jørgensen - kft410

February 10, 2025

Contents

1 Question 1 - In the following snippet of code A and B are arrays indexed from 1 to n that contain numbers.

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

1.a 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?

Algoritmet ændrer hvert element i array'en B til produktet af alle elementer i A op til samme nummer element (i). Dette gøres ved at iterere fra $i = 1 \dots n$ og heraf fra $j = 1 \dots i$, hvor hver iteration sætter $B[i] := 1$, og derefter $B[i] := B[i] \cdot A[j]$

1.b Provide an asymptotic analysis of the running time as a function of the array size n. (That is, state how the worst-case running time scales with n, focusing only on the highest-order term, and ignoring the constant factor in front of this term.)

I det værste tilfælde, har vi n gentagelser af for-loop'et, som indeholder i gentagelser i endnu et for-loop. Rent notationsmæssigt, kan mængden af iterationer af det indre for-loop beskrives matematisk som:

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

da alle konstante- og led af mindre grad ignoreres.

1.c 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? (That is, that no algorithm solving this problem can run faster except possibly for a constant factor in the highest-order term or improvements in lower-order terms.)

For at gøre vores algoritme hurtigere end $O(n^2)$, bør vi se, om vi kan få en worst-case asymptotic running time complexcity på n eller bedre.
(Bemærk at jeg bruger en anden snytax for min egen pseudokode)

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

Dette nye algortime vedholder samme funktionalitet som det tidlige, og kan beskrives som:

$$\sum_{i=2}^{n} 1 = n - 1 \Rightarrow O(n)$$

asymptotic running time, hvilket er hurtigere, da dette skalerer lineært.
Jeg vil selv mene, at dette er "asymptotically optimal", da jeg ikke ser en anden måde, hvorpå man kan iterere gennem hvert element i en array, som er hurtigere end lineær running time.

2 Question 2 - In the following snippet of code A is an array indexed from 1 to n that contain elements that can be compared

```
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 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?

Algoritmet retunerer "success" eller "failure", afhængigt af om alle elementers værdi i array'en A øges i tagt med elementnummeret, eller har samme værdi. Når "success" er returneret, ved vi, at array'en A kun indeholder elementer, der har samme eller højere værdi end elementet før for hvert element i A. Sagt med andre ord er elle elementer i A sorteret fra laveste til højeste værdi, hvis "success" er returneret.
Når "failure" er returneret, er der et eller flere elementer, hvis værdi ikke øges i tagt med elementnummeret eller ikke er den samme. Med andre ord er array'en ikke sorteret i increasing rækkefølge.

2.b Provide an asymptotic analysis of the running time as a function of the array size n. (That is, state how the worst-case running time scales with n, focusing only on the highest-order term, and ignoring the constant factor in front of this term.)

Hvis while-loop'et aldrig bliver terminated tidligt, kørerer vi ydre while-loop $n-1$ gange, da vi starter fra $j = n$, og sidste iteration køres ikke, da $j > 1$ ikke opfyldes. Heraf kører det indre while-loop $i = j - 1$ gange. Det indre while-loop kan derfor i værste tilfælde beskrives som:

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

2.c 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? (That is, that no algorithm solving this problem can run faster except possibly for a constant factor in the highest-order term or improvements in lower-order terms.)

Min forbedrede version af algoritmet er som følgende:
(Bemærk at snytax'en for pseudokoden ikke er den samme som i opgaven)

```
sorted := True
for i := 1 upto n-1
    if (A[i] > A[i+1])
        sorted := False
if (sorted)
    return "success"
else
    return "failure"
```

Jeg har afskaffet begge while-loops og erstattet dem med et enkelt for-loop, som tjekker om alle elementer er i øget- eller lig rækkefølge.
For-loopet kører til $n-1$, og er det højeste led, hvilket giver os en worst-case running time:

$$\sum_{i=1}^{n-1} 1 = n - 1 \Rightarrow O(n)$$

Jeg forsøgte at gøre dette endnu hurtigere, ved at dele array'en op i flere dele, og sammenligne flere elementer på samme tid, men kunne ikke få det til at blive hurtigere en linæer tid.

3 Question 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. In particular, when does it return a positive value, and, if it does, what is the meaning of this value?

Algoritmet søgerer først for, at alle elementer i array'en A kun indeholder værdier der er imellem 1 og længden (antallet af elementer) af array'en. Hvis dette ikke er tilfældet returneres "failure", da vi senere ellers ville få en out-of-bounds error. Hvis algoritmet ikke retunerer "failure", retunerer den -1 hvis der ikke er nogle elementer, der peger på et element, som peger tilbage igen (en cyklus). Hvis sådan en cyklus bliver fundet, bliver index'et af det gældende element returneret.

3.b Provide an asymptotic analysis of the running time as a function of the array size n. (That is, state how the worst-case running time scales with n, focusing only on the highest-order term, and ignoring the constant factor in front of this term.)

I det værste tilfælde, kører vi det ydre while-loop n gange, da i går fra $i = 1 \dots n$. Det indre for-loop køreres fra $j = 1 \dots n$, så dette for-loop kan beskrives som:

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

3.c 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 laver den antagelse, at vi ikke behøver auxiliary array'en B, da det vigtige er elementerne som udgør A, samt værdien af outputten 'found'. Her er min version af algoritmet:

```
for i := 1 upto n
    if (A[i] < 1 or A[i] > n)
        return "failure"
found := -1
for j := 1 upto n
    if (A[j] == j or A[A[j]] == j)
        found := j
        return found
return found
```

Worst-case running time her ville være udløst, når $j = 1 \dots n$ kørers helt til n. Vi beskriver dette som:

$$\sum_{j=1}^{n} 1 = n \Rightarrow O(n)$$

Dette er igen en markant forbedring af $O(n^2)$.
Jeg har ikke flere indlæg til hvordan dette kan forbedres.

4 Question 4 - In the following snippet of code A is an array indexed from 1 to n $>$ 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. If the algorithm returns something other than "failure", then what is the meaning of the value returned?

Algoritmet er rekursivt, da den kalder på sig selv. For hver rekursion er der 3 mulige cases:

- Hvis værdien i den nedre grænse (lo) er større end- eller lig den i den øvre grænse (hi), returneres "failure".
- Ellers hvis den nedre grænse +1 er lig med den øvre grænse, returneres den nedre grænse. Sagt med andre ord er der kun 2 elementer tilbage, og vi vælger den første.
- Ellers udregnes mid til midten mellem den nedre- og øvre grænse ($\lfloor \frac{lo+hi}{2} \rfloor$). Hvis værdien i denne midte er strængt større end værdien i den nedre grænse, køres algoritmet igen med lo og mid, ellers køreres algoritmet med mid og hi.

Hvis alle rekursioner opfylder, at $A[lo] < A[hi]$, vil vi til sidst ende med, at lo bliver returneret. I disse tilfælde er lo index'et hvor den mindste værdi i array'en A findes.

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

Uafhængigt af størrelsen ved vi, at vi rekursivt kører search på en størrelse, der svarer til halvdelen ($\lfloor \frac{lo+hi}{2} \rfloor$) af den sidste rekursion. Dette betyder, at nedre- eller øvre grænse halveres ved hver rekursion, hvilket kan beskrives i asymptotic worst-case running time som:

$$T(n) = T(\frac{n}{2}) + O(1) \Rightarrow O(log(n))$$

Dette kan sammenlignes med *binary search*, som har samme worst-case running time $O(log(n))$, bort set fra at vi ikke leder efter en specifik værdi i det ovenstående algoritme.

4.c 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? 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? What about if we get the additional guarantee that the array is sorted in increasing order? What if both of these extra guarantees apply?

Vi kan godt sætte vores tjek om: $A[lo] >= A[hi]$ ude fra rekursion-delen, og blot køre det i begyndelsen, da rekursionen altid sker, så $A[lo]$ ikke er $>= A[hi]$. Nede under refererer jeg til $A[lo] >= A[hi]$ som $A[lo] < A[hi]$, da dette er det ønskede scenarie, som får algoritmet til at fortsætte.
Vi kan bevise dette ved at se på hvornår mid bliver sat som hhv. den øvre- og nedre grænse. Givet at vi i første iteration ved at: $A[lo] < A[hi]$ må følgende også gælde:

$$A[mid] > A[lo] \Rightarrow search(A, lo, mid) \Rightarrow \text{næste rekursion: } A[lo] < A[hi]$$

og

$$A[mid] <= A[lo] \Rightarrow search(A, mid, hi) \Rightarrow \text{næste rekursion: } A[lo] < A[hi]$$

Selvom vi måske ved, at alle elementer er unikke, betyder det ikke at vi kan fjerne tjekket i starten, da ikke alle mulige kombinationer af elementerne i A nødtvendigvis opretholder betingelsen: $A[lo] < A[hi]$. Hvis vi derimod ved, at alle elementer er sorteret i øget rækkefølge, vil dette i kombination med, at alle elementer er unikke (hvis elementerne bare var sorteret ville der stadig være mulighed for at $A[lo] == A[hi]$), gøre det muligt, at afskaffe verificeringen, at: $A[lo] < A[hi]$, da alle mulige kombinationer af disse, ville opfylde kravet på forhånd.
Så ja, det er muligt at flytte betingelsen ud fra rekursionen med den information, som var givet tidligere. Hvis vi vil fjerne betingelsen helt, skal vi derudover også have informationen, at A er sorteret i øget rækkefølge, og at alle elementer er unikke.