Københavns Universitet
Introduktion til diskret matematik og algoritmer - Problem set 1

Victor Vangkilde Jørgensen - kft410

February 9, 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).

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.)

Lad os kigge på hvordan running time fordeles i vores algoritme:

```
for i := 1 upto n {          //O(n^2)
    B[i] := 1                //O(1)
    for j := 1 upto i {      //O(n)
        B[i] := B[i] * A[j] //O(1)
    }
}
```

Vi ser, at vi har n gentagelser af for-loop'et, som indeholder n gentagelser i endnu et for-loop. Dette resulterer i en worst-case $cn^2 + cn + c \Rightarrow O(n^2)$

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 betydligt hurtigere, bør vi se, om vi kan få en worst-case asymptotic time complexcity på n eller bedre.
(Bemærk at jeg bruger en anden snytax for min egen pseudokode)

```
B[1] := A[1]                 //O(1)
for i := 2 upto n            //O(n)
    B[i] := B[i-1] * A[i]    //O(1)
```

Dette nye algortime vedholder samme funktionalitet som det tidlige, og kører i: $cn + c \Rightarrow O(n)$ 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.
Dog kan jeg ikke bevise, at dette er den hurtigste måde.

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 retuneret, 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 retuneret.
Når "failure" er retuneret, 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.)

```
j := n                          //O(1)
good := TRUE                    //O(1)
while (j > 1 and good)          //O(n^2)
    i := j - 1                  //O(1)
    while (i >= 1 and good)     //O(n)
        if (A[i] > A[j])        //O(1)
            good := FALSE       //O(1)
        i := i - 1              //O(1)
    j := j - 1                  //O(1)
if (good)                       //O(1)
```

```
    return "success"          //O(1)
else                          //O(1)
    return "failure"          //O(1)
```

$cn^2 + cn + c \Rightarrow O(n^2)$

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

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 asymptotiske led, hvilket giver os en worst-case running time: $cn + c \Rightarrow O(n)$.

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 retuneres "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 retuneret.

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.)

```
for i := 1 upto n {                 //O(n)
    if (A[i] < 1 or A[i] > n)       //O(1)
        return "failure"            //O(1)
}
i := 1                              //O(1)
found := -1                         //O(1)
while (i <= n and found < 0) {      //O(n^2)
    for j := 1 upto n {             //O(n)
        B[j] := false               //O(1)
    }
    j := i                          //O(1)
    while (B[j] == false) {         //O(n)
        B[j] := true                //O(1)
        j := A[j]                   //O(1)
    }
    if (A[A[j]] == j)               //O(1)
        found := j                  //O(1)
    i := i + 1                      //O(1)
}
return found                        //O(1)
```

Vi ser, at vi har en worst-case asymptotic running time på $cn^2 + cn + c \Rightarrow O(n^2)$, da vi har et for-loop inde i et while-loop, som begge kører n gange.

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

Dette kører med $cn+c$ worst-case running time, hvilket svarer til $O(n)$ (linæer) tid. Dette er en markant forbedring frem for $O(n^2)$, præcist som i de andre opgaver.

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 den første værdi i A er større end den sidste, retuneres "failure".
- Ellers hvis den nedre grænse (lo) + 1 er lig med den øvre grænse (hi), retuneres den nedre grænse.
- 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.

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

```
search (A, lo, hi)                    //O(log(n))
    if (A[lo] >= A[hi])               //O(1)
        return "failure"              //O(1)
    else if (lo + 1 == hi)            //O(1)
        return lo                     //O(1)
    else                              //O(1)
        mid = floor ((lo + hi) / 2)   //O(1)
        if (A[mid] > A[lo])           //O(1)
            search (A, lo, mid)       //O(log(n))
        else                          //O(1)
            search (A, mid, hi)       //O(log(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 (lo) eller øvre (hi) grænse halveres ved hver rekursion, hvilket kan beskrives i asymptotic worst-case running time som: $O(log(n))$

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?