IDATA2302 Algoritmer og Datastrukturer

Hans Georg Schaathun

Ordinær eksamen 9. desember 2020
Revidert 20. november 2020

Sensorvegleiing 9. desember 2020 IDATA2302 Algoritmer og Datastrukturer
Eksamenstid $4\frac{1}{2}$ timar *Nynorsk*

1. Sett at me har to sorterte lister a og b. Objekta i lista kan ha vilkårleg type, so lenge der er ein ordningsrelasjon $\leq$ på dei.

   (a). Gje ein mest mogleg effektiv algoritme som flettar dei to listene, dvs. returnerer ei sortert liste c som inneheld alle elementa frå a og b.

   Solution: Dei to inputlistene er allereie sorterte og det må me utnytta i flettinga. Det fyrste elementet c_1 can altso berre vera a_1 eller b_1 for alle dei andre elementa i a er større enn a_1 og i b større enn b_1.
   Lat m og n vera lengda til hhv. a og b.

```
Merge(a, b)
i := 1 [ next element in a ]
j := 1 [ next element in b ]
k := 1 [ next element in c ]
while ( i ≤ m ) and ( j ≤ n )
    if a_i ≤ b_j ,
      c_k := a_i
      increase i
    else [ a_i > b_j ] ,
      c_k := b_j
      increase j
    increase k
if i ≤ m ) fill remaining elements from a into c
if j ≤ n ) fill remaining elements from b into c
return c
```

   (b). Forklar kvifor denne algoritmen er korrekt.

   Solution: Me fyller c frå venstre mot høgre, minste element først. Når me fyller inn element nr. k er dei $k-1$ minste elementa rydda bort. Det neste elementet må vera neste tilgjengelege element i ei av listene a eller b, fordi listene er sorterte og andre element er større enn (eller lik) det fyrste.
   (Ein kan vera meir formell, men denne vurderinga er den som eg ventar av dei fleste.)

   (c). Finn tidskompleksiteten og grunngje svaret.

   Solution: Me ser at kvar iterasjon aukar k med éin, opp til $k = m + n$ som er det siste elementet som skal fyllast inn. Dette gjev $m + n$ iterasjonar; alt anna går i konstant tid. Kompleksiteten er $O(m + n)$.

2. Redigeringsavstanden (*Edit*-avstanden eller levenshteinavstanden) mellom to strengar a og b er definert som talet på redigeringsoperasjonar ein treng for å transformera a til b eller omvendt. Tre redigeringsoperasjonar er definerte, *insert*, *delete* og *replace*.

   Fylgjande algoritme reknar ut redigeringsavstanden mellom delstrengane $s_{1..m}$ og $t_{1..n}$.

```
Algorithm editDistance(s, t, m, n):

if m = 0:    return n
if n = 0:    return m

if s[m] = t[n]: return editDistance(s, t, m-1, n-1)

return 1 + min(editDistance(s, t, m, n-1),     # Insert
               editDistance(s, t, m-1, n),     # Remove
               editDistance(s, t, m-1, n-1)    # Replace
               )
```

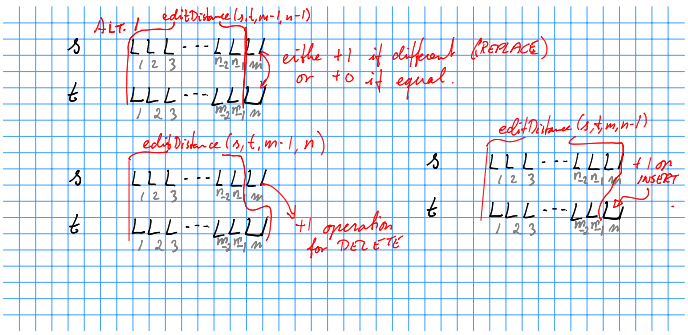
(a). Forklar kvifor denne algoritmen korrekt finn redigeringsavstanden.

Solution: Når me transformerer $s_{1..m}$ til $t_{1..n}$ (og $n, m > 0$) er der fire ulike fall:

- $s_m = t_n$, og redigeringsoperasjonane er dei same som me treng for å transformera $s_{1..m-1}$ til $t_{1..n-1}$.
- $s_m \neq t_n$ og då må me anten
  - sletta s_m og transformera $s_{1..m-1}$ til $t_{1..n}$,
  - transformera $s_{1..m}$ til $t_{1..n-1}$ og setja inn t_n, eller
  - transformera $s_{1..m-1}$ til $t_{1..n-1}$ og erstatta s_m med t_n.

Dersom $n = 0$ eller $m = 0$ er der ingen teikn i den eine strengen, og kvart teikn i den andre strengen må vera resultat av anten *insert* eller *delete*. Dette er grunnfallet som er handtert i dei to fyrste *if*-satsane.

For å skjøna rekursjonen, legg me merke til at operasjonane dannar ei fylgje, og ein operasjon må vera den siste. Det ser slik ut:



Den siste operasjonen kan gjera noko med siste teikn i den eine eller den andre strengen. Dersom dei to siste teikna er like, kan me seia at *Replace* er gratis. Totalt treng me altso alle operasjonane som trengst for å fiksa alle teikna unntatt eitt eller to av dei siste, samt (kanskje) éin operasjon til dei siste teikna.

Algoritma sjekkar då fyrst fallet der dei siste teikna er like, og reknar evt. ut ut avstanden mellom dei kortare delstrenganerekursivt.

Den siste *return*-satsen samanliknar dei tre alternativa som me lista under $s_m \neq t_n$, og vel det beste. Dermed må algoritmen vera korrekt.

(Me hoppa over provet for at det eine fallet alltid er optimalt når dei to siste teikna er like. Berre perfekte svar er venta å ta dette med.)

(b). Vurder tidskompleksiteten på algoritmen.

> **Solution:** I beste fall er strengane like og me hamnar i fallet $s_m = t_n$ kvar gong. Då får me $\min(m, n)$ rekursive kall med berre konstant tid per kall. Bestefallskompleksiteten er då $O(n)$.
>
> I verste fall er der ingen felles teikn i dei to strengane, og me treng då tre rekursive kall kvar gong. Ingen av kalla reduserer problemstorleiken med meir enn éin, og me får $\Omega(3^{\min(m,n)})$ kall, altso eksponentiell køyretid.

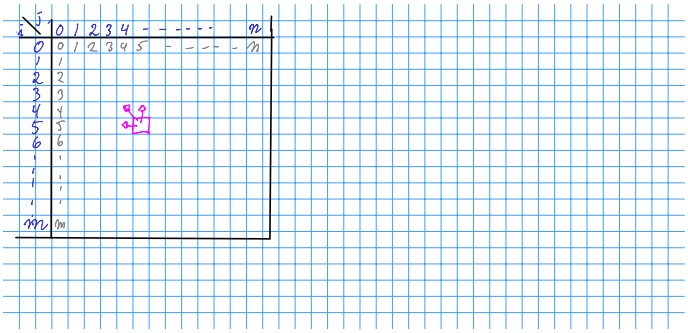
(c). Der finst ein standardteknikk som løyser problemet med vesentleg lågare tidskompleksitet (i verste fall). Kva teknikk er det tale om?

> **Solution:** Dynamisk programmering.
>
> (Memorering er òg mogleg, men det har ikkje vore nemnd på førelesing.)

(d). Gjev pseudokode for den meir effektive løysinga.

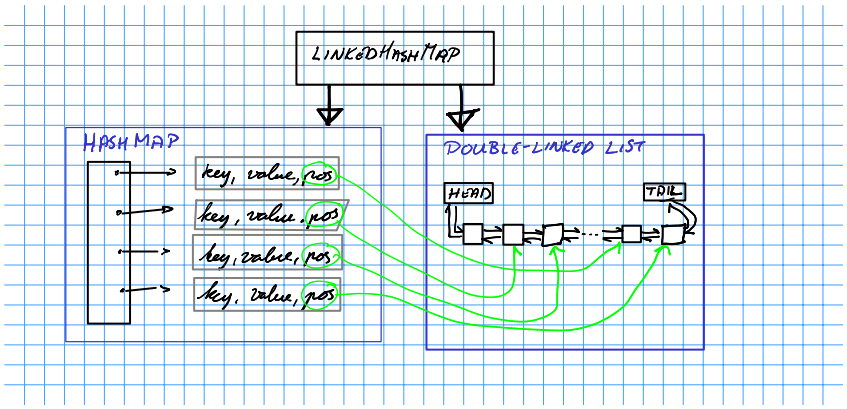
> **Solution:** Variablane er definert som i den rekursive løysinga.
>
> **Algorithm** editDistance(s, t):
>
> D is an $(m+1) \times (n+1)$ array of integers
>
> **for** $i := 0 \ldots m$: $D_{i,0} := i$
> **for** $j := 0 \ldots n$: $D_{0,j} := j$
>
> **for** $i := 0 \ldots m$:
> **for** $j := 0 \ldots n$:
> **if** $s_i = t_j$: $D_{i,j} = D_{i-1,j-1}$
> **else**:
> $D_{i,j} = 1 + \min(D_{i,j-1}, D_{i-1,j}, D_{i-1,j-1})$
> **return** $D_{m,n}$

(e). Forklar kort kvifor den nye løysinga gjev same svar som den rekursive løysinga over.

> **Solution:** Dynamisk programmering fyller alle delløysingane inn i ein tabell, som dette
>
> 
>
> Desse delproblema er dei same som vert rekna ut rekursivt i den fyrste algoritmen, men i staden for å starta på toppen med heile strengane, startar han med dei minste delproblema og fyller svara inn i ein tabell. Slik unngår han gjentakingar.
>
> Dei fyrste to *for*-løkkene reknar ut grunnfalla, som i rekursjonen. Dobbelløkka reknar ut dei større falla. Igjen kjenner me att det same min-uttrykket som i rekursjonen. Einaste skilnad er at delproblema som han treng allereie er rekna ut og kan finnast i tabellen.

3. *LinkedHashMap* er ein datastruktur som implementerer både ei lenka liste og ein spreietabell (*hash table*). I tillegg til oppslag, sletting og innsetjing i konstant tid, som i ein vanleg spreietabell, skal den lenka lista gjera det mogleg å iterera over alle elementa i den rekkjefylgja elementa vart innsetne (FIFO).

   (a). Forklar korleis datastrukturen for *LinkedHashMap* kan byggjast opp. Bruk objekt- og gjerne klassediagram til hjelp. Hugs at du må støtta *delete()* i konstant tid i neste deloppgåve.

> **Solution:** *LinkedHashMap* skal implementera to ADT-ar, *Map* og ein FIFO-iterator. Det er hensiktsmessig å bruka to underliggjande datastrukturar, ein for å støtta kvar ADT.
>
> 
>
>
> *Map* er støtta av spreidetabellen (*HashMap*) som gjev sletting, innsetjing og oppslag i konstant tid (i alle fall amortisert). (Det er ikkje naturleg å gå inn på detaljane i spreietabellen her.)
>
> FIFO-iteratoren kan implementerast som ein eller annan variant av lenka liste. For at sletting skal gå i konstant tid, er me derimot avhengig av to ting. For det fyrste treng me ein referanse frå spreietabellen for å finna den aktuelle noden i konstant tid. For det andre treng me ein dobbellenka liste for å sletta i konstant tid (når noden er funnen).
>
> Me er altso avhengige av at API-et til den underliggjande listestrukturen let oss peika direkte på nodar. Dette er tilfellet i *Positional Lists* i Java-rammeverket. Det er òg relativt enkelt å støtta det om ein implementerer lista sjølv.

   (b). Forklar korleis *delete()*-operasjonen kan gjerast i konstant tid. Hugs at du må sletta både i lista og i tabellen.

> **Solution:** Lat oss seia at *LinkedHashMap*-strukturen har ein *HashMap* H og ei liste L. Då kan me sletta objektet med nykel k som fylgjer.
>
> **Delete**(k)
> $p := H$.get(k).getPosition()
> H.delete(k)
> L.delete(p)
>
> Her har me føresett at L har same API som *positional lists* i Java, der delete(p) slettar noden i posisjon p i konstant tid. (Alternativt kan p vera ein peikar til sjølve noden, og me kallar delete() på denne noden i staden.)
>
> Alle dei tre linene tek konstant tid (amortisert) og dermed tek heile algoritmen konstant tid.

   (c). Forklar kort korleis *insert()*-operasjonen kan gjerast i konstant tid.

Solution: Det vesentlege i insert() er at me oppretter *pos*-peikaren slik at delete() fungerer som skildra over.

Insert(k, v, p)
$p := L.\mathsf{insert}(k, v)$
$H.\mathsf{append}(k, (v, p))$

Her har me føresett at append() på lista returnerer posisjonen til den nye noden. Dette er relativt enkelt å realisera når ein designar lista. Når me set inn i *Map*, set me ikkje berre inn v, men eit samansett objekt som inneheld både v og p.

Operasjonen append() legg elementet til slutten av lista, og dette går i konstant tid på lenka lister. Ogso i *Map* går innsetjing i konstant tid (amortisert).

(Ein treng ikkje problematiser verstefallskøyretida i dynamiske spreietabellar. Hovudpoenget i denne oppgåva er at ein vha. posisjonen p får konstant køyretid på delete() og ellers presenterer ryddig, systematisk og velbegrunna.)

4. Sett at me har ei usortert liste a med n element. Me skal finna det $i$te største elementet. Dersom me sorterer lista i synkande orden fyrst, kan me sjølvsagt henta ut det $i$te elementet relativt enkelt.

   (a). Kva er køyretidskompleksiteten på denne naïve løysinga inklusive sortering? Grunngje svaret kort.

Solution: Sortering i mogleg i $O(n \log n)$, ogso i verste fall. Berre med spesielle føresetnader er det mogleg å sortera raskare.

Når ein fyrst har sortert, kan ein slå opp det $i$te elementet i konstant tid (i ein *array*) eller i lineær tid (i ei liste). Uansett er dette neglisjerbart samanlikna med sorteringa.

Det tek altso $\Theta(n \log n)$.

   (b). Der er fleire moglege løysingar som gjev lågare kompleksitet (anten i verste fall eller i gjennomsnitt). Gje pseudo-kode for minst éi og forklar kvifor ho gjev korrekt svar.

Solution: (Her er det sannsynleg at studentane søkjer seg fram til algoritmar som dei ikkje har studert på førehand, og ein kan ikkje venta at dei vel den same algoritmen. Det viktigste er at dei kan vurdera algoritmen sjølv.

Den enklaste algoritmen med utgangspunkt i pensum er QuickSelect.)

Me skal finna det i største elementet i ei usortert liste. QuickSelect byggjer på QuickSort, men utnyttar det faktumet at ein ikkje treng å sortera den delen av lista der ein ikkje finn det $i$te elementet.

Lat $A_{1..n}$ vera ein usortert array, og i som brukt over.

QuickSelect$(A_{1..n}, i)$
if $n = 1$, **return** A_1.
Pick a random pivot $p \in A$
Sort (in linear time) the elements of A into three groups
  $L = \{x > p | x \in A\}$
  $M = \{x = p | x \in A\}$
  $R = \{x < p | x \in A\}$
if $k \leq |L|$ **return** QuickSelect(L, i)
else if $k \leq |L| + |M|$ **return** p
else return p QuickSelect$(R, i - |L| - |R|)$

Me ser at dei $|L|$ største elementa hamnar i L, slik at dersom $i \leq |L|$ er det berre i L me treng søkja vidare. Dei neste $|M|$ elementa hamnar i M, so dersom i ligg til venstre for det siste elementet i M, er det i M me skal leita, men alle elementa i M er like, so me kan ta einkvan. Dersom me må leita i den høgre mengda, R, skal me leita etter $(i + |L| + |M|)te$ største, fordi der er større element i dei to venstre mengdene.

(c). Vurder kompleksiteten på løysinga som du har vald.

Solution: (Her er det usannsynleg at studentane klarer å gjera ein fullstendig, formell analyse, sjølv om ein kan finna han i læreboka.)

Me kan samanlikna med QuickSort. Me veit at QuickSort har $O(n \log n)$ i gjennomsnitt (og i beste fall) fordi ein i gjennomsnitt vil om lag halvera n for kvar rekursjon og dermed få $\log n$ rekursjonar totalt. I kvar steg må ein sortera n element, som gjev $n \log n$ operasjonar totalt.

I QuickSelect har me omtrent same prinsipp, bortsett frå at me berre treng å sortera halvparten av elementa i kvar runde (ideelt). I det ideelle tilfellet endar me då opp med

$$n \cdot (1 + \frac{1}{2} + \frac{1}{4} + \cdots \frac{1}{2^n} < 2n = O(n).$$

Det er rimeleg å tru at QuickSelect er lineær i gjennomsnitt, etter same argument som for at QuickSort er $O(n \log n)$ i gjennomsnitt.

(Fullstendig prov finst i læreboka, men er ikkje gjennomgått på førelesing. Det vesentlege er at studentane ser samanhengen med og forbetringa over QuickSort.)

Vurderingsrubrikk

Rubrikken vert delt til studentane før eksamen.
Får å stå på eksamen må ein ha **minst eitt (1) poeng på kvart kriterium og minst seks (6) poeng totalt.**
Ved vurdering av validering og kompleksitetsanalyse legg me i hovudsak vekt på den oppgåva som er best løyst.
Andre oppgåver vert evt. vektlagde i tvilsfall.

Kriterium Poeng	Ikkje tilfredsstillande 0 poeng	Tilfredsstillande 1 poeng	Bra svar 2 poeng	Perfekt 3 poeng
Presentasjon	Svaret er blankt, usamanhengande eller tvetydig og gjev ikkje noko klart inntrykk av algoritmen.	Algoritmen er tilstrekkeleg formelt presentert som pseudo-kode, slik at hovudprinsippa kjem klart fram, sjølv om detaljane er skjult av usamanhengande eller upresise formuleringar.	Løysinga er overtydande og utvetydig presentert som pseudo-kode, sjølv om småfeil kan førekoma.	Lytefritt.
Validering	Ingen av dei mest vesentlege poenga kjem tydeleg fram i svaret	Dei kritiske idéane til eit bevis er korrekt identifiserte og presentasjonen er eit godt utgangspunkt for diskusjon	Valideringa er overtydande og lemnar ingen særleg tvil om at algoritmen er korrekt	Feilfritt logisk bevis
Kompleksitetsanalyse	Ingen av dei mest vesentlege poenga kjem tydeleg fram i svaret	Kritiske idéar er identifiserte og presentasjonen er eit godt utgangspunkt for diskusjon	Korrekt køyretid er funnen og analysen og lemnar ingen særleg tvil om at han er korrekt	Feilfritt logisk bevis
Algoritmisk løysing	Ingen presenterte algoritmar løyser problemet fullstendig	Nokon av oppgåvene er løyste hovudsakleg korrekte	Fleirtalet av oppgåvene er løyste hovudsakleg korrekt.	Korrekte algoritmar/-datastrukturar er gjevne på alle oppgåvene.