DM507 Algoritmer og datastrukturer

Forår 2010

Projekt, del III

Institut for matematik og datalogi
Syddansk Universitet

24. april, 2010 (let justeret 10. maj og 21. maj 2010)

Dette projekt udleveres i tre dele. Hver del har sin deadline, således at afleveringerne, og dermed arbejdet, strækkes over hele semesteret. Projektet skal besvares individuelt. Deadline for del III er torsdag den 20. maj.

Adaptive sorteringsalgoritmer

En sorteringsalgoritme kaldes *adaptiv* hvis den kører hurtigere, når inputfølgen er tæt på at være sorteret i forvejen. For at gøre dette begreb præcist, må man definere et mål for et inputs afstand fra at være sorteret. Antallet af inversioner[1] i input er et klassisk mål, med mange egenskaber som intuitivt virker fornuftige: en stigende (dvs. sorteret) følge har nul inversioner, en aftagende (dvs. omvendt sorteret) følge har det maksimale antal inversioner (nemlig $n(n-1)/2 = \Theta(n^2)$), og hvis større elementer flyttes foran mindre elementer, stiger antallet af inversioner. I dette projekt lader vi INV betegne antallet af inversioner i input.

Målet for del III af projektet er at

- Vise at InsertionSort er adaptiv mht. målet INV.
- Baseret på arbejdet i del II at udvikle en sorteringsalgoritme, FingerTreeSort, som er endnu mere adaptiv mth. INV end InsertionSort.
- Implementere InsertionSort og FingerTreeSort.
- Sammenligne disse to algoritmer, samt MergeSort (som ikke er adaptiv), med hensyn til køretid i praksis på inputs med varierende INV-mål.

[1] Se del I af projektet for definitionen af inversioner

InsertionSort

InsertionSort er beskrevet i Cormen et al., afsnit 2.1, analyseret i afsnit 2.2, og implementeret i Java som en opgave på Ugeseddel 1.

Når det j'te element x_j i inputfølgen $x_1, x_2, x_3, \ldots, x_n$ indsættes, er elementerne $x_1, x_2, x_3, \ldots, x_{j-1}$ allerede indsat og bragt i sorteret orden. Vi lader d_j være antallet af elementerne $x_1, x_2, x_3, \ldots, x_{j-1}$ som er skarpt større end x_j (specielt er d_1 altid 0).

Opgave 1

Vis at

$$\sum_{j=1}^{n} d_j = \text{INV}.$$

Opgave 2

Lad t_j være defineret som nederst på side 25 i Cormen et al. Vis at $d_j = t_j - 1$.

Opgave 3

Vis at køretiden for InsertionSort er $O(n + \text{INV})$. [Hint: brug opgave 1 og 2, samt formlen midt på side 26 i Cormen et al.]

FingerTreeSort

I ethvert binært søgetræ kan elementerne udskrives i $O(n)$ tid via et inorder gennemløb (se øverst side 288 i Cormen et al.). Dvs. at man kan sortere ved at bygge et søgetræ og derefter udskriver elementerne. Dette tager $O(n)$ tid plus tiden for at bygge træet.

FingerTreeSort ligner InsertionSort ved at elementerne x_j indsættes efter stigende j, nu blot i et balanceret søgetræ fremfor et array. Vi vil her anvende rød-sorte træer. En yderligere idé i FingerTreeSort er at vedligeholde en reference T.max til knuden med det største element i træet (dvs. den nederste knude på højrestien i træet), og foretage søgningen efter næste indsættelsespunkt derfra (i stedet for fra roden). Her betegner indsættelsespunktet det blad, som under indsættelsen bliver erstattet med en ny knude indeholdende x_j. Mere præcist indsættes hvert nyt element x_j (for $j \geq 2$) ved en procedure beskrevet ved nedenstående pseudokode. Referencen T.max til

knuden med det største element kaldes ofte en finger, heraf navnet FingerTreeSort.

```
FingerInsert(T, x_j)
    opret ny knude z indeholdende x_j
    if T.max.key ≤ x_j
        indsættelsespunkt = højre barn af T.max
        T.max = z
    else
        v = T.max
        while v.key > x_j AND v ≠ T.root
            v = v.parent
        søg på normal vis fra v efter indsættelsespunktet
    erstat bladet ved indsættelsespunktet med z
    rebalancer fra z
```

(Ovenstående pseudo-kode er en anelse mere højniveau end bogens. Detaljeniveauet kan øges med flg. bemærkninger: Koden udført ved en **if**-case kan implementeres ved "$y = T.\max$", "$T.\max = z$", efterfulgt af linierne 13 (uden **else**-keyword) og 14–17 på side 315 i Cormen et al. Koden udført ved en **else**-case kan implementeres ved **while**-løkken ovenfor (inkl. initialisering af v i linien før løkken), efterfulgt af "$x = v$", samt linierne 3–8, 11 (med **elseif** erstattet af **if**), og 12–17 på side 315 i Cormen et al..)

I algoritmen FingerTreeSort oprettes først et rød-sort træ med een knude indeholdende x_1, og $T.\max$ initialiseres til at være denne knude. Derefter indsættes $x_2, x_3, \ldots, x_n$ alle med FingerInsert. Til sidst udskrives alle elementer med et inorder gennemløb af træet.

Opgave 4

Argumenter for at træet overholder inorder efter hver indsættelse. [Hint: argumenter først for at elementerne på højrestien står i faldende orden når man går opad langs stien.]

Opgave 5

For en **else**-case, lad w være v.right.right (dvs. højre-højre barnebarnet af v) hvis $T.\max \neq T.\text{root}$, og v.right hvis $T.\max = T.\text{root}$. Argumenter for at w findes (evt. som blad, dvs. NIL), og at alle elementer i w's undertræ er skarpt større end x_j.

Opgave 6

Lad u være højeste sorte knude på højrestien som ligger under knuden w fra opgave 5. Dvs. u er w selv eller dens højre barn (pga. krav 4 på side 308 i Cormen et al.).

Bemærk at for enhver sort knude (herunder u) i et rød-sort træ overholder dets undertræ alle kravene på side 308 i Cormen et al., og er derfor selv et rød-sort træ. Bemærk også at for enhver knude (herunder v) i et rød-sort er længste sti til et blad højst to gange længere end korteste sti til et blad (pga. kravene 4 og 5 på side 308 i Cormen et al.).

Baseret på disse bemærkninger, Lemma 13.1 fra side 309 i Cormen et al., samt opgave 5, argumenter for at køretiden for FingerInsert på element x_j er $O(1 + \log(d_j + 1))$, når man *ikke* medtager tiden til rebalancering (den sidste linie i pseudokoden for FingerInsert).

Opgave 7

Man kan vise (med metoder fra Calculus) at der for alle samlinger af positive tal $y_1, y_2, y_3, \ldots, y_n$ gælder $\sum_{i=1}^{n} \log(y_i) \leq n \log(Y/n)$, hvor $Y = \sum_{i=1}^{n} y_i$. (formlen siger at for fast sum Y er udtrykket $\sum_{i=1}^{n} \log(y_i)$ størst hvis alle y_i er lige store.)

Argumenter for at FingerTreeSort kører i tid $O(n + n \log(\mathrm{INV}/n + 1))$.

[Hint: Du skal bruge oplysningen ovenfor, resultaterne fra opgave 1 og opgave 6, samt resultatet fra opgave 2 i del II af projektet].

(Man kan i øvrigt vise at dette er *bedst mulig* adaptiveness til INV for sammenligningsbaseret (jvf. Cormen et al., afsnit 8.1) sortering: en sammenligningsbaseret algoritme, der korrekt sorterer alle input af størrelse n som har et INV-tal på I eller derunder, må have en worst-case køretid (over disse input) som er $\Omega(n + n \log(I/n + 1))$.)

Opgave 8

Implementer InsertionSort (eller genbrug din kode fra ugeseddel 1) og FingerTreeSort (som i meget høj grad kan baseres på kode fra Del II).

Det er vigtigt at du under udviklingen altid tester at output er korrekt, dvs. efter en testkørsel løber output igennem og checker at det er sorteret (ved at checke at alle nabopar står i rigtig rækkefølge). Du skal i næste opgave måle køretid og sammenligne algoritmer, og det er meningsløst at sammenligne algoritmer der ikke er korrekte (det er f.eks. trivielt at lave en meget hurtig algoritme, hvis den ikke behøver løse opgaven).

Opgave 9

Du skal i denne opgave sammenligne køretiden for dine implementationer af FingerTreeSort (teoretisk set optimalt adaptiv til INV-målet), InsertionSort (adaptiv til INV-målet), samt din implementation af MergeSort (ikke adaptiv, den laver essentielt samme mængde arbejde for alle input af samme størrelse) fra Del I.

Du skal først vha. afprøvning finde en inputstørrelse n, hvor MergeSort tager ca. et sekund (den præcise tid er ikke vigtig). Dette n er fast for resten af opgaven.

For dette n skal du køre alle dine tre algoritmer på input med varierende værdi af INV. Underviseren udleverer en Java-metode `generateInput`, der som input tager n samt en parameter k, som skal ligge mellem 0 og n. Metoden leverer så et tilfældigt output med INV liggende i nærheden af nk.

Lav et program, som for hver af det tre algoritmer, og for hver af værdierne $k = 1, 2, 4, 8, \ldots, 2^i, \ldots, n$ gør flg.:

- Kalder `generateInput` for at få genereret et input (et array af `int`'s).
- Kører din $O(n \log n)$ algoritme fra Del I (fra `FastInv.java`) på en *kopi* af det genererede input (da algoritmen jo sorterer undervejs, skal den arbejde på en kopi), for at tælle den præcise værdi af INV i dette.
- Sorterer det genererede input med den pågældende sorteringsalgoritme, og måler køretiden ved at indsættte to kald til Javas biblioteksmetode `System.currentTimeMillis()`, eet lige før sorteringen går i gang, og eet lige bagefter (slå funktionaliteten af metoden op i Javas online dokumentation), hvorudfra køretiden for selve sorteringen (og kun den, dvs. uden generering af input, optælling af INV, check af output) beregnes.
- Udskriver INV og køretid.

Det kan være nødvendigt at undlade de største værdier af k for algoritmen InsertionSort, da disse vil tage for lang tid. Stop f.eks. når tiden passerer 40 sekunder.

I rapporten skal du inkludere en graf, som for alle tre algoritmer i samme koordinatsystem plotter $\log(\text{INV}/n)$ (hvor INV er den målte værdi) på førsteaksen og målt køretid på andenaksen. Du skal kommentere på resultatet (hvad kan man se, hvilken algoritme vil du foretrække hvornår?).

(Det er i øvrigt værd at bemærke at InsertionSort og FingerTreeSort *ikke* skal kende INV-tallet for at opnå den analyserede køretid. Vi bruger kun algoritmen fra `FastInv.java` fordi vi selv gerne vil kende INV-tallet for input for at kunne lave grafen.)

Formalia

Lav en rapport, som indeholder dine svar på opgaverne ovenfor. For spørgsmål med implementation skal koden være passende kommenteret, skal inkluderes i rapporten som bilag, og eventuelle ikke-trivielle aspekter af implementeringen skal diskuteres i rapportens hoveddel. Der skal afleveres rapporten i pdf-format, samt Java-implementationen som separate filer (dvs. udover deres inklusion på tryk i rapporten). Der er ingen krav til navngivning af koden her i del III.

Materialet afleveres med `aflever` kommandoen i en shell på Imadas Linux-system: Lav et directory som indeholder ovenstående filer, flyt ned i dette, og udfør kommandoen `aflever DM507`. Dette vil kopiere indholdet af dette directory til et sted i systemet, som underviseren kan tilgå. En email vil blive sendt til ens studentermail som kvittering. Man kan bruge kommandoen flere gange, senere anvendelser overskriver materialet fra tidligere anvendelser.

Aflever materialet senest:

Torsdag den 20. maj, 2010, kl. 23:59.