# DM507 Algoritmer og datastrukturer

Forår 2012

Projekt, del III

Institut for matematik og datalogi
Syddansk Universitet

29. april, 2012

Dette projekt udleveres i tre dele. Hver del har sin deadline, således at afleveringerne, og dermed arbejdet, strækkes over hele semesteret. Deadline for del III er fredag den 25. maj. Projektet skal som udgangspunkt besvares i grupper af størrelse to. Individuelle besvarelser er tilladt, men tilskyndes ikke.

## 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 (se del I af projektet for definitionen af inversioner) i input er et klassisk mål, med mange egenskaber som intuitivt virker fornuftige: en stigende (sorteret) følge har nul inversioner, en aftagende (omvendt sorteret) følge har det maksimale antal inversioner $n(n-1)/2 = \Theta(n^2)$, og hvis større elementer flyttes foran mindre elementer, stiger antallet af inversioner. I dette projekt lader vi INV betegne antallet af inversioner i input.

I Cormen et al. opgave 2-4 (side 41) på Ugeseddel 3 har vi vist at køretiden for InsertionSort er $\Theta(n + \text{INV})$. InsertionSort er altså adaptiv med hensyn til INV.

Formålet med del III af projektet er at

- Udvikle en sorteringsalgoritme, FingerTreeSort, som er endnu mere adaptiv mth. INV end InsertionSort.
- Implementere FingerTreeSort.
- Sammenligne InsertionSort, FingerTreeSort samt MergeSort (der ikke er adaptiv) med hensyn til køretid i praksis på inputs med varierende INV-værdier.

InsertionSort er implementeret på Ugeseddel 2, FingerTreeSort implementeres her i del III af projektet, mens MergeSort er implementeret i del II af projektet.

## 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.

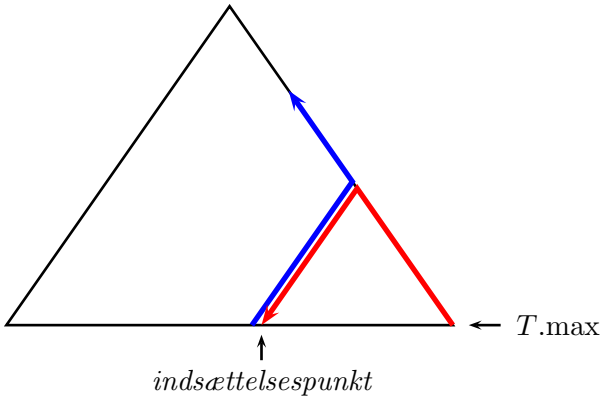
FingerTreeSort ligner InsertionSort ved at elementerne $x_1, x_2, x_3, \ldots, x_n$ i input indsættes efter stigende $j$, nu blot i et balanceret søgetræ fremfor et array. Vi vil her anvende rød-sorte træer. En yderligere idé i FingerTreeSort er at vedligeholde en reference $T$.max til knuden med det 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 x_j ≥ T.max.key
        indsættelsespunkt = højre barn af T.max
        T.max = z
    else
        v = T.max
        while x_j < v.key 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.




Figuren ovenfor illustrerer et eksempel på FingerInsert. Den røde linie illustrerer søgning, den blå linie rebalancering.

Selve algoritmen FingerTreeSort fungerer således: Først oprettes 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 1

Argumenter for at træet overholder inorder efter en indsættelse.

[Hint: argumenter først for at elementerne på højrestien står i faldende orden når man går opad langs den.]

□

Lad $s_i$ være længden af den sti, som gennemløbes i søgeprocessen under indsættelsen af $x_i$, som illustreret med den røde linie i figuren ovenfor. Dvs. $s_i$ er lig antal iterationer **while**-løkken i FingerInsert (søgning opad) plus længden af den sti som gennemløbes i linien "søg på normal vis fra $v$ efter indsættelsespunktet" (søgning nedad).

Man kan vise følgende (beviset vil blive givet ved en forelæsning):

$$\sum_{i=2}^{n} s_i = O(n + n \log(\mathrm{INV}/n + 1)).$$

### Opgave 2

Argumenter for, at FingerTreeSort kører i tid

$$O(n + n \log(\text{INV}/n + 1))$$

[Hint: du skal opgøre det samlede arbejde for alle dele af algoritmen, herunder det samlede arbejde for de $n$ indsættelser. Resultatet fra opgave 2 v) i del II af projektet vil skulle bruges under det sidstnævnte.]

□

Man kan i øvrigt vise, at dette er *bedst mulig* adaptiveness til INV for sammenligningsbaseret (se 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 (for disse input) som er $\Omega(n + n \log(I/n + 1))$.

### Opgave 3

Angiv den asymptotiske køretid for InsertionSort, FingerTreeSort og MergeSort for input af størrelse $n$, når antal inversioner INV i input er henholdsvis 0, $n$, $n^{1.5}$ og $n^2/4$.

□

### Opgave 4

Implementer FingerTreeSort (som i meget høj grad kan baseres på kode fra Del II), og find din implementationer af InsertionSort og MergeSort fra henholdsvis Ugeseddel 2 og del I af projektet frem.

Det er vigtigt at du under udviklingen tester om 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 5

Du skal i denne opgave sammenligne køretiden for InsertionSort (adaptiv til INV-målet) implementeret på Ugeseddel 2, FingerTreeSort (teoretisk set optimalt adaptiv til INV-målet) implementeret her i del III, samt MergeSort (ikke adaptiv, da den laver essentielt samme mængde arbejde for alle input af samme størrelse) implementeret i 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. På kursets hjemmeside findes 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 for hver af de tre algoritmer et program, som 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 $A$ af `int`'s).
- Kører din $O(n \log n)$ metode fra `FastInv.java` fra Del I af projektet på en *kopi* af $A$ (da algoritmen jo sorterer undervejs, skal den arbejde på en kopi for ikke at ændre indholdet af $A$), for at tælle den præcise værdi af INV i $A$.
- Sorterer $A$ med den pågældende sorteringsalgoritme, og måler køretiden ved at indsættte to kald til metoden `System.currentTimeMillis()` fra Javas bibliotek, eet lige før sorteringen går i gang, og eet lige bagefter (slå funktionaliteten af metoden op i Javas online dokumentation, husk også at du har brugt den på Ugeseddel 2), 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 ca. 30 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. Husk at skrive navnene på personerne i gruppen på forsiden af rapporten.

Materialet afleveres i Blackboard med værktøjet "SDU Assignment" (ikke at forveksle med "Assignment hand in", som er et andet afleveringsværktøj i Blackboard). Det kan findes under "Tools" i menuen i kursussiden i Blackboard. Menuen findes ved at klikke på det lille "dobbelt-firkant"-ikon i øverste halvdel af venstre kant af kursussiden i Blackboard (om nødvendigt maksimer det fremkomne vindue).

Aflever materialet senest:

**Fredag den 25. maj, 2012, kl. 23:59.**