HÖGSKOLAN I GÄVLE

Handledning till kurslitteraturen

Weiss, M.A.: *Data Structures and Problem Solving Using Java*.

Hänvisningar till avsnitt i kurslitteraturen markeras som vanligt med blå markeringsfärg. Läs också de kompletterande föreläsningsanteckningarna.

Tips: På Blackboard ('External links') finns en länk till en demo-applet som illustrerar olika sorteringar.

Föreläsning 11: Divide and conquer-sortering

ShellSort och *CombSort* är förbättringar jämfört med de enklaste sorteringarna. Som tur är går det att komma längre, till exempel med *MergeSort* och *QuickSort* som är exempel på en familj av algoritmer som kallas ”Divide and conquer” – söndra och härska.

Det finns också en annan sorteringsalgoritm vars prestanda är jämförbar med MergeSort och QuickSort, nämligen *HeapSort*. Den algoritmen ingår inte i kursen, men om du är intresserad så beskrivs den i avsnitt 21.5.

MergeSort

Avsnitt 8.5 handlar om MergeSort, som är den D&C-sortering som kanske är enklast att förstå. En skiss av algoritmen ges i inledningen.

Ett fundamentalt steg i MergeSort är att använda operationen *merge*, som kan översättas med ”sammansmälta”. Operationen tar två sorterade listor och sammansmälter dem till en enda sorterad lista, genom att plocka ett element i taget från någon av de mindre listorna.

Obs! Det är lätt att blanda ihop Merge-metoden med själva MergeSort-operationen. Gör inte det! *Merge* är en operation som sammansmälter två sorterade sublistor till en enda sorterad lista, och är alltså bara en del av MergeSort.

För att MergeSort ska bli tillräckligt snabbt måste merge-operationen kunna göras på *linjär* tid. Går det? Svaret är ja, om vi använder en ny array för att lägga den sammansmälta listan i. Se avsnitt 8.5.1 för en beskrivning av hur det går till. Studera bilderna noga, och jämför med exemplet i föreläsnings-anteckningarna!

I figur 8.8 i avsnitt 8.5.2 finns ett förslag på en Java-implementation av MergeSort. En analys av tidsåtgången för MergeSort görs i föreläsningsanteckningarna. Analysen använder sig av algoritmens rekursionsträd för att visa att tidsåtgången i både *medelfallet* och *värsta fallet* är ***O****(n*log(n))*. Idén är ganska enkel: Eftersom delningen av den lista som ska sorteras alltid delar listan i lika stora[1] delar så kommer rekursionsträdet att vara nästan perfekt balanserat. I och med att rekursionen avbryts när en lista har färre än två element så kommer höjden av trädet att vara *log(n)*, För varje nivå i rekursionsträdet så kommer högst N jämförelser att göras i både medelfallet och värsta fallet. Totalt måste alltså *log(n)*n* jämförelser göras, vilket ger tidsåtgången ***O****(n*log(n))*.

[1] Om listan har ojämnt antal element så blir förstås ena delen ett element kortare, men balansen mellan listorna är i alla fall så bra som möjligt.

För att försäkra oss om att det blir max *n* jämförelser per nivå så kan vi lite förenklat resonera så här:

Om vi till exempel studerar den nivå i rekursionsträdet som har 8 sublistor, så kommer varje sublista att ha (ungefär) *n/8* element. Dessa 8 sublistor ska sammansmältas till 4 listor. För att sammansmälta två listor med vardera *n/8* element krävs ungefär *n/4* jämförelser. Eftersom 4 sådana sammansmältningar görs på den aktuella nivån blir antalet jämförelser totalt *4*n/4 = n* st. Ett liknande resonemang kan användas för alla nivåer i rekursionsträdet.

QuickSort

Grundidén för MergeSort, framför allt en variant av merge-operationen, används ofta för *extern* sortering. För *intern* sortering så är däremot MergeSort inte lika vanlig, eftersom den kräver extra minne. Istället används ofta QuickSort, som beskrivs i avsnitt 8.6. QuickSort anses vara en av de snabbaste kända sorteringsalgoritmerna, förutsatt att den implementeras på ett bra sätt. Den har samma tidsåtgång som MergeSort i genomsnittsfallet, men i värsta fallet är tidsåtgången faktiskt $\boldsymbol{O(n^2)}$. Som tur är så är det extremt osannolikt att det inträffar om algoritmen implementeras på rätt sätt.

I likhet med MergeSort är grundidén att ”söndra och härska”, men det stora jobbet i QuickSort görs i *uppdelningssteget* istället för i *hopslagningssteget* som i MergeSort. En enkel beskrivning av QuickSort-algoritmen finns i avsnitt 8.6.1. De viktigaste stegen i QuickSort är steg 2 och 3. I steg 2 väljs ett *pivotelement*[2] som sedan styr uppdelningen (partitioneringen) i steg 3. Det ger upphov till två viktiga frågor, som besvaras längre fram:

1. Hur väljs pivotelementet på ett bra sätt?
2. Hur implementeras partitioneringen på ett effektivt (linjär tid) sätt?

Uppdelningen i steg 3 görs på ett sådant sätt att alla element som är *mindre än eller lika* med pivotelementet hamnar i en lista, medan alla element som är *större än eller lika med* pivotelementet hamnar i en annan lista. Matematiskt kan det beskrivas på följande sätt:

$L = \{x \in S\text{-}\{v\}: x <= v\}$
$R = \{x \in S\text{-}\{v\}: x >= v\}$

Listan $S\text{-}\{v\}$ betecknar alltså hela den osorterade listan förutom pivotelementet v. Observera att det finns en tvetydighet här: Hur gör vi med dubbletter, alltså element som råkar vara lika med pivotelementet? Ska de hamna i L eller R? Detta diskuteras vidare i avsnitt 8.6.5.

När uppdelningen väl är gjord kan L respektive R sorteras var för sig genom två rekursiva anrop, och därefter kan de båda sorterade sublistornalistorna enkelt slås ihop (med pivotelementet mellan sig) till en sorterad lista. Studera figur 8.10 som illustrerar hur QuickSort fungerar. Figuren visar vad som händer på den översta nivån i rekursionsträdet. Tänk på att det rekursiva anropet för den vänstra mängden (”small items”) ger upphov till ett antal nya rekursiva anrop, som i sin tur ger upphov till nya rekursiva anrop (eller ett basfall), och så vidare. Alla dessa rekursiva anrop kommer att ha avslutats innan det rekursiva anropet för den högra mängden (”large items”) görs.

Observera att inget merge-steg behövs i i QuickSort. Alla element i den sorterade L är garanterat mindre än pivotelementet, som i sin tur är garanterat är mindre än alla element i den sorterade R. Detta garanteras av partitioneringen i steg 3. Alltså är det enkelt att slå ihop de båda listorna till en lista. Att algoritmen

[2] Uttalas ”pivåelement”.

alltid ger ett korrekt resultat visas också i avsnittet. Studera resonemanget noga!

Analys av QuickSort

I avsnitt 8.6.2 görs en noggrann analys av prestanda för QuickSort i bästa fallet, sämsta fallet och genomsnittsfallet. Till att börja med kan vi konstatera att det som garanterade MergeSorts snabbhet var att algoritmen alltid delar upp den osorterade listan i två lika stora delar. Tyvärr kan detta inte garanteras av QuickSort, eftersom det alltid finns en risk att partitioneringen görs så att den ena sublistan blir väldigt liten i förhållande till den andra. I värsta fall blir den ena listan helt tom, medan den andra innehåller alla element förutom pivotelementet. Det fallet inträffar om pivotelementet råkar vara det största eller det minsta elementet i den mängd som ska sorteras. Rekursionsträdet kommer då att påminna om ett binärt sökträd som har urartat till en länkad lista, vilket ger ett *linjärt* djup i trädet. Se föreläsnings-anteckningarna för en utveckling av detta resonemang.

Kursboken använder ett annat resonemang för att visa tidsåtgången. Resonemanget för *bästa fallet* påminner om resonemanget för MergeSort: Partitioneringen delar upp den osorterade listan i två lika stora delar, vilket ger ett rekursionsträd med djupet *log(N)*.

För att visa tidsåtgången i värsta fallet används en så kallad *teleskopsumma*, en summa där många termer tar ut varandra. Först konstateras att exekveringstiden *T(N)* för QuickSort för sortering av *N* element i värsta fallet blir följande:

tiden för sortering av den sublista som innehåller *N-1* element, det vill säga *T(N-1)*
+
tiden för sortering av den sublista som innehåller 0 element, det vill säga *1*
+
tiden för partitionering av *N* element, det vill säga *N*

För enkelhetens skull bortses från den konstanta mittersta termen. Detta leder fram till ekvation **8.1** som sedan kan använda för att härleda ekvationerna i **8.2**. Om nu *T(N-1)* i översta ekvationen ersätts med uttrycket *T(N-2)+(N-1)* i näst översta ekvationen, och så vidare, så får vi följande uttryck för *T(N)*:

$$
\begin{aligned}
&T(N) = T(N-1) + N = \\
&= T(N-2) + (N-1) + N = \\
&= T(N-3) + (N-2) + (N-1) + N = \\
&\ldots \\
&= T(1) + 2 + 3 + \ldots + (N-2) + (N-1) + N = \\
&= 1 + 2 + 3 + \ldots + (N-2) + (N-1) + N
\end{aligned}
$$

Se där! Vår gamle vän, den aritmetiska summan, dyker upp igen. Vi får alltså att $T(N) = \frac{1}{2} * N(N+1) = \boldsymbol{O}(N^2)$. Se ekvation **8.3**!

För att visa tidsåtgången i genomsnittsfallet så används ett liknande, men lite mer komplicerat resonemang. Se ekvationerna **8.4** – **8.9**! Som tidigare har nämnts så har QuickSort tidsåtgången $\boldsymbol{O}(N*\log(N))$ i genomsnittsfallet. Det är samma som för MergeSort. Men eftersom inget extra minne behövs (som i MergeSort) och partitioneringen är lättare att optimera än merge-operationen, så är QuickSort för det mesta det bästa valet, trots att dess tidsåtgång i värsta fallet är sämre än MergeSorts. Det gäller då bara att välja pivotelement på ett bra sätt.

Avsnitt 8.6.3 diskuterar olika strategier för valet av pivotelement. Studera avsnittet noggrant!

I 8.6.4 diskuteras hur partitioneringsoperationen kan implementeras, och i 8.6.6 diskuteras en möjlig optimering om ”median of three” används som strategi för valet av pivotelement. Som synes går det att skapa en linjär partitioneringsoperation som kan arbeta ”direkt på plats”, det vill säga inte kräver extra minne. Figurerna 8.11 – 8.17 illustrerar hur partitioneringen fungerar.

En sista optimering diskuteras i avsnitt 8.6.7: Vi har tidigare konstaterat att den extra tidsåtgång (”overhead”) som orsakas av rekursiva anrop kan bli stor i förhållande till själva arbetet som utförs i den anropade metoden. Det innebär att de rekursiva sorteringsalgoritmerna blir förhållandevis ineffektiva jämfört med de enkla sorteringarna om de osorterade listorna är relativt små. Det skulle kunna utnyttjas i QuickSort på följande sätt: Om längden på den lista som ska sorteras är mindre än något ”cutoff”, till exempel 20, så används inte QuickSort. Istället används exempelvis vanlig insticksortering. Det är bara om längden är större än 20 som QuickSort anropar sig själv rekursivt. Vilket ”cutoff” som är bäst beror på vilken sorts element som ska sorteras, och vilken fysisk maskin som sorteringen kör på, men värden mellan 5 och 20 fungerar i allmänhet bra. De idéer som har diskuterats i 8.6.3 – 8.6.7 är alla implementerade i den Java-implementation av QuickSort som presenteras i avsnitt 8.6.8. Koden finns i figur 8.21. Som vanligt används en publik ”driver routine” som anropar en privat rekursiv hjälpmetoden.

I avsnitt 8.8 visas ett viktigt teoretiskt resultat, nämligen sats 8.3. Satsen säger att den *nedre gränsen* för tidsåtgången för alla algoritmer som bygger på jämförelser är $\boldsymbol{\Omega}$ *(N*log(N))*. Även om vi skulle kunna konstruera den ”ideala” sorteringsalgoritmen så skulle den alltså alltid ha tidsåtgången $\boldsymbol{O}$*(N*log(N))* i värsta fallet – det går alltid att hitta en ”värsta tänkbara” indatasekvens som ger en exekveringstid som är $\boldsymbol{O}$*(N*log(N))*. Se föreläsningsanteckningarna för en enkel sammanfattning av beviset.

Avsnitt 8.7, som handlar om algoritmen QuickSelect, kan hoppas över.