Algoritmer och datastrukturer I, 5p
2005-12-13, mbm
N-institutionen, avdelningen för datavetenskap

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 10: Sortering

Kapitel 8 handlar om det viktiga området *sortering*. Börja med att läsa introduktionen till kapitlet, för att få en översikt över vad som gås igenom. Notera distinktionen som görs mellan *intern sortering* och *extern sortering*:

- Intern sortering ("internal sorting") utförs i primärminnet. Det innebär att relativt få element, upp till några miljoner, kan sorteras. I kursen behandlas endast intern sortering.
- Extern sortering ("external sorting") görs på större datamängder, och då måste sekundärminnet utnyttjas. Detta ingår inte i kursen, men beskrivs närmare i avsnitt 21.6 om du är intresserad.

I avsnitt 8.1 motiveras varför sortering är en så viktig och grundläggande operation, och varför det är viktigt att kunna skapa så snabba sorteringsalgoritmer som möjligt. Ett exempel ges i kursboken: Undersökning om en datamängd innehåller dubbletter. Den föreslagna algoritmen illustrerar det faktum att många algoritmer kan göras snabbare genom att först sortera datamängden. Se figur 8.1. Den totala exekveringstiden, även inklusive sorteringen, kan då bli mindre än om en algoritm används på en osorterad mängd.

Avsnitt 8.2 diskuterar de förutsättningar som antas gälla i resten av kapitlet:

- Den sorts datastruktur som vi vill sortera är *listor*, främst arraybaserade listor.
- Alla studerade sorteringsalgoritmer bygger på *jämförelser*[1] mellan två element.

Viktig fråga: Måste alla sorteringsalgoritmer bygga på jämförelser av element?

Nej, i vissa specialfall kan man hitta betydligt bättre sorteringar än jämförelsebaserade sorteringar, som är relativt långsamma. Därför är det alltid viktigt att tänka till innan man väljer sorteringsalgoritm. Här är två exempel:

1. Använd *inte* en jämförelsebaserad algoritm om adressen för ett element kan *beräknas direkt* utifrån elementets värde. Ett exempel på detta är om vi vill sortera en arraybaserad kortlek. I det fallet vet vi från början exakt hur många element som finns, vi vet hur elementen ser ut och vi vet att alla element är olika. Vi kan då tilldela varje kort en egen plats i arrayen, och beräkna den positionens index direkt från kortets värde[2]. Sorteringstiden blir då *linjär*, eftersom det räcker med ett enda svep genom den osorterade kortleken för att lägga varje kort på rätt plats, men vi behöver å andra sidan en till array att lägga den sorterade leken i.
2. Om du vet att listan alltid är sorterad, till exempel om du har en SortedList där all insättning sker i sorterad ordning, så behöver den förstås aldrig sorteras. Sorterad insättning sker på *linjär* tid.

I allmänhet vet vi inte mer om den osorterade datamängden än att elementen kan rangordnas inbördes, så

[1] I Java-sammanhang kan vi tänka oss att de element som ska sorteras implementerar interfacet *Comparable*.
[2] Exempel: Klöver = 0, ruter = 1, spader = 2, hjärter = 3. Ess = 0, två = 1, tre = 2, o.s.v. Kortet "Ruter tre" hamnar då på position 1*13+2 = 15 i arrayen.

fall 1 är oftast inte tillämpligt. Fall 2 gäller inte heller alltid, eftersom många applikationer kräver att element kan flyttas runt i listan efter att den har skapats, och då kan den sorterade ordningen inte längre garanteras. I dessa fall måste vi använda oss av någon jämförelsebaserad (comparison-based) sorteringsalgoritm. Några av de enklaste sorteringarna är *bubbelsortering*, *insticksortering* och *urvalssortering*.

Insticksortering

I avsnitt 8.3 presenteras och analyseras en enkel sorteringsalgoritm: insticksortering. Se föreläsningsanteckningarna för en översiktlig beskrivning av algoritmen, både för sortering av länkade listor och arraybaserade listor. Algoritmen delar upp listan i en sorterad och en osorterad del. En Java-implementation presenteras i figur 8.2. Se figur 8.3 och 8.4 för en illustration av hur algoritmen fungerar.

Insticksortering är en enkel algoritm som duger till sortering av små datamängder, men eftersom tidsåtgången är kvadratisk (vilket vi strax ska visa) så är den för långsam för stora datamängder. Undantaget är om datamängden råkar vara sorterad eller nästan sorterad från början, i det fallet är insticksortering tidsåtgång linjär eller nästan linjär.

För att kunna göra en noggrannare teoretisk analys av insticksortering tidsåtgång behöver vi använda oss av lite matematik. Läs föreläsningsanteckningarna om formeln för den *aritmetiska summan*. Jämför diskussionen i anteckningarna med kursbokens diskussion, som resonerar om bästa fallet, värsta fallet respektive genomsnittsfallet:

1. I *bästa fallet* så blir testet i den inre loopen falskt på en gång, medan den yttre loopen alltid körs *N-1* gånger, vilket inträffar om listan råkar vara sorterad från början. Alltså: $\boldsymbol{O}(N)$
2. I *värsta fallet* så utförs den inre loopen *p* gånger för varje värde på index *p*, som går från 1 till *N-1*. Med hjälp av den aritmetiska summan kan vi komma fram till att $\frac{1}{2}*(N^2-N) = \boldsymbol{O}(N^2)$ jämförelser, och ungefär lika många tilldelningar, måste göras. Alltså: $\boldsymbol{O}(N^2)$
3. I *genomsnittsfallet* får vi också kvadratisk tidsåtgång. Detta visas genom att resonera om det genomsnittliga antalet *inversioner*[3] i en slumpmässigt ordnad array. Vi kan konstatera att ett platsbyte mellan två intilliggande element som mest kan upphäva *en* inversion. Sats 8.1 visar att antalet inversioner i en slumpmässigt ordnad lista som inte har dubbletter i genomsnitt är $(N^2-N)/4$.

Med samma resonemang kan vi fastställa en *teoretisk nedre gräns* - $\boldsymbol{\Omega}(N^2)$ - för tidsåtgången för alla sorteringar som bygger på platsbyten mellan intilliggande element, till exempel insticksortering och bubbelsortering. Se sats 8.2!

Shell-sortering

För att hitta en bättre sorteringsalgoritm måste vi bli mer effektiva. Vi behöver kunna byta plats på element som inte ligger intill varandra för att på så sätt upphäva fler inversioner per platsbyte. En sådan algoritm är *Shell-sortering*, som beskrivs i avsnitt 8.4.

Shell-sortering upptäcktes redan 1959, men är fortfarande en av de bästa "enkla" sorteringsalgoritmerna. Så sent som 1991 upptäcktes *CombSort*[4], som bygger på samma idé som Shell-sortering, men med bubbelsortering i botten. CombSort har lika bra eller kanske till och med bättre prestanda än Shell-sort.

Shell-sortering bygger på vanlig insticksortering. Idén är att först "grovsortera" listan som ska sorteras

[3] En *inversion* är ett par av element som är i inbördes oordning.
[4] Se Njord/Sandmark: *Turbovägen till Pascal.* Studentlitteratur.

genom att dela upp den i ett antal sublistor som består av alla element som finns på ett visst avstånd[5] från varandra i arrayen. Varje sublista sorteras sedan med insticksssortering. På detta sätt flyttas varje element lite närmare den slutliga positionen. När "grovsorteringen" är klar görs en ny grovsortering, men nu på färre sublistor. Slutligen görs en insticksssortering på hela listan. Den kommer att gå fort eftersom de flesta elementen ligger "nästan rätt". Se figur 8.5 som visar det resultat vi får om vi använder inkrementsekvensen 5-3-1. Jämför med exemplet i Föreläsningsanteckningarna!

Shell föreslog inkrementsekvensen N/2, N/4, …, 2, 1. Senare har man med praktiska experiment visat att det finns betydligt bättre inkrementsekvenser. I avsnitt 8.4.1 diskuteras tidsåtgången för Shell-sortering. Det visar sig att den är mycket svår att analysera teoretiskt, utom för några enkla inkrementsekvenser. Se resonemanget i kursboken om den sekvens som Shell själv föreslog: $\boldsymbol{O}(N^2)$ i värsta fallet, men $\boldsymbol{O}(N^{3/2})$ i genomsnitt.

Med bättre inkrementsekvenser kan vi få $\boldsymbol{O}(N^{3/2})$ även i värsta fallet. Se tabellen i figur 8.6 för en jämförelse mellan olika sekvenser. Som synes får vi enligt experimenten god prestanda med en inkrementsekvens där kvoten mellan inkrementen är ungefär 2,2. Ingen har hittills lyckats förklara varför.

Slutsatser? Vi kan konstatera att Shell-sortering är ett bra exempel på en enkel algoritm som är svår att analysera. Vi har också sett att Shell-sortering är en förbättring jämfört med insticksssortering och de andra enkla algoritmerna, men för riktigt stora listor räcker den fortfarande inte till. Som tur är går det att komma längre, till exempel med *MergeSort*, *QuickSort* och *HeapSort*.

[5] Avståndet bestäms av ett *inkrement* ("gap") som successivt minskas tills inkrementet är 1, vilket motsvarar den avslutande insticksssorteringen av hela listan.