Laboration 1

Henrik Johansson ehioja-0,
Simon Johansson johsim-0

Merge Sort

I denna laboration modifierar vi den existerande algoritmen *Merge Sort* genom att ändra sättet som algoritmen delar upp listan som skickas in till funktionen.

Algoritmen *Merge Sort* i sitt klassiska utförande fungerar på så sätt att den genom rekursiva anrop delar upp en lista i flera små sub-listor tills det endast finns ett element i vardera sub-lista. Dessa sub-listors innehåll skickas sedan in par vis till en funktion som jämför dessa element och lägger till dem i en lista som tillslut returneras som en sorterad lista. Detta tillväga gångssätt kallas för “divide & conquer”, alltså att dela upp problemet till minsta beståndsdel för att sedan bygga upp allt igen (Bild 1).

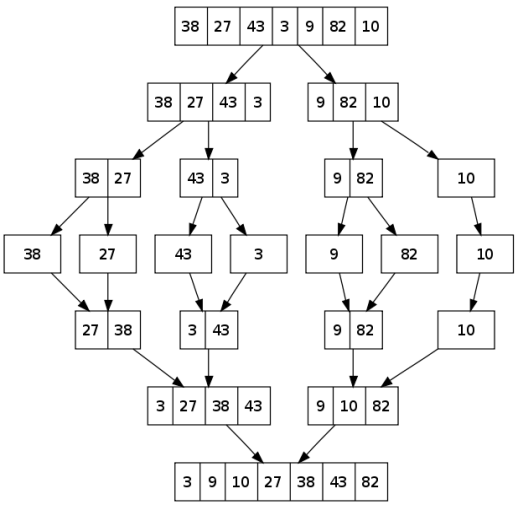

Bild 1. Vanlig Merge Sort uppdelning.

Modifierad Merge Sort

Den huvudsakliga skillnaden mellan den vanliga *Merge Sort*-funktionen och vår modifierade version är att istället för att dela upp listan genom att hela tiden dela med två så kan vi nu istället bestämma i vilken takt vi ska dela listan, t.ex. kan vi säga att funktionen ska dela listan i fem sub-listor för varje rekursivt anrop istället för två.

Vår algoritm består av 2 funktioner, “msort” och “merge”. “msort” tar hand om uppdelningen av listan, medans “merge” lägger ihop elementen för att forma en sorterad lista.

1. Listan som skickas in till funktionen “msort” delas upp i *m*-stycken sublistor genom att stega igenom listan med hjälp av två index variabler som anger start och stop element för varje sublista.

2. Varje sub-lista skickas sedan vidare rekursivt till “msort” tills endast ett element existerar i varje sublista.

3. Funktionen returnerar nu hela call-stacken och anropar “merge” funktionen med varje sub-lista.

4. “Merge” loopar igenom och jämför varje element var för sig och sätter in dem i en resultat lista som kommer returneras som våran sorterade lista.

Analys av algoritmen

Vi testade algoritmen genom att variera listans storlek samt antal sub-listor den ska delas in i.

storlek \ sub-listor	m = 2	m = 10	m = 100	m = 1000	m = 10000
n = 1000	0,009	0,453*	0,023725	0,112	0,1115
n = 10000	0.11125	0,10525	0,34525	1,99	11,1075
n = 100000	1,31975	1,46	4,8735	25,7525	222,9425

Tabell 1. Tabell över olika indata. Resultaten är i sekunder.

** (Felskrivning, ska vara något i stil med 0.006999. Vi har inte kunnat replikera den topp som uppstår vid m=10)*

Här kan man se att när *m* är lika med två beter sig algoritmen som en vanlig *Merge Sort*-funktion och har en tidskomplexitet på $O(n\ log\ (n))$. När vi sedan ökar *m* börjar det ta längre tid för den att sortera listan. Teoretisk sett borde den bli snabbare då vi ökar *m* för den gör färre steg för att dela upp listan i mindre delar. Problemet är dock att man räknar inte med *Merge*-delen som fortfarande bara tar emot två listor och jämför varje enskilda element i dem. Detta leder till att medans *m* beger sig på mot *n* kommer den få en tidskomplexitet på $O(n^2)$. När det kommer till sista skedet måste vår algoritm slå ihop två listor som tillsammans utgör ursprungs listan, detta leder till att algoritmen får en minnesanvänding på $O(n)$.

Första gången vi implementerade och testkörde vår algoritm märkte vi att den var avsevärt mycket långsammare än vanlig *Merge Sort*. Detta berodde på att vi insåg inte hur stor skillnad det blir att skriva algoritmen in-place jämför med att skriva not-in-place. Istället för att endast hantera en array, skapade vi en nya array för varje rekursivt anrop och tog bort element från ursprungs listan. Algoritmen fungerade bra när det kom till att sortera relativt få element, men när element antalet ökade var algoritmen tvungen att skapa så många ny listor och flytta runt på alla dessa element att algoritmen ej var tidseffektiv. Efter att ha insett detta klavertramp implementerade vi vår algoritm som in-place istället. Nu kör vi allt med en array och utan att behöva ta bort några element.

Tidskonsumtion

Då m är lika med 2 kommer algoritmen ha en körtid på $O(n\ log_2(n))$, vilket är samma som den vanliga merge sort algoritmen (delar helt enkelt in problemet i två sub-problem). När vi nu ökar $m > 2$ så tar algoritmen $O(n\ log_m(n))$ tid på sig (delar problemet i m-problem). Vi kan se detta genom att göra ett "recursion tree" som kommer innehålla $lg_m\ n\ +\ 1$ nivåer, då vi delar den i m stycken sublistor. För varje nivå kommer det kosta oss $c \cdot n$. Detta leder till att vi får ekvationen $c \cdot n\ (lg_m\ n\ +\ 1)$ vilket kan skrivas om som $O(n\ log_m(n))$. Vilket visar att algoritmen bör gå snabbare desto större m är.

Det merge funktionen gör är att istället för att slå ihop 2 stycken element så slår den ihop m stycken element som kommer bilda lösningen till ursprungslistan. Detta görs tills vi i slutändan har en sorterad lista. Det gör att den kommer ta $O(m \cdot n)$ tid på sig. Problemet är att då merge funktionen har en körtid på $O(m \cdot n)$ och då m närma sig n kommer den ta $O(n^2)$ tid. Detta leder till att algoritmen blir betydligt långsammare vid större m.

Divide (D): Delar upp listan i m delar där det tar $D(n)$ tid att dela upp den. Då får vi $D(n) \cdot m$.
Conquer: Vi löser m subproblem där varje tar $T(n/m)$ tid att lösa. Då får vi $m \cdot T(n/m)$.
Combine (C): Vi slår ihop m stycken listor, vilket ger $m \cdot n$.

Total: $T(n)\ =\ m \cdot T(n/m)\ +\ D(n \cdot m)\ + C(n \cdot m)$

Detta leder till:

$T(n)\ =\ m \cdot T(n/m)\ + D(n \cdot m) + C(n \cdot m)$

Då vi har $log_m(n)\ + 1$ nivåer där varje kostar $(n \cdot m)$ får vi: $(n \cdot m)\ \cdot (log_m n\ +\ 1)\ \Rightarrow$
$=\ (n \cdot m) \cdot log_m n\ + (n \cdot m)$
$=\ O((n \cdot m) \cdot\ log_m(n))$
Här kan vi se att när $m \rightarrow n$ kommer $m \cdot n$ att bli n^2