Sortering

Jonathan Nilsson Cullgert

Hösten 2023

Introduktion

I den förra uppgiften tog vi reda på att det går mycket snabbare att jobba med data ifall den är sorterad när man ska använda den. Det är då rimligt att ställa sig frågan om hur man sorterar, och hur mycket det kostar att göra det. I denna uppgift jag alltså undersöka tre olika sorteringsalgoritmer för att se vilken av dessa som är snabbast. Det kan vara så att dessa algoritmer är bra på olika saker och därför lämpligare att implementera i särskilda situationer. I denna uppgift kommer det bara användas data av en fixerad storlek, arrayerna i detta fall kommer alltså vara statiska.

Selection Sort

Den första sorteringsalgoritmen är förmodligen den enklaste, och iallafall det sätt som var det första jag tänkte att man kunde sortera data med. Denna algoritm går ut på att man jämför varje element i arrayen med alla andra element och på så sätt hittar man vilken som är minst och kan därmed sortera. Det sättet man programmerar detta på är att man börjar vid första elementet, jämför med alla andra element i arrayen. Om man hittar något element som är mindre än det man börjar på när man har gjort alla jämförelser, kommer man sedan att byta plats på dessa två element. Denna process återupprepas tills det att alla element i arrayen har hamnat på rätt plats.

```
private static void selectionSort(int[] array) {
                for(int i = 0; i < array.length - 1; i++) {
                        int min = i;
                        for(int j = i + 1; j < array.length; j++) {
                                if(array[min] > array[j]) {
                                        min = j;
                                }
                        }
            swap(array, i, j)
```

Om man har en array av n storlek kommer det uppenbarligen vara en väldigt ineffektiv algoritm vid stora datamängder. Anledningen till detta är då man jämför n element n gånger. Med andra ord kommer tidskomplexiteten av denna sorteringsalgoritm bli $O(n^2)$. En fördel med selection sort förutom att den är väldigt simpel att förstå och implementera, är att den inte tar upp mycket minne. Förutom arrayen och två index i for-loopar används bara en temporär variabel för att kunna göra substitution i arrayen.

Insertion Sort

Nästa algoritm är väldigt lik selection sort men ändå mer komplicerad, och kanske inte helt logisk första gången man tänker på det. I selection sort kollar man alltid på alla element som finns kvar i arrayen och på så sätt sorterar man. I insertion sort gör man jämförelser bakåt i arrayen och jämför alltid med det som redan är sorterat. Jag implementerade detta med en for-loop som rör sig framåt i arrayen och en while-loop som kollar om elementen bakom den nuvarande platsen är mindre eller inte.

```
for(int i = 1; i < array.length; i++) {
        int temp = array[i];
        int j = i - 1;
        while(j >= 0 && array[j] > temp) {
                array[j + 1] = array[j];
                j--;
        }
        array[j + 1] = temp;
}
```

När man når ett element i den sorterade delen som är mindre än det elementet man jämför med kan man stoppa sorterandet då man vet att allt tidigare i arrayen också kommer vara mindre. Även för denna algoritm är tidskomplexiteten $O(n^2)$ men ändå är den snabbare som man ser i 1. Detta beror på att insertion sort inte alltid behöver jämföra med alla element bakom sig. Om arrayen redan är någorlunda sorterad kommer jämförelserna vara mindre än för selection sort. I värsta fall är arrayen sorterad i fel ordning, det vill säga att den exempelvis sorterat, går från stort till litet när man vill att arrayen ska gå från litet till stort. Då behöver man göra lika mycket jämförelser som selection sort och de är då likvärdiga. Eftersom oddsen för det är små, och att båda algoritmerna är likvärdiga vid värsta scenariot är insertion sort den bättre algoritmen. Den är även inte så mycket svårare att implementera.

Merge Sort

Den sista utav de sorteringsalgoritmerna är merge sort och den är väldigt annorlunda från de två tidigare algoritmerna. Tillskillnad från selection- och insertion sort som bara kallas på och sedan utför sitt specifika syfte, grundar sig merge sort i att den är rekursiv. Med rekursivitet innebär det alltså att metoden kommer kalla på sig själv för att fungera. Att detta fungerar är väldigt ologiskt om man inte stött på det tidigare.

```
private static void mergeSort(int[] array) {
    :
    :
    :
    mergeSort(leftArray);
    mergeSort(rightArray);
    merge(leftArray, rightArray, array);
}
```

Som vi ser i denna del utav koden kallar sorteringsdelen på sig själv för att bryta ned arrayen till så små delar som möjligt. Man kan föreställa sig att om ens ursprungs array har 8 element, kommer den delas till 2 stycken med 4 element, sedan till 4 stycken med 2 element och till sist 8 stcken arrays med 1 element. Dessa arrays kommer sedan sorteras och sammanfogas för att till sist bilda en sorterad array med 8 element. Detta är ett väldigt effektivt sätt att sortera på då tidskomplexiteten för merge sort är $O(n * log(n))$.

Som man ser i 1 är denna algoritm mycket snabbare än de andra två för stora n även om det inte är uppenbart om exakt vilken tidsutveckling den har. Utöver att merge sort är snabbare än de andra algoritmerna så är även merge sort en stabil sorteringsalgoritm vilket innebär att den behåller den givna ordningen om det finns flera element som är lika. I denna uppgift är det inte nödvändigt med en stabil sorteringsalgoritm men om man till exempel sorterar en telefonkatalog efter efternamn så vill man att Anders Svensson står före Sven Svensson och det kan inte garanteras i en instabil sorteringsalgoritm.

Det negativa med merge sort är att man behöver mycket minne för att det ska fungera. För att kunna använda merge sort används en extra array som är lika stor som originalet eller flera mindre arrayer som tillsammans blir lika stora som originalet beroende på hur man gör sin kod för sorteringen. Tillskillnad då från selection- och insertion sort som använder en konstant del minne använder merge sort ha en linjär ökning av minnesanvädning. Om exempelvis selection sort ska arbeta på en dubbelt så stor array kommer inte någon extra del i minnet användas utöver arrayen. Om samma sak skulle hända för merge sort kommer man behöva lagra en kopia av arrayen i minnet. Minnesanvändningen för merge sort blir då $O(n)$.

För att försöka förbättra merge sort algoritmen skulle man ändra så att det inte är lika mycket kopiering till och från den temporära arrayen. Man minskar antalet kopieringar genom att ändra så att man i det rekursiva sorterar i den temporära arrayen, på så sätt behöver man inte kopiera in nåt i den temporära i merge-steget då allt redan är sorterat och kan placeras direkt in i original arrayen. Jag försökte implementera detta men jag tror inte att jag lyckades då jag inte direkt fick några bättre tider på benchmarken vilket borde ha skett då man inte behöver spendera lika mycket tid på kopiering.

Benchmark

Antal element	Selection Sort	Insertion Sort	Merge Sort
1000	2000	600	900
2000	7800	2100	1500
3000	17 100	4800	2700
4000	29 700	8600	2900
6000	64 400	22 000	5000
8000	111 200	33 500	6800
16 000	438 500	144 200	23 700

Table 1: Tabell som visar hur snabbt det går att sortera med olika algoritmer (mikrosekunder)

Diskussion

Det svåraste med denna uppgift var att förstå hur rekursion fungerar eftersom detta var första gången jag arbetade med det sättet att programmera på. Att det funkar att använda är väldigt ologiskt till en början men ju mer jag jobbade med merge sorteringen började det att falla på plats och jag skulle vilja påstå att jag nu har en någorlunda bra förståelse för hur rekursion funkar, åtminstone i detta problem.

Något som jag fick bättre förståelse var tidskomplexiteten. Bara för att två algoritmer har samma tidskomplexitet betyder det inte att dom kommer vara lika snabba eller långsamma, bara att de utvecklas på liknande sätt vid större mängd data. Det är fortfarande något som jag inte direkt kopplar till att vara uppenbart trots att ordo inte är en funktion för att beräkna tiden, så det är bra att nöta in det lite extra.

En väldigt bra sak med denna uppgift var att det finns extremt stora mängder resurser om just sorteringsalgoritmer på internet vilket gjorde det betydligt mycket enklare att förstå hur allt fungerar då det finns mycket information om inte bara hur man kodar detta utan även om alla möjliga fördelar och nackdelar till alla algoritmer.