Sorterad Data

Jonathan Nilsson Cullgert

Hösten 2023

Introduktion

I denna uppgift ska man hitta element i en array som är sorterad och samt jämföra några olika metoder av detta och med att hitta element i en osorterad array. Man kommer även leta efter dubbletter i två arrays med olika metoder och se vilken skillnad det är på att jobba med sorterad och osorterad data.

Osorterad Array

I den osorterade arrayen kommer vi finns ifall ett element finns i arrayen genom att söka igenom arrayen från början till slut. Det vill säga vi kollar vårt sökta element mot varje element i arrayen. Om vårt sökta element inte finns i arrayen måste vi fortfarande söka igenom hela. Det betyder att tiden det tar att hitta elementet, som värst, alltid kommer relatera till hur många element, n, som finns i arrayen. $O(n)$ kommer alltså vara tidskomplexiteten. Det vill säga att för 1 miljon element kommer det ta 150 ns och för exempelvis 2 miljoner 300 ns.

Sorterad Array

I den sorterade arrayen kommer samma princip följas som i den osorterade, skillnaden är att vi kan lägga till en förbättring. Den förbättringen är ifall elementet vi letar efter inte finns i arrayen. Det gör vi genom följande del av kod.

```
if(array[i] > key){
    return false;
}
```

Eftersom arrayen nu är sorterad kommer vi kunna sluta leta efter elementet ifall vi vet att nästa element i arrayen är större än det element vi letar efter. Det dåliga är fortfarande att tidskomplexiteten kommer vara

densamma som för den osorterade ifall elementet vi letar efter inte finns med i arrayen, eller ligger i slutet. Fördelen för denna metod är ifall vi vårt element skulle ligga någonstans i mitten. Exempelvis om vi letar efter 50 i en array med talen 0-99 men just 50 finns inte med. Då skulle det gå mycket snabbare i den sorterade än i den osorterade att ta reda på om elementet finns där eller inte. För 1 miljon element skulle det ta ungefär 70 ns medan i den osorterade tar det 150 ns. I 1 visas hur snabb den sökningen i den sorterade arrayen är och det visar verkligen att det är ett linjärt samband mellan tiden och antal element.

Det kan vara så att ibland är den osorterade snabbare än den sorterade då elementet man letar efter kan ha hamnat långt fram i arrayen men det är det genomsnittliga fallet man bryr sig om. Något som jag märkte är att den osorterade är dubbelt så snabb som den sorterade ifall man letar efter ett element som ligger utanför arrayen och är större än det största elementet. Detta låter rimligt då den sorterade algoritmen har en extra if-sats att köra varje gång man kollar element. I 2 ser man att genom fler tester med flera olika nycklar som vi söker efter blir den sorterade generellt bättre.

Binärsökning

För att hitta element ännu snabbare kan vi implementera en algoritm som heter binärsökning. Denna algoritm går ut på att flytta den högsta och lägsta punkten på arrayen och på sådant sätt eliminera halva arrayen varje gång man kör ett steg i algoritmen(delar av arrayen blir inte borttagen den bortses bara). Det viktiga i denna algoritm är att ha en indexpekare på mitten elementet och två pekare på slutet och början av stacken. Det är även viktigt att poängtera att arrayen måste vara sorterad för att denna algoritm ska fungera.

```
int index = first + ((last - first) / 2);
```

Denna kod gör beräkning på vart mittenindexet befinner sig. Man kollar sedan om mittenindexet är lika med elementet man söker, då är det hittat. Om detta inte är fallet, vilket är troligt, kollar vi om det sökta elementet är större eller mindre än mitten av arrayen. Om mitten är större än det sökta elementet flyttar vi pekaren på sista elementet till mitten - 1. Om mitten är mindre flyttar vi pekaren på sista elementet till mitten + 1. Sedan återupprepas denna process tills elementet hittas eller arrayen är helt genomsökt.

```
if (array[index] < key && index < last) {
    first = index + 1;
    continue;
}
```

```
if (array[index] > key && index > first) {
    last = index - 1 ;
    continue;
    }
```

Eftersom hälften av arrayen elimineras vid varje steg kan man ana att denna algoritm inte har linjär tidskomplexitet. Det visar sig att tidskomplexiteten är $O(log(n))$. Detta kan man se i 2 då tiden för när den osorterade och sorterade sökningen blir väldigt stor håller sig den binära sökningen snabb.

Dubbletter

I nästa del ska man finna dubbletter i två stycken arrays och med hjälp av linjärsökning, binärsökning och dubbla pekare hitta vilket sätt som är effektivast. I den linjära sökningen kommer vi gå igenom alla element i den andra arrayen för varje element i den första arrayen i värsta fall, alltså kommer vi få en tidskomplexitet på $O(n^2)$. Med den binära sökningen kommer vi för varje element i den första arrayen göra en binärsökning i den andra, i värsta fall kommer vi då få en tidskomplexitet på $O(n * log(n))$.

Med dubbla pekare kommer vi hålla koll på elementen i varje array och sedan röra oss fram i dem. Vi kommer alltid jämföra dessa element och röra oss framåt ett steg i den arrayen som innehåller det mindre elementet utav de två. Om elementen skulle vara samma så har vi hittat vår dubblett. Ifall man skulle gå utanför någon av arrayerna kommer programmet brytas då det inte finns någon dubblett, eftersom det betyder att alla element som är kvar i den man inte gått utanför kommer vara större än det sista i den man gått utanför.

```
if (array[i] < key[j]){
        i++;
        }
        else{
        j++;
        }
```

Det visar sig att denna metod att leta efter dubbletter är ännu mer effektiv än binärsökning. I 3 ser man dom olika tiderna som det tar för att hitta dubbletter med de olika metoderna. Att det linjära ökar kvadratiskt är ganska uppenbart med skillnaden mellan binär och dubbelpekare är inte lika uppenbar. Dubbelpekarna ser ju ut att öka någorlunda linjärt medan den binära verkar vara effektivare. Eftersom den binära sökningen har tidskomplexiteten $O(n*log(n))$ kommer den vara snabbare än den dubbelpekaren på små n medan på stora n kommer dubbelpekaren vara mycket mer effektiv.

Benchmarks

Antal element	Tid att leta efter element (ns)
100 000	30
250 000	70
500 000	150
750 000	220
1 000 000	290

Table 1: Tabell som visar hur snabbt det tar att leta efter ett element i en sorterad array av en viss storlek

Antal element	Osorterad (ns)	Sorterad (ns)	Binärsökning (ns)
100	80	20	30
200	150	40	40
400	300	70	50
800	560	120	50
1600	860	240	60
3200	1130	460	70
6400	1300	910	80

Table 2: Tabell som visar hur snabbt linjär i osorterad/sorterad array och binärsökning är.

Antal element	Osorterad (ns)	Binärsökning (ns)	Dubbelpekare (ns)
100	60	30	40
200	250	40	90
400	1120	50	50
800	4230	80	120
1600	17 520	110	220

Table 3: Tabell som visar olika hastigheter att hitta dubbletter

Slutsats

Det jag främst lärt mig med denna uppgift är tidskomplexitet och hur saker kan verka vara effektiva vid mindre tester men som egentligen är väldigt oanvändbara vid verkliga appliceringar då man använder data som är stor. När man skriver ett program så är det verkligen viktigt att optimera dessa tider om man jobbar på något som kan vara skillnaden på liv- och död. Det kan även vara bra att göra sig bekant med detta när man gör sina egna sidoprojekt så att man får in vanan.

Det är även tydligt att vissa algoritmer verkligen är helt oanvändbara när man använder dessa med mycket data. Bara att göra tester själv på min dator tog väldigt lång tid med inte så extremt mycket element i arrayerna. Om man då skulle använda detta när man analyserar data från serverhallar eller liknande förstår jag verkligen att man antingen låter det stå och köra i några år eller så optimiserar man koden sådan att det inte tar så lång tid.

Det har också visat sig hur användbara sorterad data är, sökningsalgoritmerna blir så extremt mycket mer effektiva om datan man jobbar med är sorterad. Förmodligen finns det en viss kostnad att sortera data som man måste balansera upp med vinsten man får av att den är sorterad. Det lär var något som man ständigt kämpar med, vad tjänar man mest på.

Viktigt att vara noggrann när man kodar, det var många delar av denna uppgift där jag fastnade för att man gjorde simpla fel men som inte blir ”fel” när man exekverar, såsom att man använder exempelvis

```
break;
```

fel eller att man råkar ändra på fel variabel. När jag skulle göra dubbelpekar-algoritmen fick jag det inte att funka på väldigt länge. Anledningen var att jag råkade röra mig framåt i bara en av arrayerna. Ibland är det bra att ta en paus och komma tillbaka till problemet vid ett senare tillfälle med nya fräscha ögon.