Fördelen med sorterad data

viktor Björkén

September 2022

1 Introduktion

I denna rapporten ska vi behandla hur sortering samt sökning av arrayer ser ut. Vi kommer att benchmarka olika tider för sökning av sorterade samt osorterade arrays. Vi kommer inse att det är mer krävande att söka igenom en osorterad array gemtemot en sorterad sådan.

2 Osorterad

I denna första del ska vi börja med att lära oss hur vi söker efter olika element i en array, vi gör detta med den kodsnippen vi får i uppgiften. Vi använder denna kod och sedan benchmarkar den för att att hur tiden växer men storleken på arrayen.

I mitt fall anänvder jag mig av kunskaperna från tidigare rapporter för att skapa en slumpmässig array med egenvald längd som ökar hela tiden.

Det vi gör för att få så säkra värden vi kan är att vi loopar varje iteration med en satt array size ex antal gånger för att få ut ett snitt.

Array(n)	key	runtime
100	2	400 ns
1000	4	1500 ns
10000	8	0,2 ms
100000	16	2,7 ms
1000000	32	12 ms

Det vi kan se är hur tiden ökar kontunerligt med antalet n som ökar. Vi kan därmed se att tidskomplexieten för denna linjära sökning av en osorterad array är O(n).

Våran tabell må ej vara 100 procent korret jämtemot våran slutsats men detta kan bero på utomstådende faktor som vad som körd i bakgrunden på datorn.

3 Sorterad

I denna del ska vi istället titta på hur tiden förändras och jämförs när vi istället söker genom en redan sorterad array.

Detta kan vara till våran fördel då det är mycket mer tidskrävande att söka igenom någon som ej ligger i ordning.

Vi gör detta på liknande sätt som den förra uppgiften, men eftersom arrayen är sorterad kan vi stoppa sökandet när vi når upp till ett värde som är större än det vi söker. Detta gör att vi kan dra ner tiden med en hel del

Array(n)	runtime
20	7 ns
40	16 ns
80	35 ns
160	75 ns
320	150 ns
640	310 ns
1280	630 ns
2560	1400 ns

Med dessa resultat kan vi dra slutsatsen att tiden dubleras när antalet element i arrayen dubbleras. Med denna information kan vi konstantera att tids komplexiteten är O(n).

Om vi skulle uppskata utan att med hjälp av att testköra tiden det tar att gå egenom en array med 1 000 000 element kan vi med hjälp av våra tidigare värden anta att trenden med ett tidsdubblering kommer fortsätta. Därför skulle vi få att det skulle ca 50000 ns

4 Binärsökning

I en binärsökning arbetar vi lite annorlunda. Denna metod går istället ut på att först hoppa till mitten av en array och sedan jämföra hurvida talet vi söker är större eller mindre. Om något av detta stämmer hoppar vi antigen till mitten av de lägre talen eller till mitten av de högre talen. Sedan fortsätter vi så tills vi hittar att talet som vi söker.

Vi börjar med att hitta mitten på arrayn, vi gör detta genom att ta vårat minsta och största värde genom 2. Vi fortsätter sedan att fylla i det som saknas av den kod vi har fått i uppgiften och får följande funktion.

```
int first = 0;
int last = array.length-1;
    while (first <= last) {
    // jump to the middle
    int index = (first + last)/2;
    if (array[index] == key) {
```

```
            // hmm what now?
            return true;}
        else if(array[index] < key)
            first = index + 1;
        else
            last = index - 1;
        }

    return false;
```

Det vi då har fyllt i gär så att koden fungerar som avsett, där vi adderar eller subtraherear från index för att få nya minsta och största element i arrayen. När vi väl uppfyller kravet att arrayen nuvarande index är lika med det vi söker retunerar vi true.

Med denna funktion får vi ut följande värden som kan visualliseras med en graf.

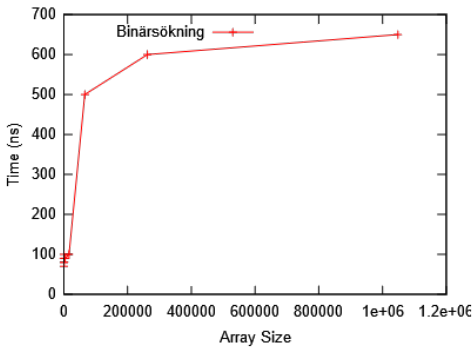

Figure 1: Binärsökning graf

Denna graf visar klart och tydligt att det är en tidskomplexitet på O(log(n)) vi har att göra med. Med denna graf kan vi se att för en array med 1 000 000 element kommer det att ta ca 600 ns. Vilket är en markant förbättring tillskilland från den tidigare algoritmen.

4.1 Dupletter

I denna deluppgift ska vi använda oss av samma algoritm som tidigare uppgift men istället inplementera en funktion som söker igenom 2 olika arrayer för att hitta dupletter inom dessa. För att detta ska funka är det viktigt att alla element är unika. Det vi gör för att inplementera att söka genom 2 arrays är följande.

```
int index2 = 0, index1 = 0;

while (index2 < b.length && index1 < a.length) {
    int first = 0;
    int last = a.length-1;

    while (first <= last) {
        int index = (first + last)/2;
        if (b[index2] == a[index]) {
            System.out.print(a[index] + " ");
            break;
        } else if (b[index2] > a[index])
            first = index + 1;
        else last = index - 1;
    }

    index2++;
}
```

Om vi nu ska jämföra våra denna nya algorim med den från en tidigare raport kan vi anta att denna kommer vara snabbre. Detta då den tidigare linjära algorimen sekvensiellt går igenom båda arrayerna till den hitta 2 element som är samma. Medans den binära använder samma metod som vi gick genom när vi sökte efter ett speciellt värde i bara 1 array. Problemet och en annan skillnad mellan de 2 är att den binära endast kan användas på sorterade arrayer.

Då är en annan punkt som kan vara värd att diskutera, hur vidare det går snabbare att först sortera våra arrayer för att sedan jämföra de med den binära metoden eller söka de osorterade med hjälp av den linjära.

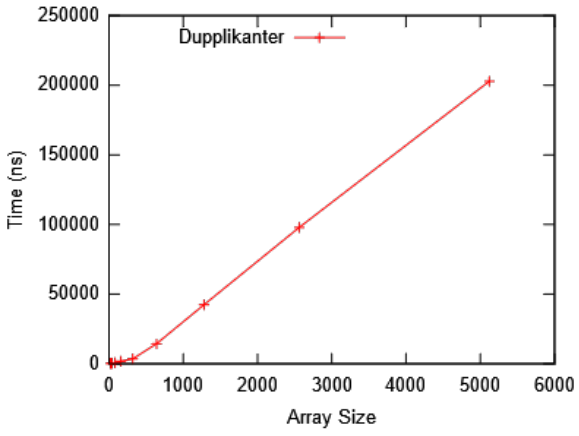

Figure 2: Hitta dupletter med binärsökning

Med denna graf kan vi anta att det är en O(n log(n)) som vi har att göra med.

Om vi återgår till punten angående hurvida det är snabbast att sortera först eller ej, vi kan då jämföra de olika tidskomplexiteterna vi har fått för att få fram en hypotes om vilken vi anser vara snabbast.

Eftersom vi inte gått igenom hur tidskomplexiteten ser ut på olika sorteringsalgoritmer är detta svårt att svara på detta i denna stund och är därför något vi får ta upp i en kommande uppgfit. Vi kan dock anta att sortera först kommer att vara det bättre allternativet då vi har flera verktyg i form av snabba och pålitliga sortingsalgoritmer i våran verktygslåda.

Det finns även flera sätt att ytterligare optimera denna denna hitta dupletter funktion. Ett annat sätt är att skriva om algoritmen så att vi håller koll på nästkommande elementet i den första listan och jämför huvida elementet i nästa element i den andra arryen är mindre. Om de skulle vara samma har vi hittat ett par dubletter. Detta är en mycket mer tids effektiv då den har koll på nästa element redan innan.