Uppgift 3

Simon Moradbakti

HT 2024

1 Introduktion

I den här uppgiften var huvudsyftet att ta reda på hur snabbt man kan söka igenom en sorterad array kontra en osorterad array. Rapporten kommer också diskutera hur och varför den ena kan vara bättre än den andra. Det kommer också undersökas hur sambanden ser ut till exempel med ordo och andra mätningar.

2 A first try

I denna första deluppgift fick skribenten en del kod att arbeta från:

```
public static boolean unsorted_search(int[] array, int
    key) {
for (int index = 0; index < array.length ; index++) {
if (array[index] == key) {
return true;
}
}
return false;
}
```

Denna kod används för att söka genom en osorterad array. Den fungerar genom på det sättet att den letar efter en ”key” eller värde i den osorterade arrayen. Syftet med denna uppgift var att undersöka hur lång tid det tar att söka igenom denna array. Det kan enklast göras genom att göra ett prestanda test, där olika arrayer med olika många element hanteras av denna kod. Man kan öka antalet element i arrayen för att se vad som påverkar denna algoritm.

2.1 prestanda test - osorterad array

tid (ns)	n-värde
5000	100
6000	200
13000	400
32000	800
59000	1600

Table 1: resultat av exekvering

Denna information överfördes till Exel för att enkelt kunna ritas upp för att undersöka ett samband. Med verktygen i Excel kom denna graf fram:

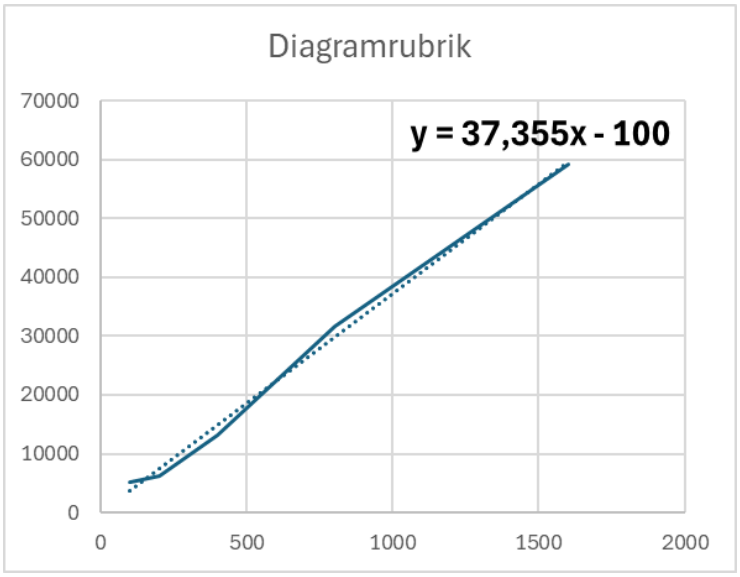

Figure 1: Data infogad i Excel

punktlinjen beskriver trend linjen och har ett linjärt format. Man ser att den heldragna linjen, som följer punktlinjen noga också är linjär. Detta med hjälp av verktygen i Excel ger oss en ekvation, som lyder: y = 37,355x - 100. Ett konstaterande kan alltså göras att vi har en algoritm med tidskomplexiteten *O(n)*. Denna algoritm kan därför visa med säkerhet hur lång tid det skulle ta att sortera en array med en miljon element. Den siffran blev ca 37 miljoner ns. Det som tar tid är det faktum att algoritmen måste kontrollera varenda element i arrayen, vilket inte är speciellet tidseffektivt.

2.2 A sorted array

Ett liknande experiment utfördes, fast med denna gång med en sorterad array. Resultatet blev liknande, fast med förbättrade tider:

tid (ns)	n-värde
4400	100
1600	200
18300	400
12800	800
40000	1600
710000	1000000

Table 2: resultat av exekvering

En slutsats som kunde dras efter detta experiment var att den tiden som man kan tjäna är större vid relativt små arrayer, medans lite mindre eller ibland större vid större arrayer. Dock blir tiden i absoluta tal oftast kortare med en sorterad array. Då ett samband inte efterfråkades skedde det ingen större utredning om det.

3 Binary search

Binär sökning, till skillnad från det vi gjorde innan, kontrollerar inte alla element i en array. Det är en mycket smartare algrotim lösning den delar upp en sorterad lista, till exempel telefonbok och fokuserar på den delen som sannorlikt innehåller det sökta objektet. Baserat på objektet letar den efter objektet i någon av dessa delar. Detta sparar tid. En del kod fick man som student i uppdrag att fylla. Skribenten har använt sig av denna lösning:

```
while (first <= last) {
    // jump to the middle
                int index = (first + last) / 2;
    if (array[index] == key) {
    return true;
    }
    if (array[index] < key) {
    first = index + 1;
    } else {
    last = index - 1;
```

Denna kod exekverandes och resultaten sattes upp i en tabell:

tid (ns)	n-värde
5600	100
6400	200
6000	400
8000	800
8900	1600
40000	1000000

Table 3: resultat av exekvering

Här går det att analysera dessa värden och se att det ökar väldigt lite i början för att sedan gå snabbare och snabbare för att sedan sakta ner. Detta är precis hur *O(log(n))* ser ut. För att avgöra hur 64 miljoner skulle se ut, utan att köra koden kan vi göra en ekvation likt tidigare. Detta gav ekv systemet: A=800k+285 där A är vårt N värde, k är log värdet. Detta ger: T(64)=800*26+285 = 21000. Medans det exekverade resultatet var 12000, alltså snabbre än man trodde.

4 Recursive programming

Antalet rekursiva anrop beror främst på hur många gånger en array kan halveras så att sökområdet endast kan innehålla ett element. Det kan beskrivas som: antalet rekursiva anrop = log2(n). Detta samband kan hjälpa en att ta reda på vad 1000,2000, 4000 och 1000000 har för antal rekursiva anrop. In stoppat i formeln fås:

anrop	element
10	1000
11	2000
12	4000
20	1000000

Table 4: Resultat instoppat i formeln

För att besvara frågan om rekursiva anrop kan vara ett problem för stacken, så är svaret nej. Rekursiva anrop växer logaritmiskt och därför långsamt. För stora element som en miljon tar det endast 20 anrop vilket de alldra flesta moderna system klarar av. Rekursiva anrop tar dock upp mer minne, men i binära sökningar är detta inte ett problem.