Arrays och prestanda

Viktor Björkén

Spring Fall 2022

Introduktion

I denna rapport kommer vi att behandla olika tidsmätningar för olika sorters funktioner som behandlar arrayer på olika sätt. Koden kommer att skrivas i java och vi kommer använda funktionen nanoTime() för att beräkna detta. Vi kommer inse att något så enkelt som att räkna tiden mellan olika operationer inte är så enkelt och förutsebart som man kan tro.

nanoTime

0.1 Uppgift 1

```
resolution 100nanoseconds
resolution 200nanoseconds
resolution 200nanoseconds
resolution 200nanoseconds
```

Våran noggranhet är måttlikt tillfredsställande då det nästan bara skiljer 100ns mellan våra mätningar. En faktor som kan göra att denna nogrannhet inte stämmer är faktumet att vårat värde är så pass litet.

Javas egna funktion nanoTime är inte fullt nogrann, nogrannheten kan variera stort mellan olika operativsystem samt hårdvara.

0.2 Uppgift 2

Det denna kodsnippet gör är den tar en redan färdig array och sedan summerar ihop de värden som denna array innehåller. nanoTime() körs där dessa värden summeras.

Problemet som uppstår när man ska göra en mätning för en singel instruktion är att själva nanoTime() funktionen behöver tid att exikveras. Detta gör att mätningen med endast 1 instruktion kan bli bristfällig

Denna kod är dessutom sekvensiell och inte slumpmässig vilket är det vi vill testa i denna rappot.

Task 1

För att komma igång med första deluppgiften använder vi koden som vi fått i uppgiften och sedan adderar de rader kod som saknas.

```
for(int i = 0; i < l; i++){
    indx[i] = rnd.nextInt(n);
}

        for(int i = 0; i < l; i++){
    array[i] = 1;
}

sum += array[rnd.nextInt(n)];

        for(int j = 0; j < k; j++){
    for(int i = 0; i < l; i++){
        sum += array[i];
    }
}
```

Våran första benchmark vi kommer att testa är en så kallad, random access funktion. Det denna funktion ör är att den skriver och läser från en slumpmässig plast i en array. När vi ökar vårat n värde, där n är antalet platser i våran array som vi ska gå igenom. När vi ökar n får vi dessa mätvärden för tiden.

vi skulle gärna velat ha mer likvärdiga resultat men pga olika andledningar får vi ej det. när det gäller n = 10 är värdet för litet för att få ett trovärdigt resultat.

n	**runtime**	
10	16ns	
100	15ns	
1000	15ns	
10000	13ns	

Table 1: random access körtid för olika n värden i nanosekunder

Det vi kan se i våran tabell är hur tiden minskar ju högre värde på n vi har. Detta beror på att vi får ett fler antal n som vi kollar vilket gör att det tiden blir mer nogrann. Med denna information kan vi antyda att nogrannheten ökar ju högre värde på n vi har, detta leder till att vi inte kan

anse att tiden är tillräckligt nogrann för de lägre värdena på n medans de högre blir mer och mer nogranna.

Task 2

Vi börjar med att använda koden vi har fått i uppgiften och sedan fyller de rader kod som saknas för att få en funkerade funktion,

```
for(int i = 0; i < m; i++){
    keys[i] = rnd.nextInt(10*n);
}

for (int i = 0; i < n; i++){
    array[i] = rnd.nextInt(10*n);
}
```

Därefter ändrar vi returnvärdet för att dela på antal n istället för k*l som i task 1

```
return ((double)(t_total ))/((double)n);
```

I deluppgift 2 ska vi istället använda oss utav en ”search” funktion och sedan mäta tiden.

Det vi gör först är att vi försöker hitta värden på k och m som visar samma värde med samma n värde. Vi får då att `k=1000 m=10000` Därefter ökar vi värdet på n för att se vad vi får för körtid. När vi ökar vårat n värde ser vi hur körtiden inte ändras.

n	**runtime**	
100	6.6ms	
1000	5.2ms	
10000	5.1ms	
100000	5.2ms	

Table 2: search körtid för olika n värden i millisekunder

Detta betyder att vi har en tidskomplexitet på O(1).

O(1) är inte en tidskomplexitet som har det vanliga utan sker oftast vid enkla operationer som att titta på ett stationert värde i en array. Det som gör att vi har denna typ av tidskomplexitet i denna uppgifter är då vi söker på varje plats i en array var för sig oberoende av hur stor den är. Tiden blir ju såklart längre dessto större array men tiden per varje index hålls dessamma.

Task 3

I den sista deluppgiften handlar det om att jämföra 2 arrays för för att se om det finns några duplikanter. Detta gär vi med hjälp av denna kodrad som vi ändrar ifrån föregående uppgift.

```
for (int ki = 0; ki < n; ki++) {
                for (int i = 0; i < m ; i++) {
                    if (array[ki] == keys[i]) {
                        sum++;
                        break;
                    }
                }
            }
```

Vi ändrar dessutom om så att vi har en given array med givna värden (i detta fall 1).

```
for(int i = 0; i < m; i++){
    keys[i] = 1;
}
```

n	runtime
100	1ms
1000	0.3ms
10000	0.2ms
100000	0.2ms

Table 3: search körtid för olika n värden i millisekunder

Denna uppgift ger samma resultat som den första deluppgiften gjorde, med denna information kan vi dra slutsatsen att vi ser en liknande mönster i denna funktion. Desto högre värde vi har på n desto mer nogrann blir tiden som vi mäter.

om man ska estimerar hur stort n skulle kunna vara för om datorn fick jobba i en timme hade tagit tiden det tog för min dator att bearbeta 10000 och sedan tagit detta värde gånger 120 då det tog mig 30 sekunder att få ut ett värde från n = 100000.