Arrayer och prestanda

Kenan Dizdarevic

30 augusti 2022

Inledning

Syftet med denna rapport är undersöka prestandan av olika operationer utförda på arrayer. Det vi tar hänsyn till är storleken på arrayen när vi mäter tiden för processen. Vi kommer att utföra prestandatester på tre olika fall. Det första fallet som vi undersöker är hur lång tid det tar att skriva ett värde till ett slumpmässigt index i en array. I fall två undersöker vi en array och hitta ett element. I det tredje fallet hittar vi par av tal i två olika arrayer och avgör hur lång tid operationen tar att genomföra.

Undersökning

Slumpmässig tillgång

I detta fall undersöker vi hur lång tid en operation tar att genomföra när den är begränsad av klockan. Vi kommer att använda oss av följande kod för att mäta tiden som det tar för att genomföra en operation i Java.

```
for(int i = 0; i < 10; i++){
          long n0 = System.nanoTime();
          long n1 = System.nanoTime();
          System.out.println("Resolution " + (n1 - n0) + " ns");
     }
```

Efter att vi körde koden första gången fick vi ett intervall mellan 100 till 400 nanosekunder för klockan. När vi senare körde koden flera gånger fick vi varierande resultat. Tiden varierade från 300 till 2500 nanosekunder. Vi ser att värdet har stigit markant, detta kan bero på bakrundsprocesser som stör. Även metoden som finns i javas bibliotek är precis men nödvändigtvis inte exakt. Problemet är att vi har för få operationer som körs för att kunna avgöra om klockan är exakt eller inte

Vi kommer nu att undersöka om klockan är precis när vi försöker komma åt ett index i en array. Vi får värden mellan 500 till 900 nanosekunder. Men som vi nämnde tidigare kör vi endast 10 operationer vilket inte är tillräckligt

för att kunna avgöra om klockan är exakt. För att göra denna undersökning verklighetstrogen bör vi itererar våra loopar fler gånger, upp emot 100000. Detta leder till att fler operationer körs samt att vi kommer åt element i arrayen flera gånger. Detta leder till att vi kan studera klockan under längre tid vilket i sin tur leder till exakta mätresultat och en tydlig slutsats.

I nästa steg ska vi försöka minimera omständigheterna som kan påverka vårt mätresultat. I detta fall är det att vi slumpmässigt ska välja ett specifikt index som vi vill komma åt. Vi har fått en färdigställd kod, men den har ett problem som påverkar vår mätning.

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

Problemet ligger i denna rad av kod. Det som händer när denna rad körs är att vi adderar värdet av ett slumpmässigt index i arrayen till vår variabel sum. Men ytterligare en sak sker som påverkar tiden det tar för detta att ske, detta är att ett tal tas slumpmässigt fram. Detta är inte gratis och tar tid. I vårt fall skulle vi mäta tiden det tar att komma åt ett slumpmässigt index i en array och inte hur lång tid det tar att få ett slumpmässigt tal. För att åtgärda detta kan vi tidigare i koden skapa en ny array och slumpmässigt sätta värden på alla index med samma metod som ovan. Detta innebär att vi inte kommer behöva mäta tiden det tar att slumpmässigt få ett tal när vi vill komma åt ett slumpmässigt index. Huvuddelen där själva testet kommer att ske ser ut som följande:

```
long t0 = System.nanoTime();
        for(int j = 0; j < k; j++){
            for(int i = 0; i < l; i++){
                sum += array[indx[i]];
            }
        }
        long t_access = (System.nanoTime() - t0);
```

Vi har tidigare i koden satt slumpmässiga värden på arrayen indx vilket leder till att vi endast studerar hur lång tid det tar att komma åt ett slumpmässigt index i arrayen. Resultaten visas i tabellen nedan.

Iterationer	Tid
10	0.39ns
1000	0,63ns
10000	0.42ns
100000	1.1ns

Tabell 1: Exekveringstid att komma åt slumpmässigt element i en array med avseende på antal iterationer.

Vi ser att tiden stiger beroende på antal iterationer. Slutsatsen som vi kan dra utifrån detta är att denna algoritm tillhör $\mathcal{O}(1)$, då det tar lika lång tid att komma åt alla element i arrayen.

Sökning

Nu ska vi studera en enkel sökningsalgoritm. Vi skall undersöka hur exekveringstiden påverkas vid olika storlekar på arrayerna. Undersökningen kommer att bestå av två olika arrayer där den ena är den vi ska söka i och den andra som innehåller våra nycklar. Det viktiga är att vi har fler nycklar än vad längden på arrayen som vi ska söka i är. Vi fyller båda arrayerna med slumpmässiga tal från 0 till 10n, där n är storleken på arrayen som vi söker i. Själva huvuddelen där sökningen kommer ske ser ut på följande vis:

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

Tiden vi mäter är från när for-loopen startar tills vi har kört upp till m och som representerar antalet nycklar. Vi kommer att utföra våra mätningar med m=10 000 och k=1000.

Iterationer [n]	Tid
100	12ns
1000	903ns
2000	4300ns
10000	26000ns

Tabell 2: Exekveringstid för att söka efter slumpmässiga tal i en array.

Med våra mätresultat kan vi göra en simpel linjär regression, vi får då en rät linje. Slutsatsen som vi kan dra utifrån detta är att grafen tillhör $\mathcal{O}(n)$.

Dubbletter

Vi ska nu avgöra hur lång tid det tar att hitta dubbletter i två lika stora arrayer.

Storlek	Tid
10	0,0000022s
1000	0,006s
10000	0,05s
50000	0,6
100000	5s

Tabell 3: Exekveringstid för att söka efter dubbletter i en array.

Med hjälp av våra mätresultat kan vi framställa en formel som beskriver grafen samt beräkna hur stora arrayer vi kan söka igenom på en timme. Denna algoritm tillhör $\mathcal{O}(n^2)$. Formeln vi får är:
$f(n) = 0.0000000007n^2 + 0.00003n + 0,07$.
1 timme innehåller 3600, vi löser ekvationen för $f(n) = 3600$.

$$3600 = 0.0000000007n^2 + 0.00003n + 0,07n \tag{1}$$

.

$$n \approx \frac{-0.00003 + \sqrt{0.0000100809}}{0.0000000014} \tag{2}$$

.

$$n \approx 2.3 * 10^6 \tag{3}$$

.
Storleken på vår array kan alltså vara ungefär 2.3 miljoner om vi ska hinna söka efter alla dubbletter på en timme.

Slutsats

Vi ser att slumpmässig sökning tillhör $\mathcal{O}(1)$ då det tar lika lång tid att komma åt varje element. Sökningsalgoritmen tillhör $\mathcal{O}(n)$. Att leta efter dubbletter i två arrayer tillhör $\mathcal{O}(n^2)$.