Assignment 1 - Algoritmer & datastrukturer

Jesper Hesselgren

August 21, 2024

Introduktion

Målet med denna uppgift är att utforska och analysera effektiviteten av tre olika operationer på arrayer. De tre operationerna som ska undersökas är **Random access**, som innebär att skriva eller läsa ett värde på en slumpmässig plats i en array. **Search**, som innebär att söka igenom en array för att hitta ett specifikt värde. **Duplicates**, som innebär att hitta alla gemensamma värden i två arrayer. Programmeringsspråket som används för att genomföra dessa analyser är Java.

Systemklockans upplösning

För att mäta effektiviteten hos de olika operationerna används metoden `System.nanoTime()`, metoden mäter med hög precision ett tidsintervall i nanosekunder. För att kunna lita på mätningar är det viktigt att undersöka klockans upplösning. Genom att köra följande kodblock, där två tidpunkter mäts direkt efter varandra, går det att utvärdera klockans upplösning:

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

Syftet med mätningen är att undersöka hur små tidsintervall klockan kan skilja på. Eftersom de två tidpunkterna mäts direkt efter varandra bör den uppmätta tiden representera klockans upplösning. Utifrån resultatet i tabell 1 går det att konstatera att klockans upplösning varierar, men de vanligaste värdena är runt 41-42 nanosekunder. Detta indikerar att klockan kan skilja på tidsintervall omkring denna storleksordning, men det finns också mätningar som visar 0 nanosekunder, vilket antyder att det finns begränsningar i klockans noggrannhet för mycket små tidsintervall.

Mätning	Upplösning (ns)
1	41
2	42
3	41
4	41
5	83
6	0
7	0
8	41
9	42
10	42

Tabell 1: Resultat av mätningarna för systemklockans upplösning

Random access

Kodblocket nedan mäter tiden det tar att utföra åtkomstoperationer i en array. Programmet genererar en array av storlekarna i arrayen sizes och anropar funktionen bench, som ansvarar för att utföra själva testet genom att utföra ett antal åtkomstoperationer och mäta den totala tiden. För att säkerställa att mätningarna är mer pålitliga, anropas bench först en gång för att värma upp JIT-kompilatorn. Därefter anropar programmet bench igen för varje arraystorlek, där varje storlek testas tio gånger, och den snabbaste tiden för varje storlek sparas i variabeln min.

```
public static void main(String[] arg) {
    int[] sizes = {100, 200, 400, 800, 1600, 3200};

    // JIT warmup
    bench(1000,10000000);

    int loop = 1000;
    int k = 10;
    for(int n : sizes) {
        long min = Long.MAX_VALUE;
        for (int i = 0; i < k; i++) {
            long t = bench(n, loop);
            if (t < min) min = t;
        }
        System.out.println(n + " " + min/1000 + " ns");
    }
}
```

Resultaten från mätningarna i tabell 2 visar att åtkomsttiden per operation i arrayerna är cirka 1,92 ns, oavsett arraystorlek. Detta indikerar att åtkomstoperationen för en array har tidskomplexiteten O(1). Den högre åtkomsttiden för den minsta arraystorleken (100 element) kan sannolikt förklaras av att JIT-kompilatorn ännu inte har optimerat koden fullt ut under kompileringsfasen. När arraystorleken ökar och koden har körts fler gånger, har JIT-kompilatorn kunnat optimera koden helt, vilket leder till mer konsekventa och lägre åtkomsttider. Ratio-kolumnen i tabellen visar förhållandet mellan åtkomsttiderna för olika storlekar, med arraystorlek 400 som referens.

Arraystorlek	**Tid/åtkomst (ns)**	**Ratio**
100	6.33	3.30
200	1.96	1.02
400	1.92	1.00
800	1.92	1.00
1600	1.92	1.00
3200	1.92	1.00

Tabell 2: Mätningar av åtkomsttid per enskild åtkomst i nanosekunder för olika arraystorlekar

Search Algoritm

Kodblocket nedan mäter den genomsnittliga tiden för att söka efter ett specifikt element i en array. Programmet genererar en array av storleken n, fylld med slumpmässiga heltal mellan 0 och n * 2, samt en uppsättning nycklar (keys) av storleken loop, som också består av slumpmässiga värden. För att få ett mer pålitligt resultat är loop satt till 1000, vilket innebär att sökningen upprepas 1000 gånger. Funktionen search itererar över varje nyckel i keys och söker efter den i array. Om en nyckel hittas, avbryts sökningen för den nyckeln. Sökningen upprepas loop antal gånger, och den totala tiden divideras med loop för att få den genomsnittliga tiden per sökning, som returneras i nanosekunder.

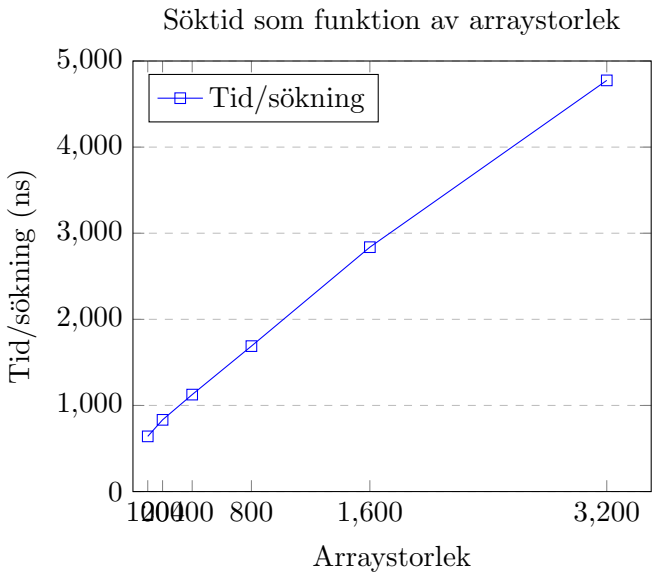
```
private static long search(int n, int loop) {

    Random rnd = new Random();

    int[] array = new int[n];
    for (int i = 0; i < n; i++) {
        array[i] = rnd.nextInt(n * 2);
```

```
        }

        int[] keys = new int[loop];
        for (int k = 0; k < loop; k++) {
            keys[k] = rnd.nextInt(n * 2);
        }

        int sum = 0;

        long t0 = System.nanoTime();
        for (int i = 0; i < loop; i++) {
            int key = keys[i];
            for (int j = 0; j < n; j++) {
                if (key == array[j]) {
                    sum++;
                    break;
                }
            }
        }
        long t1 = System.nanoTime();
        return (t1 - t0) / loop;
    }
}
```

Resultaten i tabell 3 och figur 1 nedan visar tydligt att söktiden ökar linjärt med arraystorleken och att sök algoritmen har tidskomplexiteten O(n). En enkel polynomfunktion som skulle kunna beskriva sökalgoritmen är $T(n) = k \times n$.

Arraystorlek	**Tid/sökning (ns)**	**Ratio**
100	641	1.00
200	833	1.30
400	1126	1.76
800	1689	2.64
1600	2838	4.43
3200	4775	7.45

Tabell 3: Mätningar av söktid per enskild sökning i nanosekunder för olika arraystorlekar



Figur 1: Söktid per enskild sökning i nanosekunder för olika arraystorlekar

Duplicates

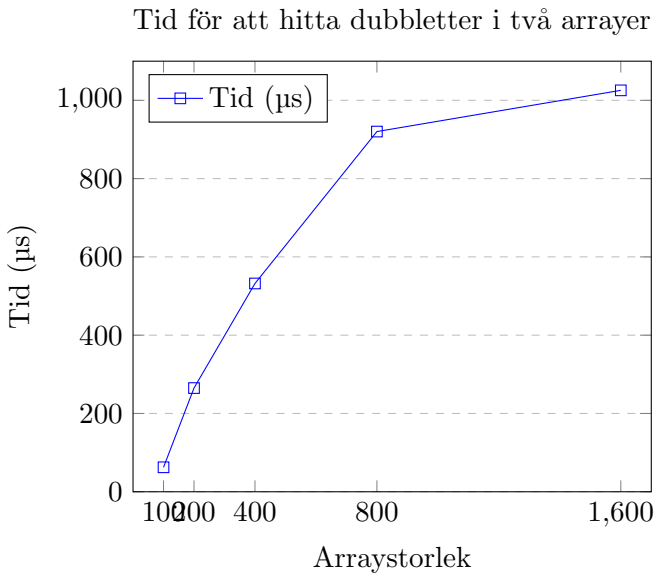
Kodblocket nedan mäter tiden det tar att hitta dubbletter mellan två arrayer. Programmet genererar två arrayer av samma storlek n, `array_a` och `array_b`, som fylls med slumpmässiga tal mellan 0 och $2 \times n$. Därefter jämförs varje element i `array_a` med alla element i `array_b` för att se om en dubblett finns. Om en matchning hittas, avbryts sökningen för det aktuella elementet och funktionen går vidare till nästa. För att undersöka tidkomplexiteten för operationen, ökas storleken på båda arrayerna samtidigt, och mätningar görs för att bestämma den totala tiden det tar att söka igenom alla element och identifiera eventuella dubbletter.

```
private static long duplicates(int n) {

    Random rnd = new Random();
    Random rnd2 = new Random();

    int[] array_a = new int[n];
    int[] array_b = new int[n];
    for (int i = 0; i < n; i++) {
        array_a[i] = rnd.nextInt(n * 2);
        array_b[i] = rnd2.nextInt(n * 2);
    }

    int sum = 0;
```

```
        long t0 = System.nanoTime();

        for (int i = 0; i < n; i++) {
            int key = array_a[i];
            for (int j = 0; j < n; j++) {
                if (key == array_b[j]) {
                    sum++;
                    break;
                }
            }
        }
        long t1 = System.nanoTime();
        return t1 - t0;
    }
}
```

Arraystorlek	Tid (µs)	Ratio
100	62.5	1.00
200	264.9	4.24
400	532.1	8.51
800	920.2	14.72
1600	1025.5	16.41

Tabell 4: Mätningar av tiden för att hitta dubbletter i två arrayer med samma storlek i mikrosekunder



Figur 2: Graf över tiden för att hitta dubbletter i två arrayer av olika storlek