ID1021 Algoritmer och Datastrukturer
Sorterade Mängder

Dipsikha (Diddi) Dutta

September 4 2023

Introduktion

Syftet med denna rapport är att undersöka fördelen med att sortera mängder data. Detta görs genom jämförelser av tidmätningar på linjär och binär sökning. Även en jämförelse av en tredje effektivare ansedd algoritm som kräver sorterade arrayer innan den tillämpas. Den linjära sökningen går igenom hela arrayen och returnerar `true` från ett if-statement om den hittas. (standard linjär sökningsalgoritm)

Binär söknings-algoritm

Den binära sökningsalgoritmen utgår ifrån konceptet att halvera mängden data att söka igenom efter varje jämförelse. På det sättet utvecklas tidskomplexiteten med en logaritmisk funktion, log(n). Med hjälp av `min` och `max` räknas en pekare i mitten ut av varje dataintervall som succesivt rör sig mot det sökta elementet, och annars returnerar `false` om det inte finns. Detta pågår så länge `min` och `max` inte överlappar varandra.

```
public static boolean binarySearch(int[] array, int value) {
    int min = 0;
    int max = array.length - 1;

    while (min <= max) {
        int middle = (min + max) / 2;

        if (array[middle] == value) {
            return true;
        } else if (value < array[middle]) {
            max = middle - 1;
        } else {
            min = middle + 1;
        }
```

```
    }

    return false;
}
```

Even Better Algoritm

Syftet med den tredje algoritmen är att hitta dupletter av element i två sorterade arrayer. Detta görs genom att skapa en pekare för båda arrayerna med startindex `pointer1,2 = 0`. I while loopen ska den hitta dupletter så länge båda pekarna inte är på det sista elementet för varje array. Om elementen är hittade returneras `true`. Om elementet i `array1` är mindre än elementet i `array2` flyttas pekaren i `array1` till höger och annars pekaren till `array2`. Detta medför att tidskomplexiteten till algoritmen utvecklas linjärt $O(n)$ i jämförelse med om arrayerna var osorterade, vilket skulle leda till en potentiell utveckling n^2.

```
public static boolean evenBetter(int[] array1, int[] array2) {
    int pointer_1 = 0;
    int pointer_2 = 0;

    while(pointer_1 == array1.length && pointer_2 == array2.length) {
        if(array1[pointer_1] == array2[pointer_2]) {
            return true;
        } else if (array1[pointer_1] < array2[pointer_2]){
            pointer_1++;
        } else {
            pointer_2++;
        }
    }

    return false;
}
```

Benchmarken

Benchmarken är implementerad på det sättet att den mäter tiden för varje exekvering av algoritmerna för olika storlekar, som använder sig av test-elementen i arrayen `int[] sizes`. Dessa storlekar initieras som längden på olika arrayer med element som genereras av metoden `sorted()` där alla element är storleksordnade.

Instansvariabeln `array1` skapas för att testa tidtagningen för både en linjär och binär sökning för att hitta något specifikt element. Dessa element genereras i arrayen `int[] index` som slumpar fram olika element som ska

hittas i `array1`. Dessa element är slumpade i ett intervall mellan `array1`'s storlek med en faktor 5 och skapas med metoden `keys()` då elementen i `array1` slumpas i ett intervall mellan 0-9. För att minimera risken att resultatvärdet inte ska vara beskriva best-case-scenarios eller worst-case-scenarios, så slumpas 10000 olika element i `int[] index` av (`int loop = 10000`).

```
private static void benchmark() {
    int[] sizes = {100, 200, 400, 800, 1600, 3200, 6400};

    System.out.printf("# searching through an array of length n, time in ns\n");
    System.out.printf("#%7s%8s%8s%14s\n", "n", "linear", "binary", "even-better");

    for (int n : sizes) {
        int loop = 10000;
        int[] array1 = sorted(n);
        int[] array2 = sorted(n);
        int[] array3 = unsorted(n);
        int[] index = keys(loop, n);

        System.out.printf("%8d", n);

        restoreMin();
        float min1 = benchmarkLinearSorted(array1, index);
        System.out.printf("%8.0f", (min1/loop));

        restoreMin();
        float min2 = benchmarkLinearUnsorted(array3, index);
        System.out.printf("%8.0f", (min2/loop));

        restoreMin();
        float min3 = benchmarkBinary(array1, index);
        System.out.printf("%8.0f", (min3/loop));

        restoreMin();
        float min4 = benchmarkEvenBetter(array1, array2);
        System.out.printf("%8.0f\n", min4/loop);
    }
}
```

För varje algoritm räknas snitttiden ut för alla element som skulle hittas av dessa och testas `tries = 1000` gånger. Koden nedan är ett exempel på hur benchmarken för de specifika algoritmerna är implementerade och returnerar den minsta tiden som mättes för att exekvera algoritmen. Den glob-

ala variabel `min = Float.POSITIVE-INFINITY` återsälls med `restoreMin()` efter varje mätning i `benchmark()`.

```
private static float benchmarkBinary(int[] array1, int[] array2) {
    for (int i = 0; i < tries; i++) {
        long t0 = takeTimer();
        binary(array1, array2);
        long t1 = takeTimer();
        float t = (t1 - t0);
        if (t < min)
            min = t;
    }
    return min;
}
```

Resultat

n	linjär osorterad	linjär sorterad	binär	even better
100	27	27	37	182
200	47	44	64	125
400	95	88	55	121
800	173	158	62	100
1600	328	297	69	96
3200	638	578	76	83
6400	1259	1138	82	81

Table 1: Tid i nano sekunder

Diskussion

Som resultatet visar är den binära algoritmen betydligt effektivare dem andra algoritmerna som utvecklas linjärt i proportion till storleken `n`. Att finna element i både osorterade och sorterade arrayer för en array för ett stort antal slumpade element, tar avsevärt mer tid än den binära sökningen, vilket beror på att datamängden halveras för varje jämförelse. `evenBetter()` förminskar tidskomplexitetetn mer än de andra linjära algoritmerna men varför resultatet visar som det gör, att tiden blir kortare ju större array, är oklart. Samtidigt är det många andra processer som spelar roll i resultatet, bl.a bakgrundsprocesser såsom uppdateringsprogram, säkerhetskopiering, kontroller etc, vilket egentligen ska kompenseras av att ta medianvärdet av 1000 gånger, men som blev som det blev. Det experimentet visar är att det

är värt att sortera datan för tiden O(n log(n)) med binär sökning jämfört med om datan var osorterad och tillämpades av en linjär algoritm för att sökas igenom.