Fördelen med att sortera data

Mattias Sandberg

8 Sep 2022

Introduktion

I denna uppgift ska vi arbeta med att sortera olika arrays. Att leta i en array kan vara tidskrävande om inte elementen i arrayen är sortarde. Men om elementen i arrayen är sortarde så går det snabbare att hitta det önskvärda elementet.

Osorterad array

I den första uppgiften ska vi söka igenom en array som är osorterad och leta efter ett "KEY element". För att göra detta så använder jag angiven kod från sorted.pdf och implementerar en class och main funktion som delvis representeras i kod nedan. Därefter mäter jag exikveringstid för programmet att köras. Jag genererar slumpmässiga tal i en array där arrayen dubbleras för varje test som körs, och där min "KEY" jag söker också dubbleras. Nedan visas vissa delar i koden som är mest triviala.

```
for (int array = 0; array < 10; array++) {
                search_unsorted(OriginalArray, key);

                int[] newArray = new int[OriginalArray.length * 2];

                System.arraycopy(OriginalArray, 0, newArray, 0,
                OriginalArray.length);
                OriginalArray = newArray;

                long t0 = System.nanoTime();
                boolean test = search_unsorted(OriginalArray,key);
                long t1 = System.nanoTime() - t0;
            }
```

Om det plottas en graf med Array som x-axel och Nanoseconds som y-axel så representeras en linjär funktion och därmed är tidskomplexitet (Ordo n) O(n).

Array	KEY	True/False	Nanoseconds
20	1	true	500
40	2	true	380
80	4	true	380
160	8	true	330
320	16	false	5900
640	32	false	13000
1280	64	true	19000
2560	128	false	47000
5120	256	false	98000
10240	512	false	180000

Table 1: Tabell över benchmark från Osorterat programm.

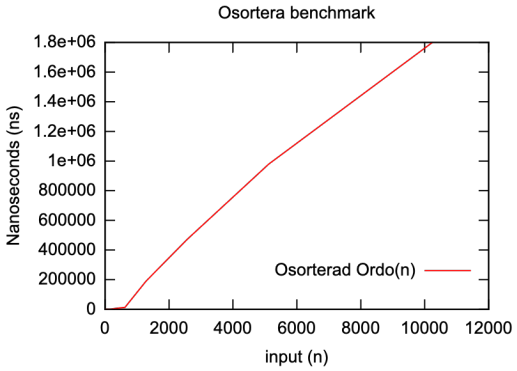

Figure 1: Graf över Osorterad array (tidskomplexitet

Sorterad array

Om vi vet att arrayen där vi ska hitta våran ”KEY value” är sorterad så kan vi optimizera programmet genom att sluta söka efter vårat önskvärda värde då vi kommer till ett element som är större än våran ”KEY”. Utan att exikvera programmet och testa så estimerar jag att tiden det tar att köra ett programm med en array som är 1 miljon lång är cirka 0.5 microsekunder då vi bara kommer behöva gå igenom 500 000 element i arrayen då vårat slumpmässiga värde i genomsnitt kommer att ligga närmast mitten och därmed kommer den genomsnitliga tiden bli cirka 0.5 microsekunder. Tidskomplexiteten för Sorterad array är O(n) fast betydligt snabbare än den Osorterade arrayen (riktningskoefficienten är inte lika stor).

Array	Nanoseconds
20	5
40	10
80	20
160	5
320	80
640	150
1280	300
2560	600
5120	1200
10240	2400

Table 2: Tabell över benchmark från Sorterat programm.

binary search

I nästa uppgift så ska det konstrueras en sorterings algorithm som är betydligt bättre än den föregående då den inte är lika kostsam för stora n-värden tidsmässigt. Denna metoden kallas för binary search och går ut på att man hoppar in i arrayen i mitten och sen kollar om man ska längre fram eller längre bak i arrayen detta betyder att man kan utesluta iprincip halva arrayen och beroende på vart värdet ligger i arrayen så kan det gå extremt fort att hitta sitt önskvärda element med hjälp av denna sorteringsalgorithm. Jag använder mig av koden från min förra mainfunktion för att få hela benchmarken automatiserad. Nedan kommer ”kod-skelettet” från pdf:en med min kod implementerad. Jag behöver inte skriva System.out.println(”false”) då min main funktion inkluderar det. Nedan visas koden:

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

while (true) {
// jump to the middle int index = ..... ;
int mid = ((max - min)/2) + min;
if (array[mid] == key) {
  // hmm what now?
    //System.out.println("true");
    return true;
      }
      if (array[mid] < key && mid < max) {
        max = mid - 1;
```

```
        continue;
// The index position holds something that is less than
// what we're looking for, what is the first possible page? first = ..... ;
      }
      if (array[mid] > key && mid > min) {
// The index position holds something that is larger than // what we're looking for
          max = mid - 1;
          continue;
}
      // Why do we land here? What shoudl we do?
      break;
  }
    //System.out.println("false");
    return false;
```

Array	KEY	true/false	Nanoseconds
50	1	true	1100
250	4	false	900
1250	16	false	800
6250	64	false	1000
31250	256	false	1100
156250	1024	false	1200
781250	4096	false	4800
3906250	16384	false	84000
19531250	65536	false	20000
97656250	5262144	false	210000

Table 3: Tabell över benchmark från binary search programm.

Vi ser enligt tabbelen att tidskomplexiteten liknar logaritmen av n, dvs Ordo (O(log(n)). Detta betyder att för större och större värden kommer tiden öka mindre och mindre, dvs att polynomet planar ut längs x-axeln. Det betyder att binary search är fördelaktigt för att snabbare hitta element i stora arrays. Programmet tar cirka 13000 nanosukender att söka igenom 1 miljon element, eftersom tidskomplexiteten är logaritmisk O(log(n)) estimerar jag att det tar cirka 38000 nanosekunder för programmet att köra igenom 64 miljoner element. Detta gjorde jag genom att använda linereg och söka för x-värde 64 miljoner.

Ännu bättre

I denna första uppgiften så ska vi hitta alla duplicerade element i två stycken osorterade arrays. För varje elemnt i den första arrayen söker vi igenom hela

den andra arrayen sökandes efter ett likvärdigt element. Tidskomplexiteten för att söka igenom båda arraysen är $O(n^2)$ om båda arraysen är n långa. Om tex ena arrayen är n lång och andra är k lång så blir tidskomplexiteten O(nk).

Array	**Nanoseconds**
20	310000
40	60000
80	170000
160	10000
320	180000
640	350000
1280	770000
2560	2200000
5120	4200000
10240	8500000

Table 4: Tabell över benchmark från duplicates programm utan binary-search.

I den andra uppgiften så skapar jag två sorterade arrays. Nu när arraysen är sortarde så kommer jag fortfarande gå igenom hela den första listan men nu så kommer vi kunna använda binary search på lista nummer 2 eftersom den är sorterad O(log(n)). Detta gör att det kommer gå betydligt snabbare för stora n-inputs och därmed kommer tidskomplexiteten att ändras från $O(n^2)$ till O(nlog(n)). Programmet som jag nu har konstruerat är betydligt snabbare och därmed mindre kostsam tidsmässigt och operationsmässigt och det är poängen med att sortera arrayen först. Nedan kommer en kort del av koden från "duplicatesbin" då rapporten börjar bli för lång.

```
for (int j = 0; j < k; j++) {
          for (int i = 0; i < m; i++) {
              keys[i] = 1;
          }
          for (int i = 0; i < n; i++) {
              array[i] = rnd.nextInt(10*n);
          }
          long t0 = System.nanoTime();
          for (int ki = 0; ki < n; ki++) {
              for (int i = 0; i < m; i++)
                  if(array[i] == keys[i]) {
                      sum++;
                      break;
```

Array	Nanoseconds
20	140
40	700
80	750
160	1700
320	3900
640	22000
1280	84000
2560	240000

Table 5: Tabell över benchmark från duplicates programm med binarysearch.

Nu ska jag skriva om algorithmen och försöka få den ännu snabbare genom att kolla med det nästa elementet i den första arrayen. Om det nästa elementet i andra listan är mindre än det nästa elementat i den första listan så går vi vidare i den andra listan. Om elementen är lika så har vi hittar våran duplikant, och om den är större så går vi vidare i den första listan och processen fortsätter. I testerna från benchmarken förutsätter vi att det inte förekommer duplikanter i dom enskilda arrayen. Jag kan nu använda binary search för båda mina arrays då dom är sorterade och detta resulterar att Tidskomplexitetn blir O(n) vilket är ett stort framsteg gämfört med föregående algoritmer. Men för att hela processen ska vara lönsam så måste sorteringen ha en tidskompexitet som är mindre än $O(n^2)$.

Array	Nanoseconds
20	110
40	190
80	410
160	830
320	1600
640	3300
1280	6500
2560	13000

Table 6: Tabell över benchmark från duplicates programm med binarysearch (final optimized.

I tabellen nedan ser vi att programmet där arrayerna sorteras i förväg utpresterar dom två andra algorithmerna och därmed är den mindre kostsam tidsmässigt och fördelaktigt att implementera i sina program.