Sökning i en sorterad vektor

Hiba Qutbuddin Habib

Spring Fall 2022

Introduktion

Att söka efter en specifik nyckel i en osorted vektor kräver lång tid, då man vill undvika tidsförlust och andra förluster så är det bättre och söka i en sorted vektor, det vill säga en vektor med en viss datastruktur. I denna Uppgiften kommer vi lära oss fördelen med sorted vektor i jämförelse med osorterad vektor.

Första Försök

Sökning genom osorterad vektor

I denna delen söker vi genom en osorterad vektor. Vektorn fylls på med slumpvis tal samt vi använder en slumpvis värde som nyckel för sökningen. Under sökning mätas tiden för växande storlek på vektor.

Resultat

I tabllen nedan kan vi se att tiden nästa dubbleras när längden dubbleras. Utifrån detta kan vi konstatera att förhållande mellan tiden och längden kan beskrivas med den linjära funktionen "$t(n) = kn + m$".

Att tiden blir inte exakt dubbel så stor kan bero på konstaten "m". Då det är termen kn som dominerar så tillhör denna algoritmen mängden $0(n)$.

Sökningen genom sorterad vektor

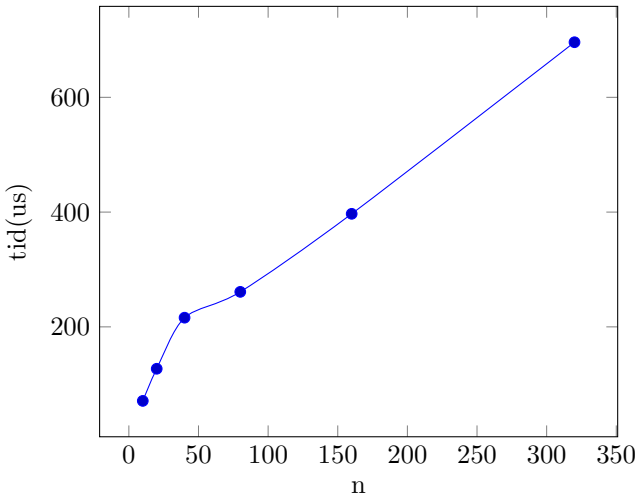
I denna delen kommer vi att söka efter en specifik nyckel element i en redan sorterad vektor. I benchmark implementeras loop samt if-satser för att avsluta sökning så fort nästa element i vektor är större än det sökte nyckel element, enligt:

n	**100**	**200**	**400**	**800**	**1600**	**3200**
Tid	19	37	72	142	289	570

Table 1: Tiden vid olika värde på n

```
for (int i = 0; i < sortedarray.length; i++) {
    if (key == sorted_array[i] ) {
        System.out.println("Found this number: " + sorted_array[i]);
            break;}
    if(key < sorted_array[i++]){
        System.out.println ("not found");
            break;}}
```

Resultat

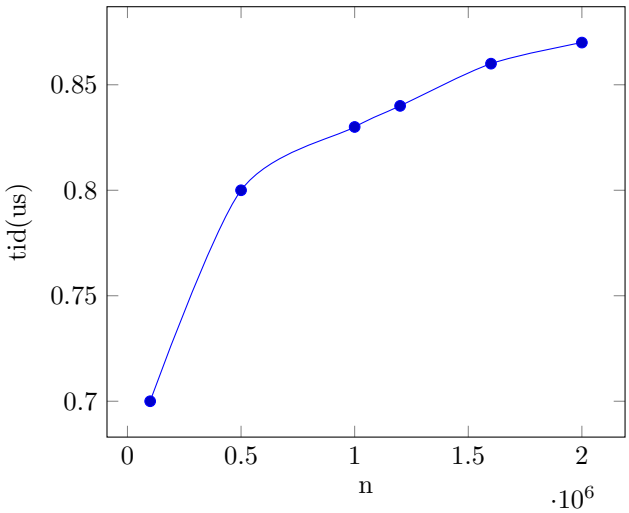


Resultatet jag får i grafen ovan beskriver en värsta fall för algoritmen. Då när jag utförde mätning fick jag nästan ingen skillnad i tid när elemenetet inte hittades vilket resulterar till en konstant tid, och då kan vi säga att algoritmen tillhör mängden 0(1), detta är rimligt då om programmet inser redan i början att den sökte nyckel element kan inte finns i vektorn, så kommer sökning att avbrutas. Däremot om den sökte elementet fanns med och programment lyckades hitta den så resultat blir likt den osorterad vektor, då varje gång längden dubbleras så dubbleras tiden också.Detta leder till att algoritmen tillhör då mängden $O(n)$ som vi kan också se i grafen ovan.

Binär sökning

Mtoden binär sökning kan beskives enligt följande: För varje gång begränsas vektor storlek enligt :$n/2, n/4, n/8$.....

Resultat



Resultatet i tabellen ovan tyder på att algoritmen binär sökning tillhör mängden $O(log(n))$. Grafen ovan visar inte en perfekt logaritm kurva, detta kan bero på mätningar jag utförde, då det var lite svårt att mäta exakt tider, vilket kan bero på hur fort den sökte nyckel element hittade, om det hittas direkt så kommer sökning att avbrytas. Däremot om programmet behöver begränsa vektors storlek flera gånger samt flytta pekar ett antal gånger så krävs det längre tid.

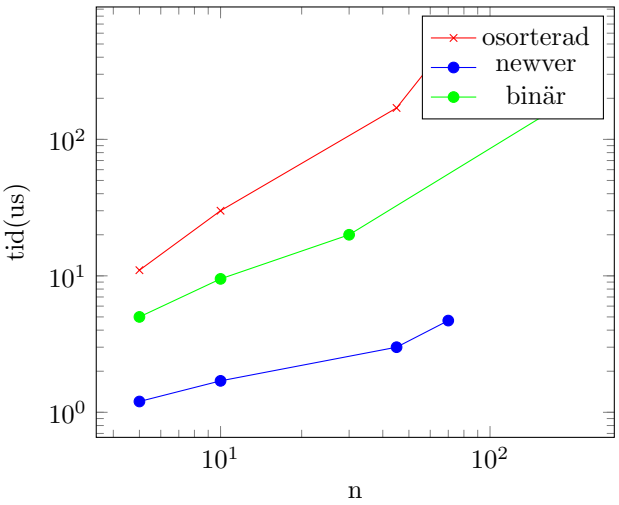
Bättre(Dubletter)

I denna delen ska vi hitta dubletter i två sorterad array. Vi implemeterar en metod som för att flytta fram eller bakåt i både vektorer, då exempelvis om det första talet i den ena vektor är större än den första talet i den andra vektor som kommer vi behöva flytta oss några steg i den andra vektor så att de första talen är lika. Det implementeras enligt nedan:

```
while (true) {
     if (i == last || j == last1) {
          break;} //I fall vi kommer till sista indexet då slutar sökning
     if (key[i] == sor[j] && i <= last && j <= last1) {
         System.out.println(" Found :" + sor[j]);
             i++;j++;//Vi steger vidare för att hitta fler dubletter
             continue;}
     if (key[i] < sor[j] && i <= last) {
         i = i + 1;
         continue;}
     if (key[i] > sor[j] && j <= last1) {
         j = j + 1;
         continue; }
```

```
        break; }
    return false;}
```

Resultat



Den första logaritmen som användes i första uppgifter gav en algoritm av typen $O(n^2)$, då för att hitta en dublette i två vektorer så krävdes det att programmet ska stega genom både vektorer, i värsta fall kommer programmet att stega genom n element i både vektorer vilket leder till att totalt kontrolleras n^2 element. Röda kurvan i grafen ovan visar funktionen för denna algoritmen, vi kan se att kurvan växer uppåt, dock det syns inte så tydligt att det är en n^2 kurva, detta kan bero på hur grafen implemterades samt det kan bero också på mätningar, då de kunde kanske ha vart mer noggranna samt koden som jag använder för att mäta tiden kan också ha påverkat resultatet, då jag tror att den bristar på något sätt.

Resultatet för den binära sökning kan vi se i den gröna kurva. Den binär algoritmen bör egentligen tillhöra mängden $0(nlog(n))$. Då eftersom vi stegar genom n element när vi söker genom den första vektor medans när vi ska söka genom den andra vektor, så används binär sökning vilket tillhör mängden $0(log(n))$. I kurvan syns inte tydligt vilket funktion vi snackar om, då detta kan beror på mätningar, när elementet ligger redan i början så hittas den direkt annars kommer tiden att bli större. Dessa problem med tiden kan bero på de slumpvisa värde jag valde för både vektorer, då dessa värde påverka positionen på elementen.

Den sista algoritmen kan vi se resultatet av den i den blå kurva. Vi kan tydligt se att ökningen av tiden där sker mycket saktare jämfört med de andra. Jag skulle säga att denna algoritmen tillhör mängden $O(n)$, då mina mätningar visar att tiden blir ungefär dubbel så står när längden dubbleras, detta är rimligt för vid sökningen stegar inte programmet genom både vektorer, dena stegar genom en i taget vid behov.