Binärsökning och Interpolationssökning

1. Inledning

Denna rapports syfte är att ge en överblick och identifiera brister samt fördelar med de två sökalgoritmerna Binärsökning och Interpolationssökning. Rapporten kommer också belysa de olika algoritmernas föredragna användningsområden samt analysera algoritmernas tidskomplexitet i teorin jämfört med praktiska resultat.

1. Algoritmerna i detalj

2.1 Binärsök

Binärsök är en av de absolut vanligaste och mest använda sökalgoritmerna i datavetenskap. Algoritmens tillvägagångssätt är att den utifrån en sorterad vektor, delar längden av vektorn på hälften och jämför nyckeln (elementet som söks) med det mittersta indexets värde, skulle indexets värde matcha nyckelns värde så är sökningen lyckad och algoritmen returnerar detta index. Om indexet inte matchar nyckelns värde så tar algoritmen bort den hälft där nyckeln inte kan befinna sig på. På detta viset arbetar algoritmen tills nyckeln är funnen. För att beräkna effektiviteten för binärsök så räknar man antalet gånger nyckeln man söker jämförs med ett index värde i vektorn. Best case för binärsök är att nyckeln befinner sig på vektorns mittersta index (Theta(1)) innan vektorn delats första gången. I och med att algoritmen alltid delar inputen med hälften för varje iteration så kan man konstatera att worst case för algoritmens tidskomplexitet aldrig blir mer än Theta(log n). Därför kommer körningstiden att stadigt öka med endast en operation vid fördubblad input. Average case för binärsök är när nyckeln man söker finns i vektorn och är något bättre än worst case, ungefär (log2n - 1) sökningar behövs. Medans average case för en nyckel som inte finns i vektorn kräver ungefär (log2(n + 1)) sökningar (Levitin 2012).

Rekurrensrelationen för binärsök kan skrivas:

T(n) = T(n/2) + O(1).

Beräkning med hjälp av Master Teoremet för divide and conquer så kan man konstatera även här att tidskomplexiteten för binärsöks worst case är T(n) = Theta(log n).

2.1.1 Pseudokod Binärsök

//Implementation av en iterativ binärsök //Input: En vektor V[0..n − 1] sorterad I stigande ordning. //Sökning efter en nyckel N.  
//Output: Det index där nyckeln befinner sig, eller −1 om nyckeln inte befinner sig i vektorn V.

l ← 0 ; r ← n − 1 **while** l ≤ r **do** m ← ⌊ ( l + r ) / 2 ⌋  
 **if N** = V[m] **return** m **else if** K < V [m] r ← m − 1 **else** l ← m + 1 **return** − 1

2.1.2 Binärsök exempelkörning

Nyckel = 16

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 75 | 100 |

Vektor =

**L** =0 **M**=5 **R** =10

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 75 | 100 |

**Nyckel** (16) < **M** (17), ta bort övre halvan.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 75 | 100 |

**L** = 0 **M**=2 **R**=4

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 75 | 100 |

**Nyckel** (16) > **M**(4), ta bort nedre halvan.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 75 | 100 |

**L**,**M**=3 **R**=4

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 75 | 100 |

**Nyckel** (16) > **M**(8), ta bort nedre halvan.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 75 | 100 |

**L,R,M=4**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 75 | 100 |

**Nyckel** (16) == **M**(16), lyckad sökning, element hittat på index 4.

2.2 Interpolationsök

Interpolationsök är lik Binärsökningen på många sätt men skiljer sig i val av index där vektorn delas vid varje iteration. Interpolationssökning är en typ av ”variable-size-decrease” algoritm som beräknar med hjälp av nyckelns värde, vart delningspunkten i vektorn bör hamna. Man kan jämföra algoritmens tillvägagångssätt med hur en människa normalt sätt hade sökt efter ett ord i en ordbok, letar vi efter ordet ”Ödla” så öppnar vi inte ordboken i mitten (som t.ex binärsök hade sökt efter ett ord) utan börjar söka ganska långt bak i boken.

Interpolationssöks tillvägagångssätt börjar med att beräkna med hjälp av det sista indexet (samt indexets värde) och det första indexet (samt indexets värde) för att hitta en plats i listan att dela. Algoritmen antar samtidigt att elementen i vektorn ökar linjärt vilket påverkar körningstiden, men inte korrektheten hos algoritmen (Levitin 2012). Efter algoritmens beräkning av vilket index nyckeln borde befinna sig på så jämförs nyckeln med indexets värde, är värdet på indexet samma som nyckeln så returneras indexet och algoritmen avslutas. Visar det sig att värdet på indexet är större än nyckeln så tas den övre delen av vektorn bort, är värdet mindre än nyckeln så tas den nedre halvan bort,. Algoritmen fortsätter arbeta på detta viset tills att nyckeln hittas eller tills att algoritmen konstaterat att nyckeln inte förekommer i vektorn. I och med att en ny beräkning av index görs vid varje iteration genom algoritmen så blir sökningen något listigare än binärsök, observera dock att detta endast sker om datan som genomsöks är uniformt distribuerad. Alltså bör man föredra att använda interpolationssök om man vet på förhand att datan som genomsöks är sekventiell ordnad, vet man istället att elementen i datan är oregelbunden i förhållande till varandra eller det förekommer väldiga skiftningar i storlek på elementen bör man binärsökning föredras före interpolationssökning. Tidskomplexiteten för interpolationssök i best case är när datan är sekventiellt sorterad och kommer då vara Theta(log log n)**.** Worst case skulle vara att alla element i datan är uniformt distribuerade förutom det sista som istället är avsevärt mycket större än det näst sista. Detta leder till att beräkningen av index i algoritmen blir maximalt skev och tidskomplexiteten hamnar på Theta(n).

2.2.2 Pseudokod Interpolationssök

//Implementation av en iterativ interpolationssök //Input: En vektor V[0..n − 1] sorterad I stigande ordning. //Sökning efter en nyckel N.  
//Output: Det index där nyckeln befinner sig, eller −1 om nyckeln inte befinner sig i vektorn V.

l ← 0 ; r ← n − 1; **while** l ≤ r & N > = V[l] & N < = V[r] **do** pos ← l + (n - V[l]) \* (r - l) // (V[r] - V[l]) **if** l **←** r **if** V[l] ← N; **return** l **return** - 1 **if** V**[**pos**] ←** N; **return** pos **if** V[pos] > N; r ← pos − 1 **else** l ← pos + 1 **return** - 1

2.2.3 Interpolationssökning exempelkörning

Nyckel = 16

Vektor =

**L** =0, **V[L]** = 1 **R** =10, **V[R]** = 60

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 55 | 60 |

Beräkning av index: **POS** = **L(**0) + (**N**(16) - **V[L]**(1) x **R**(10) - **L**(0)) // **V[R]**(60) - **V[L]**(1) = 2.54 = **2**

**Nyckel** (16) > **V**[**POS**]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 55 | 60 |

**L** = **POS** + 1, **V[L]** = 8 **R** =10, **V[R]** = 60

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 55 | 60 |

Beräkning av index: **POS** = **L(**3) + (**N**(16) - **V[L]**(8) x **R**(10) - **L**(3)) // **V[R]**(60) - **V[L]**(8) = 3.28 = **3**

**Nyckel**(16) > **V[POS]**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 55 | 60 |

**L** = **POS** + 1, **V[L]** = 16 **R** =10, **V[R]** = 60

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 55 | 60 |

Beräkning av index: **POS** = **L(**4) + (**N**(16) - **V[L]**(16) x **R**(10) - **L**(4)) // **V[R]**(60) - **V[L]**(16) = **4**

**Nyckel** (16) == **V[POS],** returnera index, lyckad sökning.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 8 | 16 | 17 | 20 | 40 | 50 | 55 | 60 |

3. Designval

Implementeringen gjordes i Python och experimentet är uppdelat i olika moduler för att få en tydlig struktur på de olika stegen i analysen av de två algoritmerna. Main funktionen är indelad i en egen modul som innehåller ett initierande av en dictionary som innehåller två nycklar i form av de två olika algoritmerna. I main modulen initieras också en dictionary för sparandet av de listor innehållande data som ska användas i undersökningen av sökalgoritmernas prestanda.

I koden för algoritmerna finns inga specifika egna designval förutom en implementering av en räknare som räknar antalet kritiska operationer för varje sökning, förutom detta så följer implementeringen pseudokoden för algoritmen, skillnaden när algoritmen funnit eller inte funnit nyckeln så istället för att returnera ett index, så returnerar algoritmen antalet kritiska operationer för varje sökning.

I och med att det endast är antalet sökningar som sparas för varje exekvering i en lista så blir minneskomplexiteten för själva sökningen O(1), detsamma gäller för den genomsnittliga totalmängden av kritiska operationer samt längden av listorna.

De två olika bilderna för varje datamängd representerar dels en genomsnittlig beräkning av antalet kritiska operationer för sökningar av alla element som befinner sig i datan som genomsöks samt ett histogram som illustrerar hur många sökningar som krävdes för att hitta ett element.

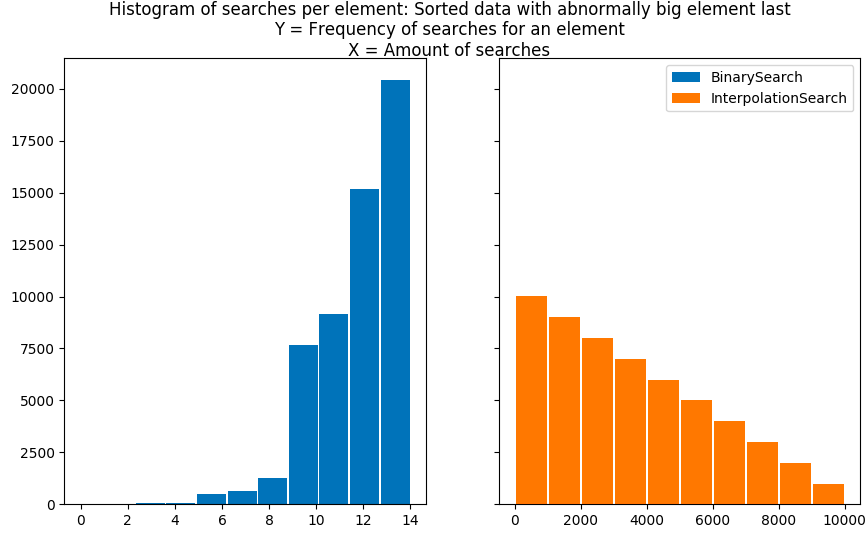
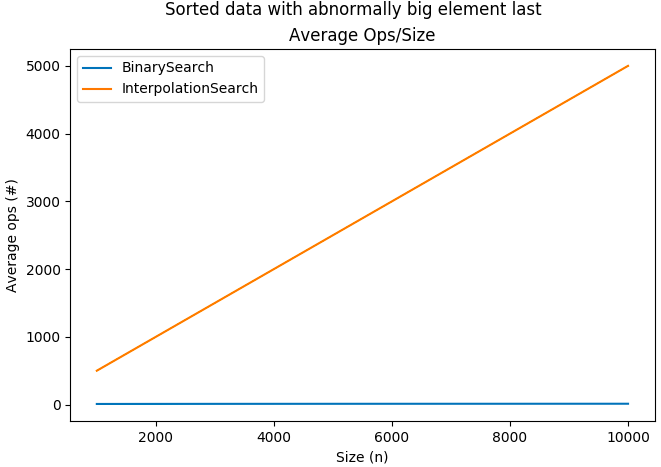
4. Experimentplanering

Experimentet är utformat för att köra fyra olika datamängder tio gånger var med de vektorer där datamängden ökar stegvis, storleken på vektorn startar med 1.000 element och ökar successivt med 1000 efter varje iteration fram tills att datamängden nått en storlek på 10.000 element. För varje element i dessa tio listor så används varje element som nyckeln i sökningen varsin gång, detta valet gjordes för att få ett verklighetstroget genomsnitt och ett utförligt histogram. Denna implementation valdes för att få en tydlig bild på hur inputs ordning och storlek påverkar algoritmernas prestanda.

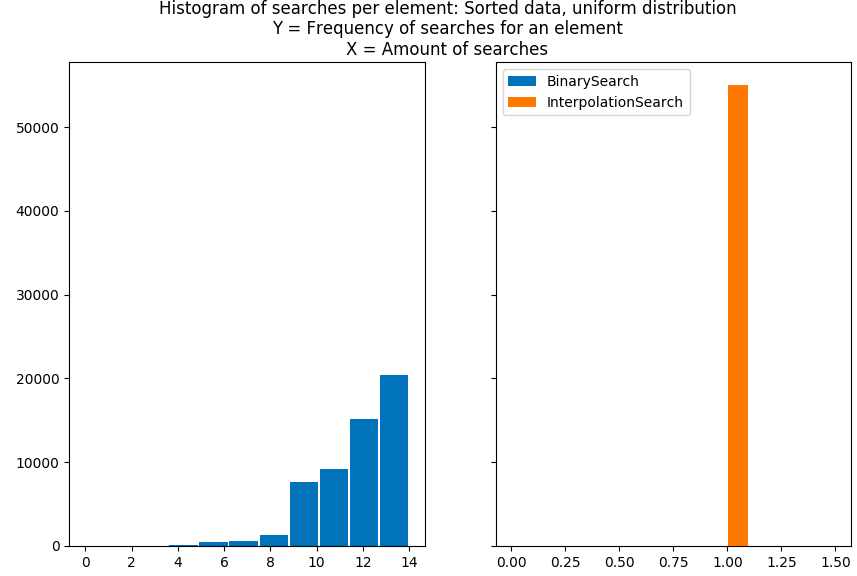
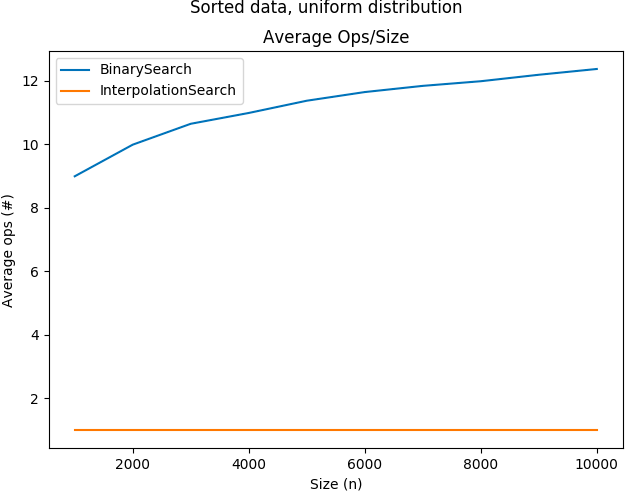
De fyra implementerade datamängderna består av ett scenario som är best case för interpolationssökning där datan är sekventiellt ordnad då ökningen mellan varje element är linjär, den andra datamängden består av genererad data som är slumpmässigt uniformt distribuerad vilket är ett ungefärligt average case för interpolationssök, den tredje datamängden består av en datamängd som är sekventiell ordnad med ett avsevärt mycket större element på sista index, detta för att testa ett scenario som blir worst case för interpolationssökning. Den sista datamängden är en datamängd utformad efter hur en normalfördelning ser ut, där det finns få förekomster av element runt min och max i vektorn, elementen är istället fördelade runt medelvärdet av vektorns storlek. Samtliga datamängder är utformade för att testa prestandan specifikt för interpolationssök och sedan jämföra prestandan mot binärsökning, detta val gjordes för att tidskomplexiteten för binärsök aldrig kan bli sämre än theta(log n) och inte påverkas av hur elementen i datan är ordnade.

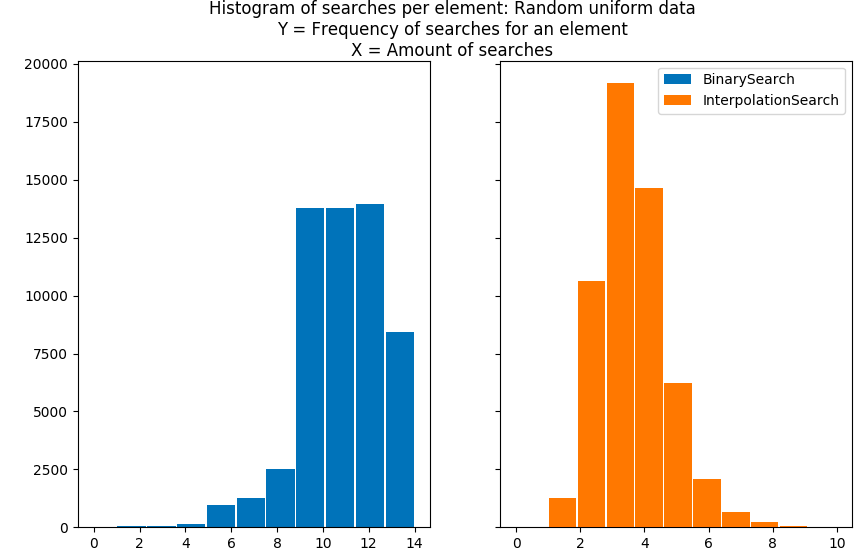
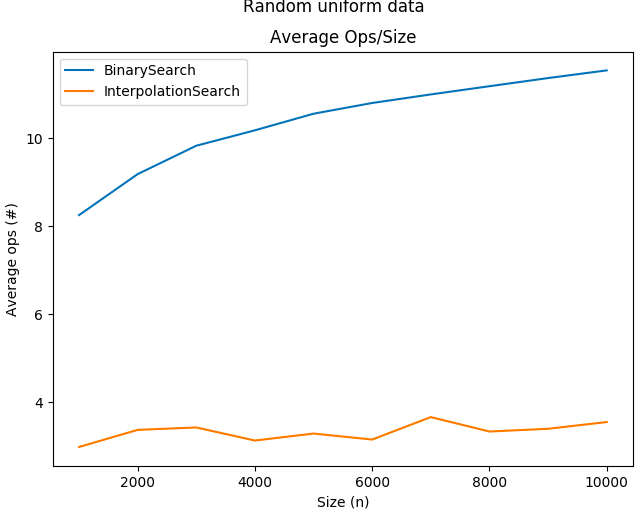
5. Resultat

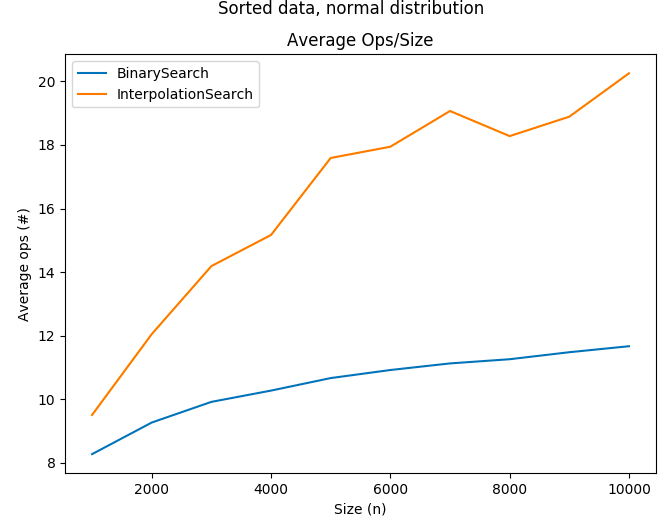
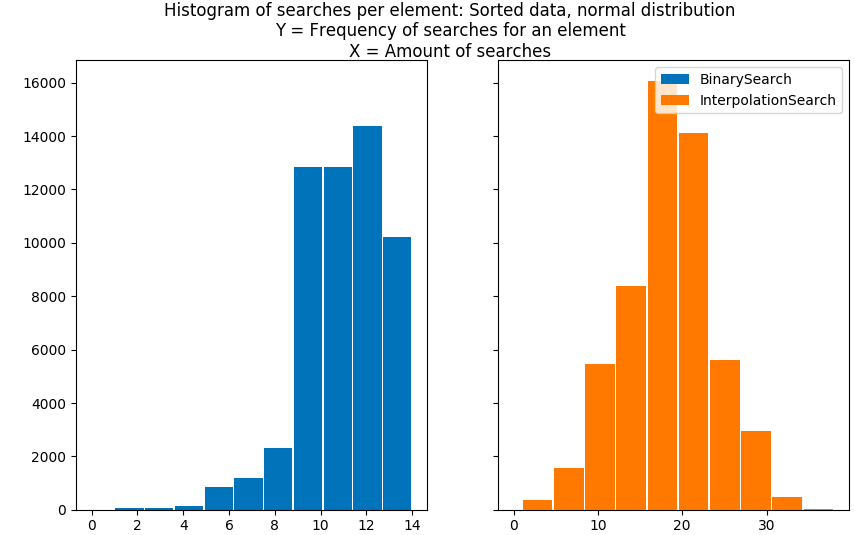
5.1 Sorterad data med stort element på sista index

****

**5.2 Uniformt ordnad data**

****

**5.3 Slumpmässig uniformt ordnad data**

**5.4 Normalfördelning på data**

6. Analys

Efter att experimentet genomförts så kan man dra direkta paralleller mellan algoritmernas teoretiska komplexitet och deras komplexitet i praktiken. Observera att x-axeln i histogrammen för de olika algoritmerna har olika storlek, detta för att tydligare kunna se hur många element som tog ett visst antal sökningar att finna då skillnaden var stor i best och worst case för algoritmerna. Tittar man på de två första bilderna som är resultatet från algoritmernas resultat i sökning av vektorer med uniform fördelning, men med ett mycket stort element på sista index, så kan man se hur binärsök har en betydligt bättre prestanda än interpolationssök i alla avseenden. Resultatet var väntat då datamängden var planerad att testa worst case för interpolationssök. Binärsök har en tidskomplexitet på O(log n) men ser nästan ut att ha en konstant tidskomplexitet på bilden, detta är på grund av att interpolationssök har en linjär tidskomplexitet och ökar parallellt med storleken på input vilket får binärsökningens ökning att se minimal ut. Analyserar man histogrammet så kan man se att binärsök oftast hittar nyckeln i spannet 9-14 sökningar, vilket är en tydlig parallel med algoritmens teoretiska worst komplexitet Theta(log n) som säger att med en input storlek på 10.000 behövs som mest 14 sökningar för att hitta nyckeln. Interpolationssöks resultat stämmer också överens med dess teoretiska worst case tidskomplexitet, antalet sökningar som krävs för att hitta nyckeln följer en linjär komplexitet.

Vid analys av resultaten från algoritmernas tester på data som är uniformt distribuerad, vilket är best case för interpolationssök i teorin, så kan vi konstatera att teorin stämmer överens med experimentets resultat. I den första bilden kan kan vi se hur binärsök kräver en extra sökning när input storleken dubblats, vilket är en logaritmisk ökning. Granskar man interpolationssökningens resultat så ser vi att den har en konstant tidskomplexitet vilket stämmer med algoritmens teoretiska best case komplexitet Theta(1). Histogrammets resultat visar också på att teorin stämmer överens med de praktiska resultatet, binärsökning hittar nyckeln på samma spann som för datamängden med ett mycket stort element på sista index. Histogrammet visar hur interpolationssök hittar nyckeln efter en sökning vid varje sökning vilket är best case för algoritmen.

De två sista datamängderna innehåller dubbletter av element, detta för att testa ett ungefärligt ”average case” för algoritmerna. Datamängden innehållande slumpmässigt uniformt ordnad data visar tydligt i sin första bild hur binärsök ökar logaritmiskt med storleken på input sett till genomsnittliga kritiska operationer. Interpolationssök har en lite skiftande ökning liknande log log n som är dess average case vid slumpmässigt uniformt fördelad data. Granskar man histogrammet så ser man att resultatet för binärsök liknar tidigare histogram, interpolationssökning kräver dock färre sökningar för att hitta nycklarna, vilket är en direkt parallel med algoritmernas teoretiska komplexitet som konstaterar att interpolations (log log n) har bättre average case komplexitet än binärsök (log n).

De avslutande bilderna är resultatet från experimentet med datamängder bestående av normalfördelad data. Binärsök har en logaritmisk ökning i genomsnittliga kritiska operationer medans interpolationssök skiftar något men kräver desto fler sökningar. Anledningen till detta är för att datamängden inte är sekventiell utan fler element befinner sig i klustret runt medelvärdet av storleken på input. Detta illustrerar tydligt att binärsök är en mer effektiv algoritm vid sökning i normalfördelad data. Histogrammet visar samma resultat för binärsök som på samtliga datamängder. I och med att datamängden har många element i ett kluster runt medelvärdet av vektorn så blir beräkningen av index för interpolationssökning skev och kräver fler sökningar än binärsök.

7. Slutsatser och diskussion

Avslutningsvis så kan man konstatera utifrån experimentresultaten att den teoretiska beskrivning av de två algoritmerna som denna rapport behandlat stämmer väl överens med resultat i praktiken. Algoritmernas användningsområden är flera, men i de fall då man vet hur datan som ska genomsökas ser ut så kan man välja att föredra den ena före den andra. Är man osäker på hur datan är ordnad så är rekommendationen att använda sig av binärsök då man vet att algoritmen inte kan ta längre än Theta(log n) tid. Vet man istället att datan är sekventiellt ordnad så kan man utifrån experimentets resultat konstatera att interpolationssökning är att föredra. Skulle datan vara slumpmässigt uniformt fördelad så är även här interpolationssökning att föredra. I fallet att datan skulle bestå av normal fördelade värden eller oregelbundet stora element mot slutet av datan så är binärsök en överlägsen preferens, då interpolationsök blir lika ineffektiv som en ordinär linjärsökning.

8. Referenser

Levitin, A. (2012). *Introduction to: The Design & Analysis of Algorithms, 3rd Edition*. New Jersey: Pearson Education, Inc., publishing as Addison-Wesley.