[LABORATION 1]

23 november 2015

Victor Persson, Viktor Fällman, Adam Lowert
Luleå Tekniska Universitet
971 87 Luleå
Algoritmer och Datastrukturer, D0012E

Innehåll

Inledning 1
Mätdata 2
Diskussion 3
Spekulationer 5

INLEDNING

Syftet med denna laboration var att implementera och analysera två stycken olika sorteringsalgoritmer. Dessa två algoritmer bygger på mergeSort, vilket är en sorteringsalgoritm. Dessa algoritmer kommer att implementeras i Python kod och modifieras på sådant sätt att när sublistorna blivit mindre än k, så kommer antingen insertion sort, nedan hänfört till som "Insertion, ev. InsertionSort", eller binary insertion sort, nedan hänfört till som bInsertion, ev. bInsertionSort", att användas för att sortera sublistorna.

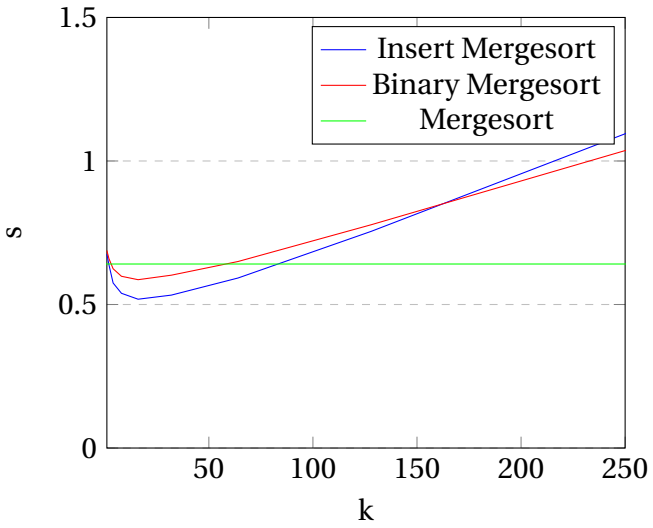
Worst case tid för både bInsertion samt Insertion är $\Theta(n^2)$. Detta gäller då varje element som mest kan swappas med $n-1$ andra element för att kunna placeras rätt i listan.

Skillnaden mellan bInsertion och Insertion är att bInsertion endast behöver jämföra varje element med $\log_2(n)$ element, medans Insertion kommer, i värsta fall, behöva jämföra med n element.

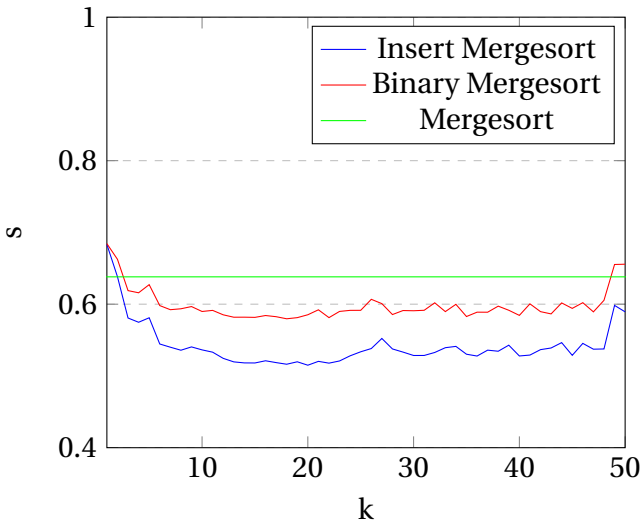
Trots detta så kommer de båda ha en worst case tid på $\Theta(n^2)$ då de andra värdena blir obetydliga för stora värden på n. Sedan har vi mergeSort vars worst case tid ligger på $O(n * \log(n))$, då varje element kommer jämföras med maximalt $\log(n)$ andra element. Faktumet är att vad som är den egentliga worst case tiden är $\Theta\left(n\log_2\left(\frac{n}{k}\right)\right)$, men för vanlig mergeSort så kommer k alltid vara 1, då listorna halveras tills endast 1 element återstår och då returnerar dem. Den teoretiska ekvationen på $\Theta\left(n\cdot k + n\log_2\left(\frac{n}{k}\right)\right)$ kommer från det faktumet att vi använder dessa både mergeSort samt en version av insertionSort. Från denna ekvation kan vi se att $n\cdot k$ kommer från insertionSort, men till skillnad från vanliga insertionSort som har $n\cdot n$, så kommer denna version bara ha $n\cdot k$ för att varje element kommer endast som swappas med de $k-1$ andra elementen i sublistan, och därför får vi $n\cdot k$. Från mergeSort ser vi $n * \log_2\left(\frac{n}{k}\right)$ vilket kan skrivas om som $n * \log_2 n - n\log_2 k = n(\log_2 n - \log_2 k)$. Anledningen till denna ekvation är att $n\log_2(n)$ kommer vara tiden det tar att sortera n element, medans $n\log_2(k)$ är tiden som inte behövs räknas med då den tas om hand av insertionSort funktionerna. Från denna worst case tids ekvation skulle man kunna anta att större k leder till större tidsåtgång, och därför sämre effektivitet. Och ett k värde på 1 borde bli optimalt, men det finns konstanter som kan ändra detta, men för stora värden på n borde de bli minimala och inte märkas av. Vi borde också se två olika kurvor, en för lägre k som är logaritmisk, och en för större k som har konstant lutning.

MÄTDATA

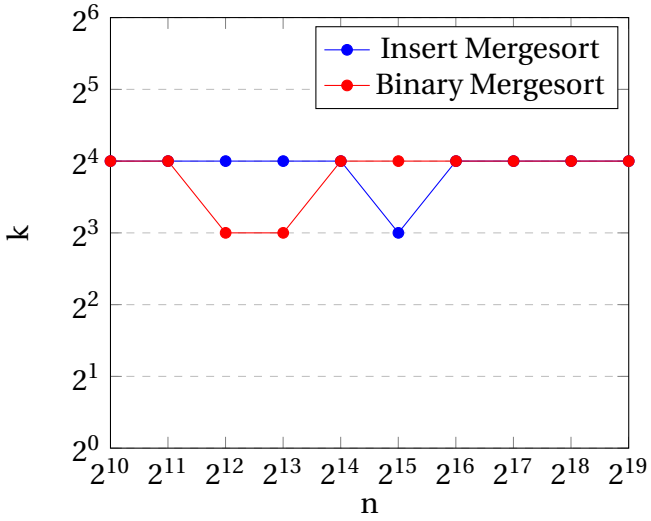
Figur 1: Python, n:100000, logarithmic increase



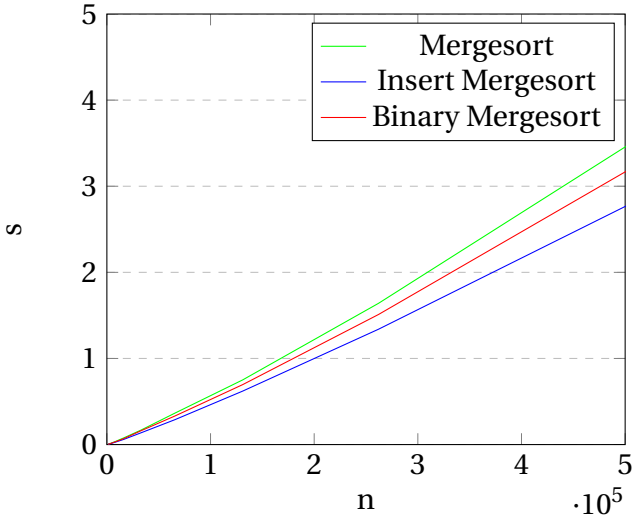
Figur 2: Python, n:100000, k:1-50



Figur 3: Python, Optimal k



Figur 4: Python, k:16, n:1024-524288, logarithmic increase



DISKUSSION

Den implementationen av mergesort som vi använder, kommer att skapa nya listor för varje rekursivt anrop, till skillnad från andra implementationer som använder index eller pointers för att skicka listorna rekursivt. Detta leder till att minnesanvädningen kommer växa med $n\log_2(n)$ på en vanlig mergesort, men med våran implementation så kommer de sista $n\log_2(k)$ inte att köras rekursivt. Därför kommer vi se att för större värden på k kommer minnesanvändandet minska medans tiden ökar. En implementation av index hade kunnat ge

ett mindre minnesanvändande men för att underlätta kodskrivandet för denna laboration så använde vi nya listor. Det hade också kunnat minska på den totala tiden om vi istället för att kopiera/skapar en ny lista, bara skickar med index, för det hade lett till mindre operationer som ska genomföras.

Vi kan enligt figur 1 se att för låga värden på k kommer Insertionsort att arbeta snabbare än bInsertionsort. Detta då fasta konstanterna som inte finns med i de teoretiska ekvationerna kan vara större för bInsertionsort, och vid större k så kommer konstanten för k värdena att vara mindre. Detta leder till att dessa funktioner kan implementeras att användas när de är som bäst, vid låga k värden så kan Insertionsort användas, och för höga k värden så används bInsertionsort.

Vi har i våra tester dels provat att för ett fast antal element n, köra igenom för olika k värden. Det vi snabbt insåg är att vi får vad som kan ses som en "trappa" med mätvärden. Detta då vi delar upp listan så måste vi multiplicera k med 2 för att få nästa del, vilket vi gjorde på figur 1. Där kan vi se att för riktigt låga k så har vi höga mätvärden, sen sjunker värdena till ett optimalt k värde för att sedan sakta stiga. Detta betyder att för denna algoritm så är det inte mest optimalt med lägsta k värden som vi ansåg från början, vilket skulle ge en funktion som liknar mergesort. Detta gav den intressanta frågan, finns det något optimalt k värde, och är detta värde beroende på n?

Vi körde igång lite tester och fick ut en figur 3, där vi snabbt kunde se att det optimala k värdet ej är beroende på n, samt att de båda funktionerna har samma optimala k värde. Detta tror vi beror på att för listor på 16 (med 8 väldigt nära i tid) element, kommer Insertionsort samt bInsertionsort att vara snabbare än mergeSort, men så fort elementen blir för så blir mergeSort snabbare. Med våran implementation hade man dock varit tvungen att överväga snabbhet mot minnesanvändning, då ett så lågt k värde för väldigt stora listor hade använt väldigt mycket minne.

Faktumet är ju som vi fastslog i teorin att de båda funktionerna kommer ha en worst case time som följer samma kurva, och att den faktiskta skillnaden mellan Insertionsort och bInsertionsort endast ligger i hur platsen i listan lokaliseras. Då förstår man varför för låga k-värden så kommer listorna vara likvärdiga, men desto större k värden leder till större sublistor som ska sorteras av insertionSort algoritmerna och denna förbättring som finns i bInsertionsort växer fram och blir tydlig.

SPEKULATIONER

Då våran implementation är gjort i Python har vi ej kunnat mäta den faktiska minnes allokeringen som krävs, utan kan endast räkna teoretiskt på detta. Samt så är vår mergeSort uppbyggd på sådant sätt att vi inte nyttjar index. Istället görs nya listor, vilket resulterar i att minnet som krävs ökar markant.
Med hjälp utav ekvationen nedan kan antalet sublistor räknas med 2^x ut vid satta värden på k och n.

n = Antal element
k = Max antal element i del lista
x = Antal halveringar

$$\frac{n}{2^x} \leq k$$

$$x \geq \frac{log(\frac{n}{k})}{log(2)}$$

$x \cdot n$ = minnesanvändning

Så för att sedan få antal listor genererade tar vi: $2^{\lceil x \rceil}$ Med avseende på detta framgår det att våran lösning utav mergeSort är långt ifrån optimal, men mycket lättare att programmera.
Hade fokus varit optimering med avseendet på minnet så hade en standard lösning med index och endast en array varit det självklara alternativet (I annat språk).
Angående insertionSort så hade en linked list varit ett bättre val för optimering utav algoritmen, men det hade varit förödande för binaryInsertionSort.