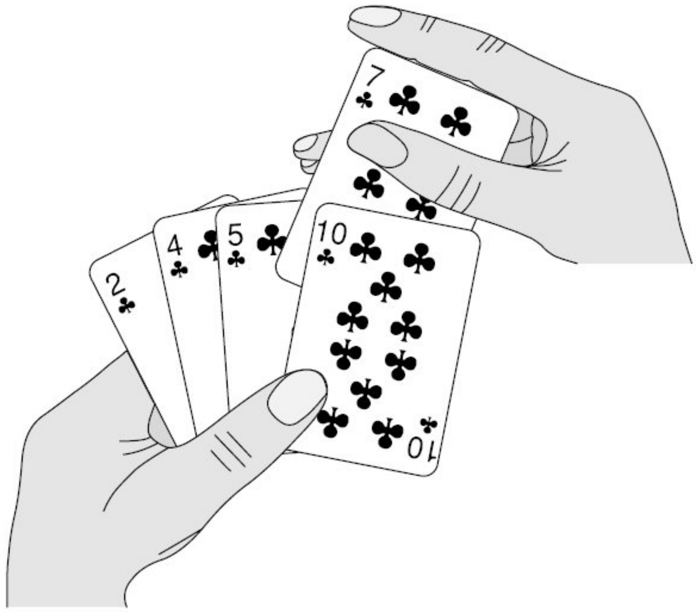
Algoritmer och datastrukturer, laboration 1



Axel Kärnebo, axekrn-7@student.ltu.se
Rickard Bemm, ricbem-8@student.ltu.se
William Gradin, wilgra-8@student.ltu.se

25 november 2019

Del I - Teori

I denna laboration modifieras den redan existerande algoritmen InsertionSort i ett försök att ge en mer tidseffektiv algoritm för att, efter storlek, sortera en lista med tal. De modifikationer som appliceras är att sökfunktionen i InsertionSort ersätts med den betydligt effektivare Binär sök funktion.

InsertionSort

Sorterings algoritmen insertionSort går ut på att linjärt jämföra ett element med en sorterad lista för att hitta elementets plats i listan. Om det element lista[p], där $1 \leq p \leq length(lista)$, är mindre än lista[$p-1$] så byter vi plats på dessa. Denna jämförelse upprepas tills det att lista[p] är större än lista[$p-1$] och upprepas för alla element i listan.

Instruktion	**Kostnad**	**Gånger**
Ta nästa element i listan	c_1	1
Byt plats på element om det valda element är mindre än föregående tills föregående element är större	c_2	n
Repetera för resterande element	c_3	$n-1$

Summan av komplexiteten blir då

$$n \cdot (n-1) \implies \Theta(n^2)$$

bSort

Algoritmen liknar insertionSort då den går igenom varje element i en given lista och sedan söker efter ett index i listan där det nya elementet ska placeras. Men istället för att gå igenom varje element för att se var elementet lista[p], där $1 \leq p \leq length(lista)$, så görs en binär sökning och ger ett index, hädanefter i, där lista[p] ska placeras. För att kunna placera lista[p] på i så måste alla element med index $i \leq length(lista)$ flyttas. Detta upprepas för resterande element.

Skillnaden mellan insertionSort och bSort är sökningen efter vilket index som elementet ska placeras på. Sökningen i den bSort algoritmen är mer tidseffektiv jämfört med insertionSort, dock gör delen där element måste skiftas för att göra plats för lista[p] att komplexiteten blir densamma som insertionSort.

Instruktion	**Kostnad**	**Gånger**
Ta nästa element i listan	c_1	1
Hitta index där det valda element är mindre eller lika med föregående med binär sökning	c_2	$\log n$
Gör plats för valt element	c_3	n
Repetera för resterande element	c_4	$n-1$

Summan av komplexiteten blir då

$$(\log n + n) \cdot (n-1) \implies \Theta(n^2)$$

MergeSort - Modifierad

Denna algoritm är en modifierad mergeSort. Den delar upp en given lista i sublistor av storlek k och mängden sublistor blir $\frac{n}{k}$. Varje sublista sorteras med hjälp av insertionSort eller bSort och sammanfogas sedan parvis med hjälp av standard mergefunktioner.

Instruktion	**Kostnad**	**Gånger**
Dela in given lista i ett antal sublistor	c_1	1
Sortera varje sublista med insertionSort eller bSort	c_2	$k \cdot n$
Sammanfoga varje sublista	c_3	$n \cdot \log \dfrac{n}{k}$

Sorteringen kostar alltså $c_2 \cdot k \cdot n$ eftersom det blir $\frac{n}{k}$ antal listor, insertionSort och bSort tar $\Theta(n^2)$ och det sorteras k element för varje sublista så ekvationen blir

$$\frac{n}{k} \cdot k^2 = k \cdot n \tag{1}$$

Mängden sammanfogningar blir totalt $\frac{n}{k} - 1$. För varje sammanfogning krävs $\lceil \log \frac{n}{k} \rceil$ jämförelser vilket ger oss kostnaden

$$n \cdot \log \frac{n}{k} \tag{2}$$

Tidskomplexiteten blir då

$$1 + k \cdot n + n \cdot \log \frac{n}{k} = a \cdot n \log n + bn + c \implies \Theta(n \log n)$$

Detta eftersom den termen som innehåller största ordningen av n är $a \cdot n \log n$ vid låga k. När k närmar sig n krymper termen med a och den största termen blir istället $b \cdot n$.

Del II - Implementation och experiment

Implementation

Algoritmerna implementerades i Java på grund av att dess starkare typsystem jämfört med Python som gör att färre typfel uppstår vid tidigt skede i implementationen.

Testerna utförs med varierande storlek på inmatade data, en lista med heltal, från 100 element upp till 100 000 element för att testa dess prestanda för både små och stora datamängder. För mergeSort algoritmerna varierade även längden på sublistorna. Dessa listor är slumpgenererade med tal $0 < n < 100\ 000$ och dess sorteringsgrad varierar som slumpgenerade samt sorterade i växande och avtagande ordning. Varje test av respektive sorterings algoritm utförs 100 gånger för att minimera möjliga felkällor.

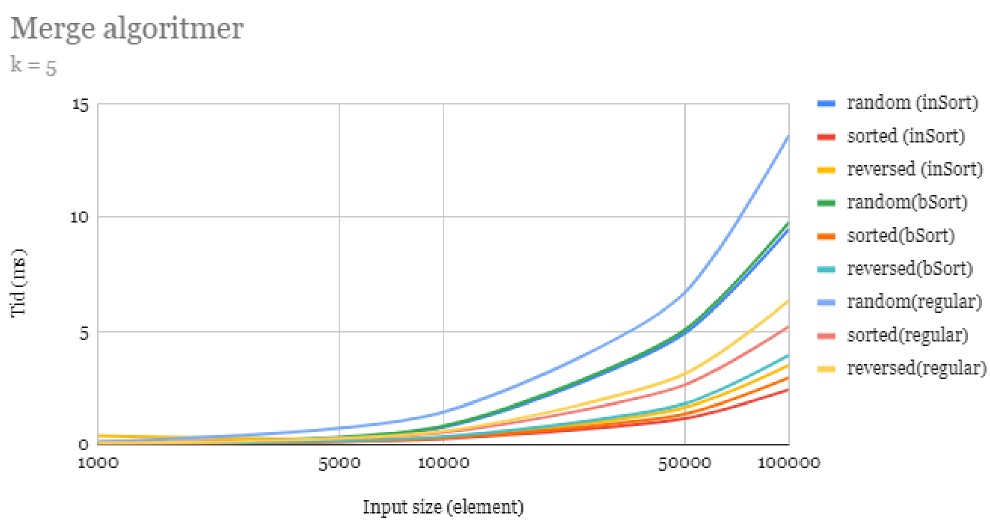
Exekveringstiden för respektive test mättes i millisekunder och fördes därefter in i kalkylark för vidare analys, se Resultat.

Resultat

Här följer tabeller och grafer utav testresultaten från utförda simuleringar.

Tabell 1: Exekveringstid, mätt i ms, för merge algoritmer med varierande input storlek n och konstant antal sublistor $\frac{n}{k}$ med längd k. Inputlistan innehåller positiva, reella heltal som är slumpgenerade, sorterade efter storlek från minsta till största och största till minsta.

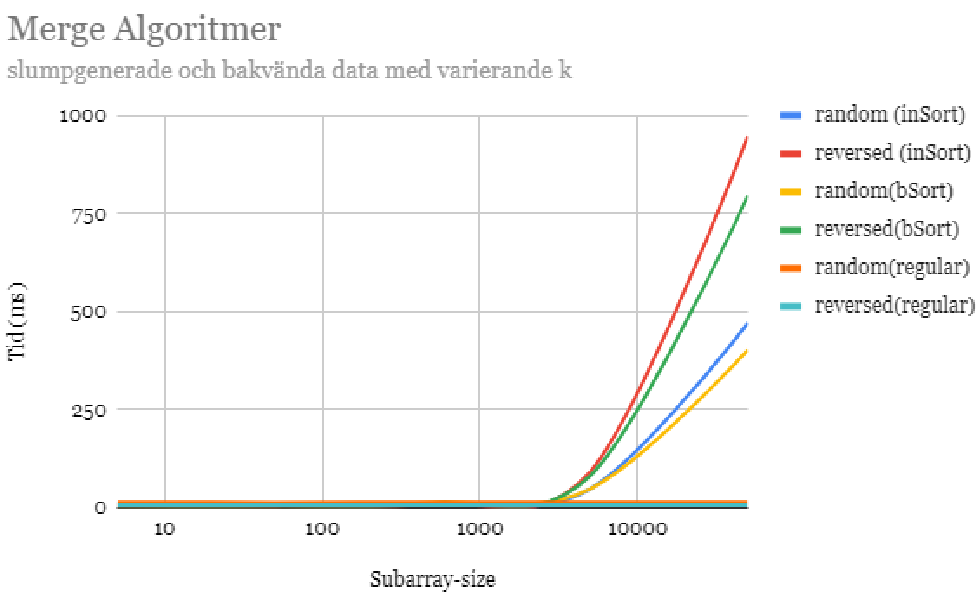
	k	5	5	5	5
	n	1000	10000	50000	100000
Random	insertionSort	0.0684	0.7773	4.9138	9.4927
	bSort	0.0839	0.8288	5.0684	9.7901
	vanlig merge	0.139	1.4432	6.6938	13.6148
Sorted	insertionSort	0.0419	0.2686	1.1615	2.4123
	bSort	0.0493	0.301	1.3549	2.9369
	vanlig merge	0.0834	0.5356	2.6276	5.2074
Reverse	insertionSort	0.388	0.3307	1.6376	3.4875
	bSort	0.0425	0.3535	1.8003	3.9424
	vanlig merge	0.0722	0.5903	3.1159	6.3388



Figur 1: Graf för merge-algoritmerna på olika sorterade data

Tabell 2: Exekveringstid, mätt i ms, för merge algoritmer med konstant inputstorlek och varierande antal sublistor $\frac{n}{k}$ med längd k. Inputlistan innehåller positiva, reella heltal som är slumpgenerade, sorterade efter storlek största till minsta.

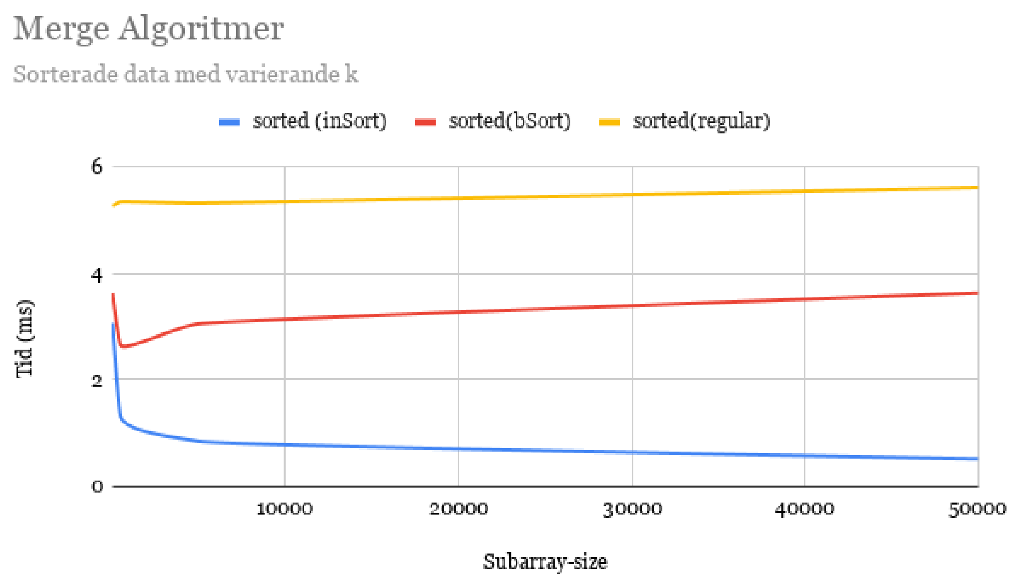
	k	5	50	500	5000	50000
	n	100000	100000	100000	100000	100000
Random	insertionSort	9.5199	6.9782	9.141	48.0927	471.5446
Random	bSort	9.8242	9.3529	13.1125	48.3234	402.3264
Random	Regular merge	13.9387	13.5704	14.2349	14.2354	14.0915
Reversed	insertionSort	4.2874	3.6213	11.0764	92.0219	948.1191
Reversed	bSort	4.7208	4.0394	11.082	81.3531	796.9815
Reversed	Regular merge	6.7966	6.9363	6.8617	6.6888	6.8126



Figur 2: Graf för merge-algoritmerna på olika storlek på subarrayer

Tabell 3: Exekveringstid, mätt i ms, för merge algoritmer med konstant inputstorlek och varierande antal sublistor $\frac{n}{k}$ med längd k. Inputlistan innehåller positiva, reella heltal som är sorterade efter storlek från minsta till största.

	k	5	50	500	5000	50000
	n	100000	100000	100000	100000	100000
Sorted	insertionSort	3.0696	1.3245	0.8500	0.5181	0.3425
Sorted	bSort	3.6304	2.6595	3.0557	3.6356	4.2163
Sorted	Regular merge	5.2641	5.3480	5.3237	5.6122	5.3799



Figur 3: Graf för merge-algoritmerna på olika storlek på subarrayer där input listan är sorterad

Diskussion

Tidsåtgång

Vi kan se i tabell 1 att båda variationer på mergeSort är snabbare än standard mergeSort när subarray storleken är 5, oavsett input storlek. Vi kan även se att vår implementation av mergeInsertionSort är snabbare än vår implementation av mergeBSort för alla input storlekar när subarray storleken är 5. Vi kan även se i tabellen att när vi skickar in en redan sorterad array in i dessa variationer så blir mergeInsertionSort snabbast och standard mergeSort långsammast igen. Någonting intressant att nämna är att variationen med insertionSort är snabbare när input är omvänt sorterade trots att det är worst case för insertionSort algoritmen. Detta för att merge delen av mergeInsertionSort blir tillräckligt mycket snabbare för att göra upp för tidsförlusten under sortering av subarrayerna.

I tabell 2 varierar vi subarraystorleken istället för inputstorleken för att se hur variationerna av mergeSort beter sig. Vi ser i tabell 2 att standard mergeSort inte använder k vilket betyder att skillnaden mellan mätdatan är enbart på grund av hur inputarrayen slumpades. Om vi jämför tiden för slumpad inputarray mellan mergeInsertionSort och mergeBSort så ser vi att mergeInsertionSort fortfarande är snabbare än mergeBSort vid lägre värden på k men när k når 50000 så blir mergeBSort snabbare. Genom detta kan vi dra slutsatsen att eftersom insertionSort är långsammare på större inputs så är även mergeInsertionSort långsammare vid större k-värden, eftersom insertionSort används för att sortera subarrayer som är k-långa. Om vi tittar på när omvänd sorterad input används så kan vi se att mergeInsertionSort växer väldigt snabbt vid högre k-värden då omvänd sorterad input är worst case för insertionSort och när k då växer till större och större värden handlar tidskostnaden mer och mer om hur snabbt insertionSort kan sortera subarrayerna.

När vi nu inspekterar alla dessa simulationer så kan vi dra slutsatsen att det optimala k-värdet bör ligga någonstans emellan $5 - 500$ för inputstorlek 100000 då vi kan se att vi får lägst tidskostnad för våra modifierade mergeSort algoritmer när k ligger runt 50.

Minnesanvändning

Den implementation av merge algoritmen som gjorts använder sig av två input listor som sedan sammanfogas och sorteras till en ny lista, som beskrivet i avsnitt MergeSort - Modifierad. Det leder till att i sista skedet i algoritmen så behöver den sammanfoga två listor av storlek $\frac{n}{2}$ till en ny lista av storlek n. Det ger att de två input listorna och den nya listan ger en minnesanvänding på $O(n^2)$. Detta kunde ha förbättrats genom att använda pekare och modifierat redan existerande input lista och fått en minnesanvändning på $O(n)$ då det inte skapas några nya listor.