Luleå tekniska universitet

Skellefteå

Laborationsrapport

D0041D Objektorienterad Programmering

Laboration 2

Handledare:

Gustav Sternbrant

Hans Granlund

hangra-4@student.ltu.se

Luleå Tekniska Universitet

Skellefteå

2016-02-05

Pontus Stenlund

ponste-5@student.ltu.se

Luleå Tekniska Universitet

Skellefteå

2016-02-05

Innehållsförteckning

Problemspecifikation **3**

Användarhandledning **4**

Algoritmbeskrivning **5**

Systembeskrivning **7**

Randtime 7

Insertionsort 8

Quicksort 8

Heapsort 8

Bucketsort 9

Lösningens begränsningar **10**

Problem och Reflektioner **11**

Diskussioner **12**

Testkörningar **13**

Problemspecifikation

I del 1 av laborationen så skulle vi implementera ett par metoder. En för att generera slumpmässiga tal. En för att kunna mäta tiden mellan algoritmers körningar. Och en för att skriva ner antal tal, tiden det tog att sortera och vilken sortering talen hade före körning (sorterad, inventerad eller random).

I del 2 så skulle vi implementera fyra olika algoritmer. Insertionsort, quicksort, heapsort och bucketsort. Alla algoritmer skulle ta emot en integer pekare med en dynamiskt allokerad array och en integer som talar om storleken på arrayen.

I del 3 så var vår uppgift att implementera algoritmerna med vektorer istället för arrayer. Samt så skulle testköra algoritmerna med start på 1000 tal och öka med en faktor av 10. Vi skulle även implementera try, catch på vår kod så att vi inte allokerar för mycket minne.

Användarhandledning

För att köra vår lösning så har vi gjort en basklass randtime som klasserna för algoritmerna ärver av. Så randtime innehåller en virtuel funktion min_sort() så att kompilatorn vet vilken av klassernas min_sort ()ska köras.

Vi skapar ett objekt av den algoritm vi vill köra. Varje algoritm får en egen array eller vektor. Sen så fyller vi i listan med fillrandom(), fillsort() eller fillinvert(). Dessa funktioner använder sig av en for-loop för att fylla listan med tal, fillsort() och fillinvert() fyller listan med tal från 0 till listans storlek.

Sedan anropar vi funktionen runtime() som skapar en timestamp före den anropar min_sort(), skapar en till timestamp när den lämnar min_sort(), räknar ut tidsskillnaden och slutligen anropar write() som skriver data till ett .txt dokument.

I vår main() så har vi gjort loopar som skapar objekt, anropar runtime(), gångrar storleken med 10 och sedan fortsätter.

Så man skapar objekt av quicksort, insertionsort, bucketsort eller heapsort för att köra med arrayer.

Och man skapar objekt av Vector_Quick, Vector_Insertion, Vector_Bucket eller Vector_Heap för att köra med vektorer.

Algoritmbeskrivning

Insertionsort:

Först tar ett värde från listan. Sedan tar vi nästa värde och jämför det med första värdet. Ifall det första värdet är större än det andra så byter vi plats på dessa. Sedan fortsätter vi och jämför alla element med varandra och byter plats till dess att vi får en sorterad lista.

Quicksort:

Quicksort använder sig av divide and conquer.

Vi tar en lista med tal, tar ett pivotelement d.v.s. ett värde som vi använder oss av och jämför alla andra element med. Beroende på ifall ett värde är större eller mindre än pivotelementet så stoppar vi det värdet före eller efter om pivotelementet. Pivotelementet kommer att hamna på rätt plats. Sedan delar vi upp listan i två rekursivt tills den har en lämplig storlek. Sedan sorterar vi de mindre listorna och sedan stoppar tillbaka listorna i ordning till den större listan.

Heapsort:

Vi tar en lista med tal och börjar med att bygga ett binärt träd som kallas heap. Heapen har oftast största talet på toppen av trädet så vi måste även vända på ordningen. Vi åstadkommer detta genom att göra vissa steg.

1. Vi tar bort roten (det största talet) och ersätter det med lövet längst åt höger. Roten stoppas in i en lista.
2. Vi återupprättar trädet
3. Upprepar steg 1 och 2 tills det inte finns några tal kvar i heapen.

De sorterade talen är nu lagrade i en array.

Bucketsort:

Fungerar som följer:

1. Vi sätter upp en lista full med tomma listor (buckets)
2. Vi går igenom original listan och lägger talen i sin bucket.
3. Sorterar alla listor som inte är tomma.
4. Vi besöker alla buckets i ordning och lägger tillbaka alla värden i den ursprungliga listan.

Systembeskrivning

Vi har även två klasser randtime och Vector_Rand som innehåller ett antal metoder:

- Runtime, som skapar två timestamps. En före den anropar sorteringen, och en annan när sorteringen är avslutad. Sen räknar den ut hur lång tid sorteringen tog och anropar sedan write.

- Write(sorttype h), går till slutet av dokumentet och skriver ner antal tal i listorna, hur lång tid det tog att sortera och hur listan såg ut före sorteringen (sorterad, osorterad, inverterad).

- Min_sort(int*arr, int size),(vector<int>* vec, int size), är en virtuel funktion som kollar på objektets pekare för att se vilken av klassernas min_sort som ska köras. Beroende på ifall man vill sortera en array eller en vektor så tar den emot en pekare av den sorten och storleken på listan.

- Name(), en virtuel funktion som bara returnerar en sträng med namnet på klassen.

- Print_array(), skriver ut arrayens element med hjälp av en loop.

- Fillsort(), fyller listan med tal mellan 1-storleken med hjälp av en loop

- Fillrandom(), använder sig av mt19937 för att generera randomtal upp till storleken på listan och lägger till dom i listan.

- Fillinvert(), fyller listan med tal mellan 1-storleken inverterad. Det största talet är det första elementet i listan.

Klasserna innehåller även, int size, int*arr eller vector<int>* vec som kommer att peka på en array eller vektor.

Vi har implementerat sorteringsalgoritmerna som klasser som ärver sig av randtime eller vector_randtime. Dessa klasser innehåller olika metoder utom min_sort() som alla klasser har.

Insertionsort:

Innehåller dessa metoder:

- Min_sort(int* arr, int size), kör sorteringsalgoritmen insertionsort

Quicksort:

Innehåller dessa metoder:

- Min_sort(int* arr, int size), anropar den andra min_sort med 0 som left.
- Min_sort(int*arr, int left, int right), kör quicksort och anropar sig själv rekursivt tills basecase har blivit uppnått.

Heapsort:

Innehåller dessa metoder:

- Min_sort(), anropar build_heap och heapsort.
- Build_heap(int* arr, int size), bygger upp heapen.
- Heapsort(int* arr, int size), byter plats på talen i listan så det största talet alltid hamnar längst bak.
- Swaps(int* arr, int i, int size), bygger om heapen med size-1 hela tiden.

Bucketsort:

Innehåller dessa metoder:

- Min_sort(int* arr, int size), skapar en lista med listor i sig. Antalet listor i bucketlistan är storleken/10.
- Sedan tar den tal från listan som skickas in som argument och delar det i 10. Svaret blir ett index till den bucket som talet ska läggas i. Sedan när vi upprepat det till listans slut så använder vi oss av quicksort för att sortera alla buckets.

Lösningens begränsningar

Eftersom bucketsort skapar en ny lista full med listor så tar det upp minst dubbelt så mycket minne. Detta innebär att man inte kan sortera allt för stora listor innan minnet tar slut.

I vår bucketsort med arrayer så har vi vektor med vektorer som buckets. Detta kan innebära att det tar längre tid att sortera talen.

Vi har inte kört insertionsort eller insertionsort med vektorer ända till vi får bad_alloc av orsaken att vi kom fram till att det skulle ta för långt tid för oss att vänta på.

Istället har vi på dessa testar satt en gräns så den sista storleken av tal som sorteras är 1 miljon när dessa 2 testas.

Testerna begränsas alltså av tid eller hårdvara och hur de olika sorteringsmetoderna är uppbyggda

Problem och reflektioner

Under laborationens gång har vi kanske kommit på andra lösningar på problem i koden som det först var tänkt att det skulle följa en viss röd tråd och struktur, och några av dess ändringar har det då medföljt att den plan vi kanske haft från början svängt lite sidledes.

Dock på grund av att vi varit två så har man alltid haft ett bollplank av hur man ska lösa problem man stött på, samt att det också är en större chans att man ser problem innan de uppkommer.

Även om vi tidigare gjort saker ensamma har man också kunnat ha givande diskussioner så har det i denna laboration varit så att vi ändå är två drar åt samma håll eftersom båda arbetar med samma sak.

Ett annat problem vi har haft är att när vi testade våra algoritmer så tog insertionsort allt för lång tid och vi fick ingen bad_alloc så den stängde av så vi blev tvungen att göra testet igen.

Våra testtider kan ha en viss skillnad eftersom ca 75 % av alla algoritmer hemma och resten på skolan. Detta problem uppstod p.g.a. vi hade igång körningen under natten och sen när vi skulle tillbaka till skolan så var inte körningen klar.

Diskussion

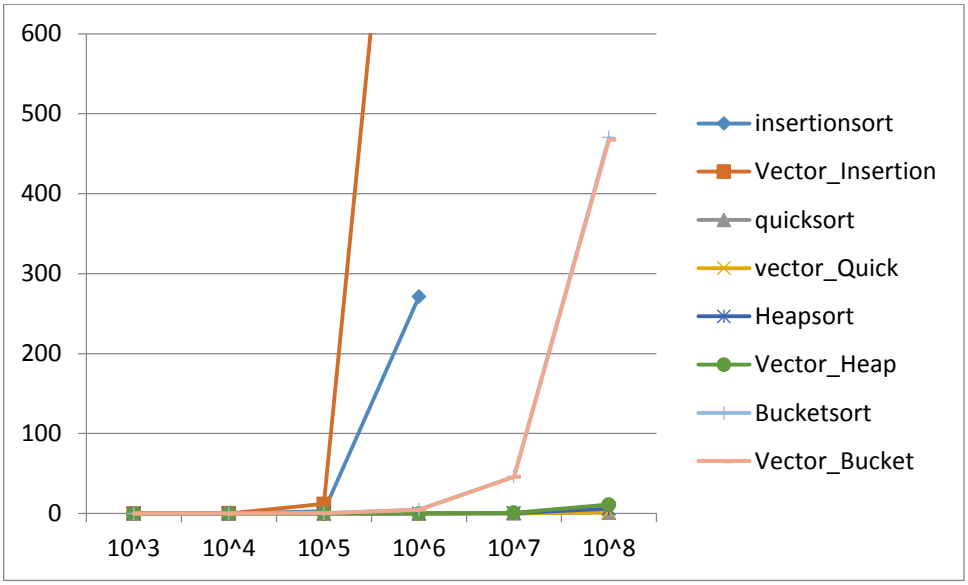
I den här labben så känner vi att vi har börjat förstå hur de olika algoritmerna fungerar. Sen har vi fått mer känsla hur mycket det skiljer i tid i körningen.

Att ha arbetat två och två har varit mycket givande. Genom att diskutera laborationen så känns det som att vi har lärt oss mycket mer.

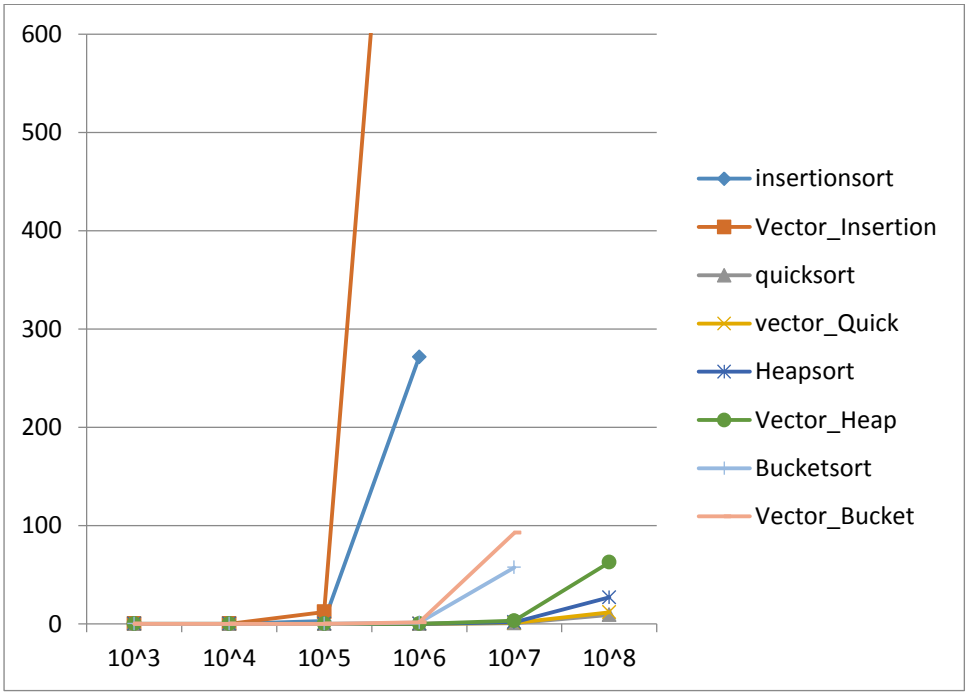
Vid felsökningen så har det även varit givande att ha haft två par ögon som har letat efter fel.

Testkörningar

Tidsdiagram för sortering från osorterad lista



Tidsdiagram för sortering från inverterad lista



Tidsdiagram för sortering från sorterad lista