Namn: Adnan Altukleh

Kurs: DV1625

Inlämningsuppgift nr: 1

Datum:2022-02-04

- **Beskrivning av algoritmerna och implementationen av dessa**

Insertion sort:

Insertion sort är en enkel algoritm som sorterar ett element i taget, därför den är ineffektiv på stora data.

Den börjar med att ta elementet i position 2 i listan, sedan jämför den med elementen till vänster om dvs elementet i position 1 är elementet i position 2 större än elementet i position 1, då byter de platserna om inte då är de redan sorterade.
Den försätter göra så ett element i taget och jämför med de elementen tills listan är sorterad, dvs den nästa iteration tar element i position 3 jämför den med elementen i position 2, om den större eller lika med då är den sorterad om den mindre då byter de plats sen jämför den igen med elementen i position 1.
Den fortsätter tills den sista element i listan är sorterad.[i]

Tiden som tar en Insertion sortalgoritm beror på hur stor input ”listan” är, jo större input desto längre tid det tar. Även beror det på om input är sorterad redan, för två input med samma längd tar sorteringsalgoritm olika tid att sortera det kan beror på att den ena är redan mer sorterad än den andra. [ii]
Det innebär att den är effektiv för mindre data och väldig ineffektiv för stora data
Insertion sort är adaptiv, vilket innebär att den minskar det total antalet steg om en delvis sorterad input, vilket gör den effektiv.
Det är en stabil sorteringsteknik, eftersom den inte ändrar den relativa ordningen av element som är lika. Den är också en jämförbar algoritm [iii]

Quick sort:

QuickSort är en Divide and Conquer-algoritm, vilket innebär att de rekursivt delar upp arrayerna i mindre och mindre sorterade subarrays tills arrayen slutligen är sorterad.
Den har körtiden O(nlogn), vilket är den snabbaste körtiden för en jämförelsebaserad sorteringsalgoritm.
Den börjar med att välja ett povit element, i mitt fall tar den sista element, därefter delar upp listan till två mindre sub-listor där sorterar elementen som större eller lika med i en lista och de som mindre i en annan lista, då har fått ett sorterat element povit, Denna partitionering kan sedan appliceras rekursivt på sub-listorna för att bilda ännu mindre partitionerade sub-listor. Basfallet för rekursionen är en sub-lista av storlek ett eller storlek noll, som redan är sorterade per definition.[iv]

I effektiva implementeringar är snabb sortering inte en stabil sortering, vilket innebär att den relativa ordningen av lika sorteringsobjekt inte bevaras.
Om arry är sorterad, har identiska element eller sorterad baklänges då är Quick sort i sin worst case dvs tiden ökar mer än om arry är inte i dessa formar.
Där emot är Quick sort väldig snabb sortering algoritm då tiden är O (n log n)[v]
Quick sort uppnår optimal prestanda om vi alltid delar upp arrayerna och sub-listor i två lika stora partitioner.

Heap sort:

Heap sort fungerar genom att visualisera elementen i arrayen som en speciell sorts komplett binärt träd som kallas en heap.

Heap sort delar upp dess input i en sorterad och en osorterad region, och den krymper iterativt den osorterade regionen genom att extrahera det största elementet från den och infoga den i den sorterade regionen. Heap sort slösar inte tid med en linjär tidsavsökning av det osorterade området; snarare bibehåller högsortering den osorterade regionen i en heap data struktur för att snabbare hitta det största elementet i varje steg[vi]

Tidskomplexitet: Tidskomplexiteten för Heapify är O (Log n). Tidskomplexiteten för bild en Heap är O(n) och den totala tidskomplexiteten för Heap Sort är O (n Log n).[vii]

Effektivitet – Tiden som krävs för att utföra Heap-sortering ökar logaritmiskt medan andra algoritmer kan växa exponentiellt långsammare när antalet objekt att sortera ökar. Denna sorteringsalgoritm är mycket effektiv.
Minnesanvändning – Minnesanvändningen är minimal eftersom förutom vad som är nödvändigt för att hålla den första listan över objekt som ska sorteras, behöver det inget extra minnesutrymme för att fungera
Enkelhet – Det är enklare att förstå än andra lika effektiva sorteringsalgoritmer eftersom det inte använder avancerade datavetenskapliga begrepp som rekursion
Heap sort är en stabil sortering algoritm

- ## Tillvägagångssätt

Under tiden jag löser uppgifter har jag utgått i sort del ifrån boken och förlisningarna, jag har läst och sedan har jag differerat hur varje algoritm ska köras. Jag skriv en lite arry i tavlan och sorterade den för varje sorteringsalgoritm, dvs jag låtsade att jag dator och skrivit vilka steg som ska göras och hur för att listan ska sorteras på ett visst sätt. Sedan började jag koda de steg och lösa uppstående problem som uppstår. För varje steg som utgörs körde jag liten test för att dubbel kolla att allting funkar innan jag går till nästa steg. Till slut gjorde jag olika tester, de viktigaste är med en tom lista och lista med ett element och lista med identiska element i.

- ## Resultat, analys och slutsatser

Resultat för körtiden hos varje algoritm med data n=1000 och n=10 000 gav:
Tabell:1

	Insertion Sort	Quick Sort	Heap Sort
T (1 000)	0,020733s	0,00103933s	$4{,}354 \times 10^{-3}$s
T (10 000)	2,1393s	0,0156933s	0,0572767 s

Resultat för körtiden hos varje algoritm med data nästan sorterad där n=1000, n=10 000 och n=50 000 gav:
Tabell:2

	Insertion Sort	Quick Sort	Heap Sort
T (1 000)	0,0098s	0,0011s	0,0024s
T (10 000)	0,029s	0,0123s	0,0318s
T (50 000)	0,725s	0,0753s	0,2016s

Beräkning värdet på C och analys för bestämning av $f(x)$ för varje algoritm:

Insertion sort:

Tidkomplexet hos insertion sort är olika i bästa fall är tiden $O(n)$ då är arry redan sorterad i avreag och worst fall är tiden $O(n^2)$ i vårt fall är datamängden osorterad alls dvs tiden för vår algoritm kommer vara $O(n^2)$

Varför n^2:

Jo för att loopen i coden tar tiden n

Medan while loopen tar tiden $\sum_{j=1}^{n} t_j$

Den totala tiden för vår algoritm är n +$\sum_{j=1}^{n} t_j$ + andra n men dominerande termen som påverkar mycket är $\sum_{j=1}^{n} t_j$ som är lika med n^2 och därför är tiden för insertion sort i worst fall är $O(n^2)$.

Quick sort:

Quick sort har sitt worst case när en av de tre fallen inträffar:

1. Datan är sorterad
2. Datan är sorterad back läng
3. Alla elementen är identiska

Och då tiden är $\sum_{j=1}^{n} t_j$ dvs $O(n^2)$ enligt datan vi har är vi inte i worst case för inget av ovan stämmer med vårt data vilket är bra.

Best case för Quick sort inträffar när povit värdet hamnar i mitten av de sub_arry i varje steg ner i datan. Förklarering om best case i Quick sort nedan:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pivot

1	2	3	4	5	6	7		9	10	11	12	13	14	15

Pivot sub_arry

1	2	3		5	6	7		9	10	11		13	14	15

1		3		5		7		9		11		13		15

Anta iteration blir $\log_2 n$ då blir tids komplexitet $O(n \log_2 n)$ n är tiden för att gå igenom data och $\log_2 n$ är tiden på hur många gånger ska utföras.

Avrage case är inte mindre än best case det tar lika mycket tid som best case. Dvs i vårt fall är datan inte worst case då är det best case och vilket ofta är det.

Då vet vi att tid komplexitet lika med $O(n \log_2 n)$ för Quick sort.

Heap sort:

För Heap sort tid komplexitet är samma oavsett hur data sorterad eller är, den kommer alltid att ta lika long tid på alla sina fall och det är $O(n \log_2 n)$. Heapify tar log n tid och att skapa max/min Heap tar n tid då hela processen tar $n \log_2 n$.

Varför tar det log n tid för bild en heapify? Jo för längde av träde är som mest log n höjd då när det ska gå igenom trädet så tar det log n tid och n tid för att bild en heapify.

Vilket gör att tids komplexitet blir $O(n \log_2 n)$

Nu när vi vet hur varje algoritm har sin funktion då kan vi beräkna konstanten C:

För att beräkna tidskomplexitet så använder vi funktionen nedan:

$T(n) = C.O(f(n))$

Vi vet T(n) för varje algoritm från tabell (1), Vår $f(n)$ här hänger ihop med vad vi har fastställt i föregående del där Quick sort $f(n) = nlogn$, Heap sort $f(n) = nlogn$ och insertion sort blir då $f(n) = n^2$ då får vi C enligt funktionen nedan:

$$C = \frac{T(n)}{f(n)}$$

Tabell 3: presenterar resultat för C för varje algoritm:

	Insertion Sort	Quick Sort	Heap Sort
C_{1000}	$2{,}938 \times 10^{-8}$	$1{,}03933\times 10^{-9}$	$5{,}552 \times 10^{-7}$
C_{10000}	$2{,}139 \times 10^{-8}$	$1{,}56933\times 10^{-10}$	$4{,}310\times 10^{-7}$
$C_{Avreage}$	$2{,}539\times 10^{-8}$	$5{,}9813\times 10^{-10}$	$4{,}931\times 10^{-7}$

Nu när vi har C kan vi beräkna förväntande värde av tiden komplexitet T(n) för varje sortering algoritm med n=50 000 och n=100 000 och resultatet representeras i Tabell 3 nedan:

	Insertion Sort	Quick Sort	Heap Sort
T (50 000)	63,47s	1,4953s	0,384s
T (100 000)	253,9s	5,9813s	0,819s

Resultaten från körning av algoritmen då n=50 000 och n=100 000:

	Insertion Sort	Quick Sort	Heap Sort
T (50 000)	77,91s	0,098s	0,441s
T (100 000)	235,46s	0,27s	0,7804s

Anmärkning resultaten från körningen är ett medelvärde av alla 3 körningar av testerna för n = 50 000 och 100 000.

Förväntade värde skiljer sig lite ifrån de värdet som fåtts när programmet körs. Det vi kan se att Heap och Quick sort är i stort sett snabba och Quick sort blir snabbare än Heap sort när det gäller stora mängd av data. Min implementation av algoritmen gav bra resultat.

Det blir större skillnad när det körs 100 000 element men det är inte stor skilnad ifrån det förväntade. Den största skillnaden mellan förväntningar jag hade och resultaten jag fått var på insertion sort och heap sort med 100 000 element. Medan när vi körde med 50 000 element hade alla tre algoritmer jätte när värde jämfört med förväntningar. Om vi hade kört medelvärde av fler körningar så skulle C värdet vara mer noggrant och då får vi bättre prediktioner

Den lämpligaste algoritm som passar datamängderna: 1000,10 000,50 000 och 100 000 är Quick sort den är bäst av de tre under alla fall och den långsammaste är insertion sort. Men om vi hade en sorterad datamängd då ska Quick sort vara sämst p.g.a. Quick sort worst case är när datamängden är sorterad. Alla sortering algoritmer har sina svagheter och styrkor, var och en har sitt bästa falla beror på sammanhang av datamängden och hur den är.

När vi få sorterat data så har Insertion sort mer chans än andra algoritmen för att vinna för då är de i sitt bästa fall medan Quick sort i sitt värsta fall och betydlig långsammare.

Vid användning av nästan sorterade data ser vi att insertionsort är snabbare än de andra och det är därför en sånt här algoritm är använd och lever tills idag för att den används i sånt område där data är nästan sorterad och den sortera det snabbast av alla andra algoritmer.

- **Källförteckning**

[i] Boken Introduction to Algorithms, sidnummer: 16–22
[ii] Boken Introduction to Algorithms, sidnummer: 25–29
[iii] Boken Introduction to Algorithms, sidnummer: 25–29
[iv] Boken Introduction to Algorithms, sidnummer: 170–173
[v] Boken Introduction to Algorithms, sidnummer: 175–178, 180–185
[vi] Boken Introduction to Algorithms, sidnummer: 151–153
[vii] Boken Introduction to Algorithms, sidnummer: 152–153