DV1625 Rapport Inlämningsuppgift A

Viktor Listi, vili22

4 September 2023

Introduktion

I danna rapport kommer tre olika sorterings algoritmer undersökas där prestandan av dessa ska testas med hjälp av flera listor av olika storlekar som innehåller tal. De algoritmerna som kommer att undersökas i rapporten är Insertionsort, Mergesort och Quicksort, dessa är vanligt använda algoritmer. Utöver tester av prestandan ska det även utredas om att genom att slå ihop två av dessa algoritmer kan man göra en snabbare algoritm. Vart skulle isåfall brytningspunkten för där den ena är snabbare än den andra vara? Detta kommer att undersökas i denna rapport.

Bakgrund

Sorterings algoritmer av olika slag är en viktig del av världen vi lever i idag och styr mycket av det vi inte ser. Om dessa bakomligande algoritmer skulle vara långsamma så skulle även våra liv vara långsammare, därför tas det konstant fram snabbare och snabbare algoritmer för att konstant höja hastigheten på det dem är till för. En sorterings algoritm är ett program som kan sortera en mängd värden i en specifik ordning. Dessa algoritmer påverkar allt från sökresultat på internet till hur effektiv dina dator eller telefon är. Sökalgoritmer har därmed en avgörande verkan på världen och vikten av att fortsätta utvecklingen av dessa och konstant ta fram nya och effektivare versioner är av högsta vikt.

Insertionsort

Insertionsort är en enkel sorteringsalgoritm som bygger på principen att bygga upp en sorterad sekvens stegvis genom att infoga varje element på rätt plats i den befintliga sorterade delen av listan. Den arbetar genom att gå från början av en lista av okänd storlek och jämföra värdet innan om det är större eller mindre, om värdet är den hanterar nu är mindre än det innan så byter den plats på dem och sen kollar alla bakomliggande värden efter samma sak och om det är större så lämnar det värdet på samma plats och går vidare.

Sorteringsalgoritmen har en tidskomplexitet på N^2 i medelfallet.

+ Enkel att förstå och implementera
+ Effektiv på små datamängder
+ In-place algoritm
+ Stabil algoritm
- Dålig prestanda på stora listor

Quicksort

Quicksort är en snabb och effektiv sorteringsalgoritm som använder "divide and conquer" strategin. Algoritmen arbetar genom att välja ett “pivot” element från listan och sedan dela upp resterande värden i två del-listor, en som innehåller värden större än pivoten och en som innehåller element mindre än pivoten. Denna process implementeras oftast rekursivt och körs sedan på del-listorna tills att hela listan är sorterad och slås sedan ihop till en sorterad lista.

Sorteringsalgoritmen har en tidskomplexitet på $N*\log_2(N)$ i medelfallet.

+ Fungerar väl med stora datamängder
+ Snabb och effektiv algoritm med bra genomsnittlig prestanda.
+ In-place algoritm
- Icke Stabil algoritm
- Ineffektiv om pivot elementet väljs dåligt.

Mergesort

Mergesort är en stabil och effektiv sorteringsalgoritm som använder "divide and conquer" strategin. Mergesort fungerar genom att dela upp den oordnade listan i mindre delar och sedan sortera varje del separat och sedan sätta ihop dessa. Algoritmen implementeras rekursivt och delar upp listan i två halvor tills att varje del bara innehåller ett element, sedan sätts listorna gradvis ihop tills hela listan är sorterad.

Sorteringsalgoritmen har en tidskomplexitet på $N*\log_2(N)$ i medelfallet.

+ Effektiv med stora datamängder
+ Snabb och effektiv algoritm med bra genomsnittlig prestanda.
+ Stabil algoritm
- Icke in-place, använder mycket minne

Metod och implementationsval

Under konstruktionen av algoritmerna som ska testas har en version som är väldigt lik olika pseudo kodade versioner av algoritmerna konstruerats. Insertionsort implementationen fungerar genom att först kolla om längden av listan är större än 1, om den är det så använder den sig av en nestad loop för att sedan sortera listan.

```
def insertionsort(lst: list) -> None:
    if len(lst) > 1:
        for i in range(1, len(lst)):
            x = i-1
            current_value = lst[i]

            while(x >= 0 and current_value < lst[x]):
                lst[x+1] = lst[x]
                x = x-1

            lst[x+1] = current_value
```

Mergesort implementationen använder sig av samma metod för att kolla längden på listan där den sedan delar upp listan och sorterar dessa delar rekursivt. Sedan kombineras de sorterade delarna genom att jämföra elementen i respektive del och placera dem i en ny lista. Sedan så uppdateras den ursprungliga listan så den är samma som den sorterade listan.

```
def mergesort(lst: list) -> None:
    if len(lst) > 1:
        middle = len(lst) // 2
        left_half = lst[:middle]
        right_half = lst[middle:]

        mergesort(left_half)
        mergesort(right_half)

        merged_list = []
        left_index, right_index = 0

        while left_index < len(left_half) and right_index < len(right_half):
            if left_half[left_index] >= right_half[right_index]:
                merged_list.append(right_half[right_index])
                right_index += 1
            else:
                merged_list.append(left_half[left_index])
                left_index += 1

        merged_list.extend(left_half[left_index:])
        merged_list.extend(right_half[right_index:])
        lst[:] = merged_list
```

Quicksort implementationen använder sig också av samma metod i början för att kolla längden, sedan väljs ett pivot element ut i mitten av listan, detta värdet väljs då detta är en vanlig strategi för att få två balanserade delar av listan som sällan ger worst-case. Den delar upp listan i tre delar, mindre än pivot, större än pivot och lika med pivot. Dessa delar sorteras sedan rekursivt och slår sedan ihop delarna till en sorterad lista.

```
def quicksort(lst: list) -> None:
    if len(lst) > 1:
        pivot = lst[len(lst) // 2]

        larger_elements, equal_elements, smaller_elements = []

        for element in lst:
            if element < pivot:
                smaller_elements.append(element)
            elif element == pivot:
                equal_elements.append(element)
            else:
                larger_elements.append(element)

        quicksort(smaller_elements)
        quicksort(larger_elements)

        lst[:] = smaller_elements + equal_elements + larger_elements
```

Resultat och Analys

Sorteringsalgoritmer

Algoritm	Körtid (ms)					
	1	2	3	4	5	Medelvärde
Insertionsort	0.1276	0.1520	0.1322	0.1341	0.1417	0.13752
Quicksort	0.1340	0.1812	0.1415	0.1649	0.1366	0.15164
Mergesort	0.1353	0.1526	0.1490	0.1472	0.1725	0.15132

Tabell 1 körtid för algoritmer där n = 10

Algoritm	Körtid (ms)					
	1	2	3	4	5	Medelvärde
Insertionsort	0.5410	0.4693	0.4659	0.3767	0.4903	0.46864
Quicksort	0.3329	0.2361	0.3536	0.2822	0.2284	0.28664
Mergesort	0.5285	0.4364	0.3305	0.3067	0.3178	0.38398

Tabell 2 körtid för algoritmer där n =100

Algoritm	Körtid (ms)					
	1	2	3	4	5	Medelvärde
Insertionsort	31.1072	20.0865	28.1248	28.3679	35.4084	35.4084
Quicksort	1.9298	2.1701	1.4525	1.4601	1.4499	1.69248
Mergesort	2.1910	2.2138	2.1873	2.1673	2.2921	2.2103

Tabell 3 körtid för algoritmer där n =1000

Algoritm	Körtid (ms)					
	1	2	3	4	5	Medelvärde
Insertionsort	2388.2962	2338.3079	2462.6624	2651.3088	2946.861	2557.4873
Quicksort	25.3441	25.7425	17.0989	19.7859	24.5495	22.5042
Mergesort	25.9263	25.8382	26.5725	24.8113	27.7293	26.1755

Tabell 4 körtid för algoritmer där n =10000

Algoritm	Körtid (ms)					
	1	2	3	4	5	Medelvärde
Insertionsort	1079534.8324	782764.7321	971917.338	917897.571	826179.41	915658.7774
Quicksort	334.7611	240.2468	296.0796	228.0164	266.0541	273.0316
Mergesort	381.3817	374.0679	314.0504	472.9154	350.738	378.6307

Tabell 5 körtid för algoritmer där n =100000

Identifiering av C-konstanter

För beräkning av C-konstanterna används ekvationen T(n) = C * f(n). Där T(n) är körtiden, f(n) är algoritmens funktion och C är c-konstanten. Omskrivet ska kan vi få ut C-konstanten med hjälp av funktionen C = T(n) / f(n). C konstanten kommer vara baserad på ms.

Insertionsort

C = T(n) / n^2

C10 = 0.13752 / 10^2 = 0.0013752 ≈ 0.001375
C100 = 0.46864 / 100^2 = 0.000046864 ≈ 0.000047
C1000 = 35.4084 / 1000^2 = 0.0000354084 ≈ 0.000035
C10000 = 2557.4873 / 10000^2 = 0.000025574873 ≈ 0.000026
C100000 = 915658.7774 / 100000^2 = 0.00009156587774 ≈ 0.000092

Cmedel = (C10 + C100 + C1000 + C10000 + C100000) / 5 = 0.000315

Quicksort

C = T(n) / (n* $\log_2(n)$)

C10 = 0.15164 / (10*$\log_2(10)$) = 0.00456481854 ≈ 0.004565
C100 = 0.28664 / (100* $\log_2(100)$) = 0.0004314361898 ≈ 0.000431

C1000 = 1.69248 / (1000* $\log_2(1000)$) = 0.0001698290824 ≈ 0.000170
C10000 = 22.5042 / (10000* $\log_2(10000)$) = 0.0001693609807 ≈ 0.000169
C100000 = 273.0316 / (10000* $\log_2(10000)$) = 0.0001643814027 ≈ 0.000164

Cmedel = (C10 + C100 + C1000 + C10000 + C100000) / 5 = 0.0010998

Mergesort
C = T(n) / (n* $\log_2(n)$)

C10 = 0.15132 / (10* $\log_2(10)$) = 0.004555185894 ≈ 0.004555
C100 = 0.38398 / (100* $\log_2(100)$) = 0.0005779474887 ≈ 0.000578
C1000 = 2.2103 / (1000* $\log_2(1000)$) = 0.0002217888665 ≈ 0.000222
C10000 = 26.1755 / (10000* $\log_2(10000)$) = 0.0001969902663 ≈ 0.000197
C100000 = 378.6307 / (10000* $\log_2(10000)$) = 0.000227958396 ≈ 0.000228

Cmedel = (C10 + C100 + C1000 + C10000 + C100000) / 5 = 0.001156

Algoritm	*c*-konstanter					
	C_{10}	C_{100}	C_{1000}	C_{10000}	C_{100000}	C_{medel}
Insertionsort	0.001375	0.000047	0.000035	0.000026	0.000092	0.000315
Quicksort	0.004565	0.000431	0.000170	0.000169	0.000164	0.001099
Mergesort	0.004555	0.000578	0.000222	0.000197	0.000228	0.001156

Tabell 6 Tabell för C konstanter av algoritmerna vid olika mängder tal

Identifiering av brytpunkter

Genom att analysera den informationen vi nu har om de olika algoritmerna kan vi ställa upp ekvationer för att konstatera brytningspunkten där quicksort liksom mergesort blir mer effektiva än insertionsort. Vi använder oss av ekvationen C * f(n) = T(n). Där T(n) är körtiden, f(n) är algoritmens funktion och C är c-konstanten. Med hjälp av denna kan vi sedan ställa upp ekvationer för att beräkna brytningspunkterna.

Mergesort:

$C_{insertion} * n^2 = C_{merge} * n * \log_2(n)$

Med denna informationen kan vi med hjälp av en grafritare se vilket n som är brytpunkten.

Detta ger brytpunkten när n = 13.955552520 ≈ 14

Quicksort:

$C_{insertion} * n^2 = C_{quick} * n * \log_2(n)$

Med denna informationen kan vi med hjälp av en grafritare se vilket n som är brytpunkten.

Detta ger brytpunkten när n = 12.8533001451≈ 13

Genom att analysera informationen vi har fått nu så kan vi konstatera att mergesort är mer effektiv än insertionsort när n >= 14 och quicksort är mer effektiv än insertionsort när n >= 13. Hybrid algoritmer implementeras där insertionsort körs när "n" är mindre än brytpunkten och quicksort respektive mergesort körs när "n" är större än brytpunkten.

Hybridsorteringsalgoritmer

Filstorlek	Algoritm	Körtid (ms)					
		1	2	3	4	5	Medelvärde
10	$Hybrid_Q$	0.1401	0.1274	0.1308	0.1320	0.1383	0.13372
	$Hybrid_M$	0.1267	0.1273	0.1359	0.1289	0.1285	0.12946
100	$Hybrid_Q$	0.2743	0.2155	0.2052	0.2043	0.2281	0.22548
	$Hybrid_M$	0.2328	0.2332	0.2315	0.2351	0.2301	0.23254
1000	$Hybrid_Q$	1.1740	1.2337	1.1849	1.1981	1.1826	1.19466
	$Hybrid_M$	1.5265	1.4698	1.5559	1.5368	1.4766	1.51312
10000	$Hybrid_Q$	15.0268	15.8093	16.2510	23.0187	15.3574	17.09264
	$Hybrid_M$	19.0863	19.4287	19.31380	19.6474	19.4680	19.38884
100000	$Hybrid_Q$	209.5491	214.4832	222.5871	223.5125	230.4958	220.12554
	$Hybrid_M$	256.9866	271.5171	264.8028	274.6864	274.1359	268.42576

Tabell 7 Hybridsorteringsalogirtmer och körtider (ms)

Slutsatser

Genom att analysera körtiderna från våra originella algoritmer mot de hybridalgoritmerna som vi skapat kan vi se att dessa hybridalgoritmer är märkvärdigt snabbare än de original algoritmerna detta bekräftar att kombinationen av olika tillvägagångssätt kan leda till betydande förbättringar av prestandan. Genom detta har vi bevisat att man kan sammanföra olika metoder för att uppnå ett bättre resultat.