Sortera en array

Adalet Adiljan

Hösten 2023

Introduktion

Till denna uppgift utforskas det olika sorteringsalgoritmer och deras för- och nackdelar. Beroende på när och i vilket sammanhang en viss sorteringsalgoritm används kan den bidra till hög eller låg effektivitet, varför det kan vara bra att förstå sig på deras algoritm och när den bör användas. Här arbetar vi endast med arrayer av en given storlek utan att lägga till nya element i den redan givna datamängden. I metodavsnittet framsätts det lösningar till deluppgifterna, medans i resultatavsnittet visas körtiden som funktion av arrayens storlek.

Metod

Selection Sort

Till varje deluppgift behövde vi implementera en särskild sorteringsalgoritm. Första deluppgiften handlade om att implementera en selection sort algoritm. För att komplettera koden skapade vi en if-villkor och byter plats med kandidat-variabeln när vi har hittat ett element som är mindre än vår nuvarande element:

```
:
for (int i = 0; i < arr.length - 1; i++) {
int candidate = i;
for(int j = i; j < arr.length; j++){
  if(arr[j] < arr[candidate]) {
  candidate = j;
  }
}
```

Nackdelarna med selection sort är dess ineffektivitet för stora arraystorlekar, dock fungerar den bra för små arrayer och har en enkel implementation. Selection sort skannar resten av arrayen och väljer det minsta elementet. Processen upprepas för att hitta det näst minsta elementet och

placerar det i den andra positionen i den sorterade delen. Tidskomplexiteten ligger på

$$O(n^2)$$

i både det sämsta och bästa fallet, varför den är ineffektiv för stora dataset.

Insertion Sort

Till vår andra deluppgift implementerade vi en liknande algoritm kallad insertion sort. Istället för att den letar efter det minsta elementet i början av arrayen, jobbar insertion sort successivt på att bygga upp den redan sorterade delen av arrayen från vänster till höger, då vi har nått slutet av arrayens längd. Vi börjar med ett element i den osorterade delen och jämför det med elementet i den sorterade delen för att hitta rätt plats att infoga det. I sämsta fallet ligger tidskomplexiteten på

$$O(n^2)$$

, men kan upplevas som mer effektiv än selection sort för små dataset och nästan sorterade data.

Merge sort

Merge sort är i generellt sett mer effektiv och användbar för att sortera stora dataset. Implementationen kan översiktligt förklaras av att vi delar upp arrayen i mindre delar. I vårt fall delade vi upp den i två delar, som vi sorterar dessa separat och slutlingsvis sammanfogar (mergar) delarna till en enda sorterad array. När det kommer till tidskomplexiteten ligger den på O(n log n) i både det sämsta och bästa fallet, vilket gör den till bättre alternativ för stora dataset. Likt de båda algoritmerna är merge sort även stabil. I bland annat introduktionen till uppgiften nämndes det också om att merge sort är en stabil typ av algoritm, vilket innebär att den behåller den ursprungliga ordningen på elementet med samma ”nyckelvärde”, (dvs. samma värde som används för att jämföra elementen) när en array sorteras. Nackdelarna med merge sort är att den kräver extra minnesuttrymme för att skapa temporära delarrayer, särskilt i jämförelse med de två ovannämnda sorteringstyperna (utan att det krävs ytterliggare minnesuttrymme). En ytterliggare nackdel är att vår implementerade version är mer komplex då vi skapade den rekursivt.

Vår slutliga uppgift var att förbättra vår mergesort algoritm med syfte att minska den extra minnesuttrymme som framkallas. Vi började med att först kopiera alla element av vår ursprungliga array till en temporär array, så kallade ”hjälparray” / auxiliary array. När vi nu har två identiska kopior, jobbar vi på att använda sort-meotden för att sortera element i hjälparrayen. Vår sort-metod tar emot två argument, den första är den array som sorteras medan den andra är hjälparrayen där resultaten lagras.

När den rekursiva sort-metoden anropas, kommer den att sortera hjälparrayen och lägga resultatet i den ursprungliga arrayen. Det knepiga som gjorts var att byta plats på ordningen på hjälparrayen och den ursprungliga arrayen vid den rekursiva anropet på sort-metoden. Däremot vid merge-metoden behöver vi inte kopiera något då hjälp-arrayen är redan sorterad, och omedelbart kan vi sammanfoga resultatet i vår ursprungliga array.

Prestandamätning

Metoden för mätningen implementerades i main-funktionen, där vi skapade en array av olika bestämda array storlekar. De totalla antalet element som finns i arrayen används som vår iterationsvillkor till vår for-each loop, medan array storlek används som argument till vår randomiserade metod för att framkalla slumpmässigt valda integer-värden för vår givna array storlek. Resultatet av tiden sparas i vår totalTime variabel, som omvandlats till millisekunder.

```
:
for ( int n : size)
{
  double totalTime = 0.0;
  int[] arr = genRandom();
  long startTime = System.nanoTime();
  selectSort(arr);
  long endTime = System.nanoTime();
  totalTime = (endTime - startTime) / 1e6;

  system.out.printf("Array size: %d, Execution time in ms: %.2f%n", n, totalTime)
}
```

Resultat

Till vår uppgift har vi implementerat och jämfört tre olika sorteringsalgoritmer. Målet var bland annat att förstå deras funktionsprinciper, analysera prestandamätningarna och även utforska deras fördelar och nackdelar. På tabell 1 kan vi sannerligen se att selection sort påvisar bra effektivitet då array storleken är liten, men ökar med ganska stora tidsvärden för ju större array storlek vi har. Trots att selection respektive insertion sort har samma tidskomplexitet, kan vi även bekräfta att insertion sort är mer effektiv för stora arrayer i jämsides med selection sort. Det nämndes även att selection sort kan vara mer effektiv än insertion sort för små arrayer, vilket tabell 1 bekräftar. Merge sort överklassar dessa som påvisar sig vara ännu snabbare för stora arrayer. Pga. merge sorts tidskomplexitet så märks inte prestandan vid små storlekar. När listans storlek ökar till det slutliga storleken

121500 blir skillnaderna i prestandan allt tydlig. Detta för att dessa två har en kvadratisk tidskomplexitet medan mergesort behåller sin

$$O(nlogn)$$

prestanda.

Tabell 2 visar prestandamätningarna för merge sort före och efter vår eventuella förbättring. Då merge sort kräver extra minnesuttrymme för att skapa temporära delarrayer (i jämförelse med selection och insertion sort som sorterar på plats) har vi tillämpat en lösning till att undvika detta. VI kan på tabell 2 se att exekveringstiden har förbättrats, där den är konsekvent något snabbare än insertion sort även när array storleken ökar.

Tabell 1

storlek	**selection**	**insertion**	**mergesort**
100	0.14	0.22	0.01
300	0.98	1.1	0.02
900	1.7	2.7	0.05
2700	6.0	7.3	0.19
8100	19.6	9.8	0.58
24300	238.7	74.5	1.8
40500	643.5	209.1	3.1
56700	1258.5	399.7	4.6
72900	2080.9	666.2	6.1
89100	3116.7	993.5	7.8
105300	4380.4	1399.4	9.0
121500	5811.7	1867.1	10.3

Table 1: Visar prestandamätningen för de olika sorteringsalgoritmerna utifrån olika array storlek, med tiden mätt i millisekunder.

storlek	**före**	**efter**
100	0.01	0.01
300	0.02	0.01
900	0.05	0.05
2700	0.19	0.17
8100	0.58	0.54
24300	1.8	1.8
40500	3.1	3.0
56700	4.6	4.5
72900	6.1	5.8
89100	7.8	7.5
105300	9.0	8.7
121500	10.3	10.1
250000	22.9	22.2
300000	28.0	27.0
350000	32.7	31.8
400000	37.5	36.6

Table 2: tabell visar prestandamätningen för merge sort före och efter förbättringen för varje stigande array storlek, med tiden mätt i millisekunder.