Quick sort

Adalet Adiljan

Hösten 2023

Introduktion

Förmågan att effektivt sortera data är av avgörande betydelse, speciellt i den ständiga digitala värld vi lever i idag. Från de tidigare uppgifterna har vi lärt oss att tillämpa olika sorteringsalgoritmer. Quick Sort är en kraftfull sorteringsalgoritm, som enligt dess namn antyder att vara snabb och effektiv. I vår rapport tänker vi undersöka närmare på hur sorteringen implementeras samt presteras hos en array respektive en länkad lista.

Quick Sort

Quick sort är en sorteringsalgoritm som delar upp en datamängd i sekvenser och sorterar dem separat. Den liknar Merge Sort algoritmen, till skillnad från att den gör mer arbete under uppdelningen men inget när de två delarna sammanfogas i slutet när sorteringen är klar. De två sekvenser motsvarar alla mindre element respektive större element.

Metod

Array

I vår QuickArray klass har vi definierat metoderna sort enligt följande vis:

inserting a code

```
public static void sort(int[]arr, int min, int max) {
      if(min < max) {
              int pivot = partition(arr, min, max);

              sort(arr, min, pivot-1);
              sort(arr, pivot+1, max);
              }
      }
```

Metoden tar emot en array(arr), minsta index (int min) och en största index (int max) som argument. Denna algoritm har vanligtvis fördelen av att använda mindre minnesallokering jämfört med MergeSort, vilket syns när vi ska sortera den osorterade arrayen utan att behöva allokera en ny array för sorterade element. I metoden väljer vi ett pivot-element, och anropar metoden partition i samband med detta. Det som sker är att elementen som är mindre än pivoten placeras på vänster sida och elementet större än pivot i höger sida, vilket uppfylls med hjälp av index-variablerna i respektive j. När i- och j- hittar ett element som är större än pivoten respektive mindre än eller lika med pivoten, byts elementets positioner. Processen fortsätter tills i och j möts eller passerar varandra. Efter att fördelningen är klar, är pivot-elementen på rätt plats i arrayen, och delarrayerna sorteras rekursivt. De sorteras var för sig, där vi anropar sort-metoden på var och en av dem. Tillslut sammanfogas dem till en array som nu är sorterad.

```
public static int partition(int[]arr, int min, int max){
int pivot = arr[min];
int i = min;
int j = max;

while(i < j) {
    while(i < j && arr[j] > pivot) {
        j--;
        }
        while(i < j && arr[i] <= pivot)
        {
        i++;
        }
        if(i < j) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
        }
    }
  }
  (:)
```

I vår partition-metod börjar vi att först välja ut ett pivot-element från vår befintliga array. Detta element används som jämförelsepunkt för att dela upp arrayen i mindre element. Varje element som är mindre än pivot-elementet förflyttas till vänster om pivot och större element placeras till höger om pivot.

Vi har även definierat index-variablerna 'i' och j'. Index 'i' kommer att röra sig från vänster till höger, medan j kommer att röra sig från höger

till vänster. Den första inre-while-loopen inuti yttre while-loopen används för att hitta ett element som är mindre än eller lika med pivot-elementet från höger mot vänster. Den andra inre while-loopen försöker att hitta element som är större än element, och går från vänster mot höger där i-variabeln ökas tills villkoret blir uppfylld. Sedan kontrollerar den villkorliga satsen om i är mindre än j, varav vi byter plats på dem (för att uppfylla partitioneringskraven). Då villkoret i while-loopen inte längre är uppfylld (i mindre än j) uppfattas arrayen färdig från partitionering.

Länkad lista

Vi använder oss av en enkel nodstruktur som innehåller en referens till nästa nod, där varje nod innehåller ett heltalsvärde. I syfte för att sortera de mindre respektive större noderna, skapades två instanser av klassen som ska motsvara två tomma listor som har döpts till smaller respektive larger. Sorteringen av de mindre respektive större noderna sker i de rekursiva anropen. När sorteringen är slutfört sammanfogas de två delarna emellan det utvalda pivotelementet. Till vår lista har vi skapat add-metod som lägger till ett nytt element med den nya instanserade klassen och associerade heltalsvärdet. Sedan implementerades algoritmen på följande vis:

```
(:)
int pivot = head.data;
(:) // instanserar två tomma listor.
Node current = head.next;
while(current != null) {
   if(current.data < pivot) {
        smaller.add(current.data);
    } else {
        larger.add(current.data);
    }
    current = current.next;
}
```

Innan vår while-loop kontrollerar vi om listan är tom eller innehåller minst ett element. Om så är fallet returnerar listan eftersom den redan är sorterad. Sedan väljer vi första elementet i listan som vår pivot element. Valet av pivot har av betydelse för sorteringens effektivitet, men då heltalsvärdena för varje nod är slumpmässigt genererade i vår benchmark, bör detta minska risken för tidskomplexiteten att luta sig till värsta fall scenario. Efter att de tomma listorna skapas, sorteras elementen till höger samt vänster om pivoten, där alla element mindre än pivot hamnar i smaller-listan och större element i larger-listan. När listan är uppdelad dvs. att vi har traverserat igenom och sorterat alla noder, uppdaterar vi huvudet till den sorterade,

mindre, listan. Råkar listan vara tom, skapas en ny nod med pivot-värdet. Om dessutom den större listan finns, kommer den mindre respektive större listan att sammafogas där head-pekaren pekar till den första elementet i den mindre listan. Slutligen efter att partitionen är klar sammanfogas mindre- och större listan med varandra, där pivot-elementet placeras emellan den minsta och största listan (dvs. i slutet av mindre listan och början av större listan) för att den ska hamna i rätt plats i förhållande till dem. Detta gör vi genom att jämföra om vår pivot-element är mindre än alla element i minsta listan respektive större än alla element i större listan. Om så är fallet, placeras den som första nod eller sista nod.

Resultat

Tabell

storlek	**array**	**länkad lista**
100	0.01	0.02
300	0.01	0.12
600	0.05	0.10
1200	0.15	0.65
2400	0.36	0.16
4800	0.81	0.26
9600	1.6	0.51
19200	3.4	1.1
38400	7.4	2.1

Table 1: Den första kolumnen motsvarar olika storlekar i stigande ordning, som vi har varierat hos arrayen och den länkade listan. Andra och tredje kolumnen visar prestandamätningen för de olika storlekarna givet i millisekunder.

Tidskomplexiteten för arrayen i bästa och genomsnittliga fall ligger på:

$$O(n * log(n))$$

och i värsta fall på:

$$O(n^2)$$

, där n motsvarar längden på arrayen (antalet element). Om vi jämför vår tabell med dessa två olika fall, ser vi att körtiden är snabbare än det värsta fallet. Körtiden ökar gradvis när antalet element i arrayen ökar exponentiellt vilket är typiskt för det bästa / genomsnittliga fallet.

Tidskomplexiteten hos Quick Sort på en länkad lista ligger också på O(n*log(n)) i bästa fall. Chansen att vi får det bästa fallet ökar när pivotelementen väljs så att den delas jämnt varje rekursiv anrop. I värsta fall

kan vi få en ökad körtid på

$$O(n^2)$$

som sker då pivotvalet är gynnsam för varje anrop, vilket resulterar i ojämna delar vid varje rekursiva anrop.

I allmänhet är Quick Sort snabbare att sortera en array jämfört med en länkad lista, vilket inte är självfallet i vår tabell. I en array lagras elementen i följd i minnet, medan elementen är utspridda i minnet hos en länkad lista eftersom att de pekar oorganiserat. Sorteringsalgoritmen hoppas på att använda minneslokalitet eftersom den använder sig av indexering (lokalitetsprincipen) och cache-minnet för att komma åt ett element, vilket leder till färre cache-missar och således snabbare åtkomst. Indexeringen hos en array har har en konstant tid på

$$O(1)$$

medan vi hos en länkad lista behöver vi traversera igenom varje nod, vilket mtosvarar en linjär tid

$$O(n)$$

i värsta fall. Listelementen är nämligen utspridda i minnet, vilket orsakar fler cache-missar och längre åtkomsttid. En ytterliggare förklaring till testresultaten kan ha att göra med fördelningen av datan. Quick Sort presterar ju bäst när datan är jämnt fördelad, men då vi i detta fall såg till att testdatan var slumpmässigt ordnade, kan detta bestrida den förväntade fallet. Vi har även nämnt i metod-avsnittet att pivot-valet kan påverka hur effektiv Quick Sort presterar, speciellt i vår länkade lista där vi valde första elementet som vår pivot. Trots att den bidrar till att vara en enkel och (i flesta fall) enkel strategi, är det inte alltid bästa valet för olika typer av indata. I vårt fall gjorde vi en slumpmässig generering av heltal mellan 0-1000, vilket minskar risken att den första noden är den ständigt dåliga utvalda elementet i varje partition.