IN2010 Oblig 3 Erlend Abrahamsen

Tuesday, October 29, 2019 4:21 PM

- **Java koden for utplukkssortering, kvikksortering og flettesortering:**

```
static int[] selectionSort(int[] arr) {
  /*
  Selection-sort / utplukkingssortering.
  Velger første element,
  looper gjennom etterfølgende elementer helt til et mindre er funnet.
  Bytter om plassen på de. Gjenta prosess.
  */

  int n = arr.length;

  for (int i = 0; i < n-1; i++) {
   for (int j = i+1; j < n; j++) {
    if (arr[j] < arr[i]) {
     int temp = arr[i];
     arr[i] = arr[j];
     arr[j] = temp;
    }
   }
  }

  return arr;
 }
```

```
static void mergeSort(int[] arr, int l, int r) {
  /*
  merge/flette sorting. Bruker sub-metoden mergeTogether.
  */

  if (l >= r) { //Ferdig
   return;
  }

  int m = (l+r)/2;

  mergeSort(arr, l, m);
  mergeSort(arr, m+1, r);
  mergeTogether(arr, l, m, r);
 }

 static void mergeTogether(int[] arr, int l, int m, int r) {
  /*
  Sub-metode i merge. Fletter sammen sortert S1, S2 inn i arr.
  */


  int n1 = m - l + 1; int n2 = r - m;
  int[] S1 = new int[n1];
  int[] S2 = new int[n2];

  //Kopiere til to halvdeler S1, S2.
  for (int i = 0; i < n1; i++) { S1[i] = arr[l+i]; }
  for (int i = 0; i < n2; i++) { S2[i] = arr[m+i+1]; }

  int i = 0; int j = 0; int k = l;
  while (i < n1 && j < n2) {
   if (S1[i] <= S2[j]) {
    arr[k] = S1[i];
    i++;
   }

   else {
    arr[k] = S2[j];
    j++;
   }
   k++;
  }

  //Resten av enten S1 eller S2 dersom n1 != n2.
  while (i < n1) {
   arr[k] = S1[i];
   i++;
   k++;
  }
```

```
static void quickSort(int[] arr, int l, int r) {
  /*
  Kvikksortering.
  Velger siste element i arr som pivot.
  Leter fra venstre mot høyre til et element er større enn pivoten.
  Leter fra høyre mot venstre til et element er mindre enn pivoten.
  Bytter plass på disse elementene.
  Oppdaterer pivoten til siste byttet element fra venstre mot høyre.
  */

  if (l >= r) {
   return;
  }

  int pivot = arr[r];       //Velger siste element i partisjon som pivot

  int i = l; int j = r;
  while (i < j) {
   while (i < j && arr[i] <= pivot) {
    i++;
   }
   while(j > i && arr[j] >= pivot ) {
    j--;
   }

   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
  }

  //Siste byttet element fra venstre som ny pivot
  int temp = arr[i];
  arr[i] = pivot;
  arr[r] = temp;

  //Rekursjon
  quickSort(arr, l, i-1);
  quickSort(arr, i+1, r);
 }
```

```
  while (j < n2) {
    arr[k] = S2[j];
    j++;
    k++;
  }
 }
```

- **Mønster fra sorteringsalgoritmene med forklaring:**

selctionSort() (Utplukssortering)

In arrays:
[**4**, **0**, 7, 5, 5, 8, 5, 3, 6, 6]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[**9**, **8**, 7, 6, 5, 4, 3, 2, 1, 0]

Selection-sort steps output:

[0, **4**, 7, 5, 5, 8, 5, **3**, 6, 6]
[0, 3, **7**, **5**, 5, 8, 5, 4, 6, 6]
[0, 3, **5**, 7, 5, 8, 5, **4**, 6, 6]
[0, 3, 4, **7**, **5**, 8, 5, 5, 6, 6]
[0, 3, 4, **5**, 7, 8, **5**, 5, 6, 6]
…
Finished...

[**8**, 9, **7**, 6, 5, 4, 3, 2, 1, 0]
[**7**, 9, 8, **6**, 5, 4, 3, 2, 1, 0]
[**6**, 9, 8, 7, **5**, 4, 3, 2, 1, 0]
[**5**, 9, 8, 7, 6, **4**, 3, 2, 1, 0]
[**4**, 9, 8, 7, 6, 5, **3**, 2, 1, 0]
…
Finished...

Denne algoritmen velger første element,
looper gjennom etterfølgende elementer helt til et mindre er funnet og
bytter om plassen på de.

For reversert input legger vi merke til et mønster hvor elementer langs første kolonne
blir byttet mot elementer langs diagonalen. (Om vi tenker på utskriften som en matrise)
Etter element 0 har fått indeks 0. Blir andre kolonne byttet mot en mindre diagonal, osv.
Dette er jo fordi listen er reversert så neste j indeks øker med 1 for hvert bytte.

For 0-9 inputen går vi gjennom alle 55 operasjoner uten å finne et par å bytte om.
Vi printer ingen endringer på arrayen.

quickSort() (kvikksortering)

New random in:
[7, 0, 1, 3, 2, 2, 0, 6, 2, 3]

Quick-sort steps output:

[**7**, 0, 1, 3, 2, 2, 0, 6, **2**, 3]
[**2**, 0, 1, 3, 2, 2, **0**, 3, 7, 6]
[0, 0, 1, 3, 2, 2, 2, 3, 7, 6]
[0, 0, 1, **3**, 2, 2, **2**, 3, 7, 6]
[0, 0, 1, 2, 2, 2, 3, 3, 7, 6]
…
[0, 0, 1, 2, 2, 2, 3, 3, 6, 7]
Finished...

[**9**, 8, 7, 6, 5, 4, 3, 2, 1, **0**]
[0, 8, 7, 6, 5, 4, 3, 2, 1, 9]
[0, **8**, 7, 6, 5, 4, 3, 2, **1**, 9]
[0, 1, 7, 6, 5, 4, 3, 2, 8, 9]
[0, 1, **7**, 6, 5, 4, 3, **2**, 8, 9]
[0, 1, 2, 6, 5, 4, 3, 7, 8, 9]
[0, 1, 2, **6**, 5, 4, **3**, 7, 8, 9]
[0, 1, 2, 3, 5, 4, 6, 7, 8, 9]
[0, 1, 2, 3, **5**, **4**, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Finished...

Denne metoden bruker siste element som pivot, søker fra høyre etter element mindre enn pivoten og søker fra venstre etter element større enn pivoten. Om paret blir funnet bytter de plass.
Når et slikt par ikke blir funnet velger vi en ny pivot og setter den helt til høyre. Gjenta prosess.

Her er lista for hvert rekursjons kall til random og reversert array.
Ingen elementer blir byttet plass på for sortert input. Dette er fordi vi har valgt siste element som pivot (9).

Markerer par som bytter plass.
For reversert input finner vi et mønster med to diagonaler rettet mot midten.
Pivoten 0 blir byttet mot 9 som blir den nye pivoten.
Vi søker etter større verdier fra venstre og mindre verdier fra høyre.
Siden inputen er reversert finner vi paret med en gang.
Indeks i øker hele tiden med 1, mens indeks j hele tiden minker med 1.

mergeSort() (flettesortering)

New random in:

[0, 5, 9, 8, 9, 8, 6, 3, 4, 1]

Merge-sort steps output:

[0, 5, 9, 8, 9, 8, 6, 3, 4, 1]
[0, 5, 9, 8, 9, 8, 6, 3, 4, 1]
[0, 5, **9**, **8**, 9, 8, 6, 3, 4, 1]

[0, 5, 8, 9, 9, **8**, **6**, 3, 4, 1]
[0, 5, 8, 9, 9, **6, 8**, **3**, 4, 1]
[0, 5, 8, 9, 9, 3, 6, 8, **4**, **1**]
[0, 5, 8, 9, 9, **3, 6, 8**, **1, 4**]

[**0, 5, 8, 9, 9**, **1, 3, 4, 6, 8**]
[0, 1, 3, 4, 5, 6, 8, 8, 9, 9]
Finished...

[**9**, **8**, 7, 6, 5, 4, 3, 2, 1, 0]
[**8, 9**, **7,** 6, 5, 4, 3, 2, 1, 0]
[7, 8, 9, **6**, **5**, 4, 3, 2, 1, 0]
[**7, 8, 9**, **5, 6**, 4, 3, 2, 1, 0]

[5, 6, 7, 8, 9, **4**, **3**, 2, 1, 0]
[5, 6, 7, 8, 9, **3, 4**, **2**, 1, 0]
[5, 6, 7, 8, 9, 2, 3, 4, **1**, **0**]
[5, 6, 7, 8, 9, **2, 3, 4**, **0, 1**]

[**5, 6, 7, 8, 9**, **0, 1, 2, 3, 4**]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Finished...

Denne metoden starter med venstre halvdel og bryter den ned i flere og flere halvdeler.
Deretter blir disse delene flettet sammen helt til hele venstre halvdel er sortert.
Deretter skjer dette med høyre halvdel.
Til slutt blir hele venstre halvdel flettet sammen med hele høyre halvdel.

Printer ut for hvert kall på mergeTogether() metoden. Markerer fletting av halvdeler når de endres.
For sortert input blir den ikke mutert så vi printer ingen steg.

For sortering av reversert input har vi identisk mønster for venstre halvdel og høyre halvdel.
Det er fordi [9, 8, 7, 6, 5] og [4, 3, 2, 1, 0] er "identiske" og blir sortert hver for seg

- **Tabell fra testresultatene: (Kjørt med Intel i7-8700)**

selectionSort random **repeat** input time table:

size	time (ms)	java time (ms)
5000	36	0
25.000	754	1
125.000	18612	2

selectionSort() random **unique** input time table:

size	time (ms)	java time (ms)
5000	19	0
25.000	644	0
125.000	16314	0

selectionSort() **decreasing** input time table:

size	time (ms)	java time (ms)
5000	6	0
25.000	565	0
125.000	22021	0

selectionSort() **sorted** input time table:

size	time (ms)	java time (ms)
5000	2	0
25.000	66	0
125.000	1646	0

quickSort() random **repeat** input time table:

size	time (ms)	java time (ms)
5000	1	0
25.000	3	0
125.000	11	2
625.000	49	0
3.125.000	267	1
15.625.000	1478	4
78.125.000	7233	22

quickSort() random **unique** input time table:

size	time (ms)	java time (ms)
5000	1	0
25.000	1	0
125.000	8	0
625.000	44	1
3.125.000	243	1
15.625.000	1329	4
78.125.000	7201	23

quickSort() **decreasing** input time table:

size	time (ms)	java time (ms)
5000	3	0

quickSort() **sorted** input time table:

size	time (ms)	java time (ms)
5000	3	0

mergeSort() random **repeat** input time table:

size	time (ms)	java time (ms)
5000	1	0
25.000	3	0
125.000	13	0
625.000	70	1
3.125.000	404	1
15.625.000	2249	6
78.125.000	11598	21

mergeSort() random **unique** input time table:

size	time (ms)	java time (ms)
5000	1	0
25.000	2	0
125.000	12	0
625.000	74	0
3.125.000	400	1
15.625.000	2218	4
78.125.000	11619	29

mergeSort() **decreasing** input time table:

size	time (ms)	java time (ms)
5000	0	0
25.000	1	0
125.000	5	0
625.000	44	0

mergeSort() **sorted** input time table:

size	time (ms)	java time (ms)
5000	0	0
25.000	1	0
125.000	4	0
625.000	24	1

Kort om resultatene.
Første tiden i quickSort() decreasing og quickSort() sorted overasker meg.
Merk at decreasing- og sorted-input bare ble kjørt med size = 5000 pga. stackoverflow problemer.

Det overasker meg hvor mye raskere java.util.Arrays.sort er. Nederst i quickSort() repeat ser vi at java sin sort metode kjører 329 ganger raskere. Forventet en god del raskere men ikke så mye.
Input typene som står ut er decreasing og sorted.
For selectionSort() ser vi at decreasing input er den treigeste typen, mens sorted er den raskest, akkurat som forventet.
For mergeSort() ser vi at deacresing- og sorted input er raskest. Dette er forventet ettersom det er lett å flette sammen en decreasing array delt i to.

Kjøretid og O-notasjon.
Utplukkingssortering, kvikksortering, flettesortering har henholdsvis O(n^2), O(n log n) og O(n log n) kjøretid.

Merk at vi inkrementerer size med en faktor 5 hver gang.
For å sjekke kjøretiden til selectionSort() bruker vi (5n)^2 = 25 n^2 og sjekker at neste kjøretid er 25 ganger forrige. For quickSort() og mergeSort() sjekker vi at tiden er "litt større" en lineær tid.

For repeat, unique og sorted input i selectionSort() ser vi at kjørtiden er O(n^2) som forventet. (25* forrige tid)
For decreasing input ser det ut til at kjøretiden er litt større enn O(n^2), men ikke O(n^3).

For repeat input i quickSort() ser vi at kjøretiden er < O(n) for size = 5000, 25.000, 125.000 og 625.000
(5* forrige tid) Fra size = 3.125.000 til size = 15.625.000 ser vi at kjøretiden er litt over O(n) som forventet.

For unique input ser vi at alle tidene er ca. O(n logn) med unntak av de to første like verdiene.

For repeat input i mergeSort() ser vi at de 3 første tidene er < O(n) mens de resterende er omtrent O(n logn).
For unique input de to første tidene < O(n), mens alle andre er omtrent O(n logn).
Decreasing- og sorted-input ser ut til å kjøre O(n logn).

Oppsumert så vil jeg si at teorien stemmer med praksis i dette tilfellet.
Det er ikke veldig overaskende at kjøretiden er litt feil for de små input arrayene ettersom kjøretiden er svært liten (1 - 5 ms), samtidig som hardware hastigheter vil ha små variasjoner.