IN2010 høsten 2019

Obligatorisk oppgave 3: Sortering

Utplukkssortering (selection-sort; algoritme 5.3 s. 160)

Java koden

```
int[] selectionSort(int[] array){
   int n = array.length;
   for(int i=0; i < n-1; i++){
        int s = i;
    for(int j=i+1; j<n; j++){
         if(array[j] < array[s]){
              s = j;
      }
     }
     swap(array,i,s);
   }
   return array;
 }
```

Valg og utfordringer

Denne algoritmen var ganske lett å implementere, jeg hadde egentlig ingen problemer med det.

Test av korrekthet (10 elementer)

tilfeldig - 0.0058944
sortert etter stigende verdier - 0.0039904
sortert etter fallende verdier (dvs reversert) - 0.0064961

Mønster

Denne algoritmen sammenligner tallet på indeksen med alle tallene som var besøkt før. Algoritmen erstatter det minste med første elementet i lista.

Kvikksortering (quick-sort; algoritme 8.9/8.11 s.255/6)

Java koden

```
int[] inPlaceQuickSort(int[] array, int fraIndex, int tilIndex){
   int left;
   while(fraIndex < tilIndex){
     left = inPlacePartition(array, fraIndex, tilIndex);
```

```
    if(left - fraIndex < tilIndex - left){
      inPlaceQuickSort(array, fraIndex, left-1);
      fraIndex = left+1;
    } else {
      inPlaceQuickSort(array, left+1, tilIndex);
      tilIndex = left-1;
    }
  }
  return array;
}

int inPlacePartition(int[] array, int fraIndex, int tilIndex){
  int pivot = array[tilIndex];
  int left = fraIndex;
  int right = tilIndex-1;
  while(left <= right){
    while(left <= right && array[left]<= pivot){
      left++;
    }
    while(right >= left && array[right]>=pivot){
      right--;
    }
    if(left<right){
      swap(array,left,right);
    }
  }
  swap(array,left,tilIndex);
  this.printArray(array);
  return left;
}
```

Valg og utfordringer

Det var ganske vanskelig å implementere denne. Derfor valgte jeg å lage to metoder - en som sorteter og den andre som partisjonerer arrayet i to deler.

Test av korrekthet (10 elementer)

tilfeldig - 0.0038464
sortert etter stigende verdier - 0.0038839
sortert etter fallende verdier (dvs reversert) - 0.002614

Mønster

Algoritmen bruker en pivot, og setter alle tallene som er større til høyre for pivoten, og alle som er mindre til venstre. Deretter velges det en ny pivot fra partisjonen til venstre, og en fra partisjonen til høyre, og prosessen gjentas til alle tallene er på korrekt plass.

Flettesortering (merge-sort; algoritme 8.3 s. 245)

Java koden

```
int[] mergeSort(int[] arr, int lengde) {
      long t = System.nanoTime(); //nanosekunder

      if (lengde < 2) {
        return arr;
      }

      int midten = lengde / 2;
      int[] left = new int[midten];
      int[] right = new int[lengde - midten];


      for (int i = 0; i < midten; i++) {
        left[i] = arr[i];
      }

      for (int i = midten; i < lengde; i++) {
        right[i - midten] = arr[i];
      }

      mergeSort(left, midten);
      mergeSort(right, lengde - midten);
      merge(arr, left, right, midten, lengde - midten);

      double tid = ( System.nanoTime() - t ) / 1000000.0; //millisekunder

      return arr;
}

 void merge(int[] array, int[] left, int[] right, int leftLength, int rightLength) {

    int i = 0;
    int j = 0;
    int indeks = 0;

    while (i < leftLength && j < rightLength) {
       if (left[i] <= right[j]) {
         array[indeks++] = left[i++];
       }
       else {
         array[indeks++] = right[j++];
       }
    }
    while (i < leftLength) {
      array[indeks++] = left[i++];
    }
```

```
    while (j < rightLength) {
      array[indeks++] = right[j++];
    }
  }
```

Valg og utfordringer

Denne algoritmen var veldig vanskelig å implementere for meg, og det var vanskelig å forstå om algoritmen fungererte riktig eller ikke. Til slutt forstod jeg at det burde være to sorterte lister som skal flettes sammen ved å sammenligne de første elementene i hver liste opp mot hverandre, og da ble det enklere.

Test av hastighet (10 elementer)

tilfeldig - 0.0042599
sortert etter stigende verdier - 0.0031783
sortert etter fallende verdier (dvs reversert) - 0.0031907

Mønster

Arrayet deles opp i mindre og mindre biter, helt til det kun er to elementer i hver bit igjen. Deretter flettes de mindre bitene sammen til større biter, og de større bitene flettes sammen osv. helt til arrayet er sortert.

Testresultater

Utplukkssortering (selection-sort; algoritme 5.3 s. 160)

	1000	5000	10000	50000	100000	250000
Tilfeldig	0.3005895	1.3849161	4.7176344	109.8386838	433.4712146	2665.2303454
sortert etter	0.0832733	1.0897626	4.3893331	108.5815858	431.3879184	2655.4665471

stigende verdier						
sortert etter fallende verdier	0.189752	0.5972987	2.4228324	63.5844937	251.4434494	1554.0761963

Kvikksortering (quick-sort; algoritme 8.9/8.11 s.255/6)

	1000	5000	10000	50000	100000	250000
Tilfeldig	0.0061534	0.0040852	0.013142	0.0196948	0.0314111	0.201140
sortert etter stigende verdier	0.0025945	0.0031189	0.0098435	0.0325169	0.0414852	0.0724531
sortert etter fallende verdier	0.0032242	0.0062379	0.0054653	0.0169931	0.025694	0.0744841

Flettesortering (merge-sort; algoritme 8.3 s. 245)

	1000	5000	10000	50000	100000	250000
Tilfeldig	0.0046061	0.0044186	0.0125577	0.0130276	0.0281193	0.0525599
sortert etter stigende verdier	0.0024727	0.0043876	0.0080718	0.0181407	0.0226753	0.052928
sortert etter fallende verdier	0.0043515	0.0033571	0.0091303	0.0148928	0.0361339	0.0487778

Faktisk kjøretid vs. Forventet kjøretid

Forventet kjøretid for de ulike algoritmene er:

- Selection sort: worstcase: O(n^2), average: O(n^2)
- Quicksort: worstcase: O(n^2), average: O(nlogn)
- Merge sort: worstcase: O(nlogn), average: O(nlogn)

Vi kan se ut ifra de forventede kjøretidene at vi forventer at selection sort er tregest, quicksort litt raskere ettersom average runtime er raskere enn selection sort, og merge sort er raskest. Dette ser vi også stemmer med de faktiske kjøretidene: selection sort er generelt tregest, quicksort litt raskere, og merge sort er klart den raskeste algoritmen.