Obligatorisk Oppgave 3

Sortering - filipcl

Rapporten svarer på spørsmålene separat for hver algoritme, tabellene viser kjøretid for hver enkelt algoritme hvor den raskeste kjøretiden er markert med grønt.

Selections-Sort:

Selection-sort var ganske rett frem, eneste utfordringen eller mer en oppdagelse jeg gjorde var å skrive en egen gjenbrukbar metode for bytting av elementer kalt swap().

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

Mønstre:

Selection-sort algoritmen har et felles mønster som er uavhengig av input type (random/sortert/reversert), Selection-sort vil alltid traversere igjennom listen for å se etter et element som er mindre enn nåværende element for å swape det foran i listen.

Kjøretid i ms:

Det overrasket meg hvor ineffektiv selection sort blir når antall noder i listen stiger, det jeg maks gadd å teste den på var når n = 500 000.

Array(n)	Random input	Sortert input	Reversert input
1000	45.2993 ms	2.0377 ms	2.2657 ms
5000	66.3277 ms	11.5508 ms	5.8618 ms
10 000	104.0017 ms	47.1154 ms	25.1942 ms
50 000	2165.6434 ms	1278.6899 ms	2677.1253 ms

500 000	5587.1976 ms	6849.6823 ms	2406.5512 ms

Faktisk kjøretid VS forventet kjøretid:

Her stemmer teori med praksis, o-notasjonen til Selection-sort er O(n^2) på grunn av at dens nøstede for-løkke. Den faktiske kjøretiden gjenspeiler den forventede kjøretiden uavhengig av input type ved at kjøretiden i ms har en eksponentiell økning ved økning i antall elementer.

Quick-Sort:

Utfordringen i Quicksort for min del var å forstå eksempelet i Algorithm 8.9 fra boken, og implementere metoden inPlaceQuickSort(). inPlacePartition() sin sudokode var enklere å forstå. Jeg brukte en del tid på å forstå hvordan jeg skulle kalle på quickSortPartition() for å sette riktig pivot element.

I quick-sort algoritmen min forekommer det ingen spesielle mønster annet enn at algoritmen bruker mest tid på å sortere en liste med random rekkefølge. Nest mest tid på å sortere en liste som er sortert i fallende rekkefølge, og minst tid på å sortere en liste i stigende rekkefølge – da algoritmen kun traverserer gjennom listen uten å sortere.

```
    void quickSort(int[] list, int start, int end){
        int left;
        while(start < end){
          left = quickSortPartition(list, start, end);

          if(left-start < end-left){
            quickSort(list, start, left-1);
            start = left+1;
          } else {
            quickSort(list, left+1, end);
            end = left- 1;
          }
        }
      }
```

```
int quickSortPartition(int[] list, int start, int end){
        int pivot = list[end];
        int left = start;
        int right = end-1;

        while(left <= right){
```

```
        while(left <= right && list[left]<= pivot){
          left++;
        }
        while(right >= left && list[right]>=pivot){
          right--;
        }
        if(left < right){
          swap(list, left, right);
        }
      }
      swap(list, left, end);
      return left;
    }
```

Mønstre:

Quick-sort algoritmen vil uavhengig av type input (random/sortert/reversert) sette et pivot punktet og utføre particion, eller deler opp i sub-array av mindre og større enn pivot punktet. Dette fungerer bra på lister av type «Random», men tar quick-sort inn en liste som er reversert vil den sjekke noe er større enn pivot punktet og deretter sjekke om noe er mindre enn pivot punktet før den swaper(). Når listen er sortert vil algoritmen sjekke om neste er større enn pivot punktet. Dette fører til at når lengden på listen blir stor er quickSort raskere til å sortere lister av typen random enn av typen sortert eller reversert.

Kjøretid i ms:
Det var kult å se at det på et punkt er raskere å sortere en random liste enn en sortert liste ved bruk av quickSort.

Array(n)	Random input	Sortert input	Reversert input
1000	45.4935 ms	1.6062 ms	1.532 ms
5000	48.8052 ms	20.1705 ms	4.0499 ms
10 000	58.8618 ms	26.7063 ms	9.0503 ms
50 000	101.1884 ms	277.206 ms	278.8942 ms
100 000	215.659 ms	1083.4659 ms	2252.6239 ms
100 0000	13996.0793 ms	158958.9651 ms	175091.7244 m

Faktisk kjøretid VS forventet kjøretid:

Worst case for quickSort oppstår når partisjonsprosessen alltid velger det største eller minste elementet som pivot. Siden vi setter det siste element til å være pivot punktet, vil det verste tilfelle oppstå når matrisen allerede er sortert i økende eller synkende rekkefølge. I dette tilfellet vil quickSort få en O-notasjon på O(n^2). Dette gjenspeiles i resultatene i tabellen over hvor tiden i millisekunder har en eksponentiell økning i antall ms det tar å sortere en liste med 50 000 elementer og en liste med 100 000 elementer. Og det vil da være raskere å sortere en liste av samme størrelse

som er random, da man kan få ett pivot punkt som er mer mot medianen. Så her stemmer teorien med praksis.

Heap-Sort:

Utfordringen med Heap-sort var for min del å forstå hvordan jeg skulle implementere buildHeap, selve prosessen var godt dokumentert i boken og teorien var grei, men jeg brukte tid på å implementere det i Java.

```
 public void heapSort(int list[]) {
        int n = list.length;
        for (int i = n / 2 - 1; i >= 0; i--
) //Bygger en heap
            buildHeap(list, n, i);
        for (int i=n-1; i>=0; i--
) { // Tar ut et og et element og bytter det største elemente
t med root
            swap(list, 0 , i);
            buildHeap(list, i, 0);
        }
    }

    void buildHeap(int list[], int n, int i) {
        int largest = i; // Største element tilsvarer root
        int left = 2*i + 1;
        int right = 2*i + 2;

        // If left child er større enn root
        if (left < n && list[left] > list[largest])
            largest = left;

        // If right child er større enn current største eleme
nt
        if (right < n && list[right] > list[largest])
            largest = right;

        // If størst is not root
        if (largest != i) {
            swap(list , i , largest);
```

```
            buildHeap(list, n, largest);
        }
    }
```

Mønstre:

Heap-sort algoritmen vil alltid bygge en heap ut ifra listen den tar inn, i vårt tilfelle vil den bygge en Max heap. Det å ta ut elementer fra en max heap vil ta like lang tid uavhengig av input typen på listen. Men uavhengig av listens type må den bygges om til en max heap for den sorteres. Dette vil føre til forskjellig kjøretid, er inputen en liste av typen random vil det ta vesentlig lengre tid å bygge om listen til en max heap, enn vis listen var av typen sortert eller reversert. Noe som fører til at uavhengig av størrelsen på listen vil heapSort alltid bruke lengre tid på å sortere en liste av type random.

Kjøretid i ms:
Jeg er imponert over hvor rask heapSort når antall noder i listen øker.

Array(n)	Random input	Sortert input	Reversert input
1000	37.9828 ms	0.33 ms	0.2684 ms
5000	39.1107 ms	0.7424 ms	0.7406 ms
10 000	43.9959 ms	1.9642 ms	1.9678 ms
50 000	52.7208 ms	3.2838 ms	3.2902 ms
100 000	56.4116 ms	9.4651 ms	9.3051 ms
100 0000	129.6474 ms	95.1322 ms	78.7375 ms

Faktisk kjøretid VS forventet kjøretid:

Heap-sort er en inplace algoritme, den må bygge en heap uavhengig av input typen det vil si at buildHeap har en O-notasjon på O(n). Når elementene er sortert til en heap, skal elementene tas ut og den største noden i heapen skal flyttes til root. Dette har en O-notasjon på O(Log n) siden kjøretiden vokser i proporsjon med input størrelsen. Dette betyr at heapSort har en O-notasjon på O(n log n). noe som gjenspeiles i tabellen ved at uavhengig av type input øker kjøretiden proporsjonalt med størrelsen på input.

Arrays.Sort():

Kjøretid i ms:
Sett opp mot de andre algoritmene er tidsbruken til .Sort() utrolig imponerende.

Array(n)	Random input	Sortert input	Reversert input
1000	36.3827 ms	0.1393 ms	0.0949 ms
5000	40.4779 ms	0.1068 ms	0.1195 ms

10 000	37.2233 ms	0.1447 ms	0.106 ms
50 000	32.7015 ms	0.2396 ms	0.1978 ms
100 000	43.9708 ms	0.2747 ms	0.3896 ms
100 0000	42.2924 ms	2.8654 ms	2.4378 ms

Testing av algoritmene:

I tabellene som viser kjøre tid over de forskjellige algoritmene har jeg kun inkludert tider hvor jeg har elementene i listen har vært på en rang mellom 0 – 10. Dette gjorde jeg for å generalisere slik at hadde likt utgangspunkt. Men jeg testet også alle algoritmene med disse forskjellige begrensninger ved å endre i konstruktøren:

Alle elementer i Arrayet er likt:

```
- this.sortedArray[i] = 1;
```

Alle elementene er random (Både negative og positive ints) :

```
- this.sortedArray[i] = random.nextInt();
```

Elementer mellom 0 og 100 0000:

```
- this.sortedArray[i] = random.nextInt(100000);
```