Sorteringsalgoritmer

Bubblesort (boblesortering)

- **Analogi**:
  De største elementer "bobler" op gennem array'et, indtil alle elementer er sorteret
- **Karakteriske operationer**:
  De karakteriske operationer er en sammenligning evt. efterfulgt af en ombytning. Foretages *n* - 1 gange for hver "boble"-fase.
- **Tidskompleksitet**:
  $O(n^2)$

```
public static void Boblesortering(int[] data, int n) {
    int antal = 0; // Angiver antal sorterede elementer
    int index; // Angiver aktuelle indeks
    while (antal < n) {
        for (index = 1; index < n - antal; index++) {
            if (data[index] < data[index - 1])
                byt(data, index, index - 1);
        }
        antal++;
    }
}
```

Selectionsort (udvælgelsessortering)

- **Analogi**:
  De største elementer udtrækkes og indsættes på det rigtige sted, ligesom når man fylder frugt eller grøntsager i kurven i den lokale brugsforening, eller tømmer posen med Matadormix lørdag aften i fjernsynets skær...
- **Karakteriske operationer**:
  De karakteriske operationer er en sammenligning. Foretages *n* - 1 gange pr. gennemløb, medens der kun foretages én ombytning.
- **Tidskompleksitet**:
  $O(n^2)$

```
public static void Selektionssortering(int[] data, int n) {
    int antal = 0; // Angiver antal sorterede elementer
    int index; // Angiver aktuelle indeks
    int max; // Angiver indeks for største element
    while (antal < n) {
        max = 0;
        for (index = 1; index < n - antal; index++) {
           if (data[max] < data[index])
               max = index;
        }
        byt(data, max, n - antal - 1);
        antal++;
    }
}
```

Insertionsort (indsættelsessortering)

- **Analogi**:
  Elementerne indsættes på deres rigtige plads i forhold til de elementer, der allerede er sorteret, efterhånden som de gennemløbes, ligesom man placerer kortene på hånden, når man excellerer i "kortenspil".
- **Karakteriske operationer**:
  De karakteriske operationer er en sammenligning evt. efterfulgt af en tildeling. Foretages n - 1 gange pr. gennemløb.
- **Tidskompleksitet**:
  $O(n^2)$

```
public static void Indsættelsessortering(int[] data, int n) {
    int antal = 1; // Angiver antal sorterede elementer
    int index; // Angiver aktuelle indeks
    int tmp;
    while (antal < n) {
        tmp = data[antal];
        for (index = antal; index > 0; index--) {
            if (tmp < data[index - 1])
                data[index] = data[index - 1];
            else
                break;
        }
        data[index] = tmp;
        antal++;
    }
}
```

Mergesort (flettesortering)

- **Analogi**:
  Hvis man fx var to om at skulle sortere et spil kort, så kunne én fremgangsmåde være at dele stakken og hver især sortere sin stak, for derefter at flette de to sorterede bunker. Hvis man var flere, så kunne man vedblive at dele stakkene, indtil en stak kun indeholdt ét eller to kort. I det første tilfælde ville en sortering være triviel, medens en sortering i det andet tilfælde ville bestå af en sammenligning samt en evt. ombytning af de to kort. Algoritmen har altså to processer: Sortering af data og fletning af data.
- **Karakteriske operationer**:
  Den karakteriske operation i metoden **flet** er en sammenligning efterfulgt af en tildeling, som foretages n - 1 gange, altså $O(n)$. Metoden kalder sig selv rekursivt i alt $\log_2 n$ gange.
- **Tidskompleksitet**:
  $O(n \ln n))$
- **Karakteristika**:
  Metoden er god til at håndtere store datamængder, men gør det ved brug af megen plads til midlertidige arrays. Metoden er specielt god til store datamængder, der er næsten sorteret.

Metoden flet (merge)

Metoden herunder fletter to sorterede arrays. Overtyd dig selv om, at det passer!

```
private static void flet(int[] data, int[] tmp, int lav, int middel, int høj) {
    int ri = lav;
    int ti = lav;
```

```
        int di = middel;
        while (ti < middel && di <= høj) {
            if (data[di] < tmp[ti])
                data[ri++] = data[di++];
            else
                data[ri++] = tmp[ti++];
        }
        while (ti < middel)
            data[ri++] = tmp[ti++];
    }
```

Metoden flettesortering (mergeSortRecursive)

Metoden herunder opdeler et array i to, hvorefter hver halvdel sorteres (ved et rekursivt kald), og derefter flettes med metoden **flet**. En stak bestående af n kort kan deles med 2 i alt $\log_2 n$ gange, prøv selv!

```
    private static void flettesortering(int[] data, int[] tmp, int lav, int høj) {
        int n = høj - lav + 1;
        int middel = lav + n / 2;
        if (n < 2) return;
        for (int i = lav; i < middel; i++)
            tmp[i] = data[i];
        flettesortering(tmp, data, lav, middel - 1);
        flettesortering(data, tmp, middel, høj);
        flet(data, tmp, lav, middel, høj);
    }
```

På hvert logisk niveau i sorteringsalgoritmen (Husk, der er $\log_2 n$ *(dvs.* $O(\ln n)$) niveauer) foretages der en fletning, som indeholder $O(n)$ operationer. Den samlede tidskompleksitet bliver derfor $O(n \ln n)$.

Metoden Flettesortering (MergeSort)

Metoden Flettesortering, (MergeSort), får nu følgende udseende, hvis vi skal være konsistente med de 3 foregående sorteringsalgoritmer:

```
    public static void Flettesortering(int[] data, int n) {
        Flettesortering(data, new int[n], 0, n - 1);
    }
```

Quicksort

Udviklet i 1961 af C.A.R. Hoare og er ligesom **Flettesortering** en "del og hersk" algoritme ("Divide et impera" - på engelsk: "Divide and conquer". Citatet tilskrives Phillip af Makedonien)

- **Analogi**:
  Sorteringen virker ved at udvælge et pivot-element (et ankerpunkt eller omdrejningselement), som skal adskille de store fra de små elementer. Man foretager altså en "råsortering" omkring pivot-elementet, der nu er på sin endelige plads. Herefter vælges et pivot-element på hver side af det oprindelige, og endnu en "råsortering" foretages. Denne "råsortering" eller opdeling udføres med metoden **opdel**.
- **Karakteriske operationer**:
  Metoden **opdel** foretager en sammenligning evt. efterfulgt af en ombytning i alt $O(n)$ gange
- **Tidskompleksitet**:
  $O(n \ln n)$ - i gennemsnit, men $O(n^2)$ i værste fald... Værste fald optræder, når elementerne på forhånd er sorteret.
- **Karakteriska**:
  Metoden er velegnet til datamængder, der usorteret, men dårlig til mængder, der er næsten sorteret.

Metoden opdel

Metoden er lidt tricky, men prøv at dissekere den og se, hvad der egentlig foregår: (Bemærk, at index for hvert gennemløb ændres, således at **højre** og **venste** mødes efter *n* skridt. Altså en tidskompleksitet på *O*(*n*).

```
private static int opdel(int[] data, int venstre, int højre) {
    while (true) {
        while (venstre < højre && data[venstre] < data[højre])
            højre--;
        if (venstre < højre)
            byt(data, venstre++, højre);
        else
            return venstre;
        while (venstre < højre && data[venstre] < data[højre])
            venstre++;
        if (venstre < højre)
            byt(data, venstre, højre--);
        else
            return højre;
    }
}
```

Metoden quickSortRekursiv

Metoden finder et pivot-element vha. metoden **opdel**, som er *O*(*n*), hvorefter den kalder sig selv rekursivt på mængderne på begge sider af pivot-elementet. Husk, at en mængde på *n* elementer kan halveres $\log_2 n$ gange - *O*(ln*n*), hvilket vil være i heldigste fald. Opdelingen kan dog i værste fald ske, således at den ene mængde kun kommer til at indeholde ét element og den anden *n* - 1 elementer. På denne måde kan mængden opdeles *n* - 1 gange - *O*(*n*).

```
private static void quickSortRekursiv(int[] data, int venstre, int højre) {
    if (venstre < højre) {
        int pivot = opdel(data, venstre, højre);
        quickSortRekursiv(data, venstre, pivot - 1);
        quickSortRekursiv(data, pivot + 1, højre);
    }
}
```

Metoden QuickSort

Motoden **QuickSort** får nu følgende udseende:

```
public static void QuickSort(int[] data) {
    quickSortRekursiv(data, 0, data.Length - 1);
}
```

Otto Knudsen
29.11.05