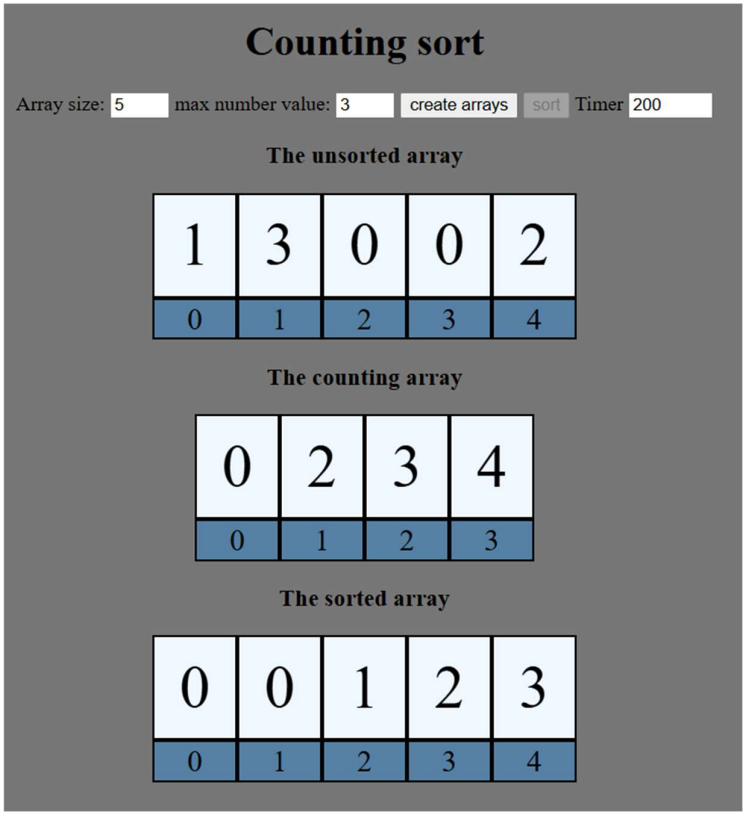
Counting Sort

Lavet af: Jon Lundby Nielsen

Github: https://jonlundby.github.io/counting-sort/



Projektet:

Som midtvejsprojekt har jeg valgt at lave en visualisering af countingsort algoritmen. Mit mål er at synliggøre de faser som algoritme gennemgår så man kan se hvert enkelt trin i algoritmens proces. Der er en knap (create arrays) til at initialisere de tre arrays som algoritmen bruger. Et usorteret unsortedArr med tilfældige numre, countingArr og et sortedArr der er tilsvarende det usorterede array men med tomme elementer. Derudover er der en knap (sort) der starter sorteringen og en timer til at sætte hastigheden for hvor hurtigt algoritmen skal gennemføres.

Datastruktur:

Datastrukturen er blot et almindeligt array og jeg har brugt det array som er indbygget i Javascript. Et array er klart at foretrække for at bruge counting sort effektivt da algoritmen gør

brug af direkte adgang til indeks som f.eks. ”myArr[i]”. Således består modellen altså også af de tre arrays unsortedArr, countingArr og sortedArr. Disse arrays er blot brug globalt i controlleren selvom man godt kunne have lavet et statisk eller dynamisk array som model.

Algoritme:

Counting sort er en kendt sorterings algoritme som hører under kategorien ”non-komparativ”, altså gør algoritmen ikke brug af sammenligninger af elementerne i arrayet. Algoritmen kan beskrives i tre faser; en tælle fase, kumulativ fase, og selve sorteringen. Før de 3 faser finder man den højeste værdi i det usorterede array. Denne højeste værdi bruges til at bestemme længden af tælle/counting arrayet.

I den første tælle fase optælles alle værdierne fra det usorterede array således at tælle arrayets indeks pladser svarende til værdien fra det usorterede array forøges med én. F.eks. hvis værdien 5 læses for første gang fra det usorterede array så vil værdien på indeks plads 5 blive til 1 og så fremdeles.

```
// optælling af værdier
async function countingPhase() {
    for (let i = 0; i < unsortedArr.length; i++) {
        let countingArrIndex = unsortedArr[i];

        countingArr[unsortedArr[i]]++;

        view.updateVisualCountingArray(countingArr, countingArrIndex);
        await sleep(timer);
    }

    cumulativePhase();
}
```

I den kumulative fase lægges værdierne i tælle arrayet sammen. Hvis tælle arrayet holder værdierne 2, 3, 1 så vil det blive til 2, 5, 6.

```
// kumulativ opsummering af countingArr
async function cumulativePhase() {
    for (let i = 1; i < countingArr.length; i++) {
        countingArr[i] += countingArr[i - 1];

        view.updateVisualCountingArray(countingArr, i);
        await sleep(timer);
    }

    sortPhase();
}
```

I den sidste fase gennemgås det oprindeligt usorterede array baglæns vha. et for-loop. Ved hver iteration bliver den usorterede værdi brugt til at finde tælle arrayets tilsvarende indeks. Derefter bruges værdien fra tælle arrayets indeks til at finde indeks i det sorterede array og den oprindelige værdi fra det usorterede array indsættes så på dette indeks i det sorterede array. Når en værdi indsættes i det sorterede array, så trækkes der én fra værdien i tælle arrayet og grunden til dette er at der kan forekomme ens værdier.

```
// sortering af usorterede værdier gennem countingArr og indexes
async function sortPhase() {
    for (let i = unsortedArr.length - 1; i >= 0; i--) {
        let countingArrIndex = unsortedArr[i];
        let sortedArrIndex;

        // finder værdien af et index i counting array baseret på en værdi fra unsorted array
        sortedArr[countingArr[unsortedArr[i]] - 1] = unsortedArr[i];
        // trækker 1 fra værdien i countingArr
        countingArr[unsortedArr[i]] = countingArr[unsortedArr[i]] - 1;

        sortedArrIndex = countingArr[unsortedArr[i]];

        view.updateVisualCountingArray(countingArr, countingArrIndex);
        view.updateVisualSortedArray(sortedArr, sortedArrIndex);
        await sleep(timer);
    }
}
```

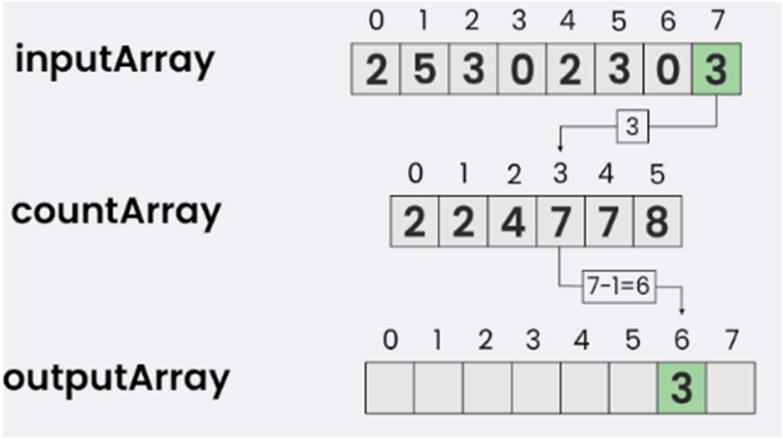
BigO(?):

Counting sort har BigO notationen BigO(n + k). Man kunne også sige at notationen hed BigO(3n + k) da det usorterede array gennemløbes 3 gange men da BigO ikke tage højde for konstante værdier så ender den altså på n + k. *k* er størrelsen på det interval som værdierne

i det usorterede array kan have og også længden på tælle arrayet som jo også gennemløbes i den kumulative fase. Dette forværrer space complexity men som sådan ændre det ikke noget for time complexity.

Inspiration:

På https://www.geeksforgeeks.org/counting-sort/ har de en ret god gennemgang af counting sort og meget gode illustrationer af de forskellige steps i counting sort som f.eks. på billedet herunder.



Udover GeeksForGeeks har jeg også læst efter behov på

https://www.w3schools.com/dsa/dsa_algo_countingsort.php

og

https://en.wikipedia.org/wiki/Counting_sort