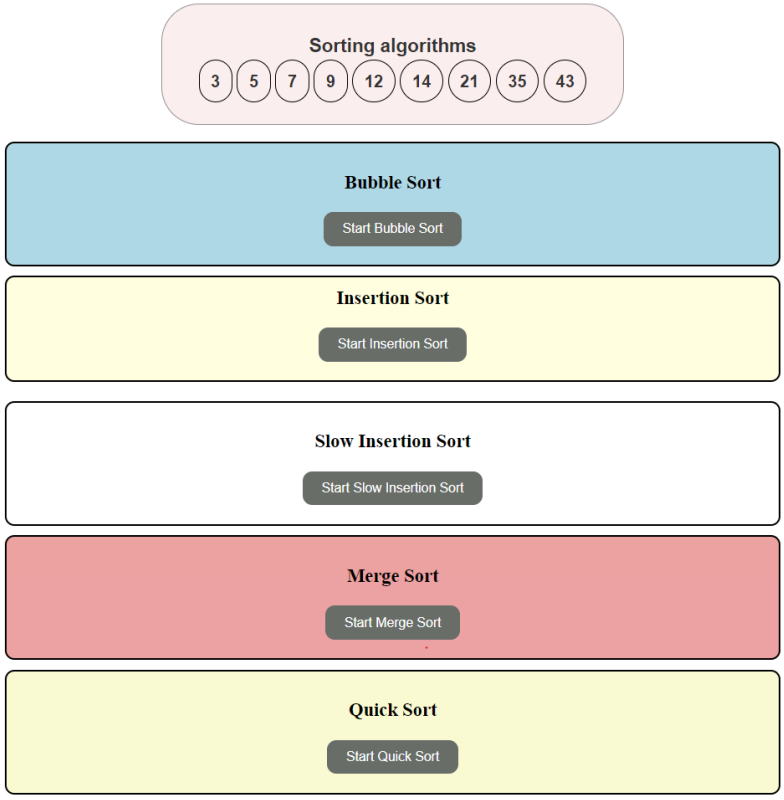
Sorterings Algoritmer



Naja Egede Moe

DSA F24

https://github.com/najamoe/sortAlgorithm

Deployet projekt fra github pages:

https://najamoe.github.io/sortAlgorithm/

Projektbeskrivelse

Algoritmer er en proces, som skal give et bestemt resultat. Det kan både udføres af mennesker, men når det kommer til større mængder af data, bruger vi programmer til at gøre det for os. En algoritme skal kunne forstås og udføres på én måde.

Når vi filtrerer vores mails fra ældst til nyeste, eller når vi sorterer listen af sko på Zalando efter højeste til laveste pris, bruger vi sorteringsalgoritmer.

Jeg har opsat fire forskellige typer af algoritmer, som sorterer en række af tal efter lavest til højest. I visualiseringen ser man fire forskellige sorteringsalgoritmer. De har alle fået den samme talrække, som de skal sortere. Ved hver algoritme kan man, når sorteringen er færdig, se hvor lang tid det har taget dem at sortere de ni cifre.

Valgte algoritmer

Der er valgt fire forskellige sorteringsalgoritmer:

- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Nedenfor vil jeg gennemgå hver algoritme og deres kompleksitet i Big-O.

Big-O giver os en algoritmes kompleksitet i forhold til mængden af input (n). Det er en måde at se, hvor effektiv en given algoritme vil være, uafhængig af hvilken maskine den kører på, ved at se på skridtene, algoritmen tager.

Indenfor big-O er der også tids- og pladskompleksitet. Tidskompleksitet beskriver hvordan algoritmens køretid ændrer sig, afhængig af størrelsen af input (n). hvor pladskompleksitet, angiver hvor meget hukommelse en algoritme bruger i forhold størrelsen af input (n).

Når vi taler om big-O, er der 3 scenarier, best-, worst- og average-case. Når jeg i opgaven beskriver big-O, taler jeg om worst-case.

Bubble Sort

Bubble Sort er en meget simpel algoritme. Den begynder fra indeks 0 i arrayet og bevæger sig igennem listen, mens den sammenligner et element med det element, der er ved siden af. Hvis de er i ukorrekt orden, bytter den dem rundt, fortsætter til næste element og gentager processen med at se, om de to er i korrekt eller ukorrekt orden, bytter eller lader dem stå. Sådan fortsætter den igennem arrayet, indtil alle elementer står i korrekt orden.

På billedet nedenfor kan du se beregningen af worst-case notationen. Da vi har 2 for-loops, hvor i iterere n gange, og j iterere n gange, får de begge O(n), men da man fjerner konstanter i big-O notationer, ender det med $O(n^2)$ da de begge skal iterere n gange.

```
async function bubbleSort(arr) {
    const start = Date.now();
    const len = arr.length;
    let swapped;

    const arrayItems = document.querySelectorAll('.array-item');

    for (let i = 0; i < len - 1; i++) {          ➔ O(n)
        swapped = false;
        for (let j = 0; j < len - 1 - i; j++) {   ➔ O(n)
            arrayItems.forEach(item => item.classList.remove('bubbleCompare'));

            arrayItems[j].classList.add('bubbleCompare');
            arrayItems[j + 1].classList.add('bubbleCompare');

            if (arr[j] > arr[j + 1]) {
                const temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;

                await swapAnimationBubbleSort(j, j + 1);
                swapped = true;
            }
            await sleep(300);
        }
        arrayItems[len - 1 - i].classList.add('bubbleSorted');

        if (!swapped) break;
    }

    for (let i = 0; i < len; i++) {
        arrayItems[i].classList.add('bubbleSorted');
    }

    const end = Date.now();
    const elapsedSeconds = (end - start) / 1000;
    document.getElementById('bubbleSortTimer').textContent = `Time: ${elapsedSeconds.toFixed(2)}s`;
}
```

O(n) + O(n) = 2 * O(n) ~ $O(n^2)$

Insertion sort

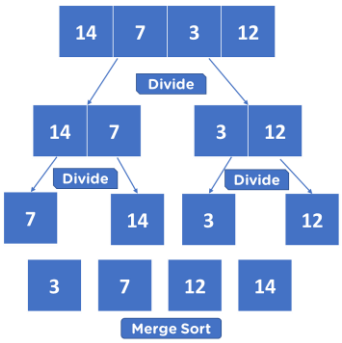
Insertion Sort fungerer ved at opdele arrayet i en sorteret og en usorteret del. Den tager elementer fra den usorterede del og indsætter dem på den korrekte plads i den sorterede del. Algoritmen starter med det andet element og sammenligner det med elementerne før det, flytter større elementer til højre og indsætter det aktuelle element på dets korrekte plads. I værste tilfælde, hvor hvert element skal sammenlignes med og flyttes forbi hvert element i den sorterende del, har den en kompleksitet på $O(n^2)$.

```
async function insertionSort(arr) {
    const startTime = performance.now();

    for (let i = 1; i < arr.length; i++) {   ➔ O(n)
        let current = arr[i];
        let j = i - 1;
        console.log("Current element:", current);

        while (j >= 0 && arr[j] > current) {   ➔ O(n)
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = current;
        console.log("Array after iteration", i, ":", arr);
    }

    const endTime = performance.now();
    const elapsedTime = endTime - startTime;

    let timeString;
    if (elapsedTime >= 1000) {
        const seconds = elapsedTime / 1000;
        timeString = `${seconds.toFixed(2)}s`;
    } else {
        timeString = `${elapsedTime.toFixed(2)}ms`;
    }

    document.getElementById('insertionSortTimer').textContent = `Time: ${timeString}`;
}
```

O(n) + O(n) = 2 * O(n) ~ $O(n^2)$

Merge sort

Merge Sort er en "divide and conquer" algoritme.



Divide – Den deler arrayet op i lige store dele, det fortsætter den med rekursivt indtil hver opdeling kun består af ét element. Den betragter herefter hver af de elementer som sorteret.

Conquer –

Den tager herefter to dele af arrayet som er individuelt sorteret og sætter dem sammen (merger dem) men denne gang sorteret i et større array. Det fortsætter den med indtil alle dele af det oprindelige array er samlet igen, men i den korrekte orden.

I mergeSort-funktionen deles arrayet op, indtil længden af hver del er 1 eller mindre. Så længe arrayet har mere end ét element, beregner den midtpunktet (middle) og bruger mergeSort igen til at opdele, indtil hvert del-array har en længde på 1 eller mindre. Når opdelingen er færdig, kalder den merge-funktionen for at kombinere og sortere delene.

```
async function mergeSort(arr) {
  if (arr.length <= 1) {
    return arr;
  }
  const middle = Math.floor(arr.length / 2);
  const left = mergeSort(arr.slice(0, middle));
  const right = mergeSort(arr.slice(middle));
  return merge(await left, await right);
}
```

Herefter opretter den et tomt array, result, som til sidst vil indeholde det fuldt sorterede array. leftIndex og rightIndex holder øje med, hvor vi er i left- og right-arrays. For at flette (merge), bruger vi en if-statement, hvor vi sammenligner elementerne i left og right. Hvis elementet i left er mindre, tilføjes det til result, og leftIndex øges. Ellers tilføjes elementet fra right, og rightIndex øges. Til sidst sammensætter den venstre og højre del ved at bruge concat (konkatinering).

```
async function merge(left, right) {
  let result = [];
  let leftIndex = 0;
  let rightIndex = 0;
  while (leftIndex < left.length && rightIndex < right.length) {
    await renderMergeState(left, right, leftIndex, rightIndex);
    if (left[leftIndex] < right[rightIndex]) {
      result.push(left[leftIndex]);
      leftIndex++;
    } else {
      result.push(right[rightIndex]);
      rightIndex++;
    }
  }
  return result.concat(left.slice(leftIndex)).concat(right.slice(rightIndex));
}
```

Til sidst sammensætter den venstre og høje ved at bruge concat (konkatinering).

Merge sort har en worst-case kompleksitet på O(n log n), hvilket gør den meget effektiv, især for store datasæt, men den kræver ekstra hukommelsesplads proportional med arrayets størrelse.

Quick sort

Quick Sort er en anden "divide and conquer" algoritme, der arbejder ved at vælge et pivot element, opdele arrayet omkring pivot elementet, og derefter rekursivt sortere de to resulterende dele. Mens det generelt er en effektiv sorteringsmetode, kan det have en dårlig ydeevne i værste tilfælde, især hvis pivot-elementet altid er enten det mindste eller største element i arrayet. Quick Sort har en worst-case kompleksitet på O(n²), hvilket sker, hvis det mindste eller største element altid vælges som pivot. Best-case får man ved at vælge pivot tilfældigt, som jeg også gør i min funktion

```
async function quickSort(
  arr,
  left = 0,
  right = arr.length - 1,
  arrayContainer,
  timerElement
) {···
}

async function partition(arr, left, right, arrayContainer, timerElement) {

  const pivotIndex = Math.floor(Math.random() * (right - left + 1)) + left;

  // Swap the pivot element with the rightmost element
  [arr[pivotIndex], arr[right]] = [arr[right], arr[pivotIndex]];

  let pivot = arr[right];
  let i = left - 1;

  for (let j = left; j < right; j++) {
    if (arr[j] <= pivot) {
      i++;
      await swapQuickSort(arr, i, j, arrayContainer);
    }
  }
  await swapQuickSort(arr, i + 1, right, arrayContainer);
  renderArray(arr, arrayContainer);
  return i + 1;
}
```

Kode-struktur

Jeg har indsat en DOM hvori mine knapper initialiseres til start af hver enkelt algoritme jeg har, der initialiserer jeg også mine arrays og timer for hvert enkelt array.

```
document.addEventListener('DOMContentLoaded', () => {
    const bubbleSortButton = document.querySelector('.bubbleSortStartButton');
    const insertionSortButton = document.querySelector('.insertionSortStartButton');
    const mergeSortButton = document.querySelector('.mergeSortStartButton');
    const quickSortButton = document.querySelector('.quickSortStartButton');
    const slowInsertionSortButton = document.querySelector('.insertionSortSlowStartButton')

    initializeArrayAndTimer('bubbleSort', generateArray());
    initializeArrayAndTimer('insertionSort', generateArray());
    initializeArrayAndTimer('insertionSortSlow', generateArray());
    initializeArrayAndTimer('mergeSort', generateArray());
    initializeArrayAndTimer('quickSort', generateArray());
```

Inde I min DOM har jeg også tilføjet alle eventListeners, de reagerer alle på click og åbner en modal hvor arrayet vises imens det sorteres og timeren også vises når algoritmen har kørt færdig.

Hver eventListener generer et array, hvor enkelte af dem også har sit eget timer-element.

```
bubbleSortButton.addEventListener("click", async () => {
  const array = generateArray();
  renderArray(array, document.querySelector("#bubbleSortArrayContainer"));

  // Show the modal
  const modal = document.getElementById("sortingModal");
  modal.style.display = "block";

  await bubbleSort(array);

  window.addEventListener("click", (event) => {
    if (event.target === modal) {
      modal.style.display = "none";
    }
  });
});

insertionSortButton.addEventListener("click", async () => {
  const array = generateArray();

  renderArray(array, document.querySelector("#insertionSortArrayContainer"));
  const modal = document.getElementById("insertionSortModal");
  modal.style.display = "block";

  const startTime = performance.now();

  await insertionSort(array);

  const endTime = performance.now();
  const elapsedTime = endTime - startTime;

  renderArray(array, document.querySelector("#insertionSortArrayContainer"));

  const timeElement = document.getElementById("insertionSortTimer");
  timeElement.textContent = `Time: ${elapsedTime.toFixed(2)}ms`;

  window.addEventListener("click", (event) => {
    if (event.target === modal) {
      modal.style.display = "none";
    }
  });
});
```

Ude af DOM'en, har jeg mine funktioner, de første funktioner er gældende for alle mine algoritmer.

I initializeArrayAndTimer, har jeg 3 parametre, det tredje, slow, er for at jeg kunne vise en langsommere version af algoritmerne imens de kører, da nogle af dem er så hurtige at en animation ikke ville kunne ses i den hastighed. Fordi arrayContainer er ens for alle algoritmer, blev den almindelige insertionSort påvirket, når jeg forsøgte med insertionSortSlow, derfor lavede jeg andre elementer til slow-versionen så det ikke havde interferens.

Jeg har valgt et fast array til alle algoritmer for at de er sammenlignelige.

```
function initializeArrayAndTimer(sortingAlgorithm, array, slow = false) {
    const arrayContainer = document.querySelector(`#${sortingAlgorithm}ArrayContainer`);
    const timerElement = document.querySelector(`#${sortingAlgorithm}Timer`);

    renderArray(array, arrayContainer);
    timerElement.textContent = 'Time: 0.00s';

    if (slow) {
        const slowArrayContainer = document.createElement('div');
        slowArrayContainer.className = 'array-container';
        slowArrayContainer.id = `${sortingAlgorithm}SlowArrayContainer`;
        document.getElementById(sortingAlgorithm).appendChild(slowArrayContainer);

        const slowTimerElement = document.createElement('div');
        slowTimerElement.className = 'timer';
        slowTimerElement.id = `${sortingAlgorithm}SlowTimer`;
        document.getElementById(sortingAlgorithm).appendChild(slowTimerElement);

        renderArray(array, slowArrayContainer);
        slowTimerElement.textContent = 'Time: 0.00s';
    }
}


function generateArray(){
    return [5, 9, 12, 7, 14, 21, 43, 35, 3];
}


function renderArray(arr, container) {
    container.innerHTML = '';
    arr.forEach(item => {
        const arrayItem = document.createElement('div');
        arrayItem.className = 'array-item';
        arrayItem.textContent = item;
        container.appendChild(arrayItem);
    });
}
```

Sleep funktionen, er for at jeg har mulighed for at sænke hastigheden imellem trinene i algoritmerne for at man kan se det visuelt.

```
async function sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}
```

Når funktionen kaldes, skabes der en promise, den vil være opfyldt (resolved) når der er gået X antal millisekunder og derefter fortsætter programmet. Inde i min funktion, for eksempelvis insertionSortSlow, har jeg kaldt sleep funktionen.

```
async function insertionSortSlow(arr) {
    const startTime = performance.now();
    const arrayContainer = document.getElementById('insertionSortSlowArrayContainer');

    for (let i = 1; i < arr.length; i++) {
        let current = arr[i];
        let j = i - 1;
        let currentIndex = i;

        arrayContainer.children[currentIndex].classList.add('current-element');


        arrayContainer.children[currentIndex].classList.add('lifted');
        await sleep(300);

        while (j >= 0 && arr[j] > current) {

            arrayContainer.children[j].style.transform = `translateX(${arrayContainer.children[currentIndex].offsetWidth}px)
            await sleep(300);

            // Swap the elements in the array
            arr[j + 1] = arr[j];

            // Move the visual representation to the left
            arrayContainer.insertBefore(arrayContainer.children[currentIndex], arrayContainer.children[j]);

            j--;


            await sleep(300);
        }
```

Efter dette kodestykke, starter alle mine funktioner for algoritmerne, bubbleSort har også en swapanimation til det visuelle.

```
async function swapAnimationBubbleSort(i, j) {
    // Get the array item elements
    const arrayItems = document.querySelectorAll('.bubble-sort-array .array-item');

    // Add class for bubble sort animation
    arrayItems[i].classList.add('bubble-swap-animation');
    arrayItems[j].classList.add('bubble-swap-animation');

    // Wait for a short duration to allow for the animation to start
    await sleep(50);

    // Calculate the distance to move the array items
    const deltaX = arrayItems[j].offsetLeft - arrayItems[i].offsetLeft;
    const deltaY = arrayItems[j].offsetTop - arrayItems[i].offsetTop;

    // Animate the movement of array items
    arrayItems[i].style.transition = 'transform 0.3s ease-in-out';
    arrayItems[j].style.transition = 'transform 0.3s ease-in-out';
    arrayItems[i].style.transform = `translate(${deltaX}px, ${deltaY}px)`;
    arrayItems[j].style.transform = `translate(-${deltaX}px, -${deltaY}px)`;

    // Wait for the animation to finish
    await sleep(300);

    // Swap the text content of array items
    [arrayItems[i].textContent, arrayItems[j].textContent] = [arrayItems[j].textContent, arrayItems[i].textContent];

    // Reset styles and remove class for bubble sort animation
    arrayItems[i].style.transition = '';
    arrayItems[j].style.transition = '';
    arrayItems[i].style.transform = '';
    arrayItems[j].style.transform = '';
    arrayItems[i].classList.remove('bubble-swap-animation');
    arrayItems[j].classList.remove('bubble-swap-animation');
}
```

```
async function bubbleSort(arr) {
    const start = Date.now();
    const len = arr.length;
    let swapped;

    const arrayItems = document.querySelectorAll('.array-item');

    for (let i = 0; i < len - 1; i++) {
        swapped = false;
        for (let j = 0; j < len - 1 - i; j++) {
            arrayItems.forEach(item => item.classList.remove('bubbleCompare'));

            arrayItems[j].classList.add('bubbleCompare');
            arrayItems[j + 1].classList.add('bubbleCompare');

            if (arr[j] > arr[j + 1]) {
                const temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;

                await swapAnimationBubbleSort(j, j + 1);
                swapped = true;
            }
            await sleep(300);
        }
        arrayItems[len - 1 - i].classList.add('bubbleSorted');

        if (!swapped) break;
    }

    for (let i = 0; i < len; i++) {
        arrayItems[i].classList.add('bubbleSorted');
    }

    const end = Date.now();
    const elapsedSeconds = (end - start) / 1000;
    document.getElementById('bubbleSortTimer').textContent = `Time: ${elapsedSeconds.toFixed(2)}s`;
}
```

Links:

https://simplecode.dk/hvad-er-en-algoritme/

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

https://www.bigocheatsheet.com/

https://www.geeksforgeeks.org/sorting-algorithms/

videoer om sorterings algoritmer:

https://www.youtube.com/playlist?list=PL9xmBV_5YoZOZSbGAXAPIq1BeUf4j20pl