Sorteringsalgoritmer teoretisk og i praksis

Af Johannes Gunge Jørgensen (S3o)
NEXT Sukkertoppen Gymnasie
April-Maj 2023

Indholdsfortegnelse

Abstract

Sorting algorithms are fundamental tools in computer science used to organize data in a specific order. But is there a difference between different algorithms' time complexity, both in practices and theoretical? In this project we implemented five different sorting algorithms in Rust to test, benchmark, and research the data. To benchmark different sorting algorithms, we use the Rust benchmarking library "Criterion" and benchmark all implemented algorithms with big and small input sizes.
The benchmarked data is analyzed to have a direct correlation to theoretical time and space complexity, but also have interesting correlations between benchmarks between two different computers with different hardware. It was found that different hardware had a big effect on benchmarks, both the number of regressed tests, but also the amount of noise in the data.

Indledning

Når store mængder af usorteret data, skal sorteres, kan det være effektivt at bruge computerkraft til sortering. Dette gøres ofte i form af en "sorterings algoritme" som kommer i mange varianter og former. Forskellen på forskellige sorteringsalgoritmer kan være stor, alt efter hvilken type, størrelse og kompleksitet det usorteret data er.
Dette projekt omhandler den teoretiske tidskompleksitet omkring forskellige sorteringsalgoritmer og hvordan algoritmernes ydeevne opholdes imod teorien. Her vil hver algoritme blive testet, samt benchmarket på deres individuelle ydeevne i forhold til forskellige størrelse af input.
Med brug af programmeringssproget Rust og Rust-biblioteket "Criterion", er det muligt at "benchmarke" forskellige sorteringsalgoritmer med forskellige størrelser af input.

Problemformulering

Hvordan opføre forskellige sorterings algoritmer i praksis i forhold til teorien?

Teori

For at kunne opstille forskellige sorterings algoritmer og analysere på dem, skal den grundlæggende teori samtidig forstås. Her er de vigtigste emner inde for sorterings algoritmer, tidskompleksitet og algoritmerne i sig selv. Tidskompleksitet er vigtig for at forstå algoritmernes ydeevne og svagheder, men samtidig skal vi forstå hvorfor nogle algoritmer er bedre end andre.

Big O-Notation

Big O-Notation (også kaldt store O-notation) er en metrik som måler ydeevnen og effektiviteten af en algoritme eller funktion, ved hjælp af tid- og rumkompleksitet. Big O repræsenterer kompleksiteten af en algoritme, ved algoritmens værste tilfælde. Der er flere forskellige notationer, som repræsenterer forskellige tilfælde i en funktion eller algoritme. Modsat O-notationen repræsenterer Omega tegnet (Ω), en algoritmes bedste tilfælde i tidskompleksitet. Følgende liste er de mest almene notationer i tid- og rumkompleksitet.

- Big O - $O(n)$: Beskriver kompleksitetens øvre grænse (Værste tilfælde)
- Theta - $\theta(n)$: Beskriver kompleksitetens nøjagtige grænse (Gennemsnitlig tilfælde)
- Omega - $\Omega(n)$: Beskriver kompleksitetens nedre grænse (Bedste tilfælde)

Den formelle definition af Big O er hvis en algoritme eller funktion $f(n)$ har den øvre grænse på $g(n)$. Her er n størrelsen af inputtet til funktionen, som kun må være en positiv heltal. Big O-notation for $f(n)$ og $g(n)$ kan skrives som:

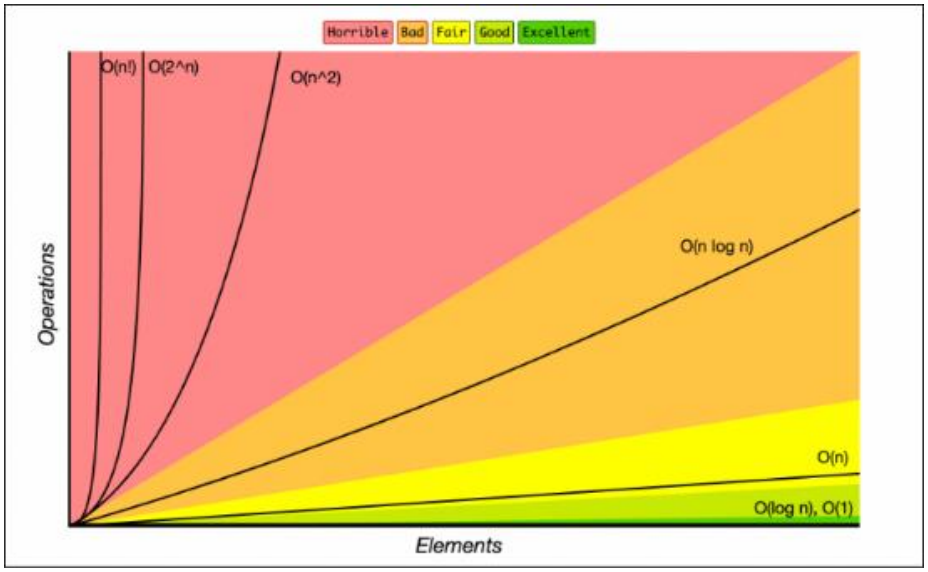
$$f(n) = O(g(n)) \text{ hvor } n \to \infty$$

Ved følgende ligning, vokser $f(n)$ lige så hurtigt som $g(n)$. Dette gør at $f(n)$ vokser lineært med størrelse af inputtet n. Dette beskrives som en af hovedtyperne af Big O, som er den lineære tid $O(n)$. Big O skrivers med brug af algebraiske termer, som primært består af i alt seks hovedtyper af kompleksitet inde for tid- og rumkompleksitet.

Big O's seks hovedtyper af kompleksiteter (Tid og Rum):

Fremragende	Godt	Fair	Dårlig	Forfærdelig/værst

Konstant tid	Logaritmisk tid	Lineær tid	Loglineær tid	Kvadratisk tid	Eksponentiel tid	Faktoriel tid
$O(1)$	$O(\log(n))$	$O(n)$	$O(n\log(n))$	$O(n^2)$	$O(2^n)$	$O(n!)$



Figur 1 Big O kompleksitetsdiagram [1]

Tids- og pladskompleksitet

Både Big O notation, tidskompleksitet og pladskompleksitet er alle relaterede til ydeevnen og ressourceforbruget af en algoritme. Men sammenlignet med Big O, er tids- og pladskompleksiteten en mere præcis måling for, hvor lang tid en algoritme eller funktion tag.

Tidskompleksiteten er et mål for, hvor meget tid det tager for en algoritme at køre, når det gives en bestemt størrelse af input. Tidskompleksiteten afhænger af antallet af operationer, som en algoritme udfører på inputtet. Tidskompleksitet giver mulighed for at estimere, hvor længe en algoritme er om at køre, på forskellige størrelser af input.

Tabel 1 Array Sorteringsalgortihmer tid- og rumkompleksitet [1]

Algoritme	Tidskompleksitet			Rumkompleksitet
	Bedste tilfælde	Gennemsnitlig tilfælde	Værste tilfælde	Værste tilfælde
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Shell Sort	$\Omega(n \log(n))$	$\theta(n \log(n)^2)$	$O(n \log(n)^2)$	$O(1)$

Sorterings algoritmer

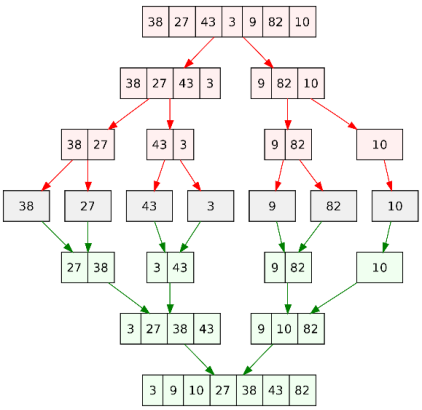
Usorteret data kan komme i mange forskellige former og størrelser. Men en af de vigtigste dele når det kommer til sortering af data, er størrelsen af data. Her er det vigtigt at vælge den rigtige algoritme både til forhold af typen af data som skal sorteres og størrelsen.

En af de mere simple sorteringsalgoritmer er Bubble sort. Bubble sort sammenligner hvert element med det næste element i rækken af data. Hvis det næste element er i forkert rækkefølge, byttes de om. Algoritmen gentages, indtil alle elementer af data er i rigtig rækkefølge. Bubble sort er dog ikke en særlig effektiv algoritme, da den har en Kvadratisk tidskompleksitet på $O(n^2)$, men en konstant rumkompleksitet $O(1)$. [2]

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            { if this pair is out of order }
            if A[i-1] > A[i] then
                { swap them and remember something changed }
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
    until not swapped
end procedure
```

Figur 2 Pseudocode af Bubble sort [3]

Merge sort er en anden form for sorteringsalgoritme, som har en bedre optimeret tidskompleksitet på $O(n \log(n))$, men en mindre optimeret rumkompleksitet på $O(n)$.
Merge sort virker på en lidt anden måde end den mere simple Bubble sort. Merge sort er en såkaldt ”Divide-and-conquer” algoritme, hvor datasættet bliver opdelt i flere små dele af et enkelt element, hvorefter elementerne bliver samlet igen sorteret.



Figur 3 Eksempel på "Divide-and'-conquer" Merge sort

Udvikling og konstruktion

Under udvikling til at teste og benchmark forskellige sorteringsalgoritmer, skal der først implementeres diverse algoritmer som kan undersøges. Her blev flere forskellige algoritmer implementeret, mange med forskellige tids- og rumkompleksiteter. Dette er gjort for at se om der er forskel i praksis mellem forskellige algoritmer, som

både har samme og forskellige tids- og rumkompleksitet. De implementeret sorteringsalgoritmer er Bubble, Insertion, Merge, Quick og Shell sort.

Derefter skal algoritmerne testes for om de virker og kan sortere et simpelt array. Dette er relativt simpelt, da mange programmeringssprog, inklusive Rust, har et standart testnings bibliotek.

Når alle sorteringsalgoritmer er testet og klar, skal de alle benchmarkes ud fra flere forskellige størrelser af input. Dette bliver gjort med brug af Rust biblioteket ”Criterion”, som gør det nemt og effektivt at benchmarke med flere forskellige størrelser af input. Samtidig giver Criterion meget detaljeret information af hver benchmarkede algoritme, som gør det nemt at analysere med baggrund af projektets problemstilling.

For at opnå algoritmer som kan sortere flere forskellige arrays med forskellige typer af data, er det effektivt at bruge ”Generics” (også kaldt generiske typer). Generics bliver brugt til at generalisere datatyper af argumenter i funktioner, så i stedet for at angive en specifik datatype for et argument, kan det generaliseres. [4] Generics bliver defineret for en funktion med brug af syntaks $< T >$. Her er T en arbitrær datatype, som kan bruges af argumenter for funktionen. F.eks. kan en generisk funktion, med navn ”algoritme”, som tag argument T som arbitrær type:

```
fn algoritme<T>(arg: T) {…}
```

Generics bliver også taget i brug, ved alle implementeret sorteringsalgoritmer i projektet. Her er alle generiske typer i hver sorteringsalgoritme, en del af ”Traiten” - ”total order”. Traiten er en gruppe af metoder som bliver påført den generiske type, som gør at for alle værdier i den generiske type kan sammenlignes, med brug af operationer som $<$ (større eller mindre end), $==$ (lig med). Dette betyder at både strings (tekststykker / enkle karakterer) og integers (heltal) kan sorteres i de implementeret algoritmer.

```
mod bubble_sort;
fn main(){
    let mut strArr = ["a","c","b","q","e"];
    let mut intArr = [8,5,9,2,7];
    // Bubble sort
    bubble_sort::bubblesort(&mut strArr); // sort the array
    println!("String - Bubble sort: {:?}", strArr); // OUTPUT: ["a", "b", "c", "e", "q"]
    bubble_sort::bubblesort(&mut intArr); // sort the array
    println!("Integer - Quick sort: {:?}", intArr); // OUTPUT: [2, 5, 7, 8, 9]
}
```

Figur 4 Bubble sort med to forskellige typer af array inputs.

Dokumentation af program

Med brug af generics og diverse test og benchmarking funktioner klar, kan vi f.eks. Bubble sort implementeres på følgende måde:

```rust
// Generic function that accepts any type that implements the Ord trait
pub fn bubblesort<T: Ord>(arr: &mut [T]) {
    let len = arr.len();
    for i in 0..len {
        for j in 0..len - i - 1 {
            if arr[j] > arr[j + 1] { // swap if the current value is greater than the next value
                arr.swap(j, j + 1);
            }
        }
    }
}
```

Figur 5 Implantation af Bubble sort i Rust

Funktionen bruger den generiske type T som bruger traiten ”Ord” (total order), som tag et ”mutable” array af type T. En mutabel datatype, betyder at datatypen ikke er konstant og kan ændres. [5] Dette er vigtigt at gøre argumentet mutable, hvis vi skal ændret og returnere det samme array.

```rust
pub fn bubblesort<T: Ord>(arr: &mut [T])
```

Derefter bliver længden af arrayet gemt, så det er muligt at itererer over arrayet med et indlejret *for* loop. Derefter bliver et element sammenlignet med det næste element i arrayet, for at se om elementerne er i forkert orden. Hvis dette er tilfældet, bliver de to elementer byttet om. Algoritmen itererer gennem hele arrayet indtil hvert element er på sin rigtige plads.

```rust
let len = arr.len(); // get the length of the array
for i in 0..len { // iterate over the array
  for j in 0..len - i - 1 {
    if arr[j] > arr[j + 1] { // swap if the current value is greater than the next value
          arr.swap(j, j + 1);
    }
  }
}
```

Test & Benchmarking

For at teste hvorledes de forskellige algoritmer fungere og ikke sortere et datasæt forkert, gøres der brug af Rust testnings bibliotek. At opstille en test for en funktion, er relativ simpel. F.eks. hvis Bubble sortalgoritmen skulle testes, skal der først defineres et simpelt testinput, som er usorteret og som vi selv nemt kan sortere.

```rust
let mut arr = [6, 2, 4, 1, 9, -2, 5];
```

Derefter skal arrayet køres igennem sorteringsalgoritmen og tjekkes hvorledes om arrayet er blevet sorteret korrekt.

```rust
    bubblesort(&mut arr);
    assert_eq!(arr, [-2, 1, 2, 4, 5, 6, 9]);
```

Her bliver *arr* testet hvorledes om outputtet fra funktionen er lig med det velvidende sorteret array, som der sammenlignes med. Sammenligningen sker med brug af Rust egen sammenligningsfunktion *assert_eq!*.
Ved tilfældet af *arr,* skal det sorteret array komme ud som $[-2, 1, 2, 4, 5, 6, 9]$.
Sådan en test af bubble sort kan se således ud:

```
#[test]
 fn test_bubble_sort() {
    let mut arr = [6, 2, 4, 1, 9, -2, 5]; // unsorted array
    bubblesort(&mut arr); // sorts array
    assert_eq!(arr, [-2, 1, 2, 4, 5, 6, 9]); // Test if new array is equal to known sorted array.
 }
```

Figur 6 Test af Bubble sort, med brug af Rust eget testnings bibliotek.

Med brug af Rust biblioteket Criterion, kan alle sorteringsalgoritmerne nu blive benchmarket. Det er vigtigt at benchmark alle funktionerne med forskellige størrelse af input. Dette er vigtigt da det skal bruges til at drage konklusioner ud fra de forskellige benchmark tests. Det skal kunne være muligt at sammenligne sorteringsalgoritmerne med hinanden, så at indhente gode data fra Criterion benchmarks er vigtigt.
Først laves en benchmark gruppe, som gemmer alle benchmarkes fra funktionerne.

```
let mut group = c.benchmark_group("Sorting Algorithms");
```

Derefter skal der vælges diverse kriterier, som sorteringsalgoritmerne skal benchmarkes med. Dette kan selvfølgelig være størrelsen af inputtet, men også hvor stor intervallet af hvert element af inputtet skal være. F.eks. hvorledes en sorteringsalgoritme skal benchmarkes mod et array af 100 elementer som kan være af heltal fra 0 til 100, eller mod et array af 50 elementer med heltal fra 0 til 1000.
Kriterierne jeg har valgt, er at hver sorteringsalgoritme skal benchmarkes 16 gange, hvert med forskellige størrelse array, med alle arraystørrelser skal være af heltal med intervallet 0 til 100. Størrelserne af arrayene vil være $\{ 1, 5, 10,\ 25,\ 50, 75, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 \}$. I kode ville dette se således ud:

```
for i in [1,5,10, 25,50,75, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000] { // size of array

        let mut rng = rand::thread_rng();
        let range = Uniform::new(0, 100); // range of random numbers
…
```

Når kriterierne er valgt, kan vi begynde at benchmark funktionerne. Her vil Criterion gøre alt arbejdet for os, med at itererer gennem hvert array med forskellige størrelser og udfylde arrayet med forskellige elementer fra intervallet 0 til 100. Derefter sorteres hvert array af den valgte sorteringsalgoritme og dette bliver gentaget op til 100 gange, for indsamle et gennemsnit af dataen.
F.eks. vil en benchmark funktion af Bubble sort se således ud:

```
group.bench_function(BenchmarkId::new("Bubble Sort", i), |b| {
            // creates a vector of random numbers
            b.iter_batched_ref(|| -> Vec<usize> { (0..i).map(|_| rng.sample(&range)).collect()},
            |v| bubblesort( v), // v is the vector/array to be sorted
            BatchSize::SmallInput, // The number of times the function is run (Small: 100 times)
           )
 });
```

Outputtet af hver benchmark er et gennemsnit af den tid det tog for algoritmen at sortere arrayet, ud fra størrelsen af inputtet. Outputtet som Criterion giver, viser samtidig detekteret usædvanlige benchmark prøver, hvor tiden på prøven er usædvanlig høj eller lav.

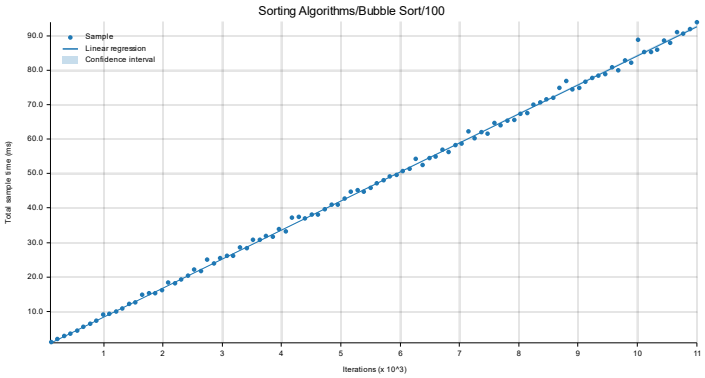
```
Sorting Algorithms/Bubble Sort/100
    time:   [8.1996 µs 8.2539 µs 8.3318 µs]
Found 10 outliers among 100 measurements (10.00%)
  1 (1.00%) high mild
  9 (9.00%) high severe
```

Diskussion

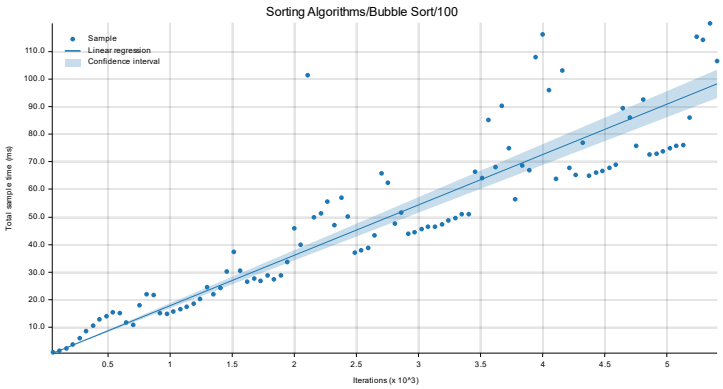
At samle, analysere og sammenligne data fra sorteringsalgoritmer kan være kompliceret. Der er mange faktorer som kan gøre at den indsamlet data er upålidelig, støjet eller på anden måde knap så brugbar. Når indsamlingen af data skal foregå, er det ikke mindst vigtig at det samme datasæt og samme parameter bliver brugt i alle sorteringsalgoritmer under benchmarking. Men også at hardware er tilstrækkelig kraftig nok, til sorteringsalgoritmerne med større input størrelser.

Opdagelse af usædvanlige benchmark prøver

Der kan være mange forskellige grunde til støj i en benchmark prøve, som gør at tiden på prøven var usædvanlig høj eller lav. Dette kan være forårsaget af uforudsigelig belastning på computeren som har fortaget benchmarkende. Det kan være alt fra tråd- eller procesplanlægning eller uregelmæssigheder i den tid, det tager for koden som benchmarkes. [6]
Dette kan specielt ses med en sammenligning af samme benchmark test, men med to forskellige computere, med forskellig hardware. Nedstående bilag fremviser et regressionsplot, hvor hvert datapunkt er en benchmark prøve for Bubble sort, med en inputstørrelse på 100 elementer. Her viser plottet antallet af iterationer versus tiden det tog for sorteringsalgoritmen. Ved et godt benchmark er regressionsligningen lineær, uden mange spredte datapunkter. Hvis mange datapunkter vildt spredte, indikerer det, at der er meget støj i dataene og at benchmark prøven måske ikke er pålidelig. [6]



Figur 7 Bubble sort (100 elementer) regressionsplot - stationær



Figur 8 Bubble sort (100 elementer) regressionsplot - Laptop

Computer Specifikationer
CPU: Intel(R) Core i9-9900K@ 3.60GHz
RAM: 32,0 GB RAM
CPU: NVIDIA GeForce RTX 2080

Plotdata

Hældning	8,43 μs
R^2	0,964
Median	8,40 μs

Computer Specifikationer
CPU: Intel(R) Core i7 @ 1.30GHz
RAM: 16,0 GB RAM
CPU: Intel(R) Iris(R) Plus (10th gen)

Plotdata

Hældning	18,167 μs
R^2	0,155
Median	17,099 μs

Her er det tydeligt at se hvordan computerspecifikationerne drastisk kan ændre datasættet for samme benchmark. Den mere kraftfulde stationær computer er ikke mindst hurtigere til hver iteration, men

samtidig giver et mere pålideligt datasæt med tætter datapunkter og tæt regresionslinje. Den stationære computer sortere dataene gennemsnitlig 103.5% hurtigere, end laptop computeren, samt har en 84% bedre korrelationskoefficient.

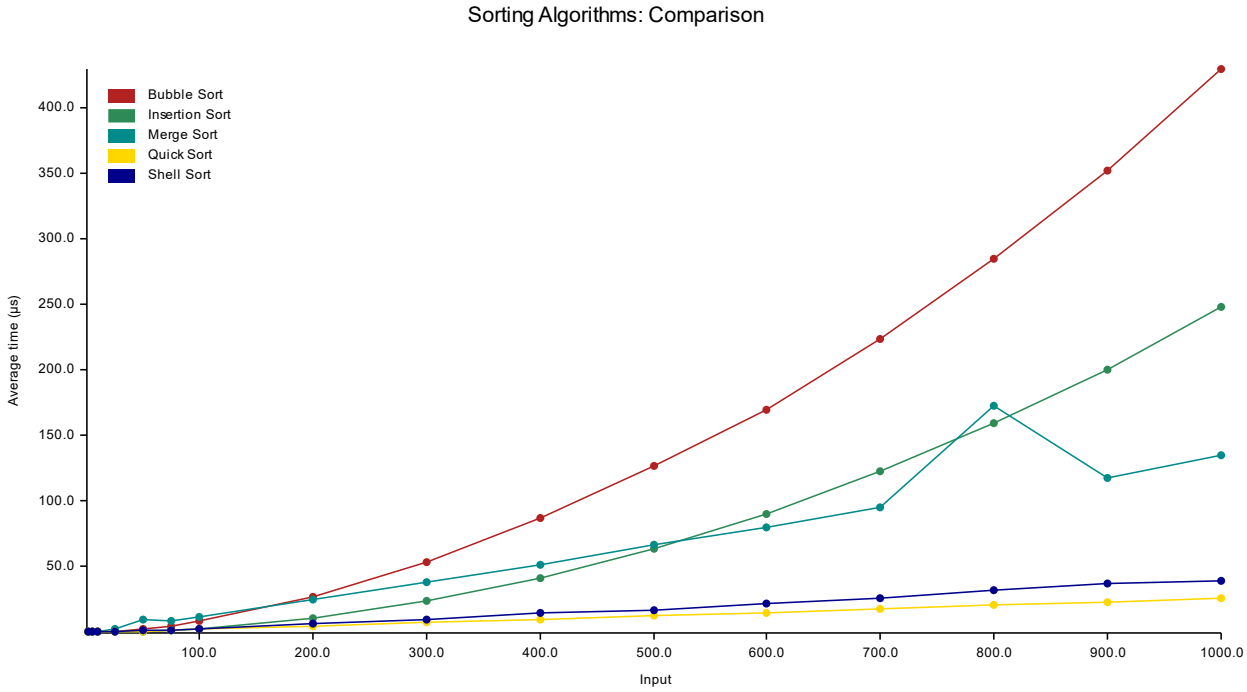
Sammenligning af tidkompleksitet

Efter benchmarking, indsamling og analyse af alle datapunkter fra alle implementeret sorteringsalgoritmer, er dette det endelige resultat for gennemsnits tiden i forhold til inputstørrelsen. BEMÆRK at følgende benchmark blev konstrueret med brug af den stationære computer med følende computer specifikationer

CPU: Intel(R) Core i9-9900K@ 3.60GHz

RAM: 32,0 GB RAM

CPU: NVIDIA GeForce RTX 2080



Figur 9 Gennemsnitligt tid mellem alle implementeret algoritme i forhold til inputstørrelse.

Her kan ses et tydeligt forhold mellem tidskompleksiteten og de individuelle sorteringsalgoritmer. Bubble sortalgoritmen er nok den som falder en i øjnene først. Bubble sort som har en tidskompleksitet på $O(n\text{^}2)$, ses også som at have en eksponentiel hældning gennem hele benchmarkingen. Samme tidskompleksitet har Insertion sort, men Insertion sort er mere optimeret i forhold til Bubble sort, da Insertion sort laver færre sammenligninger. [7] Merge og Quick sort som begge har tidskompleksitet $O(n\log(n))$, som også tydeligt kan ses at have en logaritmisk udvikling i figur 7. Men Merge sort har en bedre tidskompleksitet end Insertion sort, men Insertion sort er bedre i de lave inputstørrelser? Dette er på grund af Insertion sort har bedre og mere stabil rumkompleksitet end Merge sort. Da rumkompleksitet bliver mere og mere relevant i de større inputstørrelser, er Insertion sort hurtigere i de små inputstørrelser. [8] En lille bemærkelse er også at Merge sort ”regressed” (blev værre i forhold til tidligere benchmarks), som gjorde at Merge sort tog et uventet spring ved inputstørrelse 800. Der kan være flere grunde til dette forekommer, men det er mest sandsynlig at være en fejlkilde fra computerens side.

Konklusion

Det kan konkluderes at praktiske implementeringer af diverse sorteringsalgoritmer, vil opføre sig i takt med den teoretiske tids- og rumkompleksitet. Det kan samtidigt konkluderes at computer hardware har stor indflydelse på pålideligheden og kvaliteten af benchmarkings data fra alle sorteringsalgoritmer. Her blev det fundet at hardware med lav ydeevne har betydelig langsommere sorterings tid for alle sorteringsalgoritmer, samt skaber flere upålidelige datapunkter.

Referencer

[1] FreeCodeCamp, »Big O Cheat Sheet – Time Complexity Chart,« 5 maj 2023. [Online]. Available: https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/.

[2] productplan, »Bubble Sort,« 5 maj 2023. [Online]. Available: https://www.productplan.com/glossary/bubble-sort/.

[3] wikipedia, »Bubble sort,« 5 maj 2023. [Online]. Available: https://en.wikipedia.org/wiki/Bubble_sort.

[4] Rust-documentation, »Generics,« 5 maj 2023. [Online]. Available: https://doc.rust-lang.org/rust-by-example/generics.html.

[5] Rust-documentation, »Variables and Mutability,« 5 maj 2023. [Online]. Available: https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html.

[6] Rust-documentation, »Command-Line Output -,« 5 maj 2023. [Online]. Available: https://bheisler.github.io/criterion.rs/book/user_guide/command_line_output.html.

[7] Interview kickstart, »Comparison Among Selection Sort, Bubble Sort, and Insertion Sort,« 5 maj 2023. [Online]. Available: https://www.interviewkickstart.com/learn/comparison-among-bubble-sort-selection-sort-and-insertion-sort.

[8] geeksforgeeks, »Merge Sort vs. Insertion Sort,« 5 maj 2023. [Online]. Available: https://www.geeksforgeeks.org/merge-sort-vs-insertion-sort/.

Bilag

```
fn sorting_benchmarks(c: &mut Criterion) {
let mut group = c.benchmark_group("Sorting Algorithms");

for i in [1,5,10, 25,50,75, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]{ // size of array

let mut rng = rand::thread_rng();
let range = Uniform::new(0, 100); // range of random numbers

// Bubble sort benchmark (Time complexity: O(n^2))
group.bench_function(BenchmarkId::new("Bubble Sort", i), |b| {
       // creates a vector of random numbers
       b.iter_batched_ref(|| -> Vec<usize> { (0..i).map(|_| rng.sample(&range)).collect() },
       |v| bubblesort( v), // v is the vector/array to be sorted
       BatchSize::SmallInput, // BatchSize::SmallInput is the number of times the function is run
     )
});
…
```

Figur 10 Mindre afsnit af benchmark gruppe

```
pub fn insertionsort<T: Ord>(array: &mut [T]) {
    let mut n = array.len();
    for i in 1..n {
        let mut j = i;
// move elements of array[0..i-1], that are greater than key, to one position ahead of their
current position
        while j > 0 && array[j - 1] > array[j] {
            array.swap(j - 1, j);
            j -= 1;
        }
    }
}
```

Figur 11 Insertion sort implementering

```
pub fn shellsort<T: Ord + Copy>(arr: &mut [T]) {
    let len = arr.len();
    let mut gap = len / 2;
    while gap > 0 {
        for i in gap..len {
            let temp = arr[i];
            let mut j = i;
            while j >= gap && arr[j - gap] > temp {
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = temp;
        }
        gap /= 2;
    }
}
```

Figur 12 Shell sort implementering

```rust
pub fn quicksort<T: Ord>(array: &mut [T]) { // Takes mut array of type 'T'.
    let low: isize = 0; // Index '0' of array
    let high: isize = (array.len() - 1) as isize; // Highest index of array

    sort(array, low, high);

}
fn sort<T: Ord>(arr: &mut [T], low: isize, high: isize){
    if low < high {
        let pivot_element: isize = temp(arr, low, high); // pivot element (always the element of the highest index, in the whole or sub array)
        sort(arr, low, pivot_element - 1); // Before pi
        sort(arr, pivot_element + 1, high); // After pi
    }
    fn temp<T: Ord>(arr: &mut [T], low: isize, high: isize) -> isize {
    let pivot = high as usize; // Pivot element
    let mut i = low - 1;
    let mut j = high;
    loop {

            i += 1;
            while arr[i as usize] < arr[pivot] {
                i += 1;
            }
            j -= 1;
            while j >= 0 && arr[j as usize] > arr[pivot] {
                j -= 1;
            }
            if i >= j { // If current element is smaller than pivot element.
                break;
            } else {
                arr.swap(i as usize, j as usize);
            }
        }
        arr.swap(i as usize, pivot as usize);
        i
    }
}
```

Figur 13 Quick sort implementering

```rust
pub fn mergesort<T: Ord + Clone>(arr: &mut [T]) {
    sort(arr, 0, arr.len() - 1);
}
fn sort<T: Ord + Clone>(arr: &mut [T], left: usize, right: usize) {
    if left >= right { // recursive
        return;
    }
    let mid: usize = (left + right) / 2; // Middle index of array
    sort(arr, left, mid);
    sort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

fn merge<T: Ord + Clone>(arr: &mut [T], left: usize, mid: usize, right: usize) {
    let n1 = mid - left + 1;
    let n2 = right - mid;

    let left_arr: Vec<T> = (0..n1).into_iter().map(|i| arr[left+i].clone() ).collect();
    let right_arr: Vec<T> = (0..n2).into_iter().map(|i| arr[mid + 1 + i].clone() ).collect();

    let mut i: usize = 0;
    let mut j: usize = 0;
    let mut k: usize = left;

    while i < n1 && j < n2 {
        if left_arr[i] <= right_arr[j] {
            arr[k] = left_arr[i].clone();
            i += 1;
        } else {

             arr[k] = right_arr[j].clone();
            j += 1;
        }
        k += 1;
    }

    while i < n1 {
        arr[k] = left_arr[i].clone();
        i += 1;
        k += 1;
    }

    while j < n2 {
        arr[k] = right_arr[j].clone();
        j += 1;
        k += 1;
    }
}
```

Figur 14 Merge sort implementering