2D Projekt 2022

Algoritmer

Mergesort

Merge sort er en meget effektiv sorterings algoritme der fungere ved at en af liste af elementer op til de alle står alene, og derefter sammeligner den hosliggende lister og samler dem i den korrekte rækkefølge. Når alle listerne med de enkelte elementer i nu er i lister med 2, sker det samme igen hvor hver liste bliver sammenlignt med dens hosliggende liste indtil hele listen er samlet og sorteret. Den sammenligner 2 allerede sorterede lister og kan derfor let uden at tjekke alt igennem finde ud af hvordan de 2 lister skal sættes sammen.

```
    public static ArrayList<Integer> sort(ArrayList<Integer>
list)
    {
        if (list.size() <= 1)
        {
            return list;
        }

        ArrayList<Integer> leftList = new
ArrayList<Integer>();
        ArrayList<Integer> rightList = new
ArrayList<Integer>();

        for (int i = 0; i < list.size(); i++)
        {
            if (i < list.size() / 2)
            {
                leftList.add(list.get(i));
            }
            else
            {
                rightList.add(list.get(i));
            }

        }
            leftList = sort(leftList);
            rightList = sort(rightList);

            return merge(leftList, rightList);
    }

    public static ArrayList<Integer> merge(ArrayList<Integer>
left, ArrayList<Integer> right)
    {
        ArrayList<Integer> result = new ArrayList<Integer>();
```

```
        while (left.size() != 0 && right.size() != 0)
        {
            if (left.get(0) < right.get(0))
            {
                result.add(left.get(0));
                left.remove(0);
            }
            else
            {
                result.add(right.get(0));
                right.remove(0);
            }
        }
        while (left.size() != 0)
        {
            result.add(left.get(0));
            left.remove(0);
        }

        while (right.size() != 0)
        {
            result.add(right.get(0));
            right.remove(0);
        }
        return result;
    }
```

Kompleksiteten af algoritmen kan deles op i 2 dele, først opdelingen af elementerne og derefter samlingen

Opdelingen er antal gange tallet n kan deles i 2 hvilket også kan beskrives som $\log_2(n) + 1$, og da algoritmen er meget effektiv skal den kunde tjekke værdien af hvert element 1 gang, derfor er samlingen n og tiol sammen er de $n(\log_2(n) + 1) = n \cdot \log_2(n) \cdot n$, når man findet worst case kompleksiteten tager man kun det led der udvikler sig værst og det er $O(n \cdot \log_2(n))$

Quicksort

Quicksort minder meget om mergesort i det at den opdeler listen og samlerne den bag efter. Forskellen er hvor elementerne bliver sammelignet, for mergesort sker det når elementerne sættes sammen, mens det for quick sort sker når listerne bliver delt op. Listen bliver delt op i en højre del, venstre del og en pivot, som er det tal der bliver sammenling lige nu. Den går så igenne alle tal og sætter dem enten til højre eller venstre for pivot tallet alt efter om de er mindre eller større.

```
public static ArrayList<Integer> sort(ArrayList<Integer> list)
    {
        return sort(list, 0, list.size() - 1);
```

```
    }

    public static ArrayList<Integer> sort(ArrayList<Integer>
list, int lo, int hi)
    {

        if (hi >=0 && lo >= 0 && hi > lo)
        {
            int p = partition(list, lo, hi);
            sort(list, lo, p);
            sort(list, p + 1, hi);
        }
        return list;
    }

    public static int partition(ArrayList<Integer> list, int
lo, int hi)
    {
        int pivot = (lo + hi) / 2;

        int l = lo;
        int r = hi;

        while (true)
        {
            while (list.get(l) < list.get(pivot))
                l++;

            while (list.get(r) > list.get(pivot))
                r--;

            if (l >= r)
                return r;

            Integer temp = list.get(l);
            list.set(l, list.get(r));
            list.set(r, temp);
            l++;
            r--;
        }
    }
```

Typisk vil kompleksiteten ende med at være $O(n \cdot \log_2(n))$ men, hvis alle opdelingerne ender med at ligge forkert i for hold til tallene ved siden af, skal alle tallene ses igennem ved opdelingen. Opdelingen har derfor et worst case som n mens samlingen ligesom ved mergesort også har en kompleksitet som n og de 2 sammelagt er derfor $O(n^2)$ i worst case.

Insertionsort

Insertionsort virker ved at lave en ny liste ved siden af den der skal sorteres og starter med at tilføje første element fra listen til den nye liste, den går så igennem alle elementer i den nye liste for hvert element i listen og tilføjer den på den korrekte plads.

```
public static ArrayList<Integer> sort(ArrayList<Integer> list)
    {
        ArrayList<Integer> sortedList = new
ArrayList<Integer>();
        for (int i = 0; i < list.size(); i++)
        {
            if (i == 0)
            {
                sortedList.add(list.get(0));
                continue;
            }

            for (int j = sortedList.size() - 1; j >= 0; j--)
            {
                if (list.get(i) > sortedList.get(j))
                {
                    sortedList.add(j + 1, list.get(i));
                    break;
                }
                else if (j == 0)
                {
                    sortedList.add(0, list.get(i));
                    break;
                }
                else
                {
                    continue;
                }
            }
        }
        return sortedList;
    }
```

Worst case kompleksiteten af insertionsort er $O(n^2)$ da den i værste tilfælde skal tjekke alle sortede elementer igennem for hvert eneste element så $n \cdot n$

Selectionsort

Selectionsort er meget simpel og virker cirka omvendt af hvordan insertion sort virker. Den ser hele listen igennem der skal sorteres og finder det mindste tal og rykker det til en anden liste ved siden af. Så i stedet for at tage et tal og se en liste igennem for at finde dens plads, finder den det mindste tal og rykker den for enden af en anden liste.

```
public static ArrayList<Integer> sort(ArrayList<Integer> list)
    {
        for (int i = 0; i < list.size(); i++)
        {
            int min = i;
            for (int j = i + 1; j < list.size(); j++)
            {
                if (list.get(j) < list.get(min))
                {
                    min = j;
                }
            }
            if (min != i)
            {
                int temp = list.get(i);
                list.set(i, list.get(min));
                list.set(min, temp);
            }
        }
        return list;
    }
```

Igen ligesom insertionsort har selectionsort en worst case som $O(n^2)$ da den skal se alle tal igennem n gange. Den bliver nød til at se hele listen igennem hver gang for at være sikker på hvad det mindste tal er

Sekventiel søgning

Sekventiel søgning virker ved at starte fra en ende af og gå igennem alle tal i listen indtil tallet der søges efter er fundet. Algoritmen er meget simpel og kan derfor godt tage meget lang tid.

```
    public static int search(ArrayList<Integer> list, int n)
    {
        for (int i = 0; i < list.size(); i++)
        {

            if (list.get(i) == n)
                return i;
        }
        return -1;
    }
```

Kompleksiteten af algoritmen er $O(n)$ da den i worst case kommer til at tjekke alle tallene 1 gang og derfor n gange

Binær søgning

Binær søgning virker ved at tage det midterste tal og se om tallet er mindre eller højere end det tal vi leder efter, den tager så midten af venstre eller højre del af arrayet alt efter om tallet vi ledte efter var større eller mindre og fortsætter sådan indtil den finder tallet eller bekræfter at den ikke er i

listen. Det er dog kun muligt for algoritmen at virke fordi listen er sorteret og den vil ikke virke i en usorteret liste.

```
public static int search(ArrayList<Integer> list, int n)
    {
        int l = 0;
        int r = list.size() - 1;
        while (l <= r)
        {
            int m = (l + r) / 2;
            if (list.get(m) < n)
            {
                l = m + 1;
            }
            else if (list.get(m) > n)
            {
                r = m - 1;
            }
            else
            {
                return m;
            }
        }
        return -1;
    }
```

Ligesom for splitningen i mergesort og i quicksort vil binær søgning i worst case skulle dele en liste af n elementer så mange gange som muligt og kompleksiteten er derfor $O(\log_2(n))$ uden n ganget på da den ikke skal merge.

Tidstagning

I mit program tager jeg tid for sortering af listerne ved at tage den nuværende tid i nanosekunder før og efter sorteringen er kørt og gemmer forskellen. Hver algoritme bliver så kørt 1000 gange og gennemsnittet af tiden gemmes for hver algoritme fro forskellige lister størrelser gor både tilfældige lister, sorteret stigende lister og sorteret aftagende lister. Computeren kan være igang med en masse forskellige ting og vi får derfor et mere præcist tal hvis tidstagningen bliver tagen flere gange.

Seed: 7 Iterations: 1000

Random

MergeSort with 10 elements: 0,011112

QuickSort with 10 elements: 0,002611

SelectionSort with 10 elements: 0,002766

InsertionSort with 10 elements: 0,004312

MergeSort with 50 elements: 0,042231

QuickSort with 50 elements: 0,00802

SelectionSort with 50 elements: 0,018936

InsertionSort with 50 elements: 0,016653

MergeSort with 200 elements: 0,141559

QuickSort with 200 elements: 0,004134

SelectionSort with 200 elements: 0,036645

InsertionSort with 200 elements: 0,019917

MergeSort with 500 elements: 0,312607

QuickSort with 500 elements: 0,008797

SelectionSort with 500 elements: 0,197886

InsertionSort with 500 elements: 0,101604

MergeSort with 1000 elements: 0,609074

QuickSort with 1000 elements: 0,019491

SelectionSort with 1000 elements: 1,322448

InsertionSort with 1000 elements: 0,458958

MergeSort with 2000 elements: 1,529181

QuickSort with 2000 elements: 0,037188

SelectionSort with 2000 elements: 3,164129

InsertionSort with 2000 elements: 1,222396

Increasing

MergeSort with 10 elements: 0,00113

QuickSort with 10 elements: 0,000142

SelectionSort with 10 elements: 0,000138

InsertionSort with 10 elements: 0,000133

MergeSort with 50 elements: 0,010419

QuickSort with 50 elements: 0,000682

SelectionSort with 50 elements: 0,002097

InsertionSort with 50 elements: 0,00062

MergeSort with 200 elements: 0,064056

QuickSort with 200 elements: 0,003273

SelectionSort with 200 elements: 0,031508

InsertionSort with 200 elements: 0,002084

MergeSort with 500 elements: 0,208557

QuickSort with 500 elements: 0,00805

SelectionSort with 500 elements: 0,184581

InsertionSort with 500 elements: 0,005212

MergeSort with 1000 elements: 0,480309

QuickSort with 1000 elements: 0,018657

SelectionSort with 1000 elements: 0,769709

InsertionSort with 1000 elements: 0,009494

MergeSort with 2000 elements: 1,093794

QuickSort with 2000 elements: 0,035448

SelectionSort with 2000 elements: 2,896444

InsertionSort with 2000 elements: 0,020512

Decreasing

MergeSort with 10 elements: 0,001115

QuickSort with 10 elements: 0,000135

SelectionSort with 10 elements: 0,000134

InsertionSort with 10 elements: 0,000398

MergeSort with 50 elements: 0,011214

QuickSort with 50 elements: 0,000738

SelectionSort with 50 elements: 0,002195

InsertionSort with 50 elements: 0,003077

MergeSort with 200 elements: 0,065316

QuickSort with 200 elements: 0,003134

SelectionSort with 200 elements: 0,02984

InsertionSort with 200 elements: 0,028127

MergeSort with 500 elements: 0,203374

QuickSort with 500 elements: 0,008026

SelectionSort with 500 elements: 0,181542

InsertionSort with 500 elements: 0,148947

MergeSort with 1000 elements: 0,461942

QuickSort with 1000 elements: 0,018938

SelectionSort with 1000 elements: 0,700898

InsertionSort with 1000 elements: 0,542841

MergeSort with 2000 elements: 1,083204

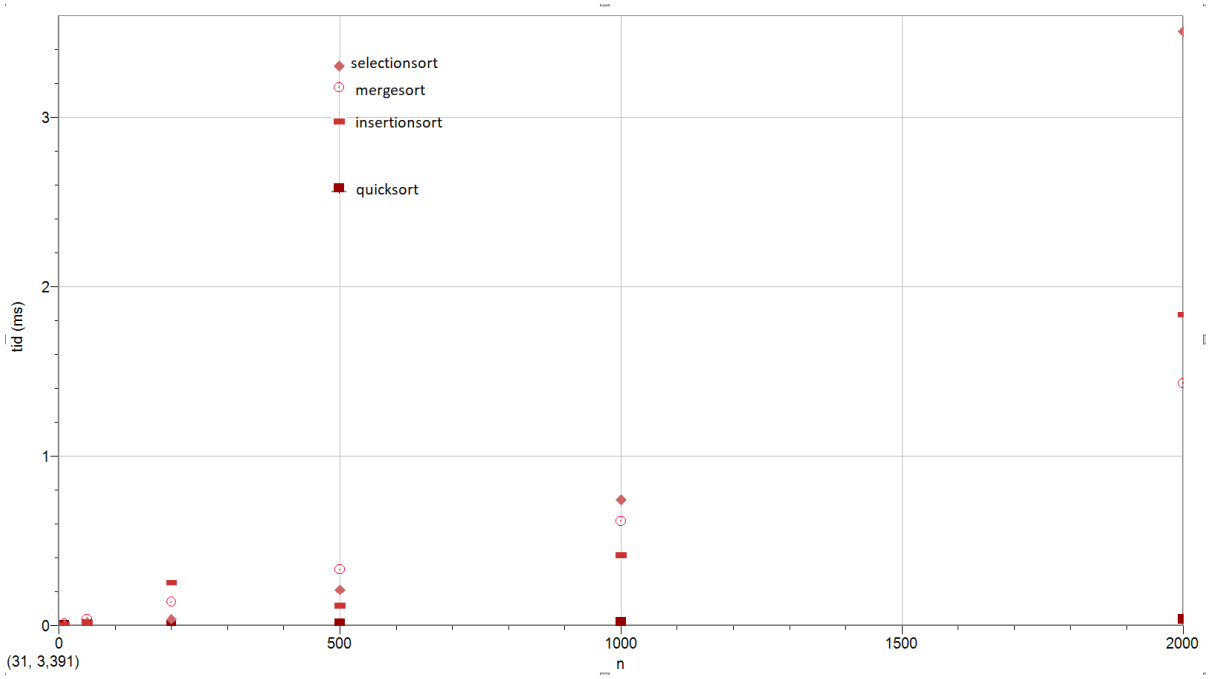
QuickSort with 2000 elements: 0,034465

SelectionSort with 2000 elements: 2,831799

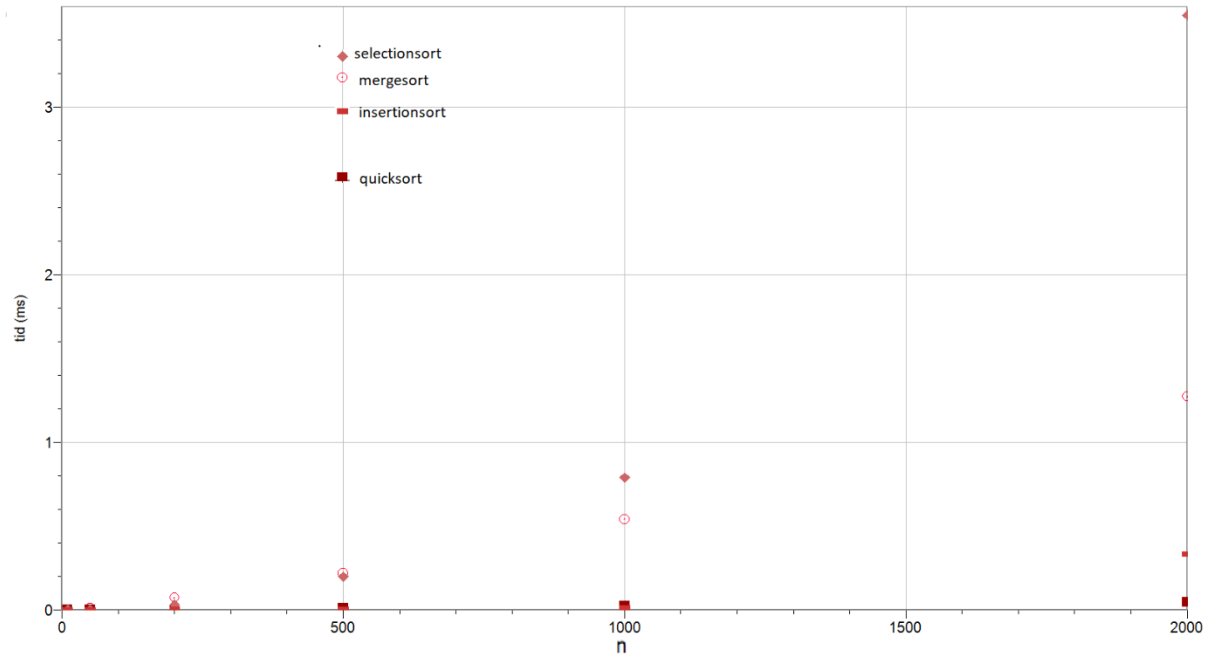
InsertionSort with 2000 elements: 2,132677

LinearSearch in sorted list with 2000: 0,010729

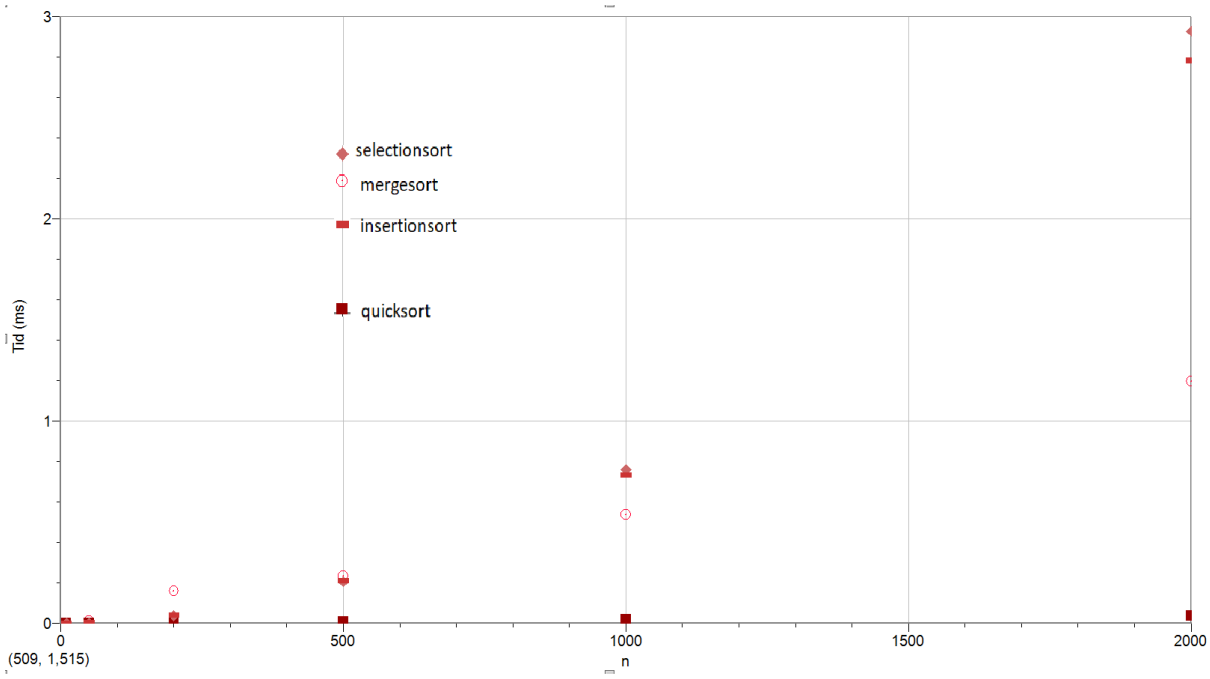
BinarySearch in sorted list with 2000: 0,001524



Sorterings tid for lister med tilfældige tal af den 4 forskellige sorterings algoritmer. I små antal elementer er de cirka lige gode men i større lister er selections sort klart den værste med mergesort og insertionsort i midten og quicksort hurtigst. Da mergesort har bedre worst case end alle de andre algoritmer regner jeg med at den ville være bedre i større lister end selv quicksort, mens insertion og selectionsort ville være endnu længere fra quicksort og mergesort.



For allerede sorterede lister har selectionsort samme resultat da den altid går gennem alle tal og altid har $O(n^2)$, den store forskel er i insertionsort der har best case i allerede sortede lister og derfor er hurtigere end mergesort i det her tilfælde, mens quicksort stadig er den hurtigste.



Til sidst er sortering af lister sorteret i aftagende orden. Igen er quicksort merkant hurtigere end alt andet mens merge sort er forholdsvist hurtig i forhold til selectionsort. Nu er det bare worstcase for insertionsort som nu i stedet for at være omkring mergesort nu er helt oppe og er ligeså langsom som selectionsort.

```
LinearSearch in sorted list with 2000: 0,010729
BinarySearch in sorted list with 2000: 0,001524
```

I sorterede lister, hvor tallet der ledes efter ikke er helt ude i enden, men et sted i midten vil binarysearch være meget hurtigere, da den udnytter at listen er sorteret mens sekventiel søgning skal igennem alt. det gode ved sekventiel søgning er dog så at den kan bruges til lister der ikke er sorterede.

Tidskonstanten

$C(n)$ kan bestemmes med formlen $T(n) = C(n) \cdot O$, hvor $T(n)$ er tiden og $O(n)$ er kompleksiteten. Det gøres så for hver liste med tilfældige tal og og gennemsnittet tages for at bestemme en enkel konstant for hver algoritme

Mergesort:

$$T(10) = C(10) \cdot O(10 \cdot \log_2(10)) \Downarrow$$
$$0{,}00919\ ms = C(10) \cdot 10 \cdot \log_2(10) \Downarrow$$
$$C(10) = \frac{0{,}00919\ ms}{10 \cdot \log_2(10)} \approx 0{,}00027665 \cdot ms$$

$$C(50) = \frac{0{,}03791\ ms}{50 \cdot \log_2(50)} \approx 0{,}00013434 \cdot ms$$
$$C(200) = \frac{0{,}139962\ ms}{200 \cdot \log_2(200)} \approx 9{,}1552 \cdot 10^{-5} \cdot ms$$
$$C(500) = \frac{0{,}210439\ ms}{500 \cdot \log_2(500)} \approx 4{,}6943 \cdot 10^{-5} \cdot ms$$
$$C(1000) = \frac{0{,}617187\ ms}{1000 \cdot \log_2(1000)} \approx 6{,}1931 \cdot 10^{-5} \cdot ms$$
$$C(2000) = \frac{1{,}429632\ ms}{2000 \cdot \log_2(2000)} \approx 6{,}5186 \cdot 10^{-5} \cdot ms$$
$$C = \frac{0{,}00027665 + 0{,}00013434 + 9{,}1552 \cdot 10^{-5} + 4{,}6943 \cdot 10^{-5} + 6{,}1931 \cdot 10^{-5} + 6{,}5186 \cdot 10^{-5}}{6}$$
$$\approx 0{,}00011277$$

Den samme metode genbruges for de andre algoritmer for deres korrosponderende kompleksiteter og tider.

Quicksort:

$$C = 0{,}0000049$$

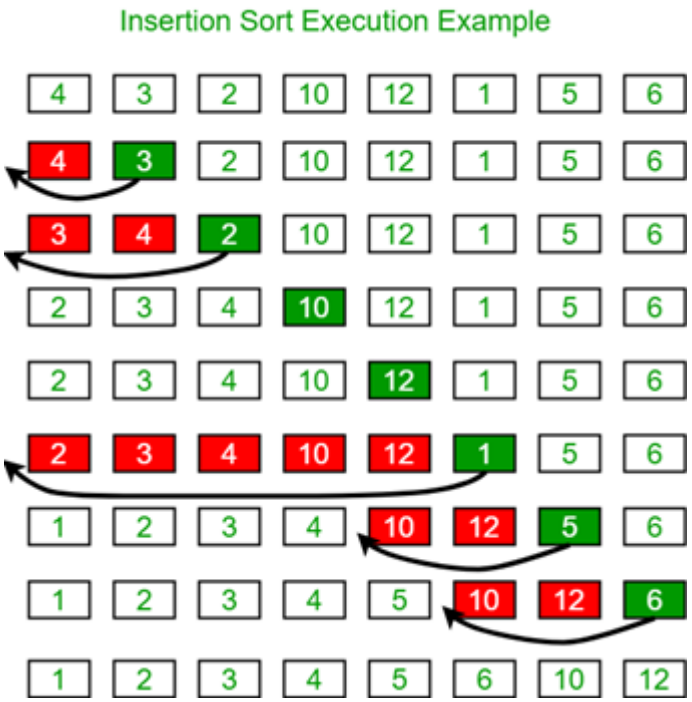
Selectionsort:

$$C = 0{,}0000065$$

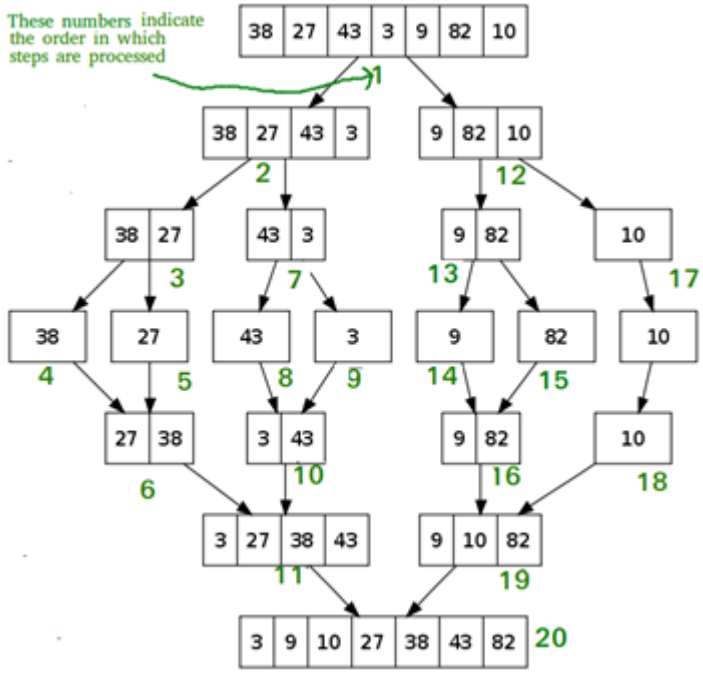
Insertionsort

$$C = 0{,}0000086$$

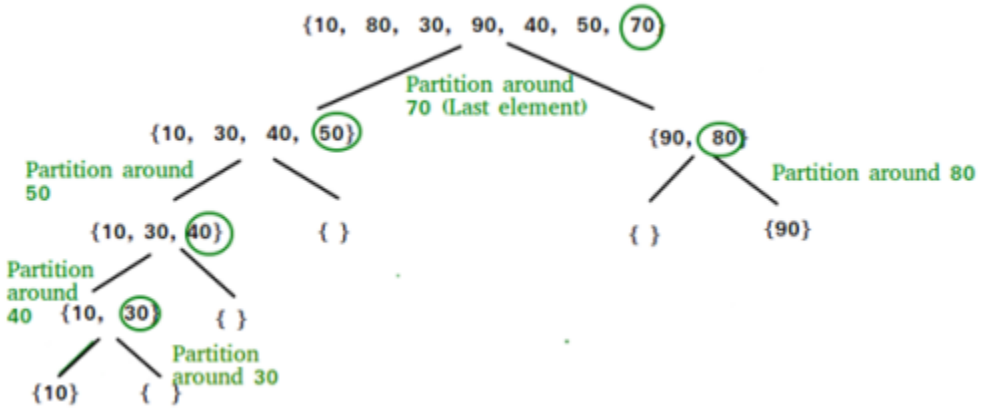
Dokumentation



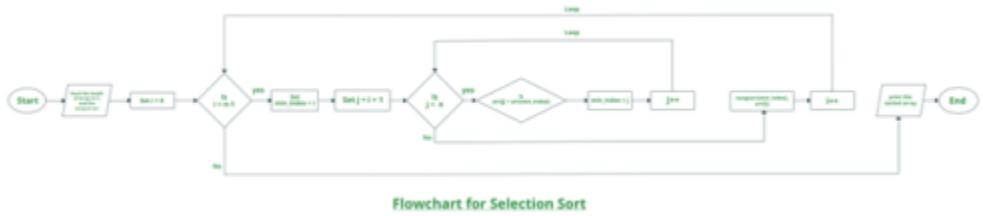
(https://www.geeksforgeeks.org/insertion-sort/)



(https://www.geeksforgeeks.org/merge-sort/)

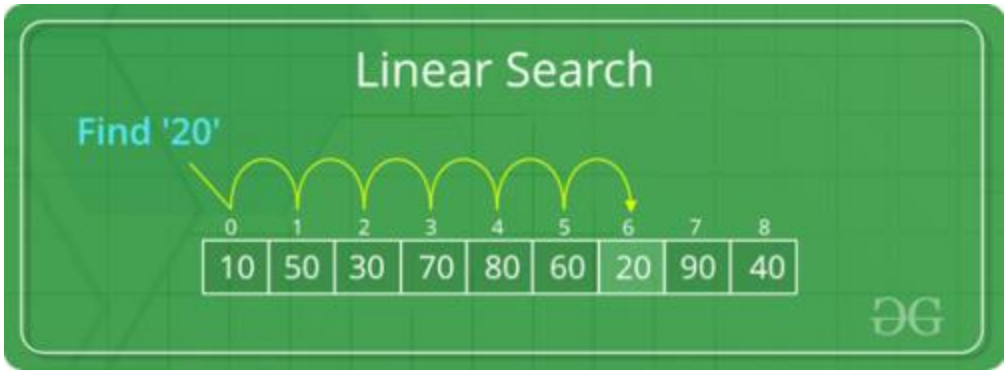


(https://www.geeksforgeeks.org/quick-sort/)

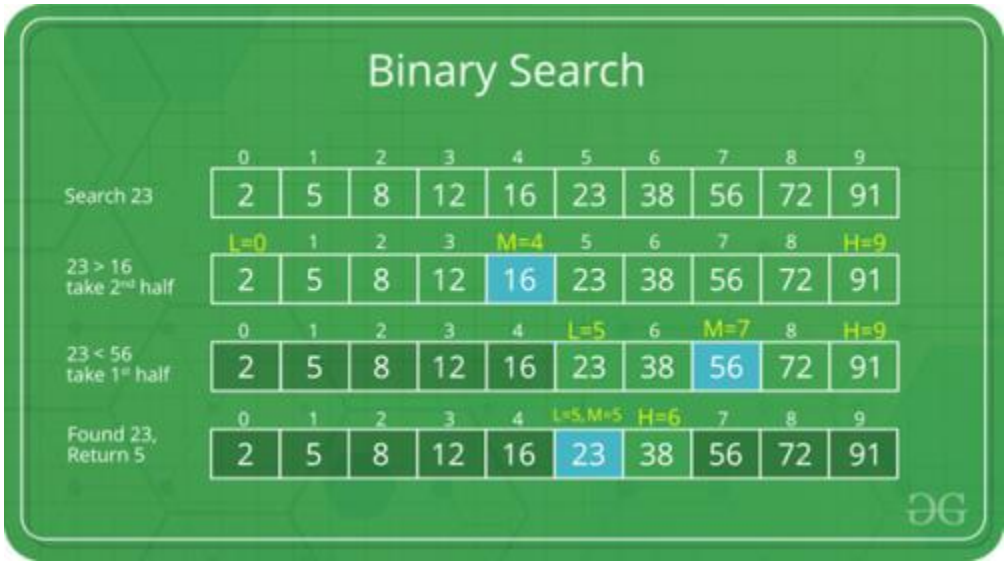


Flowchart for Selection Sort

(https://www.geeksforgeeks.org/selection-sort/)



(https://www.geeksforgeeks.org/linear-search/)



(https://www.geeksforgeeks.org/binary-search/)

Konklusioner

Den bedste teoretiske sorteringsalgoritme er mergesort, da den har en langt bedre worst case kompleksitet end alle de andre algoritmer. Men quicksort vil i mindre lister være hurtigere end mergesort.

Den bedste algoritme på min maskine for de størrelser lister vi har brugt er quicksort, den er tydeligt meget hurtigere end alle de andre sorterings algoritmer, både for lister med tilfældige tal, en liste sorteret stigende og en liste sorteret aftagende. Quicksort har også den laveste C konstant af alle sorterings algoritmer.

Alt andet der kører på computeren har betydning for tidstagningen og har betydning for hvor lang tid det tager. Det betyder også at 2 ens ting kan tage forskeligt tid.

Der er nogle ikke komparative sorteringsalgoritmer som ikke virker med flydende tal, f.eks MSD radix sort og Pifeonhole sort. De fleste andre sorterings algoritmer virker fint med flydende tal, og at man kan bruge dem giver noget mere frihed til hvis det er flydende tal man bruger.