4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se

Laboration: Analys av sorteringslgoritmer

Metod

För att undersöka prestandan hos olika sorteringsalgoritmer implementerades 4 olika sorteringsalgoritmer: InsertionSort, SelectionSort, QuickSort pivot i höger element och Quicksort pivot enligt median-of-three. Dessa algoritmer jämfördes sedan med std::sort från C++-standardbiblioteket.

Det kommer genomföras tidmätningar på dessa sorteringsalgoritmer med fyra olika typer av dataserier: slumpmässiga data, monotont stigande data, monotont fallande data och konstant värde-data. Dataseriernas storlek varieras för att ge god tidmätning. Tiden för datagenerering inkluderas inte i tidmätningarna.

Varje mätserie för varje typ av data bestod av fem tidmätningar. Denna mängd av mätningar valdes för att balansera noggrannheten i mätningarna med den tid det tar för en programkörning.

Resultatet från programkörningarna lagras i en textfil i csv-format. Dessa data används sedan för att generera grafer som illustrerar tidmätningarna för varje sorteringsalgoritm och dataserie.

Python

Koden skriven i Python i projektet är utformad för att analysera och visualisera data som samlats in från C++-programmets prestandamätningar. Koden läser in mätdata från en CSV-fil, behandlar data och genererar en rad grafer för att visualisera algoritmens prestanda under olika förhållanden.

Översikt av vad koden gör

1. Koden börjar med att läsa in data från CSV-filen som C++-programmet genererat till en pandas DataFrame, en tvådimensionell datastruktur som är lätt att manipulera och analysera.
2. Koden manipulerar sedan data genom att konvertera vissa kolumner till lämpliga datatyper och gruppera data för senare analys.
3. Slutligen genererar koden ett antal grafer för att visualisera data, inklusive linjediagram och heatmaps. Dessa grafer sparades som PNG-filer.

För att utföra dessa uppgifter används flera bibliotek från Python.
Pandas: Detta bibliotek används för att lagra och manipulera data. Det har DataFrame-strukturen som används för att lagra data från CSV-filen samt funktioner för att manipulera och analysera dessa data.
Matplotlib: ett bibliotek för att skapa linjediagram och värmekartor.
Seaborn: visualiseringsbibliotek byggt på matplotlib, används för att skapa heatmaps.
OS och datetime: används för att skapa kataloger baserade på nuvarande datum och tid för att lagra grafer.

4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se
Numpy: används för att utföra matematiska funktioner

Programkod

Main.cpp är kärnan i programmet. Det skapar en lista över sorteringslgoritmer och en lista med olika datamönster. Sedan går det igenom varje kombination av sorteringsalgoritmer och datamönster, utför sorteringen och mäter den tid detta tar. Resultatet lagras i en SortingResult-struktur och läggs i en lista över alla resultat.

För varje sorteringsalgoritm och datamönsters kombination utförs sorteringen ett antal gånger som definieras av variabeln sampleCount. Resultatet av dessa körningar används för att beräkna ett genomsnittligt värde och en standardavvikelse för tiden det tar att utföra sorteringen.

Efter att sorteringar har utförts, skriver programmet ut resultatet till en CSV-fil.

Data_reporting.cpp/h innehåller koden för att rapportera resultaten av tidmätningarna.

- saveSortingResults skriver resultaten till en CSV-fil, där varje rad i filen representerar en sortering, med sorteringalgoritmens namn, datamängdens storlek, genomsnittliga tiden och standardavvikelsen, antal prov och datamönster.
- SortingResult är en struktur som används för att lagra resultaten av en sortering. Den innehåller fält för algoritmens namn, datamönster, datamängdens storlek, genomsnittlig tid, standardavvikelse och antalet prov.

Timing_utils.cpp/h innehåller funktioner för att mäta tid och beräkna standardavvikelse.

- measureSortingTime tar en sorteringsfunktion och en vektor av data och sorterar datan med den angivna algoritmen och returnerar tiden det tog.
- calculateStandardDeviation tar en vektor av prov och returnerar standardavvikelsen för dessa prov.

Data_generation.cpp/h skapar olika typer av datamönster.

- GenerateRandomData skapar en vektor av slumpmässiga heltal. Varje element i vektorn är mellan 1 och storleken på vektorn.
- GenerateAscendingOrderData skapar en vektor av heltal i stigande ordning. Varje element i vektorn är like med sin position
- GenerateDescendingOrderData skapar en vektor av heltal i fallande ordning. Detta genomförs genom att generera en vektor i GenerateAscendinOrderData som sedan vänds på med std::reverse.
- GenerateConstantValueData skapar en vektor där alla element har samma värde, som är storleken på vektorn.

Sorting_algorithms.cpp/h innehåller implementationer av olika sorteringsalgoritmer.

- InsertionSort: använder en iterativ process för att sortera en vektor.
- SelectionSort: Går igenom vektorn och väljer det minsta elementet från den osorterade delen och byter plats med det aktuella elementet.

4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se

- QuickSort: Använder en av partitionsfunktionerna för att utföra en snabb sortering av data.
- partitionRight: är en funktion som används i QuickSort för att välja det högra elementet som pivot.
- partitionMedianOfThree: används av QuickSort där pivot väljs som medianen av tre värden (låg, hög och mitten).

Algoritmerna

InsertionSort:

Tänk dig att du har spelkort i handen som du precis dragit och nu ska du sortera dem. Du tar ett kort åt gången från vänster till höger och börjar på det andra kortet och sätter det på rätt plats i förhållande till det första kortet. Sedan tar du det tredje kortet och sätter det på rätt plats i förhållande till de första två, tills alla kort är sorterade.

```
void insertionSort(data_t &vector){
    int nextPosition = 1;
    while(nextPosition < vector.size()){
        int valueToInsert = vector[nextPosition];
        int sortedPosition = nextPosition - 1;
        while (sortedPosition >= 0 && vector[sortedPosition] > valueToInsert){
            vector[sortedPosition + 1] = vector[sortedPosition];
            sortedPosition = sortedPosition - 1;
        }
        vector[sortedPosition + 1] = valueToInsert;
        nextPosition++;
    }
}
```

SelectionSort:

Tänk dig att du har spelkort i handen som du precis dragit och nu ska sortera dem. Titta på din hand från vänster till höger och leta reda på den lägsta kortet, detta kort är det minsta värdet i den osorterade handen. Ta sedan det minsta kortet och byt plats på det med det första kortet i handen (om det inte redan är det första kortet). Nu är det lägsta kortet först i handen. Hoppa nu över det första kortet och upprepa processen med resten av korten i handen, titta på de återstående osorterade korten och välj det lägsta som du byter plats med första osorterade kortet.

4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se

```
void selectionSort(data_t &vector){
    for (int currentPosition = 0; currentPosition < vector.size() - 1; ++currentPosition) {
        int minValuePosition = currentPosition;

        for (int nextPosition = currentPosition + 1; nextPosition < vector.size(); nextPosition++) {
            if (vector[nextPosition] < vector[minValuePosition]) {
                minValuePosition = nextPosition;
            }
        }

        if (minValuePosition != currentPosition){
            std::swap( &: vector[minValuePosition],  &: vector[currentPosition]);
        }
    }
}
```

QuickSort:

välj ett kort i handen som är ”pivot”. Detta kort kommer användas för att dela upp handen i två delar: kort som är mindre än pivot och kort som är större än pivot. Omorganisera handen så att alla kort som är mindre än pivot kommer före pivot och alla kort som är större kommer efter. Detta är att partitionera handen. Använd samma process på varje del av handen, både den som är mindre och den som är större. Fortsätt dela upp och sortera tills varje del endast innehåller ett kort.

```
void quick_sort(data_t& data, int low, int high, PartitionFunc partition){
    if (low < high){
        int partitionIndex = partition( &: data, low, high);
        quick_sort( &: data, low,  high: partitionIndex - 1, partition);
        quick_sort( &: data,  low: partitionIndex + 1, high, partition);
    }
}
```

Right pivot:

För en pivot till höger väljes alltid det sista kortet i den aktuella delen av handen som pivot.

```
int partition_right(data_t& data, int low, int high){
    int pivot = data[high];
    int index = (low - 1);

    for (int current = low; current <= high ; ++current) {
        if (data[current] < pivot){
            index++;
            std::swap( &: data[index],  &: data[current]);
        }
    }
    std::swap( &: data[index+1],  &: data[high]);
    return (index + 1);
}
```

4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se

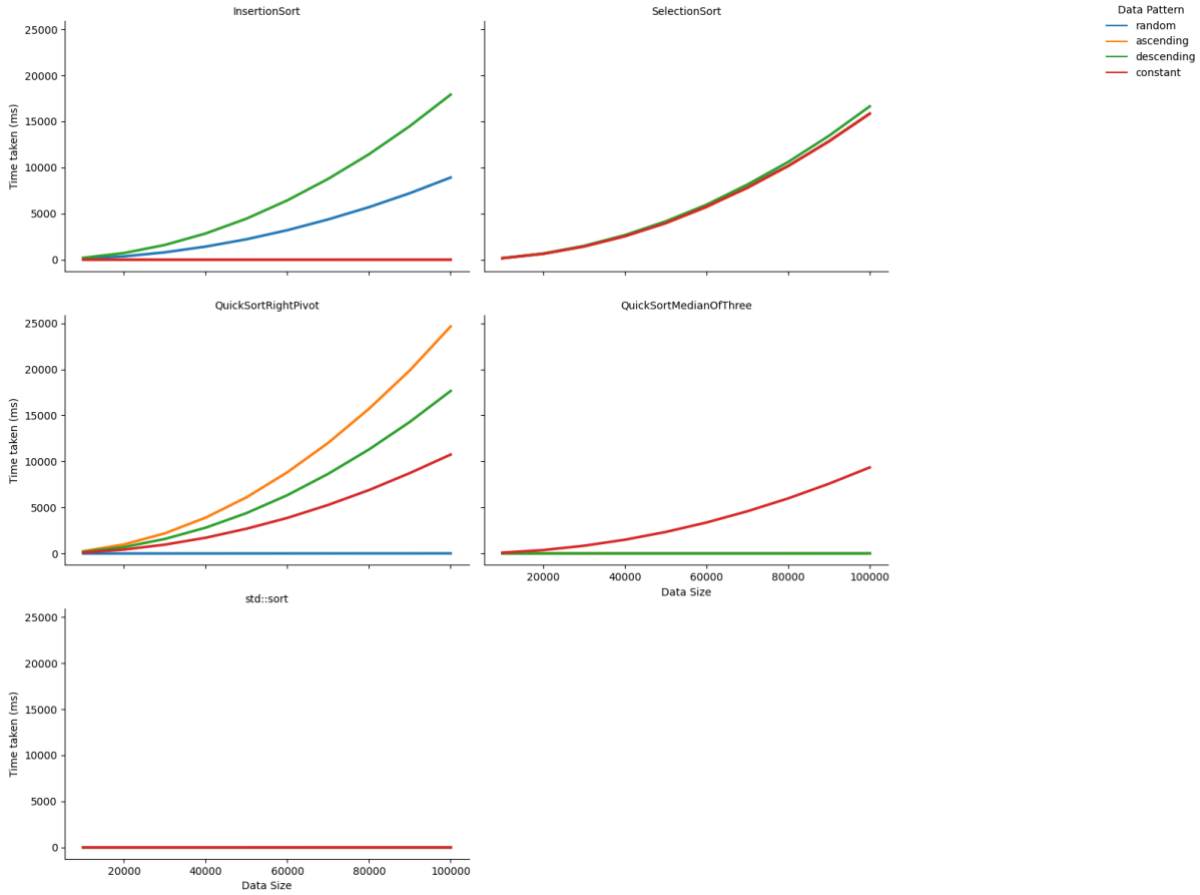
Median of three:

För ett median av tre väljs tre kort från handen, i detta fall första, mellersta och sista i den aktuella delen och använder medianen av dessa tre som pivot.

```
int partition_median_of_three(data_t& data, int low, int high){
    int mid = low + (high-low)/2;
    if (data[low] > data[high]){ std::swap( &: data[low],  &: data[high]);}
    if (data[mid] > data[high]){ std::swap( &: data[mid],  &: data[high]);}
    if (data[mid] > data[low]) { std::swap( &: data[mid],  &: data[low]);}

    // Use the pivot value as a sentinel value to help with partitioning
    int pivot = data[low];
    // Index for maintaining the partition of the array
    int index = low+1;
    for (int i = low+1; i < high ; i++) {
        if (data[i] < pivot){
            std::swap( &: data[index],  &: data[i]);
            index++;
        }
    }

    // Place the pivot at the partition index
    std::swap( &: data[low],  &: data[index-1]);

    return index-1;
}
```

4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se

Resultat och Grafer



Analysen av de olika sorteringsalgoritmerna visade skillnader i prestanda beroende på datainmatning och storlek.

InsertionSort

För slumpmässig data visade det sig att tiden ökar kvadratiskt med storleken på datamängden, O(n^2). Detta beror på att varje element måste jämföras med varje tidigare element, vilket resulterar i ett kvadratiskt antal jämförelser. För sorterad och konstanta värden behöver algoritmen bara jämföra varje element en gång vilket gör den till O(n) i hastighet. Slumpmässiga tal hamnar således någonstans mitt emellan O(n) och O(n^2) i tidskomplexitet. Mina resultat genererade från programmet stämmer alltså väl överens med vad man kan förvänta sig enligt Wikipedia. [1]

SelectionSort

Selectionsort visade sig ha O(n^2) i tidskomplexitet för alla typer av data, inklusive slumpmässiga, stigande, fallande och konstanta värden. Detta beror på att algoritmen alltid går igenom hela den osorterade listan för att hitta det minsta elementet, oavsett hur datan är ordnad. Varje nytt element måste jämföras med varje annat osorterat element, vilket alltså resulterar i ett kvadratiskt antal jämförelser. Mina resultat genererade från programmet stämmer alltså väl överens med vad man kan förvänta sig enligt Wikipedia. [2]

4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se

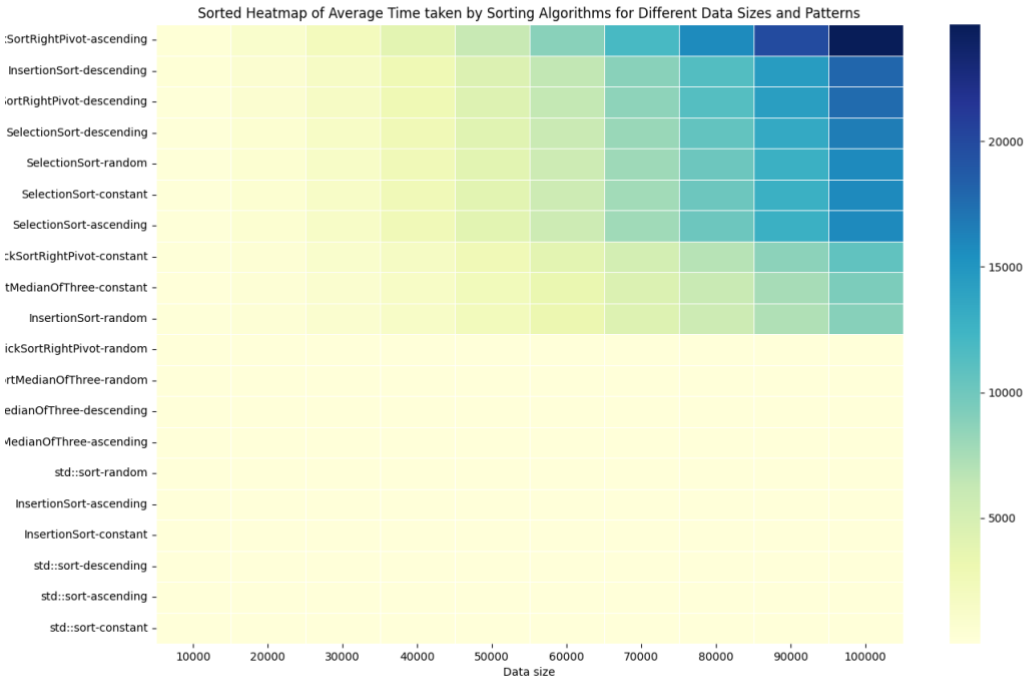
QuickSort:

Med **Right Pivot** tenderar QuickSort att ha en tidskomplexitet på O(n log n) vid slumpmässiga data. Detta beror på att man kan förvänta sig att delningarna inte alltid kommer vara helt obalanserade. Detta betyder att varje del av delningen kommer vara någorlunda lika stora och på så vis hamna i närheten av logaritmisk höjd på rekursionsträdet. För fallande, stigande och konstanta värden rör sig tidskomplexiteten mot O(n^2) och tittar vi närmre på varför stigande element blir O(n^2) så är det för att varje delning kommer resultera i en partition med n-1 element och en partition med 0 element. Detta gör rekursionsträdet mycket obalanserat eftersom höjden på trädet också blir n och resterande rekursionssteg blir en jämförelse av alla återstående element. Det är alltså särskilt sannolikt att O(n^2) inträffar när man väljer första eller sista elementet som pivot för listor som redan är sorterade.
Med **median of three**-algoritm får man en mer robust konstruktion för att hantera stigande och fallande värden också. Detta är för att algoritmen oftare kan uppnå balanserade delningar, även om den redan är sorterad i en viss ordning. Detta minimerar alltså risken att hamna i ett worst case scenario och håller tidskomplexiteten närmre O(n log n). Mina resultat stämmer även från Quick Sort in på det wikipedia beskriver under ”Formal analysis”. [3]

Std::sort

Std::sort är en hybridalgoritm som kombinerar flera sorteringsmetoder för att uppnå optimal prestanda för olika typer av data och storlekar. Std::sort börjar alltså med quicksort och byter algoritm till HeapSort om rekursionsdjupet blir för stort för att undvika O(n^2) i prestanda i worst case. Genom att nyttja att QuickSort kan vara snabbare i praktiken än HeapSort för många datamängder har algoritmen HeapSort som backup som garanterar prestandan när QuickSort rör sig mot sitt worst case, och därför har algoritmen alltid O(n log

4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se
n) i tidskomplexitet. [4]



Diskussion/slutsats

I denna laboration utfördes en detaljerad analys av olika sorteringsalgoritmer för att förstå hur de presterar under olika förhållanden. De experimentella resultaten överensstämmer med de teoretiska förväntningarna jag studerat på internet, men har också gett mig insikter som inte kan uppnås genom endast teoretisk analys.

QuickSort-algoritmen visade sig vara känslig för valet av pivot. Median of three-strategin bidrog till att förbättra robustheten mot ordnade data och minimera risken för worst case.

Std::sort demonstrerade fördelarna med en hybridalgoritm genom att kombinera styrkorna hos både QuickSort och Heapsort. Denna kombination resulterar alltså i en algoritm som alltid har O(n log n) i tidskomplexitet!

Programkod

https://github.com/anga95/doa_labb2

4 augusti 2023
VT22 DT046G/DT064G Datateknik GR (B)
Andreas Gabrielsson
Anga2001@student.miun.se

Källor

Bibliography

[1] Wikipedia, "Insertion Sort," [Online]. Available: https://en.wikipedia.org/wiki/Insertion_sort. [Accessed 16 08 2023].

[2] Wikipedia, "Selection Sort," [Online]. Available: https://en.wikipedia.org/wiki/Selection_sort. [Accessed 16 08 2023].

[3] Wikipedia, "Quicksort," [Online]. Available: https://en.wikipedia.org/wiki/Quicksort. [Accessed 16 08 2023].

[4] Wikipedia, "sort(c++)," [Online]. Available: https://en.wikipedia.org/wiki/Sort_(C%2B%2B). [Accessed 16 08 2023].