Fist2000@student.miun.se Filip Stenegren

Laboration: Analys av sorteringsalgoritmer

Introduktion

Laborationens syfte är att implementera, undersöka och diskutera prestandan hos fem olika sorteringsalgoritmer. Algoritmerna är Insertion Sort, QuickSort Median of 3, QuickSort, SelectionSort och std::sort (en standardsortering för C++). Sorteringsalgoritmerna utsätts för fyra olika sorters lister, monotont stigande/ fallande, slumpdata och konstant värde.

Metod

För att kunna undersöka prestandan implementerades varje sorteringsalgoritm, metoder för att generera data, rensa och skriva till filer och mäta sorteringstiden. Sorteringsalgoritmerna kommer att förklaras senare i rapporten. Datatypernas storlek (N) är mellan 2000 till 20000 element. För att mäta sorteringstiden utsattes varje sorteringsalgoritm för varje sort av datatyp. Storleken på datatypen varierades för att garantera jämn och icke partiska mätresultat[1]. Resultatet av programmet är att resultaten av tidmätningarna sparas i textfiler. Denna data används senare för att rita ut grafer för att visualisera och därmed kunna undersöka och diskutera enklare.

[1] Tiden att generera datatypen är inte inkluderad i tidsmätningarna.

Fist2000@student.miun.se Filip Stenegren

Programkod C++

FileManager.cpp/h

De här filerna innehåller koden för att manipulera filer.

ClearFiles tar en vector som innehåller filnamn som inparameter och rensar filerna från dess innehåll men skriver också ut en titel som är baserad på filnamnet minus 4 tecken. Så givet filnamnet ”Quicksort.txt” tar den bort de sista 4 tecken för att skriva Quicksort som titel i filen.
därefter skriver den också ut rubriker för datamängdens storlek (N), tiden i millisekunder (t [ms]), standardavvikelsen i millisekunder (Stdev [ms]) och hur många antal prov som kördes (Samples). När std::ofstream kallas tar den inga argument för hur filen ska öppnas så jag antar att den per automatik öppnar filen i *truncate* läget.

WriteFile tar en string filnamn och output som inparameter och öppnar filen i append för att skriva i slutet av filen. Den skriver outputen och stänger sedan strömmen till filen. Det krävs att output är skapat för att följa ordningen med kolumnerna som **ClearFiles** skapade.

ListGeneration.cpp/h

Innehåller olika funktioner för att skapa vectorer med olika sorters datamönster. Alla funktioner tar samma argument *int storlek* som inparameter. Parametern definierar antal element i som vectorna innehåller.

RandomVec skapar en vector med slumpmässiga heltal mellan 1 och 10 000.

RisingVec skapar en vector med heltal från 0 till vectorns storlek - 1

FallingVec denna funktion skapar en vector med heltal som startar med storleken på vectorn och minskar tills sista elementet är 1.

SameVec genererar en vector där varje element är heltalet 13.

Sorts.cpp/h

Innehåller funktionerna för sorteringsalgoritmerna.

SelectionSort väljer det minsta elementet och placerar det först, upprepar detta steg tills hela listan är sorterad. Tidskomplexitet: Best, Worst, Average = O(n^2).

InsertionSort delar upp listan i sorterad och osorterad del. Element flyttas till rätt plats i den sorterade delen. Tidskomplexitet: Best = O(n), Worst, Average = O(n^2).

QuickSort är en snabb och effektiv sorteringsalgoritm som använder metoden ”divide-and-conquer” för att sortera en lista. Den delar listan genom ett pivot-element och rekursivt sorterar de mindre och större delarna. Quicksort använder sig av partitionsfunktioner, RightsidePartition eller Median3Partition. RightsidePartition: tar det högre elementet som pivot. Median3Partition: räknar ut medianen mellan 3 och väljer sedan pivot från det värdet.
Tidskomplexitet: Best, Average = O(n log n), Worst = O(n^2).

TimeMeasure.cpp/h

Dessa filer innehåller funktionerna för att mäta och beräkna standardavvikelse. Den kallar också på funktionerna för filhantering.

TimeAllAlgorithms skapar lambdafunktioner som gör att man enkelt kan passa en funktion som inparamter. Den sparar sen dessa lambdafunktioner i en vector och sparar filnamn i en vector.

Sen kallar funktionen på TimeCalculation med en sorteringsalgoritm och ett filnamn

TimeCalculation använder SortingTime och beräknar den genomsnittliga tiden det tar att sortera och räknar också ut och standardavvikelsen för 5 mätningar.

SortingTime är funktionen som räknar tiden för sorteringsalgoritmen att exekvera. Endast tiden att sortera räknas. Tiden sparas som en flyttal och returneras till TimeCalculation.

Main.cpp

Här kallas de inledande och själva körningsfunktionerna.

Den rensar också filerna på innehåll med ClearFiles.

Main skapar 4 olika huvudvectorer som innehåller 10 vectorer med samma datastruktur men med varierande storlek. Storleken är mellan 2 000 och 20 000.

Därefter itereras varje datasort en gång i taget och datan sparas i separata textfiler. Genom att iterera körs varje vector som finns i varje huvudvector.

Programkod Python

Koden i python är till för att generera plots för att enklare kunna visualisera den data som C++ koden genererade. Koden skapar numpy arrays och skapar sedan plots för varje enskild datafil. Därefter körs en funktion som ritar all data för varje sorteringsalgoritm i en plot. Så 4 linjer för tex quicksort, där varje linje representerar en distinkt datastruktur; Falling, Random, Rising eller Same.

Den använder biblioteket Matplotlib.
Matplotlib är en Python-bibliotek som används för att skapa statiska, interaktiva och publika grafer och diagram. Det är kraftfullt och flexibelt och används ofta inom dataanalys, datavisualisering och vetenskaplig forskning. Matplotlib ger användarna möjlighet att skapa en mängd olika typer av grafer, inklusive linjediagram, stapeldiagram, cirkeldiagram, histogram och mycket mer.

Fist2000@student.miun.se Filip Stenegren

Algoritmer

Selectionsort

SelectionSort delar listan upp i två delar; en sorterad del av listan som byggs upp från vänster till höger, den osorterade resterande delen av listan är till höger/efter den sorterade delen av listan. Till början är den sorterade delen tom(i majoriteten av fallen) och den osorterade delen är hela listan som ska sorteras. Selectionsort itererar genom listan, hittar det minsta (eller största) elementet och flyttar det till början av listan. Denna process upprepas därefter för varje element i listan tills hela listan är sorterad.

```
8  void SelectionSort(std::vector<int> &vec) {
9      int min, check, index;
10     for (min = 0; min < vec.size()-1; min++) {
11         index = min;
12         for (check = min+1; check < vec.size() ; check++) {
13             if (vec[check] < vec[index]) {
14                 index = check; }
15         }
16         std::swap(vec[index], vec[min]);
17     }
18 }
```

Insertionsort

Insertion sort är en enkel sorteringsalgoritm som fungerar genom att iterera genom en lista av element och "sätta in" varje element på rätt plats i den sorterade delen av listan. Algoritmen börjar med ett enda element i den sorterade delen och bygger gradvis upp en sorterad lista genom att jämföra och flytta element tills hela listan är sorterad. Detta görs genom att iterera genom varje element i listan och placera det på rätt plats bland de tidigare sorterade elementen.

```
20 void InsertionSort(std::vector<int> &vec) {
21     int insertionIndex, currentElement, comparisonIndex;
22     for (insertionIndex = 1; insertionIndex < vec.size(); insertionIndex++) {
23         currentElement = vec[insertionIndex];
24         comparisonIndex = insertionIndex - 1;
25         while (comparisonIndex >= 0 && vec[comparisonIndex] > currentElement) {
26             vec[comparisonIndex + 1] = vec[comparisonIndex];
27             comparisonIndex--;
28         }
29
30         vec[comparisonIndex + 1] = currentElement;
31     }
32 }
```

Quicksort

Quicksort är en sorteringsalgoritm som är baserad på divide-and-conquer-principen. Den fungerar genom att välja ett element i arrayen, kallat pivot, och sedan dela arrayen in i två delar: de element som är mindre än pivot och de element som är större än pivot. Sedan sorteras var och en av dessa delar rekursivt.

```
77  void QuickSort(std::vector<int> &vec, int low, int high, bool rightSidePivot) {
78      if (low < high) {
79          //split is the partitioning index
80          int split = Partition(vec, low, high, rightSidePivot);
81          QuickSort(vec, low, split - 1, rightSidePivot);
82          QuickSort(vec, split + 1, high, rightSidePivot);
83      }
84  }
50  int Partition(std::vector<int> &vec, int low, int high, bool rightSidePivot ) {
51      int pivot;
52      if (rightSidePivot) { pivot = vec[high]; }
53      else {
54          pivot = MedianPivot(vec, low, high);
55      }
56
57      int lowIndex = (low - 1);
58      for (int element = low; element <= high - 1 ; element++) {
59          if ((vec[element] < pivot)){
60              lowIndex++;
61              std::swap(vec[lowIndex], vec[element]);
62          }
63      }
64      std::swap(vec[lowIndex + 1], vec[high]);
65      return (lowIndex + 1);
66  }
35  int MedianPivot(std::vector<int> &vec, int low, int high) {
36      int mid = (low + high) / 2;
37      if (vec[mid] < vec[low]) {
38          std::swap(vec[low], vec[mid]);
39      }
40      if (vec[high] < vec[low]) {
41          std::swap(vec[low], vec[high]);
42      }
43      if (vec[mid] < vec[high]) {
44          std::swap(vec[mid], vec[high]);
45      }
46      //median is at vec[high]
47      return vec[high];
48  }
```

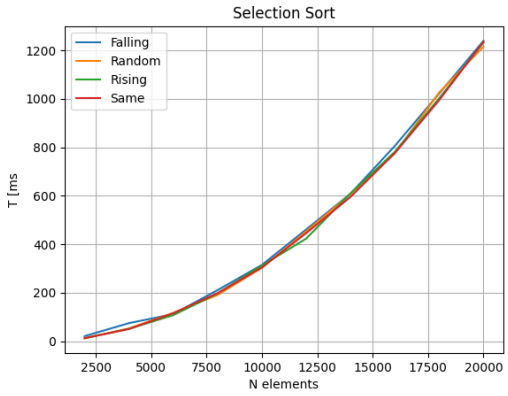
Steg 1: Välj pivot; Det första steget är att välja ett element i arrayen som ska vara pivot.

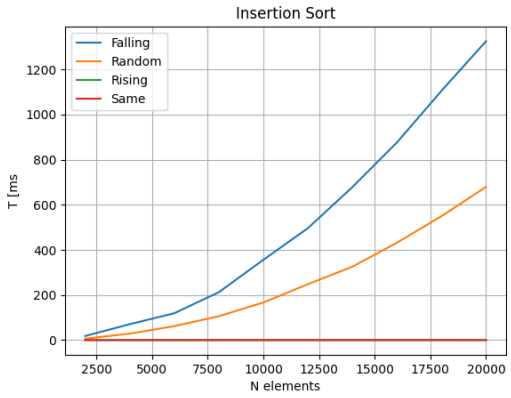
Steg 2: Partitionera arrayen; När pivot har valts används det för att partitionera arrayen i två delar: de element som är mindre än pivot och de element som är större än pivot. Detta görs genom att jämföra varje element i arrayen med pivot. Om elementet är mindre än pivot flyttas det till den vänstra delen av arrayen. Om elementet är större än pivot flyttas det till den högra delen av arrayen.

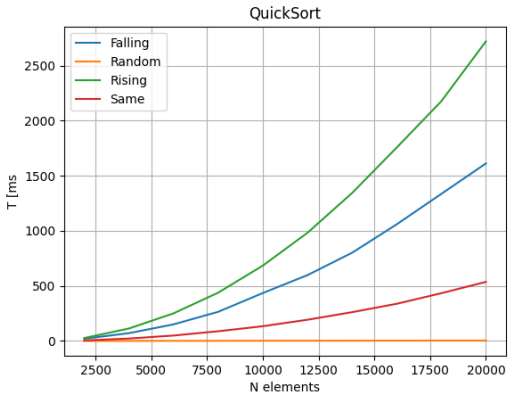
Steg 3: Sortera de två delarna; När arrayen har partitionerats sorteras var och en av de två delarna rekursivt. Detta innebär att quicksort används för att sortera den vänstra delen av arrayen, och sedan används quicksort för att sortera den högra delen av arrayen.

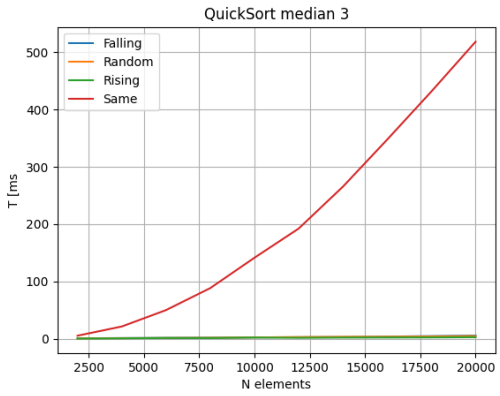
Steg 4: Rekursion; Processen fortsätter tills arrayen är sorterad. Detta sker när arrayen är så liten att den inte behöver partitioneras.

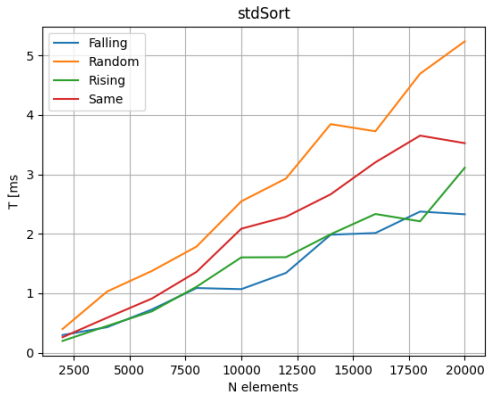
Resultat och Grafer











Grafer för varje individuell körning hittas på https://github.com/fillesten/DOA/tree/main/Lab2/Plots.

En snabb analys av dessa grafer visar att sorteringsalgoritmerna skiljde sig åt i prestanda beroende på framför allt datatyperna men också antal element.

Selectionsort

Selectionsort har en tidskomplexitet på O(N^2) för varje datatyp. Detta är för den har två nästlade loopar [1]. Varje loop tar ett element och jämför sedan det elementet gentemot alla andra element. Detta upprepas tills det sista elementet i listan. Mina resultat i grafen stämmer överens med den teoretiska aspekten till selectionsort.

Insertionsort

Både fallande och slumpmässig datastruktur har båda en tidskomplexitet på O(N^2). Med fallande data som ritar worst case och slumpmässig ett genomsnittligt resultat[2]. Den får denna tidskomplexitet eftersom varje element jämförs med varje tidigare element. Detta resulterar till kvadratiskt antal jämförelser. Insertionsorts best case sker när listan redan är sorterad vilket sker med datatyperna stigande och konstant, detta märks också i grafen då linjen uppfattas som platt vid 0. I dessa fall har algoritmen en tidskomplexitet på O (n). Resultaten i min graf speglar den teoretiska aspekten för insertionsort.

Quicksort

Quicksort med rightsidepivot kombinerat med slumpmässig datatyp kan man iaktta en tidskomplexitet på O(n log n). Partitioneringen av element kan ibland leda till obalanserade partitioner, men oftast tenderar de att vara rimligt balanserade. Detta innebär att de två resulterande partitionerna är i genomsnitt ungefär lika stora. När den rekursiva sorteringsprocessen fortsätter och involverar dessa mindre partitioner, upprepas partitioneringsproceduren. Varje steg i partitioneringen genererar partitioner av ungefär samma storlek, och dessa partitioner följer en logaritmisk höjd i rekursionsträdet.
För resterande datatyper, fallande, stigande och konstanta närmar tidskomplexiteten sig mot O(N^2). Detta sker om pivoten är det högsta minsta elementet eller om alla element är konstanta. Då partitioneras listan i två listor där den ena är n-1 stor. Detta gör att rekursionsträdet blir n steg. Mina resultat matchar den teoretiska aspekten kring quicksort. [3]

Quicksort Median of 3

Quicksort Median of 3 eliminerar chansen att uppnå O(N^2) för både fallande och stigande listor. Eftersom den räknar ut medianen mellan punkter väljs inte det största eller minsta elementet i en lista, och därmed har Quicksort Median of 3 en tidskomplexitet på O(n log n) för slumpmässiga, stigande och fallande. [3]

Median of thee worst case som wikipedias formal analysis anger är när alla element är konstanta [3]. Och det är för att medianen till 3 konstanta element är alltid detsamma. Det lägger till exekveringstid men i det stora hela bör resultaten för konstanta värden för både Quicksort median of 3 och vanlig quicksort vara detsamma. Som dessutom kan analyseras är att resultaten för en lista på 20 000 element är strax ovanför 500 för båda. Mina resultat matchar därmed den teoretiska aspekten kring quicksort

Stdsort

Är en kombinerad sorteringsfunktion som är en del av C++ standard bibliotek, den är definierad i <algortihm> headern. Stdsort har alltid en tidskomplexitet av O(n log n). Det är för std::sort är en hybrid sorteringsalgoritm som heter introsort. Introsort utnyttjar sig först av quicksort för små listor men vid större antal element byter till heap sort då heap sort har en worst case på O(n log n).

Diskussion

I och med utförandet av den här laborationen har sorteringsalgoritmer implementeras, testats och analyserats. Detta för att förstå deras prestationer kring olika datastorleker och datatyper. Resultaten som framtagits stämmer överens med de teoretiska förväntningarna kring varje algoritm.

Även fast sorteringsalgoritmer kan uppstå som enkla att förklara eller applicera i vardagliga livet är det svårare än vad som kan förväntas att implementera i kod. Insertionsort är en ganska simpel sorteringsalgoritm, då det kan liknas hur man sorterar en hand spelkort. Men kodmässigt måste man tänka lite extra för att implementera den.

Quicksort har bra average case för slumpmässiga men presterar dåligt beroende på input data och pivotelement. Att använda sig av median of 3 visade sig vara ett enkelt sätt att skära när på fall där quicksorts worst case uppstår.

Under testandet av alla algoritmer märkte jag att std::sort hade alltid väldigt bra resultat. Jag insåg först via mer undersökning att den uppnår dessa resultat via hybridalgoritmer. Detta talar om varför hybridalgoritmers styrka och är därför att föredra över enskilda algoritmer.

Programkod

https://github.com/fillesten/DOA/tree/main/Lab2

Källor

[1] Geeksforgeeks, ”Selection sort”, hämtad [2023-10-03]. Tillgänglig: https://www.geeksforgeeks.org/selection-sort/

[2] Wikipedia, “Insertion Sort”, hämtad [2023-10-03]. Tillgänglig: https://en.wikipedia.org/wiki/Insertion_sort

[3] Wikipedia, “Quick Sort”, hämtad [2023-10-03]. Tillgänglig: https://en.wikipedia.org/wiki/Quicksort

[4] Wikipedia, “std::sort för C++”, hämtad [2023-10-03]. Tillgänglig: https://en.wikipedia.org/wiki/Sort_(C%2B%2B)