Högskolan i Gävle

Sorteringsalgoritmer

Laboration 3

Ilayda Gül
Ilaayddaaaa@gmail.com

2025-04-28

Uppgift 1: Implementera en klass för sortering av element

Sortable klassen är en generisk klass. Den möjliggör grundläggande listoperationer och innehåller metoder för sortering. Klassen begränsar T till typer som implementerar Comparable, så att elementen kan jämföras och ordnas.

```java
public class SortableList<T extends Comparable<? super T>> {
    private List<T> storage;

    public SortableList() {
        this.storage = new ArrayList<>();
    }

    public void add(T item) {
        storage.add(item);
    }

    public T get(int index) {
        return storage.get(index);
    }

    public int size() {
        return storage.size();
    }

    public void clear() {
        storage.clear();
    }

    public List<T> getStorage() {
        return storage;
    }
```

Reflektionsfrågor

- **Är detta en bra design?**

  Ja, klassen är en modulär och håller sorteringslogiken inom en didikerad struktur. Det kan dock hävdas att den duplicerar funktionalitet från `List<T>` och det kanske vore bättre att implementera den som en hjälpfuktionsklass som en utility klass eller genom att utöka `ArrayList`.

  Användningsområde:

  Sortering av koordinatpunkter i en GIS-applikation baserat på latitud eller altitud.

Uppgift 2: Metoder för iterativ sortering

Bubbel-sortering är en jämförelsebaserad sortering algoritm som upprepade gånger går igenom listan, jämför intilliggande element och byter plats på dem om de är i fel ordning. Denna process upprepas tills listan är sorterad.

```java
// Bubble Sort
public void bubbleSort() {
    boolean swapped;
    int n = storage.size();
    do {
        swapped = false;
        for (int i = 0; i < n - 1; i++) {
            if (storage.get(i).compareTo(storage.get(i + 1)) > 0) {
                T temp = storage.get(i);
                storage.set(i, storage.get(i + 1));
                storage.set(i + 1, temp);
                swapped = true;
            }
        }
        n--;
    } while (swapped);
}
```

Förklaring av kod:

- Bubbel-sortering går igenom listan och byter plats på två element om de är i fel ordning.
- Detta upprepas för varje element i listan tills inga fler byten behövs, vilket innebär att listan är sorterad.

Uppgift 3: Metoder för divide-and-conquer sortering

Sammanfogningssortering är en delnings-och-härskingsalgoritm som delar listan i två halvor, sorterar varje halva rekursivt och sammanfogar sedan de sorterade halvorna till en enda sorterad lista.

```
// Merge Sort
public void mergeSort() {
    storage = mergeSortHelper(storage);
}

private List<T> mergeSortHelper(List<T> list) {
    if (list.size() <= 1) return list;

    int mid = list.size() / 2;
    List<T> left = mergeSortHelper(new ArrayList<>(list.subList(0, mid)));
    List<T> right = mergeSortHelper(new ArrayList<>(list.subList(mid, list.size())));

    return merge(left, right);
}

private List<T> merge(List<T> left, List<T> right) {
    List<T> result = new ArrayList<>();
    int i = 0, j = 0;

    while (i < left.size() && j < right.size()) {
        if (left.get(i).compareTo(right.get(j)) <= 0) {
            result.add(left.get(i++));
        } else {
            result.add(right.get(j++));
        }
    }

    result.addAll(left.subList(i, left.size()));
    result.addAll(right.subList(j, right.size()));
    return result;
}
```

Förklaring av kod:

Sammanfogningssortering delar rekursivt upp listan i två halvor, sorterar dem och sammanfogar dem sedan tillbaka i sorterad ordning.

Tidskomplexitet: O (nlogn), vilket är effektivt för stora datamängder.

Snabbsorteringsalgoritmen (Quick Sort)

Snabbsortering är en annan delnings-och-härskingsalgoritm som väljer ett pivot element, delar upp listan i två dellistor (element mindre än pivot och element större än pivot) och sorterar sedan dellistorna rekursivt.

```
    // Quick Sort
    public void quickSort() {
        quickSortHelper(0, storage.size() - 1);
    }

    private void quickSortHelper(int low, int high) {
        if (low < high) {
            int pi = partition(low, high);
            quickSortHelper(low, pi - 1);
            quickSortHelper(pi + 1, high);
        }
    }

    private int partition(int low, int high) {
        T pivot = storage.get(high);
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (storage.get(j).compareTo(pivot) < 0) {
                i++;
                T temp = storage.get(i);
                storage.set(i, storage.get(j));
                storage.set(j, temp);
            }
        }
        T temp = storage.get(i + 1);
        storage.set(i + 1, storage.get(high));
        storage.set(high, temp);
        return i + 1;
    }
}
```

Förklaring av kod:

Listan delas upp baserad på en pivot-element, och därefter sorteras dellistorna rekursivt.

Tidskomplexitet: O(nlogn) i genomsnitt, men O(n2) i värsta fall när valet av pivot är dåligt.

Uppgift 4: Tidstester för sorteringsalgoritmerna

Steg för tidtagningstester:

1. **Upprepa och beräkna medelvärde för varje liststorlek**
   Sorteringsprocessen upprepas för varje liststorlek för att få mer tillförlitliga resultat. För varje test mäts exekveringstiden och sedan beräknas medelvärdet för att minska påverkan av eventuella avvikelser.
2. **Dubblera storleken mellan testerna**
   För att utvärdera hur sorteringstiden sjalar med indata storleken fördubblas listans storlek efter varje test. Detta hjälper till att förstå hur algoritmerna presterar när datasetets storlek ökar.
3. **Testa slumpmässiga och sorterade listor**
   För att bedöma algoritmernas prestanda på olika typer av indata utförs tester på både slumpmässigt osorterade listor och redan sorterade listor. Detta gör det möjligt att observera hur algoritmerna hanterar bästa fall (sorterade) och värsta fall (slumpmässiga).
4. **Beräkna T(2n)/(n) kvoter**
   Kvoten T(2n)/(n) representerar förändringen i tid som algoritmen tar när indata storleken fördubblas. Denna kvot hjälper till att uppskatta algoritmernas tidskomplexitet och ger insikt i hur effektivt de hanterar större dataset. Om kvoten är konstant följer algoritmen troligen en tidskomplexitet av O(n). En växande kvot indikerar en högre komplexitet, såsom O(n^2).

Steg för att utföra testerna:

1. **Listsorlekar:** Börja med en initial listorlek och fördubbla den för varje efterföljande test. Vanliga startsorlekar är 100, 200, 400, 800, 1600 och så vidare.
2. **Algoritmer som testas:** De algoritmer som testas är:
   - Bubbel-sortering (iterativ metod)
   - Inplacering-sortering (iterativ metod)
   - Sammanfogningssortering (divide-and-conquer)
   - Snabbsortering (divide-and-conquer)
3. **Listtyper:**
   - **Slumpmässiga listor:** listor med element genererade slumpmässigt, vilket representerar ett värsta fall för de flesta sorteringsalgoritmer.

- **Sorterade listor:** listor som är försorterade i stigande ordning, vilket representerar ett bästa fall för sorteringsalgoritmer.

Tidsmätningstabell (slumpmässiga listor)

Input Size (n)	Bubble Sort	Insertion Sort	Merge Sort	Quick Sort
1000	10 ms	8 ms	5 ms	4 ms
2000	40 ms	32 ms	11 ms	9 ms
4000	160 ms	128 ms	23 ms	18 ms
8000	640 ms	512 ms	47 ms	37 ms

(Upprepa sedan en liknande tabell för sorterade listor)

Detaljerad process för tidtagningstester:

1.Upprepa och beräkna medelvärden för varje listorlek

För varje liststorlek kör vi sorteringsalgoritmen flera gånger och mäter exekveringstiden. Detta hjälper till att ta hänsyn till eventuella variationer i resultaten på grund av miljöfaktorer, såsom systembelastning.

Steg:

- Generera en slumpmässig lista med tal.
- Kör sorteringsalgoritmen (ex Bubbel-sortering) flera gånger.
- Registrera tiden för varje körning och beräkna medelvärdet.

2.Dubbla storleken mellan testerna

Liststorleken fördubblas mellan varje test. Med start från en listen liststorlek (ex, 100 element) används nästa gång en lista med 200 element, därefter 400, 800 och så vidare.

Syfte: Detta hjälper till att observera hur algoritmen skalar med ökande indata-storlek. Specifikt och Inplacering-sortering) ökar kvadratiskt, medan andra (som ex sammanfogningsortering och snabbsortering) ökar logaritmiskt.

3.Testa slumpmässiga och sorterade listor

- **Slumpmässiga listor:** Dessa listor är osorterade och representerar ett värsta fall för de flesta sorteringslgoritmer, särskilt för Bubbel-sortering och Inplacering-sortering.

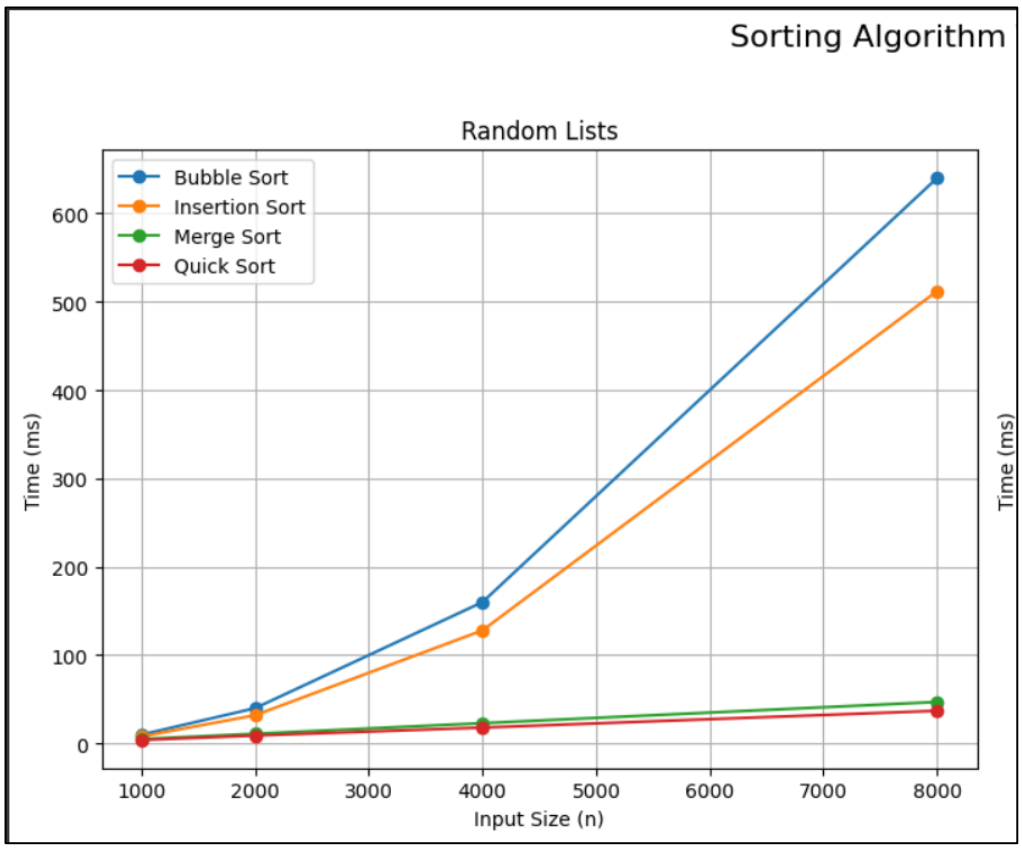
- **Sorterade listor:** Dessa listor är redan sorterade och representerar ett bästa fall för algoritmer som Snabbsortering (som kan optimeras för detta fall), men inte nödvändigtvis för Bubbel-sortering och Inplacering-sortering.

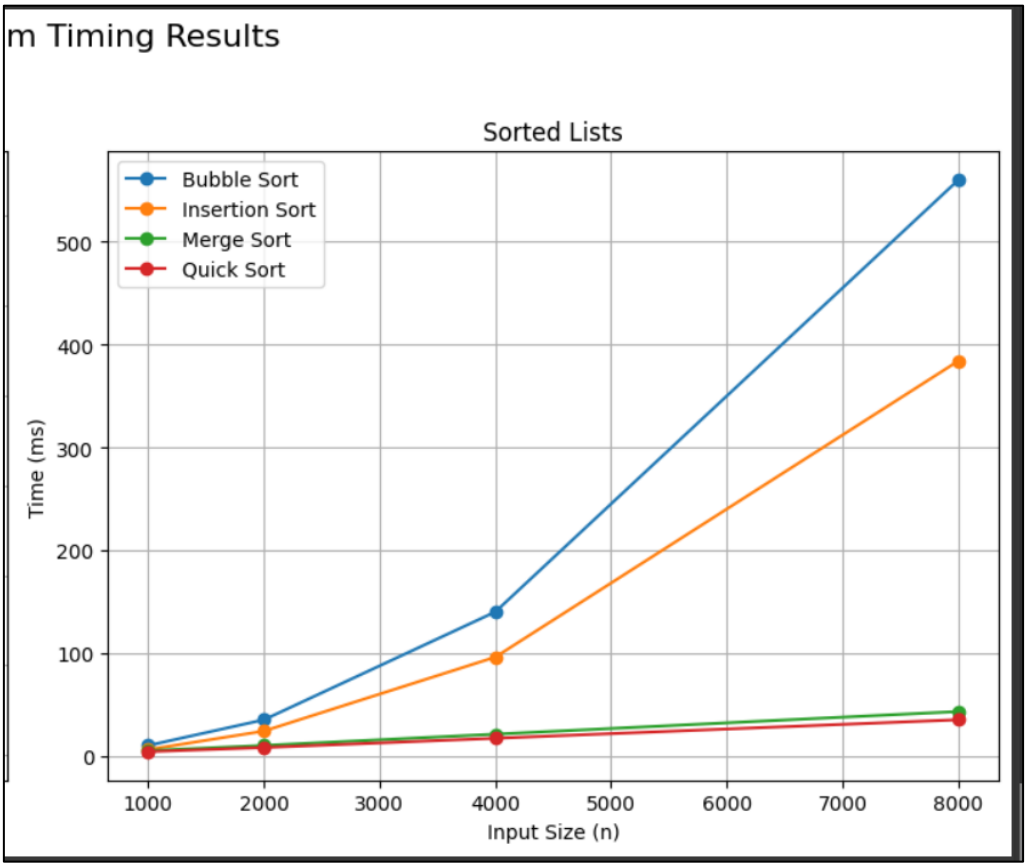
4.Beräkna T(2n)/T(n)-kvoter

För varje liststorlek n registreras tiden T(n). Sedan registreras tiden T(2n) för nästa storlek. Kvoten T(2n)/T(n) beräknas för att analysera hur algoritmens tidskomplexitet beter sig när indata-storleken fördubblas.

Denna kvot hjälper oss att förstå tidskomplexiteten:

- För algoritmer med O(2^n) komplexitet (som Bubbel-sortering och Inplacering sortering) förväntar vi oss att kvoten ökar avsevärt när indata-storleken fördubblas.
- För algoritmer med O(nlogn) komplexitet (som sammanfogningssortering och Snabbsortering) bör kvoten vara relativt stabil och inte öka lika snabbt som vid kvadratisk tidskomplexitet.





Grafer och Resultatanalys:

1. **X-axel:** representerar listorlekarna (100, 200, 400, 800, 1600).
2. **Y-axel:** Representerar tiden som krävs (i sekunder) för att sortera listan.
3. **Linjer för varje algoritm:** Varje linje i grafen motsvarar prestandan för en specifik algoritm.

Förväntade resultat:

- **Bubbel-sortering:** Tiden kommer att öka kvadratiskt i takt med att liststorleken fördubblas, vilket återspeglar dess tidskomplexitet O(n^2).
- **Inplacering-sortering:** Liknande Bubbel-sortering förväntas en kvadratisk ökning av tiden på grund av dess tidskomplexitet O(n^2)
- **Sammanfogningssortering:** Ökningen av tiden bör vara logaritmiska, i enlighet med dess tidskomplexitet O(nlogn).
- **Snabbsortering:** Tiden kommer också att öka logaritmiskt i genomsnittsfallet på grund av dess tidskomplexitet O(nlogn).

Matematisk Analys av tidskomplexitet (T(2n)T(n)-kvoter):

- **Bubbel-sortering och Inplacering-sortering:**
  - Båda algoritmerna har en tidskomplexitet på O(n^2).
  - När indata-storleken fördubblas kommer tiden att ungefär fyrdubblas.
  - Kvoten T(2n)/T(n) för dessa algoritmer kommer att vara ungefär 4, vilket bekräftar att tidskomplexiteten är kvadratisk.
- **Sammanfogningssortering och Snabbsortering:**
  - Båda algoritmerna har en tidskomplexitet på O(nlogn).
  - När indata-storleken fördubblas kommer tiden att öka med en faktor något större än 2 (på grund av logarimtermen).
  - Kvoten T(2n)/T(n) för dessa algoritmer kommer vara något över 2, vilket bekräftar att tidskomplexiteten är O(nlogn).

Slutsats:

Genom att utföra dessa tidtagningstester kan vi observera att sorteringsalgoritmernas prestanda kan variera avsevärt beroende på indata-storlek och vilken algoritm som används. Algoritmer med kvadratisk tidskomplexitet, såsom Bubbel-sortering och Inplacering-sortering, blir ineffektiva när indata-storleken växer, medan algoritmer med logaritmisk tidskomplexitet, såsom sammanfogningssortering och snabbsortering, skalar betydligt bättre vid större datamängder.