5Programe : Probleme de tip I/O local in 5 limbaje

Servicii web si tehnologii middleware (WSMT)

Prof.dr. Boian Florian

Satmari Iulius-Gabriel-Marian, gr. 244, SDI

Specificarea Problemei

Sa se implementeze rezolvarea problemei de mai jos in cinci variante, folosind respectiv limbajele: C#, Java, PHP, Python, NodeJs.

Pe cat posibil se vor păstra: numele variabilelor, algoritmul de rezolvare, meniul / interfața grafica, parametrii de intrare, ieșirile etc.

Sa se ordoneze alfabetic liniile unei matrice de stringuri prin sortare rapida (Quicksort). Doua linii se compara mai întâi prin prima coloana, la egalitate după a doua s.a.m.d Matricea se da intr-un fișier, specificând pentru fiecare celula: (numărul liniei, numărul coloanei, stringul). Numele fișierului se da in linia de comanda. Nu se fac limitări asupra dimensiunii matricei.

Aspecte Generale:

QuickSort

QuickSort este un algoritm de sortate (dezvoltat de către C. A. R. Hoare, in 1960), este similar cu MergeSort, fiind bazat pe tehnica „divide et impera” (dezbina si stăpânește), care presupune descompunerea problemei principale in doua sau mai multe subprobleme de același tip ca problema inițială, dar de dimensiuni mai mici.

Principiul de funcționare consta in alegerea unui element denumit pivot si partiționarea elementelor din jurul (stânga si dreapta) pivotului. Alegerea pivotului nu influențează in mod direct performanta algoritmului. De regula este ales unul dintre capetele vectorului de elemente (primul sau ultimul element), fie elementul din mijlocul acestuia.

Procesul cheie in algoritmul QuickSort consta in funcția de partiționare, al cărei scop este aranjarea elementelor din vector astfel încât elementele situate înaintea pivotului sa fie mai mici decât acesta, iar cele situate după acesta sa fie mai mari decât el (sau invers in cazul in care se optează pentru o sortare in mod descrescător). Acest proces este realizat in timp liniar, folosind interschimbarea elementelor. Astfel, QuickSort este un algoritm „in place” (pe loc) care

nu creează noi vectori, nefolosind spațiu suplimentar, precum mergesort, fiind mult mai eficient din punct de vedere al spațiului ocupat.

Complexitatea din punctul de vedere al timpului de execuție este $\theta(nlogn)$ in medie, iar in cel mai rău caz aceasta este $\theta(n^2)$. De cele mai multe ori, cel mai rău caz al complexității poate fi evitat prin folosirea unei variante optimizate de QuickSort numita Randomized QuickSort, care, printre altele, prevede ca pivotul sa fie ales in mod aleatoriu. Este, de cele mai multe ori, cea mai practica decizie atunci când vine vorba de alegerea unui algoritm de sortare si este folosit ca librărie standard de către majoritatea librăriilor de sortare din limbajele de programare.

Soluția Propusă

Structurarea datelor

Datele de intrare sunt preluate dintr-un fișier care conține, pe fiecare linie, cate 3 valori care specifica numărul liniei, numărul coloanei, si valoarea corespunzătoare acestora.

Pentru testarea soluției propuse s-au folosit următoarele 2 fișiere de intrare:

input.txt:

0 0 bread

1 0 ant

2 0 wolf

3 0 rabbit

4 0 chair

5 0 menu

6 0 internet

7 0 argument

Input1.txt

0 2 d

0 1 c

0 0 b

1 0 a

1 1 c

2 0 a

2 1 b

3 0 a

3 1 b

3 2 b

4 0 o

5 0 i

6 0 b

6 1 d

6 2 c

7 1 t

7 0 x

Matricea corespunzătoare fiecărui fișier arată in felul următor:

Pentru primul fișier:

bread
ant
wolf
rabbit
chair
menu
internet
argument

Pentru al doilea fișier:

b	c	d
a	c	
a	b	
a	b	b
o		
i		
b	d	c
x	t	

Observam ca datele de intrare nu trebuie sa fie in ordinea aparitiei acestora in matrice, cum de altfel, nici matricea nu trebuie sa fie pătratica.

Datele de ieșire (rezultatul obținut in urma sortării matricei de stringuri data in fișierul de intrare folosind QuickSort) vor fi salvate intr-in fișier text, sub forma de matrice.

Pentru al 2-lea exemplu de date de intrare, rezultatul obținut va fi salvat in fișierul output.txt si va arata in felul următor:

output.txt

a b

a b b

a c

b c d

b d c

i

o

x t

Implementare

Pentru a obține o implementare consistenta si similara in toate cele 5 limbaje de programare trebuie ținut cont de diferențele majore dintre ele si de particularitățile speciale ale acestora. Spre exemplu, un program scris in Java trebuie neapărat sa conțină clase, forțând astfel utilizarea de clase si in celelalte 4 limbaje.

Pentru reprezentarea datelor in fiecare dintre cele 5 limbaje s-a folosit **clasa RowElement** cu 2 parametrii: row si values, reprezentând valorile unei linii din matrice. Parametrul row, de regula de tip primitiv, integer, reprezintă numărul liniei, iar parametrul values reprezintă valorile de pe coloanele corespunzătoare liniei row. Pentru reprezentarea acestora s-a folosit o lista de perechi cheie-valoare, in care cheia reprezintă indexul/numărul coloanei iar valoarea reprezintă stringul de pe acea coloana. Diferențele in implementare sunt

după cum urmează: In Java este folosit un HashMap<Integer,String>, in C# un SortedDictionary<int,string>, un dicționar in Python, iar in PHP si Js s-a optat pentru folosirea de liste de obiecte. Clasa RowElement vine însoțita de o metoda addValue(column, value) care permite adăugarea unei noi perechi cheie-valoare in tipul de date values precum si metoda getValues() care returneaza lista de valori.

Java

```java
class RowElement {
  int row;
  HashMap<Integer, String> values = new HashMap<>();

  RowElement(int row, HashMap<Integer, String> values) {
    this.row = row;
    this.values = values;
  }

  void addValue(int column, String value) {

this.values.put(column, value);
  }

  HashMap<Integer, String> getValues() {
    return this.values;
  }
}
```

C#

```csharp
public class RowElement
  {
    public RowElement(int row, SortedDictionary<int, string> vals)
    {
      Row = row;
      Values = vals;
    }
    public int Row { get; }
    public SortedDictionary<int, string> Values { get; set; }

    public void AddValue(int column, string value)
    {
      Values.Add(column, value);
    }

  }
```

PHP

```php
class RowElement
{
  public $row;
  public $values;
  public function __construct($row, $values)
  {
    $this->row = $row;
    $this->values = $values;
  }
  public function addValue($column, $value)
  {
    $this->values[$column] = $value;
  }

  public function getValues()
  {
    return $this->values;
  }
}
```

Js	Python

class RowElement {
  constructor(row, values) {
  this.row = row;
  this.values = values;
  }
 	
  addValue(column, value) {
  this.values[column] = value;
  }
 	
  getValues() {
  return this.values;
  }
}
	class RowElement:
	  def __init__(self, row, values):
	  self.row = row
	  self.values = values
	
	  def addValue(self, column, value):
	  self.values[column] = value
	
	  def getValues(self):
	  return self.values

In **clasa Main** regăsim algoritmul de sortare (format din 2 funcții recursive) cat si restul funcțiilor utilizate in implementare: scrierea si citirea din fișier, interschimbarea a 2 elemente dintr-o lista, o funcție care compara 2 liste de stringuri, o funcție care returneaza un element dintr-o lista daca acesta exista, sau nul in caz contrar si funcția principala care combina toate aceste metode la un loc.

In **funcția principala** ne definim într-o variabila path-ul către fișierul de ieșire unde urmează sa afișam matricea sortata. Am structurat fișierele astfel încât fiecare program are propriul sau fișier de ieșire, iar citirea este realizata de către toate programele din același fișier. Pe urma, in modul specific fiecărui limbaj se obține parametrul din linia de comanda reprezentând path-ul către fișierul de intrare. Urmează instanțierea matricei folosind funcția de citire din fișier si salvarea acesteia într-o lista, cu elemente de tipul RowElement. Am optat pentru folosirea unei liste deoarece ordinea liniilor nu mai este relevanta pentru ca fiecare linie este salvata in câmpul row din RowElement. Pe urma este apelat algoritmul de sortare pe lista de elemente obținuta anterior, iar in cele de urma este apelata metoda de scriere in fișier.

```
  public static void main(String[] args) throws IOException {
    ArrayList<RowElement> elements = readFromFile(args[0]);
    String outputFilePath = "C:\\output.txt";

    quickSort(elements, 0, elements.size() - 1);
    writeToFile(outputFilePath, elements);
  }
```

Citirea din fișier: Se realizează folosind funcțiile specifice fiecărui limbaj de programare in parte, dar are o structura generala, comuna, in care se realizează citirea linie cu linie. Fiecare linie este parsata (împărțita) după caracterul spațiu, obținându-se astfel un vector cu 3 elemente corespunzătoare liniei, coloanei si valorii din matrice. Pe urma se verifica daca in lista exista deja un element de tipul RowElement pentru linia citita, caz in care acesta va fi actualizat si valoarea stringului va fi adăugată in dicționarul de valori din RowElements. In caz contrar un nou element de tipul RowElements va fi creat si adăugat la finalul listei de elemente.

```
  private static ArrayList<RowElement> readFromFile(String filePath) throws
IOException {
    ArrayList<RowElement> elements = new ArrayList<>();
    File file = new File(filePath);
    BufferedReader br = new BufferedReader(new FileReader(file));
    String data;
    String[] parsedData;

    while ((data = br.readLine()) != null) {
      parsedData = data.split(" ");
      int rowData = Integer.valueOf(parsedData[0]);
      int columnData = Integer.valueOf(parsedData[1]);
      String valueData = parsedData[2];
      RowElement row = getRow(elements, rowData);
      if (row != null) {
        int index = elements.indexOf(row);
        row.addValue(columnData, valueData);
        elements.set(index, row);
      } else {
        HashMap<Integer, String> val = new HashMap<>();
        val.put(columnData, valueData);
        elements.add(new RowElement(rowData, val));
```

```
      }
    }
    br.close();
    return elements;
  }
```

Scrierea in fișier se realizează in mod similar in toate cele 5 programe. Se parcurge lista de elemente, iar pentru fiecare element parcurgem toate elementele de tipul cheie-valoare din membrul Values, a căror valoare o afișam urmata de un spațiu si o noua linie după fiecare serie de elemente.

```
  private static void writeToFile(String filePath, ArrayList<RowElement>
elements) throws IOException {
    BufferedWriter writer = new BufferedWriter(new FileWriter(filePath));
    for (int i = 0; i < elements.size(); i++) {
      RowElement row = elements.get(i);
      for (int j = 0; j < row.getValues().size(); j++) {
        writer.append(row.getValues().get(j));
        writer.append(' ');
      }
      writer.append('\n');
    }
    writer.close();
  }
```

Algoritmul de sortare consta in 2 funcții recursive denumite quickSort si partition, care primesc 3 parametrii, **elements** - vectorul cu elemente de tipul RowElement; **low** - o valoare de tipul int care reprezintă poziția de pornire si **high** – poziția de oprire. Rolul **funcției de partiționare** este de a aranja elementele astfel încât cele situate înainte de pivot sa fie mai mici decât el, iar cele situate după el sa fie mai mari. Setam pivotul ca fiind cel mai din dreapta element de fiecare data, parcurgem vectorul începând de pe poziția low si contorizam poziția celui mai mic element într-o variabila i care reprezintă poziția pe care pivotul va trebui sa fie amplasat la finalul parcurgerii. Când găsim un element mai mic decât pivotul incrementam contorul si interschimbăm elementul curent cu elementul de pe indexul minim. La final interschimbăm elementul de poziția următoare corespunzătoare contorul minim cu pivotul si returnam poziția acestuia.

```
static int partition(ArrayList<RowElement> elements, int low, int high) {
    RowElement pivot = elements.get(high);
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
      if (compareRows(elements.get(j).getValues(), pivot.getValues()) < 1) {
        i++;
        swap(elements, i, j);
      }
    }
    swap(elements, i + 1, high);
    return (i + 1);
  }

  static void quickSort(ArrayList<RowElement> elements, int low, int high) {
    if (low < high) {
      int pi = partition(elements, low, high);
      quickSort(elements, low, pi - 1);
      quickSort(elements, pi + 1, high);
    }
  }
```

Pentru compararea elementelor este folosita **funcția de comparare** care parcurge 2 vectori de stringuri primiți ca parametrii si compara valorile aflate pe aceeași poziție. Daca acestea sunt egale trecem la următoarea poziție (coloana) si repetam procesul pana dam de 2 valori diferite, caz in care returnam 1 sau -1 in funcție de rezultatul comparării.

```
  static int compareRows(HashMap<Integer, String> a, HashMap<Integer, String> b)
{
    int mi = a.size();
    if (b.size() < mi)
      mi = b.size();
    int i = 0;
    while (i < mi) {
      if (a.get(i).compareTo(b.get(i)) == 0)
        i++;
      else if (a.get(i).compareTo(b.get(i)) < 0)
        return -1;
      else
        return 1;
    }
    if (a.size() == b.size())
      return 0;
```

```
    if (a.size() < b.size())
      return -1;
    return 1;
  }
```