Ministerul Educaţiei, Culturii și Cercetării  
Universitatea Tehnică a Moldovei

Departamentul Informatică și Ingineria Sistemelor

**RAPORT**

Lucrarea de laborator nr. 2  
la Analiza și Proiectarea Algoritmilor

A efectuat:   
st. gr. TI-206 Borș Nicoleta

A verificat: Ernest Bîtca

Chişinău – 2021

**Tema**: Metoda divide et impera

**Scopul lucrării:** Studierea metodei divide et impera. Analiza şi implementarea algoritmilor bazaţi pe metoda divide et impera.

**Sarcina:**

1. Studiaţi noţiunile teoretice despre metoda divide et impera.
2. Implementaţi algoritmii Mergesort şi Quicksort.
3. Efectuaţi analiza empirică a algoritmilor Mergesort şi Quicksort.
4. Implementați +1 algoritm de sortare: **Radix Sort.**
5. Faceţi o concluzie asupra lucrării efectuate.

#### **Întrebări de control:**

#### De ce această metodă se numeşte divide et impera ?

#### **Divide et impera** este o tehnica de elaborare a algoritmilor care constă în descompunerea în subprobleme și rezolvarea lor, ceea ne dăm seama și din numele lui „dezbină și stăpânește” .

#### Explicaţi noţiunea de algoritm recursiv.

#### Un algoritm recursiv realizează de fapt o prelucrare repetitivă a lui. Un algoritm recursiv este considerat simplu recursiv dacă conține un singur auto-apel și multiplu recursiv dacă conține două sau mai multe auto-apeluri. Un algoritm recursiv este caracterizat prin:

#### • Condiție de oprire. Specifică situația în care rezultatul se poate obține prin calcul direct fără a mai fi necesară apelarea aceluiași algoritm.

#### • Auto-apel. Se apelează cel puțin o dată pentru alte valori ale parametrilor. Valorile parametrilor corespunzătoare succesiunii de apeluri trebuie să asigure apropierea de satisfacerea condiției de oprire.

#### Descrieţi etapele necesare pentru rezolvarea unei probleme prin metoda divide et impera.

#### Descompunerea în subprobleme, rezolvarea subproblemelor și combinarea rezultatelor.

#### Ce tip de funcţie descrie timpul de execuţie al unui algoritm de tipul divide et impera?

#### În timp liniar log – O (nlogn)

#### Care sunt avantajele şi dezavantajele algoritmilor divide et impera?

Avantaje: produce algoritmi eficienți; descompunerea problemei in subprobleme facilitează paralelizarea algoritmului in vederea execuției sale pe mai multe procesoare.

#### Dezavantaje: se adaugă un overhead datorat recursivității (reținerea pe stivă a apelurilor funcțiilor).

#### Rezultatele din tabelele 1, 2 și 3 le găsiți în Anexa 1, 2, și 3

#### **Tabelul 1 – Timpul de execuție (s) pentru sortarea tabloului cu ajutorul algoritmului Quick Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | Taboul random | Taboul sortat crescător | Taboul sortat descrescător | Taboul aproape sortat |
| 10 | 0.00009 | 0.0001 | 0.0001 | 0.0001 |
| 100 | 0.0003 | 0.0014 | 0.0007 | 0.0003 |
| 1000 | 0.002 | 0.0087 | 0.0062 | 0.0043 |
| 10000 | 0.0073 | S-a umplut stiva | | 0.1044 |
| 100000 | 0.0220 | S-a umplut stiva |
| 1000000 | 0.1690 |

#### Din cauza pivotului, care e ales mereu ultimul, la sortarea tablourilor deja sortate sau aproape sortate, riscăm să ni se umple stiva, astfel în acest caz algoritmul Quick Sort e ineficient. Dacă stiva nu se umplă oricum Quick Sort este lent, având o complexitate de O(n2) pe când algoritmul Merge Sort și Radix sort nu sunt afectate din acest punct de vedere. Putem să evităm această complexitate, găsind mediana tabloului și alegând pivotul ca fiind mediana, însă aceasta necesită calcule suplimentare, astfel este mai bine să folosim alți algoritmi când știm că problema noastră întâmpinăm des așa situații.

#### **Tabelul 2 – Timpul de execuție (s) pentru sortarea tabloului cu ajutorul algoritmului Merge Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | Taboul random | Taboul sortat crescător | Taboul sortat descrescător | Taboul aproape sortat |
| 10 | 0.00009 | 0.0002 | 0.0001 | 0.0002 |
| 100 | 0.0004 | 0.0003 | 0.0002 | 0.0002 |
| 1000 | 0.0028 | 0.0031 | 0.0045 | 0.0035 |
| 10000 | 0.0097 | 0.0077 | 0.0086 | 0.0095 |
| 100000 | 0.0415 | 0.0258 | 0.0261 | 0.0336 |
| 1000000 | 0.3196 | 0.1865 | 0.1958 | 0.2095 |

#### Observăm că Merge Sortul nu întâmpină probleme când tabloul este sortat și timpul de execuție nu diferă tare în aceste 4 situații fiind aproape egal. Deși la fel ca la Quick Sort și Radix Sort nu este bine că execută tot atâtea calcule și iterații cât pentru un tablou nesortat, sau chiar și mai multe în cazul lui Quick Sort. Acest lucru puteți să-l obervați analizând anexele 1, 2, și 3 unde sunt atașate toate rezultatele din tabelele 1, 2, 3 și mai mult decât atât aveți posibilitatea să analizați și numărul de iterații, comparații, etc.

#### **Tabelul 3 – Timpul de execuție (s) pentru sortarea tabloului cu ajutorul algoritmului Radix Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | Taboul random | Taboul sortat crescător | Taboul sortat descrescător | Taboul aproape sortat |
| 10 | 0.00009 | 0.0001 | 0.0001 | 0.00009 |
| 100 | 0.00009 | 0.0001 | 0.0004 | 0.00009 |
| 1000 | 0.0008 | 0.0013 | 0.0012 | 0.0008 |
| 10000 | 0.0265 | 0.0283 | 0.0202 | 0.0274 |
| 100000 | 0.087 | 0.0947 | 0.992 | 0.0878 |
| 1000000 | 0.4815 | 0.4792 | 0.5820 | 0.4937 |

#### La fel ca Merge Sort, Radix Sort nu întâmpină probleme la tablouri sortate. Hai să analizăm cum se comportă fiecare algoritm în dependență de n:

|  |  |
| --- | --- |
| **Tablou sortat (crescător)** | |
|  |  |
| **Tablou random** | |
|  |  |

#### Din graficele de mai sus observăm că Quick Sort este cel mai lent la sortarea tabloului deja sortat iar pentru n mare se încarcă stiva și este total ineficient. La un tablou random însă este cel mai rapid pentru n mare iar pentru n mic este mai lent decât Radix Sort. Merge Sort-ul însă n-are probleme cu tablourile sortate, astfel este cel mai rapid când n-ul este mare, și la fel mai încet decât Radix Sort-ul la n-uri mici. **Din aceste observări putem traga concluzia că cel mai rapid la tablouri random cu n-uri mari este Quick Sort-ul, pentru tablouri sortate cu n-uri mari este Merge Sort, iar pentru n-uri mici în toate cazurile Radix Sort.** Acest lucru este din cauza că se completează tabloul cu valori până la n. Hai să modificăm valoarea maximală a tabloului să fie 10 pentru toate cazurile și să vedem cum se va comporta fiecare algoritm în parte:

#### **Tabelul 4 – Timpul de execuție (s) pentru sortarea tabloului random cu valori până la 10**

|  |  |  |  |
| --- | --- | --- | --- |
| n | Quick Sort | Merge Sort | Radix Sort |
| 10 | 0.00009 | 0.0002 | 0.0002 |
| 100 | 0.0004 | 0.0004 | 0.0004 |
| 1000 | 0.0036 | 0.0028 | 0.0003 |
| 10000 | 0.0303 | 0.0083 | 0.0075 |
| 100000 | S-a umplut stiva | 0.0286 | 0.0166 |
| 1000000 | 0.2507 | 0.0794 |

#### Rezultatele din tabelul 4 le găsiți în anexa 4.

|  |  |
| --- | --- |
| **Tablou random (completat cu valori până la 10)** | |
|  |  |

#### Analizând graficele de mai sus și tabelul 4 observăm că Radix Sort e cel mai rapid indiferent de n. Quick Sortul la fel ca la tablouri sortate, la n mare i s-a umplut stiva din cauza numerelor care se repetau des și la fel a fost cel mai lent dintre toate.

#### Hai să analizăm acum numărul de operații pe care îl face fiecare algoritm, în tabloul de mai jos:

#### **Tabloul 5 – Numărul de apelare a recursiei și numărul de comparații pentru sortarea tabloului random Quick Sort vs Merge Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Random | Nr. de apelare a recursiei | | Nr. de comparații | |
| n | Quick Sort | Merge Sort | Quick Sort | Merge Sort |
| 10 | 15 | 19 | 25 | 23 |
| 100 | 135 | 199 | 667 | 547 |
| 1000 | 1371 | 1999 | 11277 | 8717 |
| 10000 | 13551 | 19999 | 153473 | 120481 |
| 100000 | 135683 | 199999 | 2028131 | 1536637 |
| 1000000 | 1354587 | 1999999 | 24403171 | 18674275 |

|  |  |
| --- | --- |
| **Tablou random (nr. de apelare a recursiei)** | |
|  |  |
| **Tablou random (nr. de comparații)** | |
|  |  |

#### Observăm că indiferent de n, Quick Sortul are mai puține apelări decât Merge Sortul, iar Merge Sortul are mai puține comparații decât Quick Sortul. Totuși observăm că în tabelul 6 nu s-a inclus Radix Sort, deoarece el nu face nici o comparație și nici nu se autoapelează. El va chema funcția countSort de atâtea ori, câte cifre are cel mai mare număr din tablou astfel numărul de iterații depinde de cel mai mare număr. Din acest motiv, un **caz defavorabil** al utilizării algoritmului Radix Sort este atunci când numărul maxim este cu mult mai mare decât celelalte.

#### De exemplu dacă vom avea un tabloul din doar 2 elemente în care cel mai mare număr va fi cu mult mai mare decât primul, se vor executa foarte multe iterații. Un exemplu puteți să-l vedeți în tabelul 6, unde veți observa că funcția countSort a fost apelată de 6 ori pentru a executat sortarea a două numere și a făcut 66 calcule și 12 mutări. Quick Sortul însă a sortat acest tabel foarte ușor, dar totuși pentru el acest tablou este la fel un caz defavorabil, deoarece el deja este sortat, astfel a făcut niște iterații în plus astfel Merge Sortul a fost cel mai rapid.

#### **Tabelul 6 – Sortarea unui tablou din 2 elemente, caz defavorabil pentru Radix Sort**

|  |  |
| --- | --- |
| **Radix Sort** |  |
| **Quick Sort** |  |
| **Merge Sort** |  |

#### 

#### Dacă Radix Sort a făcut atât de multe calcule pentru 2 numere, ce va fi dacă o să avem un tabloul de 1000 elemente și doar unul va fi cu mult mai mare decât toate celelalte? O să facă o mulțime iterații în plus și respectiv și numărul de calcule va crește. Și în final, o să analizăm complexitatea fiecărui algoritm.

#### **Tabelul 7 – Complexitatea algoritmilor Quick Sort, Megre Sort și Radix Sort**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Caz favorabil** | **Caz mediu** | **Caz defavorabil** |
| **Quick Sort** | **(n\*log n)** | **(n\*log n)** | **O(n2)** |
| **Merge Sort** | **(n\*log n)** | **(n\*log n)** | **O(n\*log n)** |
| **Radix Sort** | **(kn)** | **(kn)** | **O(kn)** |
| **Counting sort** | **(n)** | **(n)** | **O(n)** |

#### k – numărul de cifre a celui mai mare număr

#### **Concluzie:**

#### În cadrul lucrării de laborator la tema: „Metoda divide et impera” am studiat această metodă și am implementat algoritmii de bază a acestei metode: Merge Sort și Quick Sort. Pe lângă aceasta am implementat și un Radix Sortul, iar studiind acest algoritm am avut posibilitatea să învăț și Counting Sortul întrucât Radix Sortul cheamă Counting Sortul să soreteze numere cuprinse între 0-9. Implementând acești algorimi și executându-i pentru mai multe cazuri mi-am tras următoarele concluzii:

#### Din cauza pivotului, care e ales mereu ultimul, la sortarea tablourilor deja sortate sau aproape sortate, algoritmul Quick Sort e ineficient, complexitatea acestuia fiind de O(n2) sau se poate umple stiva.

#### Quick Sort-ul, Merge Sortul și Radix Sortul la sortarea unui tablou sortat sau aproape sortat execută tot atâtea calcule și iterații cât pentru un tablou nesortat, sau chiar și mai multe în cazul lui Quick Sort.

#### Cel mai rapid la tablouri random cu n-uri mari este Quick Sortul, pentru tablouri sortate cu n-uri mari este Merge Sort, iar pentru n-uri mici în toate cazurile cel mai rapid este Radix Sort.

#### Indiferent de n, Quick Sortul are mai puține apelări decât Merge Sortul, iar Merge Sortul are mai puține comparații decât Quick Sortul.

#### Un caz defavorabil al utilizării algoritmului Radix Sort este atunci când numărul maxim este cu mult mai mare decât celelalte.

#### Concluzia mea finală însă este că fiecare algoritm are atât plusuri, cât și minusuri, astfel la alegerea algoritmului trebuie să atragem atenția la problema pe care trebuie să o rezolvăm și la ce cazuri defavorabile putem ajunge pentru a putea evita anumite neplăceri.

#### **Bibliografia:**

#### Else APA16.3 - <https://else.fcim.utm.md/course/view.php?id=773>

#### Else APA16.1 - <https://else.fcim.utm.md/course/view.php?id=7>

#### GitHub - <https://github.com/Nicoleta-Bors/APA.git>

#### Anexa 1 – Rezultatele Quick Sort

|  |  |
| --- | --- |
| n | Tabel random |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

|  |  |
| --- | --- |
| n | Tabel sortat crescator |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000, 100000, 1000000 |  |

|  |  |
| --- | --- |
| n | Tabel sortat descrescator |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000, 100000, 1000000 |  |

|  |  |
| --- | --- |
| n | Tabel aproape sortat |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000, 1000000 |  |

#### Anexa 2 – Rezultatele Merge Sort

|  |  |
| --- | --- |
| n | Tabel random |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

|  |  |  |
| --- | --- | --- |
| n | | Tabel sortat crescator |
| 10 | |  |
| 100 | |  |
| 1000 | |  |
| 10000 | |  |
| 100000 | |  |
| 1000000 | |  |
| n | Tabel sortat descrescator | |
| 10 |  | |
| 100 |  | |
| 1000 |  | |
| 10000 |  | |
| 100000 |  | |
| 1000000 |  | |

|  |  |
| --- | --- |
| n | Tabel aproape sortat |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

#### Anexa 3 – Rezultatele Radix Sort

|  |  |
| --- | --- |
| n | Tabel random |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

|  |  |
| --- | --- |
| n | Tabel sortat crescator |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

|  |  |
| --- | --- |
| n | Tabel sortat descrescator |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

|  |  |
| --- | --- |
| n | Tabel aproape sortat |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

#### Anexa 4.1 – Tablou random cu valori până la 10 – Quick Sort

|  |  |
| --- | --- |
| n |  |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000,1000000 |  |

#### Anexa 4.2 – Tablou random cu valori până la 10 – Merge Sort

|  |  |
| --- | --- |
| n |  |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

#### Anexa 4.3 – Tablou random cu valori până la 10 – Radix Sort

|  |  |
| --- | --- |
| n |  |
| 10 |  |
| 100 |  |
| 1000 |  |
| 10000 |  |
| 100000 |  |
| 1000000 |  |

#### Anexa 5 – Codul programului

*//prettier-ignore*

let apelRecursie = 0, comparatii = 0, interschimbari = 0, alipiri = 0, mutari = 0, counturi=0, apelCount=0;

*//--------------------------------ARRAY GENERATOR-------------------------//*

const getN = () => {

  let choice = parseInt(

    prompt(

      'Alegeti nr de elementele a tabloului:\n1 - 10\n2 - 100\n3 - 1000\n4 - 10000\n5 - 100000\n6 - 1000000\n7 - alta varianta',

    ),

  );

  let n;

  switch (choice) {

    case 1:

      n = 10;

      break;

    case 2:

      n = 100;

      break;

    case 3:

      n = 1000;

      break;

    case 4:

      n = 10000;

      break;

    case 5:

      n = 100000;

      break;

    case 6:

      n = 1000000;

      break;

    case 7:

      n = parseInt(prompt('Introduceti numarul de elemente a tabloului:'));

      break;

    default:

      console.log('Nu exista asa optiune');

      n = 0;

  }

  return n;

};

*//-------------------------------------------------------------------------------//*

const randomArray = (n, maxValue) => {

  let arr = [];

  for (let i = 0; i < n; i++) arr[i] = Math.floor(Math.random() \* maxValue);

  return arr;

};

const ascendingArray = (n) => {

  let arr = [];

  for (let i = 0; i < n; i++) arr[i] = i;

  return arr;

};

const descendingArray = (n) => {

  let arr = [];

  for (let i = 0; i < n; i++) arr[i] = n - i;

  return arr;

};

const almostSorted = (n, maxValue) => {

  let arr = [];

  for (let i = 0; i < n - 1; i++) arr[i] = i;

  arr[n - 1] = Math.floor(Math.random() \* maxValue);

  return arr;

};

*//-------------------------------------------------------------------------------//*

const makeArr = () => {

  const choice = parseInt(

    prompt(

      'Alegeti tipul tabloului:\n1 - Random\n2 - Sortat crescator\n3 - Sortat descrescator\n4 - Aproape sortat\n5 - Introdus in cod',

    ),

  );

  let arr = [];

  let n;

  switch (choice) {

    case 1:

      {

        n = getN();

        const maxValue = parseInt(

          prompt('Introduceti numarul maxim cu care sa fie completat tabloului:'),

        );

        arr = randomArray(n, maxValue);

      }

      break;

    case 2:

      n = getN();

      arr = ascendingArray(n);

      break;

    case 3:

      n = getN();

      arr = descendingArray(n);

      break;

    case 4:

      {

        n = getN();

        const maxValue = parseInt(

          prompt('Introduceti numarul maxim cu care sa fie completat tabloului:'),

        );

        arr = almostSorted(n, maxValue);

      }

      break;

    case 5:

      arr = [3, 432212];

      n = arr.length;

      break;

    default:

      console.log('Nu exista asa optiune');

  }

  console.group(`%cTabloul initial (n = ${n}):`, 'color: #D789D7');

  console.log(arr.join(', '));

  console.groupEnd();

  return arr;

};

*//-------------------------------------------------------------------------------//*

*//-------------------------------------QUICK SORT--------------------------------//*

const quickSort = (arr, start, end) => {

  let pivIndex;

  if (start < end) {

    pivIndex = partition(arr, start, end);

    quickSort(arr, start, pivIndex - 1); *//stanga*

    quickSort(arr, pivIndex + 1, end); *//dreapta*

  }

  apelRecursie++; *// contur apel quickSort*

};

const partition = (arr, start, end) => {

  let pivot = arr[end];

  let partIndex = start;

  for (i = start; i <= end - 1; i++) {

    if (arr[i] <= pivot) swap(arr, i, partIndex++);

    comparatii++; *//contor comparatii*

  }

  swap(arr, end, partIndex);

  return partIndex;

};

const swap = (arr, start, end) => {

  let temp = arr[start];

  arr[start] = arr[end];

  arr[end] = temp;

  interschimbari++; *//contor interschimbari*

};

*//-------------------------------------------------------------------------------//*

*//------------------------------------Merge Sort---------------------------------//*

const mergeSort = (arr, start, end) => {

  if (start < end) {

    let middle = parseInt((start + end) / 2);

    mergeSort(arr, start, middle);

    mergeSort(arr, middle + 1, end);

    merge(arr, start, middle, end);

  }

  apelRecursie++;

};

const merge = (arr, start, middle, end) => {

  let nLeft = middle - start + 1; *//pana la mijloc inlcusiv*

  let nRight = end - middle; *//dupa mijloc*

*//prettier-ignore*

  let leftArr = [], rightArr = [];

  for (let i = 0; i < nLeft; i++) leftArr[i] = arr[start + i];

  for (let j = 0; j < nRight; j++) rightArr[j] = arr[middle + 1 + j];

*//prettier-ignore*

  let i = 0, j = 0, k = start;

  while (i < nLeft && j < nRight) {

    if (leftArr[i] <= rightArr[j]) arr[k++] = leftArr[i++];

    else arr[k++] = rightArr[j++];

    comparatii++;

  }

*//daca mai avem elem. ramase*

  while (i < nLeft) arr[k++] = leftArr[i++];

  while (j < nRight) arr[k++] = rightArr[j++];

  alipiri++;

};

*//-------------------------------------------------------------------------------//*

*//------------------------------------Radix Sort---------------------------------//*

const radixSort = (arr, n) => {

  let exp = 1,

    maxNumber = getMax(arr, n);

  while (Math.floor(maxNumber / exp)) {

    apelCount++;

    countSort(arr, n, exp);

    exp \*= 10;

  }

};

const getMax = (arr, n) => {

  let maxNumber = arr[0];

  for (let i = 1; i < n; i++) if (arr[i] > maxNumber) maxNumber = arr[i];

  return maxNumber;

};

const countSort = (arr, n, exp) => {

  let output = [],

    count = new Array(10).fill(0),

    i;

  for (i = 0; i < n; i++) {

    count[Math.floor(arr[i] / exp) % 10]++;

    counturi++;

  } *//sum of each element*

  for (i = 1; i < 10; i++) {

    count[i] += count[i - 1];

    counturi++;

  } *//sum of previous counts*

  for (i = n - 1; i >= 0; i--) {

    output[count[Math.floor(arr[i] / exp) % 10] - 1] = arr[i]; *//nr din arr va fi inexul pt output = nr din arr*

    count[Math.floor(arr[i] / exp) % 10]--;

    mutari++;

  }

  for (i = 0; i < n; i++) arr[i] = output[i]; *// sorted arr*

};

*//-------------------------------------------------------------------------------//*

const chooseAlgorithm = () => {

  let myArr = makeArr();

*//prettier-ignore*

  let choice = parseInt(prompt('Alegeti algoritmul cu care doriti sa sortati tabelul:\n1 - Quick Sort\n2 - Merge Sort\n3 - Radix Sort'));

  if (myArr == 0) choice = 0;

  switch (choice) {

    case 1:

      {

        const startTime = window.performance.now();

        quickSort(myArr, 0, myArr.length - 1);

        const endTime = window.performance.now();

        console.group('%cTabloul sortat (Quick Sort):', 'color: #D789D7');

        console.log(myArr.join(', '));

        console.groupEnd();

*//CONTORURI*

        console.group('%cContoruri: ', 'color: #D789D7');

        console.log(`%cNr. de apelare a recursiei: ${apelRecursie}`, 'color: #FADBEB');

        console.log(`%cNr. de comparatii: ${comparatii}`, 'color: #F3BAD6');

        console.log(`%cNr. de interschimbari: ${interschimbari}`, 'color: #EA86B6');

        console.log(`%cTimpul de rulare: ${endTime - startTime} ms = ${(endTime - startTime) / 1000} s`, 'color: #E05297');

        console.groupEnd();

      }

      break;

    case 2:

      {

        const startTime = window.performance.now();

        mergeSort(myArr, 0, myArr.length - 1);

        const endTime = window.performance.now();

        console.group('%cTabloul sortat (Merge Sort):', 'color: #D789D7');

        console.log(myArr.join(', '));

        console.groupEnd();

*//CONTORURI*

        console.group('%cContoruri: ', 'color: #D789D7');

        console.log(`%cNr. de apelare a recursiei: ${apelRecursie}`, 'color: #FADBEB');

        console.log(`%cNr. de comparatii: ${comparatii}`, 'color: #F3BAD6');

        console.log(`%cNr. de alipiri: ${alipiri}`, 'color: #EA86B6');

        console.log( `%cTimpul de rulare: ${endTime - startTime} ms = ${(endTime - startTime) / 1000} s`,'color: #E05297',);

        console.groupEnd();

      }

      break;

    case 3:

      {

        const startTime = window.performance.now();

        radixSort(myArr, myArr.length);

        const endTime = window.performance.now();

        console.group('%cTabloul sortat (Radix Sort):', 'color: #D789D7');

        console.log(myArr.join(', '));

        console.groupEnd();

*//CONTORURI*

        console.group('%cContoruri: ', 'color: #D789D7');

        console.log(`%cNr. de mutari: ${mutari}`, 'color: #FADBEB');

        console.log(`%cNr. de counturi: ${counturi}`, 'color: #F3BAD6');

        console.log(`%cNr. de apelare a functiei countSort: ${apelCount}`, 'color: #EA86B6');

        console.log(`%cTimpul de rulare: ${endTime - startTime} ms = ${(endTime - startTime) / 1000} s`,'color: #E05297');

        console.groupEnd();

      }

      break;

    default:

      console.log('Nu exista asa optiune');

  }

};

*//-------------------------------------------------------------------------------//*

chooseAlgorithm();