UNIVERSITATEA TEHNICĂ A MOLDOVEI

Structuri de Date și Algoritmi

Tehnici de sortare

Profesor: Maria Guțu

UNIVERSITATEA TEHNICĂ A MOLDOVEI

Tehnici de sortare

Algoritmii de sortare:

Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Shell Sort, Counting Sort, Comb Sort, Heap Sort, Radix Sort – semnatică, sintaxă, exemple de implementare.

Tehnici de sortare

Algoritmii de sortare sunt esențiali în programare pentru a organiza date într-o ordine specifică. Fiecare algoritm de sortare are avantaje și dezavantaje care depind de contextul în care sunt utilizați. Iată câteva dintre acestea în contextul limbajului de programare C.

Tehnici de sortare: avantaje & dezavantaje

Bubble Sort

Avantaje:

Simplitatea implementării.

Funcționează bine pentru seturi de date mici.

Dezavantaje:

Performanța slabă pe seturi de date mari, deoarece are o complexitate în timp de O(n^2).

Tehnici de sortare: avantaje & dezavantaje

Insertion Sort:

Avantaje:

Eficient pentru seturi de date mici sau parțial sortate.

Simplu de implementat și înțeles.

Dezavantaje:

Nu este eficient pentru seturi de date mari, cu complexitatea în timp O(n^2).

Tehnici de sortare: avantaje & dezavantaje

Selection Sort:

Avantaje:

Simplitatea și ușurința de implementare.

Dezavantaje:

Performanța slabă pe seturi de date mari, deoarece are o complexitate în timp de O(n^2).

Tehnici de sortare: avantaje & dezavantaje

Merge Sort:

Avantaje:

Eficient pe seturi de date mari.

Complexitatea în timp este O(n log n).

Stabil și poate fi implementat pentru liste înlănțuite.

Dezavantaje:

Consum mai mare de memorie datorită necesității de spațiu suplimentar pentru stocarea intermediară.

Tehnici de sortare: avantaje & dezavantaje

Quick Sort:

Avantaje:

Foarte eficient pe seturi de date mari.

Complexitatea în timp este în medie O(n log n).

Utilizează spațiu de memorie mai puțin decât Merge Sort.

Dezavantaje:

Performanța poate fi afectată în cazul unui set de date aproape sortat.

Tehnici de sortare: avantaje & dezavantaje

Shell Sort:

Avantaje:

Eficiență moderată cu o complexitate de O(n^2) și cei cu O(n log n).

Este relativ ușor de implementat și de înțeles.

Stabil în practică pentru seturi de date de dimensiuni mici și medii.

Dezavantaje:

Complexitatea mai mare și nu atinge aceeași eficiență în timp.

Dependența de secvența de pași aleasă.

Tehnici de sortare: avantaje & dezavantaje

Counting Sort:

Avantaje:

Eficiență pe seturi de date mici și cu valori limitate.

Este stabil și eficient în practică pentru anumite tipuri de date.

Nu necesită comparații între elemente, ceea ce îl face rapid.

Dezavantaje:

Limitări în ceea ce privește valori mari deoarece necesită multă memorie.

Nestabil pentru anumite implementări.

Tehnici de sortare: avantaje & dezavantaje

Comb Sort:

Avantaje:

Performanță moderată.

Ușor de implementat.

Dezavantaje:

Dependență de factorul de reducere a distanței, și o alegere inadecvată poate afecta eficiența.

Nu este la fel de eficient ca alți algoritmi mai avansați precum Quick Sort sau Merge Sort.

Tehnici de sortare: avantaje & dezavantaje

Heap Sort:

Avantaje:

Eficiență în practică pentru seturi de date de dimensiuni mari.

Stabilitate, menținând ordinea relativă a elementelor cu aceeași valoare.

Utilizare eficientă a memoriei.

Dezavantaje:

Complexitate în implementare și mai dificilă de înțeles comparativ cu alți algoritmi de sortare.

Nu este la fel de rapid ca Quick Sort pentru anumite tipuri de date.

Tehnici de sortare: avantaje & dezavantaje

Radix Sort:

Avantaje:

Eficiență pe seturi de date mari cu valori mici, în special pentru sortarea numerelor întregi.

Este stabil și menține ordinea relativă a elementelor cu aceeași cheie.

Necesită doar O(n + k) spațiu auxiliar, destul de mic.

Dezavantaje:

Limitări (nu se aplică) pentru datele de tip string sau alte structuri complexe.

Dependență de baza radixului și numărul de cifre al valorilor.

Implementare

Tehnici de sortare: implementare

Metoda bulelor cunoscută și sub numele *BubbleSort*, metoda bulelor se bazează pe următoare idee:

- fie un vector X[] cu n elemente
- parcurgem vectorul și pentru oricare două elemente învecinate care nu sunt în ordinea dorită, le interschimbăm valorile
- după o singură parcurgere, vectorul nu se va sorta, dar putem repeta parcurgerea
- dacă la o parcurgere nu se face nicio interschimbare, vectorul este sortat

Tehnici de sortare: implementare

O reprezentare a algoritmului (BubbleSort) este:

- cât timp vectorul nu este sortat
  - presupunem că vectorul este sortat
  - parcurgem vectorul
    - dacă două elemente învecinate nu sunt în ordinea dorită
      - le interschimbăm
      - schimbăm presupunerea inițială

Tehnici de sortare: BubbleSort

```
bool sortat;
do {
  sortat = true;
  for(int i = 0 ; i < n - 1 ; i ++)
```

```
if(v[i] > v[i+1])
    {
      int aux = v[i];
      v[i] = v[i+1];
      v[i+1] = aux;
      sortat = false;
    }
} while(!sortat);
```

Tehnici de sortare: implementare

Sortarea prin selecție (Selection Sort) se bazează pe următoarea idee:

- fie un vector X[] cu n elemente;
- plasăm în X[0] cea mai mică valoare din vector;
- plasăm în X[1] cea mai mică valoare rămasă;
- etc.

Tehnici de sortare: implementare

O descriere a algoritmului (Selection Sort) este:

- parcurgem vectorul cu indicele i
  - parcurgem cu indicele j elementele din dreapta lui x[i]
- dacă elementele x[i] și x[j] nu sunt în ordinea dorită, le interschimbăm

Tehnici de sortare: Selection Sort

```
int n, X[100];
//citire X[] cu n elemente

for(int i = 0 ; i < n - 1 ; i ++)
    for(int j = i + 1 ; j < n ; j ++)
```

```
if(X[i] > X[j])
        {
            int aux = X[i];
            X[i] = X[j];
            X[j] = aux;
        }
```

Tehnici de sortare: implementare

Sortarea prin inserție (Insertion Sort) se bazează pe următoarea idee:

- fie un vector X[] cu n elemente;
- dacă secvența cu indici 0, 1, …, i-1 este ordonată, atunci putem insera elementul X[i] în această secvență astfel încât să fie ordonată secvența cu indici 0, 1, …, i-1, i.
- luăm pe rând fiecare element X[i] și îl inserăm în secvența din stânga sa
- la final întreg vectorul va fi ordonat

Tehnici de sortare: implementare

O reprezentare a algoritmului (Insertion Sort) este:

- parcurgem vectorul cu indicele i
  - inserăm pe X[i] în secvența din stânga sa; pentru inserare se mută unele elemente din secvență spre dreapta

Tehnici de sortare: Insertion Sort

```
for(int i = 1 ; i < n ; i ++){
     int p = i;
     while(p > 0 && a[p] < a[p-1]){
          int aux = a[p];
          a[p] = a[p-1];
          a[p-1] = aux;
          p --;
     }
}
```

Tehnici de sortare: implementare

Merge Sort se bazează pe ideea împărțirii și combinării (sau fuzionării) repetate a unei liste în două părți până când fiecare sublistă devine un singur element, moment în care acestea sunt fuzionate pentru a crea o listă sortată. Procesul de împărțire și fuziune continuă până când întreaga listă devine sortată.

Tehnici de sortare: Merge Sort

Împărțirea (Divide):

Lista inițială este împărțită în două subliste egale, recursiv.

Procesul de împărțire continuă până când fiecare sublistă conține un singur element.

Fuzionarea (Merge):

Subliste consecutive sunt combinate în mod repetat pentru a forma subliste mai mari, sortate.

Fuziunea continuă până când lista inițială este reconstruită într-o singură listă sortată.

Tehnici de sortare: Merge Sort

Această abordare divizează problema sortării în subprobleme mai mici și mai ușor de gestionat. Prin împărțirea listei în subliste, **Merge Sort** creează un cadru structural care permite o fuziune eficientă și ordonată a elementelor. Un aspect cheie al Merge Sort este că este un algoritm de sortare stabil, ceea ce înseamnă că menține ordinea relativă a elementelor cu aceeași valoare în lista sortată.

T

```
void merge(int arr[],
    int i, j, k;
    int n1 = middle - le
    int n2 = right - mid

    // Crearea subliste
    int leftArray[n1];
    for (i = 0; i < n1; i++
        leftArray[i] = arr[

    // Crearea subliste
    int rightArray[n2];
    for (j = 0; j < n2; j++
        rightArray[j] = ar
```

```
int main() {
// Exemplu de apel al functiei de fuziune
        int arr[] = {12, 11, 13, 5, 6, 7};
        int arr_size = sizeof(arr) / sizeof(arr[0]);
        int left = 0;
        int right = arr_size - 1;
        int middle = (left + right) / 2;
        merge(arr, left, middle, right);
// Afisarea rezultatului
        printf("Sorted array: ");
        for (int i = 0; i < arr_size; i++)
                printf("%d ", arr[i]);
        return 0;
}
```

nente ramase in

i];

nte ramase in

y[j];

Tehnici de sortare: implementare

Quick Sort este un algoritm eficient de sortare bazat pe strategia "împărțirii și stăpânirii". Acesta se bazează pe ideea de a alege un element pivot din lista de sortat și a rearanja elementele astfel încât elementele mai mici decât pivotul să fie în stânga sa, iar cele mai mari să fie în dreapta. Apoi, Quick Sort este aplicat recursiv asupra celor două subliste rezultate, până când întreaga listă devine sortată.

Tehnici de sortare: Quick Sort

Principalele etape ale algoritmului Quick Sort sunt:

Alegerea pivotului:

Un element pivot este ales din lista de sortat.

O alegere comună este să se ia ultimul element din listă.

Partiționarea:

Elementele sunt rearanjate astfel încât elementele mai mici decât pivotul să fie în stânga sa, iar cele mai mari să fie în dreapta.

După această etapă, pivotul este în poziția sa finală în lista sortată, iar lista este împărțită în două subliste.

Aplicarea recursivă:

Quick Sort este aplicat recursiv pentru cele două subliste create în urma partiționării.

Algoritmul se repetă până când fiecare sublistă conține un singur element, moment în care lista inițială este sortată.

T

```
void swap(int*
b) {
    int temp = *
    *a = *b;
    *b = temp;
}
```

```
// Funcție pentru alegerea pivotului și rearanjarea elementelor în jurul pivotului
```

```
// Funcție recursivă pentru Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
// Găsirea pivotului și plasarea acestuia în poziția corectă
        int pi = partition(arr, low, high);

// Aplicarea recursivă pentru cele două subliste
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
}
```

Tehnici de sortare: implementare

Shell Sort este un algoritm de sortare eficient care se bazează pe ideea de a compara și interschimba elemente aflate la o distanță fixă unul de celălalt și apoi reducerea acestei distanțe progresiv până când lista devine aproape sortată. Acesta este o îmbunătățire a algoritmului de sortare prin inserție.

Tehnici de sortare: Shell Sort

Principalele etape ale algoritmului Shell Sort sunt:

Alegerea secvenței de pasi:

Se alege o secvență de pasi (interval) care să fie folosită pentru compararea și interschimbarea elementelor.

Compararea și interschimbarea la distanțe fixe:

Elementele sunt comparate și interschimbate între ele la distanțe fixe, în funcție de secvența de pași aleasă. Această etapă ajută la aducerea elementelor mici sau mari la locuri corespunzătoare înainte de sortarea finală.

Reducerea distanței și repetarea procesului:

Distanta fixă este redusă și procesul este repetat. Acest pas se repetă până când distanța fixă devine 1, moment în care algoritmul se comportă similar cu un algoritm de sortare prin inserție clasic.

Shell Sort


```
void shellSort(int arr[], int n) {
  for (int gap = n / 2; gap > 0; gap /= 2) {
     // Efectuarea sortarii prin inserție pentru fiecare sublistă creată de gap
     for (int i = gap; i < n; i++) {
        int temp = arr[i];
        int j;
         // Rearanjarea elementelor în sublista
        for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
           arr[j] = arr[j - gap];
         // Plasarea elementului în poziția corectă în sublista
        arr[j] = temp;
     } } }
```

Tehnici de sortare: implementare

Counting Sort este un algoritm de sortare ne-comparativ, bazat pe numărul de apariții ale fiecărui element în lista de sortat. Acesta este eficient pentru sortarea unui set de date cu valori întregi non-negative și cu un interval de valori limitat. **Counting Sort** funcționează prin construirea unui vector de frecvență pentru fiecare valoare distinctă și apoi reconstruirea listei sortate pe baza acestor frecvențe.

Tehnici de sortare: Counting Sort

Principalele etape ale algoritmului Counting Sort sunt:

Calculul frecvențelor:

Se construiește un vector de frecvență pentru fiecare valoare distinctă din lista de sortat. Fiecare element din vectorul de frecvență reprezintă numărul de apariții ale respectivei valori în lista de sortat.

Calculul pozițiilor absolute:

Se calculează poziția absolută a fiecărui element în lista sortată pe baza vectorului de frecvență. Aceasta se face prin adunarea frecvențelor anterioare.

Construirea listei sortate:

Pe baza pozițiilor calculate, se construiește lista sortată.

Tehnici de sortare: Counting Sort

```
// Copierea listei sortate în array-ul original
for (int i = 0; i < n; i++) {
    arr[i] = sorted[i];
}
// Eliberarea memoriei alocate dinamic
free(freq);
free(sorted);
```

Tehnici de sortare: implementare

Comb Sort este un algoritm de sortare bazat pe îmbunătățirea algoritmului de sortare prin interschimb (Bubble Sort). Acesta se bazează pe ideea de a reduce distanța dintre elementele comparate și interschimbate în mod progresiv, astfel încât să poată sorta eficient liste cu elemente mari la început.

Tehnici de sortare: Comb Sort

Principalele etape ale algoritmului Comb Sort sunt:

Inițializarea distanței:

Inițial, se alege o distanță inițială mare între elementele de comparat.

Compararea și interschimbarea la distanțe fixe:

Elementele de la distanța fixă sunt comparate și interschimbate în funcție de nevoie.

Asemănător cu Bubble Sort, perechile de elemente consecutive sunt comparate și interschimbate.

Reducerea distanței:

Distanța este redusă în fiecare iterație după o regulă specifică (de obicei, se reduce prin împărțirea la un factor redus).

Repetarea procesului:

Procesul de comparare și interschimbare la distanțe fixe, urmat de reducerea distanței, este repetat până când distanța devine 1.

Tehnici de sortare: Comb Sort

Algoritmul **Comb Sort** are avantajul de a depăși în mod eficient dezavantajele Bubble Sort-ului pentru seturi de date mari, fără a sacrifica simplitatea. Distanța mai mică la început permite elementelor mari să se deplaseze rapid către pozițiile corecte, iar ulterior, la distanțe mai mici, se efectuează ajustările necesare.

Tehnici de sortare: Comb Sort

```
void combSort(int arr[], int n) {
    int gap = n; // Distanța inițială între elemente
    float shrinkFactor = 1.3; // Factorul de reducere al distanței
    int swapped = 1; // Indicator pentru a verifica dacă s-a realizat vreun interschimb

    while (gap > 1 || swapped) {
        // Reducerea distanței la fiecare iterație
        if (gap > 1) {
            gap = (int)(gap / shrinkFactor);
        }
```

Tehnici de sortare: Comb Sort

```
swapped = 0;
        // Comparare și interschimbare la distanțe fixe
        for (int i = 0; i + gap < n; i++) {
            if (arr[i] > arr[i + gap]) {
                swap(&arr[i], &arr[i + gap]);
                swapped = 1;
            }
        }
    }
}
```

Tehnici de sortare: implementare

Heap Sort este un algoritm de sortare eficient care se bazează pe structura de date cunoscută sub numele de heap. Heap-ul este o structură de date arborescentă în care fiecare nod are o valoare mai mică sau egală cu valorile nodurilor sale descendente. Heap Sort funcționează prin transformarea listei de sortat într-un heap, apoi prin extragerea repetată a elementului maxim și reconstruirea heap-ului rezultat.

https://www.youtube.com/watch?v=HqPJF2L5h9U

Tehnici de sortare: Heap Sort

Principalele etape ale algoritmului Heap Sort sunt:

Construirea heap-ului (Heapify):

Elementele din lista de sortat sunt transformate într-un heap. În cazul unui heap maxim (max-heap), fiecare nod trebuie să aibă o valoare mai mică sau egală cu valorile nodurilor sale descendente.

Construcția heap-ului începe de la mijlocul listei și merge spre început, asigurându-se că fiecare subarbore este un heap maxim.

Extragerea elementelor:

Elementele sunt extrase din heap unul câte unul. De fiecare dată când un element este extras, heap-ul este reconstruit pentru a păstra proprietatea de heap maxim.

Elementele extrase sunt plasate la sfârșitul listei, formând astfel o listă sortată în ordine descrescătoare.

Construirea listei sortate:

Procesul de extragere și reconstrucție a heap-ului se repetă până când toate elementele au fost extrase.

Lista sortată este obținută inversând ordinea elementelor extrase, astfel încât să fie în ordine crescătoare.

Tehnici de sortare: Heap Sort

```
// Funcție pentru a construi un heap maxim dintr-un subheap dat
void heapify(int arr[], int n, int i) {
    int largest = i; // Initializarea nodului radacina
    int left = 2 * i + 1; // Indexul nodului stanga
    int right = 2 * i + 2; // Indexul nodului dreapta

    // Daca nodul stanga este mai mare decat radacina
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
```

Tehnici de sortare: Heap Sort

```
// Daca nodul dreapta este mai mare decat radacina
    if (right < n && arr[right] > arr[largest])
        largest = right;
    // Daca nodul cel mai mare nu este radacina
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        // Aplicam recursiv heapify pe subarborele afectat
        heapify(arr, n, largest);
    }
```

Tehnici de sortare: Heap Sort

```
void heapSort(int arr[], int n) {
    // Construim un heap maxim (Heapify)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    // Extragerea repetată a elementelor din heap
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]); // Mutam nodul cel mai mare la sfarsitul listei
        heapify(arr, i, 0); // Reconstruim heap-ul excluzand nodul mutat
    }
}
```

Tehnici de sortare: implementare

Radix Sort este un algoritm de sortare ne-comparativ care se bazează pe ideea de a sorta elementele pe baza cifrelor lor. Acesta este eficient pentru sortarea unui set de date cu numere întregi non-negative. Radix Sort sortează elementele pe baza fiecărei cifre, începând de la cea mai mică către cea mai mare, până când toate cifrele au fost luate în considerare.

Tehnici de sortare: Radix Sort

Principalele etape ale algoritmului Radix Sort sunt:

Identificarea numărului maxim de cifre:

Se determină numărul maxim de cifre în numerele din lista de sortat. Acest lucru determină cât de multe iterații vor fi necesare pentru a sorta lista.

Sortarea pe baza cifrelor:

În fiecare iteratie, elementele sunt sortate pe baza cifrei curente (de la cea mai mică către cea mai mare). Sortarea poate fi realizată folosind un algoritm de sortare stabil, cum ar fi Counting Sort.

Reconstruirea listei sortate:

După fiecare iteratie, lista este reconstruită luând în considerare ordinea dată de cifrele sortate până în acel moment.

Repetarea procesului:

Pasul de sortare și reconstruire se repetă pentru fiecare cifră, până când toate cifrele au fost luate în considerare.

Tehnici de sortare: Radix Sort

Radix Sort este eficient pentru sortarea unor seturi mari de numere cu un număr limitat de cifre. Dacă setul de date conține numere cu un număr variabil de cifre, este necesară o adaptare a algoritmului pentru a trata această situație.

Tehnici de sortare: Radix Sort

```
int findMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}
```

```
void radixSort(int arr[], int n) {
    int max = findMax(arr, n);
// Aplicăm Counting Sort pentru fiecare cifră
for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
```

Tehnici de sortare: Radix Sort

Această implementare a algoritmului Radix Sort ilustrează modul în care cifrele sunt luate în considerare pe rând pentru a sorta lista. Algoritmul utilizează Counting Sort pentru a efectua sortarea pe baza cifrelor.

Aplicații Practice!!!

(maria.gutu@iis.utm.md)