Laborator_05

Sorting Algorithm

Bubblesort

```
-> O(n^2)
-> daca nu pui >= e stabil
```

ShakerSort

```
-> e stabil
```

Selectionsort

```
-> cautam sa mutam minimul pe prima pozitie in tot array-ul
->
```

InsertionSort

```
->
```

1 4 5 3 2
1 4 3 5 2
1 3 4 5 2
1 3 4 2 5
1 3 2 4 5
1 2 3 4 5

ShellSort

```
-> e cumva legat de Insertion Sort
-> ia termenii mai mari decat elementul de pe pozitia curenta si ii schimba
-> O(n^2) ca toate de pana acum
-> Este stabil ? Nu este stabil pentru ca avem gap ul
```

CountingSort

```
-> O(n)
-> se face vector de frecventa cu val maxim elemente
-> se aduna nr elementelor din vectorul de frecventa ( de ex primele n) si se afla pozitia celui de-al n - 1 lea termen, indiferent de ce tip de vector sortam
-> se incepe cu un for din dreapta
```

Aplicatie laborator 5

Scrieti in C

struct Msg{
unsigned int prio;
char payload[256];
unsigned long size;
char rq; // 0 sau 1
}

Fa Insertion Sort
-> rq 0 inainte de rq 1
+ descrescator dupa prioritate(daca au acelasi rq)
+ crescator dupa size(daca au rq si prio egale)
+ alfabetic dupa payload(daca au toate restul campurilor egale)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Msg {
        unsigned int prio;
        char payload[256];
        unsigned long size;
        char rq; // asta poate fi 0 sau 1
}Msg_t;



/*

void insertion_sort(tip_element a[], int n)
{
        int i, j;
        tip_element tmp;
```

```
        for (i = 1; i < n; i++)
        {
                tmp = a[i];
                for (j = i; (j > 0) && (tmp.cheie < a[j - 1].cheie); j--)
                        a[j] = a[j - 1];
                a[j] = tmp;
        }
}
*/

int cmp(Msg_t a, Msg_t b) {
        if (a.rq != b.rq) {
                return a.rq - b.rq;
        }
        if (a.prio != b.prio) {
                return b.prio - a.prio; //descrescator dupa prioritate
        }
        if (a.size != b.size) {
                return a.size - b.size;
        }

        return strcmp(a.payload, b.payload);
}

void insertion_sort(Msg_t *arr, int n) {
        int i, j;

        Msg_t tmp;

        for (i = 1; i < n; i++) {
                tmp = arr[i];

                for (j = i; (j > 0) && cmp(tmp, arr[j - 1]) < 0; j--) {
                        arr[j] = arr[j - 1];
                }

                arr[j] = tmp;
        }
}



void printArray(Msg_t* arr, int n) {
        for (int i = 0; i < n; i++) {
```

```
                printf("rq: %d < - > prio: %u < - > size: %lu < - > payload:
%s\n", arr[i].rq, arr[i].prio, arr[i].size, arr[i].payload);
        }

}

int main(void) {
        struct Msg messages[] = {
                {5, "AA", 120, 1},
                {3, "BA", 100, 0},
                {7, "AB", 50, 0},
                {3, "CD", 100, 0},
                {4, "DC", 120, 1}
        };

        int n = sizeof(messages) / sizeof(messages[0]);

        printf("Array inainte de sortare : \n");

        printArray(messages, n);

        insertion_sort(messages, n);

        printf("\n Array dupa de sortare : \n");

        printArray(messages, n);


        return 0;
}
```

Laborator 6

Heapsort -> pereche cu SelectionSort(dar asta e mai slab)

```
-> imbunatatire la selection sort
-> de n ori gasim minimul / maximul
-> avem timp logaritmic
-> N * log2 N pentru construire
-> nu este prea stabil
```

Quicksort

```
|42|17|9|5|0|3|8|59|2|15|1|

->cea mai naspa complexitate pe care o poate avea este O(n^2);
->best case poate fi cand este nimerita mediana si este O(n * log n);
-> mediana se apropie de O(n);
-> este stabil
```

Binsort

```
-> il putem folosi doar daca sortam numere de la 0 la (n - 1)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------------------------------|
| 3 | 6 | 0 | 5 | 7 | 2 | 1 | 4 |
```

RadixSort -> interschimbare SAU -> direct

```
typedef struct Student{
        int nota;
        char gen;
        char *nume;
}Student_t;
```

```
-> O ( N * (nr biti / (sizeof(grupare)));
-> trade folosesc mai multa memorie si ai o complexitate mai buna
-> este stabil, face counting sort uri
```

```
int getBit(int x, int b){
        if(x & (1 << b)){
                return 1;
        }
        return 0;
}
```

```
int getBit(int x, int b){
        return !!(x && (1 << b));
}
```

```
-> daca vreau sa fie o grupare de biti cu cate g biti
```

```
int getBit(int x, int b, int g){
        return x >> b * g & ~(~0 << g);
}
```

Mergesort

```
-> O(N) ca si complexitate
-> clasica interclasare
-> ai nevoie de doua tablouri deja sortate
-> saptamana viitoare vom vorbi tot de mergesort
```