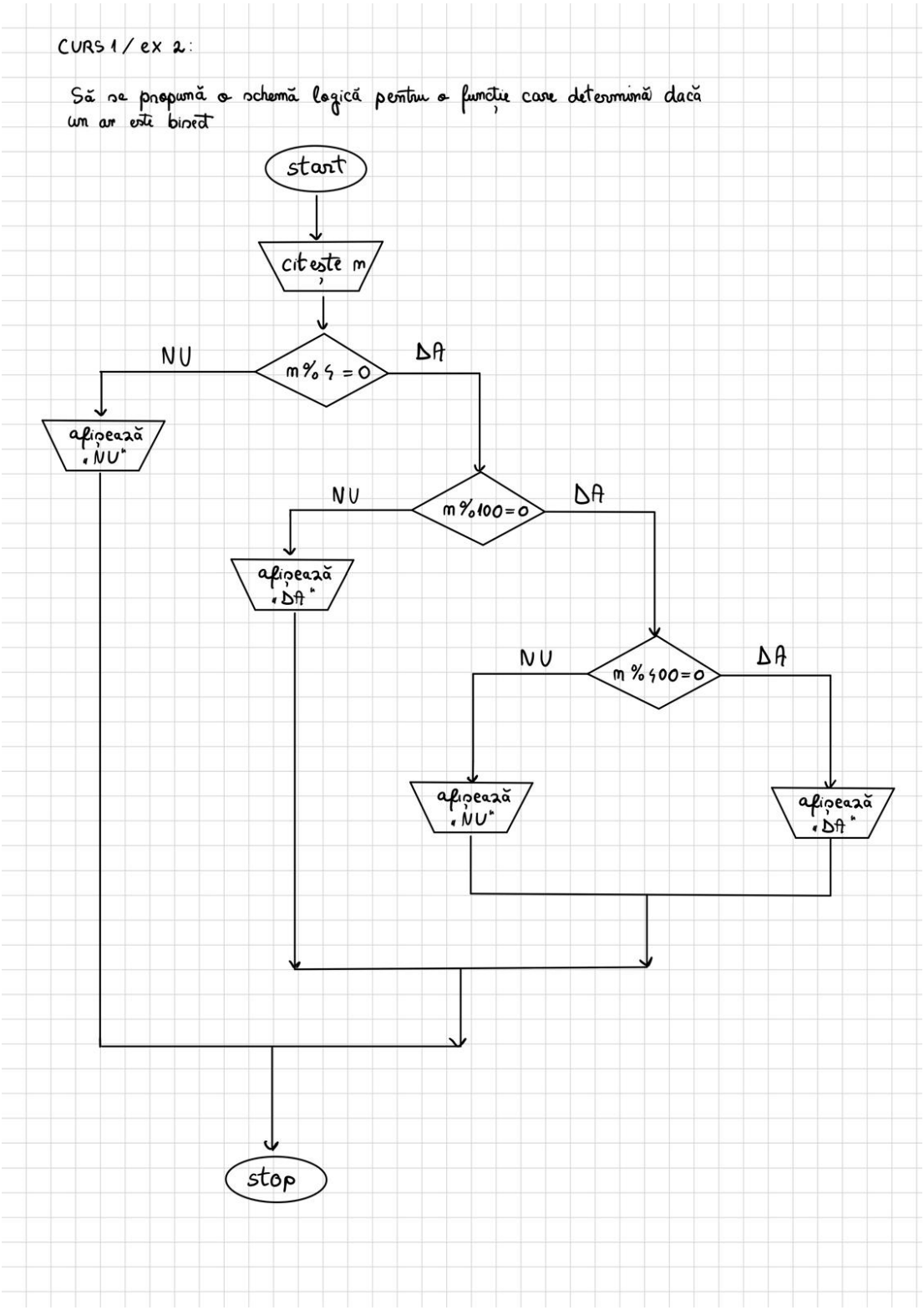
Curs1-ex2: Să se propună o schemă logică pentru o funcție care determină dacă un an este bisect.



Curs2-ex2: Definiți un TDA listă de întregi. Specificați ce operații doriți să fie definite pentru acest TDA. Apoi propuneti un fișier de tip header pentru o implementare a unei biblioteci in C pentru TDA abstract definit anterior.

```
typedef struct list
{
   int val;
   list* next;
}node;


void add_node(node** start, int val); //adaugare nod in lista
void remove_node(node** start, int val); //stergere nod din lista
node* find_val(node** start, int val); //returneaza nodul in care se afla valoarea cautata
void init_list(node** start, int val); //initializeaza lista
void sort_list(node** start, int val, int (*compar)(const void*, const void*)); //sorteaza lista folosind ca comparator functia trimisa ca parametru
void insert(node** start, int val, node* next_node); //insereaza un nod cu valoarea data inaintea unui nod al listei
```

Curs3-ex4: Rescrieți funcția factorial astfel încât să nu fie recursivă.

```
#include <stdio.h>
long fact(int n)
{
	long val = 1;
	for (int i = 1; i <= n; i++)
	{
		val = val * i;
	}
```

```
    return val;
}
int main()
{
    printf("%ld", fact(9));
}
```

Curs4-ex3: Determinați Θ(f(n)) pentru următoarele secvențe de cod.

j: 1, 2, 4, 8, ... cât timp $j > n \Rightarrow$

$\Rightarrow 2^k \leq n \Rightarrow k \leq \log_2 n \Rightarrow \log_2 n + 1$, când $j = 1$

$\Rightarrow$ Nr. iterații: $n(\log_2 n + 1) \Rightarrow O(n \log n)$

f. num = 0;

for (i=1; i<=n; i*=2) // $\log_2(n) + 1$

for (j=1; j<=n; j++) // n

num++;

$\Rightarrow$ Nr iterații: $n(\log_2 n + 1) \Rightarrow O(n \log n)$

j: 1, 2, 4, 8, ... cât timp $j > n \Rightarrow$

$\Rightarrow 2^k \leq n \Rightarrow k \leq \log_2 n \Rightarrow \log_2 n + 1$, când $j = 1$

$\Rightarrow$ Nr. iterații: $n(\log_2 n + 1) \Rightarrow O(n \log n)$

f. num = 0;

for (i=1; i<=n; i*=2) // $\log_2(n) + 1$

for (j=1; j<=n; j++) // n

num++;

$\Rightarrow$ Nr iterații: $n(\log_2 n + 1) \Rightarrow O(n \log n)$

Curs5-ex2: Modificați algoritmul pentru sortarea prin inserție astfel încât sortarea să se facă de la sfârșitul tabloului spre început. Ordinea va rămâne tot crescătoare.

```
#include <stdio.h>

#include <stdlib.h>


//Modificați algoritmul pentru sortarea prin inserție astfel încât sortarea să se facă de la

//sfârșitul tabloului spre început.Ordinea va rămâne tot crescătoare


void generate_array(long* arr, long n)

{

	for (long i = 0; i < n; i++)

	{

		arr[i] = n - i;

	}

}

void free_array(long* arr, long n)

{

	free(arr);

}

void print_array(long* arr, long n)

{

	for (long i = 0; i < n; i++)

	{

		printf("%ld ", arr[i]);

	}

	printf("\n");
```

```
}
void insert_sort(long* arr, long n) //sortarea modificata
{
	long i = 0, j = 0, to_insert = 0;
	for (i = n - 2; i >= 0; i--)
	{
		to_insert = arr[i];
		j = i;
		while (j < n - 1 && to_insert > arr[j + 1])
		{
			arr[j] = arr[j + 1];
			j++;
		}
		arr[j] = to_insert;
	}

}

int main()
{
	long n;
	printf("Type n: ");
	scanf_s("%ld", &n);
	long* arr = (long*)malloc(sizeof(long) * n);
	if (arr == NULL)
	{
```

```
            perror("Malloc error");
            exit(-1);
        }

        generate_array(arr, n);
        print_array(arr, n);
        insert_sort(arr, n);
        print_array(arr, n);
        free_array(arr, n);
    return 0;
}
```

Curs6-ex2: Pornind de la implementarea algoritmului Quicksort din acest curs, dezvoltați o serie de implementări care să fie cât mai eficiente din punct de vedere alt timpului de rulare și a memoriei utilizate considerând:

◦ Diferite variante de alegere a pivotului

◦ Ineficiența pentru un număr mic de elemente: în apelul recursiv, nu apelați tot quicksort() pe un tablou cu mai puțin de L elemente (unde valoarea lui L o alegeți dvs.), ci apelați sortarea prin inserție sub această limită (în rest se va apela quicksort în mod obișnuit)

◦ Ineficiența pentru cazul în care numărul de elemente egale este semnificativ: implementați o variantă care împarte tabloul în trei partiții în loc de două (prima conține elementele mai mici decât pivotul, a doua elementele egale cu pivotul și a treia, elementele mai mari).

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

/* Pornind de la implementarea algoritmului Quicksort din acest curs, dezvoltați o serie de
implementări care să fie cât mai eficiente din punct de vedere alt timpului de rulare și a
```

memoriei utilizate considerănd :

◦ Diferite variante de alegere a pivotului

◦ Ineficiența pentru un număr mic de elemente : în apelul recursiv, nu apelați tot quicksort() pe

un tablou cu mai puțin de L elemente(unde valoarea lui L o alegeți dvs.), ci apelați sortarea

prin inserție sub această limită(în rest se va apela quicksort în mod obișnuit)

◦ Ineficiența pentru cazul în care numărul de elemente egale este semnificativ : implementați o

variantă care împarte tabloul în trei partiții în loc de două(prima conține elementele mai mici

decât pivotul, a doua elementele egale cu pivotul și a treia, elementele mai mari)*/

```
void swap(int* e1, int* e2)
{
   int aux = *e1;
   *e1 = *e2;
   *e2 = aux;
}

int partition(int arr[], int left, int right)
{
   //int pivot = arr[right]; //pivotul e ultimul; ineficient cand array-ul este deja sortat
   int pivot = arr[rand() % (right - left + 1) + left]; //pivotul e ales aleator
   //int pivot = arr[left + (right - left) / 2]; //pivotul e elementul de la mijloc

   int i = left - 1;
   int j = right + 1;
```

```
  while (true)
  {
    do
    {
      i++;
    } while (arr[i] < pivot);

    do
    {
      j--;
    } while (arr[j] > pivot);

    if (i > j)
      return j;

    swap(&arr[i], &arr[j]);

  }
}

void partition_in_3(int arr[], int left, int right, int *m1, int *m2)
{
  //partitionarea in 3 segmente, separate de m1 si m2, care delimiteaza al doilea segment ce contine elementele
  //egale cu pivotul; de la left la m1-1 sunt elementele mai mici decat pivot, si de la m2+1 la right sunt
```

```
    //elementele mai mari decat pivotul

    int pivot = arr[right];
    int i = left;
    *m1 = left, *m2 = right;

    while (i < right)
    {
        if (arr[i] < pivot)
        {
            swap(&arr[*m1], &arr[i]);
            (*m1)++;
        }

        else if (arr[i] > pivot)
        {
            swap(&arr[*m2], &arr[i]);
            (*m2)--;
        }

        else i++;
    }
}

void quicksort(int arr[], int left, int right)
{
```

```
    if (left < right)
    {
        int pivot = partition(arr, left, right);

        quicksort(arr, left, pivot);
        quicksort(arr, pivot + 1, right);
    }

}

void first_quicksort(int arr[], int left, int right, int *m1, int *m2)
{
    partition_in_3(arr, left, right, m1, m2);

    quicksort(arr, left, *m1 - 1);
    quicksort(arr, *m2 + 1, right);
}

void print_arr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
void insert_sort(int* arr, int start, int end)
{
   int i = start, j = start, to_insert = 0;

   for (i = start + 1; i < end; i++)
   {
      if (arr[i - 1] > arr[i])
      {
         to_insert = arr[i];

         j = i;

         while (j > start && to_insert < arr[j - 1])
         {
            arr[j] = arr[j - 1];
            j--;
         }

         arr[j] = to_insert;
      }
   }
}

void quicksort_min(int arr[], int left, int right, int min_nb)
{
```

```
  //quicksort cu implementare de insert sort cand avem un nr de elemente mai mic decat min_nb dat ca parametru


  if (left < right) {
    int pivot = partition(arr, left, right);


    if (pivot - left < min_nb)
      insert_sort(arr, left, pivot + 1);
    else
      quicksort_min(arr, left, pivot, min_nb);


    if (right - pivot - 1 < min_nb)
      quicksort_min(arr, pivot + 1, right, min_nb);
    else
      insert_sort(arr, pivot + 1, right + 1);
  }
}


int main()
{
  int arr[9];
  int m1 = 0;
  int m2 = 8;
  arr[0] = 5;
  arr[1] = 4;
  arr[2] = 1;
```

```
    arr[3] = 3;
    arr[4] = 2;
    arr[5] = 4;
    arr[6] = 10;
    arr[7] = 0;
    arr[8] = 4;
    //first_quicksort(arr, 0, 8, &m1, &m2);
    quicksort_min(arr, 0, 8, 4);
    print_arr(arr, 9);
        return 0;
}
```

Curs7-ex2: Scrieți un program care inserează structuri într-un tablou, le caută și le șterge folosind una din metodele de hashing studiate în acest curs.

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

#define SIZE 10

//Scrieți un program care inserează structuri într-un tablou, le caută și le șterge folosind una
//din metodele de hashing studiate în acest curs -> tehnica dispersiei deschise

typedef struct
{
    int val;
```

```
}INFO;

typedef struct list
{
   INFO item;
   list* next;
}element;

int get_key(int val)
{
   return val % SIZE;
}

element* init(element* start)
{
   if (start == NULL)
      return start;

   element* cursor = NULL;

   while (start != NULL)
   {
      cursor = start;
      start = start->next;
      free(cursor);
```

```
    }
    return start;
}

element* create_node(INFO info)
{
    element* new_node = (element*)malloc(sizeof(element));

    if (new_node == NULL)
    {
        printf("Error malloc new node\n");
        exit(-1);
    }

    new_node->item.val = info.val;
    new_node->next = NULL;

    return new_node;

}

element* add_node(element* start, INFO info)
{
    if (start == NULL)
    {
        start = create_node(info);
```

```
    return start;
  }

  element* cursor = start;

  while (cursor->next != NULL)
  {
    cursor = cursor->next;
  }

  element* new_node = create_node(info);
  new_node->next = cursor->next;
  cursor->next = new_node;

  return start;
}

element* add_to_h(INFO info, element** h)
{
  int index = get_key(info.val);

  h[index] = add_node(h[index], info);

  return h[index];

}
```

```
void print_list(element* start)
{
   if (start == NULL)
   {
      printf("Empty\n");
      return;
   }

   element* cursor = NULL;

   for (cursor = start; cursor != NULL; cursor = cursor->next)
   {
      printf("%d ", cursor->item.val);
   }
   printf("\n");
}



void print_h(element** h)
{
   for (int i = 0; i < SIZE; i++)
   {
      print_list(h[i]);
   }
   printf("\n");
```

```
}

void find_element(int val, element* start)
{
   if (start == NULL)
   {
      printf("item not found\n");
      return;
   }

   element* cursor = NULL;

   for (cursor = start; cursor != NULL; cursor = cursor->next)
   {
      if (cursor->item.val == val)
      {
         printf("item found\n");
         return;
      }
   }

   printf("item not found\n");
}

void find_in_h(int val, element** h)
{
```

```
    int index = get_key(val);
    find_element(val, h[index]);
}

element* remove_from_list(element* start, int val)
{
    if (start == NULL)
    {
        return NULL;
    }

    element* cursor = start->next;
    element* prev_cursor = start;
    element* aux = NULL;

    if (start->item.val == val)
    {
        aux = start;
        start = start->next;
        free(aux);
        return start;
    }

    while (cursor != NULL)
    {
        if (cursor->item.val == val)
```

```
      {
         prev_cursor->next = cursor->next;
         free(cursor);
         return start;
      }
      prev_cursor = cursor;
      cursor = cursor->next;
   }

   return start;
}



void remove_from_h(int val, element** h)
{
   int index = get_key(val);
   h[index] = remove_from_list(h[index], val);

}

int main()
{
   element* h[SIZE];

   for (int i = 0; i < SIZE; i++)
   {
```

```
    h[i] = NULL;
}

INFO info1, info2, info3, info4, info5;
info1.val = 321;
info2.val = 11;
info3.val = 34445;
info4.val = 5;
info5.val = 777;

add_to_h(info1, h);
add_to_h(info2, h);
add_to_h(info3, h);
add_to_h(info4, h);
add_to_h(info5, h);
print_h(h);

//find_in_h(21, h);
remove_from_h(321, h);
remove_from_h(8, h);
remove_from_h(11, h);
remove_from_h(5, h);
remove_from_h(34445, h);

print_h(h);
```

```
    return 0;
}
```

Curs8-ex1: Să se implementeze o funcție de ștergere a unui nod cu o cheie dată dintr-o listă simplu înlănțuită. Funcția trebuie să ia în considerare toate cazurile de ștergere.

```
node* remove(node* start, const char* new_name)
{
	if (start == NULL)
	{
		return NULL;
	}

	node* cursor = start->next;
	node* prev_cursor = start;
	node* aux = NULL;

	if (strcmp(start->name, new_name) == 0)
	{
		aux = start;
		start = start->next;
		free(aux);
		return start;
```

```
        }

        while (cursor != NULL)
        {
                if (strcmp(cursor->name, new_name) == 0)
                {
                        prev_cursor->next = cursor->next;
                        free(cursor);
                        return start;
                }
                prev_cursor = cursor;
                cursor = cursor->next;
        }

        return start;
}
```

Curs9-ex1: Scrieți o funcție recursivă care numără nodurile dintr-o listă simplu înlănțuită.

```
#include <stdio.h>
int count_nodes(node* cursor)
{
        if (cursor == NULL)
                return 0;
        else
                return 1 + count_nodes((cursor->next));
}
```

Curs10-ex2: Fie un tablou de întregi cu valori unice A[0...n-1]. Să se verifice dacă există un index i astfel încât A[i]=i, folosind un algoritm de tip divide et impera care are o complexitate O(log n).

```
#include <stdio.h>


int find_index(int a[], int left, int right)
{
	if (left > right)
	{
		return 0;
	}
	int middle = left + (right - left) / 2;
	if (a[middle] == middle) return 1;
	int ret = find_index(a, left, middle - 1);
	if (ret == 0)
		ret = find_index(a, middle + 1, right);


	return ret;
}
int main()
{
	int a[10] = { 6,22,3,4,5,5,7,8,9,8 };
	printf("%d", find_index(a, 0, 9));


	return 0;
}
```