SEMINAR 7
FUNCŢII CU NUMĂR VARIABIL DE ARGUMENTE. PROGRAMARE GENERICĂ. RECURSIVITATE

1. Scrieţi o funcţie cu număr variabil de parametri care să concateneze mai multe şiruri de caractere.

Rezolvare:
Prima variantă de rezolvare constă în implementarea unei funcţii cu număr variabil de parametri având ca parametru fix numărul şirurilor de caractere care vor fi concatenate:

```
char* concat1(int n,...)
```

Un exemplu de apel corect al acestei funcţii este `s = concat1(3, "Test", " ", "apel")`, în urmă căruia se va obţine şirul `s = "Test apel"`. Atenţie, înainte de terminarea programului, zona de memorie alocată pentru şirul de caractere `s` trebuie eliberată!

Pentru a putea să alocăm dinamic un şir `aux` care va fi obţinut prin concatenarea şirurilor date, este necesar să cunoaştem lungimea sa (notată cu `lt`). Astfel, vom parcurge o dată lista parametrilor variabili `siruri_1` pentru a determina suma lungimilor tuturor şirurilor respective, adică valoarea `lt`. Deoarece o astfel de listă poate fi parcursă o singură dată, iar noi va trebui să o parcurgem încă o dată pentru a concatena şirurile respective, vom salva o copie a sa în lista `siruri_2`.

```
char* concat1(int n, ...)
{
    char *aux, *sir;
    int i, lt;
    va_list siruri_1, siruri_2;

    va_start(siruri_1, n);
    va_copy(siruri_2, siruri_1);

    lt = 0;
    for(i = 0; i < n; i++)
    {
        sir = va_arg(siruri_1, char*);
        lt = lt + strlen(sir);
    }

    va_end(siruri_1);

    aux = (char *)malloc(lt + 1);
    strcpy(aux, "");
```

```
    for(i = 0; i < n; i++)
    {
        sir = va_arg(siruri_2, char*);
        strcat(aux, sir);
    }

    va_end(siruri_2);

    return aux;
}
```

A doua variantă de rezolvare constă în implementarea unei funcţii care nu mai are ca parametru numărul şirurilor care vor fi concatenate, ci sfârşitul listei parametrilor variabili va fi marcat prin pointerul NULL:

char* concat2(char *s,...)

În acest caz, parametrul fix `s` reprezintă chiar primul şir dintre cele pe care trebuie să le concatenăm. Un exemplu de apel corect al acestei funcţii este `s = concat2("Alt", " ", "apel", NULL)`, în urmă căruia se va obţine şirul `s = "Alt apel"`.

În plus, pentru a elimina dubla parcurgere a listei parametrilor variabili, vom realoca dinamic şirul `aux` în care construim şirul rezultat prin concatenarea şirurilor date.

```
char* concat2(char *s, ...)
{
    char *aux, *sir;
    va_list siruri;

    if(s == NULL)
    {
        aux = (char *)malloc(1);
        strcpy(aux, "");
        return aux;
    }

    va_start(siruri, s);

    aux = (char *)malloc(strlen(s) + 1);
    strcpy(aux, s);

    while((sir = va_arg(siruri, char*)) != NULL)
    {
        aux = realloc(aux, strlen(aux) + strlen(sir) + 1);
        strcat(aux, sir);
    }

    va_end(siruri);

    return aux;
}
```

2. Scrieţi o funcţie cu număr variabil de parametri care să caute un cuvânt dat în mai multe fişiere text. Funcţia va scrie într-un fişier text, al cărui nume este dat ca parametru, numerele de ordine ale liniilor pe care apare cuvântul respectiv în fiecare fişier text dat.

Rezolvare:
Un posibil antet al funcţiei cerute este următorul:

```
void cautare_cuvant(char *cuv, char *nfout,...)
```

Parametrul fix `cuv` reprezintă cuvântul pe care îl vom căuta, parametrul fix `nfout` reprezintă numele fişierului text în care se vor scrie rezultatele căutării, iar lista parametrilor variabili va conţine numele fişierelor text în care va fi căutat cuvântul respectiv (lista se va termina cu pointerul `NULL`). De exemplu, prin apelul `cautare_cuvant("lac","rez.txt", "eminescu.txt","bacovia.txt",NULL)` se va căuta cuvântul `lac` în fişierele text `eminescu.txt` şi `bacovia.txt`, iar rezultatul căutării va fi scris în fişierul text `rez.txt`.

În cadrul funcţiei, preluăm, pe rând, numele fiecărui fişier text din lista parametrilor variabili în şirul de caractere `nfin` şi efectuăm următoarele operaţii:

- deschidem fişierul curent folosind variabila `fin`;
- parcurgem fişierul linie cu linie şi căutăm cuvântul dat pe linia curentă (şirul de caractere `linie`);
- dacă linia curentă conţine cuvântul căutat `cuv`, atunci scriem numărul de ordine `lcrt` al liniei curente în fişierul de ieşire `fout`, marcăm faptul că am găsit cuvântul în fişierul curent (folosind variabila `g`) şi întrerupem căutarea pe linia curentă (de ce?);
- după ce am terminat de parcurs fişierul curent, verificăm dacă nu am găsit cuvântul căutat în el (adică testăm dacă `g` este `0` sau nu) şi, în caz afirmativ, scriem un mesaj corespunzător;
- închidem fişierul curent.

```
void cautare_cuvant(char *cuv, char *nfout, ...)
{
    FILE *fin, *fout;
    char *nfin, *p;
    int lcrt, g;
    char linie[1001];
    va_list fisiere;

    va_start(fisiere, nfout);

    fout = fopen(nfout, "w");

    while((nfin = va_arg(fisiere, char*)) != NULL)
    {
        fin = fopen(nfin, "r");
        fprintf(fout, "Fisierul %s\n", nfin);
        fprintf(fout, "Liniile: ");

        lcrt = 1;
        g = 0;
        while(fgets(linie, 1001, fin) != NULL)
```

```
        {
           p = strtok(linie, " :;,.?!\n");
           while(p != NULL)
           {
               if(strcmp(p, cuv) == 0)
               {
                   fprintf(fout, "%d ", lcrt);
                   g = 1;
                   break;
               }
               p = strtok(NULL, " :;,.?!\n");
           }
           lcrt++;
        }

        if(g == 0)
            fprintf(fout, "-");

        fprintf(fout, "\n\n");

        fclose(fin);
    }

    va_end(fisiere);

    fclose(fout);
}
```

Pentru a nu complica prea mult codul, nu am analizat posibilele erori care pot să apară: lista parametrilor variabili este vidă (deci nu se indică nici un fişier text în care să se efectueze căutarea), un anumit fişier text în care trebuie efectuată căutarea nu există, fişierul de ieşire nu poate fi creat etc. Modificaţi codul sursă astfel încât erorile indicate mai sus să fie tratate!

3. Scrieţi o funcţie cu număr variabil de parametri care să sorteze mai multe mai multe tablouri unidimensionale formate din numere întregi pe baza unui criteriu comun.

Rezolvare:

Vom sorta tablourile date utilizând funcţia `qsort` din biblioteca `stdlib.h`, deci criteriul comun de sortare trebuie implementat printr-o funcţie comparator de tipul `int (*cmp)(const void*,const void*)`.

Pentru a simplifica antetul funcţiei cerute, putem defini mai întâi un tip de date `Comparator` prin `typedef int (*Comparator)(const void*,const void*)`. Astfel, un posibil antet al funcţiei este următorul:

```
void sortare(Comparator cmp,...)
```

Lista parametrilor variabili va conţine numele tablourilor pe care dorim să le sortăm, împreună cu numărul lor de elemente, iar sfârşitul listei va fi marcat prin valoarea `0`. De exemplu, pentru a sorta crescător tablourile `a` şi `b`, având `na` şi `nb` elemente, vom proceda astfel:

- definim o funcţie comparator `cmpCrescator`:

```
int cmpCrescator(const void* a, const void* b)
{
      int va = *(int *)a;
      int vb = *(int *)b;

      if (va < vb) return -1;
      if (va > vb) return +1;
      return 0;
}
```

- apelăm funcţia prin `sortare(cmpCrescator,na,a,nb,b,0)`.

Codul sursă complet al funcţiei este următorul:

```
void sortare(Comparator cmp, ...)
{
    int i, n, *v;
    va_list vectori;

    va_start(vectori, cmp);

    do
    {
        n = va_arg(vectori, int);
        if(n != 0)
        {
            v = va_arg(vectori, int*);
            qsort(v, n, sizeof(int), cmp);
        }
    }
    while(n != 0);

    va_end(vectori);
}
```

Consideraţi faptul că funcţia de mai sus ar putea fi modificată astfel încât să poată fi utilizată pentru sortarea tablourilor având elemente de un anumit tip de date primitiv indicat printr-un parametru? Argumentaţi!

4. Scrieţi o funcţie cu număr variabil de parametri care să returneze maximul dintre mai multe numere întregi. Folosind apeluri utile ale funcţiei definite anterior, scrieţi o funcţie care să afişeze mesajul DA în cazul în care 4 numere întregi a, b, c şi d îndeplinesc condiţia $a \leq b \leq c \leq d$ sau mesajul NU în caz contrar.

Rezolvare:

Considerând faptul că parametrul fix n reprezintă numărul parametrilor variabili ai funcţiei, vom extrage pe rând câte un număr întreg din lista parametrilor variabili şi vom calcula maximul cerut:

```
int maxim(int n,...)
{
    int max, val, i;
    va_list valori;

    va_start(valori, n);

    max = INT_MIN;
    for(i = 0; i < n; i++)
    {
        val = va_arg(valori, int);
        if(val > max)
            max = val;
    }

    va_end(valori);

    return max;
}
```

Pentru a verifica faptul că 4 numere întregi îndeplinesc condiţia cerută, trebuie să verificăm dacă sunt îndeplinite simultan următoarele condiţii:

- maximul dintre toate cele patru numere este al patrulea număr;
- maximul dintre primele trei numere este al treilea număr;
- maximul dintre primele două numere este al doilea număr.

```
void testare(int a, int b, int c, int d)
{
    if(maxim(4,a,b,c,d) == d && maxim(3,a,b,c) == c && maxim(2,a,b) == b)
        printf("DA");
    else
        printf("NU");
}
```

5. Scrieţi o funcţie generică care să calculeze câte elemente dintr-un tablou unidimensional sunt egale cu o valoare dată.

Rezolvare:

Algoritmul de rezolvare a problemei este unul simplu, respectiv se parcurge tabloul element cu element şi se incrementează cu 1 valoarea unei variabile de tip contor în momentul în care elementul curent este egal cu valoarea dată. Din punct de vedere al implementării, problema este mai dificilă, deoarece trebuie comparate două valori ale căror tipuri de date nu sunt cunoscute, ele fiind accesate prin intermediul unor pointeri generici (de tip `void*`). Din acest motiv, pointerii respectivi nu pot fi dereferenţiaţi şi, în consecinţă, nu poate fi utilizat direct operatorul de testare a egalităţii (==). Totuşi, în cazul funcţiilor generice care manipulează tablouri, trebuie să cunoaştem dimensiunea în octeţi a unui element pentru a putea simula aritmetica pointerilor (pointerii generici nu au aritmetică!). Astfel, putem folosi funcţia `memcmp` pentru a compara la nivel de octet conţinutul unor zone de memorie ale căror adrese şi dimensiuni în octeţi sunt cunoscute, în ipoteza în care ele au acelaşi tip de date:

```
int num1(const void *x, const void *tablou, int nr_elem, int dim_elem)
{
    int i, nrx;
    char *p;

    nrx = 0;
    for(i = 0; i < nr_elem; i++)
    {
        p = (char *)tablou + i*dim_elem;
        if(memcmp(p, x, dim_elem) == 0)
            nrx++;
    }

    return nrx;
}
```

Considerând `v` un tablou format din `n` numere întregi şi `x` o variabilă de tip întreg, putem apela funcţia `num1` de mai sus astfel: `nrap = num1(&x, v, n, sizeof(int))`.

Totuşi, funcţia `num1` de mai sus nu va furniza rezultatul corect în cazul în care elementele tabloului sunt pointeri (de exemplu, în cazul unor tablouri de şiruri de caractere), deoarece se vor compara adresele între ele, ci nu valorile memorate la adresele respective! Din acest motiv, trebuie să utilizăm pentru compararea valorilor o funcţie comparator, pe care o vom transmite funcţiei generice utilizând un pointer către o funcţie (aşa cum am procedat, de exemplu, în cazul funcţiei de bibliotecă `qsort`):

```
int num2(const void *x, const void *tablou, int nr_elem, int dim_elem,
         int (*cmp)(const void*, const void*))
{
    int i, nrx;
    char *p;

    nrx = 0;
    for(i = 0; i < nr_elem; i++)
    {
        p = (char *)tablou + i*dim_elem;
        if(cmp(p, x) == 0)
            nrx++;
    }

    return nrx;
}
```

Considerând `v` un tablou format din `n` numere întregi şi `x` o variabilă de tip întreg, putem apela funcţia `num2` prin `nrap = num2(&x, v, n, sizeof(int), cmpIntregi)`, unde funcţia comparator `cmpIntregi` poate fi definită astfel:

```
int cmpIntregi(const void *a , const void *b)
{
    int va = *(int *)a;
    int vb = *(int *)b;
```

```
    if(va < vb) return -1;
    if(va > vb) return 1;
    return 0;
}
```

Dacă `v` ar fi un tablou format din `n` şiruri de caractere şi `s` un şir de caractere, putem apela funcţia `num2` prin `nrap = num2(s, v, n, sizeof(char*), cmpSiruri)`, unde funcţia comparator `cmpSiruri` poate fi definită astfel:

```
int cmpSiruri(const void *s , const void *t)
{
    return strcmp(*(char **)s , (char *)t);
}
```

Observaţi cu atenţie modurile diferite în care au fost dereferenţiaţi cei 2 pointeri către şiruri de caractere! De ce a fost necesar acest lucru?

6. Scrieţi o funcţie generică care să furnizeze adresa unui element cu valoare maximă dintr-un tablou unidimensional. Criteriul de comparaţie dintre elementele tabloului va fi implementat printr-o funcţie comparator de tipul celei utilizate în funcţia `qsort` din `stdlib.h` (de exemplu, funcţiile `cmpIntregi` şi `cmpSiruri` de mai sus).

Rezolvare:
În cadrul funcţiei vom implementa algoritmul standard pentru determinarea valorii maxime dintr-un tablou unidimensional, însă nu vom lucra cu valorile elementelor din tablou, ci cu adresele lor. Astfel, variabila de tip pointer `pcrt` va conţine adresa elementului curent din tablou, iar variabila `pmax` va conţine adresa elementului cu valoare maximă găsit până la momentul respectiv. Pentru compararea valorilor de la adresele `pcrt` şi `pmax` vom folosi funcţia comparator, transmisă funcţiei generice printr-un pointer către o funcţie.

```
void* maxim(const void *tablou, int nr_elem, int dim_elem,
            int (*cmp)(const void*, const void *))
{
    int i;
    char *pcrt, *pmax;

    if(nr_elem == 0)
        return NULL;

    pmax = (char *)tablou;
    for(i = 1; i < nr_elem; i++)
    {
        pcrt = (char *)tablou + i*dim_elem;
        if(cmp(pcrt, pmax) > 0)
            pmax = pcrt;
    }

    return pmax;
}
```

Considerând un tablou unidimensional `a` format din `n` numere întregi şi `pmax` o variabilă de tip pointer la `int`, putem afişa toate poziţiile pe care apare valoarea maximă `vmax` în tabloul `a` astfel:

```
pmax = maxim(a, n, sizeof(int), cmpIntregi);
vmax = *pmax;

while(pmax != NULL && *pmax == vmax)
{
      printf("Valoarea maxima %d apare pe pozitia %d!\n", vmax, pmax - a);
      pmax = maxim(pmax+1, n-(pmax-a)-1, sizeof(int), cmpIntregi);
}
```

7. Scrieţi o funcţie generică care să implementeze algoritmul de căutare binară pentru un tablou unidimensional sortat crescător în raport cu o anumită relaţie de ordine. Funcţia va returna adresa unui element din tablou care este egal cu valoarea căutată sau pointerul `NULL` dacă valoarea căutată nu se găseşte în tablou.

Rezolvare:
În cadrul funcţiei vom implementa algoritmul de căutare binară, utilizând direct adresele elementelor din tablou. Pentru compararea valorii căutate (transmisă funcţiei generice tot printr-o adresă) şi valoarea elementului curent din tablou vom folosi funcţia comparator `cmp`, în care trebuie să ţinem cont de aceeaşi relaţie de ordine în raport cu care sunt sortate crescător elementele tabloului!

```
void* cbin(const void *x, const void *tablou, int nr_elem, int dim_elem,
           int (*cmp)(const void* , const void *))
{
    char *pcrt;
    int st, dr, mij;

    st = 0;
    dr = nr_elem - 1;
    while(st <= dr)
    {
        mij = st + (dr-st)/2;
        pcrt = (char *)tablou + mij*dim_elem;
        if(cmp(x, pcrt) == 0)
            return pcrt;
        if(cmp(x, pcrt) < 0)
            dr = mij - 1;
        else
            st = mij + 1;
    }

    return NULL;
}
```

Modificaţi funcţia de mai sus astfel încât să se verifice faptul că elementele tabloului sunt sortate crescător în raport cu relaţia de ordine implementată în funcţia comparator `cmp`, înainte de a utiliza algoritmul de căutare binară!

8. Scrieţi o funcţie generică de căutare cu următorul antet:

```
void* cautare(const void *x, const void *t, int n, int d,
              int (*cmp)(const void*,const void *))
```

Funcţia trebuie să returneze un pointer generic către primul element din tabloul unidimensional `t` care este *strict mai mare* decât valoarea `x` (în raport cu un anumit criteriu de comparaţie) sau pointerul `NULL` dacă nu există nici un astfel de element în tablou. Tabloul unidimensional `t` este format din `n` elemente, fiecare având dimensiunea `d` octeţi. Funcţia comparator `cmp` implementează criteriul de comparaţie astfel: returnează valoarea -1 dacă primul argument este *strict mai mic* decât al doilea, 0 dacă argumentele sunt *egale* sau 1 dacă primul argument este *strict mai mare* decât al doilea. Folosind apeluri utile ale funcţiei `cautare`, scrieţi o funcţie care verifică dacă un tabloul unidimensional de numere întregi este sortat descrescător sau nu.

Rezolvare:

În cadrul funcţiei vom compara valoarea elementului curent `pcrt` din tablou cu valoarea căutată `x`, folosind funcţia comparator. Dacă valoarea elementului curent este strict mai mare, atunci vom returna adresa sa. Funcţia va returna valoarea `NULL` în cazul în care, după parcurgerea completă a tabloului, nu a fost găsit niciun element cu proprietatea cerută.

```
void* cautare(const void *x, const void *t, int n, int d,
              int (*cmp)(const void* ,const void *))
{
    int i;
    char *pcrt;

    for(i = 0; i < n; i++)
    {
        pcrt = (char *)t + i*d;
        if(cmp(pcrt, x) > 0)
            return pcrt;
    }
    return NULL;
}
```

Pentru a verifica dacă un tablou unidimensional de numere întregi este sortat descrescător sau nu vom testa, pentru fiecare element din tablou, faptul că în dreapta sa nu există nici un element strict mai mare decât el.

```
int verif_sortat(int v[], int n)
{
    int i;

    for(i = 0; i < n-1; i++)
        if(cautare(v+i, v+i+1, n-i-1, sizeof(int), cmpIntregi) != NULL)
            return 0;
    return 1;
}
```

Observaţi faptul că în funcţia `verif_sortat` nu am efectuat conversii explicite ale pointerilor generici spre pointeri către întregi! De ce?

9. Scrieţi o funcţie recursivă care să calculeze cel mai mare divizor comun a două numere întregi.

Rezolvare:
O variantă de implementare a unei astfel funcţii se bazează pe următoarea relaţie de recurenţă, obţinută din algoritmul lui Euclid:

$$cmmdc(a,b) = \begin{cases} a, \text{dacă } b = 0 \\ cmmdc(b, a\%b), \text{dacă } b \neq 0 \end{cases}$$

Implementarea propriu-zisă a funcţiei este următoarea:

```
int cmmdc(int a, int b)
{
    if(b == 0) return a;
    return cmmdc(b, a%b);
}
```

Dacă `a=b=0`, funcţia va întoarce valoarea `0`. Este corect acest rezultat?

10. Scrieţi o funcţie recursivă care să calculeze suma valorilor strict pozitive dintr-un tablou unidimensional de numere întregi.

Rezolvare:
O variantă de implementare constă în simularea recursivă a unei instrucţiuni `for`, folosind parametrul `i` transmis prin valoare, pentru a parcurge tabloul `v` de la stânga spre dreapta:

```
int sumasp(int v[], int n, int i)
{
    if(i == n) return 0;
    if(v[i] > 0) return v[i] + sumasp(v, n, i+1);
    return sumasp(v, n, i+1);
}
```

Funcţia recursivă de mai sus se va apela prin `sumasp(v, n, 0)`.

În funcţia precedentă parametrul `n` este necesar pentru a testa dacă au fost parcurse toate elementele tabloului `v` sau nu. Dacă am parcurge tabloul în sens invers, de la dreapta spre stânga, atunci parcurgerea se termină în momentul în care se ajunge pe poziţia `0`. Astfel, se poate observa faptul că parametrul `n` este suficient pentru parcurgerea tabloului, respectiv `v[n-1]` va reţine elementul curent:

```
int sumasp(int v[], int n)
{
    if(n == 0) return 0;
```

```
    if(v[n-1] > 0) return v[n-1] + sumasp(v, n-1);
    return sumasp(v, n-1);
}
```

În acest caz funcţia recursivă se va apela prin `sumasp(v, n)`.

11. Scrieţi o funcţie recursivă care să afişeze o matrice `t` de tip `m×n` formată din numere întregi.

Rezolvare:
O prima variantă de implementare constă în simularea recursivă a două instrucţiuni `for`, folosind parametrii `i` şi `j`, pentru a parcurge matricea pe linii şi pe coloane:

```
void afisare(int t[10][10], int m, int n, int i, int j)
{
    if(i < m)
        if(j < n)
        {
            printf("%d ", t[i][j]);
            afisare(t, m, n, i, j+1);
        }
        else
        {
            printf("\n");
            afisare(t, m, n, i+1, 0);
        }
}
```

Funcţia recursivă de mai sus se va apela prin `afisare(t, m, n, 0, 0)`.

Într-un mod asemănător cu cel prezentat în problema anterioară, pentru a elimina cei doi parametrii `i` şi `j`, trebuie să parcurgem matricea invers, respectiv de la ultima linie spre prima, iar fiecare linie o vom parcurge de la dreapta spre stânga. Mecanismul de stivă asociat recursivităţii va permite într-un mod implicit inversarea ordinii de afişare a elementelor, deci matricea va fi afişată corect. Cei 2 parametrii `m` şi `n` (reprezentând numărul de linii şi numărul de coloane din matrice) îi vom înlocui cu parametrii `lcrt` şi `ccrt`, reprezentând linia curenta, respectiv coloana curentă. O problemă în acest caz o reprezintă faptul că, după parcurgerea unei linii, trebuie să reluăm parcurgerea liniei aflate deasupra sa de la ultima coloană spre prima, dar nu mai cunoaştem numărul de elemente de pe o linie (valoarea parametrului `ccrt` fiind modificată). Un artificiu care permite păstrarea numărului de elemente de pe o linie constă în utilizarea unei variabile locale statice `ncol` care să contorizeze numărul de elemente parcurse pe ultima linie.

```
void afisare(int t[10][10], int lcrt, int ccrt)
{
    static int ncol, pa = 1;

    if(pa == 1)
    {
        ncol = ccrt + 1;
        pa = 0;
```

```
    }

    if(lcrt >= 0)
    {
        if(ccrt >= 0)
        {
            afisare(t, lcrt, ccrt-1);
            printf("%d ", t[lcrt][ccrt]);
        }
        else
        {
            afisare(t, lcrt-1, ncol-1);
            printf("\n");
        }
    }
}
```

Variabila `pa` (tot statică!) a fost utilizată pentru a restricționa salvarea numărului de coloane în variabila `ncol` doar în primul apel al funcției.

12. Din numărul 4 se poate obține orice număr natural nenul aplicând în mod repetat următoarele operații:

a) împărțirea numărului la 2;
b) adăugarea cifrei 4 la sfârșitul numărului;
c) adăugarea cifrei 0 la sfârșitul numărului.

De exemplu, numărul 101 se poate obține prin șirul de operații $4 \rightarrow 40 \rightarrow 404 \rightarrow 202 \rightarrow 101$, iar numărul 133 prin șirul de operații $4 \rightarrow 2 \rightarrow 24 \rightarrow 12 \rightarrow 6 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 84 \rightarrow 42 \rightarrow 424 \rightarrow 212 \rightarrow 106 \rightarrow 1064 \rightarrow 532 \rightarrow 266 \rightarrow 133$.

Scrieți o funcție recursivă care să afișeze șirul de operații prin care se poate obține un număr natural nenul `n` din numărul 4.

Rezolvare:

Pentru a afișa șirul de operații prin care se poate obține un număr natural nenul `n` din numărul 4, vom aplica asupra lui `n` operațiile inverse operațiilor date:

a') înmulțirea numărului cu 2;
b') eliminarea ultimei cifre, dacă aceasta este 4;
c') eliminarea ultimei cifre, dacă aceasta este 0.

```
void numar4r(int n)
{
    if(n != 4)
    {
        if(n % 10 == 0 || n%10 == 4)
            numar4r(n/10);
        else
            numar4r(n*2);

        printf(" -> %d", n);
```

```
    }
    else
        printf("4");
}
```

O variantă iterativă de implementare a acestui algoritm constă în simularea recursivității folosind o stivă `st`:

```
void numar4i(int n)
{
    int st[1001], k;

    k = 0;
    st[0] = n;
    while(st[k] != 4)
    {
        if(st[k]%10 == 0 || st[k]%10 == 4)
            st[++k] = st[k-1]/10;
        else
            st[++k] = st[k-1]*2;
    }

    printf("\n");
    while(k > 0)
        printf("%d -> ", st[k--]);
    printf("%d\n", st[0]);
}
```

Puteți implementa o funcție iterativă care să nu utilizeze tablouri sau alte structuri de date auxiliare?

13. Scrieți o funcție recursivă care să calculeze câte litere mici conține un șir de caractere.

Rezolvare:

Vom testa faptul că litera curentă din șirul `s` (adică prima literă) este o literă mică folosind funcția `islower` din biblioteca `ctype.h`, după care vom reapela funcția pentru restul șirului (care începe cu adresa `s+1`):

```
int nrLitereMici(char *s)
{
    if(*s == '\0')
        return 0;
    if(islower(*s) != 0)
        return 1 + nrLitereMici(s+1);
    return nrLitereMici(s+1);
}
```

Știind că rezultatul evaluării unei expresii de tip logic este 0 sau 1, putem rescrie funcția astfel:

```
int nrLitereMici(char *s)
```

```
{
    if(*s == '\0')
        return 0;
    return (islower(*s) != 0) + nrLitereMici(s+1);
}
```

14. În 1970, John McCarthy, unul dintre întemeietorii inteligenţei artificiale, a definit următoarea funcţie recursivă, numită *funcţia McCarthy 91*, pentru a fi utilizată în verificarea formală a corectitudinii algoritmilor:

$$M(n) = \begin{cases} n - 10, \text{dacă } n > 100 \\ M(M(n + 11)), \text{dacă } n \leq 100 \end{cases}$$

Implementaţi funcţia McCarthy 91 atât sub forma unei funcţii recursive, cât şi sub forma unei funcţii iterative.

Rezolvare:
Implementarea recursivă a funcţiei McCarthy 91 se obţine direct din expresia funcţiei:

```
int mcc91(int n)
{
    if(n > 100)
        return n - 10;
    return mcc91(mcc91(n + 11));
}
```

Implementarea iterativă a funcţiei McCarthy 91 se poate realiza folosind o stivă pentru a simula mecanismul recursivităţii:

```
int mcc91(int n)
{
    int st[1000], k;

    k = 0;
    st[0] = n;
    while(k >= 0)
        if (st[k] <= 100)
        {
            k++;
            st[k] = st[k-1] + 11;
        }
        else
        {
            k--;
            if (k>=0) st[k] = st[k+1] - 10;
        }

    return st[0] - 10;
}
```

O implementare foarte simplă şi eficientă a funcţiei McCarthy 91 se obţine dacă se observă (şi se demonstrează matematic!) faptul că ea este echivalentă cu următoarea funcţiei:

$$M(n) = \begin{cases} n - 10, \text{dacă } n > 100 \\ 91, \text{dacă } n \leq 100 \end{cases}$$

Probleme propuse

1. Scrieţi o funcţie cu număr variabil de parametri care să concateneze mai multe şiruri de caractere, parcurgând o singură dată lista parametrilor variabili.

2. Scrieţi o funcţie cu număr variabil de parametri care să determine prefixul comun de lungime maximă al mai multor şiruri de caractere.

3. Scrieţi o funcţie cu număr variabil de parametri care să returneze numărul de apariţii ale unui număr întreg dat într-un şir de numere întregi. Folosind apeluri utile ale funcţiei definite anterior, scrieţi o funcţie care verifică dacă 4 numere întregi sunt distincte sau nu.

4. Scrieţi o funcţie cu număr variabil de parametri care să creeze un şir de caractere din mai multe caractere date.

5. Scrieţi o funcţie cu număr variabil de parametri care să creeze un tablou de numere întregi din mai multe numere întregi. Atenţie, funcţia trebuie să furnizeze şi lungimea tabloului!

6. Scrieţi o funcţie generică care să verifice dacă două tablouri unidimensionale sunt egale sau nu. Două tablouri unidimensionale sunt egale dacă au acelaşi număr de elemente şi elementele aflate pe aceeaşi poziţie sunt egale.

7. Scrieţi o funcţie generică de căutare care să returneze un pointer generic către prima apariţie a unei valori x în tabloul unidimensional t format din n elemente, fiecare element având dimensiunea d octeţi, sau pointerul $NULL$ dacă valoarea x nu se găseşte în tablou. Scrieţi o funcţie care să afişeze, folosind apeluri utile ale funcţiei anterioare, toate poziţiile pe care apare un număr întreg x într-un tabloul v format din n numere întregi sau mesajul "$Valoare\ inexistent$ă!" dacă numărul x nu se găseşte în tablou.

8. Scrieţi o funcţie generică care să înlocuiască într-un tablou unidimensional toate apariţiile unei valori x cu o valoare y. Atenţie la modul în care valoarea y este copiată în locul valorii x!

9. Scrieţi o funcţie recursivă care să calculeze valoarea maximă dintr-un tablou unidimensional de numere întregi.

10. Scrieţi o funcţie recursivă care să calculeze de câte ori apare o literă dată într-un şir de caractere.

11. Scrieţi o funcţie recursivă care să verifice dacă două şiruri de caractere sunt anagrame sau nu.

12. Scrieţi o funcţie recursivă care să verifice dacă un număr natural este palindrom sau nu.