Seminar 9

Exemple la metoda backtracking

Probleme propuse:

1. **Să se genereze toate permutările unei mulțimi cu n elemente.**
2. **Să se genereze toate aranjamentele dintr-o mulțime cu n elemente, luate cîte k.**
3. **Să se genereze toate combinările dintr-o mulțime cu n elemente, luate cîte k.**
4. **Problema celor 8 (n) regine. Se cere să se așeze 8 regine pe o tablă de șah astfel încît să nu existe două regine care să se atace. Trebuie găsite toate posibilitățile de așezare a celor 8 regine pe tabla de șah.**
5. **Plata unei sume (cu/fără bancnotă unitate). Fie n tipuri de bancnote, cu valorile nominale $t_i,\ i = \overline{1,n}$. Din fiecare tip este disponibilă cantitatea $nr_i,\ i = \overline{1,n}$. Să se determine toate modalitățile în care se poate plăti o sumă S folosind aceste bancnote.**

Rezolvări:

1. Să se genereze toate permutările unei mulțimi cu n elemente.

Problema generării permutărilor unei mulțimi se exprimă foarte ușor în termenii metodei backtracking, dacă se reprezintă mulțimea liniar și se asociază fiecărui element o valoare numerică, în funcție de poziția ocupată. În continuare fiecare element va fi reprezentat de o valoare numerică $x_i,\ i = \overline{1,n}$ cu semnificația următoare: x_i este poziția elementului i în permutare; pentru mulțimea inițială $x_i = i$. Cu aceste considerații, spațiul soluțiilor este definit prin intermediul mulțimilor identice S_i, $x_i \in S_i = \{1, 2, \dots, n\}$, $i = \overline{1,n}$. Aceste mulțimi pot fi exprimate ca progresii aritmetice cu elementul inițial $a_i = 1$, rația $r_i = 1$ și ultimul element $s_i = n$.

Deoarece într-o permutare nu pot fi două elemente pe aceeași poziție, valoarea atribuită unui element x_i este acceptabilă dacă este diferită de valorile atribuite elementelor anteriore din soluția parțială construită: $x_i \neq x_j,\ j = \overline{1,i-1}$. Condiția este suficientă, astfel încît nu e necesară verificarea vreunei condiții suplimentare după construirea unei soluții. La găsirea unei soluții (permutări) aceasta este numărată (și afișată pe ecran, în exemplul următor de implementare).

```
#include <stdio.h>
#include<malloc.h>
// Verifica daca valoarea elementului i este acceptabila(daca v[i] este diferit de toate
elementele din v)
int posibil(int* v, int i)
{
	for (int j = 0; j < i; j++)
		if (v[i] == v[j])  return 0;
	return 1;
}
// Afisare solutie pe ecran
// I: numarul solutiei (num), dimensiunea permutarii (nr), permutarea (v)
void retine_solutia(int num, int nr, int* v)
{
	int i;
	printf("\nSolutia numarul %d:", num);
	for (i = 0; i < nr; i++)  printf("%2d ", v[i]);
```

```
}

// Genereaza permutari de n elemente (1..n)
// I: n
// E: numar permutari
int permutari(int n)
{
	int* p, i, ok, nr;
	p = (int *)malloc(n * sizeof(int));		//vectorul solutie
	nr = 0;						//numar solutii
	i = 0;
	p[0] = 0;					//prima valoare(=1) minus ratia(=1)
	while (i >= 0)				//cat timp nu am ajuns la configuratia finala
	{
		ok = 0;
		while (p[i] < n && ok == 0) //alege urmatoarea valoare acceptabila pentru
p[i]
		{
			p[i]++;				//urmatoarea valoare pentru p[i]
			ok = posibil(p, i);	//este acceptabila?
		}
		if (ok == 0)				//impas, revenire
			i--;
		else
			if (i == n - 1)		//configuratie solutie
				retine_solutia(++nr, n, p);
			else
				p[++i] = 0;		//prima valoare(=1) minus ratia(=1)
	}
	free(p);

	return nr;
}

//Permutari recursiv
//I: dimensiune (n), pas curent (i), sol. partiala curenta (x),  nr. crt. sol. (nr)
//E: numar solutii (nr)
int permutari_r(int n, int i, int* x, int nr)
{
	if (i == n)
		retine_solutia(++nr, n, x);
	else
		for (int j = 1; j <= n; j++)
		{
			x[i] = j;
			if (posibil(x, i))
				nr = permutari_r(n, i + 1, x, nr);
		}
	return nr;
}
void main() {
	int n, nr;
	printf("n="); scanf("%d", &n);

	int* x = (int *)malloc(n * sizeof(int));

	nr = permutari(n); //apel pentru varianta iterativa
	printf("\n\nNumarul total de permutari este: %d\n\n", nr);

	//nr = permutari_r(n, 0, x, 0); //apel pentru varianta recursiva
	//printf("\n\nNumarul total de permutari este: %d\n\n", nr);
```

```
	free(x);
}
```

2. Să se genereze toate aranjamentele dintr-o mulțime cu n elemente, luate cîte k.

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

// Verifica daca valoarea elementului i este acceptabila(daca v[i] este diferit de toate elementele din v)
int posibil(int* v, int i)
{
	for (int j = 0; j < i; j++)
		if (v[i] == v[j])  return 0;
	return 1;
}

// Afisare solutie pe ecran
// I: numarul solutiei (num), dimensiunea (k), solutia (v)
void retine_solutia(int num, int k, int* v)
{
	int i;
	printf("\nSolutia numarul %d: ", num);
	for (i = 0; i < k; i++)
		printf("%2d ", v[i]);
}

//I: dimensiune (n), numarul maxim k, pas curent (i),
//sol. partiala curenta (x), nr. crt. sol. (nr)
//E: numar solutii (nr)
int aranjamente(int n, int k, int i, int* x, int nr)
{
	int j;
	if (i == k)
		retine_solutia(++nr, k, x);
	else
		for (j = 1; j <= n; j++)
		{
			x[i] = j;
			if (posibil(x, i))
				nr = aranjamente(n, k, i + 1, x, nr);
		}
	return nr;
}

void main()
{
	int n, k, nr;
	printf("n="); scanf("%d", &n);
	printf("k="); scanf("%d", &k);
```

```
	int* x = (int*)malloc(n * sizeof(int));

	if (k > 0 && k <= n) { nr = aranjamente(n, k, 0, x, 0); printf("\nNumarul de aranjamente este: %d\n", nr); }
	else printf("Nu exista solutii");

	free(x);
}
```

```
n=3
k=2

Solutia numarul 1:  1  2
Solutia numarul 2:  1  3
Solutia numarul 3:  2  1
Solutia numarul 4:  2  3
Solutia numarul 5:  3  1
Solutia numarul 6:  3  2
Numarul de aranjamente este: 6
```

3. Să se genereze toate combinările dintr-o mulțime cu n elemente, luate cîte k.

```
#include <stdio.h>
#include<malloc.h>
void init(int *n, int *k, int **x)
{
	printf("n="); scanf("%d", n);
	printf("k="); scanf("%d", k);
	*x = (int*)malloc(*n * sizeof(int));
	for (int i = 0; i < *n; i++)
		*(*x + i) = 0;
}

//afiseaza solutia
void afiseaza(int k, int *x)
{
	for (int i = 0; i < k; i++)
		printf("%d ", x[i]);
	printf("\n");
}

//verifica daca poate fi adaugat numarul la solutie
int posibil(int p, int *x)
{
	if (p > 0 && x[p] <= x[p - 1])
		return 0;
	return 1;
}

//subprogramul recursiv de generare combinari de n luate cite k
void combinari(int p, int n, int k, int *x)
{
	int pval;
	for (pval = 1; pval <= n; pval++)
	{
		x[p] = pval;
		if (posibil(p, x))
		{
			if (p == k - 1)
				afiseaza(k, x);
			else
```

```
                        combinari(p + 1, n, k, x);
                }
        }
}

void main()
{
        int n, k, *x;
        init(&n, &k, &x);
        if (k > 0 && k <= n) combinari(0, n, k, x);
        else printf("Nu exista solutii");
        free(x);
}
```

```
n=4
k=3
1 2 3
1 2 4
1 3 4
2 3 4
```

4. Problema celor 8 (n) regine. Se cere să se așeze 8 regine pe o tablă de șah astfel încît să nu existe două regine care să se atace. Trebuie găsite toate posibilitățile de așezare a celor 8 regine pe tabla de șah.

Problema se poate extinde la n regine, pe o tablă de dimensiuni $n \times n, n > 2$, de aceea în continuare se va folosi n ca dimensiune a problemei.

Considerând liniile și coloanele tablei de șah numerotate de la 1 la n (8), începînd din colțul din stînga sus, poziția unei regine pe tabla de șah este determinată de o pereche de coordonate de forma $P_i = (linia_i, coloana_i)$, $i = \overline{1,n}$, deci soluția problemei este o mulțime cu n astfel de poziții. Ținînd cont că reginele nu trebuie să se atace, rezultă că pe fiecare linie trebuie să se afle o singură regină și numai una. Ca urmare, în mulțimea soluție va exista câte un element și numai unul cu linia 1, respectiv linia 2 etc. adică $linia_i = i,\ i = \overline{1,n}$. Putem considera mulțimea soluție ca o mulțime ordonată, cu n elemente de forma $(i, coloana_i)$, $i = \overline{1,n}$. Pentru reprezentarea acestei mulțimi este necesară doar reținerea coordonatei coloană. În aceste condiții, putem exprima soluția problemei ca o mulțime $X = \{x_i |\ i = \overline{1,n}\}$, unde x_i este coloana pe care se află regina de pe linia i. Altfel spus, poziția reginei i este (i, x_i), $i = \overline{1,n}$.

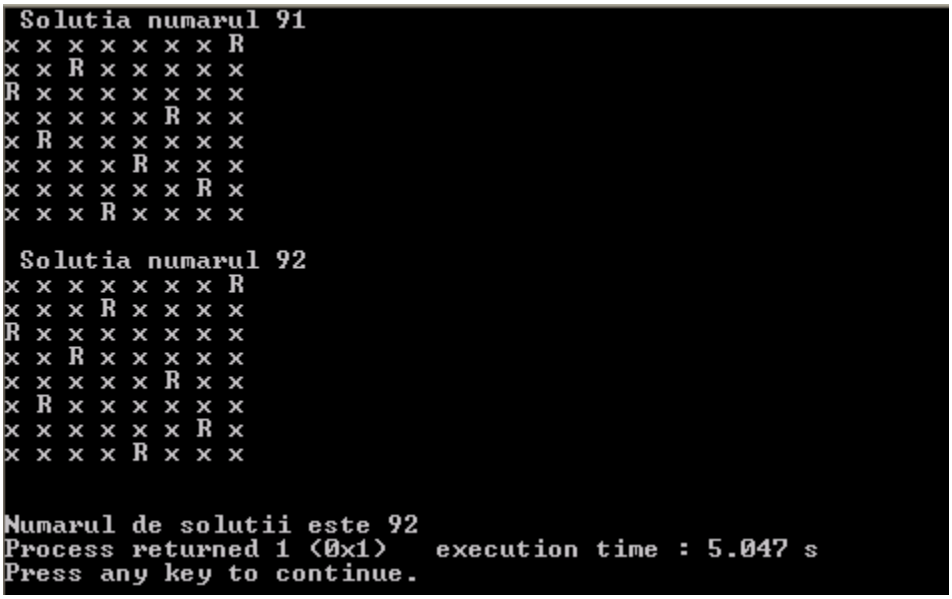
Deoarece o regină se poate afla pe una din cele n coloane ale tablei de șah, rezultă că elementele x_i pot lua valori din mulțimea $S_i = \{1, 2, \dots, n\}$, $i = \overline{1,n}$. Mulțimile S_i se pot exprima ca progresii aritmetice cu elementul inițial $a_i = 1$, rația $r_i = 1$ și ultimul element $s_i = n$.

Pentru a evita confuziile legate de utilizarea indicilor masivelor în C, se poate folosi un vector cu un element în plus (lungime $n+1$), în care elementul cu indicele 0 rămîne nefolosit (risipa de resurse este insignifiantă în acest caz).

- *posibil* – poziția aleasă pentru regina curentă, (i, x_i), este acceptabilă dacă nu este atacată de nici una din reginele anterior plasate pe tablă. Această poziție nu trebuie să se afle pe aceeași linie sau coloană cu una din reginele anterioare și nici pe diagonală cu vreuna din ele. Deoarece fiecare regină e plasată pe o linie nouă, cu siguranță prima parte a condiției este îndeplinită. Pentru a verifica dacă nu se află pe aceeași coloană cu o regină plasată anterior, se compară coloana curentă x_i cu coloanele atribuite reginelor anterioare x_j, $j = \overline{1, i-1}$. Se acceptă valoarea curentă dacă $x_i \neq x_j$, $j = \overline{1, i-1}$. Două regine se află pe diagonală dacă distanțele pe verticală și orizontală între ele sînt egale; se acceptă valoarea curentă dacă $|i-j| \neq |x_i - x_j|$, $j = \overline{1, i-1}$.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include<malloc.h>
#include<math.h>

//x[i] -> coloana; i->linia
// Conditia de continuare: reginele i si j nu se ataca daca nu sunt pe aceeasi coloana:
x[i] != x[j]
// - distantele pe verticala si orizontala sunt diferite: abs(i - j) != abs(x[i] - x[j]))

// I: solutia partiala (x), numarul de elemente (i)
// E: 1 daca e acceptabila, 0 daca nu e acceptabila

int posibil(int *x, int i)
{
	for (int j = 1; j < i; j++)
		if (x[i] == x[j] || abs(i - j) == abs(x[i] - x[j]))
			return 0;
	return 1;
}

// Retine o configuratie solutie (afisare)
// I: nr. solutie (nr), nr regine (n), vector solutie
// E: -

void retine_solutia(int nr, int n, int* x)
{
	int i, j;
	printf("\n Solutia numarul %d\n", nr);
	for (i = 1; i <= n; i++)
	{
		for (j = 1; j <= n; j++)
			printf("%c ", j == x[i] ? 'R' : 'x'); //x[i] contine indexul
coloanei...i-indexul liniei; j-indexul coloanei
		printf("\n");
	}
}

// I: numar regine/dimensiune tabla (n)
// E: numar solutii
int regine(int n)
{
	int nr;   int* x;   int i, ok;
	x = (int*)malloc((n + 1) * sizeof(int));        //vectorul solutie
	nr = 0;                    //numar solutii
```

```
	i = 1;
	x[1] = 0;                           //prima valoare minus ratia
	while (i > 0)                      //cit timp nu am ajuns la configuratia finala
	{
		ok = 0;
		while (x[i] < n && ok == 0) //alege urmatoarea valoare acceptabila pentru
x[i]
		{
			x[i]++;                          //urmatoarea valoare pentru x[i]
			ok = posibil(x, i);         //este acceptabila?
		}
		if (ok == 0)   i--;
		else
			if (i == n)    retine_solutia(++nr, n, x);
			else
				x[++i] = 0;       //prima valoare minus ratia
	}
	free(x);

	return nr;
}

// I: numar dame (n), element curent (i), vector solutie (x),
// numar solutii (nr)
// E: nr. solutii
int regine_r(int n, int i, int* x, int nr)
{
	int j;
	if (i == n + 1)
		retine_solutia(++nr, n, x);
	else
		for (j = 1; j <= n; j++)
		{
			x[i] = j;
			if (posibil(x, i))
				nr = regine_r(n, i + 1, x, nr);
		}
	return nr;
}

void main() {
	int n;
	printf("n="); scanf("%d", &n);

	int* x = (int*)malloc((n + 1) * sizeof(int));
	int nr = 0;

	printf("\n\nNumarul de solutii este %d ", regine_r(n, 1, x, nr));
	//printf("\n\nNumarul de solutii este %d ", regine(n));

	free(x);
}
```



5. Plata unei sume (cu/fără bancnotă unitate). Fie $\boldsymbol{n}$ tipuri de bancnote, cu valorile nominale $\boldsymbol{t_i}$, $\boldsymbol{i = \overline{1,n}}$. Din fiecare tip este disponibilă cantitatea $\boldsymbol{nr_i}$, $\boldsymbol{i = \overline{1,n}}$. Să se determine toate modalitățile în care se poate plăti o sumă $\boldsymbol{S}$ folosind aceste bancnote.

Pentru rezolvare, se reprezintă tipurile de bancnote disponibile într-un vector (prin valorile lor nominale) iar în alt vector cantitățile disponibile din tipurile respective (asigurând corespondența tip-cantitate prin folosirea aceleiași poziții în cei doi vectori). Soluția problemei poate fi exprimată sub forma unui vector X în care elementul x_i indică numărul de bancnote de tipul t_i utilizate: $x_i \in S_i = \{0,1,2,\dots,nr_i\}$, $i = \overline{1,n}$. Mulțimile S_i se pot exprima ca progresii aritmetice cu elementul inițial $a_i = 0$, rația $r_i = 1$ și ultimul element $s_i = nr_i$.

O valoare atribuită elementului x_i este acceptabilă dacă prin adăugarea bancnotelor respective (cu valoarea $x_i * t_i$) la bancnotele anterior alese nu se depășește suma de plată. Această condiție nu este suficientă; după ce se aleg și acceptă valori pentru toate elementele x_i, valoarea tuturor bancnotelor selectate ($\sum_{i=1}^{n} x_i * t_i$) trebuie să fie egală cu suma de plată.

Pentru a evita calcularea acestei sume la fiecare pas, se poate păstra într-o variabilă *suma curentă plătită*, care se ajustează de fiecare dată când se atribuie o valoare nouă unui element al soluției. Inițial aceasta este zero. Algoritmul funcționează și dacă între tipurile de bancnote disponibile nu se află și bancnota unitate.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include<malloc.h>

//I: solutia (x), pasul curent (i), numarul de tipuri (n), suma de plata (s)
//   suma curenta (c)
//E: 1 daca valoarea este acceptabila, 0 daca nu e acceptabila
```

```
int posibil(int* x, int i, int n, int s, int c)
{
	int rez;
	if (i == n - 1)
		rez = (s == c) ? 1 : 0;
	else
		rez = (s >= c) ? 1 : 0;
	return rez;
}


//I: tipuri (t), solutia (x), numarul de tipuri (n), numarul de solutii (ns)
void retine(int* t, int* x, int n, int ns)
{
	int i, s;   s = 0;
	printf("\n\nSolutia numarul %d", ns);
	for (i = 0; i < n; i++)  printf("\n %3d * %2d = %4d, suma=%4d", t[i], x[i], t[i] *
x[i], s += t[i] * x[i]);
}

//Plata unei sume
//I: suma, suma curenta (crt), tipuri (t), cantitati (nr), numar de tipuri (n)
//   numarul de solutii gasite (ns), pasul curent (i), solutia (x)
//E: numarul de solutii
int plata(int suma, int crt, int* t, int* nr, int n, int ns, int i, int* x)
{
	int j;
	if (i == n)
		retine(t, x, n, ++ns);
	else
	{
		for (j = 0; j <= nr[i]; j++)
		{
			x[i] = j;
			crt += t[i] * j;
			if (posibil(x, i, n, suma, crt))
				ns = plata(suma, crt, t, nr, n, ns, i + 1, x);
			crt -= t[i] * j;
		}
	}
	return ns;
}

void main() {
	int i, n, x[20], suma;
	 int t[20]={500, 200, 100, 50, 10, 5, 1};
	 int nr[20]={3, 6, 2, 5, 4, 3, 7};
	 n=7;
	//int t[20], nr[20];
	//printf("introduceti numarul de bacnote disponibile:"); scanf("%d", &n);
	//printf("Introduceti tipurile de bacnote si numarul de bacnote din fiecare
tip\n");
	//for (i = 0; i < n; i++)
	//{
	//	printf("t[%d]=", i + 1); scanf("%d", &t[i]);
	//	printf("nr[%d]=", i + 1); scanf("%d", &nr[i]);
	//}
	printf("Introduceti suma de plata:"); scanf("%d", &suma);
	printf("\nNumarul de solutii este %d:", plata(suma, 0, t, nr, n, 0, 0, x));

}
```

```
introduceti numarul de bacnote disponibile:7
Introduceti tipurile de bacnote si numarul de bacnote din fiecare tip
t[1]=500
nr[1]=1
t[2]=200
nr[2]=2
t[3]=100
nr[3]=3
t[4]=50
nr[4]=5
t[5]=10
nr[5]=3
t[6]=5
nr[6]=2
t[7]=1
nr[7]=8
Introduceti suma de plata:36


Solutia numarul 1
 500 *  0 =     0, suma=   0
 200 *  0 =     0, suma=   0
 100 *  0 =     0, suma=   0
  50 *  0 =     0, suma=   0
  10 *  2 =    20, suma=  20
   5 *  2 =    10, suma=  30
   1 *  6 =     6, suma=  36

Solutia numarul 2
 500 *  0 =     0, suma=   0
 200 *  0 =     0, suma=   0
 100 *  0 =     0, suma=   0
  50 *  0 =     0, suma=   0
  10 *  3 =    30, suma=  30
   5 *  0 =     0, suma=  30
   1 *  6 =     6, suma=  36

Solutia numarul 3
 500 *  0 =     0, suma=   0
 200 *  0 =     0, suma=   0
 100 *  0 =     0, suma=   0
  50 *  0 =     0, suma=   0
  10 *  3 =    30, suma=  30
   5 *  1 =     5, suma=  35
   1 *  1 =     1, suma=  36
Numarul de solutii este 3:
Process returned 27 (0x1B)   execution time : 46.547 s
Press any key to continue.
```