Programarea Calculatoarelor Cursul 9

Recursivitate

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Obiective

- Ce este o funcție recursivă?
- De ce este util să proiectăm funcții recursive?

Recursivitate

- O funcție este recursivă dacă se auto-apelează
- Definiția funcției include din nou definiția ei
- Trebuie să avem un caz special unde nu avem auto-apel
  - altfel avem apel recursiv infinit
- Este strâns legată de inducția matematică
- Orice funcție care se poate implementa cu bucle se poate reformula în mod recursiv
  - limbajele de programare funcționale care nu folosesc bucle sunt Turing-complete = pot fi folosite pentru simularea unei mașini Turing

De ce recursivitate?

- O formulă recursivă exprimă regula generală în spatele unui fenomen complex
- De cele mai multe ori permite o descriere foarte succintă
- Oferă o modalitate de rezolvare a problemelor dificile
  - tehnici de rezolvare ale problemelor: backtracking, divide et impera, programare dinamică folosesc de obicei recursivitate într-o formă sau alta
- Chiar dacă implementarea rezolvării sub formă de funcție recursivă este ineficientă ne ajută mult la rezolvare
- O funcție recursivă se poate transforma în una iterativă foarte ușor

De ce nu?

- Este o altă modalitate de gândire mai mult ierarhică decât liniară, poate neobișnuită pentru unii
- În limbajul C executarea funcțiilor recursive implică folosirea stivei de execuție pentru salvarea variabilelor temporare și a adreselor de revenire
  - există limbaje care evită acest lucru - eng. tail call optimization
- Dacă recursivitatea este prea adâncă se umplă stiva
- Dacă recalculăm rezultatul pe o subproblemă de mai multe ori fără să-l memorăm codul devine ineficient

Cum scriem o funcție recursivă

- Determinăm cum putem construi răspunsul dacă avem deja răspunsul la problemă mai mică (subproblemă)
  - presupunem că știm răspunsul la subproblemă și îl folosim
  - ne interesează doar cum combinăm rezultatele parțiale, nu cum le obținem
  - tratăm cu grijă cazurile simple
- Evităm
  - variabile globale sau cele statice
  - salvarea rezultatelor parțiale în argumentele trimise sau modificare lor
    - este în regulă dacă vrem să construim o soluție pe lângă numărarea soluțiilor
  - folosirea buclelor
  - adăugarea parametrilor adiționali

Recursivitate - exemplu simplu

- Fie *s(n)* suma primelor *n* numere naturale
- Se poate exprima recursiv ca

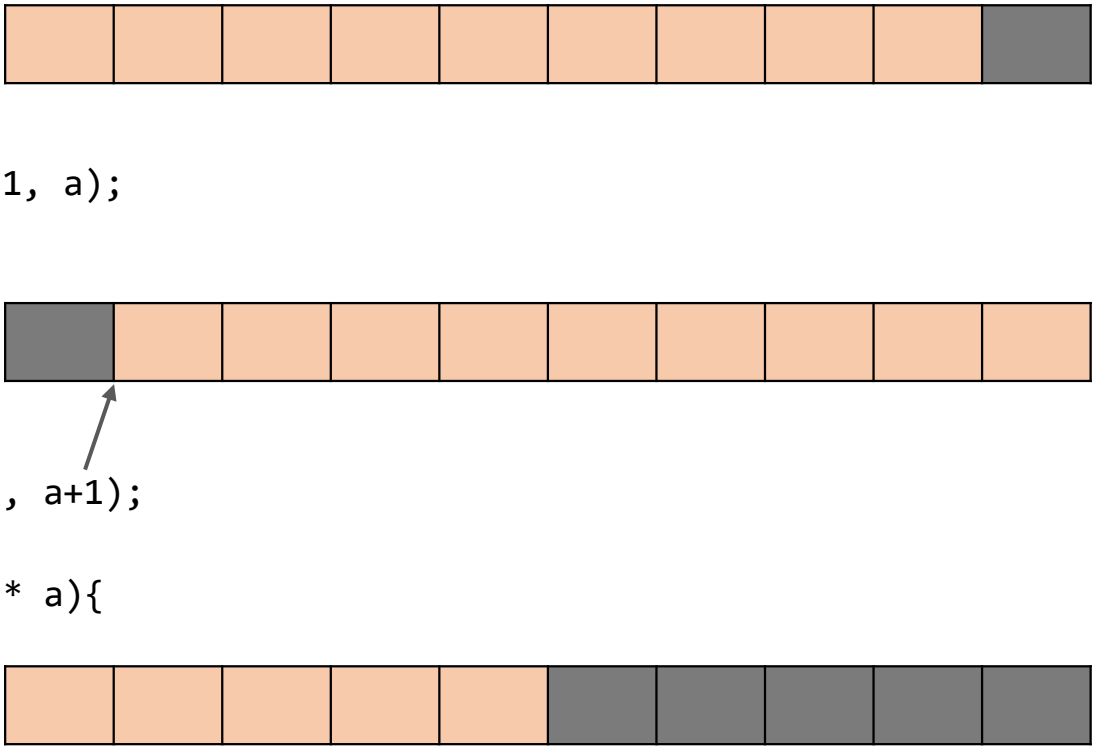
`s(n) = 0,` pentru *n<=0*

`s(n) = n + s(n-1),` altfel

- Se identifică cazul special când *n* este 0 sau negativ
- Observăm că argumentul funcției descrește la fiecare apel și se îndreaptă către cazul special

Recursivitate - exemplu simplu

```
int sum_last(int n, int* a){
    if (n==1)
      return a[n-1];
    else
      return a[n-1] + sum_last(n-1, a);
}
int sum_first(int n, int* a){
    if (n==1)
      return a[0];
    else
      return a[0] + sum_first(n-1, a+1);
}
int sum_mid(int from, int to, int* a){
    if (from == to)
      return a[from];
    else{
      int m = (from+to)/2;
      return sum_mid(from, m, a) + sum_mid(m+1, to, a);
    }
}
```



Recursivitate - exemplu simplu - main și apel

```c
#include <stdio.h>

int sum_last(int n, int* a);
int sum_first(int n, int* a);
int sum_mid(int from, int to, int* a);

int main() {
    int a[] = {1, 2, 3, 4};
    int n = sizeof(a)/sizeof(a[0]);
    printf("last  %d\n", sum_last(n, a));
    printf("first %d\n", sum_first(n, a));
    printf("mid   %d\n", sum_mid(0, n-1, a));

    return 0;
}
```

- anteturile funcțiilor definite anterior
- trimitem dimensiunea și pointer la primul element
- trimitem intervalul și pointer la primul element

- combinăm ultimul element cu restul
- combinăm primul element cu restul
- combinăm rezultatele pe cele două jumătăți

- atenție la indexare de la 0
- elementele sunt de la 0 la n-1

Recursivitate - exemplu complex

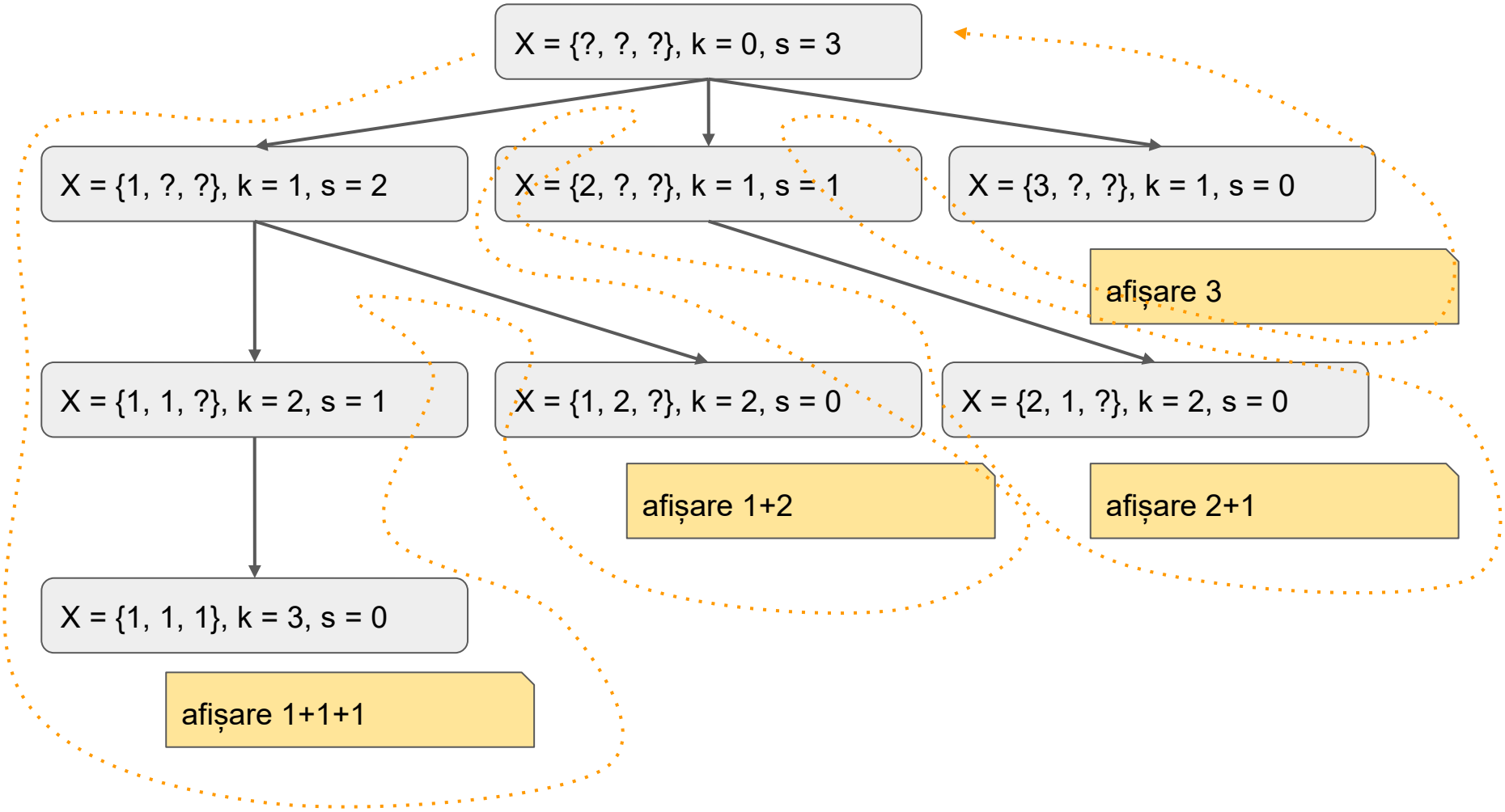
```
#include <stdio.h>
void compositions(int* X, int k, int s){
    if (s == 0){
        for(int i=0; i<k-1; i++)
            printf("%d + ", X[i]);
        printf("%d\n", X[k-1]);
        return;
    }
    for(int x = 1; x <= s; x++){
        X[k] = x;
        compositions(X, k+1, s-x);
    }
}
int main(){
    int n = 3;
    int X[n];
    compositions(X, 0, n);
    return 0;
}
```

- afișează toate compozițiile lui n
- toate modurile posibile de a scrie *n* ca o sumă de numere naturale
- în *X* stocăm termenii
- *k* este numărul de termeni
- *s* este ce a rămas din sumă
- când *s = 0* avem o soluție

- încercăm fiecare termen posibil de la 1 la *s*
- este backtracking recursiv

- *n <= 20*
- sunt 2^{n-1} compoziții
- apelăm cu tabloul alocat, *k = 0* și *s* inițial egal cu *n*

Compoziții - vizualizare - apel n = s = 3



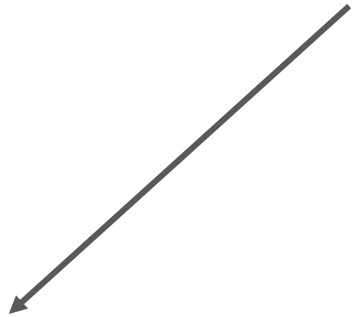
Stiva de execuție - revizitată

- Este o zonă de memorie pentru stocarea variabilelor locale, ale adreselor de revenire din funcții și ale argumentelor
- Este o stivă fiindcă la orice apel de funcție se introduc valori noi în vârful stivei și la revenire se extrage tot din vârful stivei
- Această zonă are dimensiuni reduse de câțiva MB
- Trebuie să ne asigurăm că nu depășim limita
  - să folosim variabile locale de dimensiune mică
  - să schimbăm dimensiunea stivei prin setări la linker

Recursivitate - exemplu revizitat

```c
#include <stdio.h>
void f(long long n){ //showbinary
     if (n > 1)
          f(n/2); //B
     printf("%d", n&1);
}
int main(){
     f(11); //A
     return 0;
}
```

- instrucțiunile care apar după apelul recursiv se vor executa în ordinea inversă a apelurilor din cauza stivei
- se afișează cifrele în binar ale numărului în ordine corectă
- stiva de execuție completată cu afișări pentru claritate



f apel 4 - n = 1,		afișare 1 &1 = 1
f apel 3 - n = 2,	apel f, revenire la B,	afișare 2 &1 = 0
f apel 2 - n = 5,	apel f, revenire la B,	afișare 5 &1 = 1
f apel 1 - n = 11,	apel f, revenire la B,	afișare 11&1 = 1
main, apel f, revenire la A		

Recursivitate - probleme cu string-uri

```
int strlen_rec(char* s){
    if (*s)
        return 1 + strlen_rec(s+1);
    else
        return 0;
}
```

- Se pot rezolva în mod recursiv prin mutare pointer-ului
- strlen parcurge șirul în mod inutil
- de cele mai multe ori nu este necesar să folosim funcții din string.h

```
int count_rec(char* s, char c){
    if (*s)
        return (*s == c) + count_rec(s+1, c);
    else
        return 0;
}
```

- Returnează de câte ori apare caracter c în string-ul s în mod pur recursiv (fără bucle)

Recursivitate – găsire prima apariție de substring

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

int strstr_rec(char* t, char* s, int matched) {
  if (*s == 0)
    return 0;
  if (*t == 0)
    return -1;
  if (*t == *s) {
    int r = strstr_rec(t + 1, s + 1, matched + 1);
    if (r >= 0)
      return r;
  } else {
    int r = strstr_rec(t + 1 - matched, s - matched, 0);
    if (r >= 0)
      return r + 1;
  }
  return -1;
}
```

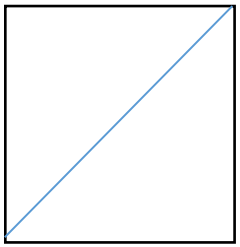
- am potrivit toate caracterele din *s*
- am ajuns la finalul lui *t* si nu am gasit o potrivire
- încercăm sa continuăm potrivirea
- revenim de la o potrivire incompletă

Etapele rezolvării unei probleme folosind recursivitate

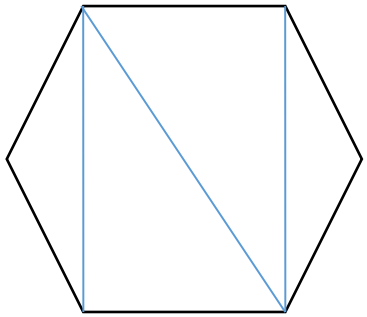
- Rezolvarea pentru un caz simplu (de bază)
- Rezolvarea pentru cazuri mici – opțional, dar recomandat
- Stabilirea relației de recurență
  - Studiem cum putem să reducem problema la o subproblemă
  - Observăm ce se întâmplă: la primul pas, la ultimul pas sau la mijloc
  - Cum putem să combinăm rezultatele de la subprobleme pentru a da răspunsul
- Implementare recursivă
  - Se adaugă memoizare (memorare)
- Implementare iterativă
- Formulă generală fără recursivitate

Triangulări n-gon

- Calculați numărul de moduri de a triangula un poligon convex cu *n* vârfuri, *3<=n<35*
  - În câte moduri se poate împărți în triunghiuri
  - Împărțirea se face de-a lungul diagonalelor
  - Diagonalele selectate pentru împărțire nu au voie să se intersecteze
  - Exemple de triangulări:
    - n = 4



n = 6

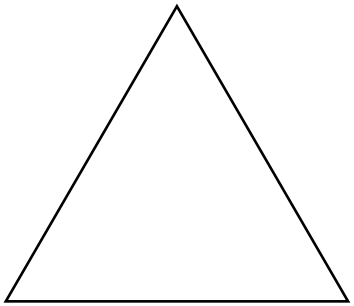


- Fie T(n) numărul de moduri

Triangulări n-gon

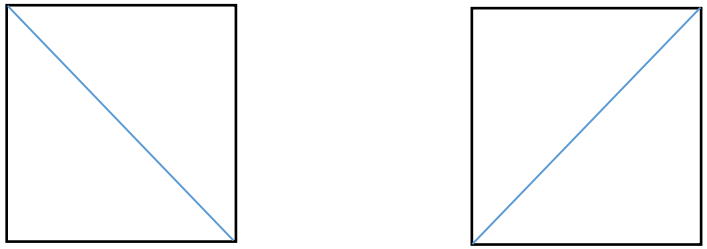
- Rezolvarea pentru un caz simplu
  - Pentru n=3
  - Un triunghi se poate triangula într-un singur mod

$$T(3) = 1$$

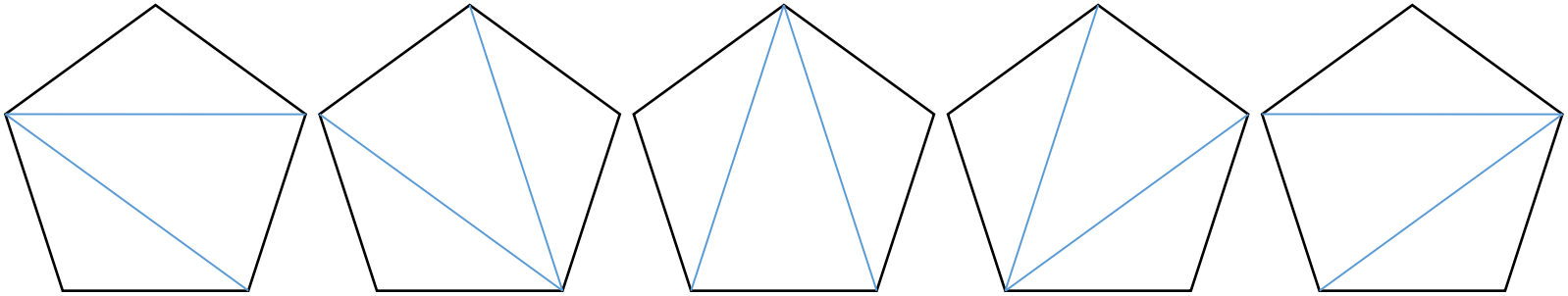


Triangulări n-gon

- Rezolvarea pentru cazuri mici
  - Pentru n=4, avem T(4) = 2

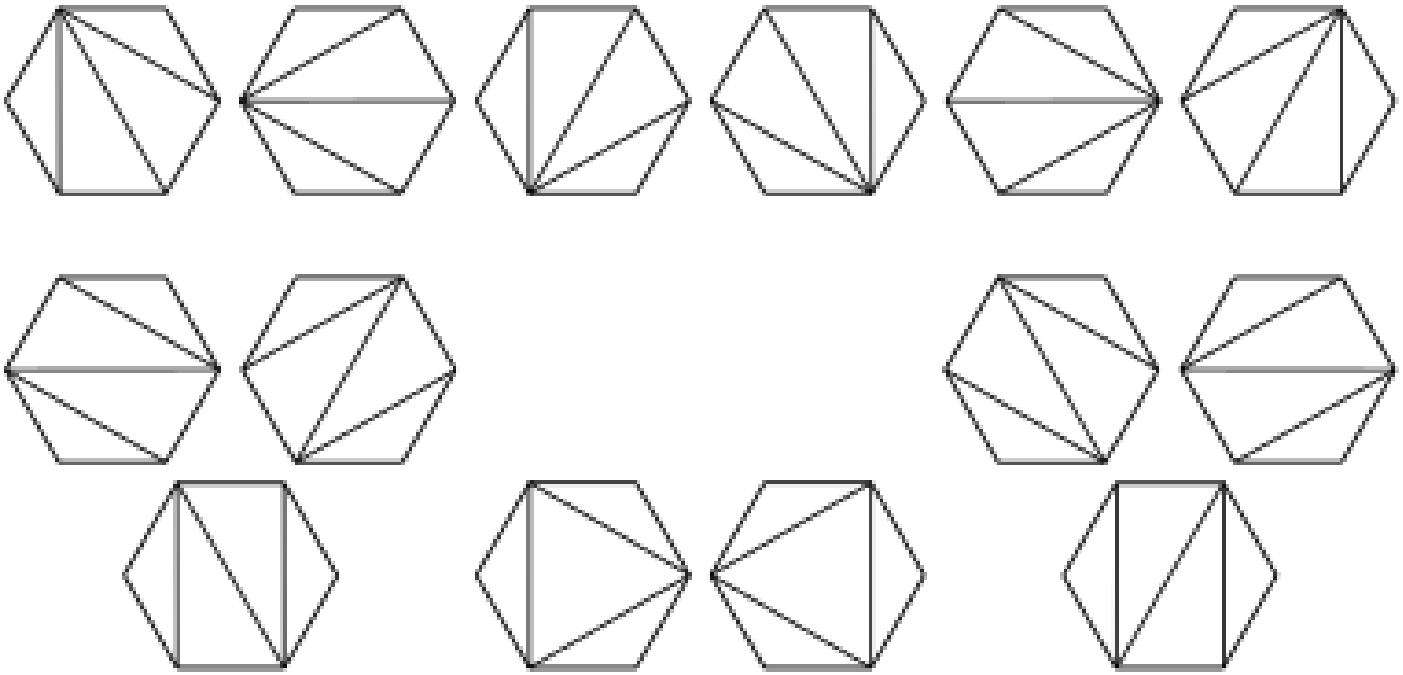


  - Pentru n=5, avem T(5) = 5



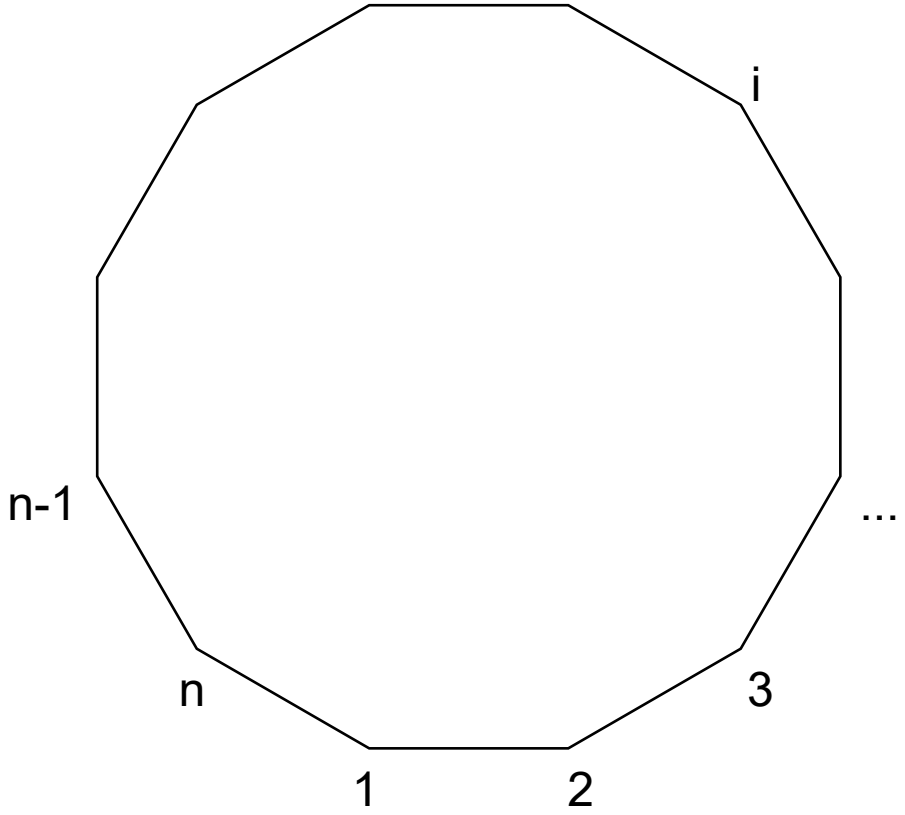
Triangulări n-gon

- Rezolvarea pentru cazuri mici
  - Pentru n=6, avem T(6) = 14



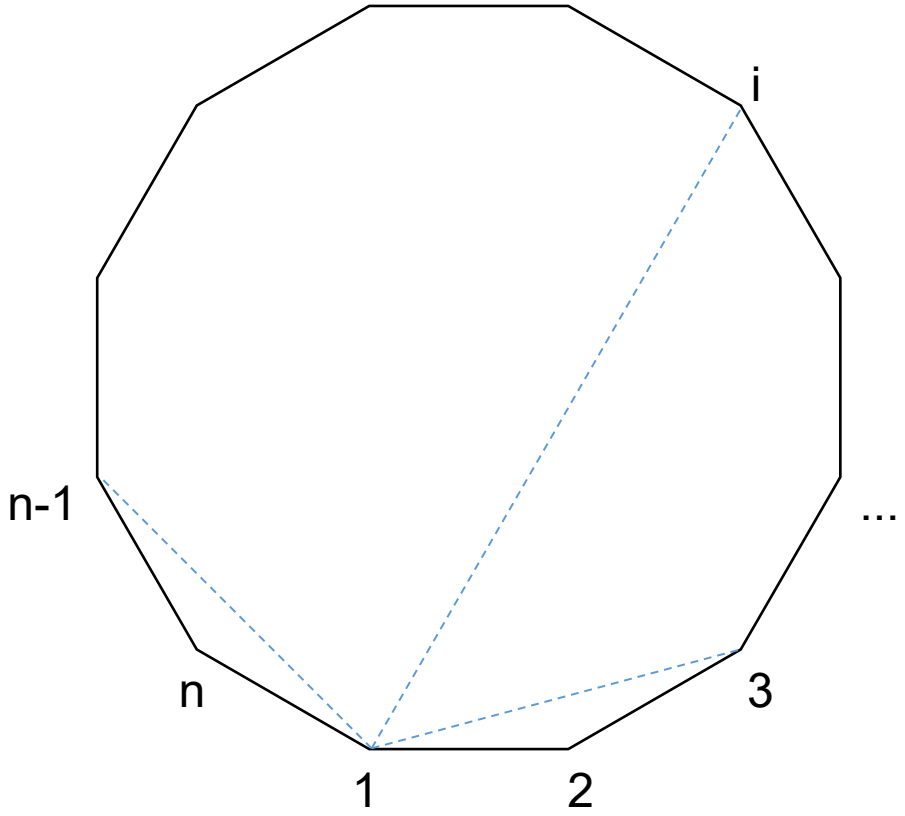
Triangulări n-gon

- Stabilirea relației de recurență
  - Numerotăm vârfurile și studiem ce se întâmplă cu vârful 1



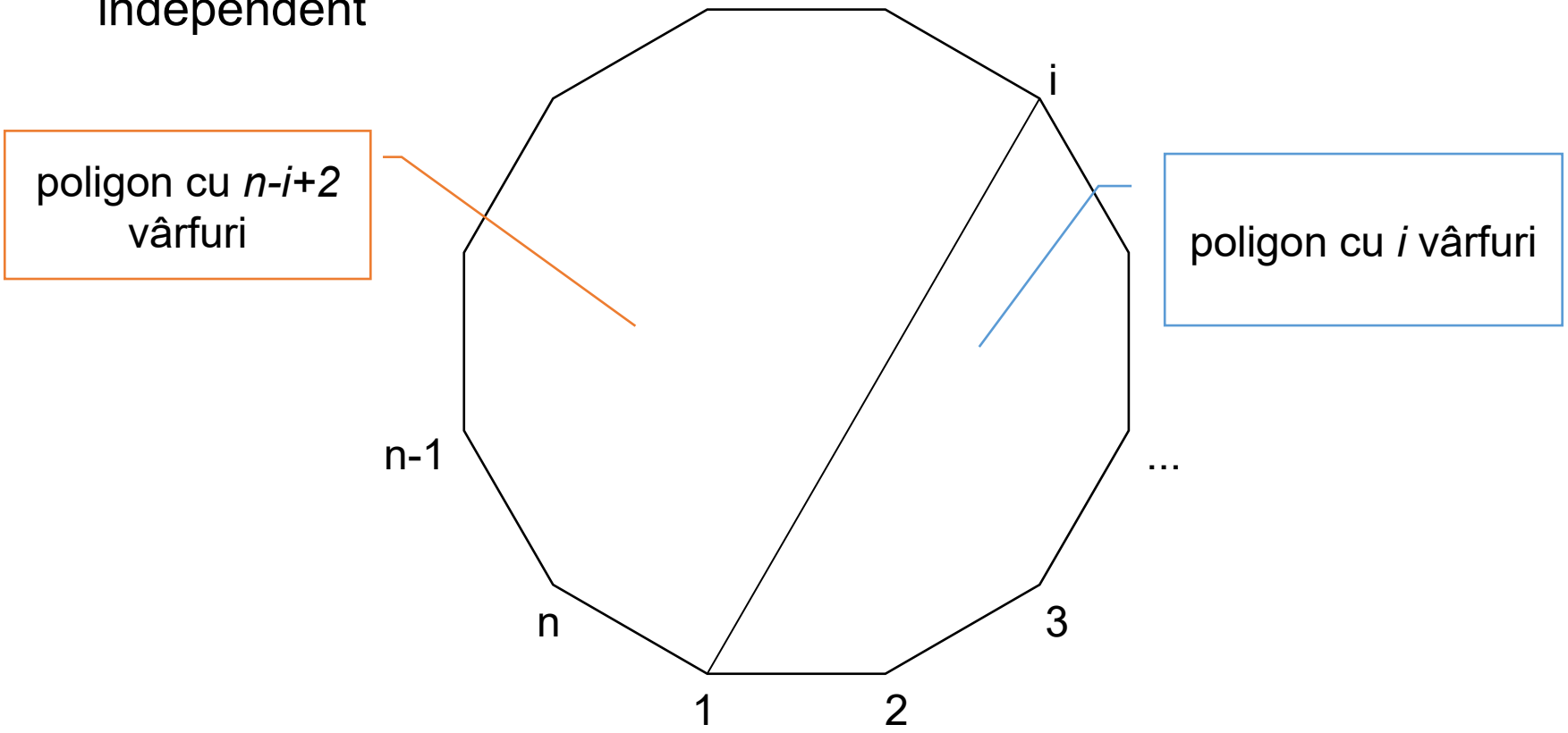
Triangulări n-gon

- Stabilirea relației de recurență
  - Vârful 1 poate fi conectat la unul din vârfurile 3, 4, ..., n-1



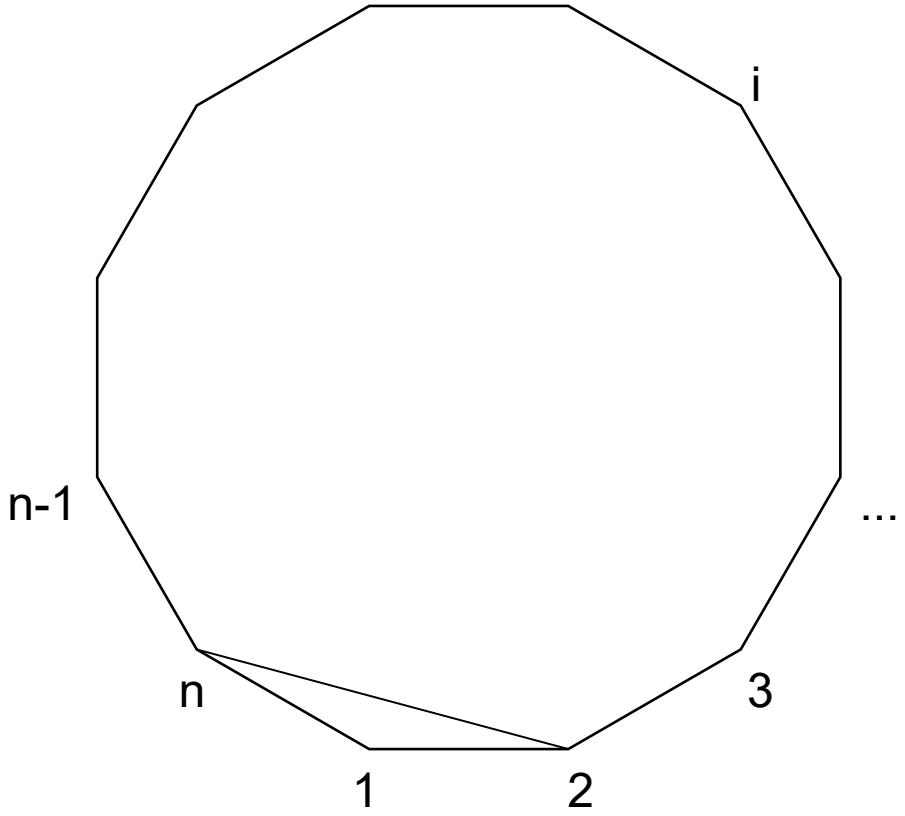
Triangulări n-gon

- Stabilirea relației de recurență
  - Dacă conectăm la vârful *i*
  - Poligonul se împarte în două părți care se pot triangula independent



Triangulări n-gon

- Stabilirea relației de recurență
  - Vârful 1 nu este conectat
  - Trebuie să existe triunghiul *(n,1,2)* și rămâne un (n-1)-gon



Triangulări n-gon

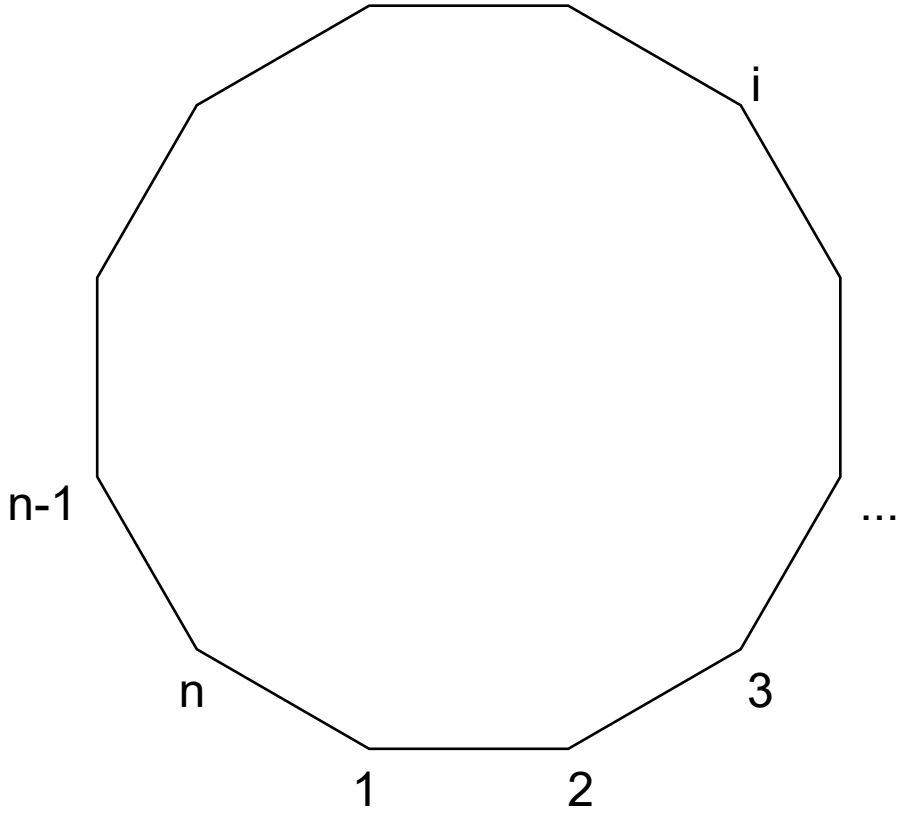
- Stabilirea relației de recurență
  - Dacă conectăm vârful 1 la vârful i avem T(i)T(n-i+2) modalități
    - i poate fi 3, 4, …, n-1
  - Dacă nu conectăm vârful 1 la nimic avem T(n-1) modalități
  - Rezultă formula de recurență

$$T(n) = \sum_{i=3}^{n-1} T(i)T(n-i+2) + T(n-1)$$

  - Dar unele cazuri se suprapun
    - conectăm vârful 1 la i și apoi la j
    - conectăm vârful 1 la j și apoi la i
  - Recurență greșită

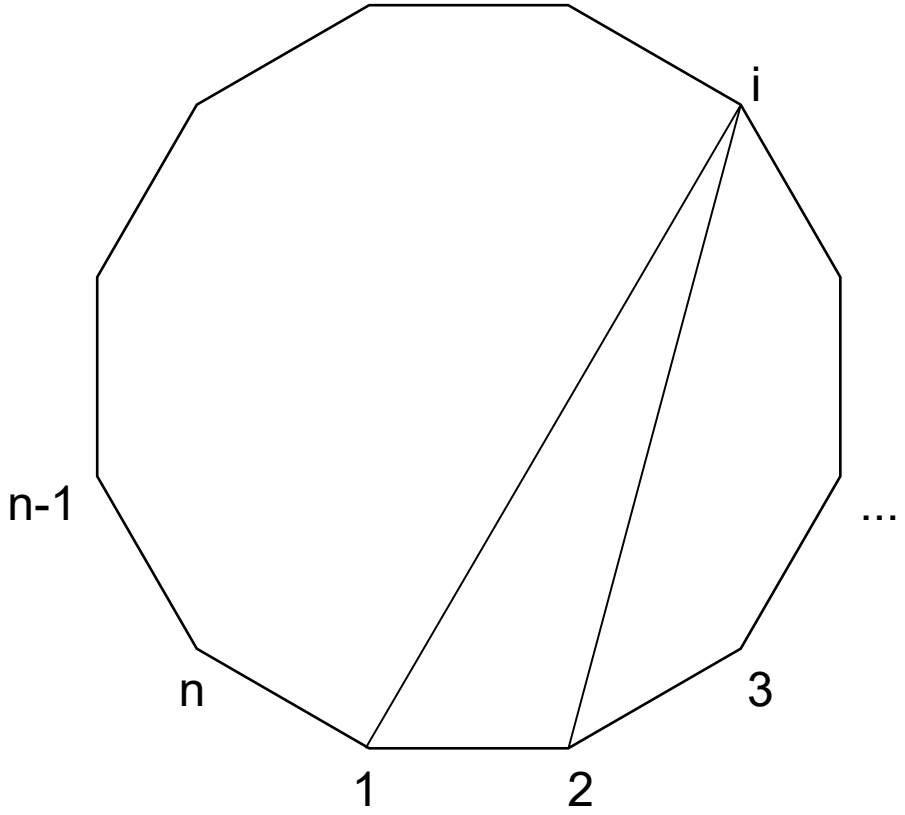
Triangulări n-gon

- Stabilirea relației de recurență
  - Numerotăm vârfurile și studiem ce se întâmplă cu muchia 1-2



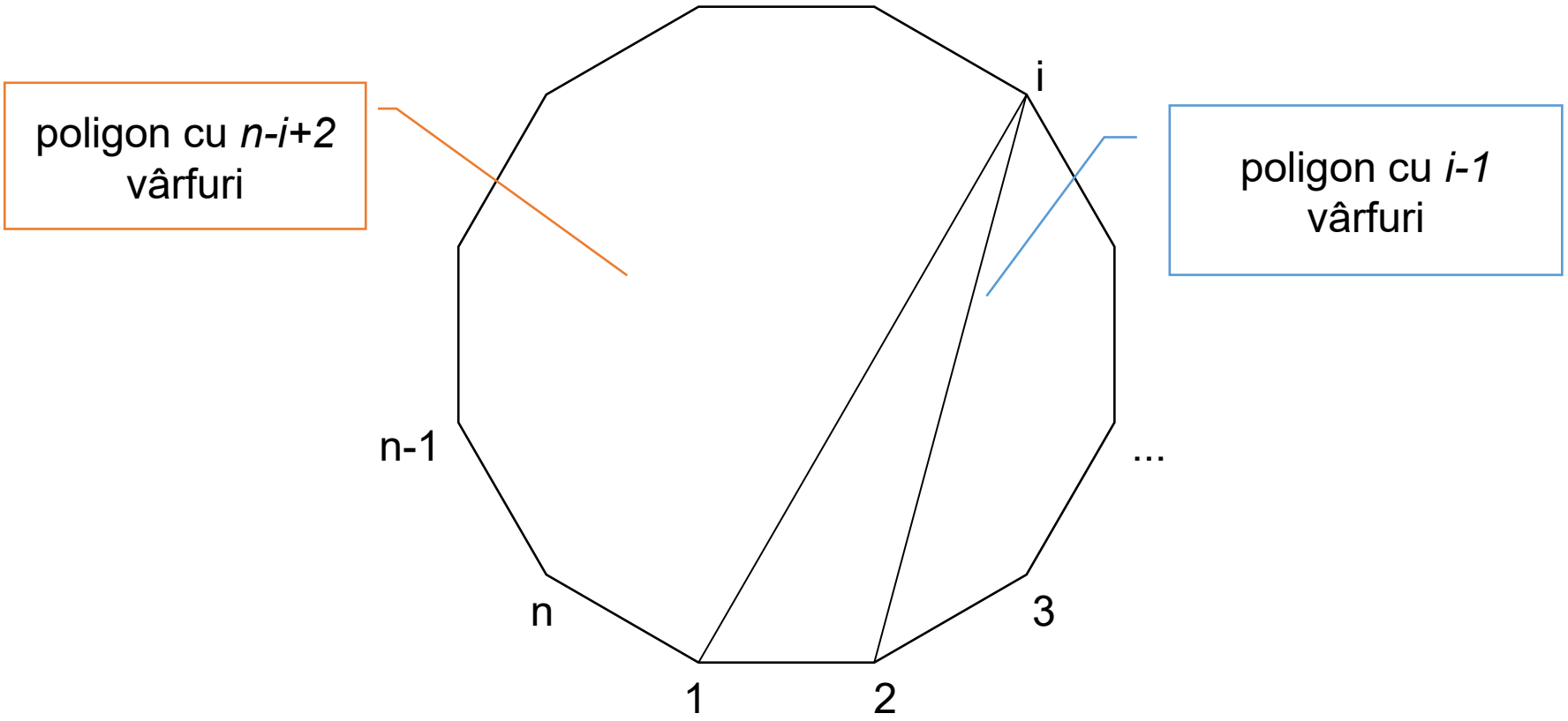
Triangulări n-gon

- Stabilirea relației de recurență
  - Muchia 1-2 poate forma un triunghi cu orice vârf 4, 5, ..., n-1



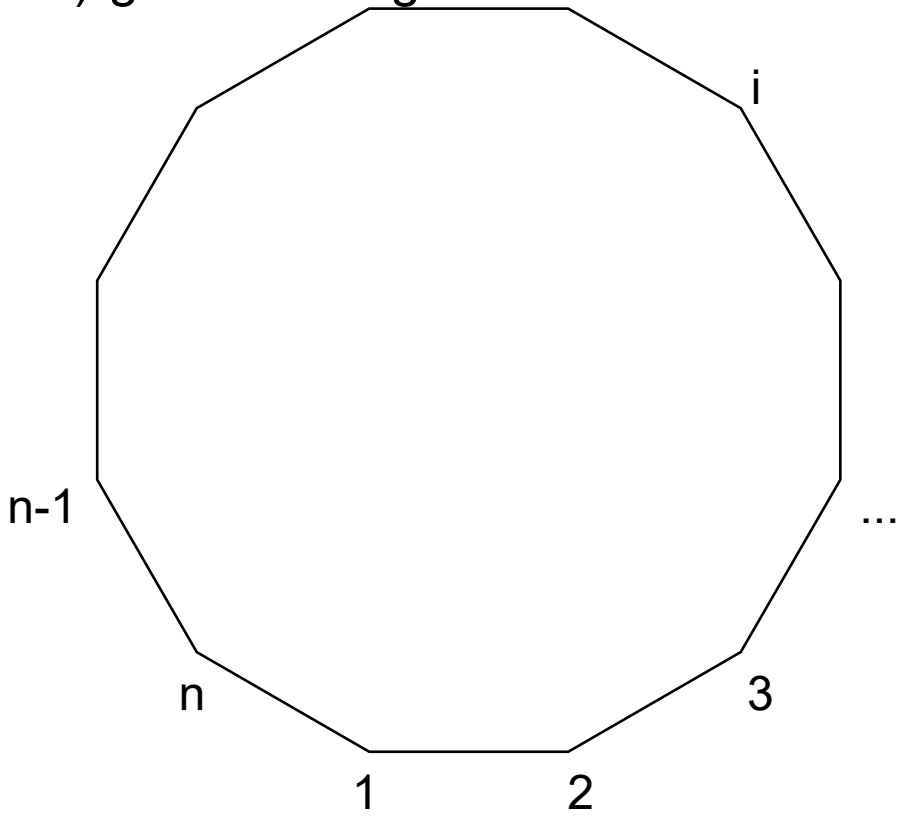
Triangulări n-gon

- Stabilirea relației de recurență
  - Mai rămân două părți care se pot triangula independent



Triangulări n-gon

- Stabilirea relației de recurență
  - Muchia 1-2 poate forma un triunghi cu vârful 3 sau cu *n*
  - Rămâne un (n-1)-gon de triangulat



Triangulări n-gon

- Stabilirea relației de recurență
  - Dacă conectăm 1-2 la *i* avem *T(i-1)T(n-i+2)* modalități
    - i poate fi 4, 5, …, n-1
  - Dacă conectăm 1-2 la 3 sau *n* avem câte *T(n-1)* modalități
  - Rezultă formula de recurență

$$T(n) = \sum_{i=4}^{n-1} T(i-1)T(n-i+2) + 2T(n-1)$$

  - Dacă adoptăm *T(2)* = *1*, atunci avem formula mai simplă

$$T(n) = \sum_{i=3}^{n} T(i-1)T(n-i+2)$$

  - Cazurile nu se suprapun
  - Recurență corectă

Triangulări n-gon - implementare

```
long long triangulari(int n){
  if (n<=3)
    return 1;
  long long rez = 0;
  for(int i=3; i<=n; i++)
    rez += triangulari(i-1)*triangulari(n-i+2);
  return rez;
}
```

- implementare recursivă
- triangulari(20) = 477638700 sub o secundă
- triangulari(30) = ?

Triangulări n-gon - implementare cu memoizare

```
long long T[50] = {[2] = 1, [3] = 1};
long long triangulari_memo(int n){
  if (T[n])
    return T[n];
  long long rez = 0;
  for(int i=3; i<=n; i++)
    rez += triangulari_memo(i-1)*triangulari_memo(n-i+2);
  return T[n] = rez;
}
```

- implementare recursivă
- triangulari_memo(20) = 477638700 sub o secundă
- triangulari_memo(30) = 263747951750360 ~ millisecunde

Triangulări n-gon - implementare iterativă

```
long long triangulari_iter(int n){
    long long T[50] = {[2] = 1, [3] = 1};
    for(int k=4; k<=n; k++){
        for(int i=3; i<=k; i++)
            T[k] += T[i-1]*T[k-i+2];
    }
    return T[n];
}
```

- implementare iterativă
- există formulă
- sunt numerele Catalan
- T(n) = C(n-2) , unde $C(n) = \frac{1}{n+1}\binom{2n}{n}$