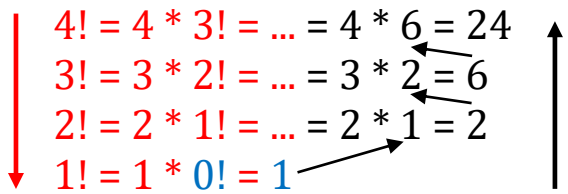
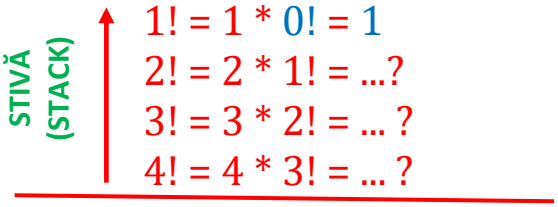
RECURSIVITATE

Recursivitate = capacitatea unei funcţii de a se autoapela

Exemplu: Calculul lui $n!$

$$n! = \underbrace{1 * 2 * 3 * \ldots * (n-1)}_{(n-1)!} * n = \begin{cases} n * (n-1)!, \text{dacă } n \geq 1 \\ 1, \quad \text{dacă } n = 0 \end{cases}$$






```
#include <iostream>

using namespace std;

unsigned long long int factorial(unsigned int n)
{
    if(n == 0)
        return 1;

    return n * factorial(n-1);
}

int main()
{
    unsigned int n;

    cout << "n = ";
    cin >> n;

    cout << n << "! = " << factorial(n) << endl;

    return 0;
}
```

Exemplu: Calculul sumei cifrelor unui număr natural

$$sc(n) = \begin{cases} n\%10 + sc(n/10), & \text{dacă } n \geq 1 \\ 0, & \text{dacă } n = 0 \end{cases}$$

$\boldsymbol{sc(n)}$ = funcție care calculează suma cifrelor numărului natural n

```
n = 12345     s = 0 + 5 = 5
n = 1234      s = 5 + 4 = 9
n = 123       s = 9 + 3 = 12
...

sc(12345) = 5 + sc(1234) = ... = 5 + 10 = 15
sc(1234) = 4 + sc(123) = ... = 4 + 6 = 10
sc(123) = 3 + sc(12) = ... = 3 + 3 = 6
sc(12) = 2 + sc(1) = ... = 2 + 1 = 3
sc(1) = 1 + sc(0) = 1
```

```
#include <iostream>

using namespace std;

//sc(n) = functie care calculeaza suma cifrelor lui n
unsigned int sc(unsigned int n)
{
    if(n == 0)
        return 0;

    return n%10 + sc(n/10);
}

int main()
{
    unsigned int n;

    cout << "n = ";
    cin >> n;

    cout << "Suma cifrelor lui " << n << " este "<< sc(n) <<
endl;

    return 0;
}
```

Exemplu: Calculul sumei elementelor unui tablou unidimensional v format din n numere întregi

$$suma(v,n) = \begin{cases} v[0] + suma(v+1, n-1), \text{dacă } n \geq 1 \\ 0, \quad \text{dacă } n = 0 \end{cases}$$

$\boldsymbol{suma(v,n)}$ = funcţie care calculează suma elementelor unui tablou unidimensional v format din n numere întregi

$\boldsymbol{v}$	0	1	2	3	4	5	6	indice
	12	-7	10	9	-7	3	1	valoare
	v	v+1	v+2	v+3	v+4	v+5	v+6	adresă

v == &v[0] (numele unui tablou este adresa primului său element)

```
suma((12,-7,10,9,-7,3,1), 7) = 12 + suma((-7,10,9,-7,3,1), 6)
```

```
#include <iostream>

using namespace std;

//suma(v, n) = functie care calculeaza suma elementelor
//unui tablou unidimensional v format din n numere intregi
unsigned int suma(int v[], int n)
{
    if(n == 0)
        return 0;

    return v[0] + suma(v+1, n-1);
}

int main()
{
    int n, a[20];

    cout << "n = ";
    cin >> n;

    for(int i = 0; i < n; i++)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
    }

    cout << "Suma elementelor tabloului: " << suma(a, n) << endl;

    return 0;
}
```

Varianta 1:

```
#include <iostream>

using namespace std;

int suma(int v[], int n)
{
    if(n == 1)
        return v[0];
    return v[n-1] + suma(v, n-1);
}

int main()
{

    int v[] = {1, 2, 3, 4, 5, 6};
    int n = 6;

    int s;

    s = suma(v, n);
    cout << "Suma: " << s << endl;

    return 0;
}
```

Varianta 2:

```
#include <iostream>

using namespace std;

int suma(int v[], int n, int k)
{
    if(k == n-1)
        return v[n-1];
    return v[k] + suma(v, n, k+1);
}

int main()
{

    int v[] = {1, 2, 3, 4, 5, 6};
    int n = 6;

    int s;

    s = suma(v, n, 0);
    cout << "Suma: " << s << endl;

    return 0;
}
```

Exemplu: Se citesc numere întregi până la întâlnirea valorii 0. Să se afișeze numerele citite în ordine inversă, fără a folosi nicio structură de date auxiliară!

```
#include <iostream>

using namespace std;

void inversare()
{
    int x;

    cin >> x;
    if(x != 0)
    {
        inversare();
        cout << x << " ";
    }
}

int main()
{
    inversare();

    return 0;
}
```

Variantă (deși nu se recomandă utilizarea recursive a funcției `main()`)**:**

```
#include <iostream>

using namespace std;

int main()
{
    int x;

    cin >> x;
    if(x != 0)
    {
        main();
        cout << x << " ";
    }

    return 0;
}
```

TEHNICA DE PROGRAMARE "DIVIDE ET IMPERA"

1. Suma elementelor unui tablou unidimensional

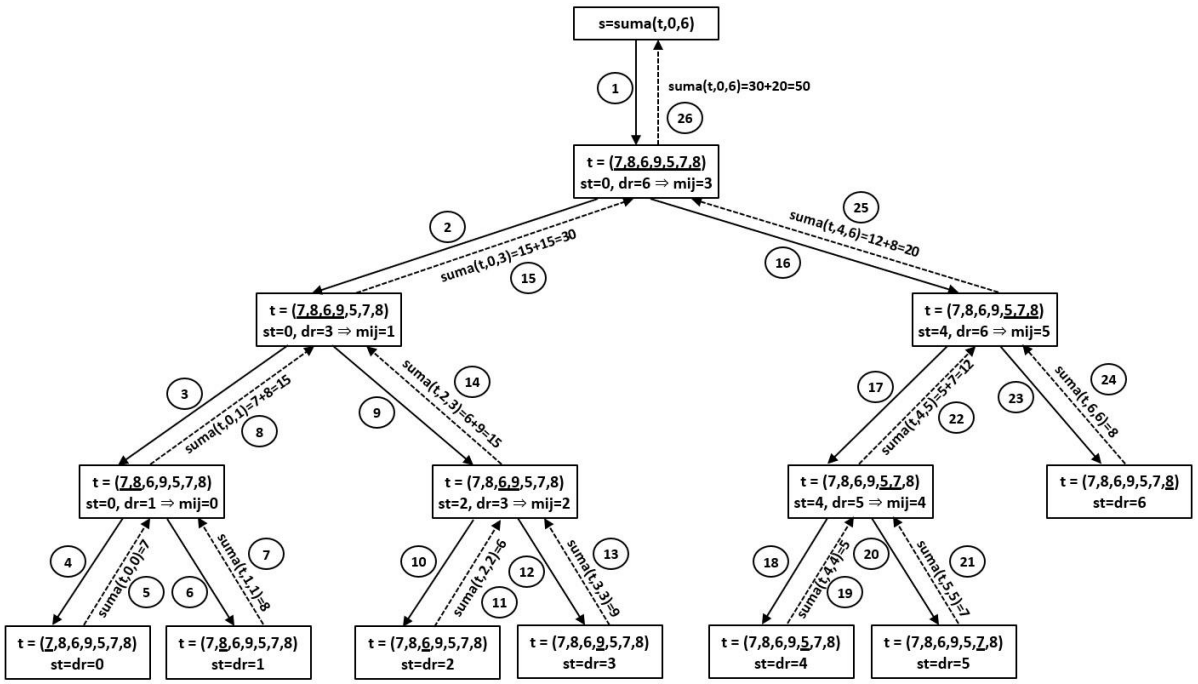
Pentru a putea manipula uşor cele două sub-tablouri care se obţin în momentul împărţirii tabloului t în două jumătăţi, vom considera faptul că tabloul curent este secvenţa cuprinsă între doi indici st şi dr, unde $st \leq dr$. Astfel, indicele mij al mijlocului tabloului curent este aproximativ egal cu $[(st + dr)/2]$, iar cele două sub-tablouri în care va fi descompus tabloul curent sunt secvenţele cuprinse între indicii st şi mij, respectiv $mij + 1$ şi dr. Considerând $suma(t, st, dr)$ o funcţie care calculează suma secvenţei $t[st], t[st + 1], \ldots, t[dr]$, putem să o definim în manieră Divide et Impera astfel:

$$suma(t, st, dr) = \begin{cases} t[st], & \text{dacă } st = dr \\ suma(t, st, mij) + suma(t, mij + 1, dr), & \text{dacă } st < dr \end{cases}$$

unde $mij = [(st + dr)/2]$.

Considerând tabloul t cu n elemente ca fiind indexat de la 0, suma s a tuturor elementelor sale se va obţine în urma apelului $s = suma(t, 0, n - 1)$.

De exemplu, pentru tabloul $t = (7, 8, 6, 9, 5, 7, 8)$ având $n = 7$ elemente, pentru a calcula suma elementelor sale, se vor efectua următoarele apeluri recursive:



```
#include <iostream>

using namespace std;

//suma(t, n) = functie care calculeaza suma elementelor
//unui tablou unidimensional t format din n numere intregi
//folosind tehnica de programare Divide et Impera

//st, dr = capetele secventei curente din tabloul t
int suma(int t[], int st, int dr)
{
    int mij, suma_st, suma_dr;

    //daca tabloul t are un singur element,
    //atunci suma ceruta este chiar acel element
    if(st == dr)
        return t[st];

    //etapa Divide
    //calculez mijlocul mij al secventei t[st], t[st+1], ..., t[dr]
    mij = (st + dr) / 2;

    //calculez suma subsecventei din stanga t[st], t[st+1], ..., t[mij]
    suma_st = suma(t, st, mij);

    //calculez suma subsecventei din dreapta t[mij+1], t[mij+2], ...,
t[dr]
    suma_dr = suma(t, mij+1, dr);

    //etapa Impera
    //calculez suma elementelor din secventa t[st], t[st+1], ..., t[dr]
    //prin suma_st + suma_dr
    return suma_st + suma_dr;
}

int main()
{
    int n, a[20], s;

    cout << "n = ";
    cin >> n;

    for(int i = 0; i < n; i++)
    {
        cout << "a[" << i << "] = ";
        cin >> a[i];
    }

    s = suma(a, 0, n-1);
    cout << "Suma elementelor tabloului: " << s << endl;

    return 0;
}
```

Pentru a determina complexitatea unui algoritm recursiv, vom considera faptul că un apel de funcție este echivalent cu o operație elementară!

Notăm cu $T(n)$ numărul de apeluri (recursive) ale funcției `int suma(...)` necesare pentru a rezolva problema, i.e. pentru a calcula suma elementelor dintr-un tablou t format din n numere întregi.

```
int suma(int t[], int st, int dr)
{
    int mij, suma_st, suma_dr;

    if(st == dr)
        return t[st];

    mij = (st + dr) / 2;

    suma_st = suma(t, st, mij);
    suma_dr = suma(t, mij+1, dr);

    return suma_st + suma_dr;
}
```

Valoarea lui $T(n)$ se calculează folosind următoarea relație de recurență:

$$T(n) = \begin{cases} 1 + T\left(\left[\frac{n}{2}\right]\right) + T\left(\left[\frac{n}{2}\right]\right) + 1, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases}$$

$$T(n) = \begin{cases} 2T\left(\left[\frac{n}{2}\right]\right) + 2, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases}$$

Presupunem faptul că $n = 2^k \Rightarrow T(n) = T(2^k) = 2T(2^{k-1}) + 2 =$

$2[2T(2^{k-2}) + 2] + 2 = 2^2T(2^{k-2}) + 2^2 + 2 = 2^2[2T(2^{k-3}) + 2] + 2^2 + 2 =$

$2^3T(2^{k-3}) + 2^3 + 2^2 + 2 = \cdots = 2^k \underbrace{T(2^0)}_{1} + 2^k + 2^{k-1} + \cdots + 2^2 + 2 =$

$2^k + 2^k + 2^{k-1} + \cdots + 2^2 + 2 = 2^k + 2^{k+1} - 2 = 2^k(1 + 2) - 2 = 3 * 2^k -$

$2 = 3n - 2$ => complexitate algoritmului este $\mathcal{O}(3n - 2) \approx \mathcal{O}(n)$.

$S_k = 2^k + 2^{k-1} + \cdots + 2^2 + 2 = 2 + 2^2 + \cdots + 2^{k-1} + 2^k$ = suma unei progresii geometrice cu primul termen $a_1 = 2$, rație $q = 2$ și k termeni

$\Rightarrow S_k = \frac{a_1(q^k-1)}{q-1} = \frac{2(2^k-1)}{2-1} = 2^{k+1} - 2$