Metoda Divide et Impera

Divide et impera se bazează pe principiul descompunerii problemei în două sau mai multe subprobleme (mai ușoare), care se rezolvă, iar soluția pentru problema inițială se obține combinând soluțiile subproblemelor. De multe ori, subproblemele sunt de același tip și pentru fiecare din ele se poate aplica aceeași tactică a descompunerii în (alte) subprobleme, până când (în urma descompunerilor repetate) se ajunge la probleme care admit rezolvare imediată.
Nu toate problemele pot fi rezolvate prin utilizarea acestei tehnici. Se poate afirma că numărul celor rezolvabile prin "divide et impera" este relativ mic, tocmai datorită cerinței ca problema să admită o descompunere repetată.
Divide et impera este o tehnică ce admite o implementare recursivă. Principiul general prin care se elaborează algoritmi recursivi este: "ce se întâmplă la un nivel, se întâmplă la orice nivel" (având grijă să asigurăm condițiile de terminare). Așadar, un algoritm prin divide et impera se elaborează astfel: la un anumit nivel avem două posibilități:

1. s-a ajuns la o problemă care admite o rezolvare imediată (condiția de terminare), caz în care se rezolvă și se revine din apel;
2. nu s-a ajuns în situația de la punctul 1, caz în care problema curentă este descompusă în (două sau mai multe) subprobleme, pentru fiecare din ele urmează un apel recursiv al funcției, după care combinarea rezultatelor are loc fie pentru fiecare subproblemă, fie la final, înaintea revenirii din apel.

Metoda presupune:

- Descompunerea problemei Pb curente in subprobleme independente $SubPb_i$.
    - In cazul in care subproblemele $SubPb_i$. admit o rezolvare imediata se compun solutiile si se rezolva problema Pb
    - Altfel se descompun in mod similar si subproblemele $SubPb_i$

Evident, nu toate problemele pot fi rezolvate prin utilizarea acestei tehnici. Majoritatea algoritmilor de Divide et Impera presupun prelucrari pe tablouri (dar nu obligatoriu).

Aceasta metoda (tehnica) se poate implementa atat iterativ dar si recursiv. Dat fiind ca problemele se impart impart in subprobleme in mod recursiv, de obicei impartirea se realizeaza pana cand sirul obtinut este de lungime 1, caz in care rezolvarea subproblemei este foarte usoara, chiar banala.

Spre exemplu fie un vector $X=[x_1, x_2, x_3, \ldots x_i \ldots x_p \ldots x_j, \ldots x_n]$ asupra caruia se aplica o prelucrare. Pentru orice secventa din vector delimitata de indecsii i si j, i<j exista o valoare p astfel incat prin prelucrarea secventelor :

$$x_i, x_{i+1}, x_{i+2}, x_{i+3}, \ldots x_p \text{ si } x_{p+1}, x_{p+2}, x_{p+3}, \ldots x_j$$

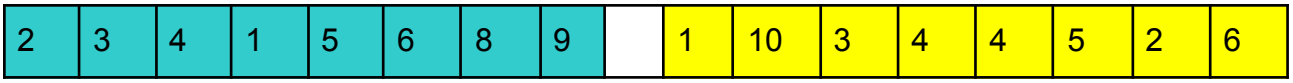
se obtin solutiile corespunzatoare celor doua subsiruri care prin compunere conduc la obtinerea solutiei prelucrarii secventei:

$$x_i, x_{i+1}, x_{i+2}, x_{i+3}, \ldots x_j$$

Aplicatie:
Ne propunem sa determinam suma elementelor unui vector de intregi utilizand metoda Divide et Impera:

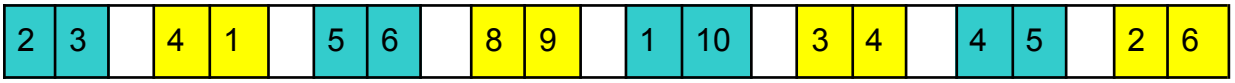
2	3	4	1	5	6	8	9	1	10	3	4	4	5	2	6

Se imparte vectorul in doi vectori:



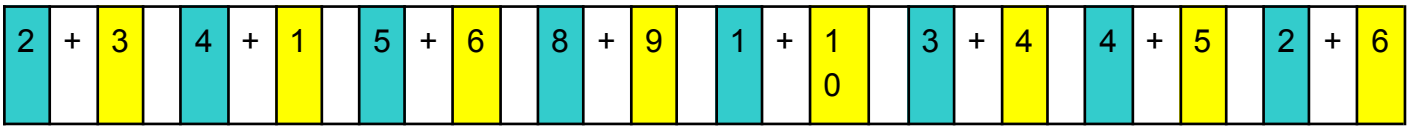
Fiecare se dintre cei doi vectori se poate imparti in continuare in alti doi vectori:

La fel se procedeaza in continuare:



Apoi:

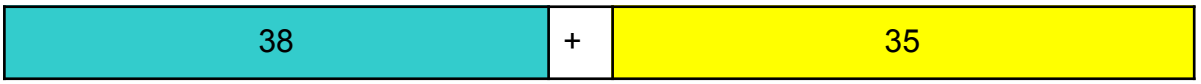
Dat fiind ca s-au obtinut secvente de vector de lungime 1, nu se mai realizeaza descompunerea.
Se compun solutiile subsecventelor si se determina solutiile corespunzatoare:



Si in continuare:

Apoi:

La fel:



La sfarsit:

Probleme clasice

1. Sortarea prin interclasare

Sortarea prin interclasare (MergeSort) este un algoritm de sortare de vectori ce folosește paradigma D&I:

- Divide: împarte vectorul inițial în doi sub-vectori de dimensiune n/2.
- Stăpânește: sortează cei doi sub-vectori recursiv folosind sortarea prin interclasare; recursivitatea se oprește când dimensiunea unui sub-vector este 1 (deja sortat).
- Combina: Interclasează cei doi sub-vectori sortați pentru a obține vectorul inițial sortat.

Pseudocod:

```
MergeSort(v, start, end)            // v - vector, start - limită inferioră, end - limită superioară
        if (start == end) return;   // condiția de oprire
        mid = (start + end) / 2;    // etapa divide
```

```
        MergeSort(v, start, mid);     // etapa stăpânește
        MergeSort(v, mid+1, end);
        Merge(v, start, end);         // etapa combină

Merge(v, start, end)                  // interclasare sub-vectori
        mid = (start + end) / 2;
        i = start;
        j = mid + 1;
        k = 1;
        while (i <= mid && j <= end)
                if (v[i] <= v[j]) u[k++] = v[i++];
                else u[k++] = v[j++];

        while (i <= mid)
                u[k++] = v[i++];

        while (j <= end)
                u[k++] = v[j++];

        copy(v[start..end], u[1..k-1]);
```

Complexitatea algoritmului este dată de formula: T(n) = D(n) + S(n) + C(n), unde D(n)=O(1), S(n) = 2*T(n/2) și C(n) = O(n), rezulta T(n) = 2 * T(n/2) + O(n).
Folosind teorema Master găsim complexitatea algoritmului: T(n) = O(n * lg n).

2. Căutarea binară

Se dă un vector sortat crescător (v[1..n]) ce conține valori reale distincte și o valoare x. Sa se găsească la ce poziție apare x în vectorul dat.
Pentru rezolvarea acestei probleme folosim un algoritm D&I:

- Divide: împărțim vectorul în doi sub-vectori de dimensiune n/2.
- Stăpânește: aplicăm algoritmul de căutare binară pe sub-vectorul care conține valoarea căutată.
- Combină: soluția sub-problemei devine soluția problemei inițiale, motiv pentru care nu mai este nevoie de etapa de combinare.

Pseudocod:

```
BinarySearch(v, start, end, x)
        if (start > end) return;      // condiția de oprire (x nu se află în v)
        mid = (start + end) / 2;      // etapa divide
        // etapa stăpânește
        if (v[mid] == x) return mid
        if (v[mid] > x) return BinarySearch(v, start, mid-1, x);
        if (v[mid] < x) return BinarySearch(v, mid+1, end, x);
```

Complexitatea algoritmului este data de relația T(n) = T(n/2) + O(1), ceea ce implica: T(n) = O(lg n).

3. Turnurile din Hanoi

Se considera 3 tije A, B, C şi n discuri de dimensiuni distincte (1, 2.. n ordinea crescătoare a dimensiunilor) situate inițial toate pe tija A în ordinea 1,2..n (de la vârf către baza). Singura operație care se poate efectua este de a selecta un disc ce se află în vârful unei tije şi plasarea lui în vârful altei tije astfel încât să fie așezat deasupra unui disc de dimensiune mai mare decât a sa. Sa se găsească un algoritm prin care se mută toate discurile pe tija B (problema turnurilor din Hanoi).
Pentru rezolvarea problemei folosim următoarea strategie:

- mutam primele n-1 discuri de pe tija A pe tija C folosindu-ne de tija B.

- mutam discul n pe tija B.
- mutam apoi cele n-1 discuri de pe tija C pe tija B folosindu-ne de tija A.

Pseudocod [10]:

```
Hanoi(n, A, B, C)     // mută n discuri de pe tija A pe tija B fol tija C
       if (n >= 1)
              Hanoi(n-1, A, C, B);
              Muta_disc(A, B);
              Hanoi(n-1, C, B, A);
```

Complexitatea: T(n) = 2*T(n-1) + O(1), recurenta ce conduce la $T(n) = O(2^n)$.

4. Cautare număr lipsă

Se dă un șir neordonat S cu n - 1 numere distincte, selectate dintre cele n numere de la 0 la n - 1. Toate sunt numere întregi, reprezentate pe 32 de biti. Folosind metoda getBit(int i, int j) care intoarce al j-lea bit din reprezentarea binara a lui S[i], determinati numarul lipsă.
Pentru un număr dat n, fie $m = 2^k$ primul număr mai mare decât n care este o putere a lui 2. Atunci, în șirul numerelor de la 0 până la n - 1 vor exista 2^{k-1} numere cu bitul numărul k - 1 egal cu 0.
Pentru n = 7, m = 8 și k = 3. Drept urmare, vor fi $2^{k-1} = 2^2 = 4$ numere între 0 și 7 cu bitul 2 egal cu 0. Dacă numărul care lipsește din șirul din problemă va avea bitul 2 egal cu 0, atunci vor fi $2^2 - 1$ numere în șir cu bitul zero. Altfel, înseamnă că numărul care lipsește are bitul 2 egal cu 1. După ce determinăm în care "jumătate" se află numărul lipsă, continuăm căutarea mai departe folosind-o doar pe aceea.
Reprezentarea în binar a numerelor întregi din intervalul [0, 7):

		2	1	0
0	=	0	0	0
1	=	0	0	1
2	=	0	1	0
3	=	0	1	1
4	=	1	0	0
5	=	1	0	1
6	=	1	1	0

Putem să aplicăm această regulă ca să determinăm elementul lipsă din șir, iterând de la cel mai semnificativ bit spre cel mai nesemnificativ.
Exemplu:

- pentru șirul {0 1 9 4 5 7 6 8 2} lipsește numarul 3
- getBit(7,3) întoarce al treilea bit din S[7]. S[7] este 8, în binar: 1000, deci bit-ul 3 este 1.
- La un prim pas, se observă că bitul 3 este 0 pentru doar 7 numere din șir, deși între 0 și 9 sunt 8 numere care ar trebui să aibă bitul 3 egal cu 0, respectiv numerele de la 0 la 7. Prin urmare, la primul pas ne dăm seama că numărul căutat este între 0 si 7.

Pseudocod:

```
getMissing(V, n):
    current_vec = V
    current_bit = 31
    result = 0

    while (getBit(n+1, current_bit) == 0)
        current_bit--
```

```
    while (current_bit >= 0)
        setBit0 = {}
        setBit1 = {}

        for (element : current_vec)
            if (getBit(element, current_bit) == 0)
                setBit0.add(element)
            else
                setBit1.add(element)

        if (setBit0.size != (1 << current_bit))
            result |= (1 << current_bit)
            current_vec = setBit1
        else
            current_vec = setBit0

    return result
```

Mai multe probleme clasice rezolvate cu divide et impera aici.