CURS 3
METODE DE OPTIMIZARE A ALGORITMILOR

În acest curs vom analiza mai multe probleme "clasice" de programare şi vom prezenta, pentru fiecare dintre ele, mai multe variante de rezolvare având diferite complexităţi. Astfel, vom putea pune în evidenţă mai multe metode cu ajutorul cărora putem să eficientizăm sau să optimizăm un algoritm.

Determinarea unei secvenţe cu sumă maximă

Considerăm un tablou unidimensional p format din n numere întregi ($n \geq 1$) şi vrem să determinăm o secvenţă a sa având suma elementelor maximă (i.e., orice altă secvenţă a tabloului va avea suma elementelor cel mult egală cu suma secvenţei respective). De exemplu, pentru tabloul $p = (10, 9, -23, 7, 10, 11, -3)$ o secvenţă cu suma elementelor maximă este $7, 10, 11$, iar pentru tabloul $p = (10, 9, -23, 7, 10, 11, -3, 10, -5)$ o secvenţă cu sumă maximă este $7, 10, 11, -3, 10$.

O primă variantă de rezolvare constă în calcularea sumelor tuturor secvenţelor posibile din tabloul p şi păstrarea celei pentru care suma elementelor este maximă:

```
void varianta_1(int p[], int n)
{
    int i, j, k, st, dr, scrt,smax;

    smax = p[0];
    st = dr = 0;
    for(i = 0; i < n; i++)
        for(j = i; j < n; j++)
        {
            scrt = 0;
            for(k = i; k <= j; k++)
                scrt = scrt + p[k];
            if(scrt > smax)
            {
                smax = scrt;
                st = i;
                dr = j;
            }
        }

    printf("Varianta 1:\n");
    printf("\tSuma maxima: %d\n", smax);
    printf("\tCapetele secventei maximale: %d -> %d\n", st, dr);
}
```

Se observă cu ușurință faptul că această variantă de rezolvare are complexitatea computațională $\mathcal{O}(n^3)$, indusă de cele 3 instrucțiuni repetitive imbricate de tip `for`.

O variantă de rezolvare cu o complexitate mai mică se poate obține observând faptul că suma $p[i] + \cdots + p[j-1] + p[j]$ se poate calcula, pentru un anumit indice i, fără a parcurge toată secvență din tabloul p delimitată de indicii i și j, ci doar adăugând elementul curent $p[j]$ la suma $scrt$ a secvenței anterioare (i.e., suma $p[i] + \cdots + p[j-1]$):

```
void varianta_2(int p[], int n)
{
    int    i, j, st, dr, scrt,smax;

    smax = p[0];
    st = dr = 0;
    for(i = 0; i < n; i++)
    {
        scrt = 0;
        for(j = i; j < n; j++)
        {
            scrt = scrt + p[j];

            if(scrt > smax)
            {
                smax = scrt;
                st = i;
                dr = j;
            }
        }
    }

    printf("Varianta 2:\n");
    printf("\tSuma maxima: %d\n", smax);
    printf("\tCapetele secventei maximale: %d -> %d\n", st, dr);
}
```

Astfel, vom obține o variantă de rezolvare cu complexitatea computațională $\mathcal{O}(n^2)$, deci mai mică decât cea a primei variante.

O soluție mai eficientă decât cea anterioară se obține observând faptul că o secvență $p[i], \dots, p[j-1]$ poate fi extinsă spre dreapta cu elementul $p[j]$ dacă noua sumă curentă care se obține (i.e., suma $p[i] + \cdots + p[j-1] + p[j]$) este pozitivă, altfel suma curentă va fi reinițializată cu valoarea elementului $p[j]$:

```
void varianta_3(int p[], int n)
{
    int    i, aux, st, dr, scrt,smax;

    smax = INT_MIN;
    aux = 0;
```

```
    scrt = 0;

    for(i = 0; i < n; i++)
    {
        if(scrt < 0)
        {
            scrt = p[i];
            aux = i;
        }
        else
            scrt = scrt + p[i];

        if(scrt > smax)
        {
            smax = scrt;
            st = aux;
            dr = i;
        }
    }

    printf("Varianta 3:\n");
    printf("\tSuma maxima: %d\n", smax);
    printf("\tCapetele secventei maximale: %d -> %d\n", st, dr);
}
```

Noua variantă de rezolvare are complexitatea computaţională $\mathcal{O}(n)$, care este optimă deoarece citirea datelor de intrare are, oricum, complexitatea $\mathcal{O}(n)$.

Problema elementului majoritar

Se consideră un tablou unidimensional v format din n numere naturale nenule. Să se afişeze, dacă există, un *element majoritar*, adică un element care apare de cel puţin $\left[\frac{n}{2}\right] + 1$ ori în tablou.

Exemple:
- dacă $v = (1, 5, 5, 1, 1, 5)$, atunci nu există element majoritar
- dacă $v = (7, 3, 7, 4, 7, 7)$, atunci elementul 7 este majoritar

Observaţie: Orice tablou are cel mult un element majoritar!

Rezolvare:
Vom prezenta mai multe variante de rezolvare a acestei probleme, având diferite complexităţi computaţionale, fiecare fiind implementată într-o funcţie.

O primă variantă de rezolvare constă în calcularea directă a numărului de apariţii ale fiecărei valori distincte din tabloul v (pentru a evita numărarea de mai multe ori a unei anumite valori, la prima numărare a sa îi vom înlocui toate apariţiile cu 0):

```
void varianta_1(unsigned int v[] , unsigned int n)
{
      unsigned int i, j, k;

      for(i = 0; i < n; i++)
            if(v[i] != 0)
            {
                  k = 1;
                  for(j = i + 1; j < n; j++)
                        if(v[j] == v[i])
                        {
                              k++;
                              v[j] = 0;
                              if(k > n / 2)
                              {
                                    printf("Candidatul %u a obtinut
                                                majoritatea!\n", v[i]);
                                    return;
                              }
                        }
            }

    printf("Nici un candidat nu a obtinut majoritatea!\n");
}
```

Deoarece pot exista cel mult n valori distincte în tabloul v, această soluţie va avea complexitatea maximă $\mathcal{O}(n^2)$.

În a doua variantă de rezolvare vom sorta crescător tabloul v şi apoi vom verifica dacă el conţine un platou (i.e., o secvenţă de elemente egale) având lungimea strict mai mare decât $n/2$:

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

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

void varianta_2(unsigned int v[] , unsigned int n)
{
    unsigned int c, i, k;

    qsort(v, n, sizeof(v[0]), cmpNumereCrescator);
```

```
    c = v[n / 2];
    k = 1;

    for(i = 1; i < n; i++)
        if(v[i] == c)
        {
            k++;
            if(k > n / 2)
            {
                printf("Candidatul %u a obtinut
                                majoritatea!\n", v[i]);
                return;
            }
        }
        else
        {
            c = v[i];
            k = 1;
        }

    printf("Nici un candidat nu a obtinut majoritatea!\n");
}
```

Sortarea tabloului v se realizează cu complexitatea $\mathcal{O}(n \log_2 n)$, iar căutarea unui platou cu lungimea strict mai mare decât $n/2$ are complexitatea maximă $\mathcal{O}(n)$, deci această soluție va avea complexitatea maximă $\mathcal{O}(n \log_2 n)$.

A treia variantă de rezolvare se obține observând faptul că în tabloul v sortat crescător singurul posibil element majoritar este elementul din mijlocul său, deoarece orice secvență formată din cel puțin $n/2 + 1$ elemente egale trebuie să conțină și elementul de pe poziția $n/2$:

```
void varianta_3(unsigned int v[] , unsigned int n)
{
     unsigned int c, i, k;

     qsort(v, n, sizeof(v[0]), cmpNumereCrescator);

     c = v[n / 2];
     k = 0;
     for(i = 0; i < n; i++)
          if(v[i] == c)
          {
               k++;
               if(k > n / 2)
               {
                    printf("Candidatul %u a obtinut
                               majoritatea!\n", v[i]);
                    return;
```

```
                }
            }

    printf("Nici un candidat nu a obtinut majoritatea!\n");
}
```

Sortarea tabloului v se realizează cu complexitatea $\mathcal{O}(n \log_2 n)$, iar determinarea numărului de apariții ale posibilului element majoritar c în tabloul v are complexitatea maximă $\mathcal{O}(n)$, deci și această soluție va avea complexitatea maximă $\mathcal{O}(n \log_2 n)$.

În a patra variantă de rezolvare vom utiliza un tablou de frecvențe fr alocat dinamic cu $max + 1$ elemente, unde max reprezintă valoarea maximă din tabloul v:

```
void varianta_4(unsigned int v[] , unsigned int n)
{
      unsigned int i, max, *fr;

      max = v[0];

      for(i = 1; i < n; i++)
            if(v[i] > max)
                  max = v[i];

      fr = (unsigned int*)calloc(max + 1, sizeof(unsigned int));

      for(i = 0; i < n; i++)
      {
            fr[v[i]]++;
            if(fr[v[i]] > n / 2)
                  {
                        printf("Candidatul %u a obtinut
                                    majoritatea!\n", v[i]);
                        free(fr);
                        return;
                  }
      }

    free(fr);
    printf("Nici un candidat nu a obtinut majoritatea!\n");
}
```

Complexitatea computațională a acestei soluții depinde de valoarea maximă max din tabloul v, astfel:

- dacă $max \leq n$, atunci complexitatea sa este $\mathcal{O}(n)$;
- dacă $max > n$, atunci complexitatea sa este $\mathcal{O}(max) \approx (2^{[\log_2 max]})$, adică o complexitatea exponențială în raport de lungimea reprezentării binare a valorii maxime din tabloul v.

De obicei, această soluție se utilizează când se cunoaște aprioric o limită superioară max pentru elementele tabloului v și valoarea sa este suficient de mică pentru a permite alocarea dinamică a unui tablou cu $max + 1$ elemente (e.g., elementele tabloului sunt numere formate din cel mult 6 cifre, deci $max = 999999$), iar în acest caz se poate renunța la calculul efectiv al valorii max de la începutul funcției. Oricum, și în cazul în care această soluție are complexitatea computațională optimă $\mathcal{O}(n)$, această soluție prezintă dezavantajul utilizării unei cantități mari de memorie suplimentară!

Algoritmul optim pentru rezolvarea acestei probleme este *algoritmul Boyer-Moore*, elaborat în anul 1981 de către informaticienii americani Robert Boyer și J Strother Moore, care au considerat elementele tabloului v ca fiind voturile exprimate de către n alegători, elementul majoritar al tabloului fiind, în acest caz, candidatul câștigător. În articolul publicat în 1981 (https://www.cs.utexas.edu/~boyer/mjrty.pdf), cei doi autori își descriu algoritmul într-un mod foarte sugestiv: "*Imaginați-vă o sală în care s-au adunat toți alegătorii care au participat la vot, iar fiecare alegător are o pancartă pe care este scris numele candidatului pe care l-a votat. Să presupunem că alegătorii se încaieră între ei, iar în momentul în care se întâlnesc față în față doi alegători care au votat candidați diferiți, aceștia se doboară reciproc. Evident, dacă există un candidat care a obținut mai multe voturi decât toate voturile celorlalți candidați cumulate (i.e., un candidat majoritar), atunci alegătorii săi vor câștiga lupta și, la sfârșitul luptei, toți alegătorii rămași în picioare sunt votanți ai candidatului majoritar. Totuși, chiar dacă nu există o majoritate clară pentru un anumit candidat, la finalul luptei pot rămâne în picioare alegători care au votat toți un același candidat, fără ca acesta să fie majoritar. Astfel, dacă la sfârșitul luptei mai există alegători rămași în picioare, președintele adunării trebuie neapărat să realizeze o numărare a tuturor voturilor acordate candidatului votat de alegătorii respectivi pentru a verifica dacă, într-adevăr, el este majoritar sau nu.*"

În implementarea acestui algoritm vom utiliza o variabilă $majoritar$ care va reține candidatul cu cele mai mari șanse de a fi majoritar până în momentul respectiv și un contor $avantaj$ care va reține numărul alegătorilor care au votat cu el și încă nu au fost doborâți de votanți ai altor candidați. Vom prelucra cele n voturi unul câte unul și, notând cu v votul curent, vom actualiza cele două variabile astfel:

- dacă $avantaj == 0$, atunci $majoritar = v$ și $avantaj = 1$ (posibilul candidat majoritar curent nu mai are niciun avantaj, deci el va fi înlocuit de candidatul căruia i-a fost acordat votul curent);
- dacă $avantaj > 0$, atunci verificăm dacă votul curent este pro sau contra posibilului candidat majoritar curent (i.e., $majoritar == v$) și actualizăm contorul $avantaj$ în mod corespunzător.

Dacă la sfârșit, după ce am terminat de analizat toate voturile în modul prezentat mai sus, vom avea $avantaj > 0$, atunci vom realiza o numărare a tuturor voturilor acordate posibilului candidat majoritar rămas (i.e., valoarea variabilei $majoritar$) pentru a verifica dacă, într-adevăr, el este candidatul majoritar:

```
void varianta_5(unsigned int v[] , unsigned int n)
{
    unsigned int majoritar, avantaj, i, k;

    avantaj = 0;
    for(i = 0; i < n; i++)
        if(avantaj == 0)
        {
            majoritar = v[i];
            avantaj = 1;
        }
        else
            if(v[i] == majoritar)
                avantaj++;
            else
                avantaj--;

    if(avantaj == 0)
    {
        printf("Nici un candidat nu a obtinut majoritatea!\n");
        return;
    }
    k = 0;
    for(i = 0; i < n; i++)
        if(v[i] == majoritar)
        {
            k++;
            if(k > n / 2)
            {
                printf("Candidatul %u a obtinut
                                majoritatea!\n", v[i]);
                return;
            }
        }

    printf("Nici un candidat nu a obtinut majoritatea!\n");
}
```

Se observă faptul că algoritmul Boyer-Moore rezolvă această problemă în mod optim, deoarece are complexitatea computațională $\mathcal{O}(n)$ și nu folosește memorie suplimentară!

Determinarea perechilor cu o sumă dată a elementelor

Considerăm un tablou unidimensional p format din n numere naturale aflate în ordine strict crescătoare și un număr natural S. Să se afișeze toate perechile distincte care se pot forma folosind două valori diferite din tabloul p a căror sumă este egală cu S. De exemplu, pentru tabloul $p = (2, 5, 7, 8, 10, 12, 15, 17, 25)$ și $S = 20$, trebuie afișate perechile $(5, 15)$ și $(8, 12)$.

Vom prezenta mai multe variante de rezolvare a acestei probleme, având diferite complexităţi computaţionale, fiecare implementată într-o funcţie având ca parametri tabloul p, numărul n de elemente ale tabloului v şi suma S.

Indiferent de varianta de rezolvare utilizată, datele de intrare pot fi ajustate în scopul eficientizării algoritmului respectiv, astfel:

- eliminăm din tabloul p toate valorile $p[i]$ care sunt mai mari sau egale cu S;
- eliminăm din tabloul p toate valorile $p[i]$ pentru care $p[i] + p[n-1] < S$.

De exemplu, dacă $p = (2, 5, 7, 8, 10, 12, 15, 17, 25, 27)$ şi $S = 25$, atunci, într-o primă fază, putem elimina valorile 25 şi 27 deoarece sunt mai mari sau egale decât S, noul tablou devenind $p = (2, 5, 7, 8, 10, 12, 15, 17)$. Din acest tablou putem elimina valorile 2, 5 şi 7, deoarece ele ar putea să formeze perechi cu suma elementelor egală cu 25 doar cu valorile 23, 20 şi 18, dar acestea sigur nu există în tabloul p!

Ambele operaţii menţionate anterior au complexitatea maximă $\mathcal{O}(n)$, deci nu vor afecta complexitatea algoritmului, deoarece aceasta este cel puţin egală cu $\mathcal{O}(n)$ datorită citirii celor n elemente ale tabloului p:

```
unsigned int *p, n, i, S;

printf("n = ");
scanf("%u", &n);

p = (unsigned int*)malloc(n * sizeof(unsigned int));

for(i = 0; i < n; i++)
    scanf("%u", &p[i]);

printf("S = ");
scanf("%u", &S);

while(n >= 1 && p[n-1] >= S)
    n--;

realloc(p, (n + 1) * sizeof(unsigned int));

i = 0;
while(i < n && p[i] + p[n-1] < S)
    i++;

n = n - i;
memmove(p, p + i, n * sizeof(unsigned int));
realloc(p, n * sizeof(unsigned int));
```

Atenţie, aceste operaţii nu vor scădea complexitatea unui algoritm de rezolvare, ci doar îl vor eficientiza în anumite cazuri particulare!

Într-o primă variantă de rezolvare vom căuta, pentru fiecare element $p[i]$ al tabloului, valoarea sa complementară $p[j]$ față de S (i.e., având proprietatea că $p[i] + p[j] == S$ sau, echivalent, $p[j] == S - p[i]$) în secvența $p[i+1], \ldots, p[n-1]$, folosind o căutare liniară:

```
void varianta_1(unsigned int p[] , unsigned int n , unsigned int S)
{
    unsigned int i, j;

    for(i = 0; i < n - 1; i++)
    {
        if(p[i] >= S/2)
            break;

        for(j = i + 1; j < n; j++)
        {
            if(p[i] + p[j] == S)
            {
                printf("%u %u\n", p[i], p[j]);
                break;
            }

            if(p[i] + p[j] > S)
                break;
        }
    }
}
```

Observați faptul că am eficientizat această variantă de rezolvare prin întreruperea forțată a executării instrucțiunii `for` exterioare în momentul în care valoarea curentă $p[i]$ devine mai mare sau egală decât $S/2$ și prin întreruperea forțată a executării instrucțiunii `for` interioare în momentul în care suma $p[i] + p[j]$ devine cel puțin egală cu S. Totuși, complexitatea maximă a acestei soluții este $\mathcal{O}(n^2)$, cele două îmbunătățiri propuse fiind utile doar în anumite cazuri particulare.

În a doua variantă de rezolvare vom îmbunătăți complexitatea soluției înlocuind căutarea liniară a valorii $S - p[i]$ în secvența $p[i+1], \ldots, p[n-1]$ cu o căutare binară:

```
void varianta_2(unsigned int p[] , unsigned int n , unsigned int S)
{
     unsigned int i , st , dr , m;

     for(i = 0; i < n - 1; i++)
     {
          if(p[i] >= S/2)
           break;

          st = i + 1;
          dr = n - 1;
```

```
            while(st <= dr)
            {
                  m = (st + dr) / 2;
                  if(S - p[i] == p[m])
                  {
                        printf("%u %u\n", p[i], p[m]);
                        break;
                  }
                  else
                        if(S - p[i] < p[m])
                              dr = m - 1;
                        else
                              st = m + 1;
            }
      }
}
```

Deoarece căutarea binară a valorii $S - p[i]$ în secvența $p[i+1], \ldots, p[n-1]$ are complexitatea maximă $\mathcal{O}(\log_2 n)$, această variantă de rezolvare va avea complexitatea computațională maximă egală cu $\mathcal{O}(n \log_2 n)$.

În a treia variantă de rezolvare vom îmbunătăți complexitatea operației de căutare a valorii $S - p[i]$ în secvența $p[i+1], \ldots, p[n-1]$ folosind un tablou de marcaje m cu $p[n-1]+1$ elemente (valoarea $p[n-1]$ este chiar valoarea maximă din tabloul p):

```
void varianta_3(unsigned int p[], unsigned int n, unsigned int S)
{
    unsigned int i;

    char *m = (char*)calloc(p[n-1]+1, sizeof(char));

    for(i = 0; i < n; i++)
        m[p[i]] = 1;

    for(i = 0; i < n; i++)
    {
        if(p[i] >= S/2)
            break;

        if(m[S - p[i]] == 1)
            printf("%u %u\n", p[i], S - p[i]);
    }

    free(m);
}
```

Utilizând tabloul de marcaje m, am scăzut la $\mathcal{O}(1)$ complexitatea operației de căutare a valorii $S - p[i]$ în secvența $p[i+1], \ldots, p[n-1]$, deci această soluție va avea

complexitatea optimă $\mathcal{O}(n)$. Totuși, această variantă poate necesita, în anumite cazuri, o cantitate de memorie suplimentară foarte mare!

Varianta optimă de rezolvare, având complexitatea $\mathcal{O}(n)$ fără a utiliza memorie suplimentară, se bazează pe *metoda arderii lumânării la două capete* (*two pointers*). Astfel, vom parcurge tabloul p simultan din ambele capete, folosind 2 indici st și dr (inițial, $st = 0$ și $dr = n - 1$), în următorul mod:

- dacă $L[st] + L[dr] < S$, atunci $st = st + 1$ deoarece suma elementelor curente este prea mică și putem să o creștem doar trecând la următorul element din partea stânga a listei;
- dacă $L[st] + L[dr] > S$, atunci $dr = dr - 1$ deoarece suma elementelor curente este prea mare și putem să o micșorăm doar trecând la următorul element din partea dreapta a listei;
- dacă $L[st] + L[dr] = S$, atunci afișăm perechea $(L[st], L[dr])$, după care actualizăm ambii indici, respectiv $st = st + 1$ și $dr = dr - 1$.

```
void varianta_4(unsigned int p[], unsigned int n, unsigned int S)
{
    unsigned int st, dr;

    st = 0;
    dr = n - 1;

    while(st < dr)
    {
        if(p[st] + p[dr] == S)
        {
            printf("%u %u\n", p[st], p[dr]);
            st++;
            dr--;
        }
        else
            if(p[st] + p[dr] < S)
                st++;
            else
                dr--;
    }
}
```

Argumentați faptul că această variantă de rezolvare are complexitatea $\mathcal{O}(n)$!

Încheiem acest capitol menționând faptul că nu există metode generale pentru optimizarea sau eficientizarea algoritmilor. În general, acest lucru este posibil prin utilizarea unor structuri de date adecvate, care să permită o implementare eficientă a operațiilor care induc complexitatea algoritmului respectiv.