

Outline

Cuprins

1	Programare Dinamică	1
1.1	Problema Șirului lui Fibonacci	1
1.2	Problema buturugii	6
1.2.1	Implementare	7
1.2.2	Reconstituirea soluției	9
1.3	Problema distanței de editare	10
1.4	Problema plății unei sume de bani folosind număr minim de banc- note	12
1.5	Problema înmulțirii optime a unui șir de matrice	13
1.6	Greedy ca instanță a programării dinamice	15
1.7	Denumirea “Programare Dinamică”	16

1 Programare Dinamică

1.1 Problema Șirului lui Fibonacci

Șirul lui Fibonacci

Fie funcția $f : \mathbb{N} \rightarrow \mathbb{N}$, definită astfel:

$$f(n) = \begin{cases} 0, & \text{dacă } n = 0; \\ 1, & \text{dacă } n = 1; \\ f(n-1) + f(n-2), & \text{dacă } n \geq 2. \end{cases}$$

În mod evident, $f(n)$ este al n -lea număr Fibonacci (numărând de la 0).

Definiția matematică de mai sus a funcției f poate fi transformată ușor într-un program recursiv care calculează al n -lea număr Fibonacci:

```
f(n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return f(n - 1) + f(n - 2);
}
```

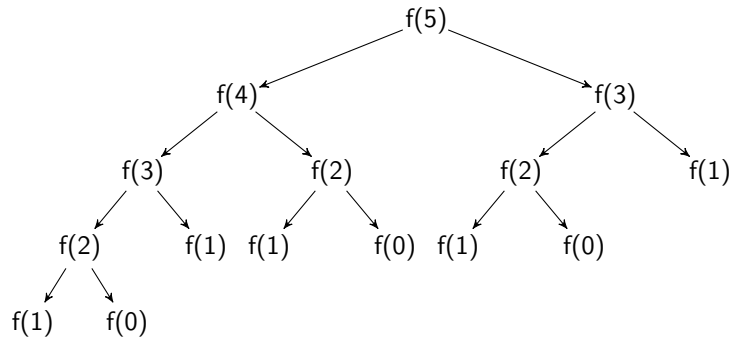
Pentru cazurile $n = 0$ și $n = 1$, timpul de rulare al funcției f este constant. Pentru cazul $n \geq 2$, dacă ignorăm apelurile recursive, timpul de rulare este constant (se face o singură adunare). Când luăm în calcul apelurile recursive, observăm că timpul de calcul necesar este exponențial. În cele ce urmează, vom demonstra acest lucru.

Notăm cu $T(n)$ timpul necesar pentru calculul efectuat de apelul $f(n)$ (pentru $n \in \mathbb{N}$). Avem că:

$T(0) = 1$	caz de bază - număr constant de operații
$T(1) = 1$	caz de bază - număr constant de operații
$T(n) = T(n-1) + T(n-2) + 1$	cazul recursiv.

În cazul recursiv timpul necesar $T(n)$ este dat de timpul necesar celor două apeluri recursive ($T(n-1)$ și respectiv $T(n-2)$) plus 1, reprezentând adunarea.

Putem reprezenta apelurile recursive ale funcției f sub forma unui arbore în care fiecare apel al lui f cu un argument i este un nod $f(i)$. Copiii unui nod $f(i)$ sunt apelurile recursive generate de evaluarea lui $f(i)$. De exemplu, pentru apelurile recursive generate de evaluarea lui $f(5)$, arborele este:

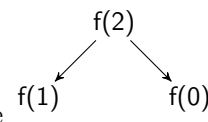


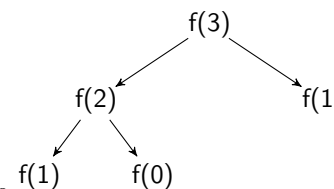
Se poate observa foarte ușor că numărul de noduri din arborele aferent lui $f(n)$ este chiar $T(n)$ (timpul de calcul necesar pentru evaluarea lui $f(n)$).

Frunzele arborelui sunt etichetate sau cu $f(0)$ sau cu $f(1)$ (singurele cazuri de bază – unde nu se fac apeluri recursive). Orice alt nod (care nu este frunză) are exact doi copii.

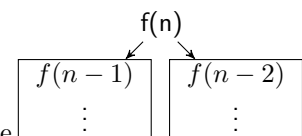
Lemma 1. Pentru orice $n \geq 2$, în arborele care are rădăcina $f(n)$ sunt exact $f(n)$ frunze etichetate cu $f(1)$ și $f(n-1)$ frunze etichetate cu $f(0)$.

Proof. Vom demonstra afirmația prin inducție după n .

- (primul caz de bază) Pentru $n = 2$, arborele  are exact $f(2) = 1$ frunze etichetate cu $f(1)$ și $f(1) = 1$ frunze etichetate cu $f(0)$.

- (al doilea caz de bază) Pentru $n = 3$, arborele  are exact $f(3) = 2$ frunze etichetate cu $f(1)$ și $f(2) = 1$ frunze etichetate cu $f(0)$.

- (cazul inductiv) Știm că arborele asociat lui $f(n-1)$ are $f(n-1)$ frunze etichetate $f(1)$ și $f(n-2)$ frunze etichetate $f(0)$ și că arborele asociat lui $f(n-2)$ are $f(n-2)$ frunze etichetate $f(1)$ și $f(n-3)$ frunze etichetate $f(0)$.

Dar arborele asociat lui $f(n)$ este ; așadar numărul

de frunze etichetate $f(1)$ din arborele $f(n)$ este egal cu numărul de frunze etichetate $f(1)$ din arborele asociat lui $f(n-1)$ plus numărul de frunze etichetate $f(1)$ din arborele asociat lui $f(n-2)$. Dar știm că numărul de frunze etichetate $f(1)$ din arborele asociat lui $f(n-1)$ este $f(n-1)$ și numărul de frunze etichetate $f(1)$ din arborele asociat lui $f(n-2)$ este $f(n-2)$. Așadar în arborele lui $f(n)$ sunt $f(n-1) + f(n-2) = f(n)$ frunze etichetate $f(1)$. Printr-un argument similar obținem că în arborele lui $f(n)$ sunt $f(n-1)$ frunze etichetate $f(0)$.

□

În total, arborele asociat lui $f(n)$ are $f(n+1)$ frunze. ($f(n)$ frunze de $f(1)$ plus $f(n-1)$ frunze de $f(0)$).

Arborele binar asociat lui $f(n)$ se numește plin, deoarece orice nod care nu este frunză are exact doi copii. Într-un arbore binar plin cu x frunze, există exact $x-1$ noduri interne (de ce?).

Deoarece arborele asociat lui $f(n)$ are $f(n+1)$ frunze și este plin, rezultă că are $f(n+1)-1$ noduri interne. În total, are $2f(n+1)-1$ noduri. Dar numărul de noduri din acest arborele este exact $T(n)$, deci putem concluziona că

$$T(n) = 2f(n+1) - 1.$$

Putem arăta ușor prin inducție că

$$f(n) \geq 1.5^{n-2}$$

pentru orice $n \geq 2$. Așadar $f(n) = o(c^n)$; cum $T(n) = 2f(n+1) - 1$, rezultă și că

$$T(n) = o(c^n),$$

adică că timpul necesar pentru evaluarea lui $f(n)$ este exponențial în n .

Sursa acestei ineficiențe este calcularea repetată a valorilor $f(i)$, unde $0 \leq i \leq n$. De exemplu, în arborele asociat lui $f(5)$, observăm că:

1. $f(1)$ este evaluat de 5 ori
2. $f(2)$ este evaluat de 3 ori
3. $f(3)$ este evaluat de 2 ori
4. $f(4)$ este evaluat de 1 ori
5. $f(5)$ este evaluat de 1 ori

În general, în cursul evaluării lui $f(n)$, valoarea $f(k)$ este calculată de $f(n-k+1)$ ori ($1 \leq k \leq n$).

Exercițiul 1. Demonstrați acest lucru (indicație: pentru $k=1$ am arătat deja mai devreme).

O variantă mai eficientă

O variantă pentru a optimiza algoritmul de mai sus este să evităm recalcularea valorilor $f(i)$ ($1 \leq i \leq n$) în felul următor:

1. La primul apel al funcției f cu un anumit argument i , se calculează valoarea $f(i)$ în mod obișnuit și se memorează rezultatul (de exemplu într-un vector x , pe poziția i).
2. La următoarele apeluri ale funcției f pentru acest argument i , se știe că $f(i)$ a fost deja calculat; așadar nu se recalculează nimic și se întoarce direct $x[i]$.

Avem nevoie de o metodă pentru a verifica dacă $x[i]$ a fost deja calculat sau nu. Presupunem că vectorul x a fost inițializat cu valoarea -1 în toate pozițiile; valoarea -1 la poziția i indică faptul că $x[i]$ nu a fost calculat încă.

Obținem așadar următorul algoritm:

```
// presupunem x[0..n] - vector global, initializat cu -1
// in toate pozitiile
f(n)
{
    if (x[n] != -1) // am calculat deja f(n)
        return x[n]; // (A)
    else {
        if (n == 0)
            // (B)
            return x[0] = 0; // marchez x[0] = 0 si intorc 0
        else if (n == 1)
            // (C)
            return x[1] = 1; // marchez x[1] = 1 si intorc 1
        else
            // (D)
            return x[n] = f(n - 1) + f(n - 2);
            // calculez f(n - 1) + f(n - 2) (valoarea f(n))
            // marchez x[n] ca fiind calculat
            // retin valoarea f(n) = f(n - 1) + f(n - 2) in x[n]
            // intorc x[n]
    }
}
```

Exercițiul 2. Cum (în ce ordine) se “umple” vectorul x pe măsura evaluării expresiei $f(5)$?

Vrem să calculăm timpul de execuție al funcției f . Presupunând că vectorul x a fost deja inițializat cu -1 în toate pozițiile și că tocmai am apelat $f(m)$ pentru un număr natural fixat m , calculăm de câte ori se execută liniile A, B, C și D:

1. linia B se execută cel mult o dată, deoarece pentru a se executa trebuie ca $n = 0$ și $x[0] = -1$. După execuția liniei B, $x[0]$ devine 0 și deci nu mai sunt îndeplinite condițiile pentru ca programul să ajungă în punctul respectiv.

2. linia **C** se execută cel mult o dată, deoarece pentru a se executa trebuie ca $n = 1$ și $x[1] = -1$. După execuția liniei **C**, $x[1]$ devine 1 și deci nu mai sunt îndeplinite condițiile pentru ca programul să ajungă în punctul respectiv.

3. linia **D** se execută cel mult o dată pentru fiecare valoare posibilă a argumentului n al funcției f , deoarece pentru a se ajunge la **D**, este necesar ca $x[n] = -1$. Dar după execuția lui **D**, $x[n]$ primește o valoare nenegativă.

Cum argumentul n variază între 0 și m , ajungem la concluzia că linia **D** este executată de maxim $O(m)$ ori.

4. linia **A** se execută cel mult o dată pentru fiecare apel recursiv al funcției f (apelul inițial, $f(m)$, va executa linia **D**, nu linia **A**). Dar singurele apeluri recursive ale lui f sunt cele generate pe linia **D**. Am văzut deja că linia **D** este executată cel mult o dată pentru fiecare valoare $0 \leq n \leq m$; fiecare astfel de apel $f(n)$ generează două apeluri recursive: $f(n-1)$ și $f(n-2)$. Așadar există cel mult $2m$ astfel de apeluri. Deci linia **A** se execută de $O(m)$ ori.

În total, pentru calculul lui $f(m)$, sunt făcuți $O(1) + O(1) + O(m) + O(m) = O(m)$ pași, ceea ce înseamnă ca algoritmul este liniar.

Tehnica de optimizare prezentată mai sus se numește *memoizare* (este corect fără “r”). Prin memoizare se înțelege faptul că rezultatele întoarse de o funcție sunt păstrate în memorie; dacă funcția este apelată pentru a doua oară pe același argument, nu se face calculul din nou, ci rezultatul este întors direct din locul în care a fost memorat.

Tehnica de memoizare poate fi folosită pentru orice funcție recursivă pură (care nu are efecte secundare – în particular, pentru aceleași date de intrare, întoarce aceleași rezultate).

Algoritmul iterativ

Observăm că, exceptând valorile $x[0]$ și $x[1]$, vectorul x este umplut de la stânga la dreapta (adica în ordinea $x[2], x[3], x[4], \dots$). Putem profita de această observație pentru a transforma algoritmul recursiv într-unul iterativ:

```
f(n)
{
  x[0] = 0;
  x[1] = 1;
  for (i = 2; i <= n; ++i) {
    x[i] = x[i - 1] + x[i - 2];
  }
  return x[i];
}
```

Timpul de rulare este evident $O(n)$.

Algoritmul de mai sus este un exemplu de programare dinamică. Pentru a rezolva o anumită problemă folosind programarea dinamică, trebuie să identificăm în mod convenabil mai multe subprobleme. În cazul șirului lui Fibonacci,

problema inițială este să calculăm $f(m)$ (pentru un anumit număr m). Subproblemele sunt: calculul lui $f(i)$, pentru fiecare $0 \leq i \leq m$ (deci sunt $m + 1$ subprobleme).

După ce identificăm subproblemele, trebuie stabilit cum se poate calcula soluția unei subprobleme în funcție de alte subprobleme. În cazul șirului lui Fibonacci, pentru a rezolva problema $f(i)$, este suficient să rezolvăm problema $f(i - 1)$, problema $f(i - 2)$ și să adunăm rezultatele. (În cazul altor algoritmi de programare dinamică, modul în care se calculează soluția unei subprobleme în funcție de soluțiile altor subprobleme este în general mai complicat.) Pentru subproblemele $f(0)$ și $f(1)$, soluția este imediată (0 și respectiv 1). După ce am ales în mod convenabil subproblemele și am identificat cum se poate rezolva o subproblemă în funcție de soluțiile altor subprobleme, este suficient să rezolvăm subproblemele “pe rând”. În cazul problemei de mai sus, ordinea în care se rezolva subproblemele este foarte simplă: întâi $f(0)$, apoi $f(1)$, apoi $f(2)$, etc.

Exemplul în care am folosit programarea dinamică pentru a calcula al m -lea element din șirul lui Fibonacci este pur didactic. În general, programarea dinamică se aplică pentru probleme de optimizare pentru care algoritmi greedy nu produc soluția optimă.

1.2 Problema buturugii

Se dă o buturugă care are n metri și care poate fi secționată în 1 sau mai multe (cel mult n) buturugi de lungime număr întreg. Secțiunile tăiate pot fi vândute, iar o buturugă de dimensiune i ($1 \leq i \leq n$) costă $a[i]$ RON pe piață. Care este câștigul optim care poate fi obținut prin secționarea buturugii inițiale de n metri?

De exemplu, pentru $n = 4$, $a[1] = 100$, $a[2] = 250$, $a[3] = 350$, $a[4] = 350$, cea mai eficientă secționare este $4 = 2 + 2$, care aduce un câștig de 500. În general, exist 2^{n-1} moduri de a secționa o buturugă de dimensiune n , deoarece există $n - 1$ puncte în care poate fi tăiată (sau nu), iar tăieturile sunt independente una de cealaltă. De exemplu, pentru $n = 4$, cele 8 moduri, împreună cu câștigul aferent, sunt:

1. $4 = 1 + 3$ (câștig 450)
2. $4 = 1 + 1 + 2$ (câștig 450)
3. $4 = 1 + 1 + 1 + 1$ (câștig 400)
4. $4 = 1 + 2 + 1$ (câștig 450)
5. $4 = 2 + 2$ (câștig 500)
6. $4 = 2 + 1 + 1$ (câștig 450)
7. $4 = 3 + 1$ (câștig 450)
8. $4 = 4$ (câștig 350)

În general, dacă buturuga are n metri, există 2^{n-1} moduri de a o tăia: putem alege independent, pentru fiecare dintre cele $n - 1$ locuri unde poate fi secționată, dacă facem tăietura în punctul respectiv sau nu (2 variante pentru fiecare punct). Deci există un număr exponențial (în funcție de lungimea buturugii) de variante

de a o tăia. Chiar dacă eliminăm variantele în care secțiunile se repetă, dar în altă ordine (cum ar fi $1 + 1 + 2, 1 + 2 + 1, 2 + 1 + 1$), numărul de posibilități rămâne exponențial (obținem numărul lui Catalan – google it).

Așadar, orice rezolvare a problemei care enumeră toate variantele de tăiere, calculează câștigul corespunzător, și păstrează maximum dintre toate variantele va avea o complexitate timp în cazul cel mai nefavorabil exponențială în n .

Vom vedea cum putem obține un algoritm mai eficient folosind programare dinamică. În acest scop, vom nota cu $d(i)$ câștigul maxim care poate fi obținut prin tăierea (într-un mod cât mai convenabil) a unei buturugi de lungime i (unde $0 \leq i \leq n$). Prin convenție, vom alege $d(0) = 0$ (buturuga “vidă” nu poate aduce niciun fel de profit).

Soluția problemei este numărul $d(n)$, unde n este dimensiunea buturugii inițiale. Putem găsi o formulă de calcul (recursivă) pentru $d(i)$ ($1 \leq i \leq n$)?

Fie $i = x_1 + x_2 + \dots + x_k$ o descompunere optimă pentru i . Avem $1 \leq x_1 \leq i$ (dacă $x_1 = i$, atunci $k = 1$ și în descompunerea optimă, buturuga nu este secționată).

Proprietatea de substructură optimă. Deoarece $x_1 + x_2 + \dots + x_k$ este o descompunere optimă pentru o buturugă de dimensiune i , înseamnă că $x_2 + x_3 + \dots + x_k$ este o descompunere optimă pentru o buturugă de dimensiune $i - x_1$. Dacă nu ar fi așa, adică dacă ar exista o descompunere $b_1 + b_2 + \dots + b_l$ pentru $i - x_1$ cu câștig mai bun decât $x_2 + x_3 + \dots + x_k$, atunci $x_1 + b_1 + b_2 + \dots + b_l$ ar fi o descompunere pentru i care ar aduce un câștig mai mare decât descompunerea $x_1 + x_2 + \dots + x_k$, ceea ce contrazice ipoteza de la care am pornit. (Această proprietate este proprietatea de substructură optimă pentru problema buturugii: într-o soluție optimă pentru o buturugă de dimensiune i , subproblema $i - x_1$ are și ea o soluție optimă).

Folosindu-ne de proprietatea de mai sus, putem observa că

$$d(i) = a[x_1] + d(i - x_1).$$

Singura problemă este că nu știm valoarea lui x_1 . Dar nu există decât i valori posibile pentru x_1 , deoarece știm că $1 \leq x_1 \leq i$ (x_1 este lungimea primei secțiuni dintr-o descompunere a unei buturugi de lungime i). Care este cea mai convenabilă valoare pentru x_1 , ținând cont că vrem să maximizăm profitul pentru buturuga de dimensiune i ? În mod obligatoriu, $x_1 \in \{1, 2, \dots, i\}$ este numărul care maximizează valoarea $a[x_1] + d(i - x_1)$. Folosindu-ne de observația de mai sus, obținem:

$$d(i) = \max_{x_1 \in \{1, 2, \dots, i\}} (a[x_1] + d(i - x_1)).$$

Am obținut, folosindu-ne de proprietatea de substructură optimă, o relație de recurență pentru soluțiile $d(i)$ ale subproblemelor $0 \leq i \leq n$ (subproblema i este: care este câștigul maxim adus de o buturugă de dimensiune i ?).

1.2.1 Implementare

Putem implementa direct recurența într-o funcție recursivă:

```
d(i)
{
```

```

    if (i == 0) {
        return 0;
    } else {
        result = a[1] + d(i - 1);
        for (int j = 2; j <= i; ++j) {
            if (a[j] + d(i - j) > result) {
                result = a[j] + d(i - j);
            }
        }
        return result;
    }
}

```

Din păcate, funcția recursivă de mai sus are complexitate timp exponențială, din cauza apelurilor recursive ale funcției care se repetă.

Exercițiul 3. Calculați câte apeluri recursive ale lui $d(0)$ se execută în cursul unui apel $d(n)$.

Putem aplica tehnica de memoizare pentru a obține un algoritm pătratic:

```

int m[]; // presupunem ca initial contine doar -1
d(i)
{
    if (m[i] == -1) {
        if (i == 0) {
            m[i] = 0;
            return m[i];
        } else {
            result = a[1] + d(i - 1);
            for (int j = 2; j <= i; ++j) {
                if (a[j] + d(i - j) > result) {
                    result = a[j] + d(i - j);
                }
            }
            m[i] = result;
            return m[i];
        }
    } else {
        return m[i];
    }
}

```

Observați că această tehnică se poate aplica mecanic: fiecare “return ...” este înlocuit cu $m[i] = \dots$, iar corpul C al funcției este protejat cu o instrucțiune condițională în felul următor:

```

    if (m[i] == -1) {
        C
        return m[i];
    } else {
        return m[i];
    }

```


De altfel, există limbaje de programare (cum ar fi Haskell) care fac această transformare de optimizare a timpului de execuție (memoizarea) automat.

Având varianta cu memoizare, putem observa că tabloul $m[]$ este completat în ordinea: $m[0], m[1], \dots$. Ținând cont de acest lucru, putem foarte ușor renunța la recursivitate și să obținem un algoritm iterativ echivalent:

```
m[];
d(n)
{
    m[0] = 0;
    for (i = 1; i <= n; ++i) {
        m[i] = a[1] + m[i - 1];
        for (j = 2; j <= i; ++j) {
            if (m[i] < a[j] + m[i - j]) {
                m[i] = a[j] + m[i - j];
            }
        }
    }
    return m[n];
}
```

Complexitatea-timp a algoritmului iterativ este evident $O(n^2)$. De altfel, din punct de vedere al timpului asimptotic de execuție, algoritmul recursiv (cel cu memoizare) are tot complexitate $O(n^2)$ (exercițiu: verificați această afirmație), dar în practică algoritmul iterativ este mai rapid deoarece “constanta din O -notație” este mai mică.

1.2.2 Reconstituirea soluției

Algoritmul de mai sus poate fi folosit pentru calculul câștigului optim. Totuși, algoritmul nu returnează cum anume trebuie secționată buturuga pentru a obține acest câștig. Putem modifica ușor algoritmul pentru a returna și soluția optimă, pe lângă câștigul adus de aceasta.

Este suficient să memorăm, pentru fiecare dimensiune i , care este lungimea x_1 a primei secțiuni dintr-o tăiere optimă $x_1 + x_2 + \dots + x_k$ a lui i :

```
m[]; // m[i] va fi castigul maxim adus de o buturuga de dimensiune i
prima[]; // prima[i] va fi lungimea x_1 a primei sectiuni dintr-o
        // descompunere optima a lui i.
d(n)
{
    descompunere = [];
    m[0] = 0;
    for (i = 1; i <= n; ++i) {
        m[i] = a[1] + m[i - 1];
        prima[i] = 1;
        for (j = 2; j <= i; ++j) {
            if (m[i] < a[j] + m[i - j]) {
                m[i] = a[j] + m[i - j];
                prima[i] = j;
            }
        }
    }
}
```

```

    }
}
// calculeaza in vectorul descompunere lungimile sectiunilor
// dintr-o descompunere optima a unei buturugi de dimensiune n
while (n > 0) {
    descompunere.add(prima[n]);
    n -= prima[n];
}
return descompunere;
}

```

1.3 Problema distanței de editare

Problema distanței de editare

O astfel de problemă de optimizare este problema distanței de editare.

În contextul problemei distanței de editare, trebuie să transformăm un șir într-un alt șir într-un număr minim de operații numite operații de editare:

1. *substituția*: transformăm o literă în altă literă. Exemplu: FOOD se transformă printr-o operație de substituție în FOND
2. *inserarea*: adăugăm o literă. Exemplu: FOOD se transformă în FROOD printr-o inserare.
3. *ștergerea*: ștergem o literă. Exemplu: FOOD se transformă în FOO printr-o ștergere.

Formal, problema distanței de editare (engl. EDIT DISTANCE problem) este următoarea:

Input: Două șiruri $s[0..n-1]$ și $t[0..m-1]$

Output: Cel mai mic număr de operații de editare (substituție, inserare, ștergere) necesare pentru a transforma șirul s în t (număr numit și distanța Levenshtein dintre cele două șiruri).

Deoarece în problemă se cere să se minimizeze numărul de operații, avem de a face o problemă de optimizare. Pentru datele de intrare $s = \text{FOOD}$ și $t = \text{MONEY}$, o soluție de cost 4 este:

```

FOOD  →      substituție
MOOD  →      substituție
MOND  →      inserție
MONED →      substituție
MONEY →

```

Exercițiu: arătați că nu există o altă soluție de cost mai mic decât 4.

Concluzionați că soluția optimă are cost 4.

Putem reprezenta soluția de mai sus și sub o altă formă, mai convenabilă:

<i>F</i>	<i>O</i>	<i>O</i>	-	<i>D</i>
<i>M</i>	<i>O</i>	<i>N</i>	<i>E</i>	<i>Y</i>

Un spațiu care apare pe prima linie indică o inserție, două litere diferite indică o substituție și un spațiu pe a doua linie indică o ștergere. Pentru a calcula costul soluției este suficient să numărăm coloanele care au conținut diferit (încăzul de mai sus sunt 4 astfel de coloane).

Un exemplu de soluție (care este de cost 3 și nu este optimă) pentru șirurile $ABCD$ și $ADBC$:

A	B	C	D
A	D	B	C

Soluția optimă (de cost 2) este:

A	-	B	C	D
A	D	B	C	-

Dacă eliminăm ultima coloană dintr-o soluție optimă, obținem o nouă soluție optimă (pentru alte șiruri). De exemplu,

A	-	B	C
A	D	B	C

este o soluție optimă pentru șirurile ABC și $ADBC$. Această proprietate se numește principiul de *substructură optimă*:

Lemma 2. (*Proprietatea de substructură optimă*)

Fie $\begin{array}{|c|c|c|c|} \hline L_1 & \dots & L_{k-1} & L_k \\ \hline L'_1 & \dots & L'_{k-1} & L'_k \\ \hline \end{array}$ o soluție optimă (pentru cele două șiruri obținute din $L_1 \dots L_k$ și $L'_1 \dots L'_k$ prin eliminarea spațiilor). Atunci $\begin{array}{|c|c|c|} \hline L_1 & \dots & L_{k-1} \\ \hline L'_1 & \dots & L'_{k-1} \\ \hline \end{array}$ e o soluție optimă (pentru cele două șiruri obținute din $L_1 \dots L_{k-1}$ și $L'_1 \dots L'_{k-1}$ prin eliminarea spațiilor).

Exercițiu: demonstrați proprietatea de mai sus.

Ne putem folosi de proprietatea de mai sus pentru a defini recursiv distanța Levenshtein între prefixul $s[0..l-1]$ de lungime l a lui $s[0..n-1]$ și $t[0..k-1]$ de lungime k a lui $t[0..m-1]$. Vom nota cu $d(l, k)$ această distanță (distanța de editare dintre prefixul de lungime l al lui s și prefixul de lungime k al lui t).

Dacă $l = 0$, atunci $d(l, k) = k$ (de ce?); dacă $k = 0$, atunci $d(l, k) = l$ (de ce?). Dacă $l > 0$ și $k > 0$, atunci:

1. (inserție) dacă pe prima linie și ultima coloană se află un spațiu, atunci s-a efectuat o inserție și $d(l, k) = d(l, k-1) + 1$. Costul de +1 se datorează costului inserției (s-a inserat elementul $t[k-1]$). Proprietatea de substructură optimă ne spune că dacă ștergem ultima coloană, obținem o soluție optimă pentru celelalte coloane cu excepția ultimei. Dar această soluție are cost $d(l, k-1)$.
2. (ștergere) un argument similar cu cel de mai sus ne arată că dacă ultima coloană de pe a doua linie conține un spațiu, atunci $d(l, k) = d(l-1, k) + 1$.
3. (substituție) dacă ultima coloană conține două caractere diferite, atunci s-a efectuat o substituție. Așadar $d(l, k) = d(l-1, k-1) + 1$. Costul de +1 se datorează substituției și $d(l-1, k-1)$ este costul obținut după ștergerea ultimei coloane.
4. (nicio schimbare) dacă ultima coloană conține două caractere egale, atunci nu s-a efectuat nicio operație. Așadar $d(l, k) = d(l-1, k-1)$.

Cum nu există nicio altă posibilitate (nu pot fi două spații într-o soluție optimă), rezultă că $d(l, k)$ trebuie să fie cea mai mică valoare dintre posibilitățile de mai sus:

$$d(l, k) = \min \begin{cases} d(l, k-1) + 1, \\ d(l-1, k) + 1, \\ d(l-1, k-1) + 1, & (\text{dacă } s[l-1] \neq t[k-1]) \\ d(l-1, k-1) & (\text{dacă } s[l-1] = t[k-1]) \end{cases}$$

Putem memoiza recurența de mai sus pentru a obține un algoritm de complexitate $O(n \times m)$ pentru problema distanței de editare (exercițiu pentru acasă) sau putem obține următorul algoritm iterativ:

```
// input: s[0..n-1], t[0..m-1]
// output: distanta de editare intre s si t
for (i = 0; i <= n; ++i) {
    d[i][0] = i;
}
for (j = 0; j <= m; ++j) {
    d[0][j] = j;
}
for (i = 1; i <= n; ++i) {
    for (j = 1; j <= m; ++j) {
        if (s[i-1] == t[j-1]) {
            d[i][j] = min(d[i-1][j] + 1,          // (*)
                          d[i][j-1] + 1,          // (**)
                          d[i-1][j-1]);
        } else {
            d[i][j] = min(d[i-1][j] + 1,
                          d[i][j-1] + 1,
                          d[i-1][j-1] + 1);
        }
    }
}
return d[n][m];
```

Exercițiu: arătați că liniile (*) și (**) nu sunt necesare.

Exercițiul 4. Arătați cum se poate reconstitui soluția (operațiile de ștergere/inserare/substituție) pornind de la matricea d completată de algoritm.

1.4 Problema plății unei sume de bani folosind număr minim de bancnote

Problema plății unei sume de bani folosind număr minim de bancnote

Input: $n \in \mathbb{N}, k \in \mathbb{N} \setminus \{0\}, b_1, \dots, b_k \in \mathbb{N} \setminus \{0\}$ astfel încât $b_1 = 1$.

Output: Numărul minim de bancnote de tipurile b_1, b_2, \dots, b_k necesare pentru a plăti suma n .

Exemplu: Dacă $k = 3, b_1 = 1, b_2 = 5, b_3 = 10$, output-ul corect pentru $n = 17$ este 4 deoarece folosim o bancnotă de 10, una de 5 și două de 1.

Exercițiul 5. Arătați că algoritmul greedy nu produce tot timpul soluția optimă (încercați cu diferite valori pentru b_1, \dots, b_k).

Arătați proprietatea de substructură optimă: dacă $b_{i_1} + \dots + b_{i_l} = n$ este o descompunere optimă a lui n (i.e. n se poate plăti optim cu l bancnote: b_{i_1}, \dots, b_{i_l}) atunci $b_{i_1} + \dots + b_{i_{l-1}} = n - b_{i_l}$ este o descompunere optimă pentru $n - b_{i_l}$ (i.e. $n - b_{i_l}$ se poate plăti optim cu $l - 1$ bancnote: $b_{i_1}, \dots, b_{i_{l-1}}$).

Exercițiul 6. Găsiți o relație de recurență pentru șirul d_n ($n \in \mathbb{N}$), unde d_n este numărul minim de bancnote necesar pentru a plăti suma n .

Exercițiul 7. Folosiți relația de recurență pentru a implementa:

1. un algoritm recursiv (exponențial) care calculează d_n ;
2. un algoritm recursiv, memoizat, care calculează d_n (arătați că algoritmul este liniar în n);
3. un algoritm iterativ, liniar în n , care calculează d_n .

Un astfel de algoritm, care este polinomial în datele de intrare ($O(n^1)$) și nu în dimensiunea datelor de intrare (n este reprezentat pe $\log_2(n)$ biți, deci $O(n) = O(2^{\log_2(n)}) = O(2^d)$ – exponențial în $d =$ dimensiunea datelor de intrare) se numește pseudo-polinomial. În cadrul seminarului trebuie să găsiți un algoritm pseudopolinomial pentru problema (discretă a) rucsacului.

1.5 Problema înmulțirii optime a unui șir de matrice

Calcula produsul R a două matrice A și B de dimensiune $n \times m$ și respectiv $m \times l$, putem folosi următorul algoritm:

```
mult(A, B)
{
    for (i = 0; i < n; ++i) {
        for (j = 0; j < k; ++j) {
            R[i][j] = 0;
            for (k = 0; k < l; ++k) {
                R[i][j] += A[i][k] * A[k][j];
            }
        }
    }
}
```

Matricea rezultată este de dimensiune $n \times l$ și complexitatea-timp pentru calculul ei este de $n \times m \times l$. Notăm cu $T(n, m, l) = n \times m \times l$ timpul necesar calculului produsului a doua matrice de dimensiuni $n \times m$ și respectiv $m \times l$.

Dacă avem de înmulțit mai mult de două matrice, ne putem folosi de asociativitatea înmulțirii matricilor: pentru orice matrice M_1, M_2, M_3 ,

$$M_1 \times (M_2 \times M_3) = (M_1 \times M_2) \times M_3.$$

Notă. Am presupus mai sus că numărul de coloane ale matricei M_1 este egal cu numărul de coloane ale matricei M_2 și că numărul de coloane ale matricei M_2 este egal cu numărul de linii ale matricei M_3 (altfel înmulțirea nu este definită).

Să considerăm că M_1 are dimensiuni 100×1000 , M_2 are dimensiune 1000×10000 și M_3 are dimensiune 10000×10 . Dacă calculăm $(M_1 \times M_2) \times M_3$, timpul de calcul este $T(100, 1000, 10000) + T(100, 10000, 10) = 1000000000 + 10000000 = 1000100000$ iar dacă calculăm $M_1 \times (M_2 \times M_3)$, timpul de execuție este $T(1000, 10000, 10) + T(100, 1000, 10) = 100000000 + 1000000 = 101000000$. Observăm așadar că a doua variantă este mult mai rapidă.

Problema înmulțirii optime a unui șir de matrice este: dându-se dimensiunile a n matrice M_1, M_2, \dots, M_n , să se determine o parantezare pentru $M_1 \times M_2 \times \dots \times M_n$ care să minimizeze timpul de calcul. Notăm cu $l[i]$ numărul de linii ale matricei M_i și cu $c[i]$ numărul de coloane ale matricei M_i . Pentru ca operația de înmulțire să fie definită, presupunem că $c[i] = l[i+1]$ pentru orice $1 \leq i \leq n-1$.

Enumerarea tuturor parantezărilor este prea costisitoare (sunt un număr exponențial de parantezări). Putem găsi o soluție bazată pe programare dinamică, folosindu-ne de următoarele subprobleme:

- pentru orice $1 \leq i \leq j \leq n$, care este timpul minim de calcul pentru a calcula $M_i \times M_{i+1} \times \dots \times M_j$?

Notăm cu $d(i, j)$ costul minim al calculului $M_i \times M_{i+1} \times \dots \times M_j$.

Fie $(M_i \dots M_k) \times (M_{k+1} \dots M_j)$ modul optim de a calcula $M_i \times \dots \times M_j$ (adică calculăm întâi $M_i \times M_{i+1} \times \dots \times M_k$ (nu știm cum), apoi $M_{k+1} \times M_{k+2} \times \dots \times M_j$ (nu știm cum), iar apoi produsul lor. Atunci are loc identitatea

$$d(i, j) = d(i, k) + d(k+1, j) + T(l[i], c[k], c[j]).$$

Exercițiul 8. Enunțați proprietatea de substructură optimă pentru problema înmulțirii optime a unui șir de matrice, proprietate care justifică identitatea de mai sus.

Dar nu știm cât este k . Totuși, cu siguranță este un număr natural între i și $j-1$. Așadar este suficient să alegem numărul $k \in \{i, i+1, \dots, j-1\}$ care minimizează partea dreaptă a identității de mai sus. Obținem o definiție recursivă pentru $d(i, j)$:

$$d(i, j) = \max_{k \in \{i, i+1, \dots, j-1\}} (d(i, k) + d(k+1, j) + T(l[i], c[k], c[j])).$$

Identitatea de mai sus este adevărată pentru $i < j$. Pentru $i = j$, convenim ca $d(i, i) = 0$ (nu ne costă nimic să calculăm M_i).

Recurența de mai sus poate fi implementată direct, caz în care complexitatea-timp va fi exponențială deoarece apar mai multe apeluri pentru aceleași subprobleme $d(i, j)$, dar prin memoizare se obține un algoritm de complexitate $O(n^3)$ (așadar, polinomial). Algoritmul memoizat poate fi transformat într-un algoritm iterativ observând că valorile $d(i, j)$ sunt calculate începând cu $j-i=1$, apoi $j-i=2$, ș.a.m.d. Așadar, dacă păstrăm valorile $d(i, j)$ într-o matrice, vom calcula întâi diagonală principală a matricei și diagonală aflată imediat deasupra și la dreapta de diagonală principală, ș.a.m.d. (se calculează doar triunghiul de “deasupra” diagonalei principale deoarece avem nevoie doar de $d(i, j)$, unde $i \leq j$).

Exercițiul 9. Implementați varianta memoizată și varianta iterativă a algoritmului de PD pentru problem înmulțirii optime a șirului de matrice.

Exercițiul 10. Modificați algoritmul de mai sus pentru a putea reconstitui și parantezarea optimă.

1.6 Greedy ca instanță a programării dinamice

Dacă ați fost atenți, ați observat că există câteva asemănări între greedy și programare dinamică. În particular, în ambele cazuri apare noțiunea de subproblemă și proprietatea de substructură optimă. De fapt, tehnica greedy poate fi gândită ca un caz particular de programare dinamică, unde rezolvarea unei probleme este determinată direct de alegerea greedy, nefiind nevoie de a enumera toate alegerile posibile.

Pentru a înțelege mai bine acest lucru, reconsiderăm problema selecției activităților: dându-se n activități, prin timpul lor de început $s[0..n-1]$ și timpul de final $f[0..n-1]$, să se determine numărul maxim de activități care nu se suprapun două câte două.

Fie $S(i, j) = \{k \in \{0, \dots, n-1\} \mid f[i] \leq s[k] \wedge f[k] \leq s[j]\}$, mulțimea de activități care încep după activitatea i și se termină înainte de activitatea j . Notăm cu $d(i, j)$ numărul maxim de activități din $S(i, j)$ compatibile două câte două între ele. Răspunsul final al problemei este $2 + \max_{x, y \in \{0, \dots, n-1\}} (d(x, y))$.

Cum putem calcula $d(i, j)$? Dacă $S(i, j) = \emptyset$, atunci evident $d(i, j) = 0$.

Dacă $S(i, j)$ nu este mulțimea vidă, atunci evident $d(i, j) > 0$ deoarece putem selecta din $S(i, j)$ măcar o activitate. Fie $x \in S(i, j)$ o activitate care apare într-o soluție optimă pentru $d(i, j)$. Atunci avem că

$$d(i, j) = 1 + d(i, x) + d(x, j). \quad (1)$$

Din păcate, din nou, nu știm cine este x . Dar e suficient să enumerăm toate activitățile $x \in S(i, j)$ și să alegem x -ul care maximizează partea dreaptă a identității de mai sus. Așadar obținem recurența:

$$d(i, j) = \max_{x \in S(i, j)} (1 + d(i, x) + d(x, j)). \quad (2)$$

Relația de recurență de mai sus poate fi transformată prin memoizare într-un algoritm de PD pentru problema selecției activităților. Totuși, observați că pentru a rezolva subproblema $d(i, j)$, suntem nevoiți să parcurgem toate *alegerile* $x \in S(i, j)$.

Proprietatea de *alegere greedy* pe care am văzut-o în cursul precedent ne asigură că, dacă alegem x ca fiind activitatea din $S(i, j)$ care se termină cel mai devreme, atunci partea dreaptă a Identității (1) este maximizată. Așadar, putem optimiza Identitatea (2) astfel:

$$d(i, j) = 1 + d(i, \text{first}(S(i, j))) + d(\text{first}(S(i, j)), j),$$

unde *first* este funcția care întoarce activitatea care se termină cel mai devreme dintre activitățile $S(i, j)$.

Observăm că, în mod necesar, $d(i, \text{first}(S(i, j))) = 0$. Așadar, forma finală devine:

$$d(i, j) = 1 + d(\text{first}(S(i, j)), j),$$

care corespunde exact algoritmului greedy din cursul precedent.

Așadar, diferența principală între greedy și programare dinamică este că la greedy alegerea pe care o facem în rezolvarea unei subprobleme este dictată de alegerea greedy, în timp ce pentru programare dinamică trebuie să enumerăm toate posibilitățile, alegând-o pe cea mai convenabilă.

1.7 Denumirea “Programare Dinamică”

Programarea dinamică nu trebuie confundată cu alocarea dinamică. Tehnica programării dinamice a fost descoperită de Richard Bellman în anii 1950. Pentru o istorie amuzantă privind alegerea denumirii “programare dinamică”, puteți consulta cartea acestuia, “Dynamic Programming”. Cuvântul “programare” din sintagma “programare dinamică” trebuie înțeles în sensul mai vechi, cel tabelar, și nu în sensul modern de program software pentru calculator. Cuvântul “dinamic” se referă la modul în care este construit “programul”, și anume puțin câte puțin (dinamic), în funcție de valorile deja calculate.