Tema 1 - Proiectarea Algoritmilor

Boldeanu Ana-Maria
Grupa 321CD

Universitatea Politehnica București
Facultatea de Automatică și Calculatoare

Anul universitar 2021-2022

1 Tehnica Divide et Impera

1.1 Enunțul problemei

Găsirea celui mai lung prefix comun (Longest Common Prefix sau LCP): Se dă o mulțime de N șiruri de caractere. Se cere găsirea prefixului de lungime maximă, adică a prefixului comun tuturor șirurilor din mulțime.

1.2 Descrierea soluției

Pentru rezolvarea problemei, se împarte mulțimea în două părți egale. Apoi se procedează la fel cu fiecare submulțime rezultată și se continuă tot așa până când se ajunge la submulțimi de un singur element (*Divide*).

La întoarcerea din recursivitate, se determină la fiecare pas șirul care reprezintă prefixul comun dintre subșirul stâng și subșirul drept (*Impera & Combină*), urmând ca la sfârșit să se obțină prefixul comun al tuturor șirurilor din mulțimea inițială.

1.3 Algoritmul de rezolvare

Vezi Algoritmul 1.

1.4 Complexitate

Fie M lungimea maximă a unui șir din mulțimea dată. În cel mai rău caz (două șiruri identice, de lungime maximă), bucla `for` din funcția `commonPrefix` va parcurge M caractere, așadar funcția are complexitatea temporală $O(M)$.

Deoarece până la sfârșitul algoritmului vor fi parcurse toate șirurile (în număr de N), algoritmul propus pentru rezolvarea problemei `Longest Common Prefix` are complexitatea temporală $O(M * N)$.

Algorithm 1 Divide et Impera

```
1:  procedure longestCommonPrefix(array, left, right)
2:      if (left == right) then
3:          return array[left]
4:      if (left < right) then
5:          mid ← (left + right) / 2
6:          str_left ← longestCommonPrefix(array, left, mid)
7:          str_right ← longestCommonPrefix(array, mid + 1, right)
8:          return commonPrefix(str_left, str_right)
9:
10: // Funcție auxiliară ce determină prefixul comun pentru două subșiruri
11: procedure commonPrefix(str_1, str_2)
12:     len_1 ← length(str_1)
13:     len_2 ← length(str_2)
14:     len ← min(len_1, len_2)
15:     prefix ← ""
16:     for (i ← 1 .. len) do
17:         if (str_1[i] ≠ str_2[i]) then
18:             break
19:         prefix ← prefix ++ str_1[i]
20:     return prefix
```

1.5 Eficiență

Algoritmul prezentat mai sus este optim pentru mulțimile nesortate.

În cazul în care, în mulțimea dată, există două șiruri care încep cu caractere diferite, acestea vor avea prefixul comun "" (șirul vid), ceea ce va duce la rezultatul "" pentru întreaga mulțime.

Astfel, dacă mulțimea de șiruri ar fi deja sortată lexicografic, algoritmul poate fi optimizat: am putea găsi prefixul comun doar al primului și ultimului șir, care ar fi și rezultatul final.

1.6 Exemplificare

Fie mulțimea de intrare *array* = {*witcher, witch, wiedzmin, witchcraft*}. Algoritmul Divide et Impera va urma pașii din Figura 1.

Divide: Se "sparge" mulțimea până se ajunge la submulțimile de un singur element (un șir).

Impera: Pe partea stângă, se calculează prefixul comun dintre *witcher* și *witch*, acesta fiind *witch*. Pe partea dreaptă, prefixul comun dintre *wiedzmin* și *witchcraft* este *wi*. Se revine din recursivitatea de la acest nivel, calculându-se acum prefixul comun dintre rezultatele intermediare *witch* și *wi*. Astfel, se găsește rezultatul final *wi*.

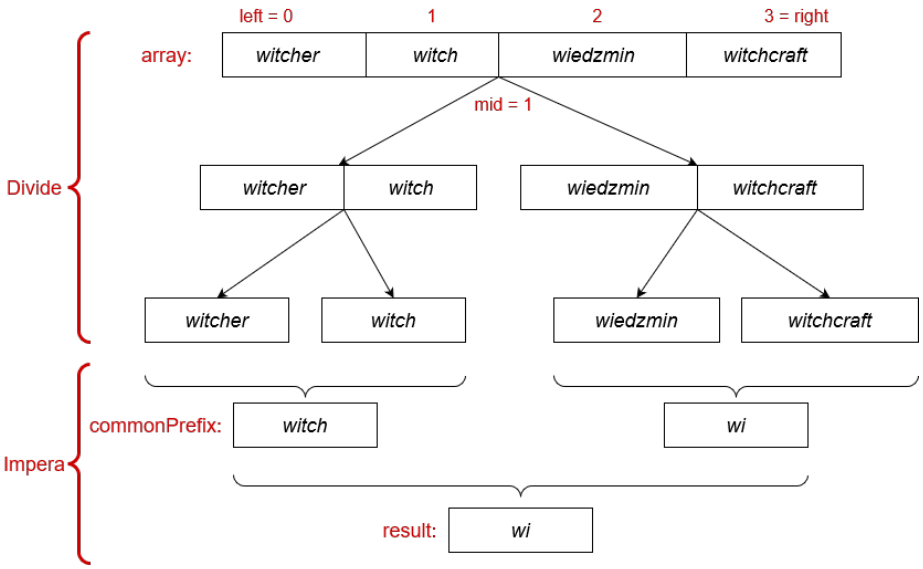

Figure 1: Exemplu - Cel mai lung prefix comun (Divide et Impera)

2 Tehnica Greedy

2.1 Enunțul problemei

Costul minim pentru conectarea orașelor: Se dă un număr N de orașe. Există drumuri între unele orașe, dar toate drumurile sunt stricate. Trebuie să se repare drumurile, astfel încât toate orașele să fie conectate din nou, iar costul reparațiilor să fie minim.

Fiecare drum are un cost dat de matricea *edges*, unde $edges[i][j]$ reprezintă costul reparării drumului dintre orașele i și j. Dacă $edges[i][j] = 0$, înseamnă că nu există drum între orașele respective.

2.2 Descrierea soluției

Considerând un graf neorientat în care fiecare nod reprezintă un oraș și fiecare muchie un drum, problema găsirii costului minim pentru conectarea orașelor se rezumă la găsirea arborelui minim de acoperire pentru acest graf. Rezultatul final se obține prin adunarea costurilor muchiilor acestui arbore.

Pentru găsirea arborelui minim de acoperire, se va folosi algoritmul lui Prim, de tip Greedy. Se începe cu un arbore de acoperire vid, păstrându-se pe parcurs două mulțimi de noduri: una conține nodurile deja incluse în arborele minim de acoperire, iar cealaltă conține nodurile rămase.

La fiecare pas, se consideră toate muchiile care ar uni cele două mulțimi de noduri și se alege muchia de cost minim, incluzând nodul de la capătul extern

în mulțimea de noduri din arborele minim de acoperire. Astfel, la final, toate nodurile vor fi conectate cu muchiile de cost minim.

Pentru alegerea muchiei de cost minim, se atribuie o valoare-cheie fiecărui nod din graful inițial. Toate valorile-cheie sunt inițializate la INF, mai puțin cea a nodului sursă, setată la 0. Cât timp arborele minim de acoperire nu include toate nodurile, se alege un nod v care trebuie inclus, apoi se actualizează valoarea-cheie a fiecărui nod adiacent cu v la minimul dintre valoarea-cheie curentă și costul drumului de la v la nodul respectiv. Practic, valoarea-cheie a fiecărui nod neadăugat încă în arbore reprezintă costul minim pentru a fi inserat.

2.3 Algoritmul de rezolvare

Algorithm 2 Greedy

```
1:  procedure findMinCost(N, edges)
2:      for (i ← 0 .. N) do
3:          keyVal[i] ← INF
4:          isConnected[i] ← false // Nodul nu este adăugat încă
5:          parent[i] ← −1
6:      keyVal[0] ← 0 // Nodul sursă
7:      for (i ← 0 .. N − 1) do
8:          // Găsește nodul de cost minim, dintre nodurile neadăugate încă
9:          u ← minNode(N, keyVal, isConnected)
10:         isConnected[u] ← true
11:         for (v ← 0 .. N) do
12:             if (edges[u][v] && !isConnected[v] && edges[u][v] < keyVal[v])
    then
13:                 keyVal[v] ← edges[u][v]
14:                 parent[v] ← u
15:     // Calculează costul final
16:     cost ← 0
17:     for (i ← 1 .. N) do
18:         cost ← cost + edges[i][parent[i]]
        return cost
19:
20: // Funcție auxiliară ce determină nodul de cost minim, dintre nodurile
    neadăugate încă în arborele de acoperire
21: procedure minNode(N, keyVal, isConnected)
22:     min ← INF
23:     for (i ← 0 .. N) do
24:         if (!isConnected[i] && keyVal[i] < min) then
25:             min ← keyVal[i]
26:             minVal ← i
         return minVal
```

2.4 Complexitate

Avem N noduri în graf. Bucla **for** de la linia 7 va parcurge N pași, adaugând fiecare nod pe rând în arborele minim de acoperire. Înăuntrul său mai există încă o buclă **for**, care actualizează nodurile adiacente, și apelul de funcție `minNode`, ambele de complexitate temporală $O(N)$. Astfel, complexitatea finală a algoritmului este $N * O(N)$, adică $O(N^2)$.

2.5 Corectitudine

Se respectă atât **Proprietatea de alegere de tip Greedy** (alegând soluții optime local, adică muchiile de cost minim, se ajunge la soluția optimă global, adică suma costurilor minime are la rândul său valoarea minimă), cât și **Proprietatea de substructură optimală** (deoarece arborele de minimă acoperire conține soluțiile problemelor locale, adică muchiile de cost minim).

2.6 Exemplificare

Fie harta orașelor din figura de mai jos. Avem N = 5 orașe numerotate de la 0 la 4, iar costul reparării drumurilor a fost reprezentat pe muchii. Arborele minim de acoperire este cel evidențiat cu roz, iar costul final al reparațiilor se obține prin adunarea costurilor muchiilor din acest arbore, adică $1 + 2 + 3 + 4 = 10$.

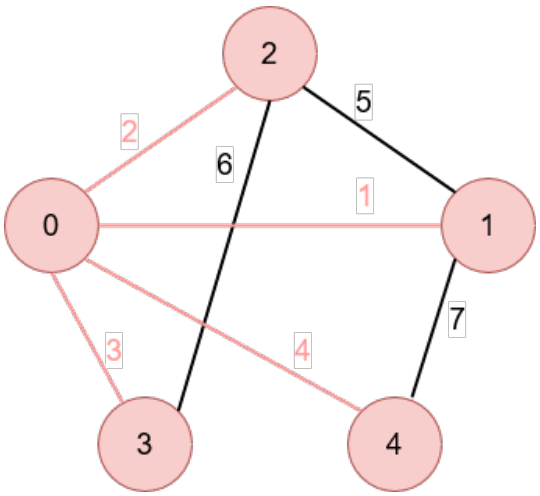

Figure 2: Exemplu - Harta orașelor / Arbore de acoperire minimă (Greedy)

Pentru obținerea acestui arbore cu algoritmul de mai sus, se realizează următorii pași:

Pas 1. Se selectează nodul 0 și se adaugă în arborele minim de acoperire (nodurile adăugate, reprezentate cu verde). Apoi se iterează prin vecinii săi, ac-

tualizându-se valorile-cheie ale acestora (la acest pas, valoarea-cheie este echivalentă cu costul drumurilor de la nodul 0 la vecinul respectiv).

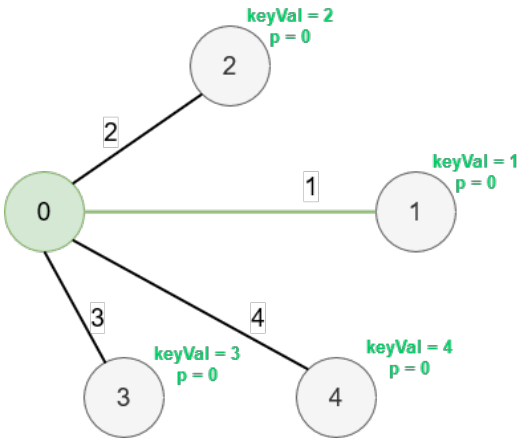

Figure 3: Exemplu - Determinare arbore de acoperire minimă (Pas 1)

Pas 2. Se selectează următorul nod neadăugat încă în arbore, cu valoarea-cheie minimă (adică nodul 1). Deoarece nodul 1 are ca părinte nodul 0, aceasta va fi și muchia de cost minim considerată. Se actualizează vecinii lui 1 (adică 2 și 4), însa valorile minime rămân aceleași, deoarece costul drumurilor de la 1 la 2, respectiv 4, este mai mare decât valorile-cheile ale acestora.

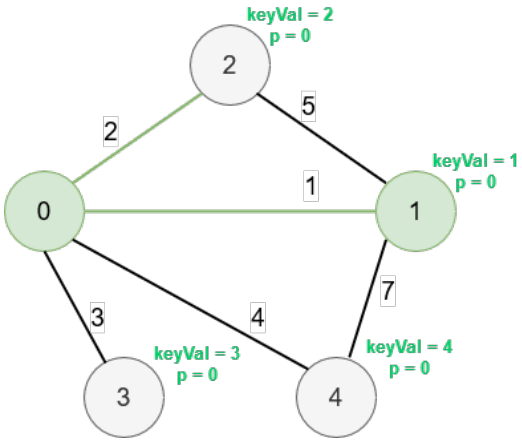

Figure 4: Exemplu - Determinare arbore de acoperire minimă (Pas 2)

Pas 3. Următorul nod cu valoarea-cheie minimă este 2, dar nici acesta nu aduce schimbări vecinilor. Practic, nodul 2 a fost selectat, iar muchia corespunzătoare este (0, 2) deoarece părintele lui 2 a rămas 0 la iterația anterioară.

În continuare, părintele nodurilor 3 și 4 va rămâne 0. La sfârșit se calculează costul minim, cu ajutorul părinților nodurilor din graf.

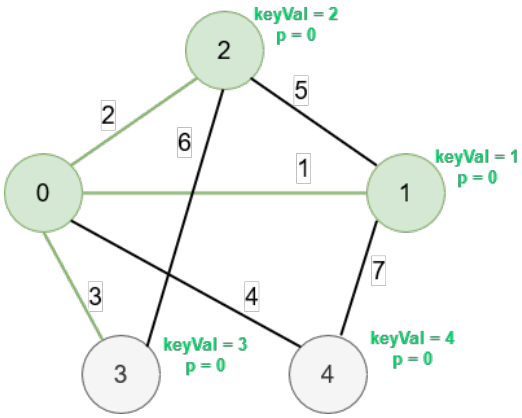

Figure 5: Exemplu - Determinare arbore de acoperire minimă (Pas 3)

3 Tehnica Programării Dinamice

3.1 Enunțul problemei

Aruncări de zaruri: Se dau N zaruri, fiecare zar având M fețe (numerotate de la 1 la M). Se cere numărul metodelor de a obține suma X la aruncarea simultană a celor N zaruri.

3.2 Descrierea soluției

Fie $throws(M,\ N,\ X)$ numărul metodelor de a obține suma X din N zaruri cu câte M fețe. Pentru a calcula această valoare, se consideră toate posibilitățile de a arunca zarurile, folosind soluțiile intermediare (cu $N-1,\ N-2$ etc.) pentru a ajunge la soluția finală.

Spre exemplu, dacă se dorește obținerea sumei $X = 8$, aruncând $N = 3$ zaruri cu câte $M = 6$ fețe, numărul metodelor este $throws(6,\ 3,\ 8)$, care se obține recursiv din adunarea posibilităților de a obține:

- Suma 7, aruncând 2 zaruri, când al treilea zar are valoarea 1
  (adică $throws(6,\ 2,\ 7)$)
- Suma 6, aruncând 2 zaruri, când al treilea zar are valoarea 2
  (adică $throws(6,\ 2,\ 6)$)
- Suma 5, aruncând 2 zaruri, când al treilea zar are valoarea 3
  (adică $throws(6,\ 2,\ 5)$)
- (...)

- Suma 2, aruncând 2 zaruri, când al treilea zar are valoarea 6
  (adică $throws(6,\ 2,\ 2)$)
- Suma 1, aruncând 2 zaruri, când al treilea zar are valoarea 7
  (acest caz e imposibil, deoarece $7 > 6 = M$, așadar $throws(6,\ 2,\ 1) = 0$)

Apoi, se explicitează recursiv fiecare din relațiile de mai sus, până se ajunge la cazul de bază.

Cazul de bază este cel în care aruncăm un singur zar ($N = 1$) și dorim să obținem suma X, unde $X \leq M$. E o singură posibilitate de a face acest lucru, obținând la aruncare chiar fața egală cu suma dorită. Deci $throws(M,\ 1,\ X) = 1,\ \forall\ X \leq M$.

În plus, $throws(M,\ 1,\ X) = 0,\ \forall\ X > M$ (nu se poate obține, din aruncarea unui singur zar cu M fețe, o valoare mai mare decât M).

Se păstrează tabela *throws* de soluții intermediare. Deoarece în recursivitate numărul de fețe M rămâne constant, în pseudocod tabela va avea doar două dimensiuni, $throws[N][X]$.

3.3 Algoritmul de rezolvare

Algorithm 3 Programare Dinamică

```
1:  procedure diceThrows(M, N, X)
2:      for (n ← 1 .. N) do
3:          for (x ← 1 .. X) do
4:              throws[n][x] ← 0 // Inițializarea tabelei
5:      for (x ← 1 .. X) do
6:          if (x <= M) then
7:              throws[1][x] ← 1 // Cazuri de bază
8:      for (n ← 2 .. N) do
9:          for (x ← 1 .. X) do
10:             for (k ← 1 .. M) do
11:                 if (k < x) then
12:                     throws[n][x] ← throws[n][x] + throws[n − 1][x − k]
        return throws[N][X] // Rezultatul final
```

3.4 Complexitate

Avem N zaruri cu câte M fețe și dorim să obținem suma X. Atunci, din cele 3 bucle **for** imbricate la linia 8, rezultă complexitatea temporală $O(M * N * X)$.

Complexitatea spațială ține de memorarea tabelei de soluții intermediare *throws*, tabela având $N + 1$ linii și $X + 1$ coloane (linia și coloana 0 nu sunt folosite, deoarece numerotarea fețelor începe de la 1). Așadar, complexitatea spațială este $O(N * X)$.

3.5 Relația de recurență

Relația de recurență generală este: $throws(M,\ N,\ X) = throws(M,\ N-1,\ X-1) + throws(M,\ N-1,\ X-2) + throws(M,\ N-1,\ X-3) + ... + throws(M,\ N-1,\ X-M)$

Această relație se determină din faptul că, pentru a obține suma X din N zaruri, nu avem decât să adunăm toate posibilitățile de a obține suma $Y < X$ din $N-1$ zaruri, contând pe faptul că al N-lea zar va cădea pe fața corespunzătoare valorii rămase până la suma dorită, adică $X-Y$. (Vezi exemplul de la paragraful 3.2. Descrierea soluției)

3.6 Exemplificare

Să presupunem că avem $N = 3$ zaruri cu câte $M = 4$ fețe. Vrem să obținem suma $X = 10$. Practic, ne interesează $throws(4,\ 3,\ 10)$.

Pentru exemplificare, am realizat diagrama de mai jos, în care se poate observa ce valori din tabelă folosește fiecare valoare intermediară. O căsuță reprezintă o intrare din tabel, de forma $throws(M,\ N,\ X)$. În dreapta ei, am notat cu gri valoarea sa. Rezultă 6 soluții posibile.

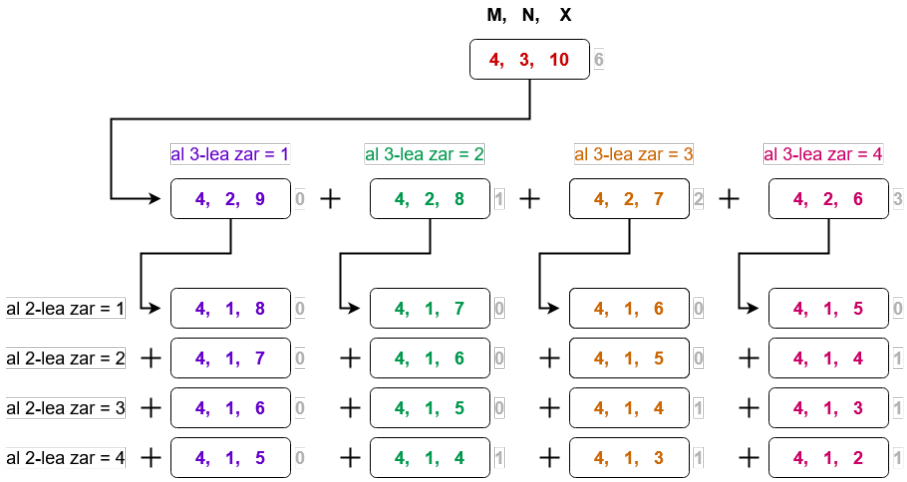

Figure 6: Exemplu - Aruncări de zaruri (Programare Dinamică)

Spre exemplu, atunci când al 3-lea zar aruncat are valoarea 1 (prima coloană), numărul de soluții este $throws(4,\ 2,\ 9)$. Pentru calculul acestei valori, se adună celelalte 4 valori din aceeași coloană. Se observă că valoarea finală este 0, deoarece nu se poate obține suma 9 aruncând 2 zaruri de valoare maxim 4.

Atunci când al 3-lea zar aruncat are valoarea 3 (penultima coloană), numărul de soluții este $throws(4,\ 2,\ 7)$ - adică trebuie obținută suma 10 - 3 = 7 din cele 2 zaruri rămase. Aceasta se poate obține doar în 2 moduri, atunci când primul zar a fost 4, iar al doilea 3, și vice-versa (primul 3, al doilea 4).

4 Tehnica Backtracking

4.1 Enunțul problemei

Șoarecele din labirint: Se dă o matrice pătrată de dimensiune N, $maze$. Un șoarece pleacă din colțul din stânga sus (căsuța $maze[0][0]$) și trebuie să ajungă la colțul din dreapta jos (căsuța $maze[N-1][N-1]$).

În matricea labirint, valoarea 0 a unei căsuțe înseamnă că acolo se află un perete (șoarecele nu se poate afla pe acea poziție), iar orice valoare $x \neq 0$ înseamnă că șoarecele poate face maxim x pași pe verticală sau pe orizontală, pornind din căsuța respectivă.

Se cere găsirea unei căi prin care șoarecele să ajungă la destinație (în cazul în care există soluție, va avea forma unei matrice în care $sol[i][j] = 1$, dacă șoarecele a trecut prin căsuța respectivă, 0 altfel).

4.2 Descrierea soluției

Cât timp șoarecele nu a ajuns la destinație, se parcurg următorii pași:

1. Se marchează căsuța curentă ca posibilă soluție, $sol[i][j] = 1$.
2. Se încearcă mutarea cu maxim x pași pe direcție orizontală, verificând recursiv dacă această mutare a dus la o soluție finală.
3. În cazul în care mutarea de mai sus nu a condus la o soluție, se procedează la fel pe direcția verticală.
4. Dacă nicio mutare de până acum nu a condus la o soluție, se marchează căsuța curentă cu 0 și se întoarce `false` (pentru verificarea din recursivitate).

Practic, se va avansa în labirint pe orizontală, apoi verticală, cât timp poziția următoare este validă și nu s-a ajuns încă la o soluție.

4.3 Algoritmul de rezolvare

Vezi Algoritmul 4.

4.4 Complexitatea

Funcția de backtracking conține apeluri recursive. Bucla **for** de la linia 18 poate executa maxim $N-1$ pași. Pentru fiecare pas, se fac două apeluri recursive. Rezultă $2^{n*(n-1)}$ apeluri în cel mai rău caz, de unde complexitatea temporală este $O(2^{n^2})$.

4.5 Eficiență

Numărul excesiv de apeluri recursive face ca algoritmul să fie ineficient temporal pentru matrici de dimensiuni mari.

O soluție mai optimă ar fi tratarea matricei-labirint ca pe un graf în care două noduri sunt adiacente dacă se poate ajunge de la o căsuță corespunzătoare la cealaltă. La parcurgerea în adâncime a grafului, se marchează nodurile vizitate, iar la sfârșit se verifică dacă nodul corespunzător destinației a fost vizitat.

Algorithm 4 Backtracking

```
1:  procedure solveMaze(maze[N][N])
2:      for (i ← 0 .. N − 1) do
3:          for (j ← 0 .. N − 1) do
4:              sol[i][j] ← 0 // Inițializarea soluției
5:      if (!(backtrackMaze(maze, 0, 0, sol)) then
6:          // nu există soluție
7:      // afișează sol
8:
9:  // Funcție auxiliară recursivă, folosită în procesul de backtracking
10: procedure backtrackMaze(maze, x, y, sol)
11:     // Verifică dacă s-a ajuns la destinație
12:     if (x == N − 1 && y == N − 1) then
13:         sol[x][y] ← 1
14:         return true
15:     // Verifică dacă x, y < N și maze[x][y] ≠ 0
16:     if (isValidPosition(maze, x, y) then
17:         sol[x][y] ← 1
18:         for (i ← 1 .. maze[x][y], i < N) do
19:             // Caută soluție prin avansare pe orizontală
20:             if (backtrackMaze(maze, x + i, y, sol) then
21:                 return true
22:             // Altfel, caută soluție prin avansare pe verticală
23:             if (backtrackMaze(maze, x, y + i, sol) then
24:                 return true
25:         // Nu s-a găsit nicio soluție pornind din această căsuță
26:         sol[x][y] ← 0
27:         return false
28:     // Poziție nevalidă
29:     return false
```

4.6 Exemplificare

Fie matricea-labirint *maze* de mai jos, cu soluția aferentă *sol*.

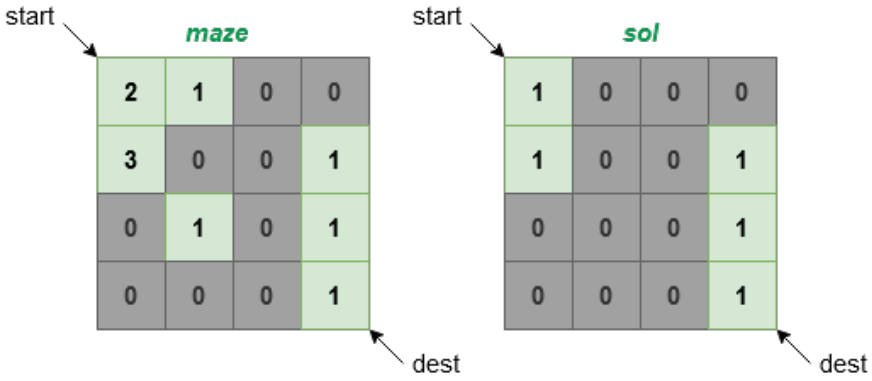

Figure 7: Exemplu - Șoarece în labirint (Backtracking)

Conform algoritmului, șoarecele va începe din căsuța $maze[0][0]$, pe care o va marca în soluție, $sol[0][0] = 1$. De aici, poate înainta cu maxim 2 pași pe orizontală sau pe diagonală.

Pe orizontală, dacă ar înainta cu 2 pași, ar ajunge pe o căsuță nevalidă (cu valoarea 0). Înaintează cu 1 pas. Acum se află pe $maze[0][1]$, cu valoarea 1, și marchează $sol[0][1] = 1$. De aici poate face 1 pas în dreapta sau în jos, dar ambele conduc la o poziție nevalidă. Marchează $sol[0][1] = 0$ și se întoarce la căsuța $maze[0][0]$.

Pe verticală, poate înainta doar cu 1 pas (2 pași ar conduce la poziție nevalidă). Ajunge pe $maze[1][0]$, cu valoarea 3, și marchează $sol[1][0] = 1$. Se poate deplasa la dreapta cu 3 pași.

Ajuns pe poziția $maze[1][3]$, cu valoarea 1, marchează $sol[1][3] = 1$ și încearcă întâi deplasarea la dreapta, care duce la o poziție nevalidă (iese din matrice). Atunci, încearcă deplasare în jos, ceea ce duce la poziția validă $maze[2][3]$, cu $sol[2][3] = 1$.

De aici, încă 1 pas în jos duce la soluția finală, marcând $sol[3][3] = 1$. Algoritmul iese din recursivitate și afișează soluția finală.

5 Analiză comparativă

5.1 Aplicare, avantaje și dezavantaje

Fiecare tehnică din cele exemplificate mai sus poate fi aplicată pe un anumit tip de problemă.

1. În cazul tehnicii **Divide et Impera**, am observat că ea se poate aplica doar atunci când problema poate fi divizată în subprobleme, ale căror soluții

pot fi rezolvate independent și apoi combinate pentru a găsi soluția probelemei inițiale. Nu rezolvă probleme de optimizare.

2. Tehnica **Greedy** este destinată problemelor de optim. La fiecare pas, se face alegerea optimă în momentul curent, sperând ca la sfârșit să obținem soluția optimă global. Acest lucru nu este garantat, motiv pentru care problema trebuie să respecte anumite condiții specifice Greedy.

Tehnica **Greedy** (de obicei iterativă) este mai eficientă decât **Divide et Impera** (deseori recursivă), deoarece **Greedy** nu consideră soluțiile precedente, pe când subproblemele din **Divide et Impera** sunt independente, ceea ce poate duce la rezolvarea unei subprobleme de mai multe ori.

3. **Programarea Dinamică**, precum **Greedy**, se folosește tot pentru rezolvarea problemelor de optim, însă diferența vine din faptul că, la fiecare pas, soluția curentă este un rezultat al tuturor soluțiilor optime precedente, astfel încât se garantează obținerea soluției optime globale la sfârșit.

Astfel, față de **Greedy**, care doar ia decizia optimă în momentul curent, **Programarea Dinamică** se bazează pe o relație de recurență care folosește rezultatele anterioare. Acest lucru face ca **Programarea Dinamică** să necesite memorie fizică suplimentară, pentru reținerea tabelei de soluții parțiale.

4. Tehnica **Backtracking** este un mod de rezolvare specific problemelor în care se pot testa forțat toate combinațiile posibile, construite incremental, până la găsirea unei soluții valide (nu neapărat optimă).

Deoarece în **Backtracking** sunt explorate foarte multe soluții care nu duc la rezultatul final, această tehnică este mai ineficientă decât **Programarea Dinamică**, ce consideră doar soluțiile intermediare optime.

5.2 Exemplu - Divide et Impera vs Greedy

Pentru compararea celor două tehnici de programare, am ales problema găsirii **Subsecvenței de sumă maximă (Maximum Subarray Sum)**. Aceasta poate fi rezolvată prin tehnicile Divide et Impera, Programare Dinamică și Greedy.

5.2.1 Maximum Subarray Sum - Divide et Impera

Ideea este să se împartă vectorul inițial în două părți egale, calculându-se maximul dintre:

- Suma maximă obținută din subsecvența stângă (apel recursiv)
- Suma maximă obținută din subsecvența dreaptă (apel recursiv)
- Suma maximă obținută dintr-o subsecvență care conține atât elemente din partea stângă, cât și elemente din partea dreaptă

Primele două sunt simple apeluri recursive ale funcției, în timp ce a treia sumă se poate obține în timp linear, în felul următor: se calculează maximul dintre: [maximul posibil adunând doar elemente din stânga], [maximul posibil adunând doar elemente din dreapta] și suma celor două.

Astfel, se obține relația de recurență pentru complexitate:

$T(N) = 2 * T(N/2) + \theta(N)$

Așadar, complexitatea temporală a acestei soluții este $\theta(N * logN)$.

5.2.2 Maximum Subarray Sum - Greedy / Programare Dinamică (Algoritmul lui Kadane)

Ideea din spatele Algoritmului lui Kadane este iterarea prin vector, ținând cont de suma curentă (*currentSum*) și suma maximă obținută până acum (*maxSoFar*). Se aplică pentru vectori care au cel puțin un element pozitiv.

1. Se inițialiează $currentSum = 0$ și $maxSoFar = -INF$.

2. Se adună elementul curent la *currentSum*, actualizând și *maxSoFar* dacă aceasta din urmă este mai mică.

3. Se verifică dacă suma curentă a devenit negativă, caz în care suma curentă se actualizează la 0 și se ignoră subsecvența de până acum, deoarece ea nu poate în niciun fel să contribuie la valoarea maximă finală.

4. Se continuă iterarea până la sfârșitul vectorului, când se returnează valoarea *maxSoFar*.

Se pot reține indecșii de început și sfârșit al secvenței de sumă maximă, prin actualizare la pasul 2, dacă s-a găsit un nou maxim, și resetarea celui de început la pasul 3.

Deoarece iterează o singură dată prin vector, algoritmul are complexitatea temporală $O(N)$.

Observație: Acest algoritm poate fi interpretat ca algoritm de Programare dinamică, dar și Greedy, din următoarele motive:

- **Greedy**, deoarece soluția parțială (suma curentă) este îmbunătățită repetat, pe baza criteriilor de optim local și substructură optimă (date de faptul că suma se resetează la 0 când devine negativă, pentru că ar fi mai rău să continuăm cu o sumă negativă decât să resetăm subsecvența considerată), până la obținerea soluției finale.

- **Programare dinamică**, deoarece pentru fiecare element avem două opțiuni posibile: să îl adăugăm la suma curentă, sau să continuăm mai departe cu suma resetată.

5.2.3 Concluzie

Se observă că, pentru această problemă, tehnica **Greedy** - iterativă, de complexitate temporală $\theta(N)$, este mai eficientă decât **Divide et Impera** - recursivă, cu $\theta(N * logN)$. De asemenea, soluția **Greedy** este mai ușor de înțeles, având o abordare directă a problemei.

Pentru ambele soluții, trebuie demonstrat faptul că ele conduc la soluția optimă. În observația făcută mai sus, am arătat deja că cea de-a doua soluție respectă criteriile **Greedy** (și poate fi asociată și cu **Programare dinamică**), așadar este garantată soluția optimă global.

6 Referințe

- **(Divide et Impera) Găsirea celui mai lung prefix comun** - `https://www.geeksforgeeks.org/longest-common-prefix-using-divide-and-conquer-algorithm/`
- **(Greedy) Costul minim pentru conectarea orașelor** - `https://www.geeksforgeeks.org/minimum-cost-connect-cities/`
- **(Programare dinamică) Aruncări de zaruri** - `https://www.geeksforgeeks.org/dice-throw-dp-30/`
- **(Backtracking) Șoarecele din labirint** - `https://www.geeksforgeeks.org/rat-in-a-maze-with-multiple-steps-jump-allowed/`
- **Maximum Sum Subarray (Kadane)** - `https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/`