Proiectarea Algoritmilor - Tema 1

Alexandru Licuriceanu
alicuriceanu@stud.acs.upb.ro

Universitatea POLITEHNICA din București
Facultatea de Automatică și Calculatoare

Abstract. Această temă presupune prezentarea unor propleme care se pot rezolva folosind câteva tehnici de programare: Divide et Impera, Metoda Greedy, Backtracking, Programare Dinamică.

Keywords: Divide et Impera, Backtracking, Greedy, Programare Dinamică.

1 Divide et Impera – „Odd occurrence number in an array” [1]

1.1 Enunțul problemei

Se dă un număr N și un vector care conține N elemente. Vectorul nu trebuie să fie obligatoriu sortat. Elementele vectorului sunt egale două câte două și nu pot exista mai mult de două elemente consecutive care să fie și egale. În acest vector se află și un element care nu are pereche și se dorește ca acesta să fie găsit folosind un algoritm eficient.

Un exemplu corect de input este:

[2, 2, 3, 3, 2, 2, 4, 4, 3, 1, 1] – elementul căutat este 3.

Exemple greșite de input sunt:

[1, 2, 1] – nu are perechi de elemente identice.
[1, 1, 2, 2, 2, 3, 3] – conține trei elemente identice consecutive.

1.2 Abordare

Soluția naivă este să sortezi vectorul și apoi să numeri aparițiile fiecărui element, returnând pe cel care apare de un număr impar de ori. Metoda aceasta este lentă și are complexitatea temporală $O(N\ logN)$ unde N este numărul de elemente din vector.

O soluție mai bună poate fi obținută prin utilizarea unui algoritm de căutare binară ușor modificat. Drept exemplu avem următorul vector și indicii elementelor:

Index	0 1 2 3 4 5 6 7 8
Vector	[2, 2, 3, 3, 2, 2, 4, 3, 3]

Așadar, se poate observa că fiecare pereche înainte de numărul căutat are primul membru la un indice par și al doilea membru la un indice impar. Perechile de după acest număr vor avea primul membru la un indice impar și al doilea membru la un indice par.

Folosind observația asta, se poate determina în ce parte a indicelui de mijloc se află elementul căutat. De aici reies două cazuri:

1. Dacă indicele elementului din mijloc este par și elementul din dreapta lui este identic, numărul căutat se află undeva în dreapta mijlocului, altfel, se află în stânga acestuia.
2. Dacă indicele elementului din mijloc este impar și elementul din stânga lui este identic, numărul căutat se afla undeva în dreapta mijlocului, altfel, se află în stânga acestuia.

1.3 Implementare, rulare și performanță

Algorithm 1 find (array, left, right)

```
if left = right then
    return array[left]
endif

mid ← (left + right) / 2

if mid % 2 = 1 then
    if array[mid] = array[mid-1] then
        return find(array, mid+1, right)
    else
        return find(array, left, mid-1)
    endif
else
    if array[mid] = array[mid+1] then
        return find(array, mid+2, right)
    else
        return find(array, left, mid)
    endif
endif
```

Algoritmul funcționează similar cu o căutare binară.
Pentru vectorul [2, 2, 3, 3, 2, 2, 4, 3, 3] pașii care se execută sunt:

Vector [2, 2, 3, 3, 2, 2, 4, 3, 3]	left = 0	mid = 4	Vector[mid] = Vector[mid+1] →
	right = 8	Vector[mid] = 2	se apelează find(Vector, mid+2, right)

Vector [2, 2, 3, 3, 2, 2, 4, 3, 3]	left = 6	mid = 7	Vector[mid] ≠ Vector[mid-1] →
	right = 8	Vector[mid] = 3	se apelează find(Vector, left, mid-1)

Vector [2, 2, 3, 3, 2, 2, 4, 3, 3]	left = 6	mid = 6	left = right →
	right = 6	Vector[mid] = 4	se returnează Vector[left]

Performanță:

- Complexitate temporală: $O(logN)$ unde N este numărul elementelor.
- Complexitate spațială: $O(1)$ spațiu auxiliar.

2 Metoda Greedy – „Rearrange string for distance D” [2]

2.1 Enunțul problemei

Se dă un șir de caractere și un număr D. Cerința este ca șirul să fie rearanjat astfel încât distanța dintre oricare două caractere egale să fie egală cu D. Caractere se pot repeta sau nu în șirul primit la input. De asemenea, pot exista mai multe modalități corecte de a rearanja șirul de caractere, iar dacă aranjarea nu este posibilă, trebuie trimisă o eroare la output.

Un exemplu de input care poate fi aranjat este:

Input: „aacbbc” D = 3
Output: „abcabc”

Input: „terracotta” D = 3
Output: „tartartceo”

Un exemplu de input care nu poate fi aranjat este:

Input: „vvvvv” D = 4
Output: Impossibil

Input: „yyygg” D = 3
Output: Impossibil

2.2 Abordare

O soluție posibilă este să numeri mai întâi de câte ori apare fiecare caracter distinct, apoi să îl iei pe cel care are cea mai mare frecvență și să îi plasezi duplicatele cât mai aproape unele de celelalte. Pentru procesarea caracterelor rămase, se repetă acest proces.

O implementare eficientă pentru memorarea frecvențelor fiecărui caracter poate fi utilizarea unei cozi cu priorități sortată după frecvența elementelor. După o traversare a șirului, caracterul cu cea mai mare frecvență se va afla pe prima poziție în coadă.

2.3 Implementare, rulare și performanță

Algorithm 2 rearrange (string, distance)

```
length ← string.length
for each c in string do
    frequency[c] ← frequency[c] + 1
end for
Fill string with NULL
pq ← pq.create(frequency)
while pq is not empty do
    x ← pq.peek
    pos ← First available position in string
    Put character x at positions: pos, pos+distance, ..., pos+(x.freq-1) * distance
    If one position > length, rearranging is impossible
    pq.dequeue
end while
```

Pentru un input de tipul: „terracotta” și D = 3:

[t e r r a c o t t a]	pq	t a r c e o	x = 0	frequency[x] = 0
	frequency	3 2 2 1 1 1	pos = 0	
[t _ _ t _ _ t _ _ _]	pq	a r c e o	x = t	frequency[x] = 3
	frequency	2 2 1 1 1	pos = 0, 3, 6	
[t a _ t a _ t _ _ _]	pq	r c e o	x = a	frequency[x] = 2
	frequency	2 1 1 1	pos = 1, 4	
[t a r t a r t _ _ _]	pq	c e o	x = t	frequency[x] = 2
	frequency	1 1 1	pos = 2, 5	
[t a r t a r t c _ _]	pq	e o	x =c	frequency[x] = 1
	frequency	1 1	pos = 7	
[t a r t a r t c e _]	pq	o	x = e	frequency[x] = 1
	frequency	1	pos = 8	
[t a r t a r t c e o]	pq	empty	x = o	frequency[x] = 1
	frequency		pos = 9	

Performanță:

- Complexitate temporală: $O(N + M\ logH)$ unde N este lungimea șirului, M este numărul de caractere diferite din șir, iar H este numărul maxim de caractere distincte.

- Complexitate spațială: $O(N)$ spațiu auxiliar.

3 Programare dinamică – „Maximum height of tree for each node considered as root” [3]

3.1 Enunțul problemei

Se dă un arbore cu N noduri și N-1 muchii, să se afle înălțimea maximă a arborelui pentru fiecare nod luat ca rădăcină.

3.2 Abordare

Soluția naivă a algoritmului este să aplicăm algoritmul DFS („*depth-first search*”) pe fiecare nod, salvând înălțimea maximă obținută. Această metodă este lentă și are complexitatea temporală $O(N^2)$ unde N este numărul de noduri din arbore.

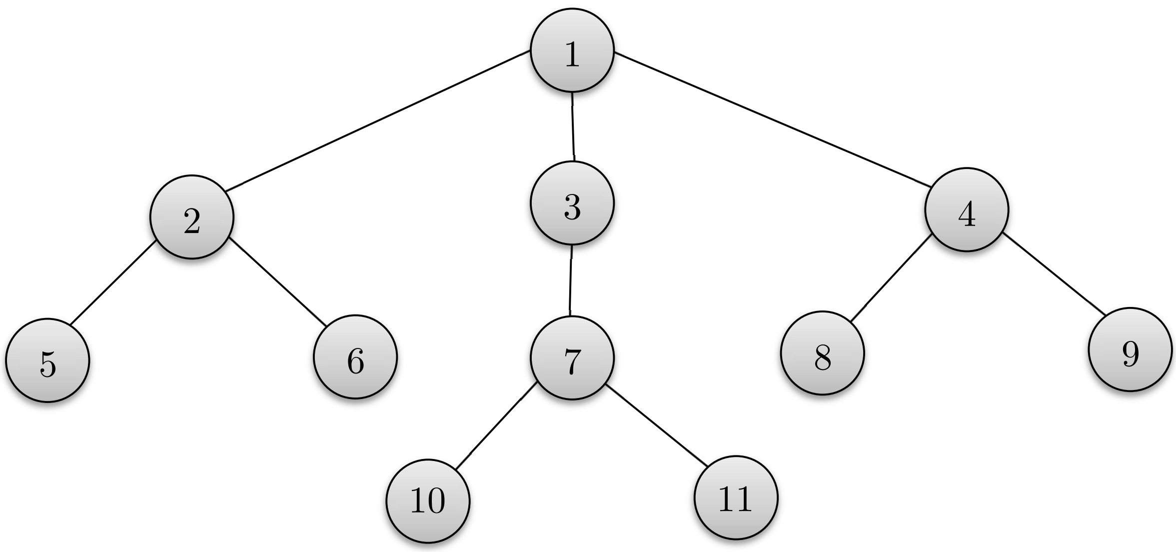
Metoda mai eficientă presupune parcurgerea DFS a arborelui și calcularea a două valori pentru fiecare nod:

- Distanța maximă până la o frunză mergând în sus pe arbore, prin părintele nodului. Această valoare va fi notată cu OUT.
- Distanța maximă până la o frunză mergand în jos pe arbore, prin copiii nodului. Această valoarea va fi notată cu *IN*.

Așadar, înălțimea maximă a arborelui pentru nodul x ales ca rădăcină va fi maximul dintre *IN[x]* și OUT*[x]*.

3.3 Implementare, rulare și performanță

Exemplu. Avem următorul arbore, cu nodul 1 ales ca rădăcină:



Pentru a calcula valorile lui *IN*:

- Începem cu un nod x, în cazul de față nodul x este nodul 1, dar oricare dintre noduri poate fi ales.
- Traversăm copiii nodului x, apelând inDFS pentru fiecare dintre aceștia. Dacă se ajunge pe un nod care este frunză, *IN[frunză]* = 0.
- După apel, calculăm recursiv *IN[x]* astfel:

$$IN[x] = \max(\ IN[x],\ 1 + IN[copil]\ )$$

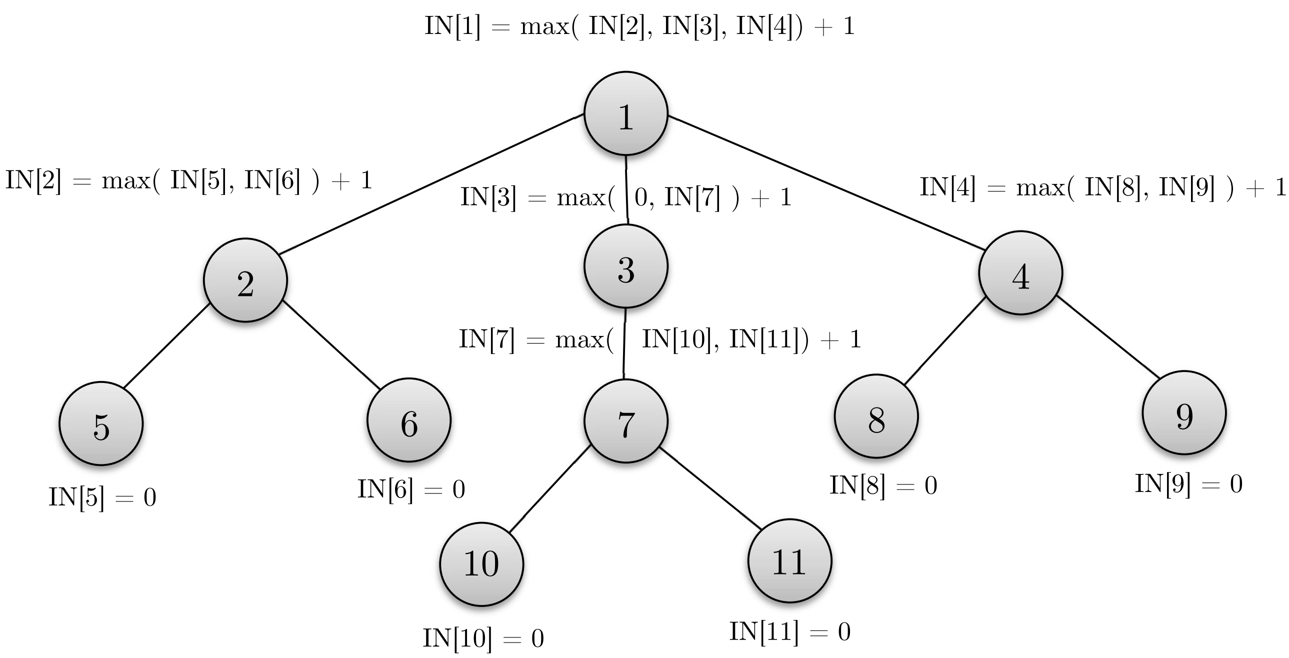
În pseudocod, algoritmul ar arăta astfel:

Algorithm 3 inDFS (tree, x, parent)

```
IN[x] ← 0

for each child in tree[x] do
    inDFS(tree, child, x)
    IN[x] ← max(IN[x], 1 + IN[child]
end for
```

După execuția algoritmului, vor fi calculate următoarele valori pentru IN:



Pentru a calcula valorile lui OUT:

- Inițial, trebuie calculate prima și a doua cea mai lungă cale de la fiecare copil direct al nodului x la o frunză. Trebuie doar căutat in vectorul *IN*, care a fost calculat înainte. Să numim lungimile acestor căi *firstLongest*, *secondLongest*.
- Parcurgem copiii nodului x. Variabila *longest* ia următoarele valori: Dacă drumul *firstLongest* trece prin copilul din iterația curentă, atunci *longest* = *secondLongest*. Altfel, *longest* = *firstLongest*.
- OUT se calculează astfel:

$$OUT[copil] = 1 + \max(\ OUT[x],\ 1 + longest\ )$$

- Apelează outDFS pentru copilul de la iterația curentă.
- În final, înălțimea maximă dacă fiecare nod x ar fi luat ca rădăcină este maximul dintre *IN[x]* și *OUT[x]*.

Pseudocod pentru calcularea valorilor OUT:

Algorithm 4 outDFS (tree, x, parent)

```
firstLongest ← -1
secondLongest ← -1

for each child in tree[x] do
    if IN[child] >= firstLongest then
        secondLongest ← firstLongest
        firstLongest ← IN[child]
    else if IN[child] > secondLongest then
        secondLongest ← IN[child]
    end if
end for

for each child in tree[x] do
    longest ← firstLongest

    if firstLongest = IN[child] then
        longest ← secondLongest
    end if

    OUT[child] ← 1 + max(OUT[x], 1 + longest)
    outDFS(tree, child, x)
end for
```

Pseudocod pentru calcularea rezultatului final:

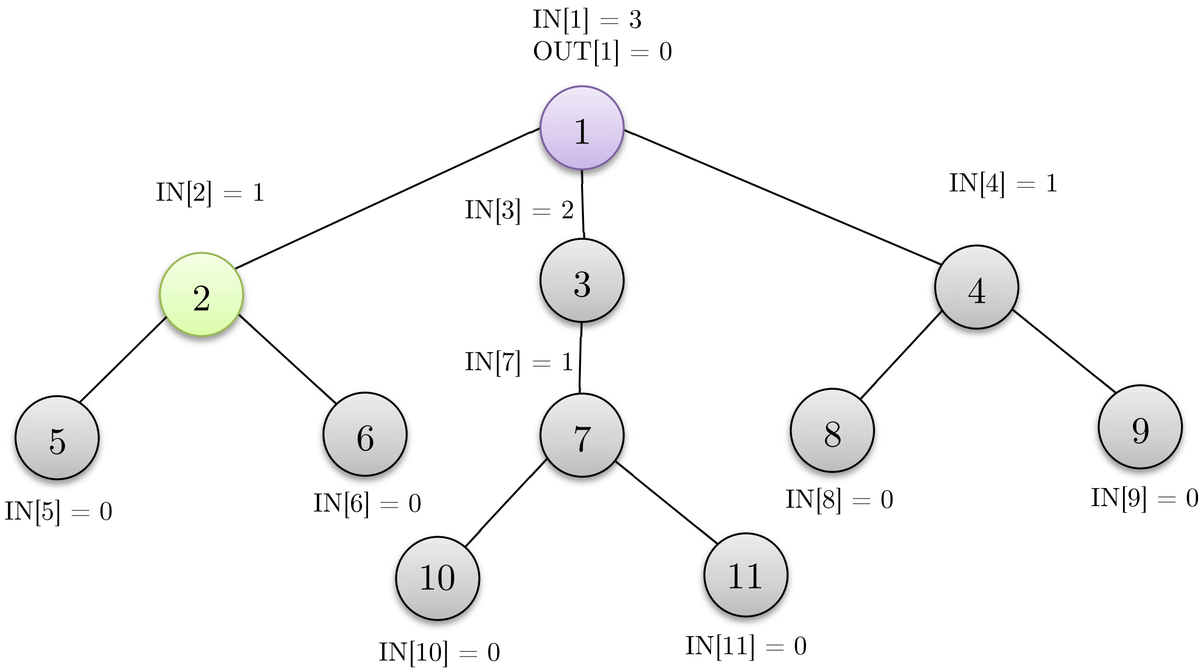
Algorithm 5 heights (tree)

```
inDFS(tree, 1, 0)
outDFS(tree, 1, 0)

result ← {}
for each node in tree do
    result[node] ← max(IN[node], OUT[node])
end for
```

Un exemplu pentru a demonstra cum funcționează algoritmul outDFS:

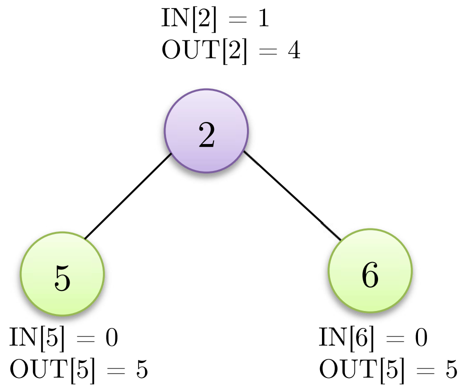
Nodul marcat cu mov reprezintă rădăcina arborelui la apelul respectiv. Nodurile marcate cu verde le reprezintă pe cele procesate la nivelul apelului recursiv și la momentul descris în explicație.



(Apel inițial) Explicație:

outDFS(tree[1], 1, 0) → children = 2, 3, 4
firstLongest = IN[3] = 2
secondLongest = IN[4] = 1

Iterează toți copii lui tree[1]: child = 2
longest = firstLongest = 2
$\boxed{\text{OUT[2]}}$ = 1 + max(OUT[1], 1 + longest) = 4
Apelează outDFS(tree[2], 2, 1)

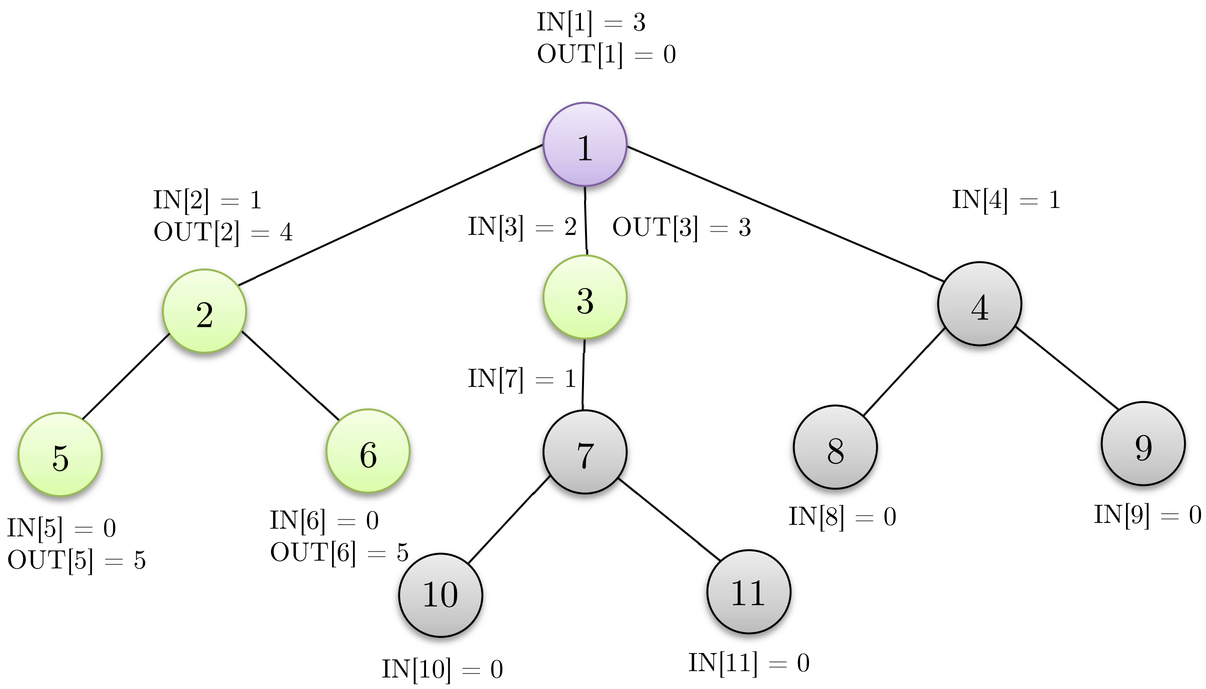


Explicație:

outDFS(tree[2], 2, 1) $\rightarrow$ children = 5, 6
firstLongest = IN[5] = 0
secondLongest = IN[6] = 0

Iterează toți copii lui tree[2]: child = 5
longest = secondLongest = 0
$\boxed{\text{OUT[5]}}$ = 1 + max(OUT[2], 1 + longest) = 5
Apelează outDFS(tree[5], 5, 2) $\rightarrow$ se închide.

child = 6
longest = firstLongest = 0
$\boxed{\text{OUT[6]}}$ = 1 + max(OUT[2], 1 + longest) = 5
Apelează outDFS(tree[6], 6, 2) $\rightarrow$ se închide.



(Apel inițial) Explicație:

outDFS(tree[1], 1, 0) → children = 2, 3, 4
firstLongest = IN[3] = 2
secondLongest = IN[4] = 1

Iterează toți copii lui tree[1]: child = 2
longest = firstLongest = 2
OUT[2] = 1 + max(OUT[1], 1 + longest) = 4
Apelează outDFS(tree[2], 2, 1) → s-a închis.

child = 3
longest = secondLongest = 1
OUT[3] = 1 + max(OUT[1], 1 + longest) = 3
Apelează outDFS(tree[2], 2, 1) → ...

Procesul este același și se repetă și pentru restul nodurilor.

Performanță:

- Complexitate temporală: $O(N) + O(N) + O(N) + O(N) \approx O(N)$ unde N este numărul de noduri din arbore. $O(N)$ pentru inDFS, $O(N) + O(N)$ pentru outDFS și $O(N)$ pentru procesarea rezultatului.

- Complexitate spațială: $O(N)$ spațiu auxiliar.

4 Backtracking – „Word break" [4]

4.1 Enunțul problemei

Se dă un dicționar format din cuvinte și un șir de caractere format din cuvinte fără spații între ele. Se cere să se găsească toate modalitățile în care se poate împărți șirul în cuvinte din dicționarul dat.

Exemplu:

Dicționar = [„i", „like", „sam", „sung", „samsung"]
Șir = „ilikesamsung"

Rezultat = „i like sam sung",
„i like samsung".

4.2 Abordare

Pentru a găsi toate posibilitățile în care se poate împărți sirul:

- Plecăm de la șirul inițial, de la poziția x = 0 (primul caracter).
- Luăm subșirul de la poziția x la poziția y, care este inițial tot 0.
- Dacă subșirul de la x la y este în dicționar, îl adăugăm într-un acumulator și apelăm recursiv funcția pentru restul șirului, y devine x, x devine poziția finală din șir. Când se termină apelurile recursive, se incrementează y, pornind din nou procesul pentru subșirul de la pozițiile x = 0, y = 1 și tot așa până la finalul șirului inițial.
- Dacă subșirul de la x la y nu este în dicționar, incrementează y.

4.3 Implementare, rulare și performanță

O implementare în pseudocod este:

Algorithm 6 wordBreak (dictionary, string, length, accumulator)

```
for i = 0 ... length do
    prefix ← substr(string, 0, i)

    if dictionary.contains(prefix) then
        if i = length then
            accumulator ← accumulator + prefix
            return accumulator
        end if

        wordBreak(substr(string, i, length − i), length − i, result + prefix
        + ” ”)
    end if
end for
```

De exemplu, pentru un input de tipul:

dictionary = [„i”, „like”, „ice”, „cream”, „icecream”]
string = „ilikeicecream”

Algoritmul va funcționa astfel:

(Apel inițial, acumulatorul este ’””)

wordBreak(„ilikeicecream”, 13, ’””)
itereză fiecare caracter din string:

string = „ilikeicecream”
prefix = substr(0, i) = substr(0, 0) = „i”
dictionary.contains(„i”) = true, dar nu am ajuns la capătul șirului

Apelează
wordBreak(„likeicecream”, 12, „i ”)

(Apelul curent al lui wordBreak încă nu s-a închis) wordBreak(„likeicecream”, 12, „i ”):

iterează fiecare caracter din string:

string = „likeicecream”

prefix = substr(0, i) = substr(0, 0) = „l”
dictionary.contains(„l”) = false, incrementează i.

prefix = substr(0, i) = substr(0, 1) = „li”
dictionary.contains(„li”) = false, incrementează i.

prefix = substr(0, i) = substr(0, 2) = „lik”
dictionary.contains(„lik”) = false, incrementează i.

prefix = substr(0, i) = substr(0, 3) = „like”
dictionary.contains(„like”) = true, dar nu am ajuns la capătul șirului.

Apelează
wordBreak(„icecream”, 8, „i like ”)

(Apelul curent al lui wordBreak încă nu s-a închis)

wordBreak(„icecream”, 8, „i like ”):

iterează fiecare caracter din string:

string = „icecream”

prefix = substr(0, i) = substr(0, 0) = „i”
dictionary.contains(„i”) = true, se va porni alt lanț de apeluri recursive, dar se vor închide pentru că în subșirul rămas vor exista cuvinte care nu se află în dicționar, deci să împarți șirul la acest „i” nu reprezintă o soluție.

prefix = substr(0, i) = substr(0, 1) = „ic”
dictionary.contains(„ic”) = false, incrementează i.

prefix = substr(0, i) = substr(0, 2) = „ice”
dictionary.contains(„ice”) = true, dar nu am ajuns la capătul șirului.

Apelează
wordBreak(„cream”, 5, „i like ice”), care va întoarce primul rezultat,
„i like ice cream”. Se reia execuția lui wordBreak(„icecream”, 8, „i like ”):

prefix = substr(0, i) = substr(0, 3) = „icec”
dictionary.contains(„icec”) = false
....
prefix = substr(0, i) = substr(0, 7) = „icecream”
dictionary.contains(„icecream”) = true și am ajuns la capăt, întoarce rezultatul
„i like icecream”. Se reia apelul wordBreak(„likeicecream”, 12, „i ”) și așa mai departe

Performanță:

- Complexitate temporală: $O(2^N)$ unde N este lungimea șirului de la input.
- Complexitate spațială: $O(N^2)$ spațiu auxiliar.

Referințe

1. https://www.techiedelight.com/find-odd-occurring-element-logn-time/
2. https://www.geeksforgeeks.org/rearrange-a-string-so-that-all-same-characters-become-at-least-d-distance-away/
3. https://www.geeksforgeeks.org/maximum-height-of-tree-when-any-node-can-be-considered-as-root/
4. https://leetcode.com/problems/word-break/