Tema 0

Iancu Alexandru-Gabriel 324CD

University Politehnica of Bucharest
Faculty of Automatic Control and Computers

1 Divide et Impera - Floor of Sorted Array

1.1 Enunt

Pentru un vector sortat se cauta cea mai mare valoare mai mica sau egala cu o valoare data la intrare. Rezultatul este indexul din vector al valorii gasite, daca nu este gasita valoarea se intoarce -1.

1.2 Descrierea solutiei

Ideea principala este folosirea cautarii binare prin vector pana gasim valoarea cautata, acest lucru ne este permis intrucat vectorul este sortat. Ne orientam in functie de cum se pozitioneaza x fata de inceputul, sfarsitul si mijlocul vectorului de la iteratia curenta:

1. Daca primul element este mai mare decat x, atunci nu exista elemente mai mici decat x care sa satisfaca conditia si se intoarce -1 (daca se va ajunge la acest caz, se va observa de la prima iteratie);
2. Daca ultimul element este mai mic sau egal decat x, atunci el este sigur cel mai mare element din vector care sa satisfaca conditia si se va intoarce ultima pozitie din vectorul curent;
3. Daca valoarea de mijloc este mai mare decat x:
   (a) Daca valoarea de dinainte de mijloc este mai mai mica sau egala cu x, atunci acela este rezultatul si se intoarce mid - 1;
   (b) Altfel se merge recursiv pe vectorul din stanga (exclusiv) mijlocului pentru ca toate elementele de la mijloc incolo sunt mai mari decat x si de neinteres.
4. Altfel daca valoarea de mijloc este mai mica sau egala decat x:
   (a) Daca valoarea de dupa mijloc este mai mare decat x, atunci inseamna ca mijlocul este elementul cautat;
   (b) Altfel se merge recursiv pe vectorul din dreapta (exclusiv) mijlocului pentru ca toate elementele de dinainte de mijloc sunt mai mici decat x si de neinteres pentru ca ne-am asigurat valoarea de la mid + 1 care este sigur mai mica sau egala decat x si mai mare decat toate celelalte elemente din stanga.

Cum este cautare binara, recursivitatea va fi mereu pe o singura ramura din cele 2 optiuni prezente. Astfel in momentul in care o iteratie se termina si intoarce ceva, se va termina tot programul.

1.3 Algoritmul de rezolvare

```
floorSearch(arr[], low, high, x)
  if (arr[low] > x or low > high)
    return -1

  if (arr[high] <= x)
    return high

  mid = (low + high) / 2

  if (arr[mid] > x)
    if (arr[mid - 1] <= x)
      return mid - 1
    else
      return floorSearch(arr, low, mid - 1, x)

  if (arr[mid + 1] > x)
    return mid
  else
    return floorSearch(arr, mid + 1, high, x)
```

In al treilea if nu trebuie verificat ca mid > 0 (pentru arr[mid - 1]), pentru ca mid $= 0$ doar pentru:

1. (low $= 1$, high $= 0$), motiv in care nu s-ar ajunge acolo pentru ca ar intra in primul if ca respecta conditia low $>$ high;
2. (low $= 0$, high $= 1$), caz in care se poate discerne rezultatul uitandu-ne doar la orientarea lui x fata de arr[low] si arr[high].

1.4 Complexitate si eficienta

1. In cadrul unei iteratii se fac doar operatii de O(1) pe langa apelurile recursiv. $\implies$ f(n) = O(1)
2. La urmatoarea iteratie se introduce un vector cu un numar de elemente injumatatit fata de cel initial. $\implies$ b $= 2$
3. Intotdeauna se va merge pe o singura ramura o data conform unei implementari obisnuite de cautare binara. $\implies$ a $= 1$

Astfel algoritmul prezentat are urmatoarea relatie de recurenta:

$$T(n) = T(\frac{n}{2}) + \theta(1)$$

Care se poate rezolva folosind teorema Master:

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$f(n) = \theta(1) \in theta(n^0 \log_2^0 n) = \theta(1),\ k = 0$$

$$\implies \text{conform cazului 2 al teoremei Master } T(n) \in \theta(\log_2 n)$$

Complexitatea rezultata este cea asteptata de la un algoritm ce foloseste cautarea binara (ce are complexitate de $\theta(\log_2 n)$) si care are complexitate O(1) la fiecare iteratie recursiva. Algoritmul din punct de vedere al complexitatii temporale este optim pentru ca foloseste cautarea binara intr-un vector deja sortat(este mediul de lucru optim pentru a folosi cautarea binara). Din punct de vedere al complexitatii spatiale, daca luam in considerare stiva de executie, este de O($log_2 n$) pentru ca toate cele $log_2 n$ apeluri recursive vor fi stocate pe stiva. Motivul pentru care este O in loc de θ este pentru ca if-urile din cadrul unei iteratii pot oferi rezultatul prematur si astfel se poate ajunge la mai putine apeluri recursive. Din punct de vedere spatial se poate optimiza pentru a ajunge la O(1) folosind varianta iterativa a cautarii binara, astfel nemaiavand nevoie de stiva pentru stocarea de apeluri.

1.5 Exemplu

Pentru exemplu iau vectorul [1 3 7 8 15 20 23 30] si voi incerca folosind algoritmul de mai sus sa caut floor-ul lui 18:

- low = 0 high = 7:
  - vectorul iteratiei - [1 3 7 8 15 20 23 30]
  - mid = $\frac{0+7}{2} = 3$ (am facut rotunjire in jos)
  - arr[mid] = arr[3] = 8 < 18, iar arr[3 + 1] = 15 < 18 $\implies$ se merge recursiv pe a doua jumatate
  - se actualizeaza low = mid + 1 = 4, high ramane la fel
- low = 4 high = 7:
  - vectorul iteratiei - [15 20 23 30]
  - mid = $\frac{4+7}{2} = 5$
  - arr[mid] = arr[5] = 20 > 18, iar arr[5 - 1] = 15 < 18 $\implies$ 15 este floor-ul
  - se intoarce mid - 1 = 5 - 1 = 4

2 Greedy - Egyptian Fraction

2.1 Enunt

Reprezentati o fractie subunitara primita la intrare ca o fractie egipteana. O fractie egipteana e un mod de a reprezenta fractiile folosind doar unitati fractionare. O unitate fractionara este o fractie cu numaratorul 1 si numitorul un numar natural pozitiv. Spre exemplu $\frac{3}{4}$ poate fi reprezentat ca $\frac{1}{4} + \frac{1}{2}$.

2.2 Descrierea solutiei

Pentru a afla prima si cea mai mare unitate fractionara luam fractia initiala si o simplificam cu numaratorul. Rezultatul va fi o unitate fractionara cu numitorul un numar zecimal. In acest moment cautam fractia mai mica sau egala cu pseudo unitatea noastra fractionara calculata, cum ambele au numaratorul 1, este destul sa comparam numitorul celor 2. Daca pentru fractia per total cautam un numar

mai mic sau egal, la numitor e invers si astfel cautam un numar mai mare sau egal cu numarul zecimal calculat si anume primul astfel de numar intreg. Apoi pentru a gasi a doua unitate fractionara calculam noua fractie care va fi restul ramas din scaderea primei unitati fractionare din fractia initiala, se aplica apoi aceeasi pasi. Se tot continua in aceasta maniera pana cand numitorul calculat este un numar intreg, fractia simplificata rezultata fiind ultima unitate fractionara.

2.3 Algoritmul de rezolvare

```
egyptianFraction(nr, dr)
  fraction = nr/dr
  unit_dr = ceil(1/fraction)

  while ((1/fraction) is not an integer)
    print("1/" + unit_dr + " ")
    fraction -= 1/unit_dr
    unit_dr = ceil(1/fraction)

  print("1/" + (1/fraction) + " ")
```

Se calculeaza fractia initiala si apoi se face ceiling (rotunjire in sus) asupra inversei fractiei. Dupa se afiseaza unitatea fractionara si se scade din fractia curenta, iar mai apoi se calculeaza numitorul pentru unitatea fractionara. Se continua in acest fel pana cand inversa fractiei este un numar intreg. Dupa ce se iese din while se afiseaza ultima unitate fractionare. Functia ciel se considera ca functioneaza conform rotunjirii in sus matematice si anume urmatorul numar intreg mai mare sau egal numeric cu acesta.

Un mod de a verifica daca inversa fractiei nu este un numar intreg este sa testam daca numarul se incadreaza intr-o anumita toleranta primita ca parametru:

$$abs(\frac{1}{fraction} - 10^{-i}) > 0, \text{ unde i este magnitudinea tolerantei}$$

O problema totusi cu implementarea de mai sus este ca e imprecisa pentru ca e dependenta de precizia calculatorului si cum numerele fractionare pot avea un sir infinit de cifre, prin rotunjire se va pierde din exactitate. Varianta de pe pagina urmatoare rezolva problema

```
egyptian_fractionsv2(nr, dr)
  unit_dr = ceil(dr/nr);
  nr = nr * unit_dr - dr;
  dr *= unit_dr;

  while (dr > 0)
    print("1/" + unit_dr + " ")
    unit_dr = ceil(dr/nr);
    nr = nr * unit_dr - dr;
    dr *= unit_dr;
```

Se poate rezolva acest lucru prin calcularea la fiecare pas a numaratorului si a numitorului care isi vor pastra valoarea fiind numere intregi nerestrictionate de precizie. Conform noului algoritm la fiecare pas intai se calculeaza numaratorul unitatii fractionare ca in varianta anterioara, iar mai apoi se actualizeaza numitorul si numaratorul fractiei curente. Se tot face acest lucru pana cand numaratorul este 0 si anume s-a epuizat toata fractia. Legat de calcularea numitorului si numaratorului, se aduce fractia anterioara si unitatea fractionara la acelasi numitor amplificand ambele fractii cu numitorul celeilalte:

$$\frac{nr}{dr} - \frac{1}{unit\_dr} = \frac{nr * unit\_dr}{dr * unit\_dr} - \frac{dr}{dr * unit\_dr} = \frac{nr * unit\_dr - dr}{dr * unit\_dr}$$

Varianta a doua functioneaza exact, insa este limitata de dimensiunea maxima a variabilelor nr, dr si unit_dr.

2.4 Complexitate si analiza proprietatilor

Complexitatea temporala este O(u), unde u este egal cu numarul de unitati fractionare, pentru ca se efectueaza 3 operatii O(1) pentru fiecare unitate fractionara calculata (functia ciel se considera ca are complexitate O(1) fiind o simpla rotunjire in sus). Nu se poate determina numarul de unitati fractionare ale unei fractii fara ca aceasta sa fie procesata in prealabil. Complexitatea spatiala este O(1) daca lucram in octeti, daca luam numarul de biti in calcul este O(log n). Insa si la aceasta problema si la cea precedenta si la urmatoarele o sa calculez complexitatea spatiala in octeti.

Algoritmul este de tip Greedy, asa ca el calculeaza optimul local si anume unitatea fractionara pentru fractia curenta in fiecare pas al problemei. Se ia mereu cea mai mare unitate fractionara si cum orice fractie poate fi reprezentata ca o suma de unitati fractionare, atunci se poate spune ca algoritmul va ajunge mereu la optimul global prin optimele locale. Dupa ce se calculeaza o unitate fractionara, se sustrage aceasta unitate fractionara din fractia curenta, rezultatul fiind fractia urmatoarei iteratii. Astfel dupa calcularea optimului local se satisface calcularea corecta a urmatorului optim local si implicit al optimului global $\implies$ se respecta proprietatea alegerii locale.

Solutiile optime ale subproblemelor sunt unitatile fractionare, iar solutia problemei este multimea acestor unitati fractionare care este unica pentru orice

fractie data la intrare. Optimul global al problemei este o multime a rezultatelor optimelor locale si astfel solutia problemei contine si solutiile subproblemei $\implies$ se respecta proprietatea de substructura optima.

2.5 Exemplu

Pentru fractia $\frac{4}{13}$ folosind varianta 2 avem urmatorii pasi:

- Calcularea primei unitati fractionare:
  - se calculeaza numitorul fractiei unitate ca fiind $ciel(\frac{13}{4}) = 4 \implies$ prima unitata fractionara este $\frac{1}{4}$
  - numaratorul noii fractii este $4 * 4 - 13 = 16 - 13 = 3$
  - numitorul noii fractii este $13 * 4 = 52$

- Fractia curenta: $\frac{3}{52}$ Unitatile fractionare: $\frac{1}{4}$
  - numitorul fractiei unitate - $ciel(\frac{52}{3}) = 18 \implies$ a doua unitate fractionara este $\frac{1}{18}$
  - numaratorul - $3 * 18 - 52 = 54 - 52 = 2$
  - numitorul - $52 * 18 = 936$

- Fractia curenta: $\frac{2}{936}$ Unitatile fractionare: $\frac{1}{4}$ $\frac{1}{18}$
  - numitorul fractiei unitate - $ciel(\frac{936}{2}) = 468 \implies$ a treia unitate fractionara este $\frac{1}{468}$
  - numaratorul - $2 * 468 - 936 = 936 - 936 = 0 \implies$ s-a epuizat fractia initiala, s-au descoperit toate unitatile fractionare

Rezultatul este: $\frac{1}{4} + \frac{1}{18} + \frac{1}{468}$

3 Programare dinamica - nth Catalan Number

3.1 Enunt

Pentru un numar natural n primit la intrare, se doreste sa se calculeze al n-lea numar catalan. Numerele catalane apar intr-o multitudine de probleme de combinatorica. Spre exemplu numarul de arbori binari de cautare posibili cu N chei este echivalent cu al N-lea numar catalan. Astfel se poate adapta un algoritm de calculare a numerelor catalane pentru toate problemele de numarare aferente, scopul principal fiind descoperirea unui algoritm eficient de calculare a acestor numere pentru a putea rezolva eficient si celelalte probleme de interes.

3.2 Descrierea solutiei si relatia de recurenta

Numerele catalane respecta urmatoarea relatie de recurenta cunoscuta:

$$C_n = \begin{cases} 1, & \text{daca } n = 0 \\ \sum_{i=0}^{n-1} C_i * C_{n-1-i}, & \text{daca } n \in \mathbb{N}^* \end{cases} \quad (1)$$

Cum fiecare numar catalan e calculat in functie de toate numerele catalane precedente, programarea dinamica se prezinta drept un mod eficient de rezolvare a problemei. Prin metoda memorizarii se pot stoca toate numerele catalane calculate anterior pentru a fi folosite ulterior, vom folosi un vector pentru asta.

3.3 Algoritmul de rezolvare

```
nth_catalan_number(n)
  catalan = new vector[n + 1]
  catalan[0] = 1;
  for (i = 1; i <= n; i++)
    catalan[i] = 0
    for (j = 0; j < i; j++)
      catalan[i] += catalan[j] * catalan[i - j - 1]

  return catalan[n]
```

Se initializeaza cazul de baza si anume al 0-lea numar catalan care este egal cu 1. Apoi pentru fiecare numar catalan urmator pana la n se calculeaza suma conform index-ului aferent si al numerelor catalane precedente.

3.4 Complexitate

Algoritmul are 2 treceri imbricate de complexitate O(n), asa ca algoritmul are complexitate temporala de O(n^2), unde n este egal cu index-ul numarului catalan cautat. Complexitatea spatiala este O(n + 1) = O(n) pentru ca se foloseste un vector care salveaza al n-lea numar catalan si toate numere precedente necesare pentru calcularea acestuia de la al 0-lea la al (n - 1)-lea.

3.5 Exemplu

Pentru exemplificarea algoritmului, il voi folosi pentru a calcula al 4-lea numar catalan:

- catalan = [1 0] index = 1 (index-ul numarului catalan ce urmeaza sa fie calculat la pasul curent)
  - se iau indexi de la 0 la 0
  - pentru indexul 0 se adauga catalan[0] * catalan[1 - 0 - 1] = 1 $\implies$ catalan[1] = 0 + 1 = 1

- catalan = [1 1 0] index = 2
  - se iau indexi de la 0 la 1
  - pentru indexul 0 se adauga catalan[0] * catalan[2 - 0 - 1] = catalan[1] = 1 $\implies$ catalan[2] = 1
  - pentru indexul 1 se adauga catalan[1] * catalan[2 - 1 - 1] = 1 * 1 = 1 $\implies$ catalan[2] = 1 + 1 = 2

- catalan = [1 1 2 0] index = 31
  - se iau indexi de la 0 la 2
  - pentru indexul 0 se adauga catalan[0] * catalan[3 - 0 - 1] = catalan[2] = 2 $\implies$ catalan[3] = 2
  - pentru indexul 1 se adauga catalan[1] * catalan[3 - 1 - 1] = 1 $\implies$ catalan[3] = 2 + 1 = 3
  - pentru indexul 2 se adauga catalan[2] * catalan[3 - 2 - 1] = 2 $\implies$ catalan[3] = 3 + 2 = 5

- catalan = [1 1 2 5 0] index = 4
  - se iau indexi de la 0 la 3
  - pentru indexul 0 se adauga catalan[0] * catalan[4 - 0 - 1] = catalan[3] = 5 $\implies$ catalan[4] = 5
  - pentru indexul 1 se adauga catalan[1] * catalan[4 - 1 - 1] = catalan[2] = 2 $\implies$ catalan[4] = 5 + 2 = 7
  - pentru indexul 2 se adauga catalan[2] * catalan[4 - 2 - 1] = 2 $\implies$ catalan[4] = 7 + 2 = 9
  - pentru indexul 3 se adauga catalan[3] * catalan[4 - 3 - 1] = 5 $\implies$ catalan[4] = 9 + 5 = 14

Al 4-lea numar cautat va fi catalan[4] si anume 14.

4 Backtracking - Word Break Problem

4.1 Enunt

Se doreste sa se imparta un sir de caractere fara spatii, primit la intrare, intr-un sir de cuvinte despartite de spatii. Dictionarul ce contine cuvintele posibile poate fi primit la intrare sau hardcodat in cazul in care se doreste ca dictionarul sa fie constant de la o rulare la alta. Algoritmul va afisa toate posibilitatile de impartire a sirului de caractere folosind dictionarul. Exemplu:

- Dictionar = [”testare” ”test” ”are” ”testament” ”ament” ”ghetari”]
- Input = ”testaretestament”
- Output:
  - test are test ament
  - test are testament
  - testare test ament
  - testare testament

4.2 Descrierea solutiei

Se primeste sirul ce urmeaza sa fie prelucrat la intrare. Pentru acest sir se vor crea toate prefixele nenule posibile, spre exemplu daca sirul are dimensiune n, atunci o sa avem prefixul care contine primul caracter, cel care contine primele doua ... cel care contine toate caracterele din sir. Pentru fiecare astfel de sir se va verifica daca acesta face parte din dictionar, daca nu face atunci este ignorant. Cand se gaseste un prefix care satisface conditia, acesta este adaugat la rezultatul pentru combinatia respectiva si se continua prelucrarea restului de sir. Se tot adauga prefixe la rezultat si se tot prelucreaza siruri intermediare pana cand fie nu a fost un gasit niciun prefix care sa respecte conditia la un anumit pas, fie s-a ajuns la finalul sirului si au fost descoperite toate cuvintele din sir care apartin dictionarului. In ambele cazuri se face backtracking si se intoarce la pasul intermediar precedent pentru a se cauta urmatoarele combinatii.

4.3 Algoritmul de rezolvare

```
wordBreakUtil(str, n, result, dictionary[])
  for (i = 1; i <= n; i++)
    prefix = str.substring(0, i);

    if (dictionary.contains(prefix))
      if (i == n)
        print(result + prefix)
        return

      rest_of_str = str.substring(i, n)
      new_result = result + prefix + "␣"
      wordBreakUtil(rest_of_str, n - i, new_result, dictionary)

wordBreak(str, dictionary[])
  wordBreakUtil(str, str.size(), "", dictionary)
```

Se incepe algoritmul cu intregul sir si drept rezultat un sir gol ce urmeaza sa fie marit de cuvintele aferente. In prima prelucrare se vor extrage prefixele folosind functia substring(start, i) care selecteaza primele i caractere de la al start-lea caracter(indexarea sirului este de la 0). Apoi folosind functia contains(str) se va cauta prefix-ul str in dictionar, daca este gasit se intoarce adevarat si se continua iteratia curenta. In interior-ul if-ului daca prefixul e fix intregul sir curent inseamna ca s-a gasit o combinatie si este afisat rezultatul stocat alaturi de sirul curent verificat. Insa daca este oricare prefix, atunci se mai face un apel al functiei pentru prelucrarea restului de sir, adaugand prefix-ul curent la rezultat.

Pentru ca algoritmul se foloseste de apeluri recursive, rezultatele intermediare vor fi stocate pe stiva recursiva.

4.4 Complexitate si eficienta

Pentru fiecare dimensiune i posibila de la 1 la n a prefixelor, se va calcula pentru fiecare dimensiune j posibila de la 1 la (n - i) ce va calcula pentru fiecare dimensiune k posibila de la 1 la (n - i - j) s.a.m.d., unde n este lungimea sirului. Suma tuturor acestor calcule poate fi reprezentata astfel:

$$n * ((n-1) * ((n-2) * ((n-3) * ((n-4) * ..) + ..) + ..) + ..)$$

Doar luand in calcul una din numeroasele inmultiri avem n * (n - 1) * (n - 2) * .. 1 ce ne da o complexitate de O(n^n), se poate astfel afirma ca tot algoritmul are complexitate de O(n^n) si astfel are o complexitate exponentiala. Desigur nu se mereu trece prin toate posibilitatile pentru ca ignoram prefixele invalide, insa in cel mai rau caz si anume dictionarul este format din toate submultimile de caractere posibile ale intrarii, se vor considera mereu prefixele valide si astfel se va trece prin toate posibilitatile posibile(acesta fiind worst case scenario).

Functia substring copiaza i caractere din sirul iteratiei curente si astfel are complexitate O(n). Cautarea in dictionar are complexitate O(n * w), unde w este numarul de cuvinte din dictionar pentru ca se presupune ca foloseste o comparare caracter cu caracter intre siruri, iar sirul comparat este un prefix ce are dimnensiunea maxima n pentru cazul in care dictionarul este un simplu vector de siruri. Astfel complexitatea de facto a algoritmului este O($2 * w * n^2 * n^n$) = O($w * n^n$) pentru ca pentru fiecare prefix sunt maxim 2 apeluri de substring(o data pentru extragerea sa din sir, a doua oara in cazul in care este valid pentru extragerea restului de sir) si un apel de cautare in dictionar.

Complexitatea spatiala a algoritmului este O(n) pentru ca pot sa fie maxim n apeluri recursive si variabilele aferente prezente pe stiva, acest lucru intamplandu-se cand dictionarul are tot alfabetul sirului primit si astfel se apeleaza pentru cate un caracter o data.

Problema nu poate fi rezolvata mai eficient pentru ca acesta necesita sa afiseze toate posibilitatile si astfel programarea dinamica nu poate fi folosita. Ce se poate eficientiza sunt functiile folosite in interiorul algoritmului. Functia de cautare in dictionar ar putea fi eficienta daca ar cauta in timp ce ar compara intr-un vector de siruri sortat folosind cautarea binara, astfel ar cauta intai dupa primul caracter, apoi dupa al doilea s.a.m.d. sau dictionarul ar putea fi reprezentat ca un hash map cu o functie de hash eficienta cu putine coliziuni.

4.5 Exemplu

O sa folosesc exemplul prezentat in enunt ce avea dictionarul [”testare” ”test” ”are” ”testament” ”ament” ”ghetari”] si intrarea ”testaretestament”:

- sirul curent = ”testaretestament”
  - se cauta prefixele ”t”, ”te” si ”tes” in dictionar si nu sunt gasite
  - se gaseste ”test” in dictionar, se actualizeaza rezultatul $\implies$ ”test”
  - se merge pe restul sirului care este ”aretestament”

- sirul curent = "aretestament"
  - prefixele "a" si "re" nu se regasesc in dictionar
  - "are" se regaseste $\Longrightarrow$ "test are"

- sirul curent = "testament"
  - se gaseste "test" in dictionar $\Longrightarrow$ "test are test"

- sirul curent = "ament"
  - se gaseste "ament" in dictionar $\Longrightarrow$ "test are test ament"
  - s-a ajuns la finalul sirului, se afiseaza rezultatul si se face backtracking

- sirul curent = "testament"
  - se gaseste "testament" in dictionar $\Longrightarrow$ "test are testament"
  - s-a ajuns la final, se afiseaza rezultatul

- sirul curent = "aretestament"
  - nu se mai gaseste alt sir in dictionar, ne intoarcem la sirul precedent

- sirul curent = "testaretestament"
  - se gaseste "testare" $\Longrightarrow$ "testare"
  - se merge pe restul sirului "testament"

- sirul curent = "testament"
  - se gaseste "test" in dictionar $\Longrightarrow$ "testare test"

- sirul curent = "ament"
  - se gaseste "ament" in dictionar $\Longrightarrow$ "testare test ament"
  - s-a ajuns la finalul sirului, se afiseaza rezultatul si se face backtracking

- sirul curent = "testament"
  - se gaseste "testament" in dictionar $\Longrightarrow$ "testare testament"
  - s-a ajuns la final, se afiseaza rezultatul

- sirul curent = "testaretestament"
  - nu se mai gaseste alt prefix in dictionar
  - se inchide programul

Output-ul este identic cu cel prezentat in enunt.

5 Analiza comparativa

5.1 Divide et impera

Este o tehnica de programare prin care o problema se imparte in 2 sau mai multe subproblemele care vor fi evaluate separat si combinate ulterior, aceste subprobleme impartindu-se la randul pana se ajunge la un caz de baza. Astfel problemele pentru care aceasta tehnica poate fi aplicata sunt cele care pot fi impartite in subprobleme de aceeasi natura ca problema initiala.

Avantajul acestei tehnici este ca poate calcula eficient acest gen de probleme. De asemenea prin faptul ca se imparte in mai multe subprobleme, ea poate fi paralelizata pentru a le calcula simultan. Cum dimensiunea problemelor scade exponential, atunci algoritmul paralel ar avea per total o complexitate logaritmica. Dezavantajul este ca se foloseste de recursivitate si ocupa stiva de executie, astfel dimensiunea problemei este limitata de dimensiunea stivei. Daca se poate folosi greedy sau programare dinamica pentru a se rezolva problema, este recomandat sa se foloseasca acestea deoarece tind sa fie mai eficiente. Pe de alta parte algoritmii divide et impera sunt mai eficienti decat backtracking in rezolvarea problemelor ce necesita interogarea recursiva de parti ale problemei initiale.

5.2 Greedy

O tehnica ce functioneaza pe problemele care respecta proprietatea alegerii locale si proprietatea de substructura optima. Proprietatea alegerii locale ne spune ca prin calcularea optimului local al problemei se ajunge la optimul global. Cea de-a doua proprietate restrictioneaza solutia problemei sa fie o multime formata din solutiile subproblemelor. Asa ca problemele care pot fi rezolvate prin aceasta tehnica trebuie sa respecte cele doua proprietati sau cel putin proprietatea alegerii locale(cum e problema rucsacului fractionara). De asemenea in general greedy e o alegere buna pentru problemele care doresc calcularea de extreme si anume numarul maxim sau minim care respecta o anumita conditie.

Avantajul acestei tehnici este eficienta pentru ca proceseaza o singura data elementele. Intrarile care nu satisfac cerintele algoritmului pot fi prelucrate in prealabil, in general folosindu-se o sortare a datelor ce are complexitate de O(n * log n), acest lucru in cazul anumitor probleme se considera un dezavantaj. Daca problema se rezolva folosind un algoritm greedy in O(n), iar la aceasta se adauga o prelucrare de O(n * log n), tot algoritmul va fi de fapt O(n * log n).

Daca exista totusi un algoritm de programare dinamica care rezolva in O(n), este de preferat sa fie folosit acesta pentru ca nu este supraincarcat de o prelucrare in prealabil. Insa daca problema functioneaza cu greedy fara prelucrare, deseori este o varianta mai eficienta fata de programarea dinamica pentru ca tehnica greedy nu este interesata de detalii ca programarea dinamica si asa calculeaza doar operatiile necesare.

5.3 Programare dinamica

Programarea dinamica se foloseste pentru rezolvarea problemelor pentru care s-a observat sau se stie o relatie de recurenta. Asa ca pentru ca problemele sa poata fi rezolvate folosind aceasta tehnica, trebuie gandita in prealabil relatia de recurenta, daca o asemenea relatie nu exista sau e foarte greu de conceput nu se poate aplica programare dinamica. Algoritmele se folosesc de metoda memorizarii conform careia se stocheaza toti pasii calculati din relatia de recurenta pana se ajunge la elementul din sir dorit.

Avantajul vizibil al tehnicii este eficienta foarte mare pentru ca stocheaza toate valorile necesare intr-un vector pentru a nu le calcula ulterior, sarind astfel

peste anumite calcule redundante. Alt avantaj este faptul ca rezolva astfel eficient si problemele care doresc la iesire toate valorile din sirul relatiei de recurenta pana la final fara a se adauga un cost la timp sau spatiu consumat. Un dezavantaj evident este faptul ca se foloseste de spatiu pentru a compensa pentru eficienta in timp, asa ca pentru a folosi metoda memorizarii va avea mereu complexitati de cateva magnitudini mai mari decat O(1), comparativ cu Greedy care nu memorizeaza pasii intermediari si astfel poate sa aiba complexitate spatiala de O(1).

Alt dezavantaj este ca e greu paralelizabil pentru ca este restrans de relatia de recurenta. Daca pentru fiecare urmator element din relatia de recurenta este nevoie de cel calculat imediat precedent, atunci algoritmul este obligat sa fie liniar. In retrospectiva daca problema se poate rezolva cu divide et impera este probabil sa se ajunga la o eficienta timp mai mare daca se adapteaza aceasta la paralelism.

5.4 Backtracking

Backtracking este o tehnica ce se foloseste pentru problemele ce necesita toate combinatiile posibile aferente unei intrari sau care doresc sa se gaseasca prima combinatie care respecta o anumita conditie. Face acest lucru folosindu-se de operatia de backtracking care intoarce algoritmul de fiecare data la pasul precedent in urma intalnirii indeplinirii conditiei de succes sau de fail.

Avantajul tehnicii este ca e cea mai buna optiune pentru prelucrarea de permutari. Dezavantajul este ca generarea de permutari duce deseori la o complexitate exponentiala, insa daca asta doreste problema este o metoda mai buna de a face asta. In comparatie metoda naiva este mult mai ineficienta pentru ca aceasta ia toate cazurile chiar daca nu toate sunt de interes, prin metoda backtracking oprindu-se cautarea intr-o directie cand se indeplineste conditia de fail si se continua pe celelalte cazuri.

Daca cumva problema nu necesita toate permutarile sau nu trebuie trecute prin toate pentru a se ajunge la o solutie, atunci nu este recomandat sa se foloseasca backtracking, desi merge, din cauza eficientei scazute. In asemenea situatii se recomanda sa se foloseasca ceilalti algoritmi prezentati in functie de tipul problemei. In cazul in care de fapt este vorba de un sir ce satisface o relatie de recurenta este de preferat sa se foloseasca programarea dinamica. Backtracking-ul este recomandat doar daca nu exista nicio alta tehnica ce ar putea rezolva problema.

5.5 Exemple

Problema rucsacului Programarea dinamica reuseste sa ofere un algoritm de complexitate O(n * w), unde n este numarul de obiecte si w capacitatea rucsacului. Pe de alta parte, problema s-ar mai putea rezolva folosind backtracking, insa astfel s-ar ajunge la o complexitate exponentiala. Sunt multe probleme care s-ar putea rezolva folosind backtracking, insa trebuie intai sa ne gandim daca problema apartine P, iar daca da sa gasim o tehnica potrivita ce ne ofera

un algoritm cu timp polinomial. In situatia problemei rucsacului, algoritmul ce foloseste programarea dinamica in cel mai rau caz poate ajunge la o complexitate pseudo exponentiala in cazul in care capacitatea rucsacului este foarte mare, totusi in majoritatea cazurilor este de multe magnitudini mai bun decat varianta cu backtracking.

Nth ugly number Problema calcularii numerelor "urate", numere a caror divizori sunt doar 2, 3 sau 5, are o metoda ce foloseste programarea dinamica si reuseste sa calculeze in timp O(n). Totusi de remarcat este faptul ca se poate folosi si cautarea binara, lucru ce ne duce la o complexitatea temporala de O(log n). Astfel programarea dinamica are dezavantajul, in anumite situatii, ca nu poate sa produca o complexitate mai buna decat O(n), pe cand divide et impera reuseste sa atinga O(log n) folosind cautarea binara. De asemenea in aceasta situatie cautarea binara este o alegere mai buna pentru ca nu foloseste spatiu auxiliar pe langa stiva de executie.

Asa ca daca problema se poate face folosind doar o singura cautare binara(sau un numar constant de cautari binare), atunci nu merita sa cauti o relatie de recurenta pentru un algoritm cu programare dinamica pentru ca e putin probabil sa o faca mai bine. In general programarea dinamica si greedy sunt bune pentru ca fac o singura trecere prin sirul dat la intrare sau cel ce trebuie creat treptat, insa astfel sunt constransi in a avea deseori o complexitate de maxim O(n).

Word Wrap Problem Pentru un sir de caractere se doreste sa se puna newline astfel incat cuvintele sa ramana intregi, se primeste la intrare dimensiunea maxima a unei linii. Prima metoda si cea mai simpla este greedy care pur si simplu ia fiecare cuvant pe rand si umple fiecare linie cat se poate ca mai apoi sa treaca la linia urmatoare. Metoda functioneaza bine in majoritatea cazurilor, insa pentru anumite cazuri nu balanseaza cum trebuie spatiile pentru ca ii pasa doar de optimul local, nu de intreaga panorama. Algoritmul este totusi destul de eficient, trecand doar o data prin toate cuvintele are o complexitate de O(n) si este inca folosit de MS Word si OpenOffice.org.

Totusi problema mai poate fi rezolvata folosind programare dinamica si anume relatia de recurenta prezentata in cartea lui Cormen. Aceasta foloseste o matrice si astfel umplerea acesteia produce o complexitate temporala a algortimului de $O(n^2)$. Algoritmul este optim in toate cazurile, insa nu este la fel de rapid ca si Greedy. A trebuit dat la schimb eficienta pentru corectitudine schimband Greedy cu programarea dinamica. In multe probleme va trebui sa facem aceasta alegere in functie de ce ne intereseaza cu adevarat sau ce intereseaza firma. Daca o simpla divergenta de la optimalitate ar duce la mari erori in sistem, atunci ne permitem sa dam la schimb eficienta. Altfel daca nu e asa de important, atunci mai bine folosim ce se executa mai repede. In problema prezentata optimalitatea nu e asa importanta, nu conteaza asa mult cat de balansate sunt spatiile, conteaza in principal ca liniile sa se incadreze in dimensiunea precizata pastrand intregimea cuvintelor. Viteza si simplitatea sunt motivele pentru care metoda Greedy inca este folosita in procesoarele word.

References

1. Floor in a Sorted Array,
https://www.geeksforgeeks.org/floor-in-a-sorted-array/

2. Greedy Algorithm for Egyptian Fraction,
https://www.geeksforgeeks.org/greedy-algorithm-egyptian-fraction/

3. Program for nth Catalan Number,
https://www.geeksforgeeks.org/program-nth-catalan-number/

4. Catalan number,
https://en.wikipedia.org/wiki/Catalan_number

5. Word Break Problem using Backtracking,
https://www.geeksforgeeks.org/word-break-problem-using-backtracking/

6. 0-1 Knapsack Problem — DP-10,
https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

7. Ugly Numbers,
https://www.geeksforgeeks.org/ugly-numbers/

8. Word Wrap Problem — DP-19,
https://www.geeksforgeeks.org/word-wrap-problem-dp-19/

9. LNCS Homepage, http://www.springer.com/lncs.

Toate link-urile au fost accesate ultima data pe 25 aprilie 2021.