UNIVERSITATEA BABEȘ-BOLYAI

ZeceLaInfo

Generarea Elementelor Combinatoriale. Backtracking.

Consultații Facultatea de Matematică și Informatică, Universitatea Babeș-Bolyai, Cluj-Napoca - 13 ianuarie 2024

Stud. Paul-Ioan Somesan

Backtracking

- Backtracking-ul este o tehnica de programare care presupune indentificarea tuturor solutiilor posibile a unei probleme.
- Este un algoritm extrem de lent, dar in foarte multe cazuri reprezinta abordarea unica prin care se poate rezolva problema.
- In ideea imbunatatirii complexitatii se pot face multe optimizari care pot afecta dramatic numarul de pasi efectuati in rezolvarea problemei.

Elemente Combinatoriale

- Permutari
  - Vom genera toate permutarile unei multimi cu N elemente.
  - {1, 2, 3, …, N} -> N! permutari => complexitate minim O(N! * N)
- Combinari
  - Vom genera toate combinarile unei multimi de N elemente luate cate K.
  - {1, 2, 3, …, N}, K -> C_n^k combinari => complexitate minim O(C_n^k * k)
- Aranjamente
  - Vom genera toate aranjamentele unei multimi de N elemente luate cate K.
  - {1, 2, 3, …, N}, K -> A_n^k aranjamente => complexitate minim O(A_n^k * k)

Permutari

- Desi complexitatea este foarte mare, scopul nostru principal este sa ne apropiem cat de mult posibil de complexitatea minima descrisa anterior.
- Algoritmul acesta reuseste sa atinga complexitatea minima O(N! * N)

```
4  int n, x[11], P[11];
5  void generare_permutari(int pas){
6      for(int i = 1; i <= n; ++i)
7          if(!P[i]){
8              x[pas] = i;
9              P[i] = 1;
10             if(pas == n){
11                 for(int j = 1; j <= n; ++j)
12                     cout << x[j] << ' ';
13                 cout << '\n';
14             }
15             else generare_permutari(pas + 1);
16             P[i] = 0;
17         }
18 }
```

Aranjamente

- Desi complexitatea este foarte mare, scopul nostru principal este sa ne apropiem cat de mult posibil de complexitatea minima descrisa anterior.
- Algoritmul acesta reuseste sa atinga complexitatea minima O(A_n^k * k)

```
4  int n, k, x[11], P[11];
5  void generare_aranjamente(int pas){
6      for(int i = 1; i <= n; ++i)
7          if(!P[i]){
8              P[i] = 1;
9              x[pas] = i;
10             if(pas == k){
11                 for(int j = 1; j <= k; ++j)
12                     cout << x[j] << ' ';
13                 cout << '\n';
14             }
15             else generare_aranjamente(pas + 1);
16             P[i] = 0;
17         }
18 }
```

Combinari

```
4  int n, k, x[11];
5  void generare_combinari(int pas){
6      for(int i = x[pas-1] + 1; i <= n - k + pas; ++i){
7          x[pas] = i;
8          if(pas == k){
9              for(int j = 1; j <= k; ++j)
10                 cout << x[j] << ' ';
11             cout << '\n';
12         }
13         else generare_combinari(pas + 1);
14     }
15 }
```

- Desi complexitatea este foarte mare, scopul nostru principal este sa ne apropiem cat de mult posibil de complexitatea minima descrisa anterior.
- Algoritmul acesta reuseste sa atinga complexitatea minima O(C_n^k * k).

Determinarea modalitatilor de a rezolva o problema

- Exista multe probleme care presupun determinarea tuturor posibilitatilor de a o rezolva si analizarea variantelor pentru a determina solutia optima.
- De asemenea, multe probleme cer pur si simplu determinarea numarului de solutii posibile.
- Un bun exemplu in aceasta directie ar putea fi determinarea partitiilor unui numar. Partitiile unui numar reprezinta toate modalitatiile de a scrie numarul ca suma de numere naturale (de obicei anumite numere naturale (pare / impare / prime / neprime etc.))

Partitiile unui numar

```
4   int n, x[21];
5   void determinare_partitii_numar(int k, int sum){
6       for(int i = x[k-1]; sum + i <= n; i += 2){
7           x[k] = i;
8           if(sum + i < n)
9               determinare_partitii_numar(k + 1, sum + i);
10          else if(sum + i == n){
11              for(int j = 1; j <= k; ++j)
12                  cout << x[j] << ' ';
13              cout << '\n';
14          }
15      }
16  }
```

Determinarea numarului de solutii la problema anterioara

- Un alt tip de problema de backtracking vizeaza determinarea numarului de solutii. Aici vom vedea un tip de algoritm backtracking diferit, mult mai interesant!
- Ceea ce vedeti pe ecran este tot un algoritm de tip backtracking.

```
6   int numar_solutii(int ant, int sum){
7       if(sum == n)
8           return 1;
9       int nr_sol = 0;
10      for(int i = ant; sum + i <= n; i += 2)
11          nr_sol += numar_solutii(i, sum + i);
12      return nr_sol;
13  }
```

```
5   int numar_solutii(int n, int ant, int sum){
6       if(sum == n)
7           return 1;
8       int nr_sol = 0;
9       for(int i = ant; sum + i <= n; i += 2)
10          nr_sol += numar_solutii(n, i, sum + i);
11      return nr_sol;
12  }
```

O grila foarte interesanta de la admiterea din Iulie 2021.

- Alaturat puteti vedea o grila din Subiectul de Admitere la FMI – UBB 2021, o grila dificila din capitolul de Backtracking.
- Secventa calculeaza si returneaza suma maxima a unui subsir de elemente neconsecutive din sirul *arr*.
- Pentru a determina rezultatul, algoritmul calculeaza TOATE sumele si o retine pe cea mai mare!

27. Se consideră subalgoritmul `f2(a,b)` cu parametrii ***a*** și ***b*** numere naturale, și subalgoritmul `f(arr, i, n, p)` având ca parametri șirul ***arr*** cu ***n*** numere întregi (***arr***[1], ***arr***[2], …, ***arr***[n]), și numerele întregi ***i*** și ***p***.

```
Subalgoritm f2(a, b):
    Dacă a > b atunci
        returnează a
    altfel
        returnează b
    SfDacă
SfSubalgoritm

Subalgoritm f(arr, i, n, p):
    Dacă i = n atunci
        returnează 0
    SfDacă
    n1 ← f(arr, i + 1, n, p)
    n2 ← 0
    Dacă p + 1 ≠ i atunci
        n2 ← f(arr, i + 1, n, i) + arr[i]
    SfDacă
    returnează f2(n1, n2)
SfSubalgoritm
```

Precizați care este rezultatul apelului `f(arr, 1, 9, -10)`, dacă șirul ***arr*** conține valorile (10, 1, 3, 4, 8, 12, 1, 11, 6).

A. 24
B. 37
C. 26
D. 56

Cum putem exersa acest tip de backtracking?

- Desi pe site-urile de probleme nu sunt / sunt extrem de putine astfel de probleme, noi putem sa determinam la orice problema de backtracking numarul de solutii fara sa le afisam. Provocarea este ca aceste functii sa nu fie de tip void, ci sa returneze rezultatul fara a se folosi variabile globale.
- Va recomand sa alegeti de pe PbInfo.ro orice problema cu backtracking si sa incercati sa scrieti un algoritm care determina numarul de solutii ale problemei respective.

Determinarea tuturor sirurilor corect parantezate de lungime n

- Un sir este corect parantezat daca fiecare paranteza deschisa, se inchide pana la finalul sirului si oricare ar fi o pozitie din sir, numarul de paranteze inchise pana acolo nu este mai mare decat numarul de paranteze deschise.

```
15  char sol[21];
16  void back(int n, int k = 0, int cnt_D = 0, int cnt_I = 0){
17      if(k == n)
18          cout << sol << '\n';
19      else{
20          if(cnt_D < n / 2){
21              sol[k] = '(';
22              back(n, k+1, cnt_D + 1, cnt_I);
23          }
24          if(cnt_I < cnt_D){
25              sol[k] = ')';
26              back(n, k+1, cnt_D, cnt_I + 1);
27          }
28      }
29  }
```

```
4   int numar_solutii(int n, int k = 0, int cnt_D = 0, int cnt_I = 0){
5       if(k == n)
6           return 1;
7       int nr_sol = 0;
8       if(cnt_D < n / 2)
9           nr_sol += numar_solutii(n, k+1, cnt_D + 1, cnt_I);
10      if(cnt_I < cnt_D)
11          nr_sol += numar_solutii(n, k+1, cnt_D, cnt_I + 1);
12      return nr_sol;
13  }
```

Va multumesc pentru atentie!

@ZeceLaInfo

@zece_la_info

UNIVERSITATEA BABEȘ-BOLYAI