Laborator 5: Backtracking

Responsabili:

- Radu Stochitoiu [mailto:radu.stochitoiu@gmail.com]
- Razvan Chitu [mailto:razvan.m.chitu@gmail.com]

Obiective laborator

- Intelegerea notiunilor de baza despre backtracking;
- Insusirea abilitatilor de implementare a algoritmilor bazati backtracking;
- Rezolvarea unor probleme NP-complete în timp exponential.

Precizari initiale

Toate exemplele de cod se gasesc in demo-lab05.zip. Acestea apar incorporate si in textul laboratorului pentru a facilita parcurgerea cursiva a laboratorului.

Ce este Backtracking?

Backtracking este un algoritm care cauta **una sau mai multe solutii** pentru o problema, printr-o cautare exhaustiva, mai eficienta insa in general decat o abordare „genereaza si testeaza", de tip „forta bruta", deoarece un candidat partial care nu duce la o solutie este abandonat. Poate fi folosit pentru orice problema care presupune o cautare in **spatiul starilor**. In general, in timp ce cautam o solutie e posibil sa dam de o infundatura in urma unei alegeri gresite sau sa gasim o solutie, dar sa dorim sa cautam in continuare alte solutii. In acel moment trebuie sa ne intoarcem pe pasii facuti (**backtrack**) si la un moment dat sa luam alta decizie. Este relativ simplu din punct de vedere conceptual, dar complexitatea algoritmului este exponentiala.

Importanţa – aplicaţii practice

Exista foarte multe probleme (de exemplu, probleme NP-complete sau NP-dificile) care pot fi rezolvate prin algoritmi de tip backtracking mai eficient decat prin „forta bruta" (adica generarea tuturor alternativelor si selectarea solutiilor). Atentie insa, complexitatea computationala este de cele mai multe ori exponentiala. O eficientizare se poate face prin combinarea cu tehnici de propagare a restrictiilor. Orice problema care are nevoie de parcurgerea spatiului de stari se poate rezolva cu backtracking.

Descrierea problemei si a rezolvarilor

Pornind de la strategiile clasice de parcurgere a **spatiului de stari**, algoritmii de tip backtracking practic enumera un set de candidati partiali, care, dupa completarea definitiva, pot deveni solutii potentiale ale problemei initiale. Exact ca strategiile de **parcurgere în latime/adancime** si backtracking-ul are la baza expandarea unui nod curent, iar determinarea solutiei se face într-o maniera incrementala. Prin natura sa, backtracking-ul este recursiv, iar în arborele expandat top-down se aplica operatii de tipul pruning (taiere) daca solutia partiala nu este valida.

Algoritm de baza

```
/* Domain - domeniul curent (cu un cardinal mai mic decat la pasul trecut)
Solution - solutia curenta pe care o extindem catre cea finala */
back(Domain, Solution):
    if check(Solution):
        print(Solution)
        return

    for value in Domain:
        NextSolution = Solution.push(value)
        NextDomain = Domain.erase(value)
        back(NextDomain, NextSolution)
```

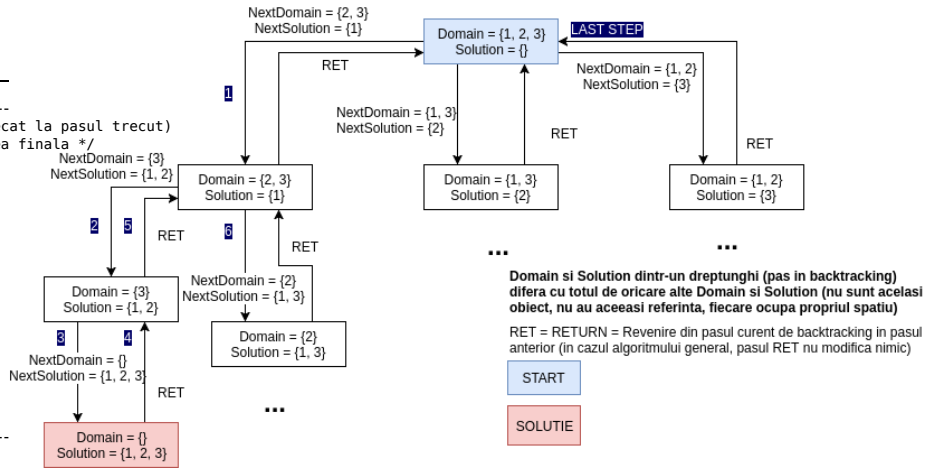
Algoritm de baza (modificat pentru transmitere prin referinta)

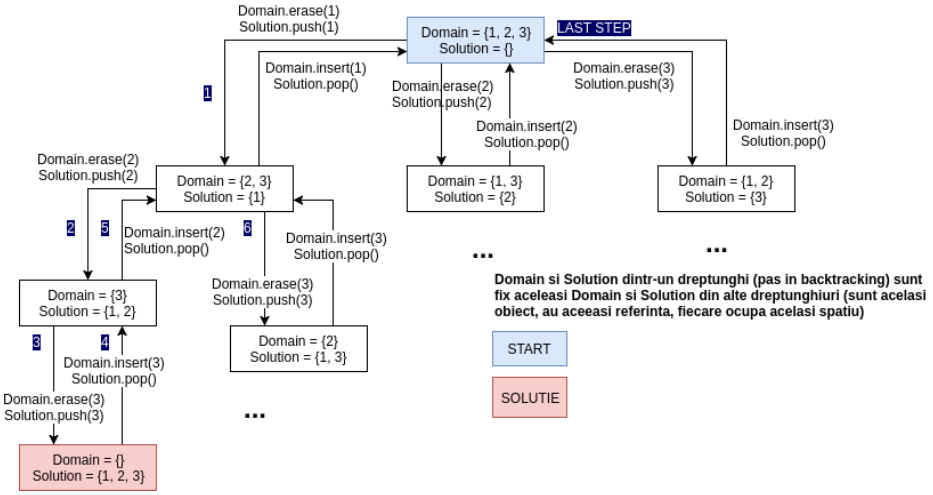
```
/* Domain - domeniul curent (cu un cardinal mai mic decat la pasul trecut)
Solution - solutia curenta pe care o extindem catre cea finala */
back(Domain, Solution):
    if check(Solution):
        print(Solution)
        return

    for value in Domain:
        /* DO */
        Solution = Solution.push(value)
        Domain = Domain.erase(value)

        /* RECURSION */
        back(Domain, Solution)

        /* UNDO */
        Solution = Solution.pop()
        Domain = Domain.insert(value)
```





Exemple clasice

Ne vom ocupa in continuare de urmatoarele probleme:

- Permutari
- Combinari
- Aranjamente
- Submultimi
- Genererare de siruri
- Problema damelor
- Problema soricelului
- Tic-Tac-Toe
- Sudoku
- Ultimate Tic-Tac-Toe

Sudoku si Ultimate Tic-Tac-Toe sunt probleme foarte grele. In general nu putem explora tot spatiul starilor pentru un input arbitrar dat.

Permutari

Enunt

Se da un numar N. Sa se genereze toate permutarile multimii formate din toate numerele de la 1 la N.

Exemple

N = 3 ⇒ M = {1, 2, 3}

Solutie:

- {1, 2, 3}
- {1, 3, 2}
- {2, 1, 3}
- {2, 3, 1}
- {3, 1, 2}
- {3, 2, 1}

Solutii

Backtracking (algoritmul in cazul general)

```
/* deoarece numerele sunt sterse din domeniu odata ce sunt folosite, solutia generata este garantata
sa nu contina duplicate. Astfel, atunci cand domeniul ajunge vid, solutia este intotdeauna corecta */
bool check(std::vector<int> solution) {
    return true;
}

void printSolution(std::vector<int> solution) {
    for (int s : solution) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}

void back(std::vector<int> domain, std::vector<int> solution) {
    /* dupa ce am folosit toate elementele din domeniu putem verifica daca
    am gasit o solutie */
    if (domain.size() == 0) {
        if(check(solution)) {
            printSolution(solution);
        }
        return;
    }

    /* incercam sa adaugam in solutie toate valorile din domeniu, pe rand */
    for (unsigned int i = 0; i < domain.size(); ++i) {
        /* cream o solutie noua si un domeniu nou care sunt identice cu cele
        de la pasul curent */
        std::vector<int> newSolution(solution), newDomain(domain);

        /* adaugam in noua solutie elementul ales din domeniu */
```

```
        newSolution.push_back(domain[i]);
        /* stergem elementul ales din noul domeniu */
        newDomain.erase(newDomain.begin() + i);

        /* apelam recursiv backtracking pe noul domeniu si noua solutie */
        back(newDomain, newSolution);
    }
}

int main() {
    /* dupa ce am citit n initializam domeniul cu n elemente, numerele de la 1 la n,
    iar solutia este vida initial */
    std::vector<int> domain(n), solution;
    for (int i = 0; i < n; ++i) {
        domain[i] = i + 1;
    }

    /* apelam backtracking pe domeniul nostru, cautand solutia in vectorul solution */
    back(domain, solution);
}
```

Apelarea initiala (din “int main”) se face astfel: “back(domain, solution);”, unde domain reprezinta un vector cu elementele de la 1 la N, iar solution este un vector gol.

Nu este indicata implementarea backtracking-ului astfel deoarece este foarte costisitor din punct de vedere al memoriei, deoarece cream noi domenii si solutii la fiecare pas.

Complexitate

Solutia va avea urmatoarele complexitati:

- complexitate temporala : $T(n) = O(n * n!) = O(n!)$
  - explicatie : Complexitate generarii permutarilor, $O(n!)$, se inmulteste cu complexitatea copierii vectorilor solutie si domeniu si a stergerii elementelor din domeniu, $O(n)$
- complexitate spatiala : $S(n) = O(n^2)$
  - explicatie : Fiecare nivel de recursivitate are propria lui copie a solutiei si a domeniului. Sunt n nivele de recursivitate, deci complexitatea spatiala este $O(n * n) = O(n^2)$

Backtracking (date transmise prin referinta)

```
/* deoarece numerele sunt sterse din domeniu odata ce sunt folosite, solutia generata este garantata sa nu contina duplicate. Astfel, atunci cand domeniul ajunge vid
bool check(std::vector<int> solution) {
    return true;
}

void printSolution(std::vector<int> &solution) {
    for (int s : solution) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}

void back(std::vector<int> &domain, std::vector<int> &solution) {
    /* dupa ce am folosit toate elementele din domeniu putem verifica daca
    am gasit o solutie */
    if (domain.size() == 0) {
        if(check(solution)) {
            printSolution(solution);
        }
        return;
    }

    /* incercam sa adaugam in solutie toate valorile din domeniu, pe rand */
    for (unsigned int i = 0; i < domain.size(); ++i) {
        /* retinem valoarea pe care o scoatem din domeniu ca sa o readaugam dupa
        apelarea recursiva a backtracking-ului */
        int tmp = domain[i];

        /* adaug elementul curent la potentiala solutie */
        solution.push_back(domain[i]);
        /* sterg elementul curent din domeniu ca sa il pot pasa prin referinta
        si sa nu fie nevoie sa creez alt domeniu */
        domain.erase(domain.begin() + i);

        /* apelez recursiv backtracking pe domeniul si solutia modificate */
        back(domain, solution);

        /* refac domeniul si solutia la modul in care aratau inainte de apelarea
        recursiva a backtracking-ului, adica readaug elementul eliminat in
        domeniu si il sterg din solutie */
        domain.insert(domain.begin() + i, tmp);
        solution.pop_back();
    }
}

int main() {
    /* dupa ce am citit n initializam domeniul cu n elemente, numerele de la 1 la n,
    iar solutia este vida initial */
    std::vector<int> domain(n), solution;
    for (int i = 0; i < n; ++i) {
        domain[i] = i + 1;
    }

    /* apelam backtracking pe domeniul nostru, cautand solutia in vectorul solution */
    back(domain, solution);
}
```

Apelarea initiala (din “int main”) se face astfel: “back(domain, solution);”, unde domain reprezinta un vector cu elementele de la 1 la N, iar solution este un vector gol.

Complexitate

Solutia va avea urmatoarele complexitati:

- complexitate temporala : $T(n) = O(n * n!)$
  - explicatie : Complexitate generarii permutarilor, $O(n!)$, se inmulteste cu complexitatea stergerii elementelor din domeniu, $O(n)$
- complexitate spatiala : $S(n) = O(n)$
  - explicatie : Spre deosebire de solutia anterioara, toate nivelele de recursivitate folosesc aceeasi solutie si acelasi domeniu. Complexitatea spatiala este astfel redusa la $O(n)$

Abordarea aceasta este mai eficienta decat cea generala prin evitarea folosirii memoriei auxiliare.

Backtracking (taierea ramurilor nefolositoare)

```
bool check(std::vector<int> &solution) {
    return true;
}

void printSolution(std::vector<int> &solution) {
    for (auto s : solution) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}

void back(int step, int stop, std::vector<int> &domain,
        std::vector<int> &solution, std::unordered_set<int> &visited) {

    /* vom verifica o solutie atunci cand am adaugat deja N elemente in solutie,
    adica step == stop */
    if (step == stop) {
        /* deoarece am avut grija sa nu se adauge duplicate, "check()" va returna
        intotdeauna "true" */
        if(check(solution)) {
            printSolution(solution);
        }
        return;
    }

    /* Adaugam in solutie fiecare element din domeniu care *NU* a fost vizitat
    deja renuntand astfel la nevoia de a verifica duplicatele la final prin
    functia "check()" */
    for (unsigned int i = 0; i < domain.size(); ++i) {
        /* folosim elementul doar daca nu e vizitat inca */
        if (visited.find(domain[i]) == visited.end()) {
            /* il marcam ca vizitat si taiem eventuale expansiuni nefolositoare
            viitoare (ex: daca il adaug in solutie pe 3 nu voi mai avea
            niciodata nevoie sa il mai adaug pe 3 in continuare) */
            visited.insert(domain[i]);

            /* adaugam elementul curent in solutie pe pozitia pasului curent
            (step) */
            solution[step] = domain[i];

            /* apelam recursiv backtracking pentru pasul urmator */
            back(step + 1, stop, domain, solution, visited);

            /* stergem vizitarea elementului curent (ex: pentru N = 3, dupa ce
            la pasul "step = 0" l-am pus pe 1 pe prima pozitie in solutie si
            am continuat recursiv pana am ajuns la solutiile {1, 2, 3} si
            {1, 3, 2}, ne dorim sa il punem pe 2 pe prima pozitie in solutie si
            sa continuam recursiv pentru a ajunge la solutiile {2, 1, 3} etc.) */
            visited.erase(domain[i]);
        }
    }
}

int main() {
    /* dupa ce am citit n initializam domeniul cu n elemente, numerele de la 1 la n,
    iar solutia este initializata cu un vector de n elemente (deoarece o permutare
    contine n elemente) */
    std::vector<int> domain(n), solution(n);
    std::unordered_set<int> visited;
    for (int i = 0; i < n; ++i) {
        domain[i] = i + 1;
    }

    /* apelam back cu step = 0 (atatea elemente avem adaugate in solutie),
    stop = n (stim ca vrem sa adaugam n elemente in solutie pentru ca o
    permutare e alcatuita din n elemente), domain este vectorul de valori
    posibile, solution este vectorul care simuleaza stiva pe care o vom
    umple, visited este un unordered_set (initial gol) in care retinem daca
    un element din domeniu se afla deja in solutia curenta la un anumit pas */
    back(0, n, domain, solution, visited);
}
```

Apelarea initiala (din “int main”) se face astfel: “back(0, n, domain, solution, visited);”, unde domain reprezinta un vector cu elementele de la 1 la N, iar solution este un vector de n elemente, 0 este pasul curent, n este pasul la care dorim sa ne oprim, iar visited este map-ul care ne permite sa tinem cont de ce elemente au fost vizitate sau nu.

Complexitate

Solutia va avea urmatoarele complexitati:

- complexitate temporala : $T(n) = O(n * n!) = O(n!)$
  - explicatie : Complexitate generarii permutarilor, O(n!), se inmulteste cu complexitatea

iterarii prin domeniu, $O(n)$

- complexitate spatiala : $S(n) = O(n)$
  - explicatie : Toate nivelele de recursivitate folosesc aceeasi solutie si acelasi domeniu.

Aceasta solutie este optima si are complexitate temporala $T(n) = O(n!)$. Nu putem sa obtinem o solutie mai buna, intrucat trebuie sa generam n! permutari.

De asemenea, este optima si din punct de vedere spatial, intrucat trebuie sa avem $S(n) = O(n)$, din cauza stocarii permutarii generate.

Combinari

Enunt

Se dau numerele N si K. Sa se genereze toate combinarile multimii formate din toate numerele de la 1 la N, luate cate K.

Exemple

N = 4, K = 2 ⇒ M = {1, 2, 3, 4}

Solutie:

- {1, 2}
- {1, 3}
- {1, 4}
- {2, 3}
- {2, 4}
- {3, 4}

Solutii

Backtracking (taierea ramurilor nefolositoare)

```
bool check(std::vector<int> &solution) {
    return true;
}

void printSolution(std::vector<int> &solution, std::vector<int> &domain, int stop) {
    for (unsigned i = 0; i < stop; ++i) {
        std::cout << domain[solution[i]] << " ";
    }
    std::cout << "\n";
}

void back(int step, int stop, std::vector<int> &domain,
        std::vector<int> &solution) {
    /* vom verifica o solutie atunci cand am adaugat deja K elemente in solutie,
    adica step == stop */
    if (step == stop) {
        /* deoarece am avut grija sa se adauge elementele doar in ordine
        crescatoare, "check()" va returna intotdeauna "true" */
        if(check(solution)) {
            printSolution(solution, domain, stop);
        }
        return;
    }

    /* daca este primul pas, alegem fiecare element din domeniu ca potential
    candidat pentru prima pozitie in solutie; altfel, pentru a elimina ramurile
    in care de exemplu {2, 1} se va genera dupa ce s-a generat {1, 2} (adica
    ar fi duplicat), vom folosi doar elementele din domeniu care sunt mai mari
    decat ultimul element adaugat in solutie (solution[step - 1]) */
    unsigned i = step > 0 ? solution[step - 1] + 1 : 0;
    for (; i < domain.size(); ++i) {
        solution[step] = i;
        back(step + 1, stop, domain, solution);
    }
}

int main() {
    /* dupa ce citim n si k initializam domeniul cu valorile de la 1 la n,
    iar solutia este initializata cu un vector de k elemente (fiindca o
    combinare de "n luate cate k" are k elemente) */
    std::vector<int> domain(n), solution(k);
    for (int i = 0; i < n; ++i) {
        domain[i] = i + 1;
    }

    back(0, k, domain, solution);
}
```

In aceasta solutie ne bazam pe faptul ca toate combinarile pot fi generate in ordine crescatoare, adica solutia {1, 3, 4} e echivalenta cu {4, 1, 3}.

Aceasta solutie este optima intrucat toate solutiile generate sunt corecte (de aceea functia check intoarce true). Deoarece problema cere obtinerea tuturor combinarilor, aceasta complexitate nu poate fi mai mica de Combinari(n, k).

Complexitate

Solutia va avea urmatoarele complexitati:

- complexitate temporala : $T(n) = O(Combinari(n, k))$
- complexitate spatiala : $S(n) = O(n + k) = O(n)$
    - explicatie : $k <= n$, deci $O(n + k) = O(n)$

Problema soricelului

Enunt

Se da un numar N si o matrice patratica de dimensiuni N x N in care elementele egale cu 1 reprezinta ziduri (locuri prin care nu se poate trece), iar cele egale cu 0 reprezinta spatii goale. Aceasta matrice are un soricel in celula (0, 0) si o bucata de branza in celula (N - 1, N - 1). Scopul soricelului e sa ajunga la bucata de branza. Afisati toate modurile in care poate face asta stiind ca acesta poate merge doar in dreapta sau in jos cu cate o celula la fiecare pas.

Exemple

- 2
- 0 1
- 0 0

Exista 1 drum posibil:

- (0, 0)→(1, 0)→(1, 1)

- 3
- 0 0 0
- 0 1 0
- 0 0 0

Exista 2 drumuri posibile:

- (0,0)→(0,1)→(0,2)→(1,2)→(2,2)

- (0,0)→(1,0)→(2,0)→(2,1)→(2,2)

- 4
- 0 0 0 1
- 0 1 1 0
- 0 0 0 0
- 0 0 0 0

Exista 4 drumuri posibile:

- (0,0)→(1,0)→(2,0)→(2,1)→(2,2)→(2,3)→(3,3)
- (0,0)→(1,0)→(2,0)→(2,1)→(2,2)→(3,2)→(3,3)
- (0,0)→(1,0)→(2,0)→(2,1)→(3,1)→(3,2)→(3,3)
- (0,0)→(1,0)→(2,0)→(3,0)→(3,1)→(3,2)→(3,3)

Solutii

Backtracking (transmitere prin referinta)

```
bool check(std::vector<std::pair<int, int> > &solution, int walls[100][100]) {
    for (unsigned i = 0; i < solution.size() - 1; ++i) {
        /* line_prev si col_prev reprezinta celula in care se afla soricelul la
        pasul i; line_next si col_next reprezinta celula in care se afla
        la pasul i + 1; trebuie sa fim siguri ca soricelul nu a ajuns pe zid
        si ca urmatoarea celula este sub sau in dreapta celulei curente */
        int line_prev = solution[i].first;
        int line_next = solution[i + 1].first;
        int col_prev = solution[i].second;
        int col_next = solution[i + 1].second;

        /* walls[x][y] == 1 inseamna ca este zid pe linia x, coloana y */
        if (walls[line_prev][col_prev] == 1 ||
                !((line_next == line_prev + 1 && col_next == col_prev) ||
                (line_next == line_prev && col_next == col_prev + 1))) {
            return false;
        }
    }

    return true;
}

void printSolution(std::vector<std::pair<int, int> > &solution) {
    for (std::pair<int, int> s : solution) {
        std::cout << "(" << s.first << "," << s.second << ")->";
    }
    std::cout << "\n";
}

void back(std::vector<std::pair<int, int> > &domain, int walls[100][100],
        std::vector<std::pair<int, int> > &solution, int max_iter) {
    /* daca am facut "max_iter" pasi ma opresc si verific daca este corecta
    solutia */
    if (solution.size() == max_iter) {
        if(check(solution, walls)) {
            printSolution(solution);
        }
        return;
    }

    /* avand domeniul initializat cu toate celulele din matrice, incercam sa
    adaugam oricare dintre aceste celule la solutie, verificand la final daca
    solutia este buna */
    for (unsigned int i = 0; i < domain.size(); ++i) {
        /* pastram elementul curent pentru a-l readauga in domeniu dupa
        apelarea recursiva */
        std::pair<int, int> tmp = domain[i];

        /* adaugam elementul curent la solutia candidat */
        solution.push_back(domain[i]);
        /* stergem elementul curent din domeniu */
        domain.erase(domain.begin() + i);

        /* apelam recursiv backtracking */
        back(domain, walls, solution, max_iter);

        /* adaugam elementul sters din domeniu inapoi */
        domain.insert(domain.begin() + i, tmp);
        /* stergem elementul curent din solutia candidat pentru a o forma pe
        urmatoarea */
        solution.pop_back();
    }
}

int main() {
    /* initializam domeniul si solutia ca vectori de perechi de int-uri;
    domeniul va contine initial toate perechile de indici posibile din
    matrice ((0, 0), (0, 1) ... (n - 1, n - 1)), iar solutia va fi initial
    vida */
    std::vector<std::pair<int, int> > domain, solution;

    fin >> n;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            /* walls[i][j] == 1 daca pe pozitia (i, j) este zid; 0 altfel */
            fin >> walls[i][j];
            domain.push_back({i, j});
        }
    }

    /* apelam back cu domeniul format initial, cu matricea de ziduri, cu
    solutia vida si cu numarul maxim de iteratii = 2 * n - 1 pentru ca
    mergand doar in dreapta si in jos, in 2 * n - 1 pasi va ajunge din
    (0, 0) in (n - 1, n - 1) */
    back(domain, walls, solution, 2 * n - 1);
}
```

Apelarea initiala (din “int main”) se face astfel: “back(domain, walls, solution, 2 * n - 1);”, unde domain reprezinta un vector cu perechi in care sunt toate celulele matricii, solution este un vector gol, walls este matricea care ne arata daca este zid sau nu pe o anumita pozitie, iar 2 * n - 1 e numarul e pasi in care ar trebui ca soricelul sa ajunga la branza pe oriunde ar merge.

Complexitate

Solutia va avea urmatoarele complexitati:

- complexitate temporala : $T(n) = O(Aranjamente(n^2, 2n-1))$
  - explicatie: Initial in domeniu avem n^2 valori. Noi dorim sa generam toate submultimile ordonate de cate $2n-1$ elemente. Acestea sunt tocmai aranjamentele de n^2 luate cate $2n-1$.
- complexitate spatiala : $S(n) = O(n^2)$
  - explicatie: Trebuie sa stocam informatie despre drum, care are $2n-1$ celule; stocam domeniul care are n^2 elemente

Backtracking (taierea ramurilor nefolositoare)

```
bool check(std::vector<std::pair<int, int> > &solution) {
    return true;
}

void printSolution(std::vector<std::pair<int, int> > &solution) {
    for (std::pair<int, int> s : solution) {
        std::cout << "(" << s.first << "," << s.second << ")->";
    }
    std::cout << "\n";
}

void back(int step, int stop, int walls[100][100],
        std::vector<std::pair<int, int> > &solution, int line_moves[2],
        int col_moves[2]) {
    /* ne oprim dupa ce am ajuns la pasul "stop" si verificam daca solutia este
    corecta */
    if (step == stop) {
        /* deoarece am eliminat ramurile nefolositoare am ajuns la o solutie care
        sigur este corecta */
        if(check(solution)) {
            printSolution(solution);
        }
        return;
    }

    /* daca este primul pas stiu ca soricelul este in pozitia (0, 0) */
    if (step == 0) {
        /* adaugam (0, 0) la solutia candidat */
        solution.push_back({0, 0});

        /* apelam backtracking recursiv la pasul urmator */
        back(step + 1, stop, walls, solution, line_moves, col_moves);

        /* scoatem (0, 0) din solutie */
        solution.pop_back();
        return;
    }

    /* sunt doar doua mutari pe care le pot face intr-un pas: dreapta si jos;
    acestea sunt encodate prin vectorii de directii line_moves[2] = {0, 1} si
    col_moves[2] = {1, 0} care reprezinta la indicele 0 miscarea in dreapta, iar
    la indicele 1 miscarea in jos */
    for (unsigned int i = 0; i < 2; ++i) {
        /* cream noua linie si noua coloana cu ajutorul vectorilor de directii */
        int new_line = solution.back().first + line_moves[i];
        int new_col = solution.back().second + col_moves[i];
        int n = (stop + 1) / 2;

        /* daca linia si coloana sunt valide (nu ies din matrice) si nu este
        zid pe pozitia lor, putem continua pe acea celula */
        if (new_line < n && new_col < n && walls[new_line][new_col] == 0) {
            /* adaugam noua celula in solutia candidat;
            NOTE: {new_line, new_col} este echivalent cu
            std::pair<int, int>(new_line, new_col) si se numeste "initializer
            list", feature in C++11 */
            solution.push_back({new_line, new_col});

            /* apelam backtracking recursiv la pasul urmator */
            back(step + 1, stop, walls, solution, line_moves, col_moves);

            /* scoatem celula adaugata din solutie */
            solution.pop_back();
        }
    }
}

int main() {
    /* initializam solutia ca vector de perechi de int-uri */
    std::vector<std::pair<int, int> > solution;

    fin >> n;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            /* citim matricea zidurilor; 1 pentru zid, 0 altfel */
            fin >> walls[i][j];
        }
    }

    /* apelam back cu step = 0, stop = 2 * n - 1 deoarece in 2 * n - 1
    pasi soricelul va ajunge la branza, vectorul de ziduri, vectorul in
    care vom stoca solutia, vectorii de directii line_moves[2] = {0, 1} si
    col_moves[2] = {1, 0}; nu avem nevoie de domeniu deoarece folosind
    vectorii de directii vom sti din ultima pozitie pusa in solutie cele
    doua solutii in care putem merge, astfel domeniul nostru va fi alcatuit
    din doua solutii la fiecare pas (daca ultima pozitie din solutie a fost
    (5, 7) => domeniul pasului curent = {(5 + 0, 7 + 1) si (5 + 1, 7 + 0)}
    care este egal cu {(5, 8), (6, 7)}. */
    back(0, 2 * n - 1, walls, solution, line_moves, col_moves);
}
```

Pentru aceasta abordare la fiecare pas de backtracking vom merge doar in doua directii, in loc sa mergem in oricare celula din matrice, lucru care imbunatateste semnificativ complexitatea temporala.

Complexitate

Solutia va avea urmatoarele complexitati:

- complexitate temporala : $T(n) = O(2^{2n})$
  - explicatie: avem de urmat un sir de $2n - 1$ mutari, iar la fiecare pas avem 2 variante posibile
- complexitate spatiala : $S(n) = O(n)$
  - explicatie: stocam maximum $2n - 1$ casute

Exercitii

In acest laborator vom folosi scheletul de laborator din arhiva skel-lab05.zip.

Aranjamente

Fie N si K doua **numere naturale strict pozitive**. Se cere afisarea tuturor aranjamentelor de N elemente luate cate K din multimea {1, 2, …, N}.

Fie N = 3, K = 2 ⇒ M = {1, 2, 3}

Solutie:

- {1, 2}
- {1, 3}
- {2, 1}
- {2, 3}
- {3, 1}
- {3, 2}

Se doreste o complexitate $T(n, k) = A(n, k)$.

Folositi-va de problema **Permutari**.

Solutiile se vor genera in ordine lexico-grafica!

Checkerul asteapta sa le stocati in aceasta ordine.

Submultimi

Fie N un **numar natural strict pozitiv**. Se cere afisarea tuturor submultimilor multimii {1, 2, …, N}.

Fie N = 4 ⇒ M = {1, 2, 3, 4}

Solutie: {} - multimea vida {1} {1, 2} {1, 2, 3} {1, 2, 3, 4} {1, 2, 4} {1, 3} {1, 3, 4} {1, 4} {2} {2, 3} {2, 3, 4} {2, 4} {3} {3, 4} {4}

Se doreste o complexitate $T(n) = O(2^n)$.

Folositi-va de problema **Combinari**.

Solutiile se vor genera in ordine lexico-grafica!

Checkerul asteapta sa le stocati in aceasta ordine.

Problema damelor

Problema damelor (sau problema reginelor) trateaza plasarea a 8 regine de sah pe o tablă de șah de dimensiuni 8 x 8 astfel incat sa nu existe doua regine care se ameninta reciproc. Astfel, se cauta **o solutie** astfel incat nicio pereche de doua regine sa nu fie pe acelasi rand, pe aceeasi coloana, sau pe aceeasi diagonala. Problema cu opt regine este doar un caz particular pentru problema generala, care presupune plasarea a N regine pe o tablă de sah N x N în aceleasi conditii. Pentru aceasta problema, există solutii pentru toate numerele naturale N cu excepția lui N = 2 si N = 3.

Fie N = 5

Solutie:

X	-	-	-	-
-	-	X	-	-
-	-	-	-	X
-	X	-	-	-
-	-	-	X	-

X reprezinta o dama, - reprezinta spatiu gol.

E nevoie sa facem backtracking pe matrice sau e suficient pe vector?

Se va cauta o singura solutie (**oricare** solutie corecta), care va fi returnata sub format unui vector cu $n + 1$ elemente.

Solutia este $sol[0], sol[1], \ldots, sol[n]$, unde $sol[i]$ = coloana unde vom plasa regina de pe linia i.

Elementul 0 este nefolosit, dorim sa pastram conventia cu indexare de la 1.

Generare de siruri

Vi se da o lista de caractere si o lista de frecvente (pentru caracterul de pe pozitia i, frecventa de pe pozitia i). Vi se cere sa generati toate sirurile care se pot forma cu aceste caractere si aceste frecvente stiind ca nu pot fi mai mult de K aparitii consecutive ale aceluiasi caracter.

Fie caractere[] = {'a', 'b', 'c'}, freq[] = {1, 1, 2}, K = 5

Solutie:

- abcc
- acbc
- accb
- bacc
- bcac
- bcca
- cabc
- cacb
- cbac
- cbca
- ccab
- ccba

Fie caractere[] = {'b', 'c'}, freq[] = {3, 2}, K = 2

Solutie:

- bbcbc
- bbccb
- bcbbc
- bcbcb
- cbbcb
- cbcbb

Solutiile se vor genera in ordine lexico-grafica!

Checkerul asteapta sa le stocati in aceasta ordine.

Bonus

Problema damelor (AC3)

Aplicati AC3 pe problema damelor.

Algoritmul AC-3 (Arc Consistency Algorithm) este de obicei folosit in probleme de satisfacere a constrangerilor (CSP). Acesta sterge arcele din arborele de stari care sigur nu se vor folosi niciodata.

AC-3 lucreaza cu:

- constrangeri
- variabile
- domenii de variabile

O variabila poate lua orice valoare din domeniul sau la orice pas. O constrangere este o relatie sau o limitare a unor variabile.

Exemplu AC-3

Consideram A, o variabila ce are domeniul D(A) = {0, 1, 2, 3, 4, 5, 6} si B o variabila ce are domeniul D(B) = {0, 1, 2, 3, 4}. Cunoastem constrangerile: C1 = “A trebuie sa fie impar” si C2 = “A + B trebuie sa fie egal cu 5”.

Algoritmul AC-3 va elimina in primul rand toate valorile pare ale lui A pentru a respecta C1 ⇒ D(A) = {1, 3, 5}. Apoi, va incerca sa satisfaca C2, asa ca va pastra in domeniul lui B toate valorile care adunate cu valori din D(A) pot da 5 ⇒ D(B) = {0, 2, 4}.

AC-3 a redus astfel domeniile lui A si B, reducand semnificativ timpul folosit de algoritmul backtracking.

Extra

Immortal

Enunt [https://www.infoarena.ro/problema/immortal]

OJI 2010 - clasele 11-12 [http://olimpiada.info/oji2010/index.php?cid=arhiva]

Ultimate Tic Tac Toe

The AI Games - Ultimate Tic Tac Toe [http://theaigames.com/competitions/ultimate-tic-tac-toe]