Backtracking

Um algoritmo que utiliza a técnica de backtracking utiliza também recursividade. O backtracking não passa de um refinamento de força bruta, em que a solução é construída recursivamente.

O backtracking constrói uma solução parcial verificando restrições. É possível que ações sejam desfeitas por quebra dessas restrições.

O algoritmo global do backtracking é:

backtracking (c)

se (rejeita(P, c)) então

retorne

fim se

se (aceita(P, c)) então

saída(P,c)

fim se

s ← primeiro(P, c)

enquanto (s ≠ A) faça

backtracking(s)

s ← próximo(P, s)

fim enquanto

fim backtracking

Saída do labirinto 1

Matriz de células.

Célula (0, 0) é a entrada.

Célula (linha-1, coluna-1) é a saída.

Há duas direções:

(L + 1, c) - abaixo

(L, c + 1) - direita

Não pode ultrapassar limites.

Utiliza recursividade para armazenar o caminho.


```
#include <iostream>

bool labirinto_1(int labirinto[20][20], int linha, int coluna
    if (l < 0 or c < 0 or l >= linha or c>=coluna or labirint
        return false;
    if (l == linha-1 and c == coluna-1)
        return true;
    if (labirinto_1(labirinto, linha, coluna, l+1, c)) //cami
        return true;
    if (labirinto_1(labirinto, linha, coluna, l, c+1)) //cami
        return true;
    return false; // nenhuma das opções chegou ao final
}
```

Saída do labirinto 2

Matriz de células.

- Célula (0, 0) é a entrada.
- Célula (linha-1, coluna-1) é a saída.

Há quatro direções:

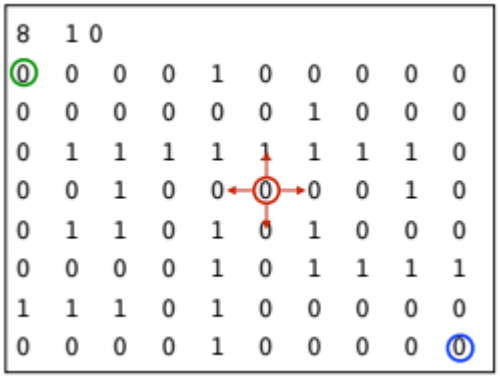
- (L - 1, c) - acima
- (L + 1, c) - abaixo

(L, c + 1) - direita

(L, c - 1) - esquerda

Não pode ultrapassar limites.

Utiliza recursividade para armazenar o caminho.




```
bool labirinto_bt(int labirinto[20][20], int linha, int colun
    if (l==-1 || c==-1 || l==linha || c==coluna || labirinto[
        return false;
    if (l==linha-1 && c==coluna-1)
        return true;

    labirinto[l][c] = 9;
    bool ans = false ;

    ans = labirinto_bt(labirinto, linha, coluna, l+1, c);
    if (ans == false)
        ans = labirinto_bt(labirinto, linha, coluna, l, c+1);
    if (ans == false)
        ans = labirinto_bt(labirinto, linha, coluna, l, c-1);
    if (ans == false)
        ans = labirinto_bt(labirinto, linha, coluna, l-1, c);

    labirinto[l][c] = 0;
    return ans;
}
```

Sub-lista de soma S

Dado um array de n elementos inteiros não negativos, existe algum subconjunto cuja soma seja S?

Exemplo: array: {8, 35, 6, 21, 12}, s: 41

Complexidade: 2^n. Para cada solução, a soma, no pior caso, tem complexidade n.

Observações:

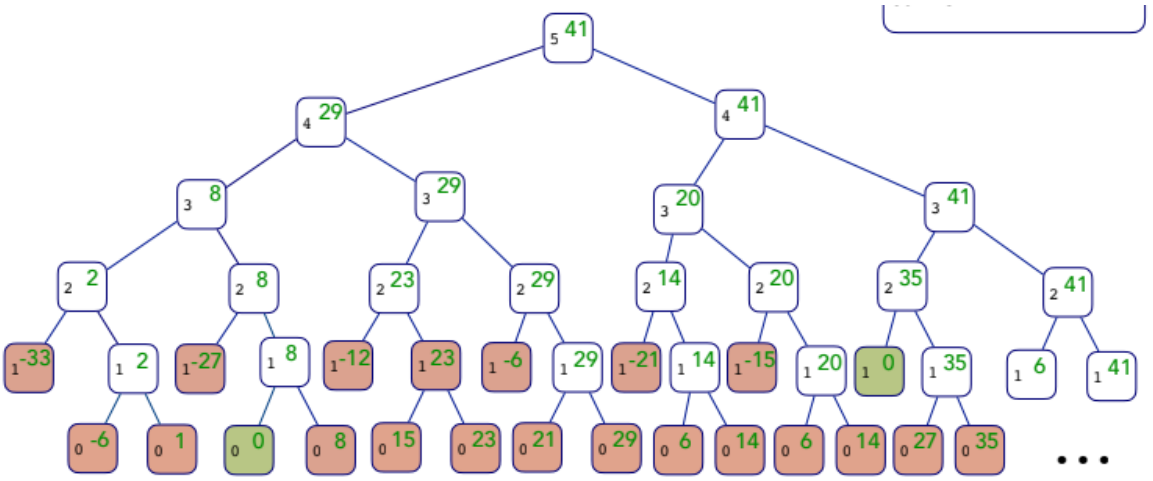
- 2 chamadas recursivas
  - Uma com o número “atual”
  - Outra sem o número “atual”
- Cada número está ou não está na solução

Solução: array: {8, 35, 6, 21, 12}, s: 41

- Se a solução contém o 12
  - Resolver problema: {8, 35, 6, 21}, s: 29 (41-12)
    - Se a solução contém o 21
      - Resolver problema: {8, 35, 6}, s: 8
        - Se a solução contém 6
          - Resolver problema: {8, 35}, s: 2
            - Se a solução contém 35
              - s: -33
            - Se a solução não contém 35
              - Resolver problema: {8}, s: 2
              - Se a solução contém 8
                - s: -6
              - Se a solução não contém 8
                - s: 2
        - Se a solução não contém 6
          - Resolver problema: {8, 35}, s: 8
            - Se a solução contém 35

        - s: -27
    - Se a solução não contém 35
        - Resolver problema: {8}, s: 8
            - Se a solução contém 8
                - s: 0
            - Se a solução não contém 8
                - s: 8

...



Respostas: {8, 21, 12} e {35, 6}

Código:

```
#include <bits/stdc++.h>

using namespace std;

int soma_sublista_bt(int a[], int tam, int soma){
    int r;
    if (soma == 0) return true;
    if (soma < 0 || tam == 0) return false;
    r = soma_sublista_bt(a, tam-1, soma - a[tam-1]);
    r = r || soma_sublista_bt(a, tam-1, soma);
    return r;
```

```
}

int main(){
    int a[] = {8, 35, 6, 21, 12};
    int soma = 41;
    int tam = 5;
    cout << soma_sublista_bt(a, tam, soma) << endl;
}
```

Problema da mochila

Há n itens em uma mochila, todos os itens possuem certo peso e preço. A mochila possui capacidade para x kg. Quais itens devemos colocar na mochila para **maximizar** o preço carregado?

Exemplo:

> Capacidade da mochila: 10kg
>
> 4 itens: { (7kg, 42r), (3kg, 12r), (5kg, 25r), (4kg, 40r)}

Será necessário fazer todos as combinações possíveis entre os itens, o que torna a complexidade do algoritmo O(2^n).

```
#include <iostream>

using namespace std;

struct item{
    int peso;
    double valor;
};

double mochila(vector<item> itens, int q, int peso){
    int ans = 0;
    //TO DO
    return ans;
}

int main(){
```

```
    int n, peso_max;
    cin >> n >> peso_max;
    vector<item> itens(n);
    for(int i = 0; i < n; i++)
        cin >> itens[i].peso >> itens[i].valor;
        double valor = mochila(itens, n, peso_max);
}
```