maratona de programação

UNIFEI

4 – **Paradigmas I**

Busca Completa & Backtracking

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação
sites.google.com/site/unifeimaratona/

Paradigmas para Solução de Problemas

Nesta aula e nas aulas seguintes, conheceremos quatro abordagens diferente para o ataque a problemas de competição:

- Busca Completa (também conhecido como “força bruta”)
- Divisão e Conquista
- Algoritmos Gulosos
- Programação dinâmica.

Todo programador deve conhecer, pelo menos, as quatro ferramentas básicas. Atacar um problema sempre com força bruta não será suficiente para um bom rendimento em competições.

Paradigmas para Solução de Problemas

Imagine que é necessário realizar quatro tarefas em um vetor A contendo n <= 10K elementos:

1. Encontrar o maior e o menor elemento de A.
2. Encontrar o k-ésimo menor elemento de A.
3. Encontrar o maior valor g, tal que x e y pertençam ao vetor A e g = |x-y|
4. Encontrar a maior subsequência crescente do vetor A.

Paradigmas para Solução de Problemas

1. Encontrar o maior e o menor elemento de A.

A solução para este problema é a **busca completa**, percorrendo todos os valores de A em busca do maior e menor elemento. Executa em O(n).

Paradigmas para Solução de Problemas

2. Encontrar o k-ésimo menor elemento de A.

Para este problema, a solução de busca completa necessitaria percorrer por k vezes o vetor A. A solução funciona, mas se tomarmos k como a média (k=n/2), temos que o algoritmo executa em O(n/2*n) que é $O(n^2)$. Qual outra solução seria interessante?

Isso mesmo! Ordene o vetor, utilizando um método como *Quick* ou *Merge Sort* e devolva o k-ésimo elemento. Executa em apenas O(n log n)! **Divisão e Conquista**.

Paradigmas para Solução de Problemas

3. Encontrar o maior valor g, tal que x e y pertençam ao vetor A e g = |x-y|

A busca completa faria a diferença entre todos os pares x e y do vetor A, resultando em $O(n^2)$. Como poderíamos encontrar mais facilmente?

Exatamente! Encontramos o maior e menor valor do vetor e fazemos apenas a diferença entre eles. Isso pode ser realizado em O(n), e é uma estratégia de **algoritmo guloso**.

Paradigmas para Solução de Problemas

4. Encontrar a maior subsequência crescente do vetor A.

Tentar todas as possibilidades de subsequência seria um trabalho de força bruta da ordem de $O(2^n)$, o que o torna impraticável. Discutiremos, em outras aula, uma abordagem que resolve este problema em $O(n^2)$, utilizando **Programação Dinâmica**.

Paradigmas para Solução de Problemas

Nas próximas aulas, tente não apenas memorizar a solução dos problemas dados. O ideal é assimilar o processo de construção da resposta a partir de cada paradigma.

Aprenda a pensar diferente!

Busca Completa

(a.k.a Força Bruta)

Método de solução de problemas baseado no percurso de todo (ou grande parte) do espaço de busca visando encontrar uma solução.

> Durante a busca, é possível realizar “podas”, ou escolhas, de maneira que deixamos de procurar pela solução em uma parte do espaço de busca que temos certeza de ser infrutífero.

É, normalmente, a primeira solução a ser pensada, mas somente deve ser utilizada em dois casos:

1. Não há outro algoritmo disponível para a realização daquela tarefa
   - Por exemplo, a tarefa de enumerar TODAS as permutações de {0, 1, 2, ..., N-1} claramente requer O(N!) operações. Não há como fugir.

2. Ou há outro método disponível, mas não há necessidade, uma vez que a entrada é suficientemente pequena.
   - As range queries da aula passada em um vetor de tamanho <= 100 podem ser feitas também por uma busca completa no vetor.
   - Lembre-se: Sempre desejamos a solução mais simples!

Exercício: Vito's Family

- Resolva o exercício A, disponível no BOCA em
  boca.unifei.edu.br/src

"Tell me, do you spend time with your **family**?" (Dom Vito Corleone)

Exercício: The Skyline Problem

- Resolva o exercício B, disponível no BOCA em
  boca.unifei.edu.br/src

Busca Completa

Mas, e se o espaço de busca for um pouco mais complexo?

- Todas as **permutações** de n itens.
- Todos os **subconjuntos** de n itens.
- Todas as maneiras de se colocar N rainhas em um tabuleiro de xadrez de NxN sem que nenhuma delas ataque outra.

Como fazemos para iterar através do espaço de busca?

Iterando pelas permutações

Há uma função implementada em C++ STL, em <algorithm>, chamada **next_permutation**.

Lembre-se de que existem n! permutações de tamanho n e, portanto, você só deve percorrê-las completamente para n <= 11.

- n! = 39.916.800 de permutações.
- Caso contrário, encontraremos soluções melhores.

Veja o código e seu resultado:

```cpp
#include <cstdio>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    int n = 5, i = 0;
    vector<int> perm(n);

    for (int i = 0; i < n; i++)
        perm[i] = i + 1;

    do {
        printf("#%d: ", ++i);
        for (int i = 0; i < n; i++)
            printf("%d ", perm[i]);
        printf("\n");
    } while (next_permutation(perm.begin(), perm.end()));

    return 0;
}
```

```
D:\Disciplinas\ECO Maratc
ests>a.exe
#1: 1 2 3 4 5
#2: 1 2 3 5 4
#3: 1 2 4 3 5
#4: 1 2 4 5 3
#5: 1 2 5 3 4
#6: 1 2 5 4 3
#7: 1 3 2 4 5
#8: 1 3 2 5 4
#9: 1 3 4 2 5
#10: 1 3 4 5 2
#11: 1 3 5 2 4
#12: 1 3 5 4 2
#13: 1 4 2 3 5
#14: 1 4 2 5 3
#15: 1 4 3 2 5
#16: 1 4 3 5 2
#17: 1 4 5 2 3
#18: 1 4 5 3 2
#19: 1 5 2 3 4
#20: 1 5 2 4 3
#21: 1 5 3 2 4
#22: 1 5 3 4 2
#23: 1 5 4 2 3
#24: 1 5 4 3 2
#25: 2 1 3 4 5
#26: 2 1 3 5 4
#27: 2 1 4 3 5
#28: 2 1 4 5 3
```

Iterando pelos subconjuntos

Lembra-se da representação binária dos subconjuntos? Cada inteiro entre 0 e 2^n-1 representa um subconjunto diferente do conjunto {1,2, ..., n}. Portanto, Apenas realiza a iteração pelos inteiros.

Lembre-se de que existe 2^n subconjuntos para um conjunto de n elementos. Portanto, somente procure por todas elas se n <= 25.

- 2^{25} = 33.554.432
- Caso contrário, encontraremos soluções melhores.

```cpp
#include <cstdio>

int main()
{
    int n = 5;

    printf("Number of subsets: %d\n", 1 << n);
    for (int subset = 0; subset < (1 << n); subset++)
    {
        for (int i = 0; i < n; i++)
            if ((subset & (1 << i)) != 0)
                printf("%d ", i+1);
        printf("\n");
    }

    return 0;
}
```

```
D:\Disciplinas\ECO Marator
ests>a.exe
Number of subsets: 32

1
2
1 2
3
1 3
2 3
1 2 3
4
1 4
2 4
1 2 4
3 4
1 3 4
2 3 4
1 2 3 4
5
1 5
2 5
1 2 5
3 5
1 3 5
2 3 5
1 2 3 5
4 5
1 4 5
2 4 5
1 2 4 5
3 4 5
1 3 4 5
2 3 4 5
1 2 3 4 5
```

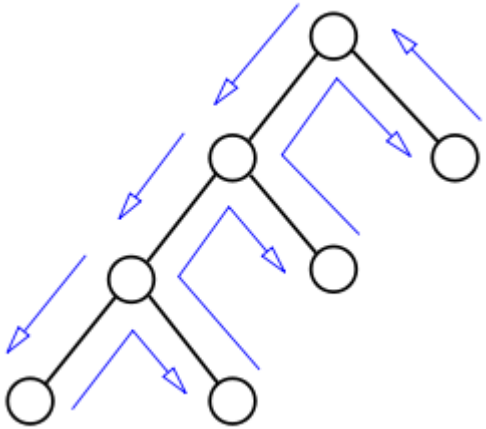
Exercício: Social Constraints

- Resolva o exercício C, disponível no BOCA em
  boca.unifei.edu.br/src

Backtracking

Ou Busca Completa Recursiva

Backtracking é um tipo de algoritmo que representa um refinamento da busca por força bruta, em que múltiplas soluções podem ser eliminadas sem serem explicitamente examinadas. O termo foi cunhado pelo matemático estadunidense D. H. Lehmer na década de 1950.



Backtracking

Ou Busca Completa Recursiva

Uma busca inicial em um programa com backtracking segue o padrão busca em profundidade, ou seja, a árvore é percorrida sistematicamente de cima para baixo e da esquerda para direita. Quando essa pesquisa falha, ou é encontrado um nodo terminal da árvore, entra em funcionamento o mecanismo de backtracking. Esse procedimento faz com que o sistema **retorne pelo mesmo caminho percorrido** com a finalidade de encontrar soluções alternativas.

Backtracking

Ou Busca Completa Recursiva

Ideia Básica:

- Iniciar a busca por um estado vazio;
- Utilizar recursão para percorrer todos os estados, através de transições definidas de um estado para seus próximos estados;
- Caso o estado atual seja inválido, pare de explorar este caminho (poda);
- Processe todos os estados completos (que são, exatamente, os estados que nos interessam)

```
state S;

void generate() {
    if (!is_valid(S))
        return;

    if (is_complete(S))
        print(S);

    foreach (possible next move P) {
        apply move P;
        generate();
        undo move P;
    }
}

S = empty state;
generate();
```

```
const int n = 5;
bool pick[n];

void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
            if (pick[i]) {
                printf("%d ", i+1);
            }
        }
        printf("\n");
    }
    else
    {
        // either pick element no. at
        pick[at] = true;
        generate(at + 1);
        // or don't pick element no. at
        pick[at] = false;
        generate(at + 1);
    }
}

int main() {
    generate(0);
    return 0;
}
```

Backtracking – Gerando Subconjuntos

Continua havendo 2^n combinações, Mas estamos pensando de modo diferente.

Neste caso, não há “poda”, ou estados inválidos.

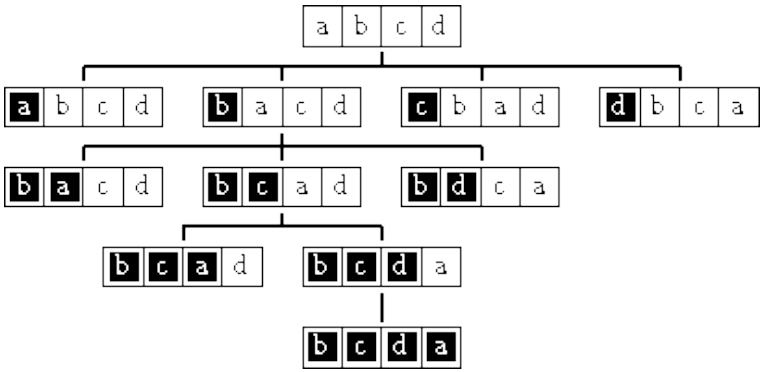
Imagine que queiramos todos os conjuntos cuja soma não exceda um valor V. Poderíamos aplicar a poda no início do código.

```
const int n = 5;
int perm[n];
bool used[n];

void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
        printf("%d ", perm[i]+1);
        }
        printf("\n");
    }
    else
    {
        // decide what the at-th element should be
        for (int i = 0; i < n; i++) {
            if (!used[i]) {
                used[i] = true;
                perm[at] = i;
                generate(at + 1);
                // remember to undo the move:
                used[i] = false;
            }
        }
    }
}

int main()
{
    memset(used, 0, n);
    generate(0);
    return 0;
}
```

Backtracking – Gerando Permutações



Também continua gerando todas as N! permutações, mas de maneira diferente.

Backtracking – Problema das N Rainhas

Dadas N rainhas em um tabuleiro de xadrez de NxN casas, encontre todas as maneiras com que se consegue colocar N rainhas no tabuleiro sem que nenhuma delas ataque uma à outra.

- É um conjunto bastante específico que teremos que percorrer e, portanto, não terá soluções prontas.

- Vamos utilizar backtracking!

Backtracking – Problema das N Rainhas

Algoritmo:

1. Percorra todas as células do tabuleiro, em ordem crescente.
2. Em cada célula, coloque ou não uma rainha (transição)
3. Não coloque uma rainha em uma célula, caso ela seja capaz de atacar outra rainha qualquer que já esteja posicionada.

Exercício: 8-Queen Chess Problem

- Resolva o exercício D, disponível no BOCA em
  boca.unifei.edu.br/src