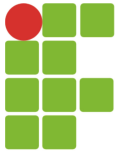
Paradigmas de Projeto de Algoritmos - Busca Completa

Tópicos Especiais em Algoritmos - Ciência da Computação

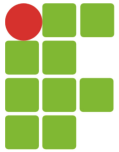
Prof. Daniel Saad Nogueira Nunes

IFB – Instituto Federal de Brasília,
Campus Taguatinga



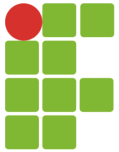
Sumário

1. Introdução
2. Problemas
3. Considerações



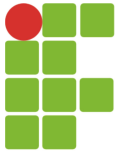
Sumário

1 Introdução



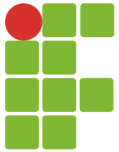
Introdução

- O paradigma de projeto de algoritmos por busca completa visa exaurir todo o espaço de busca (ou quase todo) para encontrar a melhor resposta possível.
- Uma solução baseada em busca completa nunca deveria estar incorreta, pois todas as possibilidades de solução são checadas.
- Outro nome adotado para um algoritmo baseado em busca completa é algoritmo força-bruta, mas não usaremos esta terminologia pois várias das soluções empregadas podem utilizar recursos de poda para ter que evitar percorrer todo o espaço de soluções sem abrir mão da correção do algoritmo.



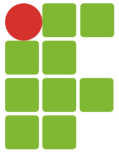
Introdução

- Quando o espaço de busca não é excessivo, podemos utilizar este paradigma.
- Verificaremos, através de situações problema, como podemos elaborar soluções baseadas em busca completa.



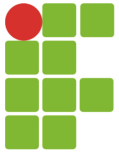
Sumário

2 Problemas



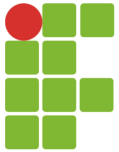
Problemas

- Examinaremos e discutiremos agora uma série de problemas em que o paradigma de busca completa pode ser aplicado.



Sumário

2 Problemas
- UVa 725
- UVa 441
- UVa 11565
- UVa 11742
- UVa 12455
- UVa 750



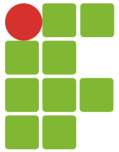
UVa 725

Descrição do Problema

Dado um inteiro N, encontrar todos os inteiros $A = abcde$ e $B = fghij$, de cinco dígitos cada, de forma que

$$\frac{abcde}{fghij} = N$$

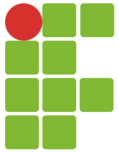
- **Restrição**: os inteiros A e B devem considerar os 10 possíveis dígitos.
- Observação: é permitido que um número comece com o dígito 0.



UVa 725

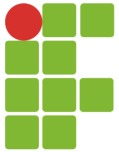
Exemplo

- Para $N = 62$ temos como possíveis respostas $A = 79546$ e $B = 01283$ ou $A = 94736$ e $B = 01528$.
- Para $N = 61$ não existem inteiros A e B que satisfaçam as restrições do problema.



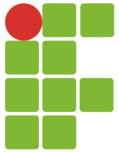
UVa 725

- Tanto A quanto B estão no espaço de busca $[01234, 98765]$.
- Como A pode ser descrito em função de B $(A = B \cdot N)$ o espaço de busca é menor ainda $\approx 10^5$.
- Estratégia, para cada B no intervalo $[01234, 98765]$, computar A e verificar se todos os dígitos foram utilizados.



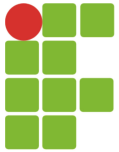
UVa 725: Solução

```
for (int fghij = 1234; fghij <= 98765 / N; fghij++) {
    int abcde = fghij * N; // Calcula abcde de fghij
    int tmp, used = (fghij < 10000);
    tmp = abcde;
    while (tmp) {
        used |= 1 << (tmp % 10);
        tmp /= 10;
    }
    tmp = fghij;
    while (tmp) {
        used |= 1 << (tmp % 10);
        tmp /= 10;
    }
    if (used == (1<<10) - 1){
        printf("%0.5d / %0.5d = %d\n", abcde, fghij, N);
    }
}
```



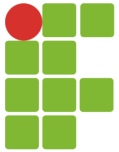
UVa 725: Solução

- Utilizamos um vetor de bits para verificar se todos os dígitos foram utilizados.
- Para cada dígito de valor i de A ou B, ligamos o i-ésimo bit deste vetor de bits.
- Caso o vetor de bits seja $V = 1111111111 = 2^{10} - 1$, sabemos que utilizamos todos os 10 dígitos.



Sumário

2 Problemas
- UVa 725
- UVa 441
- UVa 11565
- UVa 11742
- UVa 12455
- UVa 750

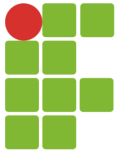


UVa 441

Descrição do Problema

Dado um inteiro k e um conjunto de k inteiros $S = s_0, s_1, \ldots, s_{k-1}$ inteiros em ordem estritamente crescente, imprimir todos os subconjuntos de S de tamanho 6 que podem ser gerados a partir de S.

- Restrição do problema: $6 \leq k \leq 12$.

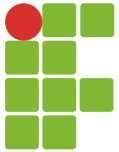


UVa 441

- Como $|S| \leq 12$, temos no máximo a seguinte quantidade de subconjuntos de tamanho 6:

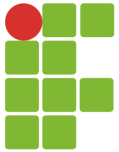
$$\binom{12}{6} = 924$$

- Com 924 conjuntos, é possível adotar uma abordagem de busca completa.



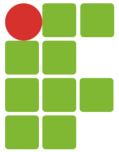
UVa 441: Solução

```
1  for (int i = 0; i < k; i++)
2      scanf("%d", &v[i]);
3  for (int a = 0; a < k - 5; a++)
4      for (int b = a + 1; b < k - 4; b++)
5          for (int c = b + 1; c < k - 3; c++)
6              for (int d = c + 1; d < k - 2; d++)
7                  for (int e = d + 1; e < k - 1; e++)
8                      for (int f = e + 1; f < k; f++)
9                          printf("%d %d %d %d %d %d\n",v[a],v[b],v[c],v[d],v[e],v[f]);
```



Sumário

2 Problemas
- UVa 725
- UVa 441
- UVa 11565
- UVa 11742
- UVa 12455
- UVa 750



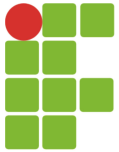
UVa 11565

Descrição do Problema

Dado três inteiros A, B e C, encontrar valores x, y e z distintos que satisfaçam o seguinte sistema linear de equações:

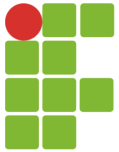
$$\begin{cases} x + y + z = A \\ xyz = B \\ x^2 + y^2 + z^2 = C \end{cases}$$

- Restrições do problema: $1 \leq A, B, C \leq 10^4$.
- No caso de múltiplas soluções (x, y, z), deve ser escolhida aquela com menor valor de x.



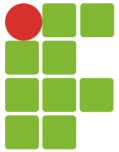
UVa 11565

- Podemos refinar a busca ao examinar as propriedades da equação.
- De acordo com a segunda equação do sistema linear, temos que $x^2 + y^2 + z^2 = B$, como $B \leq 10^4$ é fácil ver que as três variáveis tem que estar em um intervalo $[-100, 100]$.
- Como são 3 variáveis, temos um espaço de busca de tamanho $201^3 = 8120601 < 10^7$.
- Viável utilizar a busca completa.



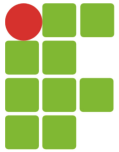
UVa 11565: Solução

```
bool sol = false;
int x, y, z;
for (x = -100; x <= 100 && !sol; x++)
    for (y = -100; y <= 100 && !sol; y++)
        for (z = -100; z <= 100 && !sol; z++)
            // x,y e z devem ser distintos
            if (y != x && z != x && z != y &&
                x + y + z == A && x * y * z == B &&
                x * x + y * y + z * z == C) {
                    if (!sol)
                        printf("%d %d %d\n", x, y, z);
                    sol = true;
            }
```



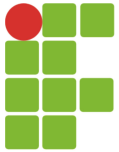
UVa 11565: Refinamentos

- É possível fazer ainda mais melhorias.
- Examinando mais cuidadosamente a primeira equação, $xyz = A$, e tomando $x = y = z$, temos que $x \leq \sqrt[3]{10^4}$. Portanto, sabemos que x deve estar no intervalo $[-22, 22]$.
- É possível ainda fazer mais refinamentos (substituição de variáveis, por exemplo). Esses refinamentos mais avançados são imprescindíveis para resolver a versão mais difícil do problema (UVa 11571).



Sumário

2 Problemas
- UVa 725
- UVa 441
- UVa 11565
- UVa 11742
- UVa 12455
- UVa 750



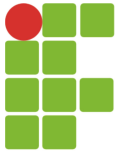
UVa 11742

Descrição do Problema

Supondo um número n de pessoas, numeradas de 1 a n, que querem ver um filme e m o número de restrições, calcular o número de possibilidades para que as n pessoas vejam o filme obedecendo às restrições. As n pessoas devem estar dispostas na mesma fila do cinema.

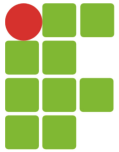
Cada restrição envolve duas pessoas e a distância mínima (ou máxima) em assentos em que essas pessoas devem estar separadas.

- Restrições da entrada: $1 \leq n \leq 8$ e $0 \leq m \leq 20$.



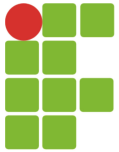
UVa 11742

- Gerar todas as permutações leva tempo $\Theta(n!)$.
- É possível, dada uma permutação, verificar se ela atende as restrições em tempo $\Theta(n+m)$.
- Complexidade total $\Theta(n! \cdot (n+m))$.
- Apesar de a complexidade de pior caso ser uma complexidade alto, temos $n \leq 8$ e $m \leq 20$.
- É viável utilizar a busca completa.
- Para gerar as permutações utilizando a STL: `std::next_permutation`.



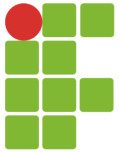
UVa 11742: Solução

```
1
2   vector<int> perm;
3   for(int i=0;i<n;i++){
4       perm.push_back(i);
5   }
6   // Gera todas as permutações
7   do{
8       // Verifica se a permutação é consistente com as restrições
9       if(check(perm,restricoes)){
10          // Se for o caso, incremente o número de possibilidades
11          count++;
12      }
13  }while(next_permutation(perm.begin(),perm.end()));
14
```



Sumário

2 Problemas
- UVa 725
- UVa 441
- UVa 11565
- UVa 11742
- UVa 12455
- UVa 750



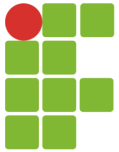
UVa 12455

Descrição do Problema

Dado um conjunto de inteiros $S = s_0, s_1, \ldots, s_{n-1}$ de n elementos e um inteiro x, deseja-se saber se existe um subconjunto $S' \subseteq S$, tal que:

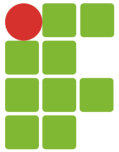
$$\sum_{s \in S'} s = x$$

- Restrições do problema: $1 \leq n \leq 20$, $0 \leq x \leq 10^4$.



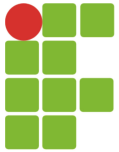
UVa 12455

- Este problema clássico é conhecido como `Subset-sum`.
- Ele pode ser resolvido usando a técnica de *programação dinâmica*.
- Contudo, como $n \leq 20$, podemos usar uma abordagem de busca completa sem se preocupar com o tempo.



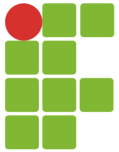
UVa 12455: Solução

- Este problema clássico é conhecido como `Subset-sum`.
- Ele pode ser resolvido usando a técnica de *programação dinâmica*.
- Contudo, como $n \leq 20$, podemos usar uma abordagem de busca completa sem se preocupar com o tempo.
- Geramos todos os subconjuntos e verificamos se a soma do subconjunto é x.
- Complexidade de tempo $\Theta(n2^n)$.



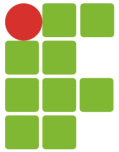
UVa 12455: Solução

```
bool res = false;
for (i = 0; i < (1 << n); i++) {
    int sum = 0;
    for (int j = 0; j < n; j++){
        if (i & (1 << j)){
            sum += S[j];
        }
    }
    if (sum == x){
        ans = true;
    }
}
```



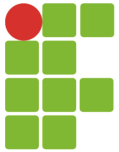
UVa 12455: Solução

- Nesta abordagem, estamos utilizando um vetor de bits para representar o subconjunto que foi escolhido.
- Este vetor de bits possui n bits e o i-ésimo bit foi escolhido caso ele pertença ao subconjunto escolhido.
- Uma vez que o subconjunto S' esteja construído, iteramos sobre o vetor de bits e verificamos se S' soma x.



Sumário

2 Problemas
- UVa 725
- UVa 441
- UVa 11565
- UVa 11742
- UVa 12455
- UVa 750

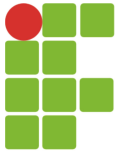


UVa 750

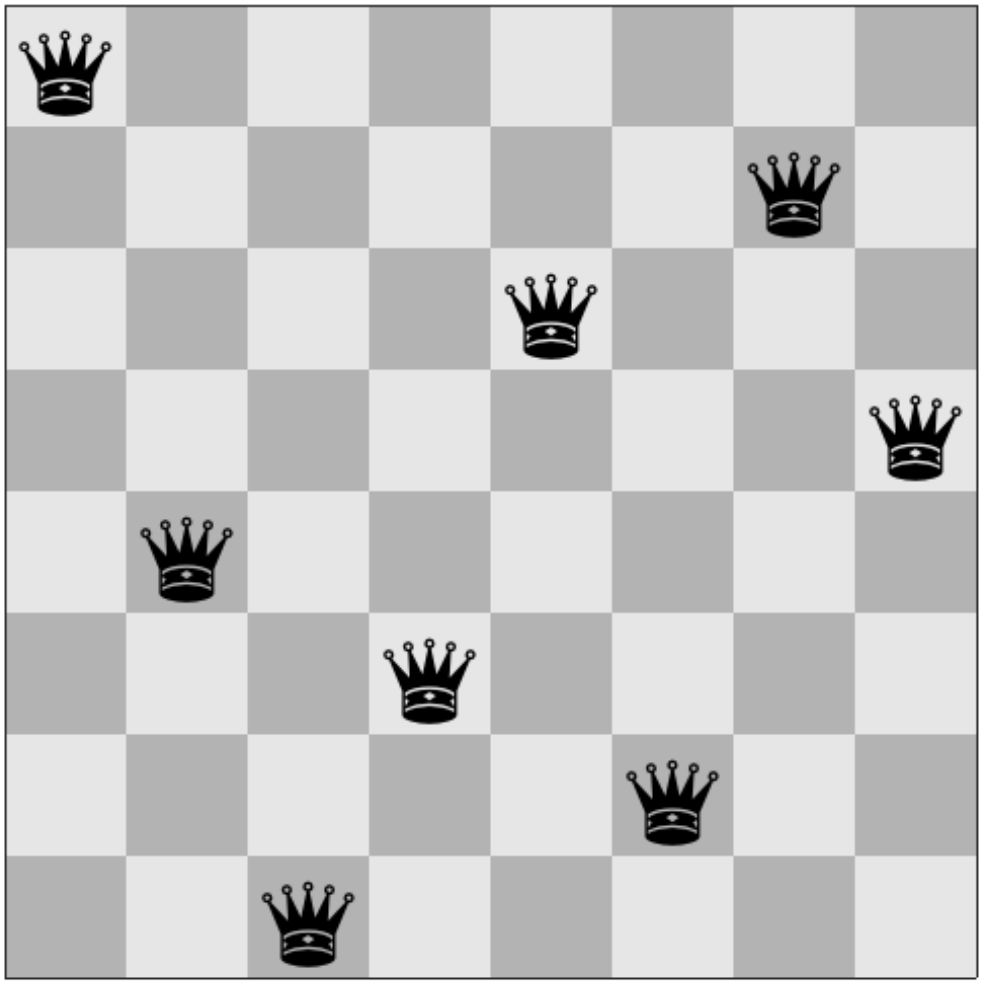
Descrição do Problema

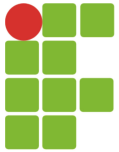
- O problema UVa 750 consiste no problema das 8 rainhas.
- O objetivo é posicionar 8 rainhas no tabuleiro de xadrez sem que uma esteja em posição de atacar a outra.
- Uma rainha está em posição de ataque caso ela esteja na mesma diagonal, linha ou coluna de outra peça de xadrez.
- Obrigatoriamente uma das rainhas deve estar na posição (a, b) do tabuleiro.

- Restrição do problema: o tabuleiro tem tamanho 8×8.



UVa 750





UVa 750

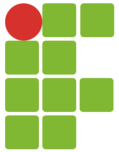
- Como o tabuleiro de xadrez possui 64 casas, um número possível de configurações de posição das 8 rainhas é:

$$\binom{64}{8} = 4426165368 \approx 10^{10}$$

- Levando em consideração que duas rainhas não podem ser dispostas na mesma coluna, o espaço de busca é reduzido para:

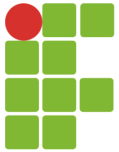
$$8^8 = 16777216 \approx 10^8$$

- Usando a mesma observação para as linhas e diagonais é possível reduzir ainda mais o espaço de busca.



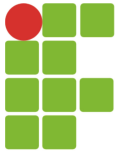
UVa 750

- Para construir as soluções de maneira incremental podemos utilizar uma técnica chamada de *backtracking*.
- O *backtracking* é uma abordagem recursiva que a cada passo faz uma escolha aumenta a solução gerada.
- Caso uma determinada escolha seja feita e constate-se que ela é inviável em termos de obtenção da solução, pode-se podar este ramo de computação e retornar para a chamada recursiva anterior, fazendo uma outra escolha.
- Isso é conhecido como poda.



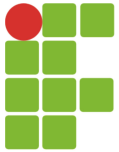
UVa 12455: Solução

```
27  void backtrack(int c) {
28      if (c == 8 && row[b] == a) {
29          ++lineCounter;
30          cout << setw(2) << lineCounter << "      "  << row[0]+1;
31          for (int j = 1; j < 8; j++){
32              cout << " " <<  row[j] + 1;
33          }
34          cout << endl;
35          return;
36      }
37       for (int r = 0; r < 8; r++){
38          if (place(r, c)) {
39              row[c] = r;
40              backtrack(c + 1);
41          }
42      }
43  }
```



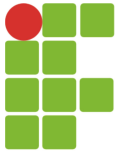
UVa 12455: Solução

```
16  bool place(int r, int c) {
17      for (int prev = 0; prev < c; prev++){
18          if (row[prev] == r || (abs(row[prev] - r) == abs(prev - c))){
19              return false;
20          }
21      }
22      return true;
23  }
```



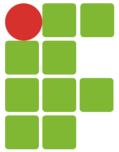
Sumário

3 Considerações



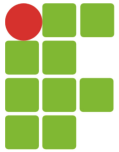
Considerações

- O principal problema de uma solução baseada em busca completa é justamente o tamanho do espaço de busca.
- Existem algumas dicas que podem ajudar no projeto de uma solução deste tipo.



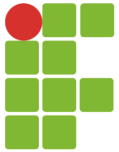
Considerações

- Geradores vs Filtros: Programas podem ser baseados em gerar todas as soluções e selecionar aquelas válidas ou gerar uma solução válida incrementalmente, partindo de subsoluções, ao mesmo tempo que poda soluções inválidas, como no caso da solução das 8 rainhas.
- A primeira situação pode ser construída incrementalmente com *backtracking* e permite podas enquanto a segunda normalmente é mais fácil de implementar e geralmente possui uma implementação interativa.



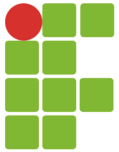
Considerações

- Poda de soluções inválidas ou piores: Pode soluções inválidas o mais cedo possível, isso acarretará em um espaço de busca menor e consequentemente, menos processamento será utilizado. Outra estratégia é, se dada uma subsolução, o possível valor da solução formada a partir desta for pior que o valor de uma solução já encontrada, a poda pode ser efetuada sem problemas.



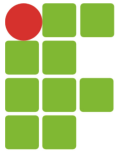
Considerações

- Explore a Simetria: algumas soluções podem ser obtidas de outras soluções considerando uma rotação ou um espelhamento, dispensando efetuar o processamento para encontrá-las. Sempre que necessário, explore essa propriedade.



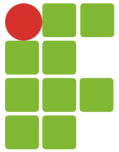
Considerações

- Pré-processamento: Dependendo do problema, é vantajoso perder um tempo construindo alguma estrutura de dados que agilize algum tipo de consulta, isto é conhecido como pré-processamento.



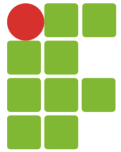
Considerações

- Otimize seu código: procure otimizar sempre que possível, métodos de leitura, acesso a memória, uso da memória cache.
- Um bom entendimento da arquitetura de computador pode ajudar a agilizar uma solução baseada em busca completa. Exemplos:
  - Acessar matrizes linha por linha é mais eficiente do que coluna por coluna;



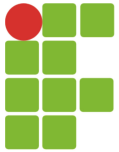
Considerações

- Otimize seu código: procure otimizar sempre que possível, métodos de leitura, acesso a memória, uso da memória cache.
- Um bom entendimento da arquitetura de computador pode ajudar a agilizar uma solução baseada em busca completa. Exemplos:
  - Utilizar vetores de bits é mais eficiente do que utilizar um vetor de booleanos ou de inteiros. Menos acessos à memória são necessários.



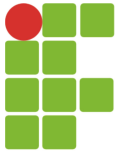
Considerações

- Otimize seu código: procure otimizar sempre que possível, métodos de leitura, acesso a memória, uso da memória cache.
- Um bom entendimento da arquitetura de computador pode ajudar a agilizar uma solução baseada em busca completa. Exemplos:
  - Utilize estruturas fixas com tamanho suficiente para a maior entrada do problema. Isto pode ser preferível do que utilizar estruturas dinâmicas como `<vector>` se o objetivo é otimizar o máximo de tempo possível.



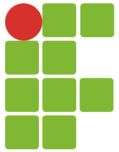
Considerações

- Otimize seu código: procure otimizar sempre que possível, métodos de leitura, acesso a memória, uso da memória cache.
- Um bom entendimento da arquitetura de computador pode ajudar a agilizar uma solução baseada em busca completa. Exemplos:
  - Com base no exemplo anterior, declare o máximo dessas variáveis em escopo global (não use isso em qualquer coisa que não seja programação competitiva).



Considerações

- Otimize seu código: procure otimizar sempre que possível, métodos de leitura, acesso a memória, uso da memória cache.
- Um bom entendimento da arquitetura de computador pode ajudar a agilizar uma solução baseada em busca completa. Exemplos:
  - Utilizar vetores para char (C-style) normalmente é mais eficiente do que usar o tipo string do C++.



Considerações

- A busca completa não serve para todos os problemas.
- Caso ela não seja suficiente, deve-se buscar utilizar um algoritmo mais eficiente.