

Força Bruta, Backtracking e Busca Exaustiva

Paradigmas de projeto de algoritmos

José Elias Claudio Arroyo

Departamento de Informática
Universidade Federal de Viçosa

INF 332 - 2022/2

1 Força Bruta

- Ordenação
- Busca Sequencial
- Casamento de Strings
- Subsequência Consecutiva
- Closest-Pair
- Problema da Envoltória Convexa

2 Backtracking

- Problema das n-Rainhas
- Problema da Soma de Subconjuntos

3 Busca Exaustiva

- Problema do Caixeiro Viajante
- Problema da Designação
- Problema da Mochila

- Estratégia mais simples para construção de algoritmos.
 - **Força bruta** geralmente utiliza diretamente a definição do problema em sua solução.

Força Bruta - Exemplo

Exemplo:

- Para calcular a^n ($a \neq 0$ e inteiro $n \geq 0$),

$$a^n = \underbrace{a \times a \times \cdots \times a}_{n \text{ vezes}}$$

- Para calcular $n!$ ($n \geq 1$),
 $n! = n * (n - 1) * \dots * 2 * 1$

Força Bruta - Exemplo

Exemplos:

- Busca sequencial
- Multiplicação de duas matrizes
- Verificação consecutiva de inteiros para determinar o $MDC(m, n)$

Força Bruta - Vantagens e Desvantagens

Pontos fracos:

- Raramente resulta em algoritmos eficientes;
- Podem ser tão lentos e inaceitáveis.

Pontos fortes:

- Simples;
- Ampla aplicabilidade;
- Algoritmos razoáveis para problemas importantes (e.g., multiplicação de matrizes, ordenação e busca);
- Útil para solucionar problemas com entradas pequenas;
- Base teórica para estudar outras técnicas de projeto de algoritmos.

Ordenação por seleção

Ordenar um arranjo $[A[0], A[1], \dots, A[n-2], A[n-1]]$ (ordem crescente)

Selection Sort

- Procure o **menor** elemento no arranjo e troque-o com o **primeiro**;
- Repita o processo anterior começando à direita do menor;
- De uma forma geral:
 - Na iteração i , com $(0 \leq i \leq n-1)$, encontre o menor elemento $A[\min]$ em $A[i..n-1]$ e troque-o com $A[i]$:

$$\underbrace{A[0] \leq A[1] \leq \dots \leq A[i-1]}_{\text{elementos ordenados}}, A[i], \dots, A[\min], \dots, A[n-1]$$

(troque $A[i]$ com $A[\min]$, onde \min é a posição do menor elemento)

$$\underbrace{A[0] \leq A[1] \leq \dots \leq A[i-1] \leq A[\min]}_{\text{elementos ordenados}}, A[i+1], \dots, A[l], \dots, A[n-1]$$

Ordenação por seleção

89	45	68	90	29	34	17
17	45	68	90	29	34	89
17	29	68	90	45	34	89
17	29	34	90	45	68	89
17	29	34	45	90	68	89
17	29	34	45	68	90	89
17	29	34	45	68	89	90

Ordenação por seleção

Algoritmo para ordenar um vetor $[A[0], A[1], \dots, A[n-2], A[n-1]]$:

```
Selecao(A[], n) {  
    Para i = 0 até n - 2: {  
        min = i;  
        Para j = i + 1 até n-1: {  
            Se (A[j] < A[min]):  
                min = j;  
        }  
        troca(A[i], A[min]);  
    }  
}
```

Ordenação por seleção

- Tamanho da entrada: número de elementos n ;
- **Operação básica:** comparação $A[j] < A[\min]$;
- O número de comparações executadas depende apenas do tamanho da entrada:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1) - (i+1) + 1 \\ &= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}. \end{aligned}$$

Ordenação por seleção executa em $\Theta(n^2)$ para todas entradas. Repare que trocas no arranjo ocorrem apenas $\Theta(n)$ vezes.

Método da bolha

- Compare elementos adjacentes e troque-os de lugar se estiverem fora de ordem.
- Em uma iteração completa o maior elemento vai para a última posição (“sobe até a superfície, como uma bolha”).
- A próxima iteração faz com que o segundo maior elemento também suba até a superfície.
- Na i -ésima iteração ($0 \leq i \leq n - 2$), temos:

$$A_0, \dots, \underbrace{A_j \leftrightarrow A_{j+1}, \dots, A_{n-i-1}}_{\text{trocar?}} \mid \underbrace{A_{n-i} \leq \dots \leq A_{n-1}}_{\text{elementos ordenados}}$$

Método da bolha

89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17		90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17		90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17		90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17		90
45		68		29		34		17		89		90

etc.

Algoritmo para ordenar um vetor $[A[0], A[1], \dots, A[n-2], A[n-1]]$:

```
Bolha(A[], n) {  
    Para i = 0 até n - 2:  
        Para j = 0 até n - i - 2:  
            Se (A[j] > A[j+1]):  
                troca(A[j], A[j+1]);  
}
```

Método da bolha

O número de comparações é dado por:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} (n-2-i) + 1 = \sum_{i=0}^{n-2} (n-1-i) \\ &= \frac{(n-1)n}{2} \in \Theta(n^2). \end{aligned}$$

O número de trocas depende da entrada.

Geralmente a aplicação do método de força bruta resulta em um algoritmo que **pode ser melhorado** sem muito esforço.

Geralmente a aplicação do método de força bruta resulta em um algoritmo que **pode ser melhorado** sem muito esforço.

Para o método da bolha:

- Se o algoritmo não fizer nenhuma troca em uma dada iteração, então o arranjo já está ordenado.

Embora o algoritmo fique mais rápido, a classe de complexidade permanece a mesma: $\Theta(n^2)$.

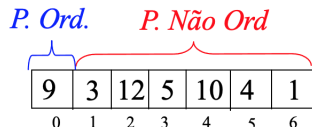
Método da bolha

```
Bolha(A[], n) {  
    Para i = 0 até < n-2:{  
        trocou = false;  
        Para j = 0 até n - i - 2:  
            Se(A[j] > A[j+1]): {  
                troca(A[j], A[j+1]);  
                trocou = true;  
            }  
        Se(!trocou) break;  
    }  
}
```

Ordenação por Inserção

- O array é “dividido” em duas partes:

Parte Ordenada e *Parte Não-Ordenada*.



- No início, a *Parte Ordenada* é formada somente pelo primeiro elemento da lista. A *Parte Não-Ordenada* é formada pelos outros elementos (a partir da segunda posição).
- Um a um, os elementos da *Parte Não-Ordenada* são inseridos na posição correta da *Parte Ordenada*, fazendo o deslocamento necessário de elementos.
- O algoritmo termina quando não há mais elementos na *Parte Não-Ordenada*.

Ordenação por Inserção

89		45	68	90	29	34	17
45		89		68	90	29	34
45		68		89		90	29
45		68		89		90	
29		45		68		89	
29		34		45		68	
17		29		34		45	

Ordenação por Inserção

ALGORITHM *InsertionSort*($A[0..n - 1]$)

//Sorts a given array by insertion sort

//Input: An array $A[0..n - 1]$ of n orderable elements

//Output: Array $A[0..n - 1]$ sorted in nondecreasing order

for $i \leftarrow 1$ **to** $n - 1$ **do**

$v \leftarrow A[i]$

$j \leftarrow i - 1$

while $j \geq 0$ **and** $A[j] > v$ **do**

$A[j + 1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j + 1] \leftarrow v$

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Ordenação por Inserção

Tempo de melhor caso (Quando o Array está em ordem crescente):

$$T(n) = \sum_{i=1}^{n-1} 1 = n - 1 .$$

Tempo de pior caso (Quando o Array está em ordem decrescente):

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} .$$

Busca sequencial

Buscar um elemento *chave* numa lista $[A[0], A[1], \dots, A[n-2], A[n-1]]$

O algoritmo de **busca sequencial** compara cada elemento da lista com a *chave* dada. Tem-se dois casos:

- a *chave* é encontrada;
- ou compara-se todos elementos na lista e chave não é encontrada.

Exemplo clássico de como a estratégia de força bruta pode ser simples.

Busca sequencial

Algoritmo para buscar uma *chave* numa lista $[A[0], \dots, A[n - 1]]$:

```
Busca(A[], chave, n) {  
    Para i = 0 até n-1:  
        Se (A[i] == chave):  
            return i;  
    return -1;  
}
```

Truque simples para busca sequencial:

- Se colocarmos a *chave* no final da lista a busca será sempre bem sucedida;
- Elimina a necessidade de checar pelo término da lista.

Truque simples para busca sequencial:

- Se colocarmos a *chave* no final da lista a busca será sempre bem sucedida;
- Elimina a necessidade de checar pelo término da lista.

Outra melhora simples se a lista estiver **ordenada**:

- Busca é interrompida assim que um elemento maior ou igual a chave é encontrado.

Truque simples para busca sequencial:

- Se colocarmos a *chave* no final da lista a busca será sempre bem sucedida;
- Elimina a necessidade de checar pelo término da lista.

Outra melhora simples se a lista estiver **ordenada**:

- Busca é interrompida assim que um elemento maior ou igual a chave é encontrado.

Algoritmo ainda é **linear** no pior caso e no caso médio.

Busca sequencial

Adicionando a chave ao final da lista reduz o número de comparações.

```
Busca(A[], chave, n) {  
    A[n] = chave;  
    i = 0;  
    Enquanto (A[i] != chave) :  
        i = i+1;  
    return i;  
}
```

Comparação de Strings de Caracteres

Neste problema são fornecidos como entrada uma string T de n caracteres (**texto**) e uma string P de m caracteres (**padrão**), $m \leq n$. O objetivo é encontrar uma substring de T que coincida com P .

Sejam $T[0 \dots n - 1]$ e $P[0 \dots m - 1]$ o texto e o padrão, respectivamente. O algoritmo que resolve o problema deve retornar o índice i do caracter mais à esquerda no texto que casa com o padrão, tal que:

$$T[i] = P[0], T[i + 1] = P[1], \dots, T[i + j] = P[j], \dots, T[i + m - 1] = P[m - 1]$$

Casamento de strings

Exemplo:

$T = [\text{NOBODY_NOTICED_HIM}]$

$P = [\text{NOT}]$

N O B O D Y _ N O T I C E D _ H I M
 N O T

Para este exemplo, o algoritmo retornará $i = 7$ (posição inicial do casamento).

Força bruta para casamento de strings

Algoritmo força bruta:

- 1 Alinhe P no início de T ;
- 2 Movendo da esquerda para direita, compare cada caracter de P com o caracter correspondente de T até:
 - todos caracteres casarem (busca bem-sucedida);
 - um dos caracteres não casa com seu correspondente.
- 3 Enquanto padrão não for encontrado e o texto não é completamente verificado, re-alinhe o padrão uma posição para direita e repita o passo 2.

Força bruta para casamento de strings

```
Casamento(T[], n, P[], m) {  
    for (i = 0; i < n - m; i++) { //Alinha  
        j = 0;  
        while(j < m && T[i+j] == P[j])  
            j = j+1;  
        if(j == m) //todos os caracteres casarem  
            return i;  
    }  
    return -1;  
}
```


Força bruta para casamento de strings

- O algoritmo geralmente re-alinha o padrão após uma comparação. No entanto, o pior caso é muito pior:
- O algoritmo pode fazer todas as m comparações antes de re-alinhar; e isso pode ocorrer em todas tentativas;
- No **pior caso** o algoritmo roda em $\Theta(nm)$.

Exemplo (Pior Caso):

$T = [N N N N N N N N N N T]$
 $P = [N N T]$

Subsequência Consecutiva Máxima

Dada uma sequência $X = [x_1, x_2, \dots, x_n]$ de n números reais.
Encontrar uma subsequência consecutiva $Y = [x_i, x_{i+1}, \dots, x_j]$ de X ,
 $1 \leq i \leq j \leq n$, tal que a soma dos elementos de Y seja máxima.

Exemplos:


- $X = [4, 2, -7, 3, 0, -2, 1, 5, -2] \Rightarrow \text{soma} = 7.$
- $X = [-2, 11, -4, 13, -5, 2] \Rightarrow \text{soma} = 20.$
- $X = [-1, -2, 0] \Rightarrow \text{soma} = 0.$
- $X = [4, 2, 8, 1] \Rightarrow \text{soma} = 15.$


Subsequência Consecutiva Máxima


Algoritmo Força Bruta:

Analisar todas as possíveis subsequências consecutivas, calcular as somas e selecionar a sequência com maior soma.

Retornar as posições inicial (i) e final (j) da subsequência máxima.

$$X = [-2, 11, -4, 13, -5, 2]$$


$$X = [-2, 11, -4, 13, -5, 2]$$


$$X = [-2, 11, -4, 13, -5, 2]$$


Subsequência Consecutiva Máxima

Algoritmo Força Bruta:

```
SCM( X[], n){  
    somaMax = 0;  
    for (i = 0; i < n; i++)           //para cada ponto de inicio  
        for (j = i; j < n; j++){     //para cada ponto final  
            soma = 0;  
            for (k = i; k <= j ; k++)  
                soma = soma + X[k];  
            if ( soma > somaMax){  
                somaMax = soma; I = i; J = j;  
            }  
        }  
    }  
    return (I, J, somaMax);  
}
```

Complexidade: $\Theta(n^3)$.

Subsequência Consecutiva Máxima

Algoritmo Força Bruta Melhorado:

```
SCM( X[], n){  
    somaMax = 0;  
    for (i = 0; i < n; i++){  
        soma = 0;  
        for (j = i; j < n; j++){  
            soma = soma + X[j];  
            if ( soma > somaMax){  
                somaMax = soma; I = i; J = j;  
            }  
        }  
    }  
    return (I, J, somaMax);  
}
```

Complexidade: $\Theta(n^2)$.

Os dois pontos de menor distância

- Seja um conjunto de n pontos em \mathbb{R}^2 : $\{p_1, p_2, \dots, p_n\}$ ($n \geq 2$). Determinar os dois pontos mais próximos (pontos de menor distância).
- A distância entre dois pontos $p_i = (x_i, y_i)$ e $p_j = (x_j, y_j)$ é determinada pela distância Euclidiana:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Os dois pontos de menor distância

Algoritmo Força Bruta:

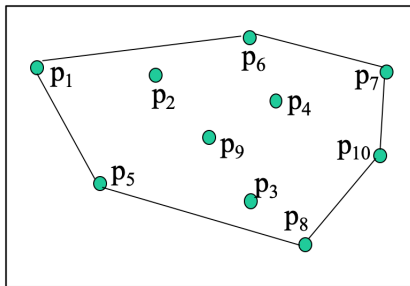
- Calcular a distância entre todos os pares de pontos diferentes (p_i e p_j) e identificar os dois pontos com menor distância.
- Retornar os índices desses pontos.

```
ClosestPair (Pontos p[], n){
    dmim = infinito;
    Para i = 1 até n-1:
        Para j = i+1 até n:{
            d= sqrt((p[i].x - p[j].x)^2 + (p[i].y - p[j].y)^2);
            Se(d < dmim){
                dmin = d;  i1 = i;  i2 = j;
            }
        }
    return (dmin, i1, i2);
}
```

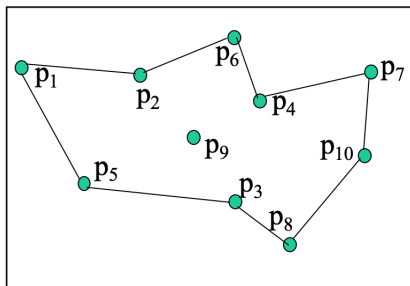
A complexidade de tempo: $\Theta(n^2)$.

Problema da Envoltória Convexa

- Seja S um conjunto de n pontos no plano \mathbb{R}^2 . A *Envoltória Convexa* ou *Casco Convexo* de S é o menor polígono convexo que contém todos os pontos de S (ou seja, todos os pontos de S devem estar dentro do polígono ou sobre sua borda).
- O problema consiste em encontrar os pontos que formam a Envoltória Convexa de S .



Polígono Convexo

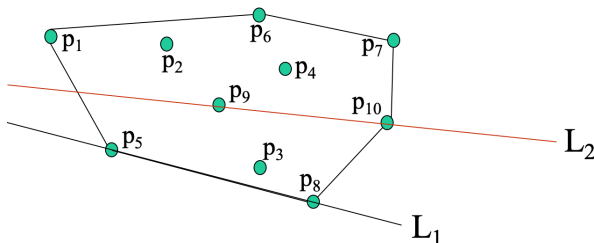


Polígono Não Convexo

Algoritmo de Força Bruta:

- $Casco = \emptyset$.
- **Para** cada par de pontos p_i e p_j de S :
 - Determinar a reta L que passa pelos pontos p_i e p_j .
 - **Se** p_i e p_j são **pontos extremos** (i.e. se todos os outros pontos de S estão em um dos semiplanos determinados pela reta L):
 - $Casco = Casco \cup \{p_i, p_j\}$.
- **Fim-Para**
- Retorne $Casco$.

Problema da Envoltória Convexa



- p_5 e p_8 são **pontos extremos**, pois a reta L_1 (determinada por estes pontos) passa pela fronteira do polígono convexo, ou seja, todos os outros pontos estão no **semiplano superior** determinado por L_1 .
- p_9 e p_{10} não são pontos extremos, pois a reta L_2 (determinada por estes pontos) não passa pela fronteira do polígono convexo, ou seja, existem pontos nos dois semiplanos determinados por L_2 .

Problema da Envoltória Convexa

- Uma reta que passa pelos pontos $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ é definida como: $ax + by - c = 0$,
onde $a = (y_2 - y_1)$, $b = (x_1 - x_2)$ e $c = x_1 y_2 - y_1 x_2$
- Todos os outros pontos (x, y) estão num mesmo semiplano se:
 $ax + by - c \leq 0$, ou $ax + by - c \geq 0$, $\forall (x, y)$.

Exercício

Calcule a complexidade do algoritmo de força bruta que determina todos os pontos extremos (os pontos da envoltória convexa).

- *Backtracking* é um método de busca de possíveis soluções de um problema combinatório.
- Sistemáticamente procura/constrói soluções, eliminando soluções inválidas (alternativas que não satisfaçam todas as condições do problema).
- A busca monta uma árvore (do espaço de estados) para organizar as soluções do problema.
- Produz soluções em tempo razoável para alguns problemas, mas o tempo de pior caso é **exponencial**.

- Na árvore do espaço de estados:
 - Nós: são soluções parciais
 - Arestas: são escolhas a partir de soluções parciais estendidas.
- Geralmente, a árvore é explorada usando **busca em profundidade**.
- Nós não promissores são **podados**:
 - Interrompa a exploração das subárvores enraizadas em um nó que não produzirão soluções viáveis e **retorne** ao pai desse nó para continuar a busca.

Problema das n-Rainhas

- Coloque n rainhas em um tabuleiro de xadrez $n \times n$, de modo que nenhuma delas esteja na mesma linha, coluna ou diagonal.

Inserção de 4 rainhas: **1, 2, 3, 4**

		1	2	3	4
Rainha 1	→	1			
Rainha 2	→	2			
Rainha 3	→	3			
Rainha 4	→	4			



	1	2	3	4
1		1		
2				2
3	3			
4			4	

Problema das n-Rainhas

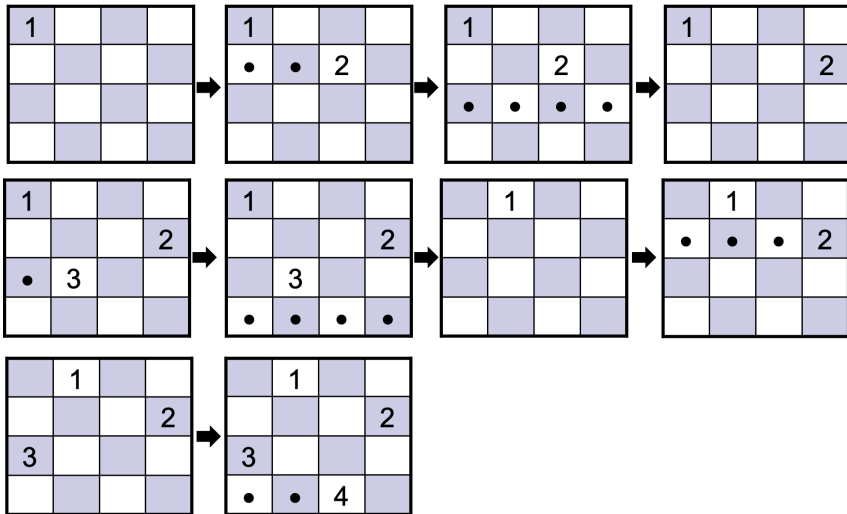
- **Representação de uma solução:** Uma solução do problema das n-rainhas pode ser representado por um **vetor** (x_1, \dots, x_n) de tamanho n , onde x_i indica a **coluna** da rainha i (ou seja, a coluna onde a rainha i é inserida).

	1	2	3	4
1		1		
2				2
3	3			
4			4	

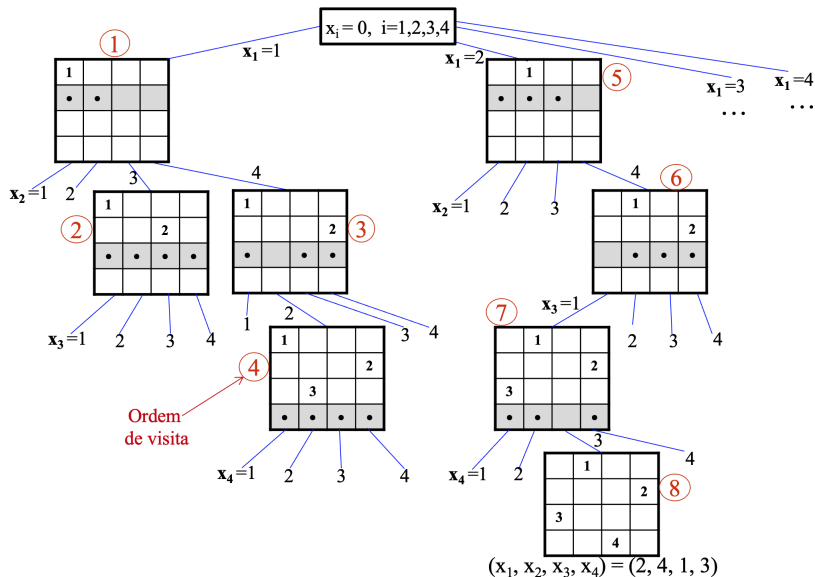
$$(x_1, x_2, x_3, x_4) = (2, 4, 1, 3)$$

Problema das n-Rainhas

Algoritmo backtracking



Árvore para o problema das 4-Rainhas



Problema das n-Rainhas

```
Backtrack_Rainhas(x[], r, n){  
  Se(r == n) Imprima solucao x;  
  Senão  
    Para c =1 até n:  
      Se(Possivel(x, r+1, c) ){  
        x[r+1] = c;  
        Backtrack_Rainhas(x, r+1, n);  
      }  
}
```

- Chamada:
Backtrack_Rainhas(x, 0, n); //x é um vetor de tamanho n .
- Possivel(x, r, c):
testa se a rainha r pode ser inserida na coluna c .

Problema das n-Rainhas

Uma rainha r não pode ser inserida na coluna c se:

- Uma rainha i , inserida anteriormente ($i < r$), já está na coluna c se: $x[i] = c$;
- Ou, uma rainha i , inserida anteriormente, está na diagonal de (r, c) se: $|r - i| = |c - x[i]|$ (distância das linhas igual à distância das colunas).

Problema da Soma de Subconjuntos

Problema

Dado um conjunto com n números positivos: $w = (w_1, \dots, w_n)$ e um número M . Encontrar todos os subconjuntos de w cuja soma seja igual a M .

Exemplo

$n = 4$, $w = (w_1, \dots, w_n) = (11, 13, 24, 7)$ e $M = 31$.

- Existem 2 subconjuntos:
 - $(11, 13, 7)$, $11 + 13 + 7 = 31$
 - $(24, 7)$, $24 + 7 = 31$.

Problema da Soma de Subconjuntos

- **Representação de uma solução:** uma solução pode ser representada por um vetor (x_1, \dots, x_n) de tamanho n , onde x_i é do tipo binário.
- $x_i = 1$ se w_i está no subconjunto, caso contrário $x_i = 0$.

Exemplo

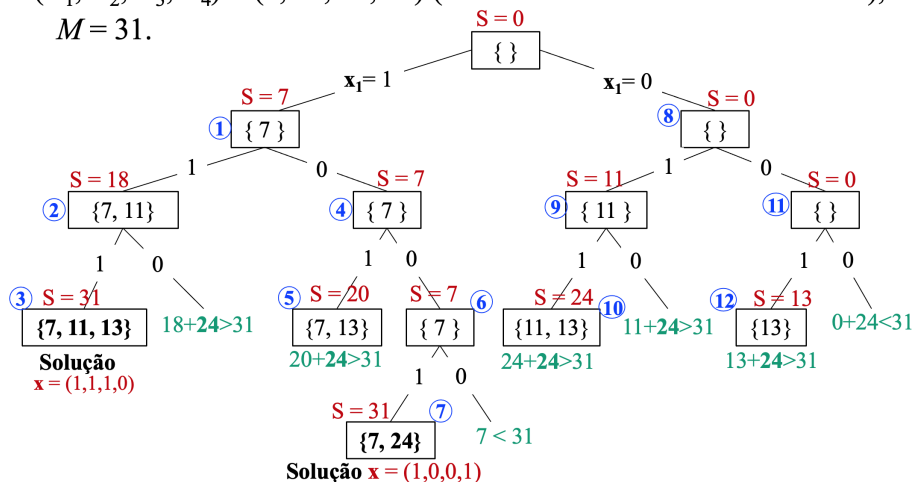
- $w = (w_1, \dots, w_n) = (11, 13, 24, 7)$ e $M = 31$.
- Soluções:
 - $x = (1, 1, 0, 1)$
 - $x = (0, 0, 1, 1)$

Problema da Soma de Subconjuntos

Árvore completa da busca backtracking

$(w_1, w_2, w_3, w_4) = (7, 11, 13, 24)$ (os números devem estar ordenados),

$M = 31$.



Problema da Soma de Subconjuntos

```
SubsetSum(w[], M, x[], S, i, T):  
    //gera filho esquerdo:  
    x[i] = 1  
    IF(S + w[i] == M):  print( x[1..i] )  
    ELSE:  
        IF ((S + w[i] + w[i+1]) <= M):  
            SubsetSum(w, M, x, S + w[i], i+1, T - w[i])  
  
    //gera filho direito:  
    IF((S + w[i+1] <= M) and (S + T - w[i] >= M)):  
        x[i] = 0  
        SubsetSum(w, M, x, S, i+1, T - w[i])
```

- $w_1 \leq M$, $T = w_1 + w_2 + \dots + w_n \geq M$
- $S = 0$ (soma parcial); $i = 1$;
- Chamada: **SubsetSum**(w[], M, x[], S, i, T);

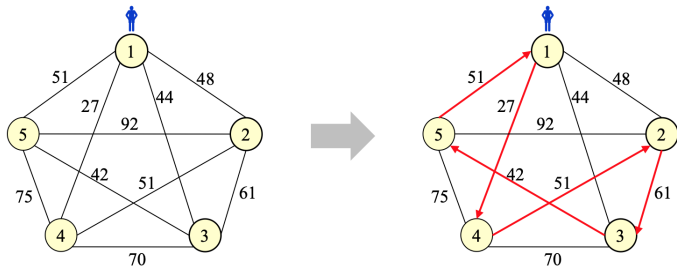
Definição: Busca exaustiva é um **método de força bruta** para problemas de otimização combinatorial (Ex. caixeiro viajante, problema da mochila, dentre outros).

Método:

- Gera-se uma lista de todas as potenciais soluções de forma sistemática;
- Avalia-se todas as soluções potenciais (calcula-se o custo), descartando soluções inválidas e armazenando a melhor solução encontrada.

Problema do Caixeiro Viajante (PCV)

Dadas n cidades (pontos ou vértices), onde são conhecidas as distâncias entre cada par de pontos. Encontrar o menor caminho que visite todos os pontos exatamente uma vez e retorne para o ponto de partida.

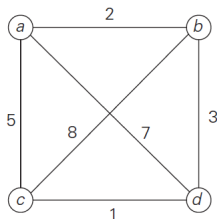


Essa é exatamente a definição de um **ciclo Hamiltoniano** em um grafo conectado e com valores nas arestas.

Problema do Caixeiro Viajante (PCV)

- Um **ciclo Hamiltoniano** pode também ser definido como uma **sequência de $n + 1$ vértices** adjacentes $i_1, i_2, \dots, i_n, i_1$, onde o primeiro e último vértices são iguais e todos os demais são distintos.
- É possível gerar todos os ciclos através da geração das **permutações** dos $n - 1$ vértices intermediários: i_2, \dots, i_n .
- Para cada permutação gerada, adicionar o vértice inicial i_1 .
- Total de permutações geradas será $(n - 1)!$.

Problema do Caixeiro Viajante (PCV)



<u>Tour</u>	<u>Length</u>	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Problema do Caixeiro Viajante (PCV)

- Existem ciclos que diferem apenas pela direção. Por exemplo:
 $a \rightarrow c \rightarrow d \rightarrow a$
 $a \rightarrow d \rightarrow c \rightarrow a$
- Pode-se reduzir o número de permutações pela metade.
- Não melhora muito a eficiência do algoritmo.
- O total de permutações ainda é $(n - 1)!/2$.
- Força bruta** é praticável apenas para valores muito pequenos de n .

Designação de tarefas

Dadas n tarefas e n pessoas, designar uma pessoa para cada tarefa. Cada pessoa é designada a exatamente uma tarefa, e cada tarefa é designada a exatamente uma pessoa.

- O custo de assinalar pessoa i para tarefa j é de $C[i, j]$.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

- O problema é encontrar a distribuição com menor custo.

Força bruta: geram-se todas possíveis designações e calculam-se seus custos; a designação com menor custo é então selecionada.

- Soluções válidas podem ser descritas através de tuplas de tamanho n : $\langle j_1, \dots, j_n \rangle$, onde o i -ésimo componente representa a tarefa designada a pessoa i .
- Busca exaustiva gera, portanto, todas **permutações** dos inteiros $1, 2, \dots, n$, calculando o custo total de cada permutação e selecionando aquela com menor custo.

Designação de tarefas

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

$\langle 1, 2, 3, 4 \rangle$	$\text{cost} = 9 + 4 + 1 + 4 = 18$	
$\langle 1, 2, 4, 3 \rangle$	$\text{cost} = 9 + 4 + 8 + 9 = 30$	
$\langle 1, 3, 2, 4 \rangle$	$\text{cost} = 9 + 3 + 8 + 4 = 24$	
$\langle 1, 3, 4, 2 \rangle$	$\text{cost} = 9 + 3 + 8 + 6 = 26$	etc.
$\langle 1, 4, 2, 3 \rangle$	$\text{cost} = 9 + 7 + 8 + 9 = 33$	
$\langle 1, 4, 3, 2 \rangle$	$\text{cost} = 9 + 7 + 1 + 6 = 23$	

- O **número de permutações** é igual a $n!$, portanto uma abordagem força bruta é válida apenas para valores muito pequenos de n .

Exercício

Escreva o pseudocódigo de um algoritmo para gerar todas as permutações de $\{1, \dots, n\}$.

Problema da mochila (PM)

Dados n itens:

- Pesos: w_1, w_2, \dots, w_n .
- Valores: v_1, v_2, \dots, v_n .
- E capacidade W de uma mochila.

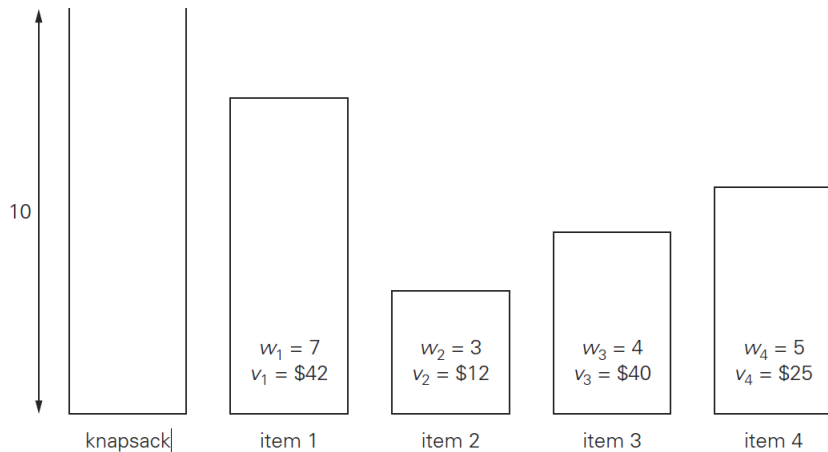
Retornar o subconjunto mais valioso de itens que cabem na mochila.

Problema da mochila (PM)

A busca exaustiva para o problema:

- Considere todos os subconjuntos dos n itens;
- Calcule o peso total de cada subconjunto para identificar soluções válidas;
- Encontre o subconjunto mais valioso dentre eles.

Problema da mochila (PM)



Problema da mochila (PM)

Subset	Total weight	Total value
\emptyset	0	\$ 0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25
{1, 2}	10	\$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	\$52
{2, 4}	8	\$37
{3, 4}	9	\$65
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Problema da mochila (PM)

O número de subconjuntos para n itens é 2^n . Portanto a busca exaustiva é $\Omega(2^n)$, independente de quão eficiente é a geração dos subconjuntos.

Gerando subconjuntos

Como gerar os 2^n subconjuntos de $A = \{a_1, \dots, a_n\}$?
(**conjunto de partes**)

Gerando subconjuntos

Como gerar os 2^n subconjuntos de $A = \{a_1, \dots, a_n\}$?
(**conjunto de partes**)

Gerar subconjuntos de $\{a_1, \dots, a_k\}$, $1 \leq k \leq n$
Os subconjuntos podem ser divididos em dois grupos:
aqueles que contêm a_k , e aqueles que não contêm a_k .

Ao obter os subconjuntos de $\{a_1, \dots, a_{k-1}\}$, os subconjuntos de $\{a_1, \dots, a_k\}$ podem ser obtidos ao inserirmos a_k em cada um deles.

Gerando subconjuntos

Exemplo: Gere o conjunto de partes de $\{a_1, a_2, a_3\}$

- 1 $k = 0: \emptyset$
- 2 $k = 1: \emptyset \{a_1\}$
- 3 $k = 2: \emptyset \{a_1\} \{a_2\} \{a_1, a_2\}$
- 4 $k = 3: \emptyset \{a_1\} \{a_2\} \{a_1, a_2\} \{a_3\} \{a_1, a_3\} \{a_2, a_3\} \{a_1, a_2, a_3\}$

Método da string de bits: crie uma correspondência entre os elementos do conjunto com uma string de bits.

Para o conjunto $\{a_1, a_2, a_3\}$ considere uma string com três bits.

Gerando subconjuntos

Método da string de bits: crie uma correspondência entre os elementos do conjunto com uma string de bits.

Para o conjunto $\{a_1, a_2, a_3\}$ considere uma string com três bits.

string	000	001	010	011	100	101	110	111
subconjuntos	\emptyset	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$