Problema do Caixeiro Viajante

❖ Integrantes:

- Caroline Akimi Kurosaki Ueda - 15445630
- Nicolas de Sousa Maia - 15481857
- Vitor Rocha Veiga - 15492449

❖ Modelagem da Solução - Força Bruta:

A nossa solução do Problema do Caixeiro Viajante utiliza a estrutura de dados lista linear sequencial. Essa estrutura permite adicionar e remover elementos no fim e acessar qualquer posição por meio de um índice com complexidade O(1), facilitando a iteração pelas cidades ao minimizar o custo computacional.

As distâncias entre as cidades são armazenadas em um vetor de listas. Uma desvantagem dessa utilização é a necessidade de funções, como “lista_buscar” e “lista_trocar”, para acessar esses dados. O uso dessas função aumenta o tempo de execução do programa, o que é problemático, principalmente, no algoritmo da “força bruta” que possui uma complexidade elevada.

Dessa forma, na solução da “força bruta”, nós utilizamos uma função, chamada de “força_bruta”, que, a partir de um método recursivo, gera todas as combinações de cidades. As combinações de cidades que atendem a condição de ter n-1 elementos (tendo em vista que a cidade inicial já foi incluída) são analisadas, verificando se a soma da distância percorrida ao passar por elas é menor do que a menor distância até então, caso verdadeiro, atualizando a melhor distância e o melhor caminho.

Entrando mais a fundo no método recursivo que a função “força_bruta” usa para montar as combinações de cidades, vemos que ela monta essa combinação a partir da estrutura de dados lista sequencial, adicionando ao fim da lista, contudo, antes de fazer isso, o algoritmo checa se aquela cidade já está dentro da lista, por meio de um vetor que guarda quem já foi adicionado.

Isso se repete, entrando em mais chamadas recursivas, até que a lista tenha tamanho n-1, o que faz ela entrar no condicional que checa se a combinação atual é melhor do que a última de, também, n-1 elementos. Dessa forma, ao sair deste “if”, essa chamada recursiva é encerrada. Depois disso, no código, após a chamada da função está o comando “lista_remover_fim”, o qual remove o último item da lista.

Dessa forma, dependendo de qual chamada recursiva o algoritmo esteja, ele irá adicionando e retirando os elementos da lista de forma a montar todas as combinações possíveis.

Vejamos um exemplo de uma lista de cidades com n=4:

2
2 3
2 3 4 → entra no condicional e acaba a primeira chamada recursiva, apagando
2 4 a cidade 4, a cidade 3 e depois adicionando de novo a cidade 4.
2 4 3
3
3 2
…

Dessa forma, o programa é capaz de analisar todas as combinações possíveis, calcular o valor de suas distâncias e decidir qual o melhor caminho.

❖ Modelagem da Solução - Programação Dinâmica:

Para esse método, nós utilizamos uma abordagem mais eficiente, a qual envolve a utilização de uma DP, que é uma estrutura de dados lista de vetores que guarda o melhor caminho da cidade 1 para uma determinada cidade i, tendo em vista as cidades que já visitamos, que são representadas por uma *bitmask*.

Por exemplo, caso nossa *bitmask* seja 010011, quer dizer que as cidades 1, 2 e 5 já foram visitadas.

Então, nós percorremos, por meio de um loop, todos os caminhos possíveis (*masks*), para conseguirmos salvar em nossa DP quais são os melhores valores para chegarmos em tais caminhos.

Por isso, nós fazemos um loop, aninhado à esse primeiro, por todas as i cidades, e, por uma questão de lógica, apenas as cidades que já estão no nosso caminho (*bitmask*) terão suas distâncias processadas para a próxima cidade, que nosso código é representada por j. Assim, dado um caminho, nós vemos qual é o melhor jeito de chegar na próxima cidade, avançando no preenchimento da DP.

Esse processo continua até termos a melhor distância para todas as cidades, que é quando nossa *bitmask* é 111…1, solucionando assim o problema.

Como nosso programa considera que a cidade inicial é sempre a 1, temos que fazer apenas um simples *shift*, de forma que a cidade inicial seja a desejada, tendo em vista que o caminho é o mesmo.

❖ Implementação:

-Arquivos do TAD: lista.c, lista.h
-Programa principal: main.c
-Extra: extra.c

❖ Instruções de execução:

- ➢ Para o normal:
  make
  ./exe
- ➢ Para o extra:
  make extra
  ./extra

(Recomenda-se usar make clean entre as execuções do normal e do extra)

❖ Complexidade:

- ➢ **Lista**

  As funções implementadas para a estrutura de dados “lista” possuem complexidade $O(1)$, ou seja, complexidade constante.

- ➢ **Força Bruta**

  A complexidade do algoritmo força bruta é $O(n!)$.

  Conseguimos deduzir a equação de recorrência do algoritmo como:

  $$T(n) = (n-1) * T(n-1) + O(n).$$

Logo, podemos usar o método da substituição para conseguirmos provar que a complexidade é $O(n!)$. Vejamos:

$$T(n) \leq c\,.\,n! \rightarrow (n-1).(n-1)! + n \leq c\,.\,n! \rightarrow (n-1)((n-1)!+1) \leq c\,.\,n!$$
$$\rightarrow (n-1)!+1 \leq c\,.\,n.(n-2)! \rightarrow n-1+\frac{1}{(n-2)!} \leq c.n$$
$$\rightarrow \frac{1}{(n-2)!} - 1 \leq n.(c-1) \rightarrow \frac{1}{(n-2)!} - 1 \leq n.c'$$

Por uma questão lógica, sabemos que o mínimo para n é 2, logo, como a função fatorial é sempre crescente, e $0!$ = 1, temos que $\frac{1}{(n-2)!} - 1 \leq 0$ e como, para uma constante $c' \geq 1$ qualquer $n.c' \geq 1$, temos como evidente que $\frac{1}{(n-2)!} - 1 \leq n.c'$. O que confirma que:

$$T(n) = O(n!).$$

➢ **Programação Dinâmica**

A complexidade da implementação com programação dinâmica é $O(2^n n^2)$. Para percorrer todas as combinações de cidades visitadas, temos um loop que itera $2^{(n-1)}$ vezes. Dentro dele, existe um loop que itera pelas n cidades de onde viemos e outro loop interno a esse último que percorre por $n-1$ cidades para onde vamos. As comparações e atribuições feitas dentro dos loops possuem complexidade constante, $O(1)$. Portanto, a complexidade, seguindo os princípios da notação *big O*, é dada por:

$$O(2^{n-1} * n * (n-1)) = O(2^n n^2).$$

❖ Análise Empírica:

➢ **Força Bruta:**

-Entrada com 5 cidades

```
❯ ./exe < input_05.in
Tempo de Execução: 0.000004 segundos
Cidade Origem: 3
Rota: 3 - 2 - 1 - 4 - 5 - 3
Menor Distância: 1750
```

-Entrada com 10 cidades

```
❯ ./exe < input_10.in
Tempo de Execução: 0.053165 segundos
Cidade Origem: 6
Rota: 6 - 2 - 7 - 5 - 1 - 8 - 9 - 3 - 4 - 10 - 6
Menor Distância: 543
```

-Entrada com 12 cidades

```
❯ ./exe < input_12.in
Tempo de Execução: 8.070295 segundos
Cidade Origem: 2
Rota: 2 - 5 - 9 - 4 - 10 - 11 - 1 - 12 - 7 - 3 - 6 - 8 - 2
Menor Distância: 969
```

➢ **Programação Dinâmica:**

-Entrada com 5 cidades

```
❱ ./extra < input_05.in
Tempo de Execução: 0.000012 segundos
Cidade Origem: 3
Rota: 3 - 2 - 1 - 4 - 5 - 3
Menor Distância: 1750
```

-Entrada com 10 cidades

```
❱ ./extra < input_10.in
Tempo de Execução: 0.001640 segundos
Cidade Origem: 6
Rota: 6 - 2 - 7 - 5 - 1 - 8 - 9 - 3 - 4 - 10 - 6
Menor Distância: 543
```

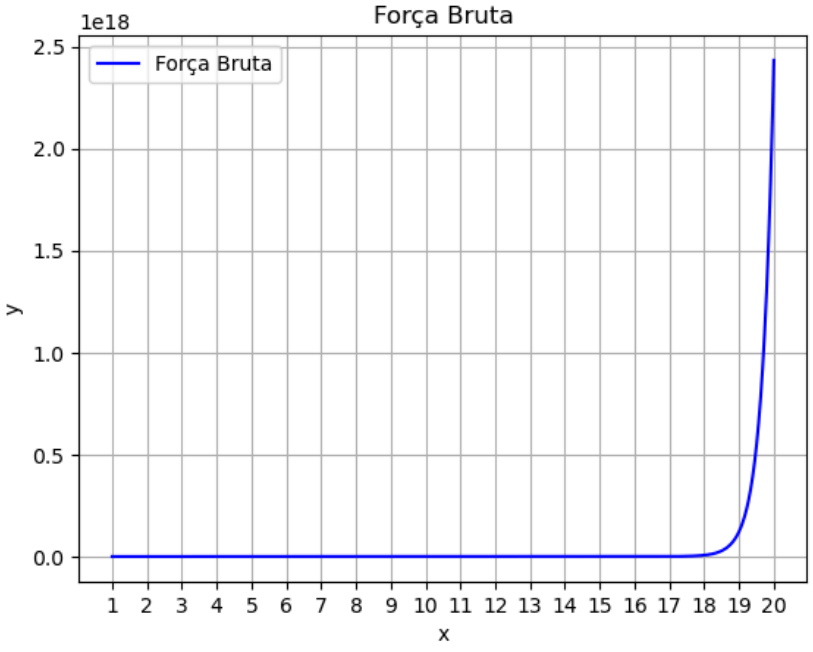
-Entrada com 12 cidades

```
❱ ./extra < input_12.in
Tempo de Execução: 0.005360 segundos
Cidade Origem: 2
Rota: 2 - 5 - 9 - 4 - 10 - 11 - 1 - 12 - 7 - 3 - 6 - 8 - 2
Menor Distância: 969
```

-Entrada com 16 cidades

```
❱ ./extra < input_gen
Tempo de Execução: 0.059915 segundos
Cidade Origem: 4
Rota: 4 - 7 - 16 - 13 - 3 - 2 - 14 - 1 - 6 - 8 - 5 - 10 - 15 - 11 - 9 - 12 - 4
Menor Distância: 225
```

❖ **Gráficos:**



❖ Conclusão:

Analisando os tempos de execução e as complexidades calculadas, podemos notar que os resultados estão coerentes. A complexidade elevada, $O(n!)$, do algoritmo "força bruta" faz com que o tempo de execução aumente consideravelmente, mesmo que sejam acrescentadas apenas mais uma ou duas cidades para a entrada. Em comparação, o algoritmo de programação dinâmica possui um tempo de execução menor, já que sua complexidade é inferior $O(2^n n^2)$ que o da primeira implementação. Observe a diferença entre o tempo para a mesma entrada (12 cidades) nos dois casos, o "força bruta" levou aproximadamente 8 segundos em relação ao outro com apenas 0.005 segundos.