## Análise Exploratória dos Dados

O código SQL está criando várias tabelas em um banco de dados chamado "csgo" para armazenar dados relacionados a partidas de um jogo de tiro chamado "CS:GO" (Counter-Strike: Global Offensive).

As tabelas criadas são as seguintes:

- Tabela "players": Armazena informações dos jogadores, como a data, nome do jogador, equipe, oponente, país, ID do jogador, ID da partida, ID do evento, nome do evento, melhor de quantas partidas, informações sobre os mapas jogados e estatísticas de desempenho dos jogadores.

- Tabela "results": Armazena informações sobre os resultados das partidas, como a data, as duas equipes envolvidas, o mapa jogado, os placares de cada equipe, o vencedor do mapa, o lado inicial escolhido (CT ou T), estatísticas de pontuação por lado e informações sobre o evento e a partida.

- Tabela "economy": Armazena informações sobre a economia do jogo, incluindo a data, o ID da partida, o ID do evento, as duas equipes envolvidas, o formato da partida (melhor de quantas), o mapa jogado, as escolhas iniciais de cada equipe, informações sobre a economia de cada equipe em cada rodada e o vencedor de cada rodada.

- Tabela "picks": Armazena informações sobre as escolhas de mapas feitas pelas equipes, incluindo a data, as duas equipes envolvidas, um indicador para verificar se as equipes foram invertidas, o ID da partida, o ID do evento, o formato da partida, informações sobre o sistema de escolha de mapas, as remoções de cada equipe e as seleções finais de mapas.

Essas tabelas foram criadas para armazenar dados relacionados a partidas de CS:GO e permitir consultas e análises futuras desses dados. Cada tabela tem suas colunas específicas que correspondem aos diferentes aspectos das partidas e estatísticas dos jogadores.

**Utilizando o banco de dados no Jupyter Notebook**

```
from mysql.connector import errorcode
import mysql.connector
import plotly.graph_objects as go
import plotly.express as px
from os import listdir
import pandas as pd
import numpy as np
```

```
def connect(host, database, user, password):

    try:
        connection = mysql.connector.connect(host=host, database=database, user=user, password=password)
        if connection != None:
            return connection

    except mysql.connector.Error as error:
        if error.errno == errorcode.ER_BAD_DB_ERROR:
            print("Database não existe")

        if error.errno == errorcode.ER_ACCESS_DENIED_ERROR:
            print("Usuáro ou  senha  está errado")
            print(error)
    return

def execute_query(connection, query:str, data:tuple=''):

    try:
        cursor = connection.cursor()
        if data != '':
            cursor.execute(query, data)
            result = cursor.fetchall()
            connection.commit()
        else:
            cursor.execute(query)
            result = cursor.fetchall()
            connection.commit()

            return result
    except mysql.connector.Error as error:
        return "Erro: {}".format(error.msg)
    finally:
        if connection.is_connected:
            connection.close()

def loop_result(results):
    from_db = []

    for result in results:
        result = list(result)
        from_db.append(result)

    if from_db != None:
        return from_db
    return
```

O código apresentado realiza as seguintes ações:

Importa os módulos necessários, como mysql.connector, plotly.graph_objects, plotly.express, os, pandas e numpy.

Define uma função chamada connect que recebe parâmetros de conexão do banco de dados (host, banco de dados, usuário e senha) e tenta estabelecer uma conexão com o banco de dados MySQL usando a biblioteca mysql.connector. Em caso de sucesso, retorna o objeto de conexão.

Define uma função chamada execute_query que recebe a conexão do banco de dados, uma consulta SQL e, opcionalmente, dados para a consulta. Essa função executa a consulta no banco de dados, busca os resultados e, se houver resultados, os retorna. Em caso de erro, retorna uma mensagem de erro.

Define uma função chamada loop_result que recebe os resultados de uma consulta SQL e percorre cada resultado, adicionando-o a uma lista. Retorna a lista resultante.

Realiza a conexão com o banco de dados MySQL usando a função connect e armazena o objeto de conexão na variável connection.

Define uma consulta SQL na variável query_results para selecionar todos os registros da tabela "results".

Define uma lista de colunas na variável columns_results correspondente às colunas da tabela "results".

Executa a consulta SQL usando a função execute_query, passando a conexão e a consulta como parâmetros, e armazena os resultados na variável results_results.

Chama a função loop_result passando os resultados da consulta para converter os resultados em uma lista.

Cria um DataFrame do pandas chamado df_results a partir da lista de resultados, usando as colunas definidas anteriormente.

Exibe o DataFrame df_results utilizando a função display (provavelmente importada de um ambiente interativo como Jupyter Notebook ou IPython).

Em resumo, o código estabelece uma conexão com um banco de dados MySQL, executa uma consulta SQL para obter os resultados da tabela "results" e exibe os resultados em um DataFrame do pandas.

- Consultando DataFrame:

```
In [5]: display(df_results)
```

| | Date_ | team_1 | team_2 | _map | result_1 | result_2 | map_winner | starting_ct | ct_1 | t_2 | t_1 | ct_2 | event_id | match_id | rank_1 | rank_2 | map_wins_1 | map_wins_1 | map_wins_2 | match_winner |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 2020-03-18 | Recon 5 | TeamOne | Dust2 | 0 | 16 | 2 | 2 | 0 | 1 | 0 | 15 | 5151 | 2340454 | 62 | 63 | 0 | 0 | 2 | 2\r |
| **1** | 2020-03-18 | Recon 5 | TeamOne | Inferno | 13 | 16 | 2 | 2 | 8 | 6 | 5 | 10 | 5151 | 2340454 | 62 | 63 | 0 | 0 | 2 | 2\r |
| **2** | 2020-03-18 | New England Whalers | Station7 | Inferno | 12 | 16 | 2 | 1 | 9 | 6 | 3 | 10 | 5243 | 2340461 | 140 | 118 | 12 | 12 | 16 | 2\r |
| **3** | 2020-03-18 | Rugratz | Bad News Bears | Inferno | 7 | 16 | 2 | 2 | 0 | 8 | 7 | 8 | 5151 | 2340453 | 61 | 38 | 0 | 0 | 2 | 2\r |
| **4** | 2020-03-18 | Rugratz | Bad News Bears | Vertigo | 8 | 16 | 2 | 2 | 4 | 5 | 4 | 11 | 5151 | 2340453 | 61 | 38 | 0 | 0 | 2 | 2\r |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **91541** | 2015-11-05 | G2 | E-frag.net | Inferno | 13 | 16 | 2 | 1 | 8 | 7 | 5 | 9 | 1970 | 2299059 | 7 | 16 | 1 | 1 | 2 | 2\r |
| **91542** | 2015-11-05 | G2 | E-frag.net | Dust2 | 16 | 13 | 1 | 1 | 10 | 5 | 6 | 8 | 1970 | 2299059 | 7 | 16 | 1 | 1 | 2 | 2\r |
| **91543** | 2015-11-04 | CLG | Liquid | Inferno | 16 | 12 | 1 | 1 | 7 | 8 | 9 | 4 | 1934 | 2299011 | 10 | 14 | 16 | 16 | 12 | 1\r |
| **91544** | 2015-11-03 | NiP | Dignitas | Train | 16 | 4 | 1 | 2 | 4 | 1 | 12 | 3 | 1934 | 2299001 | 6 | 12 | 16 | 16 | 4 | 1\r |
| **91545** | 2015-11-03 | NiP | Envy | Cobblestone | 16 | 9 | 1 | 2 | 4 | 6 | 12 | 3 | 1934 | 2299003 | 6 | 1 | 16 | 16 | 9 | 1\r |

91546 rows × 19 columns

A função display(df_results) é usada para exibir o DataFrame df_results de forma mais amigável e interativa em um ambiente como Jupyter Notebook ou Python.

Ao chamar a função display, ela renderiza o DataFrame como uma tabela formatada, facilitando a visualização e análise dos dados. No exemplo fornecido, o DataFrame possui várias colunas (Date_, team_1, team_2, _map, result_1, result_2, map_winner, starting_ct, ct_1, t_2, t_1, ct_2, event_id, match_id, rank_1, rank_2, map_wins_1, map_wins_2, match_winner) e 91546 linhas de dados.

A exibição do DataFrame permite visualizar os dados tabulares de forma organizada, com os nomes das colunas na parte superior e os valores correspondentes em cada linha. Isso facilita a análise,

filtragem, ordenação e visualização dos dados contidos no DataFrame, proporcionando uma representação visual dos dados de maneira eficiente.

- Informações da tabela:

```
In [6]: df_results.info() #Verificar as informações sobre as variáveis

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 91546 entries, 0 to 91545
Data columns (total 19 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   Date_         91546 non-null  object
 1   team_1        91546 non-null  object
 2   team_2        91546 non-null  object
 3   _map          91546 non-null  object
 4   result_1      91546 non-null  int64 
 5   result_2      91546 non-null  int64 
 6   map_winner    91546 non-null  object
 7   starting_ct   91546 non-null  object
 8   ct_1          91546 non-null  int64 
 9   t_2           91546 non-null  int64 
 10  t_1           91546 non-null  int64 
 11  ct_2          91546 non-null  int64 
 12  event_id      91546 non-null  int64 
 13  match_id      91546 non-null  int64 
 14  rank_1        91546 non-null  int64 
 15  rank_2        91546 non-null  int64 
 16  map_wins_1    91546 non-null  int64 
 17  map_wins_2    91546 non-null  int64 
 18  match_winner  91546 non-null  object
dtypes: int64(12), object(7)
memory usage: 13.3+ MB
```

A função df_results.info() é usada para exibir informações sobre um DataFrame no pandas. Aqui está uma descrição dos resultados fornecidos:

A linha class 'pandas.core.frame.DataFrame' indica que estamos trabalhando com um objeto DataFrame.

A linha RangeIndex: 91546 entries, 0 to 91545 informa que o DataFrame possui 91546 linhas no intervalo de índices de 0 a 91545.

A seção Data columns lista todas as colunas do DataFrame e fornece informações adicionais sobre cada uma delas:

Column: Nome da coluna.

Non-Null Count: Número de valores não nulos na coluna, o que indica a quantidade de dados disponíveis.

Dtype: Tipo de dados da coluna.

A linha memory usage: 13.3+ MB mostra a quantidade de memória usada pelo DataFrame.

No exemplo específico fornecido, o DataFrame df_results possui 19 colunas com uma variedade de tipos de dados, incluindo inteiros (int64) e objetos (object). Existem 91546 entradas no DataFrame, o que indica que cada coluna possui 91546 valores não nulos. A função df_results.info() é útil para obter uma visão geral dos dados presentes no DataFrame, incluindo o número de linhas, tipos de dados das colunas e a presença de valores nulos.

- Verificando a presença de valores ausentes:

```
In [7]: display(df_results.isna().sum()) #Verificar a presença de valores ausentes.
```

```
Date_          0
team_1         0
team_2         0
_map           0
result_1       0
result_2       0
map_winner     0
starting_ct    0
ct_1           0
t_2            0
t_1            0
ct_2           0
event_id       0
match_id       0
rank_1         0
rank_2         0
map_wins_1     0
map_wins_2     0
match_winner   0
dtype: int64
```

A função display(df_results.isna().sum()) é usada para verificar a presença de valores ausentes (ou nulos) em um DataFrame do pandas. Os resultados exibidos indicam o número de valores ausentes em cada coluna do DataFrame df_results.

Aqui está uma descrição dos resultados fornecidos:

Cada linha representa uma coluna do DataFrame df_results.

O nome da coluna é exibido à esquerda, seguido pelo número de valores ausentes naquela coluna.

A última linha, dtype: int64, indica o tipo de dados dos valores contados (neste caso, são inteiros).

No exemplo fornecido, os resultados mostram que não há valores ausentes em nenhuma das colunas do DataFrame df_results. Todos os campos possuem contagem de valores nulos igual a zero, indicando que não há dados faltantes no conjunto de dados. Isso é útil para garantir que os dados estejam completos antes de realizar análises ou manipulações adicionais.

- Agrupando dados:

```
In [17]: df_results.groupby('_map').Date_.count()
```

```
Out[17]: _map
         Cache          9226
         Cobblestone    7026
         Default          42
         Dust2          8228
         Inferno       14970
         Mirage        18042
         Nuke           8412
         Overpass      11250
         Train         13132
         Vertigo        1218
         Name: Date_, dtype: int64
```

A expressão df_results.groupby('_map').Date_.count() está sendo usada para agrupar os dados do DataFrame df_results pela coluna '_map' e, em seguida, contar o número de ocorrências da coluna Date_ em cada grupo. O resultado é uma série que exibe o número de ocorrências de cada valor único na coluna '_map'.

Aqui está uma descrição dos resultados fornecidos:

Cada linha representa um valor único da coluna '_map'.

Os valores únicos da coluna '_map' são exibidos na coluna do índice.

A coluna à direita, intitulada Name: Date_, dtype: int64, indica que estamos contando o número de ocorrências da coluna Date_.

Os valores numéricos representam o número de ocorrências de cada valor único na coluna '_map'.

No exemplo fornecido, os resultados mostram a contagem de ocorrências para cada valor único na coluna '_map' do DataFrame df_results. Por exemplo, temos 9226 ocorrências do valor 'Cache', 7026 ocorrências do valor 'Cobblestone', 42 ocorrências do valor 'Default' e assim por diante para os demais valores únicos. Essa informação é útil para analisar a distribuição dos dados em relação aos diferentes valores da coluna '_map'.

- Mapa seja mais favorável para CT

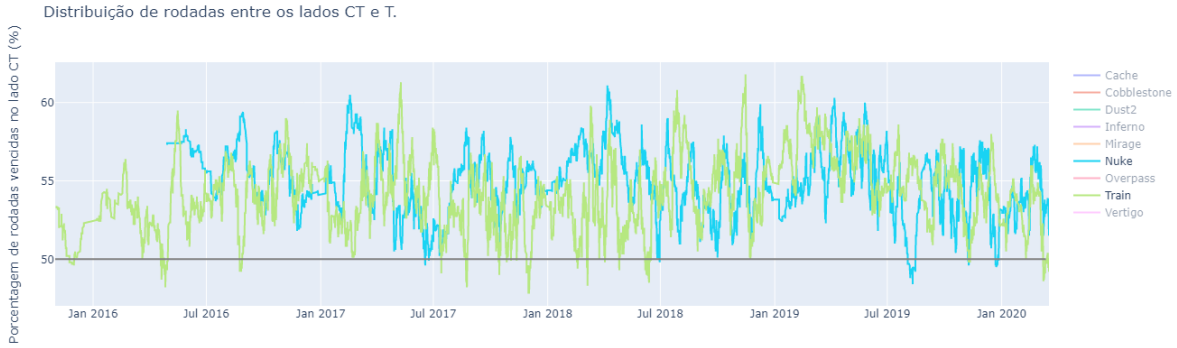
Para analisar qual dos mapas é mais favoráveis para CT(Counter Terrorist), determinei essa característica calculando as pontuações médias obtidas em cada lado do mapas e, em seguida, comparando ambos os lados.

In [19]:

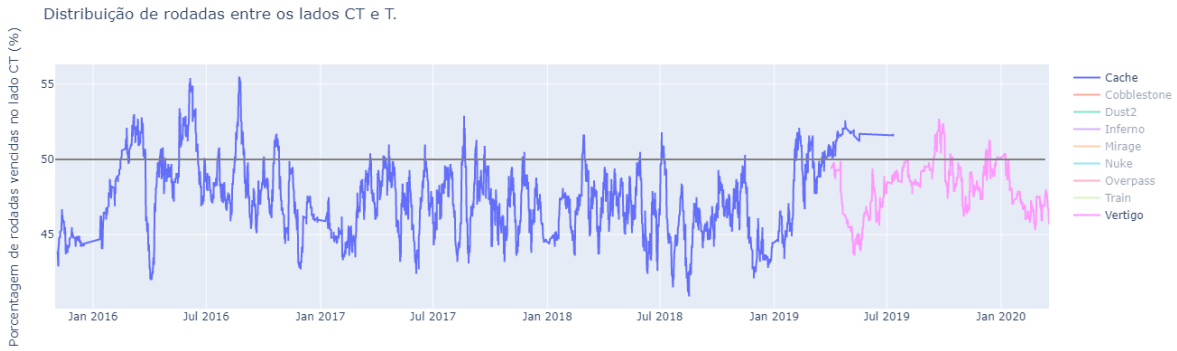
```
maps = ['Cache','Cobblestone','Dust2','Inferno','Mirage','Nuke','Overpass','Train','Vertigo']

ct_1 = df_results[['Date_','_map','ct_1']].rename(columns={'ct_1':'ct'})
ct_2 = df_results[['Date_','_map','ct_2']].rename(columns={'ct_2':'ct'})

ct = pd.concat((ct_1, ct_2))

t_1 = df_results[['Date_','_map','t_1']].rename(columns={'t_1':'t'})
t_2 = df_results[['Date_','_map','t_2']].rename(columns={'t_2':'t'})

t = pd.concat((t_1, t_2))

t = t.sort_values('Date_')
ct = ct.sort_values('Date_')

series_t, series_ct, how_ct = {},{},{}

for i, key in enumerate(maps):
    t_map = t[t._map == maps[i]]
    ct_map = ct[ct._map == maps[i]]
    y_t = t_map.t.rolling(min_periods = 20, window= 200, center=True).sum().values
    y_ct = ct_map.ct.rolling(min_periods = 20, window= 200, center=True).sum().values

    series_t[key] = pd.Series(data=y_t,index=t_map.Date_)
    series_ct[key] = pd.Series(data=y_ct,index=ct_map.Date_)

    how_ct[key] = series_ct[key]/(series_ct[key]+series_t[key])//0.001/10

fig = go.Figure()

for _map in maps:
    fig.add_trace(go.Scatter(x=how_ct[_map].index, y=how_ct[_map].values, name=_map))

fig.add_trace(go.Scatter(x=['2015-11-01', '2020-03-12'], y=[50,50], mode='lines',line=dict(color='grey'),showlegend=False))
fig.update_layout(title='Distribuição de rodadas entre os lados CT e T.', yaxis_title='Porcentagem de rodadas vencidas no l
fig.show()
```

Existem longos períodos sem dados para um mapa no gráfico. Isso ocorre porque os mapas são adicionados e removidos constantemente pelos administradores do jogo.

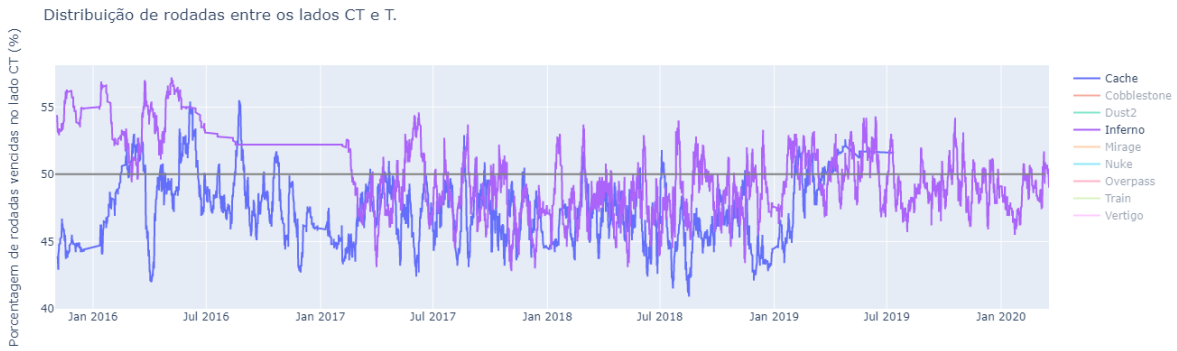
Observando os gráficos o Nuke e Train oscilam como sendo os mapas mais favoráveis ao lado CT, tendo uma aproximação de 57% das rodadas vencidas pelo lado CT, enquanto Dust2 e Cache são historicamente os mapas mais favoráveis ao lado T.




interessante em 2019 no mês abril o mapa Vertigo entrou no jogo tendo somente 4 meses, após o Cache foi removido do jogo.




É interessante observar que o Inferno era conhecido por ser um mapa fortemente favorável ao lado CT antes de 2016, o que foi uma das razões para sua atualização. Desde sua atualização, Inferno tem sido na verdade o mapa mais equilibrado nesse aspecto.



- Mapas mais removidos das partidas:

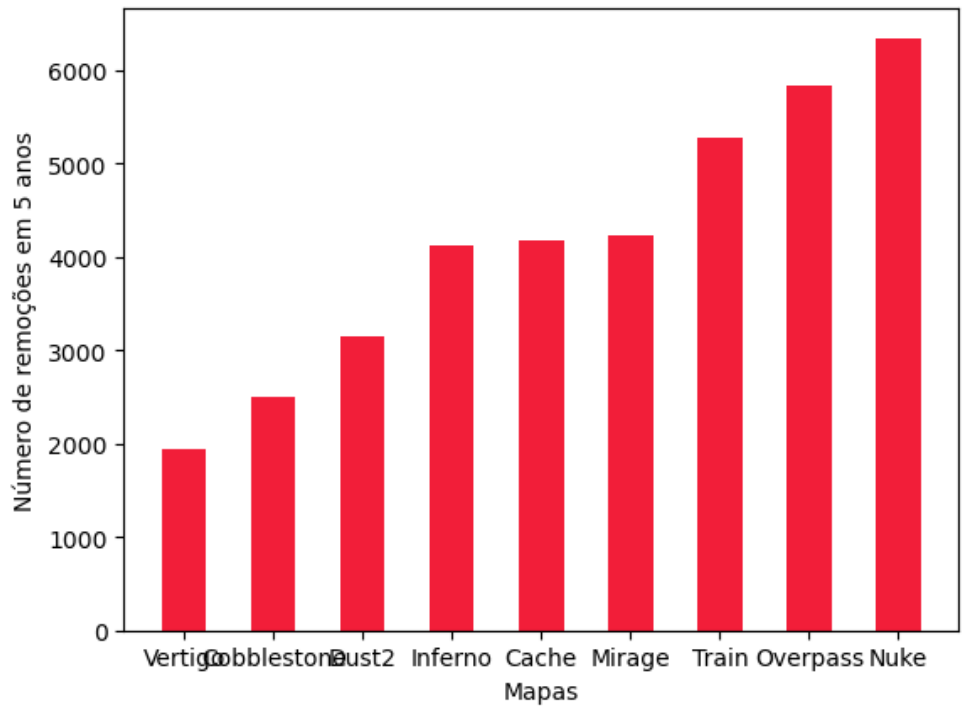
```
In [4]: df_removidos = df_picks[['t1_removed_1', 't1_removed_2','t1_removed_3', 't2_removed_1', 't2_removed_2','t2_removed_3']]

In [5]: df_removidos
```

Out[5]:

| | t1_removed_1 | t1_removed_2 | t1_removed_3 | t2_removed_1 | t2_removed_2 | t2_removed_3 |
|---|---|---|---|---|---|---|
| 0 | Vertigo | Train | 0.0 | Nuke | Overpass | 0.0 |
| 1 | Dust2 | Nuke | 0.0 | Mirage | Train | 0.0 |
| 2 | Mirage | Dust2 | Vertigo | Nuke | Train | Overpass |
| 3 | Inferno | Nuke | 0.0 | Overpass | Vertigo | 0.0 |
| 4 | Train | Mirage | 0.0 | Nuke | Inferno | 0.0 |
| ... | ... | ... | ... | ... | ... | ... |
| 16030 | Dust2 | Cobblestone | Mirage | Cache | Inferno | Overpass |
| 16031 | Inferno | Train | Mirage | Overpass | Cobblestone | Cache |
| 16032 | Dust2 | Cache | Inferno | Train | Overpass | Cobblestone |
| 16033 | Overpass | Cobblestone | Cache | Dust2 | Inferno | Mirage |
| 16034 | Cache | Train | Inferno | Overpass | Mirage | Dust2 |

16035 rows × 6 columns

```
In [6]: # conta as remoções dos mapas na equipe 1
        t1_removed_counts = df_picks['t1_removed_1'].value_counts() + \
                            df_picks['t1_removed_2'].value_counts() + \
                            df_picks['t1_removed_3'].value_counts()

In [7]: t1_removed_counts_sort = t1_removed_counts.sort_values()

In [8]: # plota o gráfico de linhas
        plt.bar(t1_removed_counts_sort.index, t1_removed_counts_sort.values, width=0.5, color='#f21e39')

        # configura os rótulos dos eixos x e y
        plt.xlabel('Mapas')
        plt.ylabel('Número de remoções em 5 anos')

        # exibe o gráfico
        plt.show()
```



Neste gráfico, vemos os mapas mais removidos nas escolhas de mapas de cada partida, entre os anos de 2015-2020. Porém, vale lembrar, segundo o gráfico de comparações dos mapas mais escolhidos, que é possível notar que o mapa Vertigo, começou a ser jogado em -, e o Cobblestone foi retirado das escolhas para sua

- Desempenho dos melhores jogadores de cada ano:

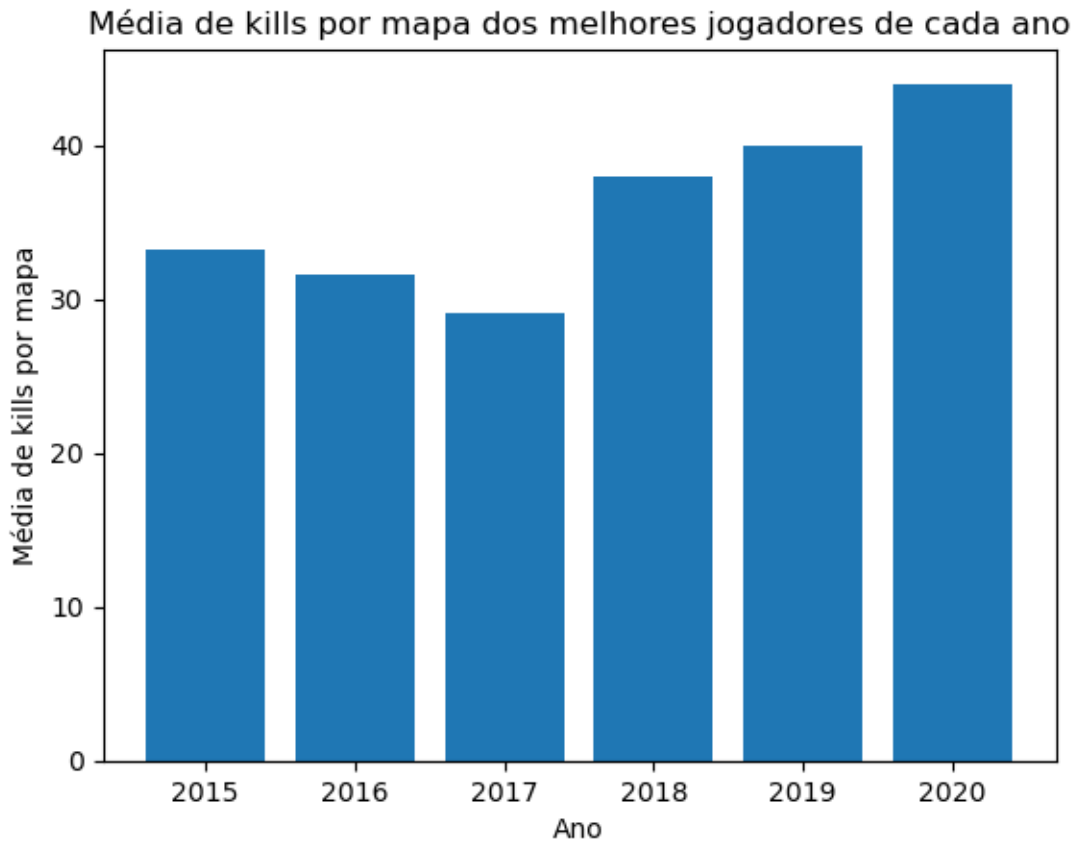
De acordo com a matéria do GE (https://ge.globo.com/esports/csgo/noticia/2022/08/21/10-anos-de-csgo-relembre-os-melhores-de-cada-ano.ghtml), os melhores jogadores de cada ano do CSGO competitivo são:

- 2015 - olofmeister
- 2016 - coldzera
- 2017 - coldzera
- 2018 - s1mple
- 2019 - ZywOo
- 2020 - ZywOo

Com isso, iremos analisar a performance individual de cada jogador nos seus respectivos anos, comparando um com o outro para analisarmos as diferenças de cada ano.

In [18]:
```
# Adicionar coluna de ano
df_players['year'] = pd.DatetimeIndex(df_players['date']).year

# Filtrar dados para anos desejados
start_year = 2010
end_year = 2020
df_players_filtered = df_players[(df_players['year'] >= start_year) & (df_players['year'] <= end_year)]
```

In [19]:
```
# Selecionando os dados dos jogadores específicos
df_olof = df_players.loc[(df_players['player_name'] == 'olofmeister') & (df_players['year'] == 2015)]
df_cold1 = df_players.loc[(df_players['player_name'] == 'coldzera') & (df_players['year'] == 2016)]
df_cold2 = df_players.loc[(df_players['player_name'] == 'coldzera') & (df_players['year'] == 2017)]
df_s1mple = df_players.loc[(df_players['player_name'] == 's1mple') & (df_players['year'] == 2018)]
df_zywoo1 = df_players.loc[(df_players['player_name'] == 'ZywOo') & (df_players['year'] == 2019)]
df_zywoo2 = df_players.loc[(df_players['player_name'] == 'ZywOo') & (df_players['year'] == 2020)]

# Calculando a média de pontos por mapa em cada ano
olof_mean = df_olof['kills'].mean()
cold1_mean = df_cold1['kills'].mean()
cold2_mean = df_cold2['kills'].mean()
s1mple_mean = df_s1mple['kills'].mean()
zywoo1_mean = df_zywoo1['kills'].mean()
zywoo2_mean = df_zywoo2['kills'].mean()

# Criando a lista com as médias de kills por mapa em cada ano
medias = [olof_mean, cold1_mean, cold2_mean, s1mple_mean, zywoo1_mean, zywoo2_mean]

# Criando a lista com os anos
anos = [2015, 2016, 2017, 2018, 2019, 2020]

# Criando o gráfico de barras
plt.bar(anos, medias)

# Adicionando título e labels aos eixos
plt.title('Média de kills por mapa dos melhores jogadores de cada ano')
plt.xlabel('Ano')
plt.ylabel('Média de kills por mapa')

# Exibindo o gráfico
plt.show()
```




Temos alguns pontos a se observar:

Um mapa tem 30 rounds.

Apenas Coldzera em 2017 fez uma media menor do que 30 kills por partida no ano em que foi o melhor jogador. O restante dos jogadores fizeram uma media de no minimo 1 kill por round do mapa, que é um dado muito alto e dificil de se alcançar.

Com este gráfico podemos ver como houve um aumento entre os anos de 2018 e 2020 na média dos kills por mapa entre os melhores daquele respectivo ano. Aumentando ainda mais as espectativas para os próximos anos.

- Maiores equipes vencedoras:

In [4]:

```
# Gera dois novos DataFrames para os times das colunas "team_1" e "team_2"
df1 = pd.DataFrame(df, columns=['team_1', 'map_wins_1', 'map_wins_2'])
df2 = pd.DataFrame(df, columns=['team_2', 'map_wins_1', 'map_wins_2'])

# Novo dataframe que compara as colunas "map_wins_1" e "map_wins_2" para saber em quais casos o time 1 ganhou.
team_1_w = pd.DataFrame(df1.query('map_wins_1 > map_wins_2'), columns=['team_1'])

# Após filtrar as linhas, uma nova variável DataFrame gera uma tabela com o quantitativo de vitórias de cada
# time da coluna "team_1".
count_df1 = team_1_w['team_1'].value_counts().reset_index().rename(columns={'index': 'team_1', 'team_1': 'count'})
display(count_df1)
```

| | team_1 | count |
|---|---|---|
| 0 | Astralis | 376 |
| 1 | Liquid | 353 |
| 2 | TYLOO | 344 |
| 3 | Natus Vincere | 314 |
| 4 | Tricked | 310 |
| ... | ... | ... |
| 901 | Magistra | 1 |
| 902 | walkover | 1 |
| 903 | 1UP | 1 |
| 904 | ex-Mentality | 1 |
| 905 | Dynasty | 1 |

906 rows × 2 columns

In [5]:

```
# Realiza as mesmas operações do notebook acima, só que dessa vez, com a coluna "team_2"
team_2_w = pd.DataFrame(df2.query('map_wins_1 < map_wins_2'), columns=['team_2'])
count_df2 = team_2_w['team_2'].value_counts().reset_index().rename(columns={'index': 'team_2', 'team_2': 'count'})
display(count_df2)
```

| | team_2 | count |
|---|---|---|
| 0 | mousesports | 277 |
| 1 | Liquid | 266 |
| 2 | G2 | 262 |
| 3 | fnatic | 234 |
| 4 | Spirit | 229 |
| ... | ... | ... |
| 931 | Wizards | 1 |
| 932 | Normal People and Rustun | 1 |
| 933 | Shadows | 1 |
| 934 | Operation Kino | 1 |
| 935 | TEAM1231 | 1 |

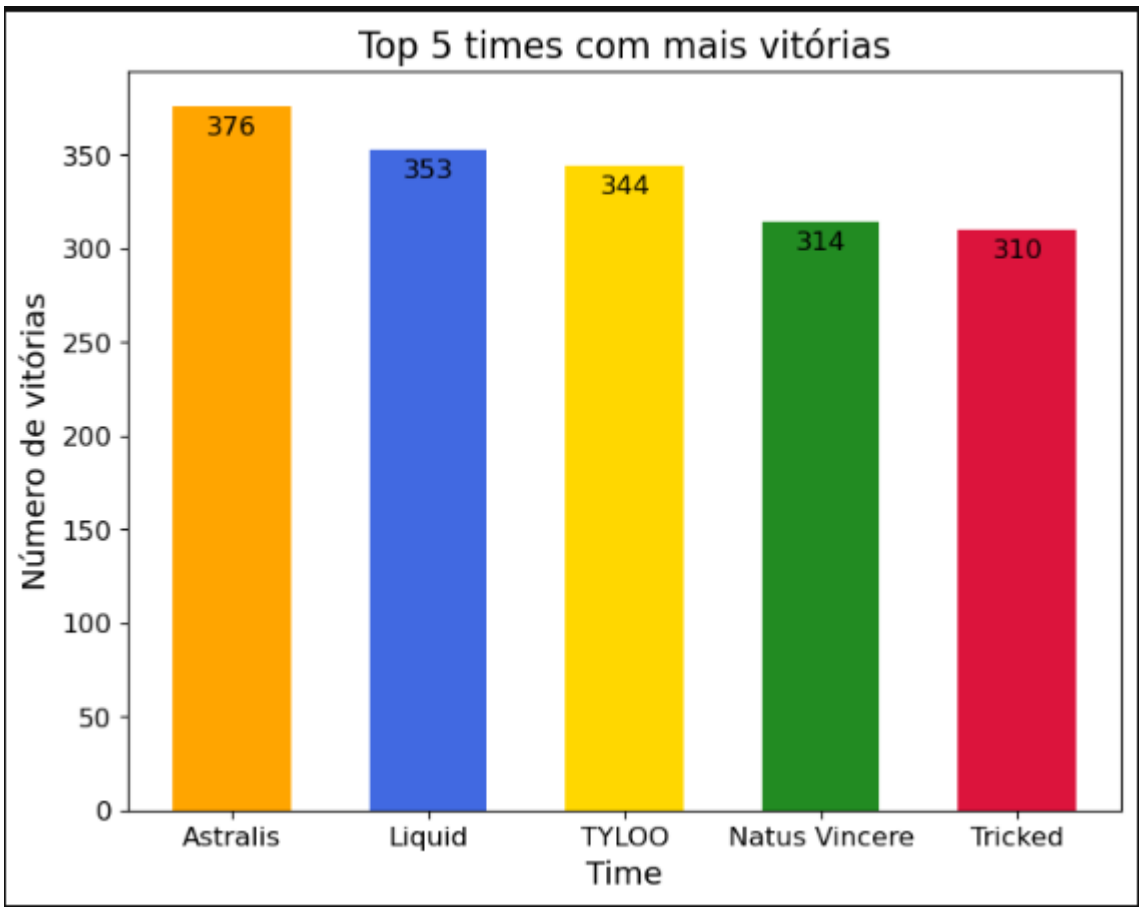
In [6]:

```
# Concatenar as colunas "team_1" e "team_2" e somar as contagens de vitórias
count_df = pd.concat([count_df1, count_df2], axis=0).groupby('team_1').agg({'count': 'sum'}).reset_index()

# Renomear as colunas do novo DataFrame
count_df.columns = ['team', 'count']

# Somar as contagens de vitórias para cada time
count_df = count_df.groupby('team').agg({'count': 'sum'}).reset_index()

# Ordenar os times pelo número de vitórias
sorted_count_df = count_df.sort_values(by='count', ascending=False)

# Exibir o DataFrame ordenado
display(sorted_count_df)
```

| | team | count |
|---|---|---|
| 72 | Astralis | 376 |
| 418 | Liquid | 353 |
| 696 | TYLOO | 344 |
| 482 | Natus Vincere | 314 |
| 718 | Tricked | 310 |
| ... | ... | ... |
| 193 | Downfall | 1 |
| 66 | ArkAngel | 1 |
| 775 | WinOut | 1 |
| 637 | Saltbae | 1 |
| 905 | zzz | 1 |

In [15]:

```
fig, ax = plt.subplots(figsize=(8,6))
bar_container = ax.bar(sorted_count_df['team'].head(5), sorted_count_df['count'].head(5), color=['#FFA500', '#4169E1', '#FF
ax.set_title('Top 5 times com mais vitórias', fontsize=16)
ax.set_xlabel('Time', fontsize=14)
ax.set_ylabel('Número de vitórias', fontsize=14)
ax.tick_params(axis='both', labelsize=12)
for i, val in enumerate(sorted_count_df['count'].head(5)):
    ax.text(i, val-15, str(val), ha='center', fontsize=12)
plt.show()
```




O código apresentado utiliza a biblioteca matplotlib.pyplot para criar um gráfico de barras horizontal que representa os top 5 times com mais vitórias. Vamos analisar cada linha do código:

fig, ax = plt.subplots(figsize=(8,6)): Cria uma figura (fig) e um conjunto de eixos (ax) para o gráfico. Define também o tamanho da figura como 8 polegadas de largura por 6 polegadas de altura.

bar_container = ax.bar(sorted_count_df['team'].head(5), sorted_count_df['count'].head(5), color=['#FFA500', '#4169E1', '#FFD700', '#228B22', '#DC143C'], width=0.6): Cria um gráfico de barras utilizando os dados do DataFrame sorted_count_df. O eixo x é preenchido com os nomes dos times (sorted_count_df['team'].head(5)) e o eixo y é preenchido com o número de vitórias correspondente a cada time (sorted_count_df['count'].head(5)). O parâmetro color define as cores das barras e o parâmetro width define a largura das barras.

ax.set_title('Top 5 times com mais vitórias', fontsize=16): Define o título do gráfico como "Top 5 times com mais vitórias" e define o tamanho da fonte do título como 16.

ax.set_xlabel('Time', fontsize=14): Define o rótulo do eixo x como "Time" e define o tamanho da fonte do rótulo como 14.

ax.set_ylabel('Número de vitórias', fontsize=14): Define o rótulo do eixo y como "Número de vitórias" e define o tamanho da fonte do rótulo como 14.

ax.tick_params(axis='both', labelsize=12): Define os parâmetros dos ticks (marcas) nos eixos. Neste caso, define o tamanho da fonte dos ticks como 12.

for i, val in enumerate(sorted_count_df['count'].head(5)):: Cria um loop que percorre os valores das vitórias dos top 5 times (sorted_count_df['count'].head(5)).

ax.text(i, val-15, str(val), ha='center', fontsize=12): Adiciona um texto no gráfico próximo a cada barra representando o valor das vitórias correspondente. O parâmetro i representa a posição da barra, val-15 define a posição vertical do texto e str(val) converte o valor para string. O parâmetro ha='center' centraliza o texto horizontalmente e fontsize=12 define o tamanho da fonte do texto.

plt.show(): Exibe o gráfico de barras.

Em resumo, esse código cria um gráfico de barras horizontal que representa os top 5 times com mais vitórias. Cada barra representa um time e seu tamanho é proporcional ao número de vitórias. O gráfico é acompanhado por um título, rótulos nos eixos e valores das vitórias próximos às barras.

Alem de Esse código ser uma coleção de funções relacionadas à conexão e consulta a um banco de dados MySQL.

A função connect recebe os parâmetros host, database, user e password para estabelecer a conexão com o banco de dados. Ela utiliza a biblioteca mysql.connector para conectar ao servidor MySQL. Se a conexão for bem-sucedida, retorna o objeto de conexão. Caso ocorra algum erro, trata as exceções e exibe mensagens adequadas para os erros ER_BAD_DB_ERROR (banco de dados inexistente) e ER_ACCESS_DENIED_ERROR (usuário ou senha incorretos).

A função execute_query recebe a conexão ao banco de dados, uma consulta SQL em formato de string (query) e opcionalmente dados para serem passados como parâmetros para a consulta (data). Essa função executa a consulta utilizando um cursor e retorna o resultado da consulta. Se data for fornecido, a consulta é executada com os parâmetros. O resultado da consulta é retornado e a conexão é confirmada (commit). Se ocorrer algum erro na execução da consulta, a função retorna uma mensagem de erro formatada. A conexão é fechada no bloco finally para garantir que a conexão seja encerrada independentemente de ter ocorrido algum erro ou não.

A função loop_result recebe uma lista de resultados e itera sobre ela, convertendo cada resultado em uma lista e adicionando-a a uma lista from_db. Em seguida, verifica se a lista from_db não é vazia e a retorna. Caso contrário, retorna None.

Essas funções são úteis para estabelecer uma conexão com um banco de dados MySQL, executar consultas e processar os resultados. Elas fornecem uma abstração conveniente para realizar operações comuns em um banco de dados MySQL, como conexão, consulta e processamento dos resultados.