# Documentação TP1

Gabriel Bordoni (2018050715)

Algoritmos I
Universidade Federal de Minas Gerais

24 de novembro de 2021

## 1 Resumo do Problema

A Black Friday é uma das datas mais esperadas pelo varejo ao longo do ano devido ao seu grande apelo comercial. Especialmente nesse ano, devido a um reaquecimento do mercado após meses de pandemia, essa data se tornou ainda mais relevante e, portanto, houve a necessidade de que o comercio se preparasse para aproveitar os consumidores da melhor forma possível.

E pensando nessa oportunidade, uma grande empresa do varejo decidiu contratar uma equipe de desenvolvedores que alocasse possíveis clientes de uma região a suas lojas de maneira que obtessem o melhor aproveitamento de vendas possíveis. Nessa alocação, além de tentar esgotar todo o estoque que uma determinada loja poderia ter, existia a necessidade de se preocupar com o fato de que cada loja deveria vender seus produtos a partir de uma fila de prioridades dos clientes, baseados em seus tickets - equação 1, e os clientes por sua vez não aceitariam fazer suas compras longe de casa se assim sua prioridade não os obrigassem.

$$ticket = \frac{|60 - \text{Idade}| + \text{ScoreEstado}}{\text{ScorePagamento}} \quad (1)$$

O ScoreEstado e o ScorePagamento são dois valores fornecidos pela empresa, tabela 1 e 2 que são referentes tanto a forma de pagamento do cliente quanto ao estado que ele reside.

| Estado | Score |
|---|---|
| MG | 0 |
| PR | 10 |
| SP | 20 |
| SC | 30 |
| RJ | 40 |
| RN | 50 |
| RS | 60 |

Tabela 1: Score por estado onde o cliente possa residir.

| Tipo de Pagamento | Score |
|---|---|
| DINHEIRO | 1 |
| DEBITO | 2 |
| CREDITO | 3 |

Tabela 2: Score por tipo de pagamento do cliente.

Além dos requisitos já citados, a empresa contratante ainda determinou que em caso de empate nos critérios de casamento comentados, o desempate deveria ser feito com base nos ids - index de entrada no sistema - da loja e cliente, que deveriam ser o menor possível.

## 2 Modelagem e Implementação

Antes de se começar a resolver o problema do casamento de fato, a primeira preocupação que nós temos é com respeito ao armazenamento dos dados. Para isso, foram-se criado uma lista para os clientes e uma para as lojas, com alocação de memória de tamanho proporcional ao indicado pelo arquivo de entrada.

Essas listas, a partir do momento que craidas, começavam a ser preenchidas de acordo com que a iteração sobre as linhas do arquivo progrediam. Por meio da funcão de inserção presente nas listas, a cada nova loja criada ela era armazenada na lista com seu id, localização, quantidade de estoque, estoque já vendido e uma lista de ids dos clientes para quem vendeu. Os clientes por sua vez, ao serem criados eram armazenados na lista com um id, uma localização, seu ticjet e um id de loja a qual foi designado já estanciados.

Com o fechamento do arquivo e todas as listas montadas, então começamos de fato com o algoritmo de casamento. A base de toda a lógica utilizada foi pautada em primeiramente entender que, como a lista de prioridade para cada uma das lojas seriam a mesma, pois ambas querem vender para o cliente de maior ticket, uma boa aproximação para o problema seria aquele em que todas a lojas fizessem uma proposta para o cliente de maior ticket disponível com este sendo livre para aceitar a primeira proposta de loja que fosse mais próxima à ele.

Logo, tendo o objetivo traçado, o primeiro desafio encontrado foi o de encontrar o melhor algoritmo de ordenação para o problema. Tendo como desejo obter um algoritmo que fosse eficiente e também estável, já que isso garantiriamos que a ordem dos ids entre clientes de mesmo tickets seriam mantidas obedecendo então o critério de desempate, a escolha final de implementação foi pelo MergeSort. Esse algoritmo é baseado no paradigma de dividir para conquistar e portanto obtém a ordenação pelo remanejamento de itens a partir de lis-

tas menores advindas da divisão da lista original. Uma breve esquematização desse algoritmo é mostrada no box de **Algorithm 1** e mostra como foi-se arquitetado a ordenação nessa solução, enfatizando que, diferentemente de outras implementações, aqui queremos que a lista saia em ordem decrescente.

**Algorithm 1** MergeSort

```
1:  procedure MERGE(clientes, inicio, meio, fim)
2:      esquerda ← clientes[inicio : meio];
3:      direita ← clientes[meio+1 : fim];
4:      i, j ← 0;
5:      k ← inicio;
6:      while i < meio+1 and j < fim - meio do
7:          if esquerda[i].ticket > direita[j].ticket then
8:              clientes_ordenados[k] ← esquerda[i];
9:              i++, k++;
10:         if not then
11:             clientes_ordenados[k] ← direita[j];
12:             j++, k++;
13:     while existir algum lado não vazio do
14:         clientes_ordenados ← resto;
15:     clientes ← clientes_ordenados
16: procedure MERGESORT(clientes, inicio, fim)
17:     if inicio < fim then
18:         meio ← (meio+fim)/2;
19:         recursive ← (clientes, inicio, meio)
20:         recursive ← (clientes, meio+1, fim)
21:         MERGE ← (clientes, inicio, meio, fim)
```

**Algorithm 2** StableMatcher

```
    procedure MATCH(lojas, clientes, grid_sides)
2:      for cliente in clientes do
            distancia_min ← max(grid_sides);
4:          for loja in lojas do
                if loja sem estoque then
6:                  continue;
                distancia ← dist(loja, cliente);
8:              if distancia < distancia_min then
                    distancia_min ← distancia;
10:                 cliente.designado_a ← loja;
            loja_designada ← cliente.designado_a
12:         loja_designada.clientes ← cliente;
            loja_designada.estoque −−;
```

Com a lista de clientes devidamente ordenada pelo valor de seus tickets, o que se foi feito a seguir foi o casamento entre as lojas e os clientes. Tal como mencionado, o que se foi feito foi, as lojas, em ordem de seus ids, faziam propostas ao clientes de maior ticket, que ainda não estivessem alocados, enquanto eles decidiam ou não optar pelo casamento. Daí, uma vez que o cliente recebesse uma proposta, ele só trocaria de loja se recebesse uma proposta de uma loja mais perto. Logo, tendo isso em conta, teremos que ao final de cada iteração uma loja sempre ficará com o cliente de maior ticket disponível ao mesmo tempo que este não trá intenção de trocar-la por outra, pois não existirá outra mais próxima, o que nos leva a que no final de cada iteração de porpostas por clientes temos a conclusão de um novo casamento estável, e por fim, ao final de iterar sobre todos os clientes, teremos uma solução estável, além de ótima do ponto de vista das lojas. Um esboço da implementação em pseudo-código como forma de exemplificação pode ser analisado no box de **Algorithm 2**.

# 3 Complexidade

Desconsiderando dá análise o custo de buscar os dados no arquivo de entrada, teremos que a ordem complexidade final do algoritmo será puxada pela função do *main* que houver o maior custo. Se levamos em conta que essa função é composta pelas iterações de iserções, a função de ordenação e a de casamento, temos que para chegar a uma conclusão final antes teremos que analisar cada um dos procedimentos separadamente.

Primeiramente, ao analisarmos as inserções, vemos que suas complexidades no tempo devem ser respectivamente **O**(m) para os clientes e **O**(n) para as lojas. Isso devido ao fato de que, realizar a inserção de uma loja ou um cliente em si é de fato **O**(1), mas como temos que fazer a mesma operação $m$ e $n$ vezes, a complexidade da inserção passam a tomar aspectos lineares.

O MergeSort por sua vez, sendo um algoritmo do paradigma de dividir para conquistar, tem uma ordem de complexidade dada por **O**(m.log m). Além disso, um ponto negativo desse algoritmo que o difere de outros métodos é que esse trás um prejuízo em relação memória, como ele tem que ficar alocando novas listas sempre que realiza uma processamento, isso poderia significar um alerta para o caso onde memória fosse algo limitado - o que acredito não ser o caso deste problema. De toda forma, por mais que o MergeSort tenha seu lado negativo, a propriedade de ser estável foi mais que suficiente para que optassemos por sua utilização.

Por fim, na sequência do *main*, o que temos então é a utilizacão do algoritmo de casamento que tem sua ordem de complexide temporal dada por **O**(m.n). A justificativa para isso é que temos na implementação duas iterações, a mais externa iterando sobre todos os $m$ clientes que por De forma geral, e aceitando que a ordem de complexidade temporal do algoritmo é dada pela função de maior ordem de complexidade, teremos que a complexidade total da solução será **O**(m.n).

# 4 Compilação

A solução deste problema foi totalmente implementado em C++, considerando como padrão sua versão 14, e devidamente compilado sem detectação de falhas ao rodar *make* em um Ubuntu 16.04.12 de g++ versão 5.4.0 20160609. Ao final da compilação será gerado um executável de nome *tp*01 e o mesmo pode ser utilizado com um arquivo de entrada tal como o seguinte exemplo: $ *tp*01 *nome_do_arquivo_de_entrada.txt*.