Documentação do primeiro Trabalho Prático da disciplina de Algoritmos I

Luis Gabriel Caetano Diniz – Matrícula: 2019075711

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG – Brazil

lgcaetano@ufmg.br

1. Modelagem computacional do problema

O problema proposto no trabalho consistia no desenvolvimento de um sistema que alocasse pacientes aos postos de vacinação disponíveis, sendo que há preferência para pessoas de maior idade e para postos mais próximos da residência de determinado paciente.

O programa recebe um arquivo contendo a quantidade de vagas disponíveis em cada posto de vacinação, a idade de cada paciente, e a localização de cada membro de ambas as categorias. Foram criadas estruturas de dados que representam tanto pessoas, quanto postos de vacinação para auxiliar no desenvolvimento da solução.

Após ler e armazenar as informações disponibilizadas no arquivo de entrada, o algoritmo aloca as pessoas de acordo com as especificações do trabalho e imprime os pacientes que foram alocados para cada posto de vacinação.

2. Estruturas de Dados e explicação do algoritmo

2.1 Classes e estruturas utilizadas

Como dito anteriormente, foram criadas duas classes para representar as pessoas e postos, as classes são denominadas **Pessoa** e **Posto**, respectivamente. Além delas, uma outra estrutura chamada **PostoDistancia** foi criada para auxiliar na solução.

A classe Pessoa possui diversos métodos e atributos, incluindo dois métodos construtores, dois métodos **get** que retornam a idade e o identificador da pessoa, um **vector** denominado **postosPreferidos** que contém instâncias da classe **PostoDistancia**, onde estão disponíveis os identificadores de cada posto de vacinação junto de suas respectivas distâncias em relação à residência do paciente, contém um método denominado **inserirPosto**, que recebe um posto de vacinação como parâmetro e insere seu identificador e a sua distância em relação ao paciente no **vector postosPreferidos**, um método chamado **popPostoPreferido**, que remove o último elemento do **vector** (que se espera ser o posto mais próximo do paciente, desde que o **vector** se encontre ordenado) e retorna o seu identificador, o método **ordenarPostosPreferidos**, que evidentemente ordena o **vector**, o método **alocarPosto**, que atribui ao atributo **idPostoAlocado** o identificador do posto no qual a pessoa foi alocada, contém também o método **pessoaJaAlocada**, que evidentemente indica que a pessoa foi alocada para algum posto e, finalmente, possui o método **postosEsgotados**, que indica se ainda há alguma instância da classe **postoDistancia** no **vector postosPreferidos.** Há também nesta classe, uma sobrecarga do operador “<” (“menor que”), o que é necessário para que possamos usar o método **sort**, disponibilizado pela própria linguagem **C++,** para

ordenar um array de instâncias da classe Pessoa de acordo com a idade, algo que encontramos no nosso **main.** O operador funciona da forma esperada, por exemplo se **Ana** e **André** forem instâncias de **Pessoa**, **Ana** < **André** retornará **true** caso Ana seja mais nova que André e **false** caso ele seja mais novo que ela. No caso de ambos possuírem a mesma idade, o operador considera como "mais novo" aquele que possui maior identificador.

A classe **PostoDistancia** é bem simples e serve apenas para auxiliar a classe Pessoa, contendo como atributos apenas um número inteiro **id** para identificar a qual posto a instância da classe se refere, além de um número de ponto flutuante **distancia,** que evidentemente indica a distância ao paciente que contém dentro de si a instância de **PostoDistancia**. Isto pode parecer confuso, mas faz sentido devido ao fato de que a classe **PostoDistancia** só contém instâncias dentro de outras instâncias da classe **Pessoa.** Além destes dois, há, assim como na classe **Pessoa**, uma sobrecarga ao operador booleano "<", para que se ordene os **vectors** de instâncias da classe **PostoDistancia** que são encontrados em instâncias da classe **Pessoa** de acordo com a distância de cada um em relação à residência da pessoa, possibilitando a fácil determinação da ordem de preferência de cada pessoa por cada posto de vacinação. Contudo, a lógica da sobrecarga neste caso é invertida, com o operador "<" retornando **true** no caso de **a > b,** caso tenha o atributo **distancia** de **MAIOR** valor que **b**, já que quanto maior a distância, menor a preferência do paciente pelo posto, ou seja, o método **sort** ordena as instâncias de **PostoDistancia** de acordo com a preferência do paciente, garantindo assim que a lógica se mantenha similar à da ordenação da classe **Pessoa**, que também se ordena em termos de preferência.

Finalmente, temos a classe **Posto**, que, assim como a classe **Pessoa,** possui diversos métodos e atributos. Entre os atributos, temos **_x, _y, _vagas** e **_id,** que contém as posições geográficas em relação aos eixos, o número de vagas iniciais disponíveis no posto em questão e o identificador do posto, respectivamente. Além disso, encontramos dentro da classe um **vector** de inteiros chamado **idsPessoasAlocadas** que indica os identificadores de cada pessoa que já foi alocada para ser vacinada no posto e um inteiro **numPessoas** que indica o número de pessoas que estão alocadas ao posto. Além disso, a classe possui também diversos métodos como dois construtores, 5 métodos **get** convencionais que retornam os valores para os atributos da classe com exceção, evidentemente, do **vector**, além de um método chamado **getIdPessoaAlocada**, que recebe um inteiro **index** como parâmetro e retorna o id da pessoa contido na posição **index** do **vector.** Além disso, há também um método denominado **cheio** que retorna **true** caso hajam vagas ainda disponíveis no posto e **false** caso contrário, e o método **alocarPessoa** que recebe um inteiro como parâmetro (que se espera ser o identificador de um paciente) e adiciona-o ao **vector** contendo os identificadores das pessoas alocadas para aquele posto.

2.2 Explicando o algoritmo

Após receber os diversos postos de vacinação, o programa recebe os diversos pacientes para serem alocados, e cria instâncias da classe **Pessoa** para representar cada um. Após instanciá-los, o programa adiciona todos os postos de vacinação aos **vectors** pertencentes às instâncias da classe **Pessoa**, e ao final deste processo, ordena os **vectors.**

Assim, ao final do **loop** que lê e instancia as pessoas a serem alocadas, já se tem disponível as preferências dos pacientes com relação aos postos, como se cada paciente tivesse uma lista com todos os postos disponíveis para vacinação ordenados em função da distância deles em relação ao paciente. Após isso, temos que ordenar os pacientes em função de idade. A ordenação dos pacientes em função da idade nos garante que uma das condições exigidas no trabalho: “se uma pessoa não for alocada num posto mais próximo a ela, não deve haver naquele posto vagas não alocadas ou alocações de pessoas mais jovens que ela.” Como iremos alocar cada pessoa em ordem decrescente de idade, caso uma pessoa **x** não tenha sido alocada no seu posto de preferência, é impossível que haja uma pessoa mais jovem que ela alocada no posto, pois todas as pessoas que vieram antes dela possuíam igual ou maior idade.

Após ordenarmos os pacientes em ordem crescente de idade, percorremos o array de trás pra frente, do mais velho para o mais novo, e realizamos o processo de alocação. O processo é simples: se obtém o posto de preferência do paciente, caso ele possua vagas se aloca o paciente neste posto, caso contrário, este posto é removido do **vector postosPreferidos** do paciente (através do método **popPostoPreferido**) e se realiza o mesmo processo com o próximo posto de vacinação preferido, até que se aloque o paciente ou que se esgotem os postos, o que, devido ao paciente ter todos os postos disponíveis, só ocorre quando não há mais vagas em nenhum posto, e, portanto, o algoritmo para de alocar pacientes. Como os postos se esgotam apenas quando um paciente verificou que não há vaga em **NENHUM** posto de vacinação, se garante que uma das condições de instabilidade não será atendida: “Há uma pessoa sem alocação, mas ainda existem vagas em um dos postos.” A alocação só acaba quando todos os pacientes são alocados, ou quando se torna impossível alocar um deles, o que só ocorre quando não há mais vagas em nenhum posto. Também pode-se garantir que a outra condição de instabilidade nunca ocorrerá: “Uma pessoa p1 está alocada para um posto β, mas há um posto α mais próximo que possui vagas disponíveis ou está alocando uma pessoa p2 de menor idade.” Como sempre se tenta alocar uma determinada pessoa em postos ordenados em termos de menor distância relativa, pode-se garantir que, se uma pessoa p1 está alocada em um posto **x**, que não é o de maior proximidade, é devido aos postos mais próximos não possuírem vagas, e, como pessoas mais jovens só serão alocadas posteriormente, elas também não serão alocadas nestes postos, pois eles não terão vagas para ela também.

2.3 Pseudocódigo do algoritmo

Após ter lido o arquivo de entrada;

```
PARA1 i DE 0 ATÉ numeroDePessoas:
    PARA2 j DE 0 ATÉ numeroDePostos:

          pessoas[i].adicionarAoVetorDePostos(postosDeSaude
          [j]);
    FIM PARA2;
    OrdenarConformeADistância(pessoas[i].VetorDePostos);
FIM PARA1;

OrdenarDeAcordoComAIdade(pessoas);

PARA1 i de numeroDePessoas ATÉ 0:
    PARA2 j de numeroDePostos ATÉ 0:
        postoPreferido = pessoas[i].vetorDePostos[j]
        SE postoPreferido.contemVagas():
            Alocar pessoas[i] em postoPreferido;
            FIM PARA2;
        FIM SE;
    FIM PARA2;
    SE pessoas[i].naoFoiAlocada:
        FIM PARA1;
    FIM SE;
FIM PARA1;
```

3. Análise da complexidade assintótica

Com relação à quantidade de memória utilizada pelo programa, temos uma complexidade de O(m*n), onde m é o número de pacientes e n o número de postos de vacinação. Isso ocorre devido ao fato de termos um vetor contendo n elementos representando os postos de saúde em cada instância da classe **Pessoa**, e já que possuímos m instâncias desta classe, o programa acaba alocando m*n espaços de memória. Como não há alocações de ordens de grandeza maiores que esta, não há muito o que questionar sobre a complexidade assintótica espacial.

Já no caso da complexidade em relação ao tempo, temos duas ocorrências de loops aninhados, e em ambos os casos, um dos loops vai de 0 a m e o outro vai de 0 a n. Contudo, no primeiro caso de loops aninhados, temos a execução de um algoritmo de ordenação m vezes. Este algoritmo ordena um vetor com n elementos. Isso introduz um fator novo, já que algoritmos de ordenação sempre possuem complexidade de ordem maior que O(n), a não ser no melhor caso. Como o algoritmo utilizado é disponibilizado pela linguagem, temos de verificar na documentação da mesma a complexidade. De acordo com artigos encontrados na internet (fonte abaixo) o algoritmo **sort** é híbrido, ou seja, escolhe um dentre diversos algoritmos de ordenação de acordo com os parâmetros recebidos. Além disso, sua complexidade é estimada em todos os casos ser O(n log n).

Portanto, como estamos ordenando um vetor de n elementos m vezes com um algoritmo de complexidade O(n log n), nosso algoritmo tem ordem O(m*n log n) em todos os casos, já que o número de vezes em que o algoritmo é chamado não altera e sua complexidade em todos os casos é estimada em O(n log n). Como não há parte do algoritmo central que possua maior ordem de complexidade assintótica, esta é a resposta para o algoritmo como um todo.

Fonte: https://www.geeksforgeeks.org/internal-details-of-stdsort-in-c/