

**Atenção: para obter ajuda nesta prática (e nas seguintes) será necessário mostrar que fez algum rascunho/exemplo/etc em papel.**

## INF 213 - Roteiro da Aula Prática

Arquivos disponibilizados para a prática:

<https://drive.google.com/file/d/1iNz9Z3knfwZHDRuOdZprWRJPSWCMGqn7/view?usp=sharing>

### **Etapas**

#### **Etapas**

Na aula vimos ordenação de arrays de inteiros. Modifique o programa `ordenaStruct1.cpp` de modo que a função `insertionSort` seja capaz de ordenar arrays de structs "Pessoa".

Você deverá ordenar um array de `Pessoa` com base no nome (em ordem lexicográfica). Se duas pessoas tiverem o mesmo nome, elas deverão ser ordenadas com base no CPF (um número inteiro).

Como o método `insertion sort` **é estável**, uma forma de se fazer isso consiste em criar duas funções de ordenação (modificando a função `insertionSort` passada como exemplo): uma ordena com base no nome (`ordenaNome`) e outra com base no CPF (`ordenaCPF`). Então, você deverá ordenar o array utilizando uma das funções e, a seguir, a outra (qual deverá ser chamada primeiro?).

Obs: normalmente não implementamos os algoritmos de ordenação de modo que eles ordenem apenas com base em um critério: em vez disso criamos algoritmos mais genéricos e passamos para ele a função (ou *functor*), que deverá ser utilizada para fazer a comparação (porém, para não complicar a prática não faremos isso agora).

#### **Etapas**

Considere agora o programa `ordenaStruct2.cpp`. Você deverá modificar esse programa para que ele faça o mesmo que o programa da etapa anterior faz. Porém, ele utilizará o algoritmo `selectionSort` (que, conforme vimos em sala, não é estável!).

Para atingir esse objetivo, você deverá chamar o método de ordenação apenas uma vez. Esse método deverá tratar a possibilidade de desempate no próprio `if` que determina se um struct é menor do que o outro (ou seja, esse `if` deverá verificar se um nome é menor do que o outro e, no caso de empate, comparar os CPFs).

Em geral, mesmo se o algoritmo de ordenação for estável é melhor utilizar a estratégia desta etapa (pois a ordenação será feita apenas uma vez). Porém, em algumas situações um algoritmo estável é melhor (por exemplo, quando realmente é preciso fazer múltiplas ordenações)

### Etapa 3

Considere o programa buscaBinaria3.cpp. Nele, temos uma implementação de uma busca binária e de uma busca sequencial.

Ao pesquisar por um elemento  $X$  em um array já ordenado, ambas buscas retornam a posição desse elemento. Porém, se  $X$  não estiver no array a busca sequencial retorna a primeira posição do array contendo um elemento maior do que  $X$  (ou  $-1$  se não houver nenhum elemento maior).

A busca binária, por outro lado, retorna  $-1$  quando o elemento não está no array. Além disso, se procurarmos por um elemento que aparece de forma repetida no array ela retornará qualquer posição contendo esse valor (exemplo: se procurarmos por  $2$  no array  $[1, 2, 2, 2, 3]$  qualquer uma das 3 posições contendo  $2$  poderá ser retornada pela busca binária, enquanto a sequencial retornará sempre a primeira).

Sua tarefa será modificar o código da busca binária (modifique apenas a função buscaBin) para que ele retorne o mesmo que a busca sequencial retorna! (ou seja, a saída dos dois métodos deverá ser sempre igual!). Faça sempre um rascunho em papel e pense: ao acessar a posição do meio do array, o que podemos concluir se o elemento dessa posição for igual ao valor procurado, menor ou maior?

Se o valor do meio for a “chave” procurada, já se retorna ele de imediato. Se for menor ele continuará a busca na metade superior do array, já que ele está ordenado de forma crescente a chave só poderá ser encontrada nesta seção. Se for maior, buscará a chave entre o início e meio do array. Faz isso de forma recursiva poupando muito tempo de busca.

### Etapa 4

Considere o seguinte problema: você deseja assistir a um vídeo (por streaming) que possui tamanho de  $T$  segundos. Suponha que cada segundo do vídeo gaste  $B$  bytes para ser baixado e sua internet tenha uma velocidade de  $V$  bytes por segundo. Qual o tempo mínimo (vamos chama-lo de tempoCarregamento) que você deveria aguardar para conseguir assistir ao vídeo todo (sem que ele pause para baixar mais dados)?

Por exemplo, se  $B$   $V$   $T$  valem, respectivamente,  $10$ ,  $3$  e  $2$  você teria que aguardar  $5$  segundos para assistir ao vídeo. Nesses  $5$  segundos você teria baixado  $15$  bytes de dados. A seguir, assistiria a  $1$  segundo de vídeo (nesse tempo mais  $3$  bytes seriam baixados e você teria consumido  $10 \rightarrow$  assim, ao final desse segundo teria:  $15 + 3 - 10 = 8$  bytes para serem consumidos nos próximos segundos). No próximo (e último) segundo, você poderia baixar mais  $3$  bytes (totalizando  $11$  bytes baixados -- na verdade apenas  $10$ , já que só faltam  $10$  para acabar o vídeo) e consumiria  $10$ .

Se nesse caso você esperar  $6$  segundos  $\rightarrow$  também poderia assistir ao vídeo sem pausas.

Porém, se esperar apenas 4 segundos → após a espera teria 12 bytes. Conseguiria assistir ao primeiro segundo (baixando mais 3 bytes nesse tempo e consumindo 10 → vão sobrar 5 bytes). Porém, no próximo segundo de vídeo você baixaria 3 bytes (totalizando 8 bytes) e teria que pausar (já que precisa de 10 bytes para assistir a esse segundo completamente).

Vamos dividir esta etapa em três partes (para facilitar a implementação).

### Parte 1:

Considere o programa `buscaBinariaFuncoes1.cpp`. Ele lê os valores de B, V e T da entrada padrão. A seguir, testa sequencialmente 20 valores para `tempoCarregamento` (de 0 até 19) e imprime 1 (true) se esse tempo for suficiente para que a pessoa possa assistir ao vídeo sem pausa e 0 (false), caso contrário.

Por exemplo, se a entrada for “10 3 2” a saída seria:

```
0: 0
1: 0
2: 0
3: 0
4: 0
5: 1
6: 1
7: 1
8: 1
9: 1
10: 1
11: 1
12: 1
13: 1
14: 1
15: 1
16: 1
17: 1
18: 1
19: 1
```

Termine a implementação desse programa, implementando a função `consigoAssistirSemPausas(int tempoCarregamento)`. **Atenção: use um método “força-bruta” (ou seja, não tente usar fórmulas matemáticas para ver se é possível assistir sem pausa ou não) -- simplesmente faça um “for” para “simular” cada segundo do vídeo. Queremos que essa função fique lenta para que fique**

**mais claro como uma busca binária poderia ajudar na melhora do desempenho.**

### Parte 2:

O código desta parte deverá ser salvo como `buscaBinariaFuncoes2.cpp`. Use a função `consigoAssistirSemPausas(int tempoCarregamento)` para encontrar o tempo mínimo de carregamento para a pessoa ver o vídeo sem pausa. Implemente isso fazendo uma busca sequencial a partir do 0.

Seu programa deverá ler da entrada padrão os valores de B, V e T e, então, imprimir uma linha contendo o tempo mínimo de carregamento. Veja abaixo 4 exemplos de entradas/saídas.

Obs: Sabemos que  $1 \leq B, V \text{ e } T \leq 5000$ .

Entradas	Saídas
4 1 1 3	
10 3 2 5	
1 1 1 0	
13 12 1 1	

### Parte 3:

Qual a complexidade de pior caso do seu código da parte 2? Meça o tempo dele para valores que você considera desafiadores.

A complexidade de maior caso é  $O(t^2)$ , em que o tempo é o maior possível, a taxa de download é pequena e a taxa de vídeo alta, a ponto de se chegar ao máximo da busca sequencial.

Veja, na parte 1, que sua função (`consigoAssistirSemPausas`) retorna vários valores “0” e, depois, vários valores “1”. Após um valor 1 nunca teremos um valor “0” (se conseguimos ver um vídeo esperando X segundos, também podemos ver o vídeo sem pausa esperando mais que X segundos).

Se esses valores estivessem em um array, esse array estaria ordenado (os 0s viriam antes dos 1s). Veja, abaixo, como seria um “array” armazenando as respostas (da parte 1) com a entrada “10 3 2”

pos	0	1	2	3 4 5 6 7 8	9
resp	0	0	0	0 0 1 1 1 1	1

A resposta para o problema (de encontrar o tempo mínimo) seria simplesmente encontrar no array acima a menor posição contendo o valor 1. Podemos fazer isso com uma busca sequencial (igual fizemos na parte 2) ou, melhor, com uma busca binária!

O desafio é que, a princípio, precisaríamos criar um array para fazer a busca binária nele. O tempo de criar o array seria o tempo gasto para fazer N chamadas à sua função “*consigoAssistirSemPausas*” (onde N é o tamanho do array). A boa notícia é que não precisamos criar esse array! Podemos calcular os valores dinamicamente na busca binária (apenas quando realmente necessário)!

Por exemplo, ao fazer a busca binária no intervalo acima (entre 0 e 9) na primeira etapa tentávamos acessar o valor do meio (posição 4 do array) para ver se ele é 1. Em vez de acessarmos o array, podemos simplesmente chamar sua função *consigoAssistirSemPausas* (passando 4 para ela). Como ela retornará 0, a busca binária será reiniciada entre as posições 5 e 9 (não vamos precisar chamar a função “*consigoAssistirSemPausas*” para nenhum valor do lado esquerdo do array!).

Sua tarefa nessa parte (implemente seu código em *buscaBinariaFuncoes3.cpp*) será implementar essa busca binária para resolver o problema (da parte 2) de forma eficiente (ou seja, fazendo a busca binária na sua função *consigoAssistirSemPausas*, em vez de fazê-la em um array). Apenas esse último arquivo (desta etapa) será avaliado automaticamente pelo submittity (mas os outros também devem ser submetidos).

Veja (não é preciso escrever nada sobre isso -- apenas observe) a diferença entre o tempo dessa implementação e o tempo da implementação anterior (para valores mais desafiadores).

Dica: cuidado com overflow de inteiros nos cálculos. Como a busca binária pode testar se “é possível assistir” passando um valor muito grande, os cálculos podem gerar overflow (seu programa falharia em um dos casos de teste por causa disso).

### **Submissao da aula pratica:**

A solucao deve ser submetida utilizando o sistema submittity ([submittity.dpi.ufv.br](http://submittity.dpi.ufv.br)). Fique sempre atento aos nomes dos arquivos enviados (que devem corresponder aos nomes descritos em cada etapa)