UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO

LABORATÓRIO DE PROGRAMAÇÃO – CK0215 2023.1 T01A

TRABALHO 1: ALGORITMOS DE ORDENAÇÃO (VERSÃO 1: 2023-03-27)

PROFESSOR: PABLO MAYCKON SILVA FARIAS

1 Resumo

Você deverá implementar em C++ alguns algoritmos de ordenação, bem como avaliar seus desempenhos sobre instâncias de diferentes tipos. A implementação será entregue ao professor através de tarefa no SIGAA, e depois será apresentada em horário agendado. Tanto a implementação submetida via SIGAA quanto a apresentação contam para a nota. O trabalho é individual e cada estudante deve produzir seu código por conta própria.

Atenção: Colabore para o bom andamento da disciplina e do aprendizado de todos, não usando códigos que não tenham sido escritos por você nem compartilhando seu próprio código, nem mesmo em estágio inicial de desenvolvimento. Em caso de dificuldade sua ou de algum colega, fale com o professor.

2 Pivô do Quicksort

Parte importante deste trabalho consiste em analisar o desempenho do algoritmo Quicksort em instâncias de pior caso, observando a necessidade de lidar com elas. Para viabilizar a geração dessas instâncias, a sua versão do Quicksort com escolha fixa de pivô terá que realizar essa escolha através de uma função determinística

```
int pivo (int inicio, int fim) { ... sua implementação aqui ... }
```

A função acima deve retornar o índice do pivô escolhido para um intervalo que inicie na posição `inicio` e que termine na posição `fim` de um dado vetor. Observe que a declaração acima da função permite que você implemente uma entre diferentes escolhas de pivô, como o primeiro elemento do intervalo, o último, o elemento do meio, etc. Por outro lado, como a função não possui acesso ao vetor a ser ordenado, então estratégias como a "mediana de três" não podem ser implementadas. Da mesma forma, como a função tem que ser determinística, então um cálculo total ou parcialmente baseado numa escolha aleatória não é uma opção.

3 Instâncias de Pior Caso

Você terá que implementar uma função

```
template <typename T>
bool gerar_pior_caso (T *v, int n) { ... sua implementação aqui ... }
```

que receba um vetor `v` de `n` posições e que então escreva nele uma instância de `n` números *distintos* que leve ao pior caso de qualquer implementação do Quicksort que:

1. Utilize a função `pivo` para escolher seus pivôs, e

2. Assim como os algoritmos de particionamento de Hoare e de Lomuto, comece por trocar o pivô com o primeiro elemento do intervalo, depois realize um particionamento que não realize movimentos exceto em caso de elementos "fora de ordem" em relação às 2 partes, e por fim coloque o pivô entre as 2 partes através de uma troca.

A estratégia a ser implementada é a seguinte. Toda instância de pior caso com n elementos possuirá exatamente os números de 1 a n, em alguma ordem. A ideia básica é então obter uma ordem tal que a escolha da função `pivo` seja sempre tão ruim quanto possível, isto é, seja a de um elemento que, após o particionamento, será necessariamente posicionado numa das extremidades do intervalo (isto é, o início ou o final). Como nós sabemos de antemão que serão os números de 1 a n aqueles que estarão na instância, então nós podemos escolher suas posições à nossa conveniência. Mais especificamente, nós podemos gerar a instância de forma que o número 1 seja o $1^{\underline{o}}$ pivô escolhido, que o número 2 seja o $2^{\underline{o}}$ pivô escolhido, etc.

Assim, para uma explicação através de exemplo, suponha que foi solicitada a geração de uma instância com 3 elementos, que a seguir nós denotaremos por `[ a b c ]`. Além disso, para tornar o exemplo ainda mais concreto, suponha, sem perda de generalidade, que a função `pivo` sempre retorna o elemento do meio do intervalo. Nesse caso:

1. Nós sabemos que, quando o Quicksort for ordenar a instância, a sua chamada mais externa receberá o intervalo `[0..2]` do vetor para ordenar. Consequentemente, nós sabemos que `pivo(0,2)` será a primeira chamada à função `pivo`, e, pela hipótese do exemplo, ela retornará 1. Logo, sendo `[ a b c ]` a instância a ser gerada, o elemento `b`, de índice 1, será o primeiro pivô da ordenação.

2. Assim, como desejamos gerar uma instância de pior caso, nós podemos colocar o número `1` na posição do primeiro pivô, ou seja, podemos escolher `b == 1`. Nesse caso, a instância será `[ a 1 c ]`, para algum número `a` e algum número `c`.

3. Prosseguindo com a simulação do Quicksort, nós sabemos que o primeiro passo do primeiro particionamento será trocar o pivô com o primeiro elemento do intervalo `[0..2]`. Logo, `b` será trocado com `a`, passando o vetor ao estado `[ 1 a c ]`. Além disso, como nós sabemos que a instância será composta pelos números de 1 a 3, então tanto `a` quanto `c` serão maiores que 1, e portanto o particionamento manterá o vetor no estado `[ 1 a c ]`.

4. Continuando com a simulação do Quicksort, como o intervalo à esquerda do número 1 é vazio, então restará ordenar o intervalo à direita: `[1..2]`. Nesse caso, a próxima chamada à função `pivo` será `pivo(1,2)`, que retornará 1. Nesse caso, como o vetor está no estado `[ 1 a c ]`, então o $2^{\underline{o}}$ pivô escolhido será o número `a`. Assim, seguindo a estratégia explicada inicialmente, nós podemos escolher `a == 2`, e nesse caso o vetor estará no estado `[ 1 2 c ]`.

5. Prosseguindo na simulação do Quicksort, o particionamento do intervalo `[1..2]` não alterará o vetor `[ 1 2 c ]`, pois `c > 2`. Além disso, como o intervalo à esquerda do número 2 será vazio, então restará ordenar o intervalo à direita, `[2..2]`, que possui apenas o número `c`, e nós podemos escolher `c == 3`.

6. A conclusão do raciocínio acima é que `[ a b c ] = [ 2 1 3 ]` é uma instância de pior caso para qualquer versão do Quicksort no âmbito das restrições estabelecidas.

Observe que a estratégia acima se aplica independentemente de qual for a implementação utilizada para a função `pivo` (dentro das restrições estabelecidas na seção anterior), e que a versão acima retornando sempre o elemento do meio do intervalo foi escolhida apenas para tornar o exemplo concreto. No fim das contas, o acesso à função `pivo` e o conhecimento básico de como procede o algoritmo de particionamento são os elementos que permitem que a função

`gerar_pior_caso` "preveja" como se dará a execução do Quicksort, viabilizando a geração de uma instância de pior caso.

4 Algoritmos de Ordenação a Implementar

Você deverá implementar pelo menos os seguintes algoritmos, que devem ordenar um vetor em ordem crescente:

1. **Heapsort** (aula 06).
2. **Quicksort com Pivô Fixo** (aula 07): esta versão deverá escolher o pivô usando a função `pivo` que você implementar.
3. **Quicksort com Pivô Aleatório** (aula 09): esta versão deverá escolher o pivô usando os recursos de geração de números (pseudo)aleatórios da linguagem (naturalmente, você deve criar um motor gerador de números aleatórios apenas uma vez no programa, e não toda vez que for gerar um número).
4. **Introsort sem Insertion Sort** (aula 08).
5. **Introsort com Insertion Sort** (aula 08): Esta versão deverá utilizar a ordenação por inserção para ordenar instâncias abaixo de um certo tamanho fixo, que deverá ser escolhido por você empiricamente, de forma a minimizar o tempo de execução do algoritmo inteiro.

É importante que cada algoritmo seja implementado como um molde (*template*) de função, parametrizado pelo tipo `T` dos elementos do vetor a ser ordenado.

5 Instâncias a Gerar

O seu programa deverá permitir fornecer como entrada aos algoritmos de ordenação pelo menos os seguintes tipos de instância:

1. **Aleatórias:** criadas usando os recursos de geração de números (pseudo)aleatórios da linguagem.
2. **Pior Caso:** criadas usando a estratégia explicada em seção anterior.
3. **Ordem Crescente**: instâncias que já começam ordenadas em ordem crescente.
4. **Ordem Decrescente**. instâncias que já começam ordenadas em ordem decrescente.

6 Aferição do Tempo

Você deverá escrever uma função para aferir o tempo de ordenação de um dado vetor por um dado algoritmo. A função poderá realizar tarefas adicionais, como copiar o conteúdo de um vetor "original" (onde esteja uma instância que será ordenada por diferentes algoritmos) para outro vetor, o qual será efetivamente ordenado pelo algoritmo em questão. Em todo caso, é importante que a contagem de tempo realizada pela função seja relativa apenas ao tempo efetivamente gasto na ordenação, não incluindo o tempo gasto em procedimentos auxiliares.

A função deverá retornar o tempo gasto na ordenação, de forma que esse valor possa ser adicionado ao tempo gasto pelo algoritmo em questão durante a ordenação de diferentes instâncias, tempo esse a ser exibido ao final da execução do programa.

Diferentes chamadas da função em questão deverão servir para registrar o tempo de ordenação de diferentes vetores por diferentes algoritmos de ordenação. Naturalmente, os vetores devem ser recebidos em tempo de execução, através de argumentos fornecidos à função; já o algoritmo de ordenação poderá ser recebido tanto em tempo de execução, como um ponteiro para a função de ordenação, como em tempo de compilação, como um parâmetro *template* (conforme explicado na aula 05).

7 Compilação e Execução do Programa

O seu programa deverá ser escrito em C++17 padrão, e deve ser compilável, sem erros ou avisos, através de uma linha como

```
g++ -Wall -Wextra -std=c++17 -pedantic -o programa  main.cpp
```

Caso algo além disso seja necessário para compilar o programa (por exemplo, caso o seu programa esteja em mais de um arquivo), por favor inclua um arquivo `explicacoes.txt` com as devidas explicações.

A chamada do programa terá o formato (use `argc` e `argv`, conforme a aula 02)

```
./programa [Tipo de Instância] [Tamanho do Vetor] [Número de Instâncias]
```

sendo os argumentos (que não incluirão colchetes) os seguintes:

- `Tipo de Instância`: Esse argumento determinará o tipo das instâncias que serão geradas e fornecidas a todos os algoritmos de ordenação implementados. O argumento será uma única letra maiúscula, que representará um dos 4 tipos de instância aceitos pelo programa: `A` (aleatória), `C` (ordem crescente), `D` (ordem decrescente), e `P` (pior caso).
- `Tamanho do Vetor`: Esse argumento será um número natural positivo informando o tamanho das instâncias a serem geradas pelo programa.
- `Número de Instâncias`: Esse argumento também será um número natural positivo e informará o número de instâncias a serem geradas e ordenadas por cada algoritmo implementado.

Não será exigido um comportamento específico do programa caso o usuário forneça argumentos inválidos na chamada.

Em cada chamada correta realizada ao programa, ele deverá criar tantas instâncias quantas solicitado pelo usuário, do tipo e com o tamanho requeridos, e então deverá submeter cada instância gerada a cada um dos algoritmos implementados. Obviamente, todos os algoritmos devem ser executados sobre exatamente as mesmas instâncias; portanto, o natural é, para cada instância gerada, criar uma cópia dela e submeter ao primeiro algoritmo, depois criar outra cópia e submeter ao segundo algoritmo, etc. Ao final, o programa deve exibir na tela, para cada algoritmo implementado, o tempo total (em segundos, incluindo a parte fracionária) utilizado pelo algoritmo para ordenar todas as instâncias geradas. **Atenção:** não imprima o tempo levado por cada algoritmo para uma instância específica (imagine a saída do programa quando fossem solicitadas 1000 instâncias!); ao invés disso, para cada um dos 5 algoritmos solicitados, o programa deve imprimir o tempo total que o algoritmo levou para ordenar todas as instâncias geradas.

Importante: após cada ordenação, o seu programa deve checar se o vetor foi corretamente ordenado, emitindo uma mensagem de erro caso um ocorra. Naturalmente, o tempo utilizado nessa checagem não deve ser contabilizado no tempo utilizado pelo algoritmo para a ordenação.

8 Submissão do Trabalho

A solução do trabalho deverá consistir num arquivo `[Matrícula].zip` (sem colchetes) entregue através do SIGAA na tarefa cadastrada pelo professor. Naturalmente, apenas o código-fonte deve ser submetido, sem qualquer arquivo executável. Quaisquer explicações adicionais podem ser incluídas através de um arquivo `explicacoes.txt`.

Em caso de dúvida, contate o professor rapidamente.

9 Extras

Por falta de tempo, dentro e fora de sala, o professor não abordou diferentes otimizações que podem ser realizadas sobre os algoritmos abordados neste trabalho. Caso você deseje realizar algumas delas em suas implementações, incluindo implementar diferentes versões para efeito de comparação, ou mesmo implementar algoritmos adicionais, fique à vontade. Em todo caso, uma solução que atenda a todos os requisitos deste trabalho, incluindo os requisitos básicos de eficiência presumidos para cada algoritmo, deverá receber nota 10.

– Que você tenha uma prática rica em aprendizados. Bom trabalho! –