Trabalho prático 1: Muita ordenação!

Algoritmos e Estruturas de dados II (2021/1)

1 Introdução

Ordenação é fundamental em computação pois muitos outros algoritmos que serão vistos durante o curso se baseiam em entradas pré-ordenadas ou necessitam de ordenação em algum passo intermediário do algoritmo. Compreender os algoritmos de ordenação presentes na literatura permite ao cientista da computação (i) construir algoritmos mais eficientes, selecionando algoritmos de ordenação mais adequados para um determinados cenários e (ii) dispor de um conjunto maior de algoritmos que podem ser modificados para resolver determinado problema como, por exemplo, modificar algoritmos de ordenação existentes para atender critérios de desempate específicos.

Neste trabalho, você deverá implementar os seguintes algoritmos de ordenação: seleção, inserção, *Merge Sort* (*mergesort*), *Quick Sort* (*quicksort*), *Heap Sort* (*heapsort*) e o *Bucket Sort* (*bucketsort*)[1]. Além da implementação, você deverá realizar uma avaliação experimental dos algoritmos em relação ao tempo de execução.

2 Bucket Sort

Da lista de algoritmos a serem implementados, o *bucketsort* foi o único algoritmo que não foi visto em aula. Entretanto, ele é extremamente simples. A ideia central do algoritmo é criar k intervalos numéricos e organizar a sequência de números de entrada em cada intervalo. Uma vez organizada a entrada, o algoritmo ordena os números em cada um dos k intervalos utilizando algum outro algoritmo de ordenação. Por exemplo, seja a entrada $\{9, 30, 2, 49, 44, 3, 5, 15, 35\}$ e k=5 o número de intervalos desejados. Inicialmente, definimos os cinco intervalos como sendo $\{$0-10, 11-20, 21-30, 31-40, 41-50$\}$ e organizamos os números da entrada em cada intervalo conforme mostrado pela figura 1. Para imprimir a entrada de forma ordenada em ordem crescente, basta seguir a sequência dos intervalos e imprimir os números ordenados daquele intervalo. No exemplo dado, primeiro imprimimos os números ordenados do intervalo 1-10, resultando em $\{2, 3, 4, 5\}$. Em seguida, imprimimos os números ordenados do intervalo 11-20, resultando em $\{2, 3, 4, 5, 15\}$, e assim sucessivamente, até o último intervado (41-50) que resulta em $\{2, 3, 5, 9, 15, 30, 35, 44, 49\}$.

[1] O nome entre parênteses é o que será utilizado no restante do texto.

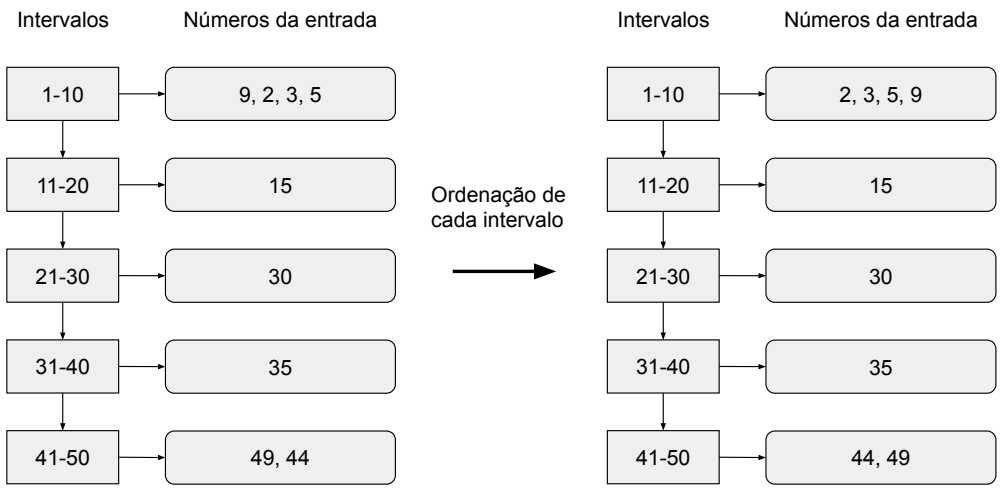

Figura 1: Exemplo de ordenação com o *bucketsort*.

A principal vantagem do *bucketsort* é que ele consegue reduzir o tamanho da entrada para outros algoritmos de ordenação, fazendo com que executem mais rapidamente (algo que você deverá mostrar na avaliação experimental). Para definir os intervalos, utilize o método mostrado no livro "Algoritmos: Teoria e Prática 3^{rd} edição" do Cormen (página 200), que trabalha apenas no intervalo de [0,1). Para isso, você terá que normalizar os números inteiros da entrada para o intervalo [0,1) usando o maior número da entrada como referência.

3 Instruções sobre a Implementação

Seu código deverá ser escrito em C e deverá conter, pelo menos, os seguintes arquivos: *selection.c*, *insertion.c*, *mergesort.c*, *quicksort.c*, *heapsort.c*, *bucketsort.c* e *main.c*. O código de cada algoritmo de ordenação deverá estar no arquivo correto como, por exemplo, o arquivo *quicksort.c* deverá possuir apenas o código relativo à implementação do algoritmo do *quicksort*. Você pode criar arquivos de cabeçalho para os arquivos definidos (por exemplo, *quicksort.h*) para tornar o código mais modular. Porém isso não é necessário.

A implementação dos intervalos do *bucketsort* deverão utilizar listas encadeadas. Entretanto, a organização dentro do intervalo pode utilizar listas, vetores ou qualquer outra estrutura de dados desejada. Além disso, a função interna do *bucketsort* para ordenar os números dentro de um intervalo deverá ser genérica para receber qualquer um dos outros algoritmos de ordenação implementados (seleção, inserção, *mergesort*, *quicksort* e *heapsort*)[2].

4 Formato de Entrada e Saída

O trabalho deve ser executado da seguinte forma, ou seja, sem argumentos:

[2]Dica: padronize a entrada dos algoritmos de ordenação implementados e use ponteiro para função

```
./tp
```

Seu programa deverá ler a entrada da entrada padrão ($stdin$) e gravar a saída na saída padrão ($stdout$). A **entrada** possui múltiplos casos de teste. Cada caso de teste é organizado da seguinte forma. A primeira linha possui o número de elementos N a serem ordenados ($N < 10^8$). A segunda linha possui N números inteiros positivos que são o conjunto de números a serem ordenados. A terceira linha contém dois valores V e M, separados por espaço. O valor V indica o algoritmo de ordenação a ser utilizado para ordenar cada intervalo do bucketsort (selecao, insercao, *mergesort*, *quicksort*, *heapsort*) e o valor M indica o número de intervalos ($1 < M \leq N$) que o *bucketsort* deverá possuir. A entrada termina quando $N = 0$.

Cada linha i da **saída** representa o caso de teste i da entrada e deverá possuir os números fornecidos na entrada ordenados em ordem crescente e separados por espaço.

Exemplo de Entrada:

```
7
9 5 2 1 3 8 2
quicksort 3
8
5 6 9 1 2 3 10 11
heapsort 4
7
9 5 2 1 3 20 2
mergesort 4
5
5 4 3 2 1
insercao 5
6
4 5 3 2 1 6
selecao 1
0
```

Exemplo de Saída:

```
1 2 2 3 5 8 9
1 2 3 5 6 9 10 11
1 2 2 3 5 9 20
1 2 3 4 5
1 2 3 4 5 6
```

5 O que entregar

Você deve submeter uma documentação de até 10 páginas, em formato PDF, contendo uma descrição de como seu código foi implementado, além de uma análise de complexidade de tempo

dos algoritmos envolvidos e de complexidade de espaço da estrutura de dados. Siga as diretrizes sobre como fazer uma documentação que foram disponibilizadas no Portal Didático.

Além da documentação, você deve submeter um arquivo compactado no formato *.tar.gz* contendo todos os arquivos de código (*.c* e *.h*) que foram implementados. Além dos arquivos de código, esse arquivo compactado deve incluir um *makefile*. Consulte os tutoriais disponibilizados no Portal Didático para descobrir como fazer um. Finalmente, lembre-se de não incluir nenhuma pasta no arquivo compactado.

6 Avaliação

Para que seu trabalho seja elegível para correção, seu código deve acertar todos os casos de teste do exemplo desta especificação! Seu trabalho será avaliado quanto a documentação escrita e à implementação. Eis uma lista **não exaustiva** de critérios de avaliação utilizados.

Documentação

Introdução Inclua uma breve explicação do problema que está sendo resolvido no seu trabalho e um resumo da sua solução.

Solução do Problema Você deve descrever a solução do problema de maneira clara e precisa. Para tal, artificios como pseudo-códigos, exemplos ou figuras podem ser úteis. Note que documentar uma solução não é o mesmo que documentar seu código. **Não** é preciso incluir trechos de código em sua documentação nem mostrar detalhes de sua implementação, exceto quando os mesmos influenciem o seu algoritmo principal, o que se torna interessante. Por exemplo, é interessante discutir quais estruturas de dados foram utilizadas no *bucketsort* para mostrar como os dados foram organizados em memória.

Análise de Complexidade Inclua uma análise de complexidade de tempo dos principais algoritmos implementados e uma análise de complexidade de espaço das principais estruturas de dados de seu programa. Cada complexidade apresentada deverá ser devidamente **justificada** para que seja aceita. Apesar da complexidade dos algoritmos implementados serem conhecidos na teoria, a complexidade pode variar de acordo com a implementação. Um dos principais pontos a serem abordados na documentação é a complexidade de tempo do *bucketsort* e de espaço dos outros algoritmos. Busque entender e explicar com suas próprias palavras. Somente dessa forma você aprenderá um dos tópicos essenciais em computação.

Avaliação Experimental Sua documentação deve incluir os resultados de experimentos que avaliem o tempo de execução de seu código em função de características da entrada. Cabe a você gerar entradas para esses experimentos. Como este trabalho é de ordenação, é interessante mostrar como o tempo de execução de cada algoritmo varia quando o número de items a serem ordenados aumenta. Para tal, um gráfico mostrando o tempo de execução em função do tamanho da entrada pode ser interessante. Você também deve interpretar os resultados obtidos. Comente sobre cada gráfico ou tabela

que você apresentar mostrando o que é possível concluir a partir dele. Por fim, não esqueça de variar o impacto do número de intervalos do *bucketsort* na velocidade da ordenação.

Limite de Tamanho Sua documentação deve ter no máximo 10 páginas. Todo o texto a partir da página 11, se existir, será desconsiderado.

Guia Há um guia e exemplos de documentação disponíveis no Portal Didático.

Implementação

Linguagem e Ambiente

- Implemente tudo na linguagem **C**. Você pode utilizar qualquer função da biblioteca padrão da linguagem em sua implementação, mas **não** deve utilizar outras bibliotecas e muito menos funções que realizem o trabalho para você como o *qsort*. **Trabalhos em outras linguagens de programação serão zerados. Trabalhos que utilizem outras bibliotecas também.**
- Os testes serão executados em **Linux**. Portanto, garanta que seu código compila e roda corretamente nesse sistema operacional. A melhor forma de garantir que seu trabalho rode em Linux é escrever e testar o código nele. Você pode fazer o download de uma variante de Linux como o **Ubuntu** (`http://www.ubuntu.com`) e instalá-lo em seu computador ou diretamente ou por meio de uma máquina virtual como o **VirtualBox** (`https://www.virtualbox.org`). Há vários tutoriais sobre como instalar Linux disponíveis na web.

Casos de teste Seu trabalho será executado em um conjunto de entradas de teste.

- Essas entradas *não* serão disponibilizadas para os alunos até que a correção tenha terminado. Faz parte do processo de implementação testar seu próprio código.
- Você perderá pontos se o seu trabalho produzir saídas incorretas para algumas das entradas ou não terminar de executar dentro de um tempo limite pré-estabelecido. Esse tempo limite é escolhido com alguma folga. Garanta que seu código roda a entrada de pior caso em no máximo alguns minutos e você não terá problemas.
- A correção será **automatizada**. Esteja atento ao formato de saída descrito nessa especificação e o siga precisamente. Por exemplo: se a saída esperada para uma certa entrada o número 10 seguido de uma quebra de linha, você deve imprimir apenas isso. Imprimir algo como "A resposta e: 10" contará como uma saída **errada**.
- Os exemplos mostrados nessa especificação são parte dos casos de teste.
- **Você deve entregar algum código e esse código deve compilar e executar corretamente para, pelo menos, o *exemplo* de entrada desta especificação. Se isso não ocorrer, a nota do trabalho prático será zerada.**

Alocação Dinâmica Você também deve liberar *tudo* o que for alocado utilizando `free()`. Verifique o tutorial disponibilizado no Portal Didático sobre como utilizar o Valgrind para verificar vazamentos de memória.

Makefile Inclua um makefile na submissão que permita compilar o trabalho.

Qualidade do Código Seu código deve ser bem escrito:

- Dê nomes a variáveis, funções e estruturas que façam sentido.
- Divida a implementação em módulos que tenham um significado bem definido.
- Acrescente comentários sempre que julgar apropriado. Não é necessário parafrasear o código, mas é interessante acrescentar descrições de alto nível que ajudem outras pessoas a entender como sua implementação funciona.
- Evite utilizar variáveis globais.
- Mantenha as funções concisas. Seres humanos não são muito bons em manter uma grande quantidade de informações na memória ao mesmo tempo. Funções muito grandes, portanto, são mais difíceis de entender.
- Lembre-se de indentar o código. Escolha uma forma de indentar (tabs ou espaços) e mantenha-se fiel a ela. Misturar duas formas de indentação pode fazer com que o código fique ilegível quando você abri-lo em um editor de texto diferente do que foi utilizado originalmente.
- Evite linhas de código muito longas. Nem todo mundo tem um monitor tão grande quanto o seu. Uma convenção comum adotada em vários projetos é não passar de 80 caracteres de largura.
- **Tenha bom senso**.

7 Considerações Finais

- Essa especificação não é isenta de erros e ambiguidades. Portanto, se tiver problemas para entender o que está escrito aqui, pergunte ao professor da disciplina ou aos seus colegas de classe.
- Não existe política de atraso! Após a data de entrega, a nota é zero.
- Você **não** precisa de utilizar uma IDE como Netbeans, Eclipse, Code::Blocks ou VScode para implementar esse trabalho prático. No entanto, se o fizer, **não** inclua os arquivos que foram gerados por essa IDE em sua submissão.
- **Seja honesto**. Você não aprende nada copiando código de terceiros nem pedindo a outra pessoa que faça o trabalho por você. Se a cópia for detectada, sua nota será zerada e seu caso será comunicado ao colegiado do curso, conforme resolução da UFSJ.