Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Estruturas de Dados Básicas I • DIM0119
– Análise Empírica de Algoritmos de Ordenação –
24 de abril de 2023

Sumário

1 Introdução **1**

2 O Problema Computacional **2**

3 Cenários da Simulação **2**

4 Algoritmos **3**

5 Recomendações e Dicas de Implementação **3**
5.1 Organização do código . . . 3
5.2 Tamanho das amostras . . . 3
5.3 Gerando gráficos . . . 4
5.4 Precisão dos dados . . . 4
5.5 Alocação de memória para os dados . . . 5
5.6 Considerações sobre o pivô do quicksort . . . 6

6 Relatório Técnico **6**

7 Avaliação do Trabalho **7**

8 Autoria e Política de Colaboração **8**

9 Entrega **9**

1 Introdução

Neste trabalho você deverá atuar como um consultor técnico contratado para ajudar na escolha do algoritmo ideal para realizar ordenação em um arranjo. Para cumprir seu objetivo você deverá utilizar análise empírica de algumas opções de algoritmos sugeridos tentando simular determinados cenários. Os resultados da análise deverão ser apresentados na forma de um **relatório técnico** com suas recomendações para os cenários avaliados em seus experimentos.

A seguir serão apresentados os cenários a serem simulado e os algoritmos para os quais você deverá realizar a análise empírica. Também serão listadas orientações para auxiliar na elaboração do relatório técnico a ser redigido e entregue.

Ao final do exercício você deve submeter, através do Sigaa, tanto o relatório técnico quanto o conjunto de programas desenvolvidos para realizar a análise empírica.

Nas próximas seções serão fornecidos mais detalhes sobre o trabalho, algumas instruções para a elaboração do relatório e a política de colaboração na execução do trabalho em equipe.

2 O Problema Computacional

O problema da **ordenação de um arranjo sequencial** tem como entrada:

> Uma *sequência*, $\langle a_1, \ldots, a_n \rangle$, de n elementos, com $n \in \mathbb{Z}$ e $n > 0$, tal que os elementos da sequência pertencem a um conjunto totalmente ordenável por '$\leq$'.

e como saída:

> Uma *permutação*, $\langle a_{\pi 1}, \ldots, a_{\pi n} \rangle$, da sequência de entrada tal que $a_{\pi 1} \leq a_{\pi 2} \leq \cdots \leq a_{\pi n}$.

Para solucionar este problema você deverá analisar sete algoritmos, com complexidade temporal variadas. Em suas simulações você deve considerar alguns cenários, descritos na próxima seção.

3 Cenários da Simulação

Você deve considerar que a simulação será realizada sobre um arranjo de inteiros (`unsigned int`) cujo comprimento, ou **tamanho de amostra** n, deverá ser "*muito grande*". Entenda o termo "*muito grande*" como sendo um vetor com $10^5 elementos$.

Como estamos interessados em avaliar o *comportamento assintótico* dos algoritmos em relação ao seu *tempo de execução*, os cenários deverão simular a execução dos algoritmos para diversos tamanhos de amostras com valores de n crescentes, até atingir ou passar um pouco de 10^5. Por esta razão serão necessários, *pelo menos*, 25 tamanhos diferentes de amostras, igualmente espaçados entre a menor amostra e a maior amostra. A menor amostra sugerida seria com 10^2 elementos.

Com relação a **organização das amostras** é necessário simular seis situações:

1. arranjos com elementos em ordem *não decrescente*,
2. arranjos com elemento em ordem *não crescente*,
3. arranjos com elementos 100% *aleatórios*,
4. arranjos com 75% de seus elementos em sua posição definitiva,
5. arranjos com 50% de seus elementos em sua posição definitiva, e
6. arranjos com 25% de seus elementos em sua posição definitiva.

Note, portanto, que é possível a ocorrência de elementos repetidos no arranjo. Nos últimos 3 casos, queremos simular situações de arranjos semi-ordenados em graus diferentes (25%, 50%

e 75%) para verificar se algum dos algoritmos avaliados vai apresentar desempenho favorável nessas situações.

Para gerar um vetor com, digamos, 75%, dos elemento em sua ordem definitiva você pode proceder da seguinte forma (sugestão):

1. Comece com o arranjo A de dados já em ordem *não decrescente*.
2. Crie um arranho I com valores de 0 até $n-1$, onde n é o tamanho do arranjo A.
3. Embaralhe (*shuffle*) os elementos de I.
4. Como desejamos, no exemplo, 75% dos elementos na ordem correta, precisamos modificar apenas 25% dos elementos originais.
   Faça um laço de $i = 0$ até $p = \lceil (0.25 * n) \rceil$, com o i sendo incrementado de 2 em 2, trocando $A[I[i]]$ com $A[I[i+1]]$.

Para os demais valores, 75% e 50%, você precisa substituir esse valor pela quantidade que se deseja embaralhar.

4 Algoritmos

Você deve implementar e analisar **sete algoritmos** em seus experimentos. São eles: insertion sort, selection sort, bubble sort, shell sort, quick sort, merge sort e radix sort (LSD).

5 Recomendações e Dicas de Implementação

5.1 Organização do código

Para facilitar a construção do programa de testes para medição dos tempos de execução, recomenda-se que os sete algoritmos sejam armazenados em um vetor de ponteiros para função. Portanto, todos os sete algoritmos devem apresentar a **mesma assinatura**.

A segunda recomendação é fazer com que o programa seja flexível a ponto de poder ser facilmente configurável. Na prática, essa configuração significa, por exemplo, poder selecionar quais algortimos executar, quais cenários devem ser configurados, quantas amostras, etc. Esta estratégia permite a execução dos testes para algoritmos individuais ou grupos de algoritmos, conforme a necessidade.

5.2 Tamanho das amostras

A princípio a faixa de tamanhos de amostras a serem gerados é $[10^2, 10^5]$, portanto temos um intervalo de 99900 elementos entre a menor e maior amostra. Se desejamos 25 amostras igualmente espaçadas, temos que o incremento de amostra seria $\lfloor \frac{99900}{24} \rfloor = 4162$. Portanto temos um tamanho de amostra inicial $n_0 = 10^2$ e para os tamanhos seguintes faça $n_i = n_{i-1} + 4162$, chegando até a última amostra $n_{24} = 99988$ (isto é, $n_{24} = 100 + 24 * 4162$). Se você desejar gerar mais do que 25 tamanhos de amostras terá que calcular o fator de

incremento adequado. Também se desejar ampliar o intervalo de testes, precisa ajustar o cálculo do tamanho das amostras.

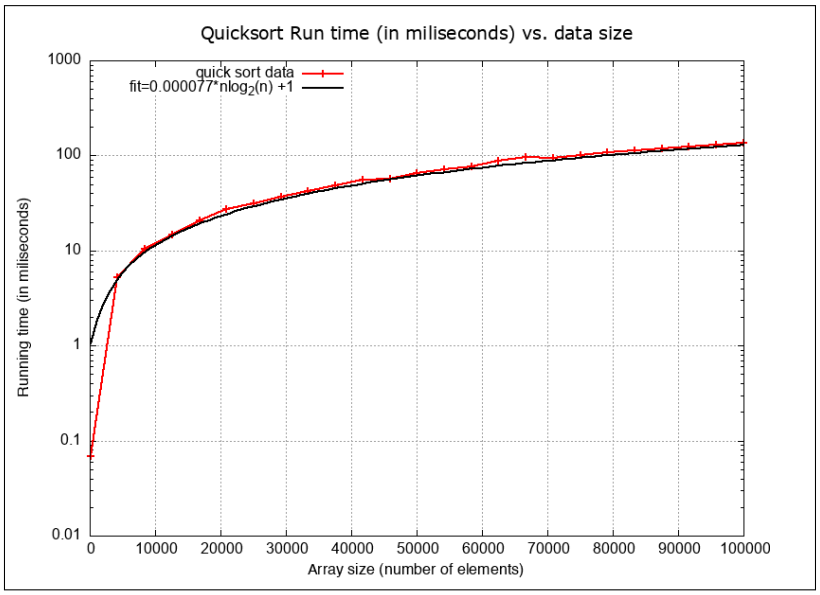
5.3 Gerando gráficos

Se você for adotar o `gnuplot` como ferramenta de geração de gráficos, lembre-se de gravar os resultados na forma de colunas ao invés de linhas. Veja abaixo um exemplo de arquivo de dados que correspondem ao tempo execução (em milissegundos) do algoritmo quicksort no cenário em que todos os dados são aleatórios, com 25 amostras variando de 100 a 100000 elementos, que podem ser lidos pelo `gnuplot`. Na figura abaixo a listagem dos dados está do lado esquerdo e o gráfico correspondente é apresentao no lado direito. No gráfico, a linha vermelha ligando os pontos representa os dados coletados e a curva preta é uma curva ajustada para $n \log(n)$, demonstrando que os dados coletados se ajustam a complexidade teórica esperada do algoritmo.

Dados

```
# n QUICKSORT
100 0.0688228
4262 5.2412508
8424 10.492653
12586 14.613872
16748 20.806611
20910 27.109978
25072 31.546439
29234 36.675401
33396 42.448622
37558 48.288919
41720 56.114686
45882 57.394152
50044 65.239432
54206 71.672244
58368 76.832384
62530 88.174173
66692 95.324963
70854 93.758452
75016 100.17496
79178 108.39861
83340 112.65011
87502 118.07573
91664 124.30478
95826 129.60344
99988 135.9328
```

Gráfico correspondente



5.4 Precisão dos dados

Para aumentar a precisão de suas medidas e evitar algum tipo de flutuação que pode ocorrer em uma única execução, recomenda-se que sejam realizadas (pelo menos) 5 execuções para cada instância de tamanho n_i. Depois calcule a média aritmética dos tempos das 5 execuções: este será o tempo médio da execução do algoritmo para uma instância do problema com tamanho n_i.

Para calcular a média aritmética das 5 tomadas de tempos, recomenda-se a utilização da média progressiva, ou seja, a média vai sendo atualizada a cada novo tempo computado. Para

$k = 1, 2 \ldots, m$ execuções, temos

$$\begin{aligned} M_0 &= 0, && \text{valor inicial da média} \\ M_k &= M_{k-1} + \tfrac{x_k - M_{k-1}}{k}, && \text{atualização progressiva da média} \end{aligned}$$

onde x_k é tempo mensurado para o k-ésima execução e $M_{k=m}$ corresponde a média aritmética final da sequência de m tempos medidos. Esta fórmula evita que seja necessário somar todos os tempos primeiro para depois dividir a soma total por m.

Por exemplo, se a sequência de valores fosse $\{7, 15, 20, 38\}$, o cálculo progressivo da média seria:

$$\begin{aligned} M_0 &= 0 \\ M_1 &= 0 + \tfrac{7-0}{1} = 7 \\ M_2 &= 7 + \tfrac{15-7}{2} = 7 + \tfrac{8}{2} = 7 + 4 = 11 \\ M_3 &= 11 + \tfrac{20-11}{3} = 11 + \tfrac{9}{3} = 11 + 3 = 14 \\ M_4 &= 14 + \tfrac{38-14}{4} = 14 + \tfrac{24}{4} = 14 + 6 = 20. \end{aligned}$$

Note que a média final calculada de forma progressiva no exemplo acima, $M = 20$, corresponde ao valor da média calculada "tradicionalmente" com $M = \frac{\sum_{k=1}^{m} x_k}{m}$: $(7 + 15 + 20 + 38)/4 = 20$.

5.5 Alocação de memória para os dados

Com relação a alocação do arranjo, ao invés de alocar/desalocar vários arranjos com tamanhos crescentes de n_i, recomenda-se que um único arranjo com tamanho máximo de $n = 10^5$ elementos seja alocado e preenchido no início do programa. Na hora de executar os algoritmos, você deve passar apenas a parte do arranjo correspondente ao tamanho da amostra da vez. Para isso você pode utilizar *ranges*, isto é, ponteiros (ou índices) que definem um intervalo de dados no vetor original. Por exemplo, se você precisa testar uma amostra $n = 10^4$ elementos, invoque a busca passando $l = 0$ e $r = 10^4 - 1$, embora o arranjo possua $n = 10^5$ elementos no total. Neste caso a ordenação será feita em uma subconjunto do arranjo original.

Esta simples estratégia evita que seu programa perca muito tempo alocando e desalocando arranjos de tamanhos menores. Vale ressaltar que o tempo necessário para alocar o vetor não deve entrar no cálculo do tempo de execução dos algoritmos.

Talvez seja interessante manter um "backup" dos valores originais antes de ordenar, pois a cada nova execução é necessário desfazer a ordenação, ou seja, devemos voltar a condição inicial antes da execução de uma rodada. Nesse caso você pode usar uma função de cópia sobre ranges (como o `std::copy`) para restaurar os valores originais do subvetor que tenha sido ordenado, a partir do vetor (backup) original.

Também é importante lembrar que a organização dos dados descritas nos cenários precisam ser ajustadas de forma a afetar apenas o *range* de interesse. Por exemplo, para gerar um cenário com 25% dos elementos aleatórios considerando uma amostra com, digamos, 10^3 elementos, é preciso aplicar o algoritmo descrito na Seção 3 sobre o *range* $[0, 10^3]$ e não sobre o arranjo completo $[0, 10^5]$.

5.6 Considerações sobre o pivô do quicksort

Se você utilizar a versão tradicional do quicksort na qual o pivô é sempre o último elemento do intervalo a ser ordenado, seu programa vai gerar *segmentation fault* para os cenários com dados em ordem não-decrescente e não-crescente. Você consegue saber o porque? *(pense um pouco antes de ler a resposta a seguir)*

Isso porque esses são considerados o pior caso do quicksort, no qual a *partição* do algoritmo é extremamente desbalanceada, ficando $n-1$ em uma partição e zero na outra. Essa situação vai fazer com que sejam realizadas muitas chamadas recursivas do quicksort (a cada chamada, apenas 1 elemento é eliminado). Como a pilha de execução é limitada, rapidamente ocorrerá um estouro de pilha.

Para resolver o problema, pesquise estratégias para selecionar um pivô de forma a evitar essa situação. Você pode escolher um pivô aleatoriamente ou recorre a regra das "*medianas-dos-três*", que envolve trocas condicionais entre o primeiro elemento, o último e o do meio.

6 Relatório Técnico

Projete e implemente, os algoritmos listados na Seção 4. A seguir, realize testes empíricos comparativos entre seus desempenhos para pelo menos 25 entradas de tamanhos diferentes, cobrindo os cenários descritos na Seção 3.

Em seguida, elabore um relatório técnico com:

1. Uma **introdução**, explicando o propósito do relatório;
2. Uma seção descrevendo o **método** seguido, ou seja quais foram os materiais e a metodologia utilizados.
   São exemplos de materiais a caracterização técnica do computador utilizado (i.e. o processador, memória, placa mãe, etc.), a linguagem de programação adotada, tipo e versão do sistema operacional, tipo e versão do compilador empregado, a lista de algoritmos implementado (com código), os cenários considerados, dentre outros. São exemplos de metodologia uma descrição do método ou procedimento empregado para gerar os dados do experimento, quais e como as medições foram tomadas para comparar os algoritmo (tempos, passos, memória), etc. Note que uma boa parte da descrição da metodologia já foi feita nesse próprio documento e poderá ser reaproveitada em seu relatório.
3. Os **resultados** alcançados (gráficos e tabelas).
   Após coletar e calcular a média dos tempos de execução para todos os tamanhos de instância n_i, seu objetivo é criar gráficos que demonstrem a curva de crescimento (tamanho da amostra no eixo X e tempo de execução no eixo Y) dos algoritmos. A sugestão é gerar, pelo menos, 2 gráficos por cenário proposto, totalizando 12 gráficos. Para cada gráfico do mesmo cenário agrupe algoritmos de mesma classe de complexidade. Por exemplo, para o cenário 75% dos elementos em ordem, gere um gráfico considerando apenas os algoritmos quadráticos $O(n^2)$ e outro com os algoritmos $O(n \lg n)$. Pro-

cure identificar como incluir o algoritmo radix nesses gráficos, visto que ele possui uma complexidade diferente dessas duas.

Uma possibilidade é utilizar *escala logarítmica* no eixo Y (tempo). Se você adotar essa estratégia, será possível gerar um gráfico com todos os algoritmos juntos, considerando um mesmo cenário. Mas mesmo nesse caso, ainda é recomendado gerar gráficos individuais ou por categorias com escala regular no eixo Y, visto que é a forma mais tradicional de visualizar esse tipo de gráfico.

4. A **discussão** dos resultados, no qual você deve responder as seguintes questões:
   (a) O que você descobriu de maneira geral?
   (b) Quais algoritmos são recomendados para quais cenários?
   (c) Como o algoritmo de decomposição de chave (radix) se compara com o melhor algoritmo baseado em comparação de chaves?
   (d) É verdade que o quick sort, na prática, é mesmo mais rápido que o merge sort?
   (e) Aconteceu algo inesperado nas medições? (por exemplo, picos ou vales nos gráficos) Se sim, por que?
   (f) Com base nos resultados empíricos, faça uma estimativa, para os 7 algoritmos estudados, de quanto tempo seria necessário para ordenar 10^{12} elementos.
   (g) A análise empírica é compatível com a análise matemática?

   Outras observações que você julgar interessante e pertinente ao trabalho, além destas, também podem ser acrescentadas à discussão. Uma boa discussão dependente da sua criatividade e/ou habilidade em contar uma "história" sobre o que aconteceu nos experimentos.

Um relatório técnico de algum valor acadêmico deve ser escrito de tal maneira que possibilite que uma outra pessoa que tenha lido o relatório consiga reproduzir o mesmo experimento. Este é o princípio científico da **reprodutibilidade**.

Note que para realizar uma comparação correta entre algoritmos, os *mesmos* valores gerados para cada tamanho n devem ser fornecidos como entrada para os algoritmos que estão sendo comparados.

Para a medição de tempo, recomenda-se a utilização da biblioteca `std::chrono` do C++, cuja descrição pode ser encontrada aqui.

7 Avaliação do Trabalho

O trabalho completo consiste do relatório e dos programa usados para gerar os dados presentes no relatório e devem ser de autoria própria. O trabalho completo vale 100 pontos, distribuídos de acordo com os seguintes itens:-

- ⋆ Relatório técnico (65 dos pontos):
  - Documento PDF com formato adequado, contendo capa, índice, seções citadas na Seção 6 e conclusão (16 pts).

  - Responde os itens 4a–4g da Seção 6, cada um valendo (7 pts)

⋆ Programas de comparação desenvolvido para gerar os dados (35 pts);

A pontuação acima não é definitiva e imutável. Ela serve apenas como um guia de como o trabalho será avaliado em linhas gerais. É possível a realização de ajustes nas pontuações indicadas visando adequar a pontuação ao nível de dificuldade dos itens solicitados.

Os itens abaixo correspondem à descontos, ou seja, pontos que podem ser retirados da pontuação total obtida com os itens anteriores:-

- Presença de erros de compilação e/ou execução (até -20%)
- Falta de documentação do programa com Doxygen (até -10%)
- Vazamento de memória (até -10%)
- Falta de preenchimento do arquivo `README.md` (ou `author.md`) contendo, entre outras coisas, identificação de autoria; instruções de como compilar e executar o programa; lista dos erros que o programa trata; e limitações e/ou problemas que o programa possui/apresenta, se for o caso (até -20%).

8 Autoria e Política de Colaboração

O trabalho pode ser realizado **individualmente** ou em **duplas**, sendo que no último caso é importante, dentro do possível, dividir as tarefas igualmente entre os componentes.

Qualquer equipe pode ser convocada para uma entrevista. O objetivo da entrevista é duplo: confirmar a autoria do trabalho e determinar a contribuição real de cada componente em relação ao trabalho. Durante a entrevista os membros da equipe devem ser capazes de explicar, com desenvoltura, qualquer trecho do trabalho, mesmo que o código tenha sido desenvolvido pelo outro membro da equipe. Portanto, é possível que, após a entrevista, ocorra redução da nota geral do trabalho ou ajustes nas notas individuais, de maneira a refletir a verdadeira contribuição de cada membro, conforme determinado na entrevista.

O trabalho em cooperação entre alunos da turma é estimulado. É aceitável a discussão de ideias e estratégias. Note, contudo, que esta interação **não** deve ser entendida como permissão para utilização de código ou parte de código de outras equipes, o que pode caracterizar a situação de plágio. Em resumo, tenha o cuidado de escrever seus próprios programas.

Trabalhos plagiados receberão nota **zero** automaticamente, independente de quem seja o verdadeiro autor dos trabalhos infratores. Fazer uso de qualquer assistência sem reconhecer os créditos apropriados é considerado **plagiarismo**. Quando submeter seu trabalho, forneça a citação e reconhecimentos necessários. Isso pode ser feito pontualmente nos comentários no início do código, ou, de maneira mais abrangente, no arquivo texto `README`. Além disso, no caso de receber assistência, certifique-se de que ela lhe é dada de maneira genérica, ou seja, de forma que não envolva alguém tendo que escrever código por você.

9 Entrega

Você deve submeter um único arquivo com a compactação da pasta do seu projeto, incluindo o relatório técnico no formato PDF. O arquivo compactado deve ser enviado **apenas** através da opção Tarefas da turma Virtual do Sigaa, em data divulgada no sistema.

Alternativamente, se você utilizar o GitHub Classroom, basta enviar pelo Sigaa a url do repositório no GitHub, que deve conter também o relatório em PDF.

$\sim$ FIM $\sim$