Algoritmos e Programação II — Exercício-Programa 2

Prof. Samuel Plaça de Paula

Centro Universitário Senac — São Paulo, novembro de 2021

Instruções gerais

A tarefa aqui descrita deve ser feita de maneira **individual ou em dupla**. Sua entrega será na atividade do Black Board e consistirá de:

- Um programa em linguagem C. Seu programa pode ser espalhado em vários arquivos, se assim desejar, ou também pode ser um único arquivo, por exemplo `ep3.c`;
- Um pequeno arquivo `.txt` informando nome e RA das pessoas integrantes da dupla.

Você fará a entrega na forma de um único arquivo `.zip` ou `.tar.gz` contendo tudo, e com o nome identificando os integrantes da dupla. Por exemplo, `samuel-e-suzana.zip`.

Descrição do problema

Nesse EP, nosso objetivo é testar na prática a velocidade de diferentes algoritmos de ordenação. Vamos criar vetores aleatórios (isto é, vetores com conteúdo gerado por sorteios simulados) e fornecê-los para todos os algoritmos que fazem parte do experimento. Queremos medir o tempo que cada um deles demora para terminar a tarefa de ordenar o mesmo vetor.

A ideia é que façamos testes com diferentes tamanhos de vetor. Como sempre discutido em sala de aula, tão importante quanto os tempos concretos, é saber quão *escalável* é cada algoritmo. Isto é, como o algoritmo lida com o aumento no tamanho da entrada? Seu tempo de execução tem um aumento razoável ou exagerado conforme aumentamos o tamanho do vetor? Frequentemente, esse tipo de pergunta não tem respostas absolutas. É algo que desejamos comparar entre diferentes algoritmos que realizam a mesma tarefa.

Neste EP, vamos comparar o tempo de execução dos algoritmos:

- Bubble Sort;
- Merge Sort.

Lembre-se que fizemos análises em sala de aula que nos fornecem parâmetros teóricos de comparação. Os algoritmos acima descritos têm tempo de execução, respectivamente: $O(n^2)$ e $O(n \log n)$. Sabendo disso, esperamos que o Bubble Sort seja mais lento, pelo menos a partir de um certo tamanho de vetor. (Qual tamanho seria esse? Vamos descobrir na prática.)

Obrigatoriamente, deverão ser testados vetores de tamanho $N = 10$, 20, 100, 200, 1000, 2000, 10 mil, 20 mil, 100 mil, 200 mil. Você pode fazer testes adicionais caso tenha curiosidade!

Sua tarefa

Parte 1 – gerar vetores aleatórios

Escreva uma função

```
int * vetor_aleatorio(int N)
```

que recebe um tamanho N e retorna um vetor aleatório de N posições, com valores no intervalo de 1 até $2N$. (Por exemplo, se $N = 100$, o vetor terá 100 posições e cada uma delas terá valor no intervalo de 1 a 200.)

Exemplo de uso da função:

```
int * v = vetor_aleatorio(1000);
// agora, v é um vetor de 1000 posições, preenchidas com números alea
```

Parte 2 – os testes

Para a Parte 2, você precisará ter em seu programa funções que implementam os algoritmos Bubble Sort e Merge Sort. É recomendável que você use as implementações fornecidas nos códigos que trabalhamos em sala de aula.

Seu programa deverá, **para cada** N na seguinte lista de valores: 10, 20, 100, 200, 1000, 2000, 10 mil, 20 mil, 100 mil, 200 mil; fazer o seguinte para cada algoritmo de ordenação testado:

- Gerar 10 vetores de tamanho N;
- Executar a ordenação (usando o algoritmo testado) sobre cada um desses vetores;

- Guardar o tempo médio de execução do algoritmo para esse N. **Mais adiante é explicado como medir esse tempo!**

Para cada N, seu programa imprimirá um pequeno relatório, como no exemplo a seguir:

```
N = 10000
Bubbl: 0.124 segundos em media
Merge: 0.005 segundos em media
```

Como são 10 valores diferentes de N, seu programa imprimirá 12 parágrafos similares ao do exemplo acima.

(Lembrando que você pode testar outros tamanhos de vetor além dos obrigatórios. Na minha experiência, para vetores de tamanho 1 milhão o Bubble Sort já fica chato de esperar, mas pode ser um teste interessante.)

Como medir o tempo de execução?

A biblioteca `time.h` fornece uma função `clock` que funciona como cronômetro que mede o tempo não em segundos mas em "tiques do relógio". Esse tempo é medido usando um tipo de dados próprio chamado `clock_t` ("clock type", tipo de dados relógio).

O uso típico para medir o tempo de execução de um determinado pedaço de código é assim:

```
#include <stdio.h>
#include <time.h>

int main() {
    clock_t inicio, final;
    double tempo;

    inicio = clock(); // mede tempo antes da tarefa

    // ALGUMA TAREFA AQUI! Exemplo:
    quicksort(v, 0, N-1);

    final = clock(); // mede tempo logo após a tarefa

    tempo = ((double) (final-inicio)/CLOCKS_PER_SEC);

    printf("A tarefa demorou %f segundos.\n", tempo);

    return 0;
}
```

Algumas reflexões sobre metodologia

Esse EP pretende ser um exemplo didático de uma área que chamamos de *computação científica*, em que usamos o computador para fazer testes empíricos sobre alguma coisa. Nesse caso, estamos testando a velocidade de alguns algoritmos de ordenação.

A tarefa, tal como aqui descrita, fornece apenas uma ideia de como fazer essa comparação entre algoritmos. Um teste mais sério deveria levar em consideração alguns outros aspectos, como:

- Em vez de gerar vetores novos para cada algoritmo, o ideal seria usar o mesmo conjunto de vetores para testar os dois algoritmos. Isso dá um pouco mais de trabalho e não deve fazer muita diferença, mas seria o mais correto. (Precisaríamos criar duas cópias de cada vetor aleatório gerado);
- Para cada N, seria interessante gerar muito mais vetores de tamanho N. O número 10 nos fornece uma boa ideia mas é pouco para um teste rigoroso;
- Além do tempo médio, gostaríamos de ter outras estatísticas, como mínimo, máximo, mediana e desvio padrão;
- Além do vetor aleatório, seria importante testar outros modelos de dados de entrada, tais como: vetor ordenado, vetor "invertido" (ordenado de forma decrescente), vetor "quase ordenado" (pequeno número de inversões). Isso permite flagrar em quais casos um determinado algoritmo se dá melhor ou pior. Por exemplo, o vetor ordenado e o "quase ordenado" são ótimos para o Bubble Sort e péssimos para o Quicksort (que não estamos testando aqui). O vetor invertido é o pior caso possível para o Bubble Sort. O Merge Sort, por sua vez, é completamente indiferente a essas variações.

Tudo isso foi deixado de lado para não complicar muito o EP. No semestre 4

do curso vocês realizarão um PI de computação científica em que poderão aprofundar nessas discussões.

Considerações finais

Será levado em conta na correção o seu cuidado em *liberar* (usando `free`) a memória alocada. Isso deve ser feito antes do programa terminar, como discutimos em aula.

Caso tenha dúvidas, fique à vontade para me procurar na aula ou no endereço `samuel.ppaula@sp.senac.br`.

É isso. Um ótimo trabalho! Boa sorte!