

Algoritmos e Estruturas de Dados II

Trabalho Prático 3

Entrega: 03/11/09

Devolução: 24/11/09

O trabalho pode ser feito em grupo de dois alunos.

Este trabalho consiste em analisar o desempenho de diferentes algoritmos de ordenação em diferentes cenários, descritos a seguir. Esta análise consistirá em comparar os algoritmos considerando três métricas de desempenho: número de comparações de chaves, o número de cópias de registros realizadas, e o tempo total gasto para ordenação (tempo de processamento e **não** o tempo de relógio). As entradas deverão ser conjuntos de elementos com chaves *aleatoriamente* geradas. Para obter o tempo de processamento na linguagem C, você **pode** utilizar o comando *getrusage*, lembrando de somar os tempos gastos em modo de usuário (*utime* ou *user time*) e em modo de sistema (*stime* ou *system time*) (vide exemplo no final do enunciado).

Cenário I: Impacto de diferentes estruturas de dados

Neste cenário, você deverá avaliar o desempenho do método de ordenação Quicksort (recursivo) considerando as seguintes entradas:

- Os elementos a serem ordenados são inteiros armazenados em um vetor de tamanho N .
- Os elementos a serem ordenados são inteiros armazenados em uma lista duplamente encadeada com N elementos.
- Os elementos a serem ordenados são registros armazenados em um vetor de tamanho N . Cada registro contém:
 - Um inteiro, a chave para ordenação.
 - Dez cadeias de caracteres (*strings*), cada uma como 200 caracteres.
 - 1 booleano
 - 4 números reais.

Para cada tipo de dado, você deverá implementar o Quicksort Recursivo (como apresentado em sala de aula) que recebe, como entrada, o conjunto de elementos a serem ordenados (uma Lista ou um vetor) e o número de elementos a serem ordenados. Você deverá instrumentar os algoritmos para contabilizar o número de comparações de chaves, o número de cópias de registros e o tempo total gasto na ordenação. Estes números deverão ser impressos ao final de cada ordenação para posterior análise.

Você ainda deverá implementar funções para criação dos conjuntos de elementos aleatórios. Estas funções devem ser chamadas uma vez para cada um dos N elementos a serem ordenados. Para o caso dos elementos serem registros, a função de criação deve inicializar todos os campos do registro com valores aleatórios. Note que dois elementos podem ter a mesma chave.

Análise:

O algoritmo Quicksort deverá ser aplicado a entradas aleatórias com diferentes tamanhos (parâmetro N). Para cada valor de N , você deve gerar 5 (cinco) conjuntos de elementos diferentes, utilizando sementes diferentes para o gerador de números aleatórios. Experimente, **no mínimo**, com valores de $N = 1000, 5000, 10000, 50000, 100000, 500000$ e 1000000 . Os algoritmos serão avaliados comparando os valores médios das 5 execuções para cada valor de N testado.

O seu programa principal deve ser organizado da seguinte maneira:

- Recebe a semente do gerador de números aleatórios bem como os nomes dos arquivos de entrada e de saída. Estes parâmetros devem ser passados pela linha de comando (*argc* e *argv* em C). Por exemplo:

```
quicksort 10 entrada.txt saida10.txt
```

onde entrada.txt tem o formato:

```
7 -> número de valores de N que se seguem, um por linha
1000
5000
10000
50000
100000
500000
1000000
```

- Para cada valor de N , lido do arquivo de entrada:

- Gera cada um dos conjuntos de elementos, ordena, contabiliza estatísticas de desempenho
- Armazena estatísticas de desempenho em arquivo de saída

Ao final, basta processar os arquivos de saída referentes a cada uma das seções, calculando as médias de cada estatística, para cada valor de N e estrutura de dados considerados.

Resultados:

Apresente gráficos e/ou tabelas para as três métricas pedidas, número de comparações, número de cópias e tempo de execução (tempo de processamento), comparando o **desempenho médio** do Quicksort para os três tipos de estruturas de dados e diferentes valores de N . Discuta seus resultados. Quais são os compromissos de desempenho observados?

Cenário 2: Impacto de variações do Quicksort

Neste cenário, você deverá comparar o desempenho de diferentes variações do Quicksort para ordenar um conjunto de N inteiros armazenados em um vetor. As variações do Quicksort a serem implementadas e avaliadas são:

- Quicksort_Recursivo: este é o Quicksort recursivo apresentado em sala de aula.
- Quicksort_Mediana(k): esta variação do Quicksort recursivo escolhe o pivô para partição como sendo a *mediana* de k elementos do vetor, aleatoriamente escolhidos. Experimente com $k = 3$ e $k = 5$.
- Quicksort_Insercao(m): esta variação modifica o Quicksort Recursivo para utilizar o algoritmo de Inserção para ordenar partições (isto é, pedaços do vetor) com tamanho menor ou igual a m . Experimente com $m = 10$ e $m = 100$.
- Quicksort_Empilha_Inteligente(): esta variação otimizada do Quicksort Recursivo processa primeiro o lado menor da partição.
- Quicksort_Iterativo: esta variação escolhe o pivô como o elemento do meio (como apresentado em sala de aula), mas não é recursiva. Em outras palavras, esta é uma versão iterativa do Quicksort apresentado em sala de aula.
- Quicksort_Empilha_Inteligente(): esta variação otimizada do Quicksort Iterativo processa primeiro o lado menor da partição.

Realize experimentos com as cinco variações considerando vetores aleatoriamente gerados com tamanho $N = 1000, 5000, 10000, 50000, 100000, 500000$ e 1000000 , no mínimo. Para cada valor de N , realize experimentos com 5 sementes diferentes e avalie os valores médios do tempo de execução, do número de comparações de chaves e do número de cópias de registros. Estruture o seu programa principal como sugerido acima para facilitar a coleta e posterior análise das estatísticas de desempenho.

Apresente gráficos e/ou tabelas com os resultados obtidos. Discuta os resultados e conclusões obtidos. Qual variação tem melhor desempenho, considerando as diferentes métricas. Por quê? Qual o impacto das variações nos valores de k e de m nas versões `Quicksort_Mediana(k)` e `Quicksort_Insercao(m)`?

Cenário 3: Quicksort X Mergesort X Heapsort X Radixsort X Meusort

Neste cenário, você comparará a melhor variação do Quicksort (justificada pelos resultados do Cenário 2) com o Mergesort, o Heapsort, uma versão do Radixsort, e um outro algoritmo de ordenação de sua escolha (não pode ser nenhum algoritmo dado em sala de aula) para ordenar um conjunto de N inteiros *positivos*, aleatoriamente gerados, armazenados em um vetor. Você pode escolher qualquer uma das duas versões do Radixsort apresentadas em sala de aula, mas deverá explicitar sua escolha na documentação apresentada. Você deverá também apresentar, no seu relatório o algo quinto algoritmo escolhido, explicitando sua fonte.

Realize experimentos considerando vetores aleatoriamente gerados com tamanho $N=1000, 5000, 10000, 50000, 100000, 500000, 1000000$, no mínimo. Para cada valor de N , realize experimentos com 5 sementes diferentes. Para a comparação de Quicksort, Mergesort, Heapsort e Meusort, avalie os valores médios do tempo de execução, do número de comparações de chaves e do número de cópias de registros. A comparação do Radixsort com os outros três algoritmos deve focar apenas no tempo médio de execução. Apresente gráficos e/ou tabelas com os resultados obtidos. Discuta os resultados e conclusões obtidas. Qual algoritmo tem melhor desempenho, considerando as diferentes métricas. Por quê?

Note que o grande desafio deste trabalho está na avaliação dos vários algoritmos nos diferentes cenários, e não na implementação de código. Logo, na divisão de pontos, a documentação receberá, no mínimo, 50% dos pontos totais. Uma boa documentação deverá apresentar não somente resultados brutos mas também uma discussão dos mesmos, levando a conclusões sobre a superioridade de um ou outro algoritmo em cada cenário considerado, para cada métrica avaliada.

Medindo o tempo de execução de uma função em C:

O comando `getrusage()` é parte da biblioteca padrão de C da maioria dos sistemas Unix. Ele retorna os recursos correntemente utilizados pelo processo, em particular os tempos de processamento (tempo de CPU) em modo de usuário e em modo sistema, fornecendo valores com granularidades de segundos e microssegundos. Um exemplo que calcula o tempo total gasto na execução de uma tarefa é mostrado abaixo:

```
#include <stdio.h>
#include <sys/resource.h>

void main () {
    struct rusage resources;
    int    rc;
    double utime, stime, total_time;

    /* do some work here */

    if((rc = getrusage(RUSAGE_SELF, &resources)) != 0)
        perror("getrusage failed");

    utime = (double) resources.ru_utime.tv_sec
            + 1.e-6 * (double) resources.ru_utime.tv_usec;
    stime = (double) resources.ru_stime.tv_sec
            + 1.e-6 * (double) resources.ru_stime.tv_usec;
    total_time = utime+stime;
    printf("User time %.3f, System time %.3f, Total Time %.3f\n",
           utime, stime, total_time);
}
```