Estruturas de Dados e Algoritmos
Série de Problemas
2021

José Jasnau Caeiro

26 de abril de 2021

Conteúdo

1 Complexidade Computacional e Algoritmos de Ordenação

Esta série de problemas é relativa às aulas teóricas 1, 3, 3 e 4.

1.1 Introdução ao Problema da Ordenação

Estes problemas correspondem à aula teórica 1.

- problema da ordenação
- algoritmo de ordenação Insertion-Sort
- complexidade computacional
- taxas de crescimento dos tempos de execução e de utilização de memória
- notações assintóticas
- algoritmo de ordenação Bubble-Sort
- medição experimental
- noção de protocolo experimental
- gráficos dos tempos de execução

Crie uma sub-pasta da pasta `/repositorio/eda2020/problemas` designada por `p1`. Utilize o seguinte comando para criar esta pasta:

```
nomeXXXXX@odin:~/repositorio/eda2020/problemas$ cargo new p1
```

Resolva os seguintes problemas:

1. (a) Edite o ficheiro `src/main.rs`.
   (b) Crie uma função que devolve uma tabela com 1000 números aleatórios do tipo `i32` uniformemente distribuídos entre 0 e 99 inclusivé. Designe a tabela resultante por `a1`. Consulte a documentação alargada disponível em The Rust Programming Language. The Rust Rand Book.
   (c) Crie uma função que devolve uma tabela com 100 entradas em que cada entrada apresenta o número de inteiros correspondente ao índice presente em `a1`.
   (d) Crie uma função que devolve a média duma tabela. Aplique à tabela `a1`.
   (e) Crie uma função que devolve o desvio-padrão duma tabela. Aplique à tabela `a1`.
   (f) Crie uma função que devolve a variância duma tabela. Aplique à tabela `a1`.
2. (a) Programe uma versão recursiva da função de *Fibonacci*.
   (b) Programe uma versão iterativa da função de *Fibonacci*.
   (c) Programe uma versão direta da função de *Fibonacci*.
   (d) Meça os tempos de execução de cada versão para valores entre $n = 10$ e $n = 40$.
3. (a) Crie uma função que devolve uma tabela com 10 números aleatórios do tipo `i32` uniformemente distribuídos entre 0 e 99 inclusivé. Designe a tabela resultante por `a1`.
   (b) Crie uma função que ordene a tabela `a1` usando o algoritmo Insertion-Sort.
   (c) Crie uma função que tem como argumento N. Esta função deve devolver uma tabela de N números aleatórios do tipo $i32$ uniformemente distribuídos entre 0 e 99 inclusivé.

(d) Aplique o algoritmo Insertion-Sort sucessivamente a tabelas com dimensões entre $N = 10$ e $N = 40$. Meça os tempos de execução para cada valor de N.

4. (a) Crie uma função de ordenação de tabelas que realiza o algoritmo Bubble-Sort.

   (b) Aplique o algoritmo Bubble-Sort sucessivamente a tabelas com dimensões entre $N = 10$ e $N = 40$. Meça os tempos de execução para cada valor de N.

   (c) Tente determinar os tempos de execução de forma a que o erro relativo da medida seja inferior a 5%.

5. (a) Determine experimentalmente qual o valor de N que faz com que o algoritmo Insertion-Sort demore aproximadamente 60 segundos a ordenar uma tabela de números aleatórios do tipo `i32`.

   (b) Determine experimentalmente qual o valor de N que faz com que o algoritmo Bubble-Sort demore aproximadamente 60 segundos a ordenar uma tabela de números aleatórios do tipo `i32`.

   (c) Determine os tempos de execução dos algoritmos para valores de $n = 10$ até $n = N$ com erro relativo máximo inferior a 5%.

   (d) Escreva num ficheiro designado por `tempos.txt` os dados resultantes das medidas de tempo de execução. Em cada linha há várias colunas. As colunas devem ser separadas por um espaço e são:

      1. n, a dimensão da tabela a ordenar;
      2. $t_{insertion}$, tempo de execução do algoritmo Insertion-Sort;
      3. $\delta_{insertion}$, desvio padrão da medida do tempo de execução do algoritmo Insertion-Sort;
      4. $\epsilon_{insertion}$, erro relativo da medida do tempo de execução do algoritmo Insertion-Sort;
      5. t_{bubble}, tempo de execução do algoritmo Bubble-Sort;
      6. δ_{bubble}, desvio padrão da medida do tempo de execução do algoritmo Bubble-Sort;
      7. ϵ_{bubble}, erro relativo da medida do tempo de execução do algoritmo Bubble-Sort.

   (e) Use a linguagem de programação `Python` e a biblioteca `matplotlib` para produzir gráficos representado os tempos de execução comparativos dos diversos algoritmos.

   (f) Produza gráficos representando os erros relativos comparativos dos diversos algoritmos.

1.2 Divisão e Conquista

Estes problemas correspondem à aula teórica 2.

- algoritmo de ordenação Merge-Sort

Crie uma sub-pasta da pasta `/repositorio/eda2020/problemas` designada por `p2`. Utilize o seguinte comando para criar esta pasta:

```
nomeXXXXX@odin:~/repositorio/eda2020/problemas$ cargo new p2
```

Resolva os seguintes problemas:

1. (a) Edite o ficheiro `src/main.rs`.

   (b) Crie uma função que devolve uma tabela com 10 números aleatórios do tipo `i32` uniformemente distribuídos entre 0 e 99 inclusivé. Designe a tabela resultante por `a1`.

   (c) Crie uma função que ordene a tabela `a1` usando o algoritmo Merge-Sort.

   (d) Crie uma função que tem como argumento N. Esta função deve devolver uma tabela de N números aleatórios do tipo $i32$ uniformemente distribuídos entre 0 e 99 inclusivé.

(e) Aplique o algoritmo MERGE-SORT sucessivamente a tabelas com dimensões entre $N = 10$ e $N = 4000$, saltando de 100 em 100. Meça os tempos de execução para cada valor de N.

2. (a) Determine experimentalmente qual o valor de N que faz com que o algoritmo MERGE-SORT demore aproximadamente 60 segundos a ordenar uma tabela de números aleatórios do tipo `i32`.

   (b) Determine os tempos de execução dos algoritmos para valores de $n = 10$ até $n = N$ com erro relativo máximo inferior a 5%.

   (c) Escreva num ficheiro designado por `tempos.txt` os dados resultantes das medidas de tempo de execução. Em cada linha há várias colunas. As colunas devem ser separadas por um espaço e são:

      1. n, a dimensão da tabela a ordenar;
      2. t_{merge}, tempo de execução do algoritmo MERGE-SORT;
      3. δ_{merge}, desvio padrão da medida do tempo de execução do algoritmo MERGE-SORT;
      4. ϵ_{merge}, erro relativo da medida do tempo de execução do algoritmo MERGE-SORT;

   (d) Use a linguagem de programação `Python` e a biblioteca `matplotlib` para produzir gráficos representado os tempos de execução do algoritmo.

1.3 Heap Sort

Estes problemas correspondem à aula teórica 3.

- algoritmo de ordenação HEAP-SORT

Crie uma sub-pasta da pasta `/repositorio/eda2020/problemas` designada por `p3`. Utilize o seguinte comando para criar esta pasta:

```
nomeXXXXX@odin:~/repositorio/eda2020/problemas$ cargo new p3
```

Resolva os seguintes problemas:

1. (a) Edite o ficheiro `src/main.rs`.

   (b) Crie uma função que devolve uma tabela com 10 números aleatórios do tipo `i32` uniformemente distribuídos entre 0 e 99 inclusivé. Designe a tabela resultante por `a1`.

   (c) Crie uma função que ordene a tabela `a1` usando o algoritmo HEAP-SORT.

   (d) Crie uma função que tem como argumento N. Esta função deve devolver uma tabela de N números aleatórios do tipo $i32$ uniformemente distribuídos entre 0 e 99 inclusivé.

   (e) Aplique o algoritmo HEAP-SORT sucessivamente a tabelas com dimensões entre $N = 10$ e $N = 4000$, saltando de 100 em 100. Meça os tempos de execução para cada valor de N.

2. (a) Determine experimentalmente qual o valor de N que faz com que o algoritmo HEAP-SORT demore aproximadamente 60 segundos a ordenar uma tabela de números aleatórios do tipo `i32`.

   (b) Determine os tempos de execução dos algoritmos para valores de $n = 10$ até $n = N$ com erro relativo máximo inferior a 5%.

   (c) Escreva num ficheiro designado por `tempos.txt` os dados resultantes das medidas de tempo de execução. Em cada linha há várias colunas. As colunas devem ser separadas por um espaço e são:

      1. n, a dimensão da tabela a ordenar;
      2. t_{heap}, tempo de execução do algoritmo HEAP-SORT;
      3. δ_{heap}, desvio padrão da medida do tempo de execução do algoritmo HEAP-SORT;
      4. ϵ_{heap}, erro relativo da medida do tempo de execução do algoritmo HEAP-SORT;

(d) Use a linguagem de programação `Python` e a biblioteca `matplotlib` para produzir gráficos representado os tempos de execução do algoritmo.

3. (a) Crie uma estrutura de dados designada por **Cidades** com os seguintes campos: uma chave do tipo inteiro; uma *string* para representar o nome duma cidade; uma entrada do tipo *double (float64)* para a latitude e uma entrada com a longitude.

   (b) Modifique o algoritmo HEAP-SORT de forma a que se aplique uma tabela de **Cidades** e em que a métrica de ordenação é dada em função da distância de cada cidade até **Lisboa**.

1.4 Quick Sort

Estes problemas correspondem à aula teórica 4.

- algoritmo de ordenação QUICK-SORT

Crie uma sub-pasta da pasta `/repositorio/eda2020/problemas` designada por `p4`. Utilize o seguinte comando para criar esta pasta:

```
nomeXXXXX@odin:~/repositorio/eda2020/problemas$ cargo new p4
```

Resolva os seguintes problemas:

1. (a) Edite o ficheiro `src/main.rs`.

   (b) Crie uma função que devolve uma tabela com 10 números aleatórios do tipo `i32` uniformemente distribuídos entre 0 e 99 inclusivé. Designe a tabela resultante por `a1`.

   (c) Crie uma função que ordene a tabela `a1` usando o algoritmo QUICK-SORT.

   (d) Crie uma função que tem como argumento N. Esta função deve devolver uma tabela de N números aleatórios do tipo $i32$ uniformemente distribuídos entre 0 e 99 inclusivé.

   (e) Aplique o algoritmo QUICK-SORT sucessivamente a tabelas com dimensões entre $N = 10$ e $N = 4000$, saltando de 100 em 100. Meça os tempos de execução para cada valor de N.

2. (a) Determine experimentalmente qual o valor de N que faz com que o algoritmo QUICK-SORT demore aproximadamente 60 segundos a ordenar uma tabela de números aleatórios do tipo `i32`.

   (b) Determine os tempos de execução dos algoritmos para valores de $n = 10$ até $n = N$ com erro relativo máximo inferior a 5%.

   (c) Escreva num ficheiro designado por `tempos.txt` os dados resultantes das medidas de tempo de execução. Em cada linha há várias colunas. As colunas devem ser separadas por um espaço e são:

      1. n, a dimensão da tabela a ordenar;
      2. t_{quick}, tempo de execução do algoritmo QUICK-SORT;
      3. δ_{quick}, desvio padrão da medida do tempo de execução do algoritmo QUICK-SORT;
      4. ϵ_{quick}, erro relativo da medida do tempo de execução do algoritmo QUICK-SORT;

   (d) Use a linguagem de programação `Python` e a biblioteca `matplotlib` para produzir gráficos representado os tempos de execução do algoritmo.

3. (a) Crie uma estrutura de dados designada por **Cidade** com os seguintes campos: uma chave do tipo inteiro; uma *string* para representar o nome duma cidade; uma entrada do tipo *double (float64)* para a latitude e uma entrada com a longitude.

   (b) Modifique o algoritmo QUICK-SORT de forma a que se aplique uma tabela de **Cidades** e em que a métrica de ordenação é dada em função da distância de cada cidade até **Lisboa**.

4. (a) Use a biblioteca `matplotlib` de `Python` para comparar num gráfico os limites assintóticos superiores, no pior cenário, dos algoritmos: INSERTION-SORT, MERGE-SORT e QUICK-SORT.

(b) Use a biblioteca `matplotlib` de `Python` para comparar num gráfico os limites assintóticos superiores, no melhor cenário, dos algoritmos: Insertion-Sort, Merge-Sort e Quick-Sort.

2 Estruturas de Dados Elementares

Esta série de problemas é relativa às aulas teóricas sobre:

- aula 5
  - conjuntos dinâmicos;
  - pilhas;
  - filas de espera;
  - listas ligadas;
  - gestão de memória de listas ligadas;
  - árvores binárias;
  - árvores n-árias.
- aula 6
  - *hash-tables*
  - endereçamento direto;
  - resolução de colisões;
  - funções de *hash*;
  - método da divisão;
  - método da multiplicação;
  - endereçamento aberto;
  - teste linear e fenómeno da agregação primária;
  - teste quadrático e fenómeno de agregação secundária;
  - dupla função de *hash*;
- aula 7
  - árvores de pesquisa binária;
  - pesquisa, inserção e remoção de nós;
- aula 8
  - árvores de pesquisa binária balanceadas;
  - árvores `RED-BLACK`;
  - pesquisa, inserção e remoção de nós.

2.1 Estruturas de Dados Elementares

Crie uma sub-pasta da pasta `/repositorio/eda2020/problemas` designada por `p5`. Utilize o seguinte comando para criar esta pasta:

```
nomeXXXXX@odin:~/repositorio/eda2020/problemas$ cargo new p5
```

Resolva os seguintes problemas:

1. (a) Crie uma estrutura de dados do tipo pilha de inteiros.
   (b) Crie uma estrutura de dados do tipo pilha para *strings*.
   (c) Experimente as funções `pop()` e `push()`. Tente ultrapassar os limites de endereçamento das pilhas e processar os seus erros.

(d) Programe um tipo de dados que representa um cartão de cidadão. Experimente colocar este tipo de objetos numa pilha com a operação `push` e a sua remoção com a operação `pop()`.

2. (a) Crie uma fila de espera que permita armazenar até 20 elementos inteiros.
   (b) Crie as funções que operam com uma fila de espera.
   (c) Crie uma fila de espera para cartões de cidadão.
   (d) Torne as operações relacionadas com a fila de espera robustas.
3. (a) Programe uma lista duplamente ligada que permita colocar cartões de cidadão.
   (b) Programe um gestor de memória que permita operar com várias listas ligadas.
4. (a) Programe um gestor de memória para árvores binárias.
   (b) Programe uma árvore binária em que coloca informação sobre cartões de cidadão.
   (c) Programe as operações típicas de conjuntos dinâmicos para estas árvores binárias.
5. (a) Programe um gestor de memória para árvores n-árias.
   (b) Programe uma árvore quaternária que armazena informação sobre cartões de cidadão.
   (c) Programe as funções usadas em conjuntos dinâmicos para operarem sobre estas árvores quaternárias.

2.2 Hash Tables

Crie uma sub-pasta da pasta `/repositorio/eda2020/problemas` designada por `p6`. Utilize o seguinte comando para criar esta pasta:

```
nomeXXXXX@odin:~/repositorio/eda2020/problemas$ cargo new p6
```

Resolva os seguintes problemas:

1. (a) Crie uma *tabela de endereçamento direto* que relacione nomes completos com um objeto que representa um cartão de cidadão.
   (b) Crie uma *hast-table* com resolução de colisões. Admita que vai adotar cerca de 13 listas ligadas.
   (c) Programe uma função de *hash* pelo método da **divisão**.
   (d) Programe uma função de *hash* pelo método da **multiplicação**.
2. (a) Crie uma *hash-table* adotando endereçamento aberto. Esta *hash-table* deve relacionar nomes de cidades com o seu número de habitantes. Programe as funções relacionadas.
   (b) Programe a inserção com teste linear.
   (c) Programe a inserção com teste quadrático.
   (d) Programe a inserção com dupla função de *hash*