Trabalho 1 - Análise de Algoritmos

Entrega: 1 de Maio

Esse trabalho consiste em implementar o algoritmo de seleção em tempo linear visto em sala, e comparar com o algoritmo de seleção baseado em ordenação.

O trabalho deve ser realizado **em dupla**, e pode ser feito em qualquer linguagem. Ele deve ser entregue **no EAD**, por um dos integrantes (uma submissão por dupla).

O que tem que ser entregue:

1. Relatório de 2-3 páginas com o conteúdo pedido abaixo
2. Código fonte
3. Caso o código não seja feito em Python, **executável windows**

Note que posso utilizar sistema para controle de plágio de código, como o MOSS.[1]

Tarefa 1: Seleção em tempo linear

Implemente o procedimento LinearSelection(A, k) visto em sala que recebe uma lista A de números[2] e um inteiro k e retorna o k-ésimo menor elemento em A (ou seja, caso a lista esteja ordenada, deve retornar o número na k-ésima posição). Caso k esteja fora dos limites (i.e., ≤ 0 ou $> |A|$), retorne mensagem de erro/exceção. **A implementação tem que ser complexidade de tempo linear no pior-caso.** Além disso, tem que funcionar mesmo quando existem **números repetidos** na entrada.

Como base do código utilize os slides de aula, o Capítulo 9.3 do livro-texto "Introdução a Algoritmos" de Cormen, Leiserson, Rivest, e Stein, ou qualquer outra referência. Lembre de definir o caso-base da recursão, e de tratar o caso de números repetidos.

No relatório você deve colocar:

1. Uma cópia da parte principal da função LinearSelection(A, k) (deve ter uma estrutura semelhante à do pseudo-código dos slides de aula)
2. Indique **brevemente** porque o seu código satisfaz uma equação de recorrência da forma

$$T(n) \leq cst \cdot n + T(n/5) + T(7n/10)$$

   (só precisa justificar a parte em azul).
3. Breve indicação de como rodar seu executável e formato da entrada esperado.

[1] `https://theory.stanford.edu/~aiken/moss`

[2] Pode fixar o tipo da entrada como lista/vetor de int/float/etc. caso ajude, não precisa fazer um código muito genérico.

Tarefa 2: Experimentos

Realize experimentos comparando o LinearSelection(A, k) implementado contra o algoritmo seguinte algoritmo SortSelection(A, k) baseado no **BubbleSort**:

```
SortSelection(A, k):
  Aord = BubbleSort(A)
  Return Aord[k]
```

Note que voce tem que usar o **BubbleSort** como ordenação, não pode usar outros algoritmos.[3]
Para os experimentos:

1. Sete sempre $k = \lfloor n/2 \rfloor$ (metade do tamanho da lista de entrada arredondada para baixo)
2. Para cada $n = 1.000, 2.000, \ldots, 10.000$, gere 10 instâncias **aleatórias** de tamanho n (total de 100 instâncias). Os números em cada instância devem ser gerados aleatoriamente entre 1 e 100.000 (inteiros ou float/double).
3. Rode as algoritmos LinearSelection(A, k) e SortSelection(A, k) sobre essas instâncias. Para testar a corretude, teste que o valor retornado pelos algoritmo é o mesmo.
4. Para cada tamanho de instância n, compute a **média do tempo de execução** de cada um dos algoritmos (ou seja, compute 10 médias para cada algoritmo).

No relatório você deve reportar:

1. Um gráfico com o eixo x correspondendo ao tamanho da entrada n, e o eixo y correspondendo às médias dos tempos de execução dos algoritmos. Portanto o gráfico tem 2 séries, uma para cada algoritmo.
2. Compare **brevemente** esse gráfico com a complexidade teórica dos algoritmos.

[3] Por que **BubbleSort**? Porque o efeito que vocês verão não aparece de forma tão pronunciada utilizando outros algoritmos de ordenação. De fato, caso utilize algoritmos com bastante engenharia dentro (e.g., da STL do C++), esse efeito só ocorre para entradas muito grande, especialmente para entradas aleatórias como utilizamos aqui.