4ªaula prática

Algoritmos de ordenação

- Faça download do ficheiro ***aed2223_p04.zip*** da página e descomprima-o (contém a pasta ***lib***, a pasta ***Tests*** com os ficheiros ***funSortProblem.h***, ***funSortProblem.cpp***, ***product.h***, ***product.cpp***, ***piece.cpp***, ***piece.h*** e ***tests.cpp***, e os ficheiros ***CMakeLists*** e ***main.cpp***)
- No CLion, abra um ***projeto***, selecionando a pasta que contém os ficheiros do ponto anterior.

1. Implemente a função:

void FunSortProblem::expressLane(vector<Product> &products, unsigned k)

O João foi ao supermercado e, como está com pressa, pretende usar a caixa expresso para efetuar o pagamento das suas compras. A caixa expresso limita o número de produtos a comprar a *k* unidades, pelo que o João tem de escolher *k* produtos de entre os que pretendia comprar inicialmente (*products*). Como o João é ganancioso, vai escolher os produtos mais baratos; se dois produtos têm o mesmo preço, escolhe o mais leve. Considere que nunca existem dois produtos com o mesmo preço e peso. Esta função remove do vetor *products* os produtos que o João opta por não comprar, mantendo neste vetor os produtos escolhidos pelo João.

Implemente esta função usando apenas estruturas de dados lineares, e algum(uns) algoritmo(s) de ordenação.

Qual a complexidade temporal desta função?

Nota: os elementos do vetor são objetos da classe **Product**. Pode alterar e/ou adicionar membros a esta classe.

Exemplo de execução

input: *products* = [(pasta, 3.2, 500), (peanuts, 5.4, 700), (cookies, 4.2, 600), (milkas, 5.4, 600)]
, sendo Product = (id, preço, peso)

k = 3

output: [(pasta, 3.2, 500), (cookies, 4.2, 600), (milkas, 5.4, 600)]

os 3 produtos mais baratos são *pasta*, *cookies* e *milkas* (mesmo preço que *peanuts*, mas mais leve)

2. Implemente a função:

int FunSortProblem::minDifference(const vector<unsigned> &values, unsigned nc)

O vetor *values* representa vários pacotes de chocolate, onde cada inteiro representa o número de chocolates de um dos pacotes. Os pacotes de chocolate serão distribuídos por *nc* crianças de modo que:

- cada criança recebe exatamente um pacote
- a diferença entre o maior e o menor número de chocolates dados a uma criança é mínima.

A função retorna a diferença mínima entre o maior e menor número de chocolates dados a uma criança. Se o número de crianças é superior ao número de pacotes de chocolate existentes, a função retorna -1. Implemente esta função usando apenas estruturas de dados lineares, e algum(uns) algoritmo(s) de ordenação.

Complexidade temporal esperada: $O(n \times \log n)$

Exemplo de execução

input: *values* = [3, 4, 1, 9, 56, 7, 9, 12] , *nc* = 5

output = 6 (chocolates entregues: 3, 4, 7, 9, 9)

3. Implemente a função:

unsigned FunSortProblem::minPlatforms(const vector<float> &arrival, const<float> &departure)

Dados os horários de chegada (*arrival*) e partida (*departure*) de todos os comboios que chegam a uma estação ferroviária, encontre o número mínimo de plataformas necessárias, de modo que nenhum comboio fique em espera. Em qualquer momento, a mesma plataforma não pode ser usada tanto para a partida de um comboio como para a chegada de outro. A função retorna o número mínimo de plataformas que a estação ferroviária deve ter.

Os vetores *arrival* e *departure* representam o horário de chegada e partida dos comboios, respetivamente, sendo cada elemento do tipo float em que a parte inteira representa a hora e a parte decimal os minutos (ex: 9.50 significa 9h50minutos). A hora de partida de um comboio acontece após a sua chegada, pelo que $departure[i] > arrival[i], \forall i.$

Implemente esta função usando apenas estruturas de dados lineares, e algum(uns) algoritmo(s) de ordenação.

Complexidade temporal esperada: $O(n \times \log n)$

4. Implemente a função:

unsigned FunSortProblem::numInversions(const vector<int> &v)

O número de inversões num vetor indica quão longe o vetor está de se encontrar ordenado. Note que num vetor ordenado (na ordem desejada), o número de inversões é zero.
A função a implementar deve descobrir quantas inversões existem no vetor *v*. Dois elementos $v[i]$ e $v[j]$ formam uma inversão, se $v[i] > v[j]$ e $i < j$.
Complexidade temporal esperada: $O(n \times \log n)$

Exemplo de execução
input: *v*=[10, 50, 20, 40, 30]
output =4 , inversões encontradas: (50,20), (50,40), (50,30), (40,30)

Sugestão:
Efetue a contagem do número de inversões durante a ordenação do vetor usando o algoritmo *Merge Sort*. Neste algoritmo, na passo em que se juntam as duas metades ordenadas do vetor, podem encontrar-se as inversões existentes (como?)

Exercício Extra

5. Implemente a função:

void FunSortProblem::nutsBolts(vector<Piece> &nuts, vector<Piece> &bolts)

O vetor *nuts* representa um conjunto de porcas (objetos da classe *Piece*) identificadas por um *id* e *diâmetro*. O vetor *bolts* representa um conjunto de parafusos (objetos da classe *Piece*) identificados por um *id* e *diâmetro*. Cada porca corresponde exatamente a um parafuso e cada parafuso corresponde exatamente a uma porca. Uma porca e um parafuso são correspondentes se e só se possuem o mesmo diâmetro.
É possível comparar uma porca com um parafuso, mas não é possível comparar diretamente duas porcas ou dois parafusos. A função deve atualizar os vetores *nuts* e *bolts* que deverão conter no mesmo índice a porca e o parafuso correspondentes.
Implemente esta função usando apenas estruturas de dados lineares.
Complexidade temporal esperada: $O(n \times \log n)$.

Sugestão: baseie-se no conceito de partição do algoritmo QuickSort. Escolha um parafuso aleatório, compare-o com todas as porcas e encontre a porca correspondente. Compare a porca correspondente agora encontrada com todos os parafusos, dividindo assim o problema em dois problemas: um consistindo em porcas e parafusos menores que o par encontrado e outro consistindo em porcas e parafusos maiores que o par encontrado.