Programação II

Exercícios 3
Busca e ordenação

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Tecnologias da Informação

2020/2021

1. Considere a seguinte lista:
   `['a', 'f', 'h', 'e', 'a', 'z', 'b', 'c', 'd']`.

   (a) Usando o algoritmo de busca linear, para determinar se cada um dos seguintes elementos está presente na lista, quantas comparações são feitas:

      i. `'z'`
      ii. `'a'`
      iii. `'x'`

   (b) Considerando uma qualquer lista de tamanho n, qual é a complexidade (em notação $\mathcal{O}$) da busca linear no melhor caso, no pior caso, e no caso médio?

   (c) Seria possível usar um algoritmo de busca dicotómica para melhorar a eficiência da busca nesta lista em concreto? Justifique.

2. Considere a seguinte lista:
   `[5, 6, 7, 10, 20, 21, 25, 28, 29, 31, 50]`

   (a) Usando o algoritmo de busca dicotómica, para determinar se cada um dos seguintes elementos está presente na lista, indique quantas comparações são feitas:

      i. `7`
      ii. `29`
      iii. `5`
      iv. `21`

(b) É possível usar esse mesmo algoritmo de busca dicotómica se a lista estivesse com a ordem invertida? Isto é:
`[50, 31, 29, 28, 25, 21, 20, 10, 7, 6, 5]`
Que modificações teria que fazer no algoritmo para tal ser possível?

(c) Considerando uma qualquer lista ordenada de tamanho n, qual é a complexidade (em notação $\mathcal{O}$) da busca dicotómica no melhor caso, no pior caso, e no caso médio?

3. Recorde a implementação recursiva da busca dicotómica. Implemente uma versão iterativa desse mesmo algoritmo e analise a sua complexidade.

```
def busca_dicotomica(l, e):
    def busca(primeiro, ultimo):
        if primeiro > ultimo:
            return False
        meio = (primeiro + ultimo) // 2
        if l[meio] == e:
            return True
        if l[meio] < e:
            return busca(meio + 1, ultimo)
        return busca(primeiro, meio - 1)
    return busca(0, len(l) - 1)
```

4. Considere o algoritmo de ordenação por inserção (*insert sort*).

(a) Utilize este algoritmo para ordenar a lista `[4, 9, 3, 7]` por ordem crescente. Apresente os valores intermédios a cada novo estado da lista e indique o número de comparações e de trocas realizadas cumulativamente.

(b) Repita o exercício para a lista `[9, 7, 4, 3]`.

(c) Apresente a análise da complexidade deste algoritmo (em notação $\mathcal{O}$) para o melhor caso e para o pior, relativamente a comparações e a trocas envolvidas.

5. Considere a seguinte descrição de um algoritmo de ordenação (*bubble sort*):[1]

```
enquanto a lista não estiver ordenada:
    para cada par de elementos adjacentes:
        se o par está fora de ordem:
            ordená-lo
```

(a) Implemente este algoritmo de ordenação em Python.

[1] *Barack Obama—Computer Science Question*

(b) Utilize o algoritmo para ordenar a lista `[4, 9, 3, 7]` por ordem crescente, apresentando os valores intermédios a cada novo estado da lista e indicando o número de comparações e de trocas realizadas cumulativamente.

(c) Repita para a lista `[9, 7, 4, 3]`.

(d) Apresente a análise da complexidade deste algoritmo (em notação $\mathcal{O}$) para o melhor caso e para o pior, relativamente a comparações e a trocas envolvidas.

6. Considere o algoritmo de ordenação por fusão (*merge sort*).

(a) Utilize o algoritmo para ordenar a lista `[54, 26, 93, 17, 77, 31, 44, 55, 20]` por ordem crescente, apresentando os valores intermédios a cada novo estado da(s) lista(s) e indicando o número de comparações realizadas cumulativamente.

(b) Repita para a lista `[93, 77, 55, 54, 44, 31, 26, 20, 17]`.

(c) Apresente a análise da complexidade deste algoritmo (em notação $\mathcal{O}$) para o melhor caso e para o pior, relativamente a comparações envolvidas.

7. Considere a seguinte lista cujos elementos são tuplos que representam estudantes:

```
alunos = [('pedro', 'A', 15), ('ana', 'B', 15),
          ('david', 'B', 16), ('mariana', 'C', 12),
          ('pedro', 'C', 12), ('david', 'A', 10)]
```

com o primeiro, o segundo e o terceiro argumento do tuplo a representar, respetivamente, o nome, a turma e a idade.

(a) Qual seria o resultado de executar a instrução `alunos.sort(key = lambda x : x[0])` seguida de `print(alunos)`?

(a)
```
[('pedro', 'A', 15), ('ana', 'B', 15),
 ('david', 'B', 16), ('mariana', 'C', 12),
 ('pedro', 'C', 12), ('david', 'A', 10)]
```

(b)
```
[('ana', 'B', 15), ('david', 'B', 16),
 ('david', 'A', 10), ('mariana', 'C', 12),
 ('pedro', 'A', 15), ('pedro', 'C', 12)]
```

(c)
```
[('ana', 'B', 15), ('david', 'A', 10),
 ('david', 'B', 16), ('mariana', 'C', 12),
 ('pedro', 'A', 15), ('pedro', 'C', 12)]
```

(d) pode ser (b) ou (c) dependendo da implementação.

(b) Use a função `list.sort` para ordenar a lista `alunos` original por ordem crescente da idade dos estudantes.

(c) Use a função `sorted` para obter a partir da lista `alunos` original uma outra lista em que os elementos se encontram pela ordem crescente da idade dos estudantes e, em caso de empate, pela ordem lexicográfica crescente dos nomes.

8. Considere a seguinte string `"ahbdfre"`.

   (a) Use a função `sorted` para obter a partir desta string, uma lista com os seus caracteres por ordem lexicográfica crescente.

   (b) Escreva uma função `ordena_string` que dada uma string, devolve uma string que é uma versão por ordem lexicográfica crescente da string dada.

9. Escreva uma função `ordena_conjunto` que recebe um conjunto e devolve a lista com os elementos do conjunto por ordem crescente (pode assumir que os elementos do conjunto são strings). Utilize a função `list.sort`.

10. Escreva uma função `ordena_dicionario` que recebe um dicionário e devolve uma lista com os pares (chave,valor) do dicionário, por ordem crescente das chaves (pode assumir que as chaves do dicionário são strings). Utilize a função `sorted`.

11. *QuickSort*, desenvolvido por Sir Tony Hoare, 1960, apoia-se numa estratégia de *dividir para conquistar* em que uma sequência inicial S é dividida em várias sub-sequências às quais o algoritmo é aplicado recursivamente. O resultado é posteriormente concatenado obtendo-se uma sequência ordenada. Tipicamente o *QuickSort* pode ser dividido em três passos:

Dividir Se S tem zero ou um elementos, a sequência está ordenada. Caso contrário, selecionar o último elemento de S, que passa a ser o *pivot*.[2] Retirar todos os elementos de S construindo as seguintes duas sequências:

- B (baixo) contendo todos os elementos *inferiores ou iguais* ao pivot;
- A (alto) contendo todos os elementos *superiores* ao pivot;

Conquistar Aplicar recursivamente o algoritmo às sequências B e A.

Combinar Concatenar B, pivot, A, por esta ordem para obter o resultado.

(a) Implemente o algoritmo na linguagem Python.

[2] Diferentes implementações podem selecionar o pivot de forma diferente.

(b) Qual a complexidade do algoritmo quando a lista já se encontra ordenada?

(c) Discuta a complexidade computational do algoritmo no melhor e no pior caso.

12. O algoritmo de ordenação por baldes (*bucket sort*) apoia-se no pressuposto que os elementos a ordenar estão associados a chaves, as quais podem ser usadas como índices de uma lista de baldes (*buckets*). Eis o algoritmo:

    - seja S uma lista de elementos com chaves no intervalo $[0, N-1]$;
    - seja B uma lista de N baldes inicialmente vazios.
    - para cada elemento e de S:
        - seja k a chave de e;
        - retirar e de S;
        - adicionar e ao balde $B[k]$;
    - para cada k entre 0 e $N-1$:
        - para cada elemento e de $B[k]$:
            * remover e de $B[k]$;
            * adicionar e ao final de S;

    (a) Implemente o algoritmo na linguagem Python. Considere que `S` é uma lista de pares (chave, elemento). Por exemplo, para
    ```
    S = [(5, 'a'), (2, 'b'), (5, 'c'), (1, 'd')]
    ```
    o algoritmo deverá modificar `S` para
    ```
    S = [(1, 'd'), (2, 'b'), (5, 'c'), (5, 'a')]
    ```

    (b) Discuta a complexidade computational do algoritmo. Se chegar a uma solução sub $\mathcal{O}(n \log n)$ explique porque é que esta não contradiz o resultado que estabelece $\mathcal{O}(n \log n)$ como limite inferior para qualquer algoritmo de ordenação baseado em comparação.