# Pesquisa e Classificação de Dados
# Lista 2 (Busca e Ordenação)

Prof. Ricardo Oliveira

Esta lista **não** vale nota e **não** deve ser entregue, mas apenas utilizada como material de apoio para estudo. Naturalmente, você pode tirar eventuais dúvidas com o professor.

Exercícios marcados com (B) são básicos e essenciais para a matéria. Exercícios marcados com (C) são complementares. Recomenda-se fortemente a resolver todos os exercícios.

1. (B) Considere uma aplicação que manipula um vetor de tamanho $N$ fazendo e $B$ buscas ao todo.
   (a) Determine o custo de pior caso utilizando sempre a busca linear;
   (b) Determine o custo de pior caso inicialmente ordenando o vetor em $O(N \lg N)$ e então sempre utilizando a busca binária;
   (c) Qual estratégia é a melhor para $N = 1000$ e $B = 10$?
   (d) Qual estratégia é a melhor para $N = 1000$ e $B = 1000$?
   (e) Qual estratégia é a melhor para $N = 1000$ e $B = 1000000$?
2. (B) Implemente um algoritmo $O(N)$ para inserir um novo valor $x$ em um vetor já ordenado $v$ de forma que $v$ continue ordenado.
3. (C) Explique por que *não* é vantajoso alterar seu algoritmo do item anterior para, antes de seu início, chamar a busca binária para determinar a posição do elemento a ser inserido.
4. (B) Considere a seguinte modificação da busca binária para buscar um elemento $X$ em um vetor ordenado $v$:

```
inicio = 0, final = n-1
enquanto inicio <= final:
    terco1 = inicio + (final-inicio)/3
    terco2 = inicio + 2*(final-inicio)/3
    Se v[terco1]==X ou v[terco2]==X retorne "sim"
    Se X < v[terco1]
        final = terco1-1
    Senao, Se v[terco1] < X < v[terco2]
        inicio = terco1+1, final = terco2-1
    Senao
        inicio = terco2+1
retorne "Nao"
```

   Analise o melhor caso, o caso médio e o pior caso para este algoritmo.
5. (C) Implemente os algoritmos de ordenação estudados em aula.
6. (B) Analise a complexidade de tempo de caso médio do *InsertionSort*, assumindo distribuição uniforme.
7. (C) Implemente versões recursivas para o *BubbleSort*, o *InsertionSort*, o *SelectionSort* e o *HeapSort*.

8. O *número de inversões* de um vetor é o número de pares ordenados de elementos que estão com sua ordem relativa incorreta, isto é, o número de pares $i < j$ tais que $v[i] > v[j]$. Por exemplo, o vetor $[4, 3, 2, 5]$ tem três inversões (correspondentes aos pares 4 e 3, 3 e 2, 4 e 2). Implemente um algoritmo que recebe um vetor e determina seu número de inversões modificando:
   (a) (B) o *BubbleSort*
   (b) (B) o *InsertionSort*
   (c) (C) o *SelectionSort*
   (d) (C) o *MergeSort*
9. (B) Considere a seguinte modificação para o *MergeSort*:

```
MergeSort(v)
    Divida v em quatro partes de igual tamanho [A|B|C|D]
    MergeSort(A)
    MergeSort(B)
    MergeSort(C)
    MergeSort(D)
    retorne Merge( Merge(A,B) , Merge(C,D) )
```

   Analise o pior, melhor e médio caso deste algoritmo.
10. (C) Implemente uma versão iterativa (não recursiva, "*bottom-up*") do *MergeSort*.
11. (B) Considere que sua linguagem favorita tem uma função pronta `quicksort(vetor v)` que implementa o *QuickSort*. Segundo a documentação, esta função sempre utiliza, a cada chamada recursiva, o primeiro elemento do vetor como pivô. Construa um algoritmo que, dado $N$, gera um vetor de tamanho $N$ que *obriga* esta função a fazer $\Omega(N^2)$ comparações para ordená-lo.
12. (B) Considere agora que você é o(a) programador(a) da função `quicksort(vetor v)`. Proponha uma estratégia de pré-processamento para o vetor de forma que seja muito improvável que o algoritmo execute em $\Omega(N^2)$, mesmo quando sua entrada for o vetor construído no exercício anterior.
13. (B) Seja $v$ o vetor $v = [3, 4, 1, 2, 5]$.
   (a) Apresente o vetor modificado de forma a representar uma *max-heap*;
   (b) Apresente o vetor após cada passo do laço principal do *HeapSort*.
14. (B) O *CountingSort* assume que todos os elementos do vetor são menores que um limite $M$. Em computadores modernos, o tipo `unsigned int` tem 32 bits. Explique por que é inviável fazer ordenação linear com o *CountingSort* assumindo $M = 2^{32}$.
15. (B) Analise a complexidade de pior caso do *BucketSort* assumindo 2 (dois) *buckets* e um método ótimo de ordenação comparativa para cada um.
16. (B) Refaça o exercício anterior, generalizando para $B$ *buckets*.
17. (B) Ordene o vetor $[578, 321, 458, 427]$ usando o *RadixSort*, apresentando o vetor resultante após cada passo do laço principal do algoritmo.
18. (B) Explique por que o *RadixSort* deve utilizar como subrotina um método estável, e apresente um exemplo no qual o uso de um método instável não ordena o vetor corretamente.

19. (B) Classifique os métodos *Bubble*, *Insertion*, *Selection*, *Quick*, *Merge*, *Heap*, *Counting*, *Bucket* e *RadixSort* quanto à estabilidade e a *in-placement*.
20. (C) Resolva o problema 1566 do URI Online Judge.