# Tarefa 01 de Geometria Computacional 2021.1

Aluna: Carolina Herbster Mesquita Jorge

## Questão 01

Assinale V ou F nas afirmações abaixo, justificando suas respostas:

1. Um algoritmo é O(f(n)) quando existe k, n0 tal que existe alguma instância de tamanho n ≥ n0, onde o número de passos é ≥ kf(n).
    a. A resposta é verdadeiro, por aplicação da definição.
2. Um algoritmo é Ω(f(n)) quando existe k, n0 tal que para qualquer instância de tamanho n ≥ n0, o número de passos é ≤ kf(n).
    a. A resposta é verdadeiro, por aplicação da definição.
3. Se um algoritmo é O(n), ele é também $O(n^2)$.
    a. A resposta é verdadeiro, pois f(n) = n é $O(n^2)$, porque para k = 1 e $n_o$ = 1, $n^2 \geq 1 \times n$.
4. Todo algoritmo é pelo menos Ω(n).
    a. A resposta é falso. Existem algoritmos, como o de busca binária, que são O(log n) e Ω(log n).
5. A etapa mais importante no algoritmo de ordenação quicksort é a combinação, enquanto que no algoritmo de ordenação mergesort é a separação.
    a. A resposta é falso. A etapa mais importante do algoritmo quicksort é a separação, através do algoritmo de particionar, no qual o vetor é dividido em dois subvetores de elementos menores e maiores que um pivô. Não é feito nenhum processamento posterior nos vetores ao combiná-los. Já no mergesort a etapa mais importante é a combinação, através do algoritmo combinar, no qual dois subvetores ordenados são combinados a fim de formar um vetor ordenado. A etapa de separação do mergesort apenas divide o vetor em dois, sem nenhum processamento.

## Questão 02

Algoritmo:

```
"""
Selection sort algorithm. Receives a vector and orders it in-place
"""
def selection_sort(v):
    n = len(v)
    # At each loop iteration, the invariant is that the
    # sublist from 0..i-1 is sorted and i..n is unsorted.
```

```
    for i in range(0, n):
       # Find the smallest item in the unsorted sublist
        smallest = i
        for j in range(i+1, n):
            if v[j] < v[smallest]:
                smallest = j
        # Swap the smallest element with the first one in the unsorted sublist
        if smallest != i:
            aux = v[i]
            v[i] = v[smallest]
            v[smallest] = aux
        # Grow the sorted sublist
```

Para testar, o algoritmo foi rodado 1000 vezes com inputs aleatoriezados e a média dos tempos de execução obtida:

Average time for a random input of size 10: 2.0940303802490234e-05
Average time for a ordered input of size 10: 9.932756423950196e-06
Average time for a random input of size 100: 0.0007118461132049561
Average time for a random input of size 1000: 0.0680426483154297

Pelos tempos obtidos, podemos observar que o algoritmo é extremamente rápido para inputs pequenos, porém seu tempo de execução cresce bastante a medida que o input cresce

# Questão 02

Algoritmo:

```
"""
Quick sort algorithm
"""

def quick_sort(v, lo, hi):
    n = hi - lo + 1
    if n > 1:
        q = partition(v, lo, hi)
        quick_sort(v, lo, q-1)
        quick_sort(v, q+1, hi)


"""
Chooses the pivot element as the last element and partitions the input
vector into smaller and bigger elements
"""
def partition(v, lo, hi):
    pivot = v[hi]
```

```
    i = lo
    # At each loop iteration, the invariant is that the elements from
    # 0..i-1 are smaller than the pivot
    for j in range(lo,hi+1):
        # If we find another element that's smaller than the pivot, place
        # into the i-th position and grow i to keep the invariant
        if v[j] < pivot:
            aux = v[i]
            v[i] = v[j]
            v[j] = aux

            i += 1

    # Place the pivot into the correct position
    aux = v[i]
    v[i] = v[hi]
    v[hi] = aux
    # Return the pivot index
    return i
```

Para testar, o algoritmo foi rodado 1000 vezes com inputs aleatorizados e a média dos tempos de execução obtida:

Average time for a random input of size 10: 2.7051210403442382e-05
Average time for a ordered input of size 10: 6.777572631835937e-05
Average time for a random input of size 100: 0.00041715764999389647
Average time for a random input of size 1000: 0.005378573894500732

Podemos observar que o tempo de execução do Quick Sort é um pouco maior que o Selection Sort no início, porém ele cresce menos lentamente que o Selection Sort.

# Questão 03

Algoritmo:

```
"""
Merge sort algorithm. Receives a vector and returns an ordered
version of it
"""
def merge_sort(v):
    n = len(v)
    if n > 1:
        mid = floor(n/2)
        l1 = merge_sort(v[0:mid])
        l2 = merge_sort(v[mid:n])
        return merge(l1, l2)
    return v
```

```python
def merge(l1,l2):
    # The merged list
    m = []
    i1 = 0
    n1 = len(l1)
    i2 = 0
    n2 = len(l2)
    # While there are still elements to copy from on both lists,
    # copy the smallest element
    while i1 < n1 and i2 < n2:
        if l1[i1] <= l2[i2]:
            m.append(l1[i1])
            i1 += 1
        else:
            m.append(l2[i2])
            i2 += 1
    # If only the elements of l1 remain, copy them all to the merged
    # list. Else, copy the elements of l2.
    if i1 < n1:
        m.extend(l1[i1:n1])
    elif i2 < n2:
        m.extend(l2[i2:n2])
    return m
```

Para testar, o algoritmo foi rodado 1000 vezes com inputs aleatorizados e a média dos tempos de execução obtida:

Average time for a random input of size 10: 6.192159652709961e-05
Average time for a ordered input of size 10: 5.099797248840332e-05
Average time for a random input of size 100: 0.0006983215808868408
Average time for a random input of size 1000: 0.00808120584487915

Assim como o Quick Sort, a média dos tempos cresce mais lentamente com o crescimento do input, porém a performance em inputs menores é mais lenta que no Quick Sort.

# Questão 05

Uma tabela comparando os tempos dos algoritmos encontra-se a seguida:

| Avg of 1000 runs | Selection Sort | Insertion Sort | Merge Sort | Quick Sort |
|---|---|---|---|---|
| 10 random entries | 0.000022 | 0.000024 | 0.000053 | 0.000034 |
| 10 ordered entries | 0.000012 | 0.000016 | 0.000057 | 0.000069 |
| 100 random entries | 0.000684 | 0.001525 | 0.000795 | 0.000548 |
| 1000 random entries | 0.077338 | 0.152426 | 0.008895 | 0.005912 |

Através da tabela, podemos perceber que o Selection Sort e Insertion Sort obtém um menor tempo em inputs pequenos, especialmente quando eles estão ordenados. Já o Quick Sort é mais lento em inputs pequenos ordenados, e o Merge sort mais lento em inputs pequenos aleatórios. Porém, a medida que o tamanho do input cresce, podemos perceber que o Merge Sort e Quick Sort tornam-se melhores, especialmente o Quick Sort.

# Questão 06

Para resolver o problema da linha poligonal através da redução, podemos transformá-lo em um problema de ordenação. Após ordenar os vértices de entrada em seus respectivos eixos em uma ordem definida (ex: x, y, z), a linha poligonal será formada pela sequência ordenada de vértices. Como existem algoritmos de ordenação cuja ordem é $\Theta(n \log n)$, e a redução envolve apenas dispor os vértices de entrada em um vetor, esta é de ordem $\Theta(n)$, ou seja, linear.

A seguida, encontram-se os tempos dos algoritmos de linha poligonal através da redução, medidos como a média de 100 execuções aleatorizadas:

| Avg of 100 runs | Selection Sort | Insertion Sort | Merge Sort | Quick Sort |
|---|---|---|---|---|
| 10 random entries | 0.000098 | 0.000027 | 0.000103 | 0.000035 |
| 10 ordered entries | 0.000010 | 0.000000 | 0.000067 | 0.000026 |
| 100 random entries | 0.000873 | 0.001660 | 0.000782 | 0.000549 |
| 1000 random entries | 0.128753 | 0.162386 | 0.009568 | 0.006277 |

Na tabela, podemos ver relações semelhantes à análise anterior, com algumas pequenas diferenças como o Selection Sort não ser tão rápido para inputs pequenos. Uma possível causa disso seria a que como Selection Sort precisa realizar mais comparações, e esta operação tornou-se mais cara (pois devemos comparar 3 números representando as coordenadas de cada ponto em vez de apenas 1), isso refletiu-se no tempo do algoritmo.