Geometria Computacional - Tarefa 01

Carolina Herbster

Desenvolvimento

- Linguagem: Python
  - Vantagens: Expressividade, concisa, biblioteca padrão bastante completa
  - Desvantagens: Performance
- Bibliotecas Extras: Matplotlib
  - Opcional
  - Apenas para visualização da linha poligonal
  - pip install matplotlib
- Tentei gerar um executável com as bibliotecas pyinstaller e py2exe, mas em ambas o matplotlib não conseguiu ser linkado de modo a mostrar imagens

Desenvolvimento

- Funções genéricas para medir o tempo do código

```python
def run_sort_algorithm(l, name):
    print(f'===== Running algorithm {name} ======')
    # Check the algorithm's correctness
    check_correctness(l)

    example_10 = time_function_rand_input(l, 10, 1000, lambda s: random.randint(0, s*10))
    example_ordered = time_function_ordered_input(l, 10, 1000, lambda s: s*2)
    example_100 = time_function_rand_input(l, 100, 1000, lambda s: random.randint(0, s*10))
    example_1000 = time_function_rand_input(l, 1000, 1000, lambda s: random.randint(0,
s*10))

    return {
        "example_10": example_10,
        "example_ordered": example_ordered,
        "example_100": example_100,
        "example_1000": example_1000,
    }
```

Desenvolvimento

```python
def time_function_rand_input(fn, size, repeats, genfn):
    sum = 0
    for i in range(0, repeats):
        random_list = [genfn(size*10) for y in range(size)]
        start_time = time.time()
        fn(random_list)
        elapsed = time.time() - start_time
        sum += elapsed
    return sum/repeats
```

```python
def time_function_ordered_input(fn, size, repeats, genfn):
    sum = 0
    for i in range(0, repeats):
        start_num = random.randint(0, size*10)
        ordered_list = [genfn(y) for y in range(start_num, start_num+size)]
        start_time = time.time()
        fn(ordered_list)
        elapsed = time.time() - start_time
        sum += elapsed
    return sum/repeats
```

Desenvolvimento

```python
def main():
    # Run standalone sorting algorithms
    selection_times = run_sort_algorithm(lambda l: selection_sort(l), "Selection Sort")
    insertion_times = run_sort_algorithm(lambda l: insertion_sort(l), "Insertion Sort")
    merge_times = run_sort_algorithm(lambda l: merge_sort(l), "Merge Sort")
    quick_times = run_sort_algorithm(lambda l: quick_sort(l, 0, len(l)-1), "Quick Sort")

    # Make a comparative table
    print_table_with_times(selection_times, insertion_times, merge_times, quick_times)

    print('\n')

    # Run polygonal line algorithms
    polygonal_selection_times = run_polygonal_line_reduction(lambda l: selection_sort(l), "Selection Sort")
    polygonal_insertion_times = run_polygonal_line_reduction(lambda l: insertion_sort(l), "Insertion Sort")
    polygonal_merge_times = run_polygonal_line_reduction(lambda l: merge_sort(l), "Merge Sort")
    polygonal_quick_times = run_polygonal_line_reduction(lambda l: quick_sort(l, 0, len(l)-1), "Quick Sort")
    print_table_with_times(polygonal_selection_times, polygonal_insertion_times, polygonal_merge_times, polygonal_quick_times)

    # Show ordenations on graphs
    show_polygonal_line_reduction(lambda l: selection_sort(l), 10, "Selection sort")
    show_polygonal_line_reduction(lambda l: insertion_sort(l), 10, "Insertion Sort")
    show_polygonal_line_reduction(lambda l: merge_sort(l), 10, "Merge Sort")
    show_polygonal_line_reduction(lambda l: quick_sort(l, 0, len(l)-1), 10, "Quick Sort")
```

Selection sort

```python
"""
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
```

Quick Sort

```python
"""
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)
```

```python
"""
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
```

Merge Sort

```python
"""
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
```

Comparação

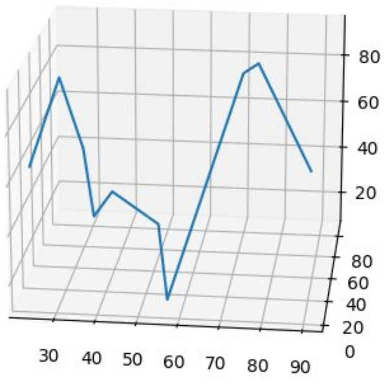
Avg of 1000 runs	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
10 random entries	0.000022	0.000024	0.000053	0.000034
10 ordered entries	0.000012	0.000016	0.000057	0.000069
100 random entries	0.000684	0.001525	0.000795	0.000548
1000 random entries	0.077338	0.152426	0.008895	0.005912

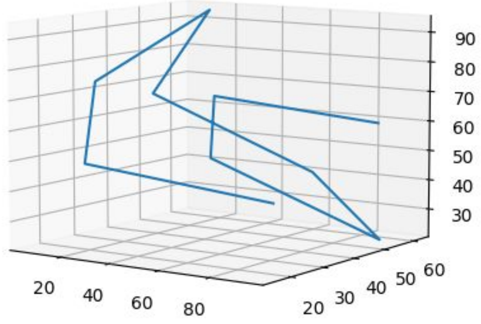
Linha Poligonal

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 Θ(nlogn), e a redução envolve apenas dispor os vértices de entrada em um vetor, esta é de ordem Θ(n), ou seja, linear.

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

Questão 06 (Exemplos de linhas poligonais com 10 vértices)





Questão 06 (Exemplos de linhas poligonais com 10 vértices)