# Project:
# Algorithmic complexity and graphs

Group: Johannes Timmreck, Alexander Prosch, Jonas Conrad

# Table of Contents

# 1. The snowplow problem

## 1.1) Configuration Proposal

Assuming an input of size n, with n=9, and the street length being 10 in both directions around the snowplow's starting point (0). We propose the input list [-10, -9, -8, -7, -5, 1, 2, 7, 10], visualized in figure 1.

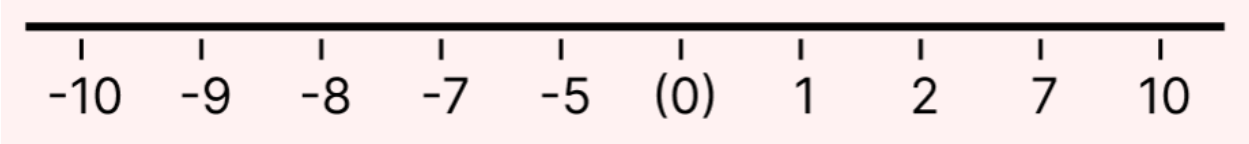


Figure 1. Proposed Input List

We created three tables to visualize why neither the shortest path nor simply going left to right in a sorted list is the optimal way of achieving a minimal average waiting time for each house. The first table, seen in table 1, displays the waiting times using the shortest path approach, while the second table, seen in table 2, shows the waiting times in a sorted list approach. The last table, seen in table 3, displays the waiting times using a more optimized method.

| Shortest Path | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Path | 1 | 2 | 7 | 10 | -5 | -7 | -8 | -9 | -10 |
| Wait time | 1 | 2 | 7 | 10 | 25 | 27 | 28 | 29 | 30 |
| Avg Wait Time | (1 + 2 + 7 + 10 + 25 + 27 + 28 + 29 + 30) = 159<br>159 / 9 = 17,66666 | | | | | | | | |

Table 1. Shortest Path Solution for Proposed List

| Sorted List | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Path | -5 | -7 | -8 | -9 | –10 | 1 | 2 | 7 | 10 |
| Wait time | 5 | 7 | 8 | 9 | 10 | 21 | 22 | 27 | 30 |
| Avg Wait Time | (5 + 7+ 8 + 9 + 10 + 21 + 22 + 27 + 30) = 139<br>139 / 9 = 15,44444 | | | | | | | | |

Table 2. Sorted List Solution for Proposed List

| Optimized Path | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Path | 1 | 2 | -5 | -7 | –8 | -9 | –10 | 7 | 10 |
| Wait time | 1 | 2 | 9 | 11 | 12 | 13 | 14 | 31 | 34 |
| Avg Wait Time | (1 + 2 + 9 + 11+ 12 + 13 + 14 + 31 + 34) = 127<br>127 / 9 = 14,111111 | | | | | | | | |

Table 3. Optimized Solution for Proposed List

As seen in tables 1, 2, and 3, the average waiting time decreased with each new approach. By analyzing the improvements from the shortest path to the sorted list, we derived that it is beneficial to go for large and densely packed areas first. This gave us the idea to apply a clustering method to the list allowing us to identify two clusters within the list [1,2] and [-7, -8, -9, -10]. We now prioritized the visit of these clusters over other parameters, such as the closest neighbor, and thus were able to reduce the average waiting time.

## 1.2) Algorithm Proposal

Taking the insights obtained by step 1.1) Configuration Proposal, we constructed an algorithm. Said algorithm would first create a prioritized list of items by identifying dense areas within its length. It would then iterate over the list to create the route of the snowplow from it.

The algorithm works in two separate steps. First, it initializes, constructs, and formats the various lists needed to properly evaluate the path most optimal in minimizing waiting time. It does so by first sorting the list of houses received as input. Then it separates them into left and right sides, seeing as the only choice the algorithm must make is going left or right at each iteration. Then after separating the sides, the algorithm creates two lists of clusters. Each cluster contains several houses, with each entry no more than 1% of the street length away from another entry in the list. After the lists are generated, they are sorted based on cluster size.

In the second phase, the algorithm iterates over the number of clusters and then decides if it wants to go to the left or right. It decides so by checking which cluster is the largest. The algorithm continues to choose between the clusters until both lists are empty.
If both sides should present a cluster of the same size, the algorithm chooses the one closest. Should both be the same distance, it takes the side with the shortest overall length. If both sides have the same length, it goes in the previously chosen direction. If there was no previous direction it will go right.

When given an array of 1000 entries with a value between -1000 and 1000, our algorithm performed around 21% better than the shortest path algorithm. When running the algorithm 1000 times, only 47 results were below 10%, creating an error margin of around 4,7%. These errors appear in cases where the greedy solution of the shortest path performs itself very efficiently.

## 1.3) Complexity Evaluation

To properly evaluate the runtime, we will compute the worst case-complexity for each function within the algorithm and then combine these results based on how they are called within the algorithm to receive the overall worst-case complexity. However, seeing that the algorithm makes various checks and comparisons, we cannot assume a “universal” worst-case scenario, as sometimes the worst scenario for one case is a good scenario for other functions. Thus the result we will arrive at will most likely be slightly worse than the actual worst-case scenario. As we are not interested in the exact runtime complexity we will disregard things that would add a runtime of 1 as we use various functions and loops within the algorithm making the complexity already O(n**2) at the least. The next section Function Complexity will break down the exact complexity of each function. A summary of our approximations regarding the function complexity can be seen in table 4.

| Function name | Function complexity | Big O’Notation |
|---|---|---|
| __init__ | 3(n log n) + 7n | O(n) |
| __clusterSides | n | O(n) |
| __turnRight \|\| __turnLeft \|\| __calculateDirection | n + n*(n**2 + 2n) | O(n**3) |
| createRoute | n * (n + n*(2n + n*(n))) | O(n**4) |
| parcours | (3(n log n) + 7n) + n * (n + n*(2n + n*(n))) | O(n**4) |

Table 4. Function complexities of task1.py

### Function Complexity

There are seven functions within task1.py. __init__(), __clusterSides(), __turnRight(), __turnLeft(), __calculateDirection(), createRoute(), parcours().
These seven functions can be redacted to five functions since __turnLeft() and __turnRight() are the same function but operate on different objects. While __calulateDirection() is just a container for __turnLeft() and __turnRight(). Thus they all possess the same overall complexity.

__clusterSides() iterates over a given list of objects and calls append() if a condition is met. The complexity is O(n) since the loop only iterates over functions of complexity O(1).

__initi__() contains no loops but calls sort methods two times in the beginning, for the complexity of O(2(n log n)). It also makes various calls to remove(), index(), and split list methods ([:]), which adds 6n to the complexity. Lastly the function features two conditions to call __clusterSides(). The given sublists have a combined size of n, so we can add 1n to the list, it then sorts both sublists which adds another n log n to the complexity. For a total complexity of O(3(n log n) + 7n) or O(n).

__turnLeft() and __turnRight() are the same function as stated before. They first call the index method with a complexity of O(n). Then they loop for the size of the return, which in worst-case is equal to n. For each loop, they call another loop and the del operator twice for n*(child loop+2n).
The child loop iterates over the size of a cluster, which makes the worst-case be size n. However, if we expect size n, then the parent loop would only be of complexity O(1), and if the parent loop is of complexity O(n) then the child loop would be complexity O(1). We will disregard this fact and assume both to be of complexity O(n) regardless of input. The child loop calls the remove method for each iteration, so its complexity is O(n * n) or O(n**2).
By adding these together we arrive at the following complexity, O(n + n*(n**2 + 2n)) or O(n + n**3 + 2n). For a complexity of O(n**3).

The __calculateDirection() method contains various if-statements and makes a single call to __turnLeft() or __turnRight() making its complexity the same as __turnLeft() or __turnRight().

The createRoute() method iterates over all clusters, so in worst-case n iterations. For each iteration it calls either __turnLeft(), __turnRight(), or __calculateDirection(). Since all three have the same complexity we can derive the complexity of createRoute() as O(n*n**3) or O(n**4).

The parcours() method calls the __init__() function once, followed by the createRoute function for a total complexity of O(n + n**4). The function's complexity is O(n**4). A polynomial function runs with a complexity of O(n**k), meaning the function is polynomial.