Algorithms and Data Structures - Assignment 2

Introduction

The dimensions of the 2D grid (width and height) are given and the locations (encoded) and the package codes (encoded). Moreover, also the Caesar shift constant c is given with which the messages have been encoded.

The task is to search for a package in a 2D grid with a given width and height, and locations and package codes, which are both encoded in a Caesar cipher. Furthermore, the Caesar shift constant, which is necessary to encode the input is also given. Hence, the first subtask is to encode the locations and package codes with the given Caesar shift constant to obtain the decoded versions of the locations and package codes.

The algorithm for the decoding works as follows:

1. The algorithm loops over the single decoded locations/ packages
2. The algorithm loops over every single character (in binary form) in each location/ package
3. Each decoded character of the location/ package is transformed into an ordinal number
4. Then the Caesar shift constant is subtracted from the ordinal number
5. The shifted ordinal number is transformed into the corresponding character and added to a temporary list
6. The list of characters is joined to the actual word
7. The algorithm outputs the decoded location/packages

Also, after that, the codes are filled into the grid of locations. Then we obtain a grid like this:

1	10	14	22	27
11	15	24	28	38
12	23	32	36	42

Table 1: Package Code we use for our example, search value= 42

The divide and conquer algorithm recursively finds squares within the rectangle to search for the value in diagonals. The algorithm starts to check on the diagonals for the value. In the example, the first value on the diagonal is 1, the second value is 15 and the third value is 32. The next values are 22, 38, 36, 42 respectively.

1. Each entry on the diagonal is compared to the search value from top-left to bottom-right.
2. If the entry on the diagonal is smaller than the search value then we move on to the next entry on the diagonal (or the next subgrid). *(see section: optimality)
3. If the value is bigger or equal to the search value, then we check all entries in the grid above the diagonal entry, left to the diagonal entry, and the entry itself.

   The algorithm either finds an entry equal to the search value and we found the solution or the search value is not in the subgrid.

The algorithm either finds an entry equal to the search value and we found the solution or the search value is not in the subgrid. We repeat the above algorithm for all subgrids.

In the example, the search value is not within the first subgrid with the diagonal 1-15-32. Hence, the algorithm moves to the rest of the grid starting the diagonal search in the top left corner. The new diagonal of the next subgrid is 22-38. The 3-step algorithm is applied anew, but for this example, the value is not found in this subgrid. At last, the two leftover squares are checked starting again with top left entry 36 which is not the

search value. Finally, the last value is checked which is 42 equal to the search value (also 42). Only then the algorithm returns the location tuple of the found value. If there is no 42 in the grid, the algorithm would return "None".

The traversing of the grid is actualized by differentiation of three cases:

1. The variable x-from exceeds x-to, but y-from does not exceed y-to. x-from is reset to 0 and 1 is added to y-from. The algorithm moves on to the next subgrid.
2. The variable y-from exceeds y-to, but x-from does not exceed x-to. y-from is reset to 0 and 1 is added to x-from. The algorithm moves on to the next subgrid.
3. The variables x-from and y-from both exceed x-to and y-to respectively. The algorithm stops because the whole grid has been searched.

This approach is more efficient than a scan over all cells because the constraints are used in a way that computation is saved. Every time the entry on the diagonal is smaller than the search value, we save the computation of the entries above and left of the diagonal value. This is reasonable because the entries left and above need to be smaller than the diagonal entry which is given by the constraints. The saved computation are explicitly given in 2.

Search Tree

In Figure 1, all the possible search paths for the grid shown in Table 1 are represented. The rightmost elements are the elements on the diagonal (as explained in the previous section) of the searched rectangles. The algorithm follows the rightmost children as long as the search value is bigger than the entry on the diagonal. If the search value is equal to the rightmost entries the algorithm stops and returns the position of the element in the grid. If the search value is smaller than the entry on the diagonal, the algorithm further traverses the search tree in the left part of the search tree. Technically one needs another node of the diagonal element because the comparison between the search value and the diagonal entry being equal is performed in an additional step. This step is not shown in the graph for improved clarity.

Since, for every node, the algorithm checks: "Is the search value bigger than the diagonal entry?". Then the left child embodies the boolean answer "False" and the right child embodies the boolean answer "True". However, those left nodes representing "False" are also omitted for the values 1,22,36 for improved clarity.

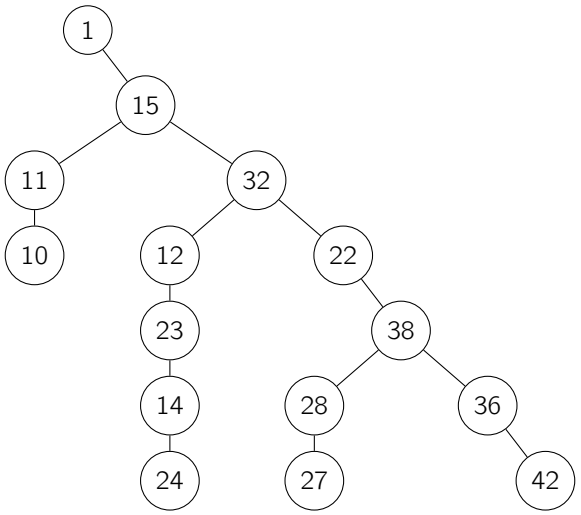

Figure 1: A search tree for the example grid shown in Table 1.

Greedy Approach

The algorithm encapsulates all the criteria for a greedy approach because it applies the most effective approach in every step.

Feasibility

Algorithm description:
The diagonal of the biggest possible square is searched, then the diagonal starting from the top left of the leftover rectangle. The process is repeated for the leftover rectangle until the whole grid is searched. The algorithm stops when the value is found and returns the location tuple of the found value or the whole grid is searched and the value is not found, so "None" is returned.
This process is dependent on the constraints, so the algorithm is only feasible if the constraints are not violated. Followingly, it is crucial to implement the "check_constraints"_test which checks whether the constraint in the grid hold. If a constraint is violated the algorithm returns "None". This secures that the algorithm always returns a valid solution.

Optimality

The constraint of the grid is that the package codes always increase in one column from top to bottom and increase in one row from left to right. Hence, it makes sense to check the diagonal entries to exclude checking the entries of the left part of the row and the top part of the column of each diagonal entry. So, for every diagonal entry that is smaller than the search value the computation for the entries above and left of the diagonal is saved. There is no entry in the grid that is not checked indirectly (by parsing the diagonal) or directly (parsing the values above and left of the diagonal entry). Furthermore, the algorithm is locally optimal because it finds every possible solution. This is secured by the test "test_search_solution_ex_grid" where the test loops over all enumerated entries in the grid (matrix). Hence, every possible value in the grid can be found by the algorithm.

Irrevocability

The algorithm is irrevocable because at every point that a decision is made by the algorithm it can not be. This can be illustrated by 1 where an edge in the tree represents an action (moving on to the next entry) and each node represents a state (current checked entry). From the tree, it becomes clear that every action is irreversible, and every level of the tree embodies one alternative option that is unique. So, whenever an entry has been checked the decision on the entry is never changed in subsequent steps of the algorithm.

Runtime Comparison: Linear Scan vs Divide and Conquer

To compare the runtime of our Divide and Conquer and a linear approach we implemented a linear algorithm. We did this by simply iterating over the array and returning the row and column if the entry is equal to the search value in the iteration step. If the iteration is run through without finding the value, the algorithm returns "None". This approach to searching a value represents a worst-case scenario for running time in terms of the number of cells that are searched by both methods. Table 2 shows the performed step for both algorithms for an assumed y-size of 3 and variable x-size. The number of steps that are at most performed in the Divide and Conquer algorithm increase by 1 with increased x-size, while the number of steps permed in the linear algorithm increase by the y-size. The same behavior is observable for all x and y sizes.

Grid x-size	max steps Linear	max steps Divide and Conquer
2	6	4
3	9	5
4	12	6
5	15	7
6	18	8

Table 2: Number of performed steps for both algorithms for an assumed y-size of 3 and variable x-size. The number of steps that are at most performed in the Divide and Conquer algorithm increase by 1 with increased x-size, while the number of steps performed in the linear algorithm increase by the y-size. The same behavior is observable for all x and y sizes.

Summary and Discussion

Goals

In the assignment, we aimed to find a Divide and Conquer algorithm to find a specific value in a grid of variable size. The algorithm needs to be more efficient than a linear scan. Moreover, we should learn to handle data that is encrypted with a Cesar cipher, since both the locations in the grid and the package codes filled in the grid should be encoded. We should analyze this code in terms of greedy approach criteria and should compare its (theoretical) runtime to the one of a linear approach.

Results

We developed a Divide and Conquer algorithm that traverses the diagonal, and the column above and row left of the diagonal entry that is greater than the search value.
We found that:

- Our algorithm is feasible, optimal, and irrevocable and thus fulfills greedy approach criteria
- The theoretical runtime of the algorithm increases by one for each additional row and column regardless of the length of the row respectively column in a linear algorithm. Thus the Divide and Conquer Algorithm is faster than the linear algorithm.

Contribution

- David Wünsch (s3665534):
  - Contribution to code
  - Search Tree
  - Runtime Comparison
  - Summary and Discussion
- Jesse Kroll (s3666778):
  - Contribution to code
  - Introduction
  - Greedy Approach
  - Summary and Discussion