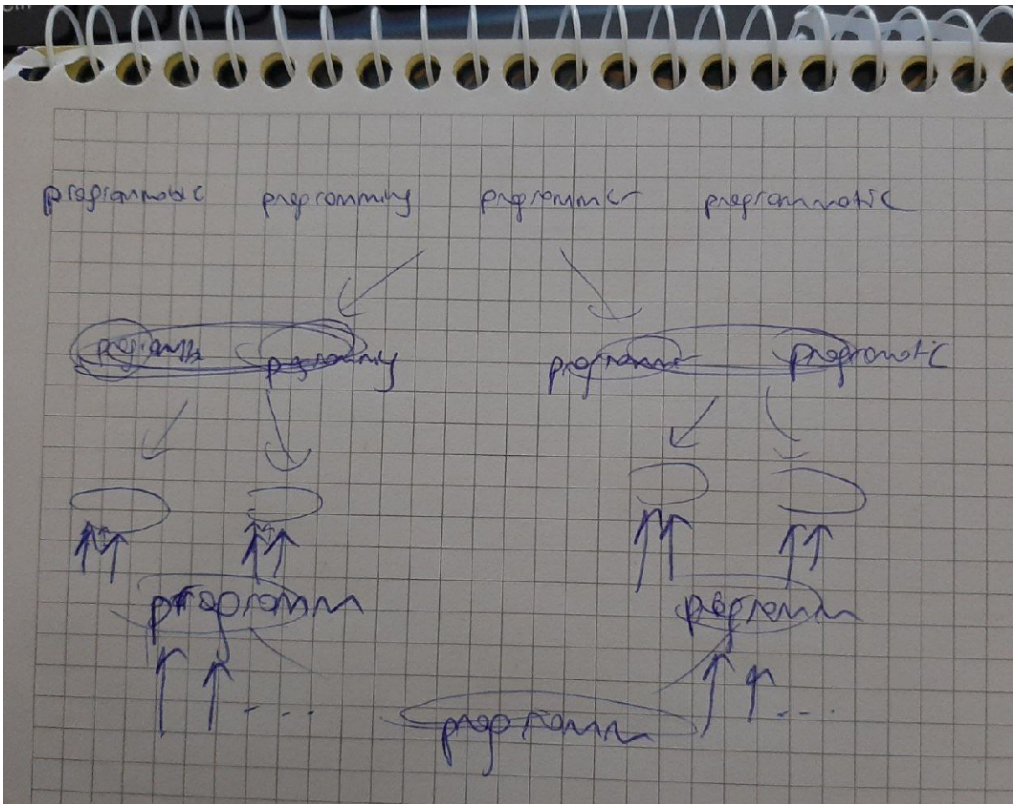
Answer to Question - 1

The most typic divide and conquer algorithm is Merge Sort. We can think a similar approach to this. In merge sort, after the divide an array for example, into it's smaller part: integers, we're comparing the integers according to their value. After conquering divided steps, we're merging divided steps according to a rule.

We have similar things. Words instead of integer array, character equality instead of value magnitude etc. We need to change the isBigger property with isCharEqual for example. General structure of our algorithm will be similar to the book's merge sort:

ALGORITHM *Mergesort*$(A[0..n-1])$
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A) //see below

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$
//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
if $B[i] \le C[j]$
$A[k] \leftarrow B[i];\ i \leftarrow i+1$
else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
$k \leftarrow k+1$
if $i = p$
copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$



Answer to Question - 2

In this question, critical point is not just finding the minimum and then the greatest after that minimum.
After looking the

minimum vs greatest

This also should be controlled

"element before minimum" vs new greatest

This operation should be done recursively, by holding the difference. After a much bigger difference obtained, day to buy and day to sell should be updated. If it is find a lower difference from previous difference, stop searching and return the result.

Also due to principle "you cannot sell before you buy", we first detect the minimum, and then search the maximum inside rest of the array from minimum.
Looking for minimum should be start from 0th index each time, to handle situations like

Input: [100, 110, 80, 90, 110, 70, 80, 80, 90]

that same values exist. This approach provides after selecting minimum, there will be a larger searching scale for maximum.

a) Most divide and conquer approaches divides the problem into two. Each time we need to find the min, max and the difference between these max and min. We're understanding that we've completed the comparisons after the pointer at the left reaches the pointer at the right.

b) This remind me quicksort approach. Where the two pointer exist and according to comparison of the elements that is pointed by those pointers, pointers can move forward/backward. Like in course book:

ALGORITHM *Quicksort*$(A[l..r])$
//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$

I define the partition according to smallest value in the current array each time.

c) Comparison

	Divide and Conquer	Other Solution
Worst Case Complexity	As the other divide and conquer algorithms, again worst case is O(nlogn), because we're dividing the problem into two subparts.	Function calls itself recursively, so that it can reach the O(n^2) time complexity in worst case. Partition takes the O(n) time in each call.

Answer to Question - 3

Definition of dynamic programming, taken from the course book is suggests:

“Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem’s solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.”

Here, I apply this suggestion.
Our subproblem is detecting the longest consecutively increasing subarray in the array. The only important information about the subarray we found is “how many element is increased consecutively” and we just need the maximum of that subarrays increasing amount. What I mean is that,
for the input:

Input: [1, 4, 5, 2, 4, 3, 6, 7, 1, 2, 3, 4, 7]

1, 4, 5
2, 4
3, 6, 7
1, 2, 3, 4, 7
are subarrays that consecutively increasing. But we only need the element amount of that subarrays.
Which is:
3
2
3
5

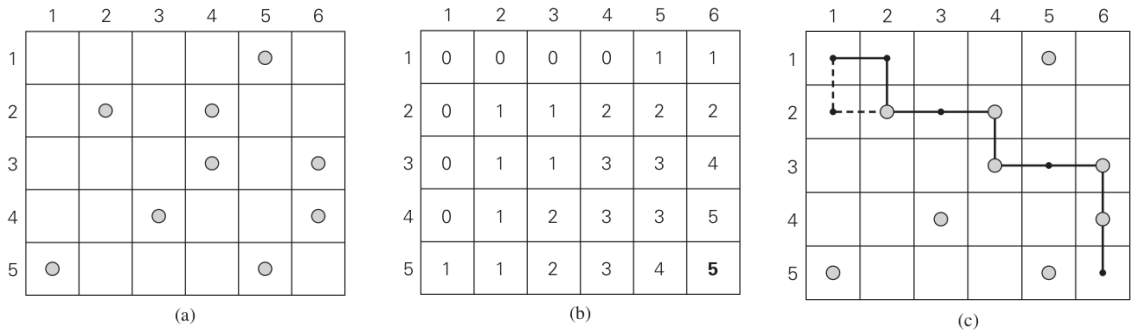
This is even more, if the next amount is not bigger than previous, dont hold it. Hold just the maximum
3
still 3
still 3
5

return 5

So, their occurrence hold in another array. And according to “is there a decreasing between adjacent elements”, make all occurrences 1111111… will provides solving the problem with dynamic approach. And update the maxOccurence when necessary.

Answer to Question - 4

a) Dynamic approach to that question is similar to the RobotCoinCollection in the course book, which is:



ALGORITHM *RobotCoinCollection*$(C[1..n, 1..m])$

//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at $(1, 1)$
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (n, m)
$F[1, 1] \leftarrow C[1, 1]$; **for** $j \leftarrow 2$ **to** m **do** $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
for $i \leftarrow 2$ **to** n **do**
    $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
    for $j \leftarrow 2$ **to** m **do**
        $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
return $F[n, m]$

Again we're constructing a 2D "F" board, that has the same size with our map board and it's first cell also assigned the first cell of the board, then filled considering our board. Dynamic part is here, if a value can be computed using previously computed values, compute it.

```
for i in range(1, n):
    for j in range(1, m):
        F[i][j] = max(F[i-1][j], F[i][j-1]) + board[i][j]
```

b) Greedy approach

Like other greedy approach methods, make a choice, “hoping” it is the best choice. In our case, we’re going to the cell that has the biggest value. But this has a correctness only for some specific inputs.

c) Comparison

	Correctness	Worst Case Complexity
Brute Force	The brute force approach considers the paths using for loops and recursion. It gives the correct result in any case, like as it is in any other programmes.	In each move we have 2 possible movement, so that worst case complexity is O(2^n)
Dynamic Programming	The dynamic programming has no significant difference than brute force, in terms of correctness. Because it only changes the way to solving approach. So it is also correct.	How many subproblem exists determines our complexity, which is number of row and column’s multiplication, n*m, which is nm and which is O(nm)
Greedy	Due to we’re “hoping” our choice is optimal for that step locally, this may not always be the true case.	Again, how many subproblem exists determines our complexity, which is nm and which is O(nm)