This assignment is **due on March 17** and should be submitted on Gradescope. All submitted work must be *done individually* without consulting someone else's solutions in accordance with the University's "Academic Dishonesty and Plagiarism" policies.

As a first step go to the last page and read the section: Advice on how to do the assignment.

Problem 1. (10 points)
Using O-notation, upperbound the running time of the following algorithm, where A is an array containing n integers. You can assume that mod is a basic mathematical operation that takes $O(1)$ time.

```
1: function ALGORITHM(A)
2:     b ← true
3:     for i ← 0; 2i < n; i++ do
4:         if A[2i] mod 2 ≠ 0 then
5:             b ← false
6:     for i ← 0; 2i < n − 1; i++ do
7:         if A[2i + 1] mod 2 ≠ 1 then
8:             b ← false
9:     return b
```

Solution 1. Line 2, 4, 5, and 7-9 take $O(1)$ time as they consist of assignments, basic mathematical operations, and comparisons. The loop on line 3 loops over $n/2$ elements and thus line 3-5 take $n/2 \cdot O(1)$ time, so $O(n)$ time. Similarly, the loop on line 6 loops over $n/2$ elements, so line 6-8 take $O(n)$ time. Thus the total running time is the sum of the running time of line 2, lines 3-5, lines 6-8, and line 9: $O(1) + O(n) + O(n) + O(1)$, so $O(n)$ in total.

Problem 2. (25 points)
We want to build a stack for integer elements that in addition to the operations we saw during the lecture (PUSH(e), POP(), TOP(), SIZE(), and ISEMPTY()), also supports a GETMINIMUM() operation that returns the minimum value of all elements stored in the stack. All operations should run in $O(1)$ time. Your data structure should take $O(n)$ space, where n is the number of elements currently stored in the data structure.

Example execution:

PUSH(23)		[23]
PUSH(4)		[23, 4]
GETMINIMUM()	returns 4	[23, 4]
POP()	returns 4	[23]
GETMINIMUM()	returns 23	[23]

Your task is to:

a) Design a data structure that supports the required operations in the required time and space.

b) Briefly argue the correctness of your data structure and operations.

c) Analyse the running time of your operations and space of your data structure.

Solution 2.

a) We need to ensure that the GETMINIMUM() operation always returns the correct minimum. In particular this means that if the current minimum is popped from the stack we need to report the correct minimum of the remainder of the stack, and do all this in constant time.
In order to do this efficiently, instead of pushing only the element we're supposed to, we're going to push tuples of $(element, minimum)$. This way we can easily access the current minimum by reading the second part of the top tuple. When pushing a new element, we need to compare the new element to the current minimum and if the new element is smaller, we push $(element, element)$, as the new element is the minimum.
Recall that the stack as we saw it in the lecture uses $O(N)$ time, as we used an array to implement it. Since we're allowed to use only $O(n)$ space, we need to change something here as well. Specifically, instead of using an array, we use a doubly-linked list D where we add nodes only at the front.
The SIZE() and ISEMPTY() operations just call the doubly-linked list versions of these operations. The PUSH(e), POP(), TOP(), and GETMINIMUM() operations now become a check to see if the stack/list is currently empty, followed by an inspection of the top tuple, and finally the insertion (for PUSH) or the removal (for POP) of the tuple from the stack/list.

```
1: function PUSH(e)
2:     if ISEMPTY() then
3:         D.INSERTBEFORE(head, (e, e))
4:     else
5:         (element, minimum) ← D.FIRST().element
6:         if e < minimum then
7:             D.INSERTBEFORE(head, (e, e))
8:         else
9:             D.INSERTBEFORE(head, (e, minimum))
```

```
1: function POP()
2:     if ISEMPTY() then
3:         return null
4:     else
5:         (element, minimum) ← D.FIRST().element
6:         D.REMOVE(head)
7:         return element
```

```
1: function TOP()
2:     if ISEMPTY() then
3:         return null
4:     else
5:         (element, minimum) ← D.FIRST().element
6:         return element
```

```
1: function GETMINIMUM()
2:     if ISEMPTY() then
3:         return −∞
4:     else
5:         (element, minimum) ← D.FIRST().element
6:         return minimum
```

b) We maintain the invariant that the minimum stored on the top of the stack is the minimum of the entire stack. When the first element is pushed, this indeed holds, since that element is by definition the minimum over all (1) elements on the stack. Assuming that the invariant holds before a later push, we observe that it also holds afterwards, as we compare the current minimum to the to be pushed element and push either (e, e) if the new element is smaller than the minimum, or $(e, minimum)$ otherwise.
The above invariant also ensures that we return the correct minimum when needed, as we return the minimum stored on the top of the stack. POP() "unpacks" the top tuple and returns the element it stores and removes the top of the stack, while TOP() only returns the element without removing it, as required.
All operations also check whether the stack is empty before they do anything, ensuring that we never remove something that doesn't exist and communicate this to the user (you should check for this, but how you handle this is up to you, as this isn't specified in the assignment).

c) All doubly-linked list operations used take $O(1)$ time as per the lecture. Other than that, we only use assignments and basic comparisons, each of which takes constant time. Since every function we defined performs a constant number of these operations, each takes $O(1)$ time as required.
Since we use a doubly-linked list and we use a constant amount of space per node in the list (storing two integers per node, plus a constant number of variables to keep track of the head and size), we ensure that the size of our stack is proportional to the number of elements we have pushed. In other words, when our data structure contains n elements, it uses $O(n)$ space.

Problem 3. (25 points)
We have $n \geq 1$ lighthouses of distinct positive integer height on a line and these lighthouses need to communicate with each other. We are given the heights of these lighthouses in an array A, where $A[i]$ stores the height of lighthouse i. Unfortunately, the area these lighthouses are in is so remote that it doesn't have phone lines and cell phone coverage is spotty at best. So the lighthouse keepers have found a different way to talk to each other: paper airplanes.

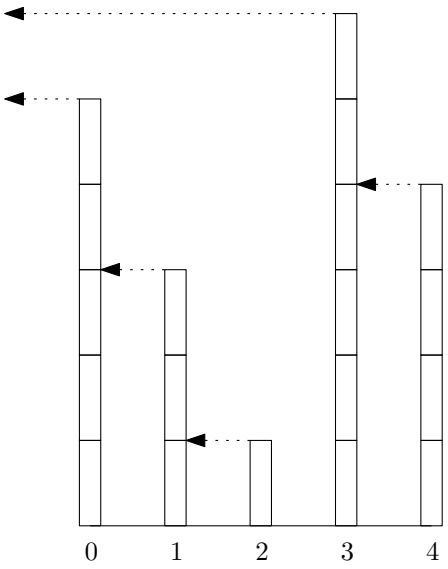
Assume all lighthouse keepers are (former) olympic champions in paper airplane throwing, meaning that they can reach any lighthouse as long as there's no lighthouse higher than the thrower's lighthouse between them (the lighthouses are equipped with paper airplane catchers, so hitting the wall of the lighthouse counts as reaching it). Of course, because of the wind coming from the ocean, lighthouse keepers can only send their paper planes to the left (towards the land). More formally, lighthouse k can send a paper airplane to lighthouse i ($i < k$), if there is no lighthouse j such that $i < j < k$ and $A[k] < A[j]$. Note that lighthouse i itself can be higher than lighthouse k because of the paper airplane catchers. We are interested in determining the leftmost lighthouse that each of the lighthouse keepers can reach. To distinguish between a paper airplane stopping at lighthouse 0 and passing over all lighthouses to the left of its starting point, we record the latter as -1 (imagining an infinitely high lighthouse preceding all others where the airplane stops).

Example:
$A : [5, 3, 1, 6, 4]$: return $[-1, 0, 1, -1, 3]$. A paper airplane thrown by the lighthouse keeper of lighthouse 0 passes over all lighthouses to its left and is thus recorded as -1. Lighthouse 1 can send their airplanes to lighthouse 0, but not beyond. Lighthouse 2 can communicate with lighthouse 1, but not with lighthouse 0, since lighthouse 1 stops any airplanes throw from lighthouse 2. Lighthouse 3 is higher than any of the previous lighthouses and thus its airplanes pass over them. Finally, lighthouse 4 can send airplanes to lighthouse 3, but no further.

Your task is to design an algorithm that computes for each lighthouse the leftmost lighthouse that its keeper can reach. For full marks, your algorithm should run in $O(n)$ time.

a) Describe your algorithm in plain English.

b) Prove the correctness of your algorithm.



c) Analyze the time complexity of your algorithm.

Solution 3.

a) We could loop over all lighthouses and for each then loop over all lighthouses preceding it to determine the answer. However, this would take $O(n^2)$ time and thus not satisfy our time bound. So we'll need to do something better. One observation we can make is that when processing the lighthouses from left to right, when we find a lighthouse of height x, we'll never again output any lighthouse to its left of height at most x, since any airplane passing over the lighthouse of height x will also pass over any of at most that height. So we can speed up the search by keeping track of an ordered set of lighthouses of decreasing height. When our current search passes over the smallest in this set, we know we'll never need it again, so it can be removed. By repeatedly doing this, we can find the leftmost lighthouse that we can reach.
To maintain this ordered set of lighthouses of decreasing height, we'll use a stack S. When processing a lighthouse, we compare its height to the height of the top of the stack. If its height is smaller, the top of the stack is the leftmost lighthouse it can reach. Otherwise, we repeatedly pop the top of the stack until we either find a lighthouse taller than our current one (in which case we output that lighthouse), or until the stack is empty (in which case we output -1 as we just passed over all lighthouses). Finally, we push our lighthouse onto the stack so it's considered in future iterations. Once our loop concludes, we return our result.

```
1:  function reachableLighthouses(A)
2:      S ← empty stack fo size n
3:      result ← empty array of length n
4:      for i ← 0 to n − 1 do
5:          while S ≠ ∅ and A[S.top()] ≤ A[i] do
6:              S.pop()
7:          if S = ∅ then
8:              result[i] ← −1
9:          else
10:             result[i] ← S.top()
11:         S.push(i)
12:     return result
```

b) We start by proving our observation: when we process a lighthouse j of height x, we will never need to report any lighthouse i ($i < j$) of height at most x. We distinguish two cases for lighthouse k ($j < k$): lighthouse k is smaller than lighthouse j and lighthouse k is at least the same height as lighthouse j. If it's smaller, its paper airplane can't pass over lighthouse j and since lighthouse i is to the left of lighthouse j, the plane would stop before reaching lighthouse i. If it's at least the same height, the airplane of lighthouse k passes over lighthouse j, but it would also pass over lighthouse i, since i's height is at most that of j. Hence, in this case we also don't report lighthouse i.
The above observation shows that by maintaining a decreasing set of lighthouses (from left to right) and removing the ones we pass over, we correctly keep track of the lighthouses that could still be reported. Since we process the lighthouses from left to right, our stack contains the in that order. Thus, since our algorithm pops the lighthouses that are passed over and pushes the last processed lighthouse, we indeed have all lighthouses that could be reported on the stack. By reporting the first lighthouse on the stack that's higher than our current lighthouse (or -1 if the stack is empty and therefore no such lighthouse exists), we thus report the leftmost lighthouse it can reach.

c) Line 2-3 take $O(1)$ time if we don't explicitly set the content of these empty structures, which in this case we don't have to do since we're going to overwrite it anyway. Line 6-11 take $O(1)$ time, as these are all basic comparisons, assignments and constant time stack operations. If we pass *result* by reference instead of copying it over explicitly on line 12 this takes $O(1)$ time as well (we can also copy it in $O(n)$ time, if preferred). The meat of the analysis is in the loops: for the while loop, we observe that we can pop only as much as we push and since we push every element exactly once, we can pop at most n times over the full execution of the algorithm. This implies that line 5-6 take $O(n)$ time over the full execution. The for loop is executed once for each element in the array and thus we spend $n \cdot O(1)$ time on lines 4 and 7-11 in total. Thus the total running time comes down to $O(1) + O(1) + O(n) + O(n) + O(1)$ which simplifies to $O(n)$ as required.

Advice on how to do the assignment

- Assignments should be typed and submitted as pdf (no pdf containing text as images, no handwriting).
- Start by typing your student ID at the top of the first page of your submission. Do **not** type your name.
- Submit only your answers to the questions. Do **not** copy the questions.
- When asked to give a plain English description, describe your algorithm as you would to a friend over the phone, such that you completely and unambiguously describe your algorithm, including all the important (i.e., non-trivial) details. It often helps to give a very short (1-2 sentence) description of the overall idea, then to describe each step in detail. At the end you can also include pseudocode, but this is optional.
- In particular, when designing an algorithm or data structure, it might help you (and us) if you briefly describe your general idea, and after that you might want to develop and elaborate on details. If we don't see/understand your general idea, we cannot give you marks for it.
- Be careful with giving multiple or alternative answers. If you give multiple answers, then we will give you marks only for "your worst answer", as this indicates how well you understood the question.
- Some of the questions are very easy (with the help of the slides or book). You can use the material presented in the lecture or book without proving it. You do not need to write more than necessary (see comment above).
- When giving answers to questions, always prove/explain/motivate your answers.
- When giving an algorithm as an answer, the algorithm does not have to be given as (pseudo-)code.
- If you do give (pseudo-)code, then you still have to explain your code and your ideas in plain English.
- Unless otherwise stated, we always ask about worst-case analysis, worst case running times, etc.
- As done in the lecture, and as it is typical for an algorithms course, we are interested in the most efficient algorithms and data structures.
- If you use further resources (books, scientific papers, the internet,...) to formulate your answers, then add references to your sources and explain it in your own words. Only citing a source doesn't show your understanding and will thus get you very few (if any) marks. Copying from any source without reference is considered plagiarism.