Problem 1. (10 points)
Consider the following algorithm that given an array A of length n produces an array B where $B[i]$ is the sum of the 16 elements of A following $A[i]$.

```
1: function ALGORITHM(A)
2:     n ← length of A
3:     B ← new array of size n − 16
4:     for i ∈ [0 : n − 16] do
5:         B[i] ← 0
6:         for j ∈ [0 : 16] do
7:             B[i] ← B[i] + A[i + j + 1]
8:     return B
```

Use O-notation to upperbound the running time of the algorithm.

Solution 1. Line 2 takes $O(1)$ time, while Line 3 takes $O(n)$ time. The inner for loop (lines 6-7) is always executed 16 times. Each iteration takes $O(1)$ time as it only performs a constant number of constant time operations, so the overall running time of the inner loop is $16 \cdot O(1) = O(1)$. The outer loop (lines 4-7) is executed $n - 16$ times. Each iteration takes $O(1)$ time, so the overall running time of the outer loop is $O(n)$. Returning the array B takes at most linear time, so the overall running time is $O(n)$.

Problem 2. (25 points)
We want to build a queue for integer elements that in addition to the operations we saw during the lecture, also supports a DIFFERENCE() operation that on a given queue holding the elements $Q_1, \ldots, Q_n$ returns the difference between the sum of the first half of the elements and the sum of the second half of the elements:

$$\sum_{i=1}^{\lceil n/2 \rceil} Q_i - \sum_{i=\lceil n/2 \rceil + 1}^{n} Q_i.$$

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. 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) The first thing we note is that since all operations need to run in constant time, we can't compute the difference when this is queried, so we'll have to maintain this. The main problem with this is that the elements in the middle of the queue can switch from being part of the first half to being part of the second half and vice versa as we enqueue or dequeue elements, making maintaining this difference tricky. Since we need to use $O(n)$ space, we also can't use a very big array, so we'll have to base this on a doubly-linked list instead.
So we'll maintain two queues, each being a doubly linked list and we'll ensure that the first queue L_1 contains the first $\lceil n/2 \rceil$ elements and the second queue L_2 contains the remaining elements. As we enqueue and dequeue elements, we'll be moving elements from L_1 to L_2 and vice versa to maintain this property. Each of these two queues will also maintain the sum of its elements in a sum_1 and sum_2 variable (both initialized as 0), respectively.
Next, we describe the queue's operations ENQUEUE(e), DEQUEUE(), FIRST(), SIZE(), and ISEMPTY() and the new operation DIFFERENCE(). We start with the last four operations: FIRST() simply returns L_1.FIRST(), SIZE() returns L_1.SIZE() + L_2.SIZE(), ISEMPTY() returns whether SIZE() is equal to 0, and DIFFERENCE() returns $sum_1 - sum_2$.
ENQUEUE(e) adds the new element e at the end of L_2 (i.e., L_2.INSERTAFTER (L_2.LAST(), e)). We also add e to sum_2. Next, we need to check if there aren't too many elements in L_2. To do this, we check whether L_1.SIZE() is less than $\lceil \text{SIZE}()/2 \rceil$. If this is the case, we remove the first element e of L_2 (i.e., L_2.REMOVE(L_2.FIRST())) and add e at the end of L_1. We also subtract e from sum_2 and add e to sum_1.
DEQUEUE() works similarly, except that it removes from the front instead of adding at the back. Let *return* be the current first element in L_1. We remove this element from L_1 (i.e., L_1.REMOVE(L_1.FIRST())). We also subtract *return* from sum_1. Next, we need to check if there aren't too few elements in L_1. To do this, we again check whether L_1.SIZE() is less than $\lceil \text{SIZE}()/2 \rceil$. If this is the case, we remove the first element e of L_2 and add e at the end of L_1, again also subtracting e from sum_2 and adding e to sum_1. Finally, we return *return*.

b) We maintain the invariant that L_1 contains the first half of the elements, L_2 contains the second half of the elements, sum_1 is the sum of the elements in L_1, and sum_2 is the sum of the elements in L_2, while enqueuing and dequeuing elements. Assuming this invariant, FIRST(), SIZE(), and ISEMPTY() are correct, since they build on the existing doubly-linked list operations combined with the fact that L_1 stores the first half of the elements and L_2 stores the second half of the elements. Similarly, DIFFERENCE() correctly returns the specified difference between L_1 and L_2.
Initially (before any elements are inserted), the invariant holds, as L_1 and L_2 are empty and the sums are initialized to 0.
When we enqueue a new element, it's added at the end of the queue (i.e., at the end of L_2). If this disrupts the balance between L_1 and L_2, we move the first element of L_2 to the end of L_1. Since the invariant held before the new element was added, we know we're moving the middle element to restore balance and moving a single element suffices. By updating the sums accordingly, we maintain the second half of the invariant as well.
When we dequeue an element, it's removed from the front of the queue (i.e., the front of L_1). If this disrupts the balance between L_1 and L_2, we move the first element of L_2 to the end of L_1. Using the same argument as before, this ensures that the invariant holds after the operation as well.

c) Next, we analyse the running time of each of the operations. FIRST() calls an $O(1)$ time operation of L_1 and thus runs in $O(1)$ time itself. SIZE() calls two constant time operations and adds up their results, which also takes constant time, resulting in $O(1)$ time in total. ISEMPTY() calls an $O(1)$ time operation and performs a single comparison, resulting in $O(1)$ time. DIFFERENCE() subtracts two variables from each other and returns the result, which takes constant time, as required. ENQUEUE(e) calls a constant number of constant time operations on L_1 and L_2 and performs a constant number of additions and subtractions, which results in $O(1)$ time overall. Similarly, DEQUEUE() calls a constant number of constant time operations on L_1 and L_2 in addition to a constant number of additions and subtractions, also resulting in $O(1)$ time overall. Hence, all operations run in constant time, as required.
Finally, we analyse the space complexity. Since we use two doubly-linked lists that contain a total of n elements (requiring $O(n)$ space) and a constant number of extra variables (requiring an additional $O(1)$ space), the total space complexity of our data structure is $O(n)$.

Problem 3. (25 points)
In robotics, a common problem is figuring out what type of configuration a set of robots is in using very limited information. Here, we look at a small example of this.

You're given n robots (each robot is uniquely identified simply using an integer between 1 and n) and you're told that we know that the n robots are on a line (say the x-axis), but we don't know the order of the robots along this line.

Unfortunately, the robots don't have GPS, so they can't tell you their coordinate along the line. What you do have is a function ADJACENT(A,B) which takes two robots (i.e., integers) a and b as input and returns whether robot a and robot b are adjacent. You can assume that a single call to this function runs in $O(1)$ time. You're asked to design an algorithm that outputs the order of the robots along the line. You can output the robots from left to right, or from right to left. For full marks, your algorithm should run in $O(n^2)$ time.

Example:
If we have three robots 1, 2, and 3 and the (unknown) order along the line from left to right is $2, 3, 1$, your algorithm should return either $2, 3, 1$ (the left to right order) or $1, 3, 2$ (the right to left order). Returning any order other than these two is incorrect. In this example ADJACENT(1,2) would return false, since robot 1 and robot 2 aren't adjacent on the line. ADJACENT(1,3) would return true, since robot 1 and robot 3 are adjacent.

Remember to:

a) describe your algorithm in plain English,

b) argue its correctness, and

c) analyze its time complexity.

Solution 3.

a) We don't have a lot to work with, so we'll have to find a way to determine the order using only adjacency information. When we consider the robots on a line, we observe that there are two special robots: the leftmost one and the rightmost one, as these two robots are adjacent to only a single other robot, while all other robots are adjacent to two robots.
This observation leads us to the following idea: first find one of the two special robots (either one will do) and then repeatedly find the next robot in the sequence. For the first robot, the next robot is simply the only robot it's adjacent to, while for subsequent robots, we need to ensure we report the correct one of its two neighbours (i.e., the one not preceding it in the ordering).
To find the first robot, we check the robots one by one, checking for each robot i how many robots j robot i is adjacent to using the ADJACENT(i,j) function on each pair of robots. When we find a robot i that's adjacent to only a single other robot, we add robot i to our result. We also append its one neighbour to the result, as we now know that this is the second robot along the line.
Now that we have two robots, we can use them to find the next one: we keep track of the current robot (initially robot j) and the previous robot (initially robot i). We then test every robot k to determine which other robot is adjacent to the current robot (ignoring the previous robot to ensure we keep extending our result in the correct direction), again using the ADJACENT(*current*,k) function. When we find the next robot k, we add k to the result, update the previous robot with the current one and setting the current one to k. We repeat this process until we reach a robot that has only a single neighbour, indicating that we reached the end of the line of robots, at which point we output our result.
We note that the above idea assumes that there are at least three robots. If there are 0, 1, or 2 robots, we simply return the robots in any order.

b) We'll first argue that our special case (0, 1, or 2 robots) is correct: when there are 0 robots, there's nothing to report, so reporting all 0 robots in any order suffices. If there's only 1 robot, that robot by itself is the ordering and thus reporting it suffices. If there are 2 robots, reporting them in either order ensures that we report them either from left to right or from right to left, as there are only two possible orderings of 2 robots.
To show that our algorithm is correct when there are at least 3 robots, we'll maintain the following invariant: when our result contains x robots, these robots are consecutive along the line, where the first robot is either the leftmost or rightmost robot. We prove this by indiction. For the base case, we can see that this invariant holds when our result contains no robots, one robot (it contains only the leftmost or rightmost robot by design), or two robots (the first robot is either the leftmost or rightmost one by design and the next robot is the robot adjacent to it). Now, say that we know our invariant holds for y robots. When adding the next robot to our result, we know that the newly added robot is adjacent to the last of the earlier y robots and that it isn't the $(y-1)$th robot, since we explicitly check for that. Hence, it's adjacent to the yth robot and thus we have that these $y+1$ robots are consecutive along the line. Since we didn't change the first robot, it's still either the leftmost or rightmost robot, proving the invariant. Since the invariant holds throughout the execution, this implies that at the end, our result contains all n robots and we have compute the ordering of the robots as required.

c) Checking the special cases when $n = 0, 1, 2$ takes constant time, as does returning the result in those cases. Otherwise, finding the first robot takes (at most) a quadratic number of calls to ADJACENT(i,j), as we check for each of the n robot whether it's the leftmost or rightmost one and checking a single robot requires checking whether it's adjacent to each of the other robots. Since a single call (plus some bookkeeping of counters) takes $O(1)$ time, this part thus takes $O(n^2)$ time. Once we have the first robot, finding each subsequent robot takes (at most) a linear number of calls to ADJACENT(i,j), as we need to find the next robot in the sequence and the only way we have of doing that is explicitly checking each of them. Each call (plus the subsequent bookkeeping of the current, previous, and next robot) takes constant time, thus requiring linear time per robot. Since there are a linear number of robots in total, this part thus takes $O(n^2)$ in total for all robots. Maintaining the result can be done in linear time overall, if we use for example an array of size n, where we can set the next element in constant time and report the ordering by returning a pointer to the array (constant time) or by explicitly outputting the array (linear time). Either way, the overall running time is $O(n^2)$, as required.