Chapter 15 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. You are given two containers of size n: a sorted array and a sorted linked list. What are the time complexities of searching for a value in each of these two containers, if you use the most efficient algorithm?
   A) Sorted array: $\Theta(\log(n))$, Sorted linked list: $\Theta(\log(n))$
   B) Sorted array: $\Theta(\log(n))$, Sorted linked list: $\Theta(n)$
   C) Sorted array: $\Theta(n)$, Sorted linked list: $\Theta(\log(n))$
   D) Sorted array: $\Theta(n)$, Sorted linked list: $\Theta(n)$
   E) None of the above

2. You are given a mystery container of size n, and you are told that inserting an element into the position directly before any given iterator takes $\Theta(1)$ time (i.e., `insert_before(iterator, val)`). Which of the following could be the mystery container you were given?
   A) Array
   B) Deque
   C) Singly-linked list
   D) Doubly-linked list
   E) More than one of the above

3. Which one of the following statements is **TRUE** about the contents of the underlying data vector used to implement a binary heap?
   A) The binary heap's underlying data vector is an ordered and sorted container
   B) The binary heap's underlying data vector is an ordered container, but it is not a sorted container
   C) The binary heap's underlying data vector is a sorted container, but it is not an ordered container
   D) The binary heap's underlying data vector is neither an ordered nor sorted container
   E) None of the above

4. If you are given a sorted array of n integers, what is the worst-case time complexity of finding *and* removing a specified value from this array, if you use the most efficient algorithm?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

5. You are given a sorted array of size n that contains all but one integer in the range $[0, n]$. What is the worst-case time complexity of finding the missing number, if you use the most efficient algorithm? Assume all integers in the array are unique. For example, given the array $[0, 1, 2, 4, 5]$, the algorithm would return 3.
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

6. Suppose you are given an unsorted array of n integers (where n is odd), and you are told that all but one of the values also have their additive inverse inside the array. What is the worst-case time complexity of finding the value **without** an additive inverse inside the array if you use the most efficient algorithm? For example, given the array [4, 8, -3, -2, -8, 5, -5, -4, 3], the algorithm would return -2.
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

7. You are given an *unsorted* array of positive integers that is sorted in ascending order, and you are told that two numbers in this array sum up to a target number T. What is the worst-case time complexity of finding the two numbers that sum to T, if you use the most efficient algorithm that does **NOT** create any additional containers? For example, given the array [5, 9, 13, 2, 7] and $T = 11$, the algorithm would return the pair [9, 2], since this is the pair of values that sum to 11. If multiple pairs sum to T, the algorithm may return any of these pairs.
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

8. Given a sorted (ascending) array of integers of size n, implement a function that returns a sorted (ascending) array that contains the square of each number. For example, given the array [-7, -3, -2, 1, 4, 6], you would return the array [1, 4, 9, 16, 36, 49].

   ```
   std::vector<int32_t> sorted_squares(const std::vector<int32_t>& vec);
   ```

   Your solution must perform **BETTER** than $\Theta(n\log(n))$ time.

9. You are given a non-empty vector of distinct elements, and you want to return a vector that stores the previous greater element that exists before each index. If no previous greater element exists, `-1` is stored.

   Example 1: Given `vec = [11, 16, 15, 13]`, you would return `[-1, -1, 16, 15]`.
   Example 2: Given `vec = [19, 18, 12, 14, 13]`, you would return `[-1, 19, 18, 18, 14]`.

   ```
   std::vector<int32_t> previous_greater_element(const std::vector<int32_t>& vec);
   ```

   Your solution should be implemented in $O(n)$ time and with $O(n)$ auxiliary space, where n is the length of the vector.

10. You are given a sorted array consisting of integers where every element appears exactly twice, except for one element which appears exactly once. Implement a function that returns the single element.

    Example: Given `vec = [1, 1, 2, 3, 3, 4, 4]`, you would return `2`.

    ```
    int32_t find_single_element(const std::vector<int32_t>& vec);
    ```

    Your solution should be implemented in $O(\log(n))$ time and with $O(1)$ auxiliary space, where n is the length of the vector.

11. You are given a collection of intervals. Implement a function that merges all of the overlapping intervals.

    Example 1: Given `vec = [[1, 3], [2, 6], [8, 10], [15, 18]]`, you would return `[[1, 6], [8, 10], [15, 18]]`.
    Example 2: Given `vec = [[4, 5], [1, 4]]`, you would return `[[1, 5]]`.

    ```
    struct Interval {
      int32_t start;
      int32_t end;
    };

    std::vector<Interval> merge_intervals(const std::vector<Interval>& vec);
    ```

    Your solution should be implemented in $O(n\log(n))$ time and with $O(n)$ auxiliary space, where n is the length of the vector of intervals.

12. You are given a vector of integers, `vec`, and you are told to implement a function that moves all elements with a value of 0 to the end of the vector *while maintaining the relative order of the non-zero elements*.

    Example: Given the initial vector `[0, 1, 0, 4, 3]`, you should rearrange the contents of the vector so that the final ordering of elements is `[1, 4, 3, 0, 0]`.

    ```
    void shift_zeros(std::vector<int32_t>& vec);
    ```

    Your solution should be implemented in $O(n)$ time and with $O(1)$ auxiliary space, where n is the length of the vector.

13. You are given a $m \times n$ matrix in the form of a vector of vectors that has the following properties:
    - integers in each row are sorted in ascending order from left to right
    - integers in each column are sorted in ascending order from top to bottom

    Implement a function that searches for a value in this matrix and returns whether the element can be found.

    Example: Given the following matrix:

    ```
    [ [ 1,  4,  7, 11, 15],
      [ 2,  5,  8, 12, 19],
      [ 3,  6,  9, 16, 22],
      [10, 13, 14, 17, 24],
      [18, 21, 23, 26, 30]  ]
    ```

    and a target value of 5, you would return `true`. Given a target value of 20, you would return `false`.

    ```
    bool is_value_in_matrix(const std::vector<std::vector<int32_t>>& matrix, int32_t target);
    ```

    Your solution should be implemented in $O(m+n)$ time and with $O(1)$ auxiliary space.

14. You are given a vector of integers that is sorted in ascending order. However, this vector has been rotated at some pivot unknown to you beforehand. For instance, the vector `[1, 2, 3, 4, 5]` may be given to you in the form `[3, 4, 5, 1, 2]`, where the vector is rotated at 3. You may assume that no duplicate exists in this array. Implement a function that finds the minimum element in the vector.

    Example: Given the input vector `[3, 4, 5, 1, 2]`, you would return 1.

    ```
    int32_t find_rotated_minimum(const std::vector<int32_t>& vec);
    ```

    Your solution should be implemented in $O(\log(n))$ time and with $O(1)$ auxiliary space, where n is the length of the vector.

15. You are given a vector of integers and a target value k. Implement a function that returns the pair of elements whose sum is closest to k. The smaller element should go first in the pair that is returned.

    Example: Given the input vector `[29, 30, 40, 10, 28, 22]` and a target of $k = 54$, you would return the pair `[22, 30]`.

    ```
    std::pair<int32_t, int32_t> closest_sum_to_k(const std::vector<int32_t>& vec, int32_t k);
    ```

    Your solution should be implemented in $O(n\log(n))$ time and with $O(\log(n))$ auxiliary space, where n is the length of the vector.

16. You are given an array of positive integers `vec` and a positive integer `target`. Implement a function that returns the length of the smallest subarray whose sum is greater than or equal to `target`. If there is no such subarray, return `std::numeric_limits<int32_t>::max()` (`INT_MAX`). A subarray is a contiguous section of an array (e.g., `[2, 3, 4]` is a subarray of `[1, 2, 3, 4, 5]`).

    Example: Given the input vector `[4, 2, 9, 1, 2, 8, 5, 3]` and a `target` value of `12`, you would return `2`, since the smallest subarray that sums to a value of at least 12, `[8, 5]`, has a length of 2.

    ```
    int32_t min_subarray_length(const std::vector<int32_t>& vec, int32_t target);
    ```

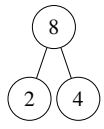
    Your solution should be implemented in $O(n)$ time and with $O(1)$ auxiliary space, where n is the length of the vector.

Chapter 15 Exercise Solutions

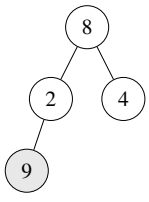
1. **The correct answer is (B).** The time complexity of finding an element in a sorted array is $\Theta(\log(n))$ because you are able to use binary search. However, the $\Theta(\log(n))$ performance of binary search is only possible if random access is available, which linked lists do not support. Hence, even if a linked list is sorted, the time complexity of finding a value would still require a $\Theta(n)$ traversal in some form.

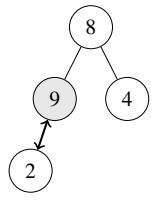
2. **The correct answer is (D).** The doubly-linked list is the only container that supports $\Theta(1)$ before any given iterator. Arrays and deques may require elements to be shifted after the insertion point, which could take $\Theta(n)$ time. Singly-linked lists do not provide direct access to previous node, so you may need a $\Theta(n)$ traversal to find and update the next pointer of the the node directly before the insertion point.

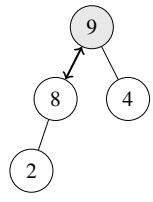
3. **The correct answer is (D).** The data vector of a binary heap is neither sorted nor ordered. For example, `[1, 3, 2, 6, 5, 4]` is a valid min-binary heap, but the contents of the vector are not sorted in ascending order. Additionally, adding an element to a binary heap may change the relative ordering of elements already in the heap, which indicates that the data vector is also not ordered. To illustrate this, consider the array representation of `[8, 2, 4]`:

    Now, let's add 9 to the binary heap and fix the heap invariant:

    The underlying data vector of the binary heap now stores the values `[9, 8, 4, 2]`. Notice that the relative order of 2 and 4 were switched after the insertion, with 2 now ending up after 4 when it was before 4 earlier.

4. **The correct answer is (C).** Finding a given value in a sorted array takes $\Theta(\log(n))$ time due to binary search, but removing any element may take up to $\Theta(n)$ in an array (since elements may need to be shifted after the removal point). The cost of removal is dominant, so the overall time complexity of finding and removing any element ends up being $\Theta(n)$.

5. **The correct answer is (B).** This is actually a binary search question. Consider the following array where the number 7 is missing.

0	1	2	3	4	5	6	8	9	10
0	1	2	3	4	5	6	7	8	9

    Notice that all numbers before the missing number share their values with their index positions (e.g., 0 at index 0, 1 at index 1, etc.), while all numbers after the missing number have a value one larger than their index positions (e.g., 8 at index 7, 9 at index 8, etc.). We can use this knowledge to do a binary search. First, we visit the element in the middle and see if its value is equal to its index. If it is, then the missing number has not occurred yet and must be to the right of the middle value. Otherwise, something before the middle value must have messed the indices up, and the missing element must be to the left. Since binary search cuts the search space in half at every iteration, the worst-case time complexity of solving this problem is $\Theta(\log(n))$.

6. **The correct answer is (C).** The most efficient solution can be obtained with the following insight: since a number and its additive inverse must sum to zero, we can find the value without an additive inverse by just summing up the entire vector (since all other additive inverse pairs cancel each other out). Thus, a sum of the values would give us our solution, which takes $\Theta(n)$ time.

7. **The correct answer is (D).** This is a problem that can be solved efficiently if the vector were sorted beforehand. Without sorting, we would have to compare each element with all the remaining elements, which would take $\Theta(n^2)$ time. However, if the vector were sorted, we could keep track of two pointers, one moving from the left to right and the other moving from the right to left. If the sum of left and right is greater than the target value, we decrement the right pointer. If the sum is less than the target value, we increment the left pointer. This allows us to obtain the solution with just a linear pass of the vector. Since sorting acts as the bottleneck of this entire process, the overall time complexity is $\Theta(n\log(n))$.

8. The core challenge of this problem comes from the fact that our solution must be better than $\Theta(n\log(n))$ time, which prevents us from just squaring each value and sorting the entire array afterward. It turns out that this problem can be solved in linear time using the two-pointer technique, thanks to the fact that the initial array is sorted. We start with two pointers (or indices), one that points to the leftmost element, and one that points to the rightmost element. Then, we use the absolute values of the two values pointed to by left and right to construct the sorted order of squares in descending order, since we know the largest square must be one of these two values. Consider the input array in the provided example:

The value at the left pointer is -7, while the value at the right pointer is 6. -7 has the larger absolute value, so we know that its square of 49 must be the largest value in our sorted output. We write this value to the end of our output vector and increment the left pointer.

The value at the left pointer is -3, while the value at the right pointer is 6. 6 has the larger absolute value, so we know that its square of 36 must be the next largest value in our sorted output. We write this value to the end of our output vector and decrement the right pointer.

The value at the left pointer is -3, while the value at the right pointer is 4. 4 has the larger absolute value, so we know that its square of 16 must be the next largest value in our sorted output. We write this value to the end of our output vector and decrement the right pointer.

Continuing this process, we will eventually build our entire solution in sorted order.

An implementation of this solution is shown below:

```
1   std::vector<int32_t> sorted_squares(const std::vector<int32_t>& vec) {
2     std::vector<int32_t> result(vec.size());
3     int32_t left = 0, right = vec.size() - 1;
4     for (int32_t i = vec.size() - 1; i >= 0; --i) {
5       if (std::fabs(vec[right]) > std::fabs(vec[left])) {
6         result[i] = vec[right] * vec[right];
7         --right;
8       } // if
9       else {
10        result[i] = vec[left] * vec[left];
11        ++left;
12      } // else
13    } // for i
14    return result;
15  } // sorted_squares()
```

9. The naïve solution would be to run a doubly-nested loop, where the outer loop visits every element and the inner loop finds the previous element that is greater. This solution would take $\Theta(n^2)$ time. Can we use our knowledge of data structures to improve this solution? It turns out that we can use a stack to solve this problem in linear time. Since we want to identify the largest element directly before each given element, pushing each of the elements into the stack (from left to right) gives us the ability to access the most recent element that is larger, provided that we pop out all of the smaller elements every time we push in a new value. For example, consider the example `[19, 18, 12, 14, 13]`. We know that the is no previous greater element for the first value, `19`, so the first value of our solution array is `-1`. However, we also push the value `19` into an auxiliary stack, which will be used to solve the remaining elements in our input:

The next value in our input is `18`. We look at the top of the auxiliary stack and compare its value with `18`. Since the top value, `19`, is larger, `19` must be the previous greater element of `18`. We push `19` into our solution array, and `18` onto the auxiliary stack.

The next value in our input is `12`. We look at the top of the auxiliary stack and compare its value with `12`. Since the top value, `18`, is larger, `18` must be the previous greater element of `12`. We push `18` into our solution array, and `12` onto the auxiliary stack.

The next value in our input is `14`. We look at the top of the auxiliary stack and compare its value with `14`. However, the top value, `12`, is smaller than the current value of `14`. This means that `12` is not the previous greater element of `18`, so we pop it out of the stack to reveal the next most recent value, which is `18`. `18` is larger than `14`, so the previous greater element of `14` is `18`, and we push `18` into our solution array and `14` onto the auxiliary stack. Notice that we are able to safely pop out `12` because `12` cannot be the previous greater element for any future value, given that the next element of `14` is already larger, and any element for which `12` is greater is also less than `14`.

The last value in our input is `13`. The top element in the auxiliary stack, `14`, is larger, so `14` is our final previous greater element. This gives us our final solution of `[-1, 19, 18, 18, 14]`. An implementation of this solution is shown below:

```
1  std::vector<int32_t> previous_greatest_element(const std::vector<int32_t>& vec) {
2    std::stack<int32_t> s;
3    s.push(vec[0]);
4    std::vector<int32_t> result = {-1};  // first value is always -1
5    int32_t left = 0, right = nums.size() - 1;
6    for (size_t i = 1; i < vec.size(); ++i) {
7      while (!s.empty() && s.top() < vec[i]) {
8        s.pop();
9      } // while
10     s.empty() ? result.push_back(-1) : result.push_back(s.top());
11     s.push(vec[i]);
12   } // for i
13   return result;
14 } // previous_greatest_element()
```

Why does this solution run in $\Theta(n)$ time? Notice that each of the n elements can only be pushed/popped from the stack at most once. Since each push/pop takes $\Theta(1)$ time, and this serves as the bottleneck of the entire algorithm, the total cost of our solution is $n \times \Theta(1) = \Theta(n)$.

10. This problem can be solved with binary search. To see why, consider the example:

1	1	2	3	3	4	4
0	1	2	3	4	5	6

The number missing a second value is `2`. All of the values before `2` have their first occurrence located at an even index, while all values after `2` have their first occurrence located at an odd index. Therefore, you can use the index position of the first repeated value to halve the search space: if the first occurrence of a number has an even index, then the single number must be to the right; otherwise, if the first occurrence has an odd index, then the single number must be to the left. An implementation of this is shown below:

```
int32_t find_single_element(const std::vector<int32_t>& vec) {
  size_t left = 0, right = vec.size() - 1;
  while (left < right) {
    size_t mid = (left + right) / 2;
    if ((mid % 2 == 0 && vec[mid] == vec[mid + 1]) ||
        (mid % 2 == 1 && vec[mid] == vec[mid - 1])) {
      left = mid + 1;
    } // if
    else {
      right = mid;
    } // else
  } // while
  return vec[left];
} // find_single_element()
```

11. To merge the intervals together, it would be helpful for the initial input to be sorted by the start time. Then, you would be able to iterate over the sorted input and use the end time of each value to determine if it should be merged with the most recent interval inserted into the solution. If the end time of the input interval being considered is larger than or equal to the end time of the most recent interval in the solution, then the solution interval may be extended. Otherwise, there is no overlap, and the new interval can be pushed into the solution. An implementation of this solution is shown below:

```
struct Interval {
  int32_t start;
  int32_t end;
};

std::vector<Interval> merge_intervals(const std::vector<Interval>& vec) {
  std::vector<Interval> result;
  if (vec.empty()) {
    return result;
  } // if
  std::vector<Interval> sorted_input{vec};
  std::sort(sorted_input.begin(), sorted_input.end(), [](const Interval& a, const Interval& b) {
    return a.start < b.start;  // sorts input by start time, can also use comparator instead
  });
  result.push_back(sorted_input.front());
  for (size_t i = 1; i < sorted_input.size(); ++i) {
    if (result.back().end < sorted_input[i].start) {
      result.push_back(sorted_input[i]);
    } // if
    else {
      result.back().end == std::max(result.back().end, sorted_input[i].end);
    } // else
  } // for i
  return result;
} // merge_intervals()
```

12. This problem can be solved with a linear-time loop over the input, where we move non-zero elements to the beginning of the array. While looping over the input, we will keep track of another index, `shifted_index`, which represents the next available position that a non-zero element can be moved to — every time we move a value, we increment this shifted index. Then, after we have moved all the non-zero elements over, we write a `0` value to all positions from `shifted_index` to the end of the array. Note that we are able to overwrite elements while iterating over the input because once we move past an element during our iteration, we no longer need to reference it again. An implementation of this solution is shown below:

```
void shift_zeros(std::vector<int32_t>& vec) {
  size_t shifted_index = 0;
  for (size_t i = 0; i < vec.size(); ++i) {
    if (vec[i] != 0) {
      vec[shifted_index++] = vec[i];
    } // if
  } // for
  while (shifted_index < vec.size()) {
    vec[shifted_index++] = 0;
  } // while
} // shift_zeros()
```

13. If you try to look through all the values in the matrix to search for the target value, then your worst-case time complexity would be $\Theta(mn)$ since there are mn values in the matrix. Instead, we can use the sorted nature of the matrix to reduce the number of elements we have to check to at most $m+n$. This can be done by starting at the bottom left or top right corner of the matrix and comparing its value with the target value to determine which direction to search. For example, if we start from the top right, and the target value is smaller, we continue the search by looking at the value to the left (since this value is smaller); if the target value is larger, we look at the value below (since this value is larger) — if we start from the bottom left instead, we would look up and to the right, respectively. If our search cannot continue because we have reached the edge of the matrix, then the value must not be found. For example, when searching for 10 in this matrix starting from the top right corner, our search path would be $15 \rightarrow 11 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 14 \rightarrow 13 \rightarrow 10$.

```
[ [ 1,  4,  7, 11, 15],
  [ 2,  5,  8, 12, 19],
  [ 3,  6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]  ]
```

Note that we cannot start our search from the top left or bottom right corners, since these are the smallest and largest values in the matrix, and thus comparing this with the target value can help use conclude whether the target exists in the matrix, but not where to continue the search if it does. An implementation of this solution is shown below:

```
1  bool matrix_search(const std::vector<std::vector<int32_t>>& matrix, int32_t target) {
2    if (matrix.empty()) {
3      return false;
4    } // if
5    int32_t curr_row = 0, curr_col = matrix[0].size() - 1;
6    while (curr_row < matrix.size() && curr_col >= 0) {
7      if (matrix[curr_row][curr_col] == target) {
8        return true;
9      } // if
10     else if (matrix[curr_row][curr_col] > target) {
11       --curr_col;
12     } // else if
13     else {
14       ++curr_row;
15     } // else
16   } // while
17   return false;
18 } // matrix_search()
```

14. This is another problem that can be solved with binary search, since there is information that allows us to get rid of half the search space with every value we check in the array. Consider the input provided in the example:

3	4	5	1	2
0	1	2	3	4

If the value we are checking is larger than the value at the last position of our search space, then the point of rotation must be to the right; otherwise, if a value is smaller than the value at the last position of our search space, then the point of rotation must be to the left. In the example, the value 4 is smaller than the last value in the array, 2, so the point of rotation must be to the right of the value 4. A binary search solution is provided below:

```
1  int32_t find_rotated_minimum(const std::vector<int32_t>& vec) {
2    int32_t left = 0, right = vec.size() - 1;
3    while (left < right) {
4      int32_t mid = left + (right - left) / 2;
5      if (vec[mid] > vec[right]) {
6        start = mid + 1;
7      } // if
8      else {
9        right = mid;
10     } // else
11   } // while
12   return vec[left];
13 } // find_rotated_minimum()
```

15. This is similar to the solution for question 7 — we can sort the vector and use the two pointer approach to determine which pair of elements sums closest to `target` (moving the pointers to adjust whether the target is larger or smaller than the current sum). The main difference is that you are no longer guaranteed that there are two values that sum to the target value; this can be addressed by keeping tracking of the best solution encountered so far while running the algorithm. An implementation of this solution is shown below:

```
std::pair<int32_t, int32_t> closest_sum_to_k(const std::vector<int32_t>& vec, int32_t k) {
  std::vector<int32_t> sorted_input{vec};
  std::sort(sorted_input.begin(), sorted_input.end());

  std::pair<int32_t, int32_t> idx;
  int32_t left = 0, right = vec.size() - 1, best = std::numeric_limits<int32_t>::max();
  while (left < right) {
    int32_t curr = std::fabs(sorted_input[left] + sorted_input[right] - k);
    if (curr < best) {
      idx.first = left;
      idx.second = right;
      best = curr;
    } // if
    if (sorted_input[left] + sorted_input[right] > k) {
      --right;
    } // if
    else {
      ++left;
    } // else
  } // while
  return std::make_pair(sorted_input[idx.first], sorted_input[idx.second]);
} // closest_sum_to_k()
```

16. This problem can be efficiently solved using the sliding window approach, and most similar matches the solution to the shortest substring problem demonstrated in example 15.8. Instead of determining whether the window contains every character in a target string (as with this example), we will want to determine whether the window sums to a value larger than the target and keep track of the shortest window encountered that satisfies this condition. We will begin our sliding window from the beginning of the array and extend it rightward as long as the sum is less than the target value. Once the sum is greater than or equal to the target value, we compare it with the best solution we have encountered so far to determine if it could be a viable solution, and then we contract the sliding window by removing values on the left. This is done until the window sum goes below the target value again. After repeating this until the window reaches the end of the input array, the best solution encountered must also be the solution to the entire problem. An implementation of this is shown below:

```
int32_t min_subarray_length(const std::vector<int32_t>& vec, int32_t target) {
  int32_t left = 0, window_sum = 0, min_len = std::numeric_limits<int32_t>::max();
  for (int32_t right = 0; right < vec.size(); ++right) {
    window_sum += vec[right];
    while (window_sum >= target) {
      min_len = std::min(min_len, right - left + 1);
      window_sum -= vec[left];
      ++left;
    } // while
  } // while
  return min_len;
} // min_subarray_length()
```