Chapter 15
Binary Search and Additional Algorithms

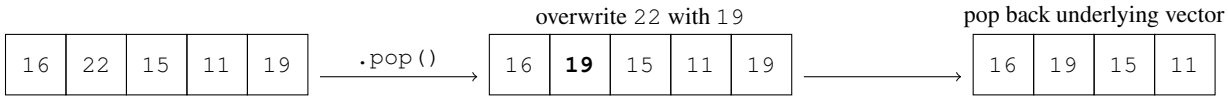
15.1 Ordered and Sorted Containers

Now that we have covered several different data structures, we will introduce a new way of categorizing different container types based on their behavior. These new categorizations include *ordered* and *sorted* containers.

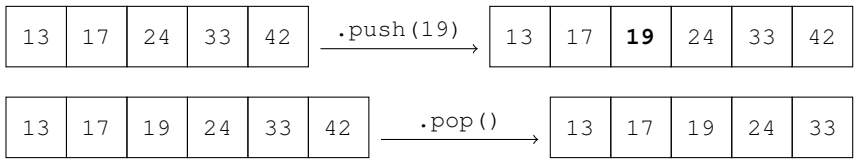
We define an **ordered container** as a container that maintains its current ordering of elements, regardless of how the container is modified. Elements in an ordered container must maintain their relative positions unless they are removed. For example, if element *A* comes before element *B* in an ordered container, and we insert element *C* into the container, then *A* would still be before *B* after the insertion. Similarly, if *C* were then removed, *A* would still come before *B* in the container. As long as *A* and *B* are in the ordered container, *A* will always be positioned before *B* in the container's ordering, irrespective of other elements that may be inserted or removed. In the STL, ordered containers typically support an insertion method that accepts an iterator, which defines the position at which the new element should be inserted at.

On the other hand, we define a **sorted container** as a sequential container whose elements are always in some predefined order. In a sorted container, you cannot insert elements wherever you want in the container; a newly inserted element's position must adhere to this predefined ordering designated by a sorted container. For example, if you have a sorted container that contains the values 1 and 3, and then you insert 2 into this container, then 2 must be inserted in between 1 and 3 (assuming the sorted container sorts its values in ascending order).

So far, we have seen several examples of ordered and sorted containers. Linked lists, arrays, and deques are good examples of ordered containers on their own, since the insertion or deletion of elements from these containers do not change the relative ordering of other elements. On their own, these containers are also not sorted by default, as there are no rules that dictate the order in which values must be positioned. However, these containers can also be used as the underlying container of other data structures that may be unordered or sorted. For example, the unordered sequence container used to implement a priority queue in section 10.2.2 is an unordered container, since popping off an element may change the relative ordering of other elements in the container:



Similarly, the sorted sequence container in section 10.2.3 implements a priority queue using an underlying array whose elements are required to follow a predefined order. In this case, we consider this underlying array as a sorted container.



As another example, the set implementation introduced in section 13.1.2 uses a sorted array as its underlying structure, as this invariant allowed us to perform set operations efficiently. Therefore, we would also categorize this set implementation as a sorted container.

When implementing an ordered or sorted container, arrays, linked lists, and deques are typically good candidates for storing the container's underlying data. The ideal selection of container depends on the requirements of the application that uses it, and knowledge of operations that are called frequently can aid in this decision (for example, if you need something that relies on random access, a list will not be the way to go). The table below summarizes the worst-case complexities of operations on an *ordered* container, using different underlying containers to store its data (review chapters 6-9 if you are not sure where these complexities come from).

Operation	Array	Linked List	Deque
`add_value(val)` *(inserts value anywhere in container)*	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`remove(val)` *(remove value from container, but you must find it first)*	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`remove(iterator)` *(remove value from container, but you are given its iterator)*	$\Theta(n)$	$\Theta(n)$ if singly-linked $\Theta(1)$ if doubly-linked	$\Theta(n)$
`find(value)` *(find a value in the container)*	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`iterator::operator*()` *(dereference an iterator)*	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`operator[](index)` *(access element with a position at index)*	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
`insert_after(iterator, val)` *(insert an element after a provided iterator)*	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
`insert_before(iterator, val)` *(insert an element before a provided iterator)*	$\Theta(n)$	$\Theta(n)$ if singly-linked $\Theta(1)$ if doubly-linked	$\Theta(n)$

The following table summarizes the worst-case complexities of a *sorted* container, with each container type holding is underlying data.

Operation	Array	Linked List	Deque
`add_value(val)` *(inserts value anywhere in container)*	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`remove(val)` *(remove value from container, but you must find it first)*	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`remove(iterator)` *(remove value from container, but you are given its iterator)*	$\Theta(n)$	$\Theta(n)$ if singly-linked $\Theta(1)$ if doubly-linked	$\Theta(n)$
`find(value)` *(find a value in the container)*	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$
`iterator::operator*()` *(dereference an iterator)*	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`operator[](index)` *(access element with a position at index)*	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
`insert_after(iterator, val)` *(insert an element after a provided iterator)*	N/A	N/A	N/A
`insert_before(iterator, val)` *(insert an element before a provided iterator)*	N/A	N/A	N/A

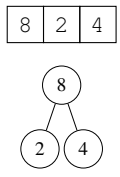
There are a few notable differences in these complexities between ordered and sorted containers. The time complexity of adding a value in a sorted container becomes $\Theta(n)$, since each new element must be inserted in a position that adheres to the correct sorted order (which takes linear time for a list since you cannot use binary search, and linear time for vectors and deques since elements may need to be shifted after the insertion). The time complexity of finding an element improves to $\Theta(\log(n))$ for the random-access containers since we can now use binary search, but remains $\Theta(n)$ for a list since they do not provide random access. Lastly, inserting before and after a given iterator is no longer supported, since you do not have the freedom to insert values wherever you want in a sorted container.

Example 15.1 Is the underlying data vector used to implement a binary heap sorted? Is it ordered? If not, provide an example.

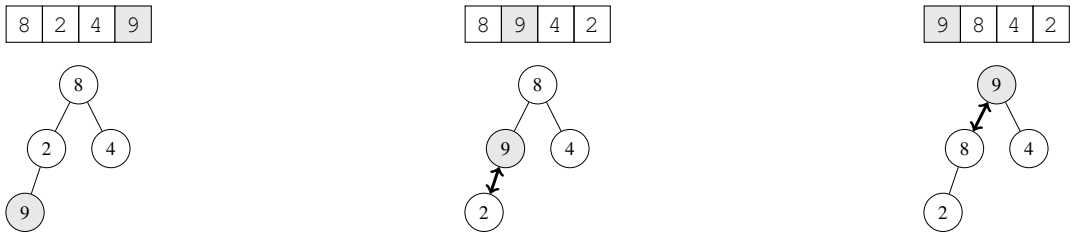
The underlying data vector used to implement a binary heap is neither sorted nor ordered. This may seem counterintuitive, since a binary heap certainly seems like a container that is sorted by priority. However, if you recall the definition of a binary heap, the relationship between nodes of the heap only requires no parent to have a lower priority than any of its descendants. Because of this, there are multiple ways to represent the same elements in the underlying container of a binary heap.

Both of these variations are valid structures for a min-binary heap, and we can clearly see that the data in the underlying container does not need to be in a specific, predefined order. Thus, the underlying vector used to implement a binary heap is not a sorted container.

The underlying vector in a binary heap is also not an ordered container. To illustrate why this is the case, consider the following array representation of a max-binary heap:



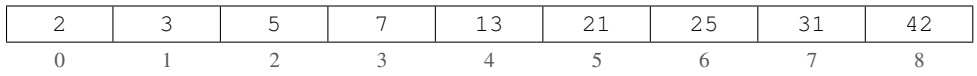
Suppose we insert 9 into this binary heap. 9 would get inserted to the back of the heap, and then fixed up to its correct position.



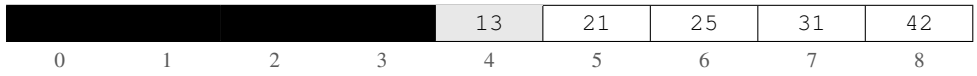
Notice that the addition of 9 to this binary heap changed the relative ordering of 2 and 4! Prior to insertion, 2 was before 4 in the underlying vector; however, after 9 was added, 2 is after 4. This shows that the binary heap's underlying data vector is not an ordered container.

15.2 Binary Search

As mentioned earlier, searching in a sorted container is much more efficient if the container is able to support binary search. **Binary search** is a searching algorithm that can be used to find a value in logarithmic time if the underlying data is sorted. During the binary search process, we first look at the middle element and compare it with the value we are trying to find. If the middle element is larger than the element we want to find, then we know that the our target element must be to the left of the middle value. On the other hand, if the middle element is smaller than the element we want to find, then we know that our target element must be to the right of the middle value. We are then able to repeat this process until we find (or fail to find) our target value, eliminating half the search space every time. For example, suppose we wanted to search for 21 in the following sorted vector:



We will first look at the element in the middle, 13. Since 21 is greater than 13, we know that 21 (if it exists), must be to the right of 13. As a result, we can eliminate all the elements to the left of 13 as possible solutions.



Next, we would look at the middle value of the remaining elements, or 25 (integer division chooses the value on the left if there are two middle values). Since 21 is less than 25, it must be to the left of 25 (if it exists). Thus, we can eliminate all the elements to the right of 25.

We then look at the remaining element to the left of 25. It is equal to our target of 21, so the value has been found.

The following code implements binary search. It returns the index of the target element if found, and `-1` otherwise:

```
1  int32_t binary_search(const std::vector<int32_t>& vec, int32_t target, int32_t left, int32_t right) {
2    while (right > left) {
3      int mid = left + (right - left) / 2;
4      if (target == vec[mid]) {
5        return mid;
6      } // if
7      else if (target < vec[mid]) {
8        right = mid;
9      } // else if
10     else {
11       left = mid + 1;
12     } // else
13   } // while
14   return -1;  // target not found
15 } // binary_search()
```

Here, `target` represents the value you want to find, and `[left, right)` represents the range you want to search in. At the beginning, `left` should be 0 and `right` should be `size` if you want to search the entire range. The `while` loop on line 2 runs as long as there are elements that still need to be searched. On line 3, we calculate the middle index of our range. Note that we did not use `(left + right)/2`; this is because adding `left` and `right` could lead to overflow if our indices are very large. The if-else statements from line 4-13 deal with the comparisons; since half of the search space is eliminated at each iteration, the complexity of binary search is $\Theta(\log(n))$ for a search space of size n.

However, this code is not entirely optimal, and there are ways we can speed it up. First, observe that the == on line 4 rarely returns `true`, since it is more likely for the target to be less than or greater than the first element you check. Thus, it would be more efficient to check < and > before checking ==, as that could potentially save you a few comparison checks per iteration. This change is made below:

```
1  int32_t binary_search(const std::vector<int32_t>& vec, int32_t target, int32_t left, int32_t right) {
2    while (right > left) {
3      int32_t mid = left + (right - left) / 2;
4      if (target < vec[mid]) {
5        right = mid;
6      } // if
7      else if (target > vec[mid]) {
8        left = mid + 1;
9      } // else if
10     else {
11       return mid;
12     } // else
13   } // while
14   return -1;  // target not found
15 } // binary_search()
```

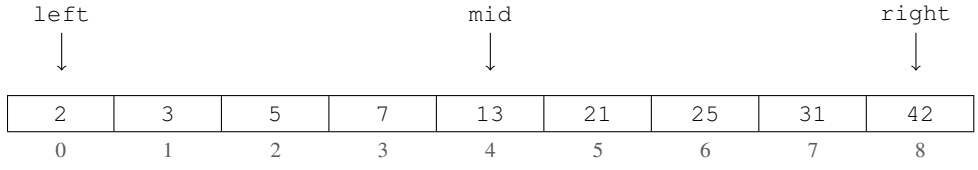
Binary search can also be implemented recursively. In the recursive approach, a recursive call is made on the right half if the middle element is smaller than the target value, and a recursive call is made on the left half if the middle element is larger than the target value. The code for a recursive binary search approach is shown below:

```
1  int32_t binary_search(const std::vector<int32_t>& vec, int32_t target, int32_t left, int32_t right) {
2    if (left > right) {  // base case, failed to find value
3      return -1;
4    } // if
5    int32_t mid = left + (right - left) / 2;
6    if (vec[mid] < target) {
7      return binary_search(vec, target, mid + 1, right);
8    } // if
9    else if (vec[mid] > target) {
10     return binary_search(vec, target, left, mid - 1);
11   } // else if
12   else {
13     return mid;
14   } // else
15 } // binary_search()
```

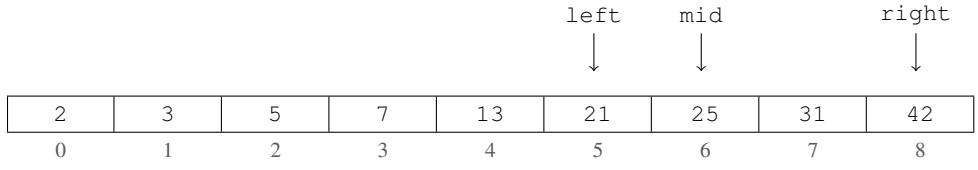
A similar approach to binary search would be finding the *lower bound* of a target element. When calculating a lower bound, you aren't finding if an element actually exists in a range; instead, you are finding the index that an element should be found at if it did exist. A possible implementation for the lower bound operation is shown below; since there is no need to check for equality, the algorithm only makes two comparisons per loop (rather than three, as was the case before).

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

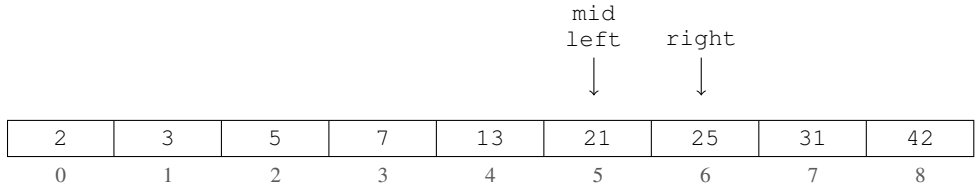
For example, suppose we wanted to find the lower bound of 20 in the following sorted vector:



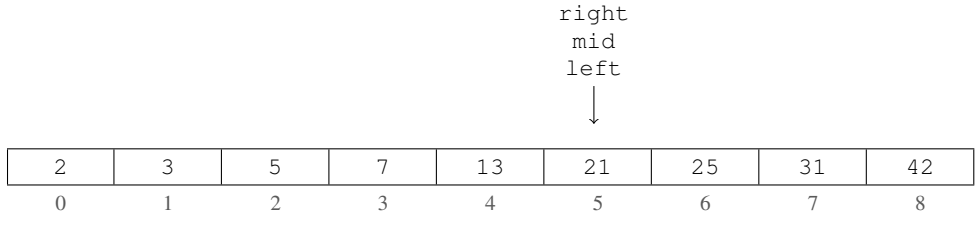
We will first look at the element in the middle, 13. Since 20 is greater than 13, we update `left` to the position of 21.



Now, `mid` refers to 25, which is larger than 20. Thus, we would set `right` to the position of 25.



The value of `mid` is now 21, which is still larger than 20. Thus, `right` is updated to the position of 21.

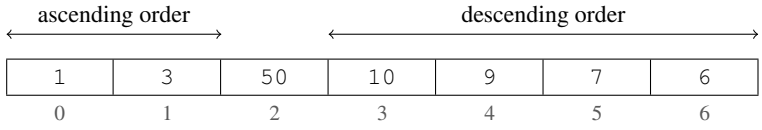


Both `left` and `right` refer to the same element, so the `while` loop on line 2 exits, and `left` is returned (in this case, index 5). Even though 20 itself does not exist in the sorted array, the position of `left` is the index that 20 would be located at if it did exist in the array. This can be useful if you want to insert an element into a sorted container, as the lower bound operation would give you the correct position of insertion.

The binary search algorithm is a classic topic for many interview problems. We will discuss several examples below:

Example 15.2 You are given an array that is first increasing and then decreasing. Find the maximum value in this array. For example, given the array `[1, 3, 50, 10, 9, 7, 6]`, you would return `50`.

Notice that the maximum value splits the array into two segments with unique properties. All of the elements to the left of the maximum value are in ascending order, while all of the elements to the right of the maximum value are in descending order. This allows us to easily determine whether an element is to the left or right of the maximum value, making binary search an efficient algorithm for solving this problem.



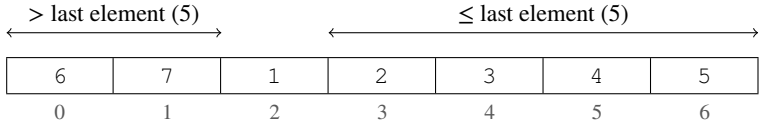
First, we would look at the middle element and compare it with the elements that are adjacent to it. If the middle element is larger than both adjacent elements, it must be the maximum, and we return this value. If the middle element is greater than the element to its left and smaller than the element to its right, we are on the ascending portion of the array, and the maximum value must be to the right of the middle value (this allows us to eliminate all elements to the left immediately). Similarly, if the middle element is smaller than the element to its left and greater than the element to its right, we are on the descending portion of the array, and the maximum value must be to the left of the middle value (this allows us to eliminate all elements to the right immediately). The code for this algorithm is shown below:

```
1   int32_t find_max_value(const std::vector<int32_t>& vec, int left, int right) {
2     int32_t start = left, end = right - 1;
3     while (start < end) {
4       int32_t mid = start + (end - start) / 2;
5       // if this condition is true, the target must be to the right of mid
6       if (vec[mid] < vec[mid + 1] && vec[mid] > vec[mid - 1]) {
7         start = mid + 1;
8       } // if
9       // otherwise, the target must be to the left of mid
10      else {
11        end = mid;
12      } // else
13    } // while
14    return vec[start];
15  } // find_max_value()
```

Since we are simply performing a binary search, the time complexity of this solution is $\Theta(\log(n))$, where n is the size of the vector. The auxiliary space of this solution is $\Theta(1)$, since the amount of memory allocated is not dependent on the input size.

Example 15.3 You are given a sorted array that is rotated at a certain position. Return the minimum element in this array. For example, given the array `[6, 7, 1, 2, 3, 4, 5]`, you would return `1`.

Similar to the previous example, the minimum element can be used to partition the array into two segments that have different properties. All the elements to the left of the minimum value must be larger than the element at the back of the rotated array, and all the elements to the right of the minimum value must be less than or equal to the element at the back of the array. We can use this information to conduct a binary search.



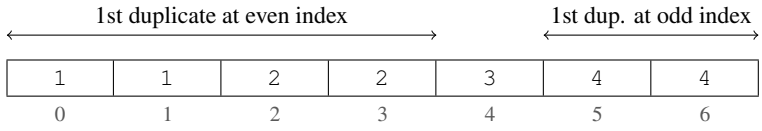
First, we look at the element in the middle of the array and compare it with the last element. If this middle element is smaller than the last element, we can eliminate every element to the right of the middle element since they cannot be the minimum. If this middle element is larger than the last element, we can similarly eliminate every element to the left of the middle element. The code is shown below:

```
1   int32_t find_min_rotated(const std::vector<int32_t>& vec, int32_t left, int32_t right) {
2     int32_t start = left,  end = right - 1;
3     while (start < end) {
4       int32_t mid = start + (end - start) / 2;
5       // if this condition is true, the target must be to the right of mid
6       if (vec[mid] > vec[end]) {
7         start = mid + 1;
8       } // if
9       // otherwise, the target must be to the left of mid
10      else {
11        end = mid;
12      } // else
13    } // while
14    return vec[start];
15  } // find_min_rotated()
```

Similar to the previous problem, the time complexity of this solution is $\Theta(\log(n))$, and the auxiliary space usage is $\Theta(1)$.

Example 15.4 You are given a sorted array where every number occurs twice except for one number. Return the number that only appears once. For example, given the array `[1, 1, 2, 2, 3, 4, 4]`, you would return `3`.

Once again, the element that appears once can be used to partition elements to the left and right into two groups with different properties. This may be a bit tricky to notice at first, but consider the following array:



Notice that if a pair occurs before the single element, the first element of the pair has an even index (e.g., the first `1` and `2` are located at indices 0 and 2). However, if a pair occurs after the single element, the first element of the pair would have an odd index (e.g., the first `4` is located at index 5). We can use this information to conduct a binary search.

First, we look at the middle index. If the middle index is even, we check if the element at `mid` and `mid + 1` are the same. If the middle index is odd, we check if the element at `mid` and `mid - 1` are the same. If this is true, the single element must be to the right; otherwise, it must be to the left. The code is shown below:

```
1  int32_t find_single_element(const std::vector<int32_t>& vec, int32_t left, int32_t right) {
2    int32_t start = left,  end = right - 1;
3    while (start < end) {
4      int mid = start + (end - start) / 2;
5      // if this condition is true, the target must be to the right of mid
6      if ((mid % 2 == 0 /* even */ && vec[mid] == vec[mid + 1]) ||
7          (mid % 2 == 1 /* odd  */ && vec[mid] == vec[mid - 1])) {
8        start = mid + 1;
9      } // if
10     // otherwise, the target must be to the left of mid
11     else {
12       end = mid;
13     } // else
14   } // while
15   return vec[start];
16 } // find_single_element()
```

Similar to the previous problems, the time complexity of this solution is $\Theta(\log(n))$, and the auxiliary space usage is $\Theta(1)$.

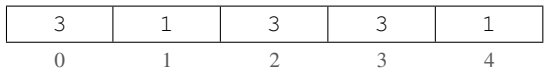
15.3 Moore's Voting Algorithm (*)

Suppose you are given an array of size n, and you are told that one element in the array is repeated over $n/2$ times (i.e., it is a majority element). How can you identify what this majority element is? One algorithm that you can use is **Moore's voting algorithm**, which can be used to determine the majority element in a container, if there is one, in $\Theta(n)$ time.

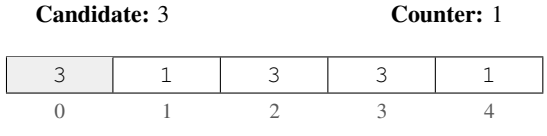
When running this algorithm, you will need to keep track of a counter and a "candidate" element that could potentially be the majority element. The counter is initialized to zero. Then, begin a traversal of the array using the following steps:

1. Initialize the candidate to the first element and the counter to 1.
2. Continue traversing the array. If an element is equal to the candidate value, increment the counter.
3. If an element is not equal to the candidate value, decrement the counter.
4. If the counter ever hits 0, update the candidate value to the element currently being visited, and set the counter back to 1.
5. Repeat steps 2-4 until the entire array is processed. The "candidate" element that remains at the end of the traversal is guaranteed to be the majority element if a majority element exists.
6. If a majority element is not guaranteed to exist, traverse the array again, but this time check to see if the candidate actually appears $n/2$ times. This is done to ensure that the candidate is indeed a majority element.

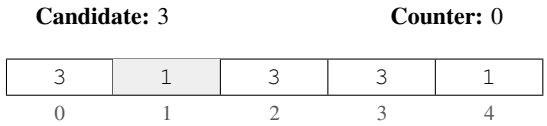
Let's explain this algorithm using an example. Consider the following array, where 3 is the majority element:



When traversing the array, the first element we encounter is 3, which is equal to our current candidate value. Thus, we would increment the counter to 1.

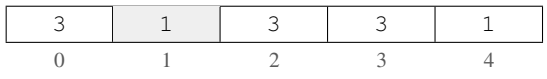


Next, we visit element 1. Since 1 is not equal to our candidate, we decrement the counter.



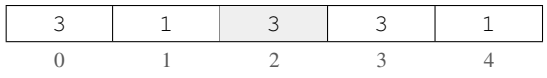
Notice that our counter is now 0. When this happens, we update the candidate to the current value we are on, or 1. The counter gets reset to 1.

Candidate: 1 **Counter:** 1



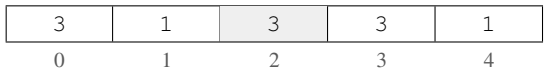
The next element is 3. Since 3 is not equal to our candidate, we decrement the counter.

Candidate: 1 **Counter:** 0



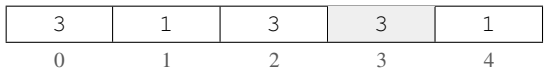
The counter is zero, so we will need to update our candidate. The candidate becomes 3, and the counter gets set to 1.

Candidate: 3 **Counter:** 1



The next element is 3, which is equal to our candidate. Thus, we increment the counter.

Candidate: 3 **Counter:** 2

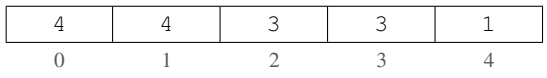


The next element is 1, which is not equal to our candidate. Thus, we decrement the counter.

Candidate: 3 **Counter:** 1

We are done traversing the array, and our candidate is 3. Therefore, we can conclude that if a majority element exists in the array, then it must be 3. We must do another traversal to make sure that 3 appears more than $\lfloor 5/2 \rfloor = 2$ times. This additional traversal is needed in case there is no majority element. To see why this additional traversal is important, consider the following:

Candidate: 4 **Counter:** 0



If we follow the rules above, we would obtain the following after the traversal:

Candidate: 1 **Counter:** 1

However, it is clear that 1 is not our majority element! In fact, there is no majority element, since no element appears more than 2 times. This is why the additional traversal at the end is needed, as we need to ensure our candidate is actually a majority element.

The correctness of Moore's voting algorithm is actually quite intuitive. Each element "votes" for itself as the candidate, and opposing votes cancel each other out. If there exists a majority element, then the candidate that remains after the traversal must have had enough votes to not be negated by the other elements. The code for this algorithm is shown on the next page.

The code for Moore's voting algorithm is shown below, to identify a majority element. `-1` is returned if no majority element exists.

```
int32_t majority_element(const std::vector<int32_t>& vec) {
  int32_t counter = 0, candidate;
  for (int32_t val : vec) {
    if (counter == 0) {
      candidate = val;
    } // if
    counter += (val == candidate) ? 1 : -1;
  } // for val
  int32_t threshold = vec.size() / 2;
  counter = 0;
  for (int32_t val : vec) {
    if (val == candidate) {
      ++counter;
    } // if
  } // for val
  if (counter > threshold) {
    return candidate;
  } // if
  return -1;
} // majority_element()

int main() {
  std::vector<int32_t> vec = {3, 1, 3, 3, 1};
  int32_t majority = majority_element(vec);
  if (majority != -1) {
    std::cout << "The majority element is " << majority << '\n';
  } // if
  else {
    std::cout << "There is no majority element.\n";
  } // else
} // main()
```

Remark: In the above code, `-1` is returned if there is no majority element. However, what if `-1` were actually the majority element? This leads to an interesting conundrum: we need to return a value that indicates that the function failed to find a majority value, but we also need to ensure that the value we return upon failure cannot be the majority element itself! This would be a good place to use `std::optional<>`, which is an object that manages an optional value. That way, the function would only need to return an integer if a majority element exists, and we can check the contents of the `std::optional<>` to determine the existence of the majority element. To use `std::optional<>`, you must `#include <optional>`. The modified code is shown below:

```
std::optional<int32_t> majority_element(const std::vector<int32_t>& vec) {
  int32_t counter = 0, candidate;
  for (int32_t val : vec) {
    if (counter == 0) {
      candidate = val;
    } // if
    counter += (val == candidate) ? 1 : -1;
  } // for
  int32_t threshold = vec.size() / 2;
  counter = 0;
  for (int32_t val : vec) {
    if (val == candidate) {
      ++counter;
    } // if
  } // for val
  if (counter > threshold) {
    return candidate;
  } // if
  return std::nullopt;  // return optional with no value if no majority element
} // majority_element()

int main() {
  std::vector<int32_t> vec = {4, 4, 3, 3, 1};
  std::optional<int32_t> majority = majority_element(vec);
  if (majority.has_value()) {  // if a majority element exists
    std::cout << "The majority element is " << *majority << '\n';
  } // if
  else {
    std::cout << "There is no majority element.\n";
  } // else
} // main()
```

For more on `std::optional<>`, review section 11.13.2.

15.4 Dutch National Flag Algorithm (*)

The **Dutch national flag algorithm** can be used to sort a container in linear time using no auxiliary space, provided that the container only contains elements that take on three distinct values (named after the Dutch flag, with three colored stripes). For example, suppose you are given an array filled with 0s, 1s, and 2s, and you want to sort the array in linear time:

```
[2, 1, 0, 1, 2, 0]
```

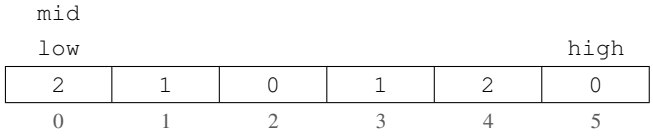
The goal behind the Dutch national flag algorithm is to swap all the 0s to the left end of the array, and swap all of the 2s to the right end of the array. At the very end, all the 1s will be left in the middle, leaving the array fully sorted. The steps of the algorithm are as follows:

1. Create a `low` index that references the first element in the array, and a `high` index that references the last element in the array.
2. Create a `mid` index that starts from the beginning of the array and iterates through each element. The algorithm stops once `mid` passes `high`. During the iteration, the following steps are followed based on the value at index `mid`.
   - If `vec[mid]` is 0, swap `vec[mid]` with `vec[low]` and increment both `low` and `mid` by 1.
   - If `vec[mid]` is 1, don't swap anything and increment `mid` by 1.
   - If `vec[mid]` is 2, swap `vec[mid]` with `vec[high]` and decrement `high` by 1.

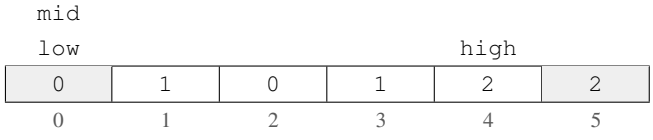
The code for the Dutch national flag algorithm is shown below, for a vector containing only 0s, 1s, and 2s:

```
1  void dutch_national_flag(std::vector<int32_t>& vec) {
2    size_t low = 0, high = vec.size() - 1;
3    for (size_t mid = 0; mid <= high; ) {
4      if (vec[mid] == 0) {
5        std::swap(vec[mid++], vec[low++]);
6      } // if
7      else if (vec[mid] == 2) {
8        std::swap(vec[mid], vec[high--]);
9      } // else if
10     else {
11       ++mid;
12     } // else
13   } // for mid
14 } // dutch_national_flag()
```

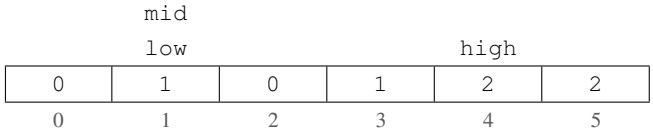
Let's go through this algorithm using the example above:



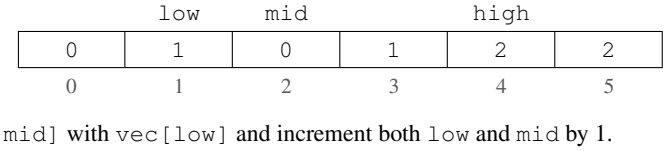
`vec[mid]` is 2, so we swap `vec[mid]` with `vec[high]` and decrement `high` by 1.



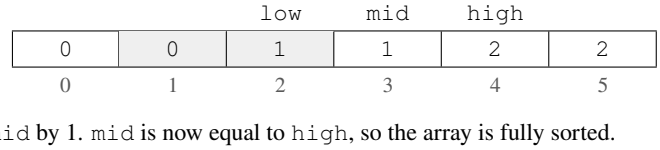
`vec[mid]` is 0, so we swap `vec[mid]` with `vec[low]` and increment both `low` and `mid` by 1. Nothing is swapped here since `low` and `mid` refer to the same index.



`vec[mid]` is 1, so we increment `mid` by 1.



`vec[mid]` is 0, so we swap `vec[mid]` with `vec[low]` and increment both `low` and `mid` by 1.



`vec[mid]` is 1, so we increment `mid` by 1. `mid` is now equal to `high`, so the array is fully sorted.

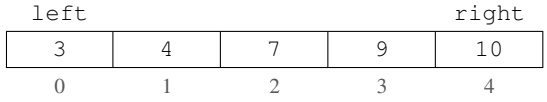
It is important to note that `mid` is incremented if `vec[mid]` is 0 or 1, but it is *not* incremented if `vec[mid]` is 2. To see why, consider the array `[1, 2, 0]`. If we increment `mid` in the `vec[mid] == 2` case, we would swap 0 and 2, but we would never be able to swap 0 with 1. As a result, our final array would be `[1, 0, 2]`, which is not sorted. Thus, to prevent `mid` from exceeding `high` before the array is fully sorted, we do not increment `mid` if `vec[mid]` is 2.

15.5 Two Pointer Technique (⁎)

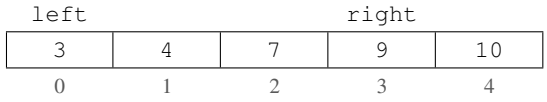
The **two pointer technique** is an algorithmic technique that can be used to solve several different types of programming problems. With the two pointer technique, we start with two pointers (or indices): one that references the first element and one that references the last element. We then move the two pointers toward each other, using the values they reference at each step to help us solve the problem. The Dutch national flag algorithm covered earlier is an example of this technique. Two additional examples of this algorithmic approach are provided below.

Example 15.5 You are given an array of positive integers that is sorted in ascending order. Write an algorithm that, when given this array and a target number, returns a pair of two numbers in the array that sums to the target number. For example, given the array `[3, 4, 7, 9, 10]` and a target of `11`, you would return `[4, 7]`. Return `[-1, -1]` if there is no way to sum to the target number.

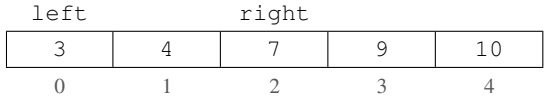
A naïve solution would be to sum up every possible pair in the array and check it against the target value, which would take $\Theta(n^2)$ time. However, since we know that the underlying array is sorted, we can use the two pointer technique to reduce the runtime to $\Theta(n)$. The idea is as follows. First, we initialize two indices, `left` and `right`, that refer to the first and last element in the vector.



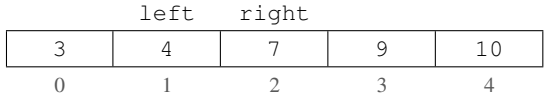
What happens when we add the values at `left` and `right`? We get a sum of `13`, which is not the target value we want. However, this actually gives us valuable information — we know that our sum is too large. To decrease the sum, we will have to decrement the `right` index.



Now, if we add the values at `left` and `right`, we get a sum of `12`. This is still too large, so we decrement the `right` index by one.



Now, the sum is `10`, which is too small. The only way to increase the sum is to increment `left`. Note that both indices only move toward the middle of the array. This ensures that every value is only visited once and guarantees a $\Theta(n)$ runtime.

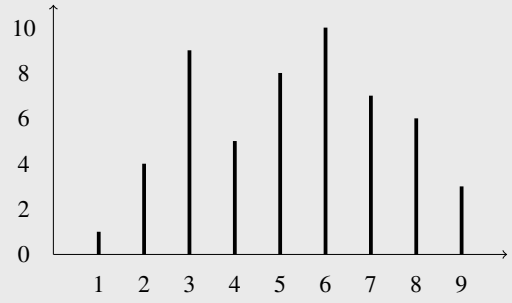


The sum of `left` and `right` is now equal to our target value, `11`, so we can return this pair. Notice that if `left` and `right` ever meet, there must exist no pair that sums to the given target. The code for this solution is shown below:

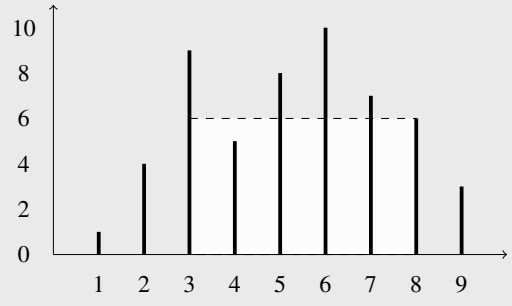
```
1  std::pair<int32_t, int32_t> target_sum(const std::vector<int32_t>& vec, int32_t target) {
2    size_t left = 0, right = vec.size() - 1;
3    while (left < right) {
4      if (vec[left] + vec[right] < target) {
5        ++left;
6      } // if
7      else if (vec[left] + vec[right] > target) {
8        --right;
9      } // else if
10     else {
11       return {vec[left], vec[right]};
12     } // else
13   } // while
14   return {-1, -1};  // no solution
15 } // target_sum()
```

Since we are moving the left and right pointers toward each other and terminating the algorithm once they cross, we only complete a single pass of the input vector, and the time complexity of this algorithm is $\Theta(n)$. The memory allocated by this solution does not depend on the input size, so the auxiliary space usage of this solution is $\Theta(1)$.

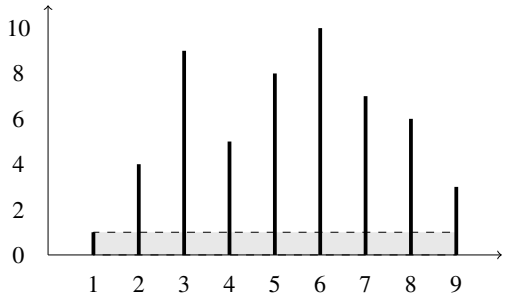
Example 15.6 You are given an array of n non-negative integers that represent the height of vertical bars on a map. For example, the array `[1, 4, 9, 5, 8, 10, 7, 6, 3]` represents the following map:



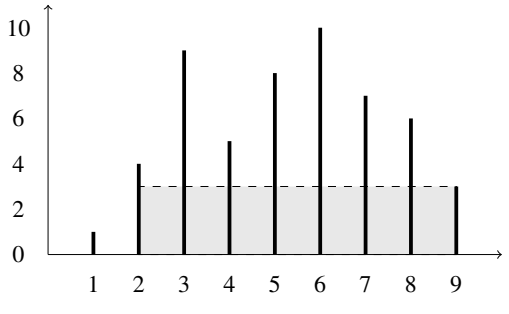
Each bar is a distance of 1 away from adjacent bars. Implement a function that returns the maximum amount of water that can be held by the bars on the map. The most water that can be held in the example above is 30:



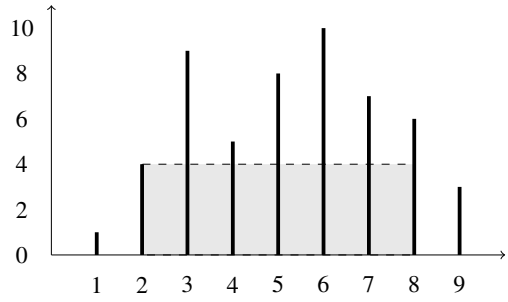
Like with the previous example, a naïve solution would be to check every possible pair of bars to determine the maximum amount of water that can be held. However, we can use the two pointer approach to reduce the complexity of this problem to linear time. To do so, we first start by considering both the leftmost and rightmost bars, as shown below. Here, the amount of water we can hold is $1 \times (9 - 1) = 8$.



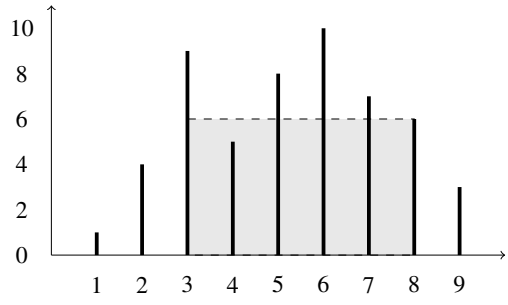
Next, we will move the two pointers (or indices) toward each other in order to discover potentially better solutions. Which one do we move first? Since we are trying to maximize the amount of water we can hold, we should move the pointer referencing the shorter bar. In fact, we should continue moving this pointer until we encounter a bar that is taller than the ones we have encountered before, as only a taller bar can potentially yield a better solution. In this case, we will increment the left pointer to the right by one.



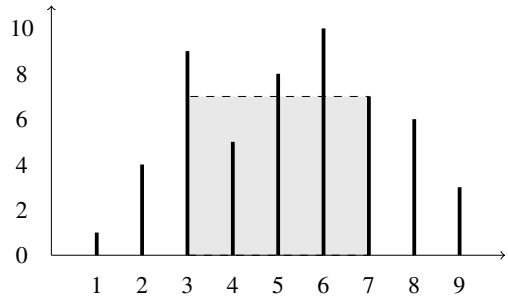
The amount of water we can hold is now $3 \times (9-2) = 21$. This is better than our previous solution of 8, so we will keep track of 21 as the largest value we have encountered so far. At this point, the bar on the right is now the smaller one under consideration, so we decrement its pointer until we encounter a bar that is higher.



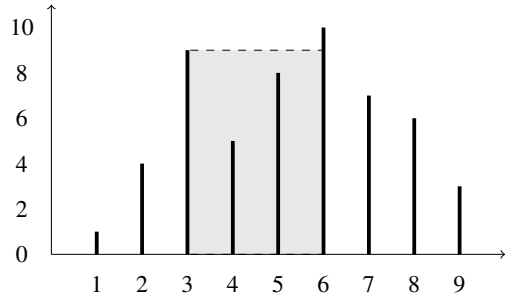
The amount of water we can hold is now $4 \times (8-2) = 24$, which is better than our previous solution of 21. 24 now becomes the largest value we have seen so far. Since the bar on the left is the smaller of the two, we then increment the left pointer until we encounter a bar that is higher.



The amount of water we can hold is now $6 \times (8-3) = 30$. This is better than 24, so 30 becomes the best we have seen so far. Since the bar on the right is shorter, we decrement its pointer until we encounter a bar that is taller.



The amount of water we can hold is now $7 \times (7-3) = 28$. This is not better than 30, so we ignore this result. The bar on the right is the shorter of the two, so we decrement its pointer until we encounter a taller bar.



The amount of water we can hold is now $9 \times (6-3) = 27$. This is not better than 30, so we ignore this result. The bar on the left is the shorter of the two, so we increment its pointer until we encounter a taller bar (note that this allows us to skip bars 4 and 5, since they cannot produce a better solution). This causes our left and right pointers to point to the same bar, which indicates that our algorithm is complete. Since 30 was the best value we encountered, it must be the solution for this example.

An implementation of this solution is shown below. This solution uses indices, but you can also use iterators to traverse the bars instead.

```
1  int32_t max_water(const std::vector<int32_t>& bars) {
2    size_t left = 0, right = bars.size() - 1;
3    int32_t best = 0;
4    while (left < right) {
5      int32_t min_height = std::min(bars[left], bars[right]);
6      best = std::max(best, min_height * (right - left));
7      while (bars[left] <= min_height && left < right) {
8        ++left;
9      } // while
10     while (bars[right] <= min_height && left < right) {
11       --right;
12     } // while
13   } // while
14   return best;
15 } // max_water()
```

Similar to the previous problem, we only complete a single pass of the input vector, so the time complexity of this solution is $\Theta(n)$. The memory allocated by this solution does not depend on input size, so the auxiliary space usage is $\Theta(1)$.

15.6 Sliding Window Technique (*)

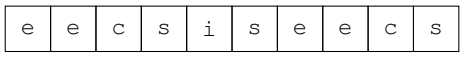
The **sliding window technique** is another technique that can be used to solve different programming problems. The sliding window technique considers individual subsets of data and expands or shrinks that subset based on the conditions of the problem, giving it the effect of "sliding" a window through the data. In many cases, the sliding window technique is a useful tool for solving problems where you are given an ordered and iterable container (like a vector or a string) and asked for information on a subrange of that container (such as the longest or shortest subrange that satisfies a given condition).[1] We will look at a few problems that can be solved using a sliding window approach below.

Example 15.7 Given a string `str`, find the length of the longest substring without any repeating characters. You may assume that the string you are given only contains the letters a-z, all in lowercase.

Example: Given the string `"eecsiseecs"`, return `4`, since the longest substring without any repeating letters (`"ecsi"`) has length 4.

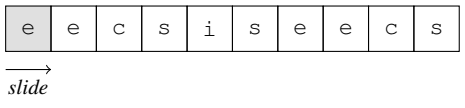
The simplest solution would be to generate all possible substrings of the given string, and among the substrings that have no duplicate letters, identify the one with the largest length. However, such an approach is clearly inefficient. Instead, since the problem deals with a contiguous subrange that needs to satisfy a given condition, we will approach it using the sliding window technique.

It turns out that the sliding window approach can indeed be used to solve this problem efficiently. In this solution, we start at the left edge of the string, increasing the size of our window one character at a time. Along the way, we keep track of which letters appear in our window, allowing us to immediately detect the presence of a duplicate letter. We also keep track of the length of the longest viable string we have encountered so far. If the right boundary of our window ever encounters a duplicate letter, we must increment the left boundary until the duplicate is gone. This process is illustrated in the example below.



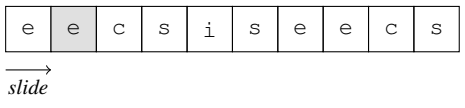
Best window size encountered: 0
Letters currently in window:

First, we add `e` to our sliding window. The best window size encountered becomes 1.



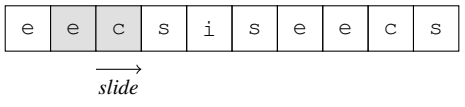
Best window size encountered: ***1***
Letters currently in window: ***e***

Next, we add the second `e` to our sliding window. However, `e` already exists in our window, so the addition of this new `e` would make our window contain a duplicate letter. To fix this, we would continuously increment the left boundary of our window until there is only one `e` remaining. In this case, the left boundary only needs to move forward by one position. Our new window is shown below:



Best window size encountered: 1
Letters currently in window: e

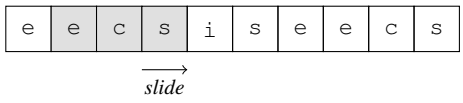
We then slide the right boundary of the window forward, adding in `c`. This is still a valid window without any duplicate letters, so we update the best window size encountered to 2 and add `c` to the collection of letters currently in our window.



Best window size encountered: ***2***
Letters currently in window: ***e, c***

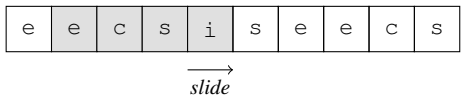
[1] This is just a rule of thumb, and there is no guarantee that a sliding window can be used to solve all problems that follow this structure. For certain problems, a more advanced algorithmic technique may be necessary, such as dynamic programming or divide-and-conquer (both of which will be covered later). However, this is still a good pattern to have in your algorithm toolbox, as it can help you come up with solutions to problems that can be solved using this technique.

We slide our window forward again, adding in `s`. There are no duplicate letters yet, so we update the best window encountered to 3 and add `s` to the collection of letters currently in our window.



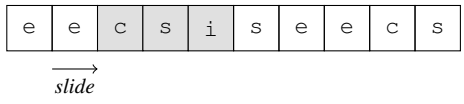
Best window size encountered: ***3***
Letters currently in window: ***e, c, s***

We slide our window forward again, adding in `i`. There are no duplicate letters yet, so we update the best window encountered to 4 and add `i` to the collection of letters currently in our window.

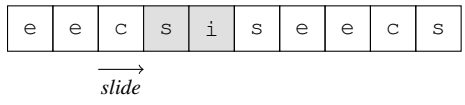


Best window size encountered: ***4***
Letters currently in window: ***e, c, s, i***

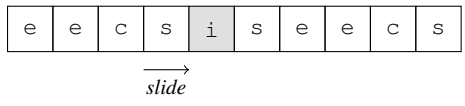
Next, we slide our window forward again, adding in `s`. However, there is already an `s` in our window, so adding this new `s` would result in a duplicate letter. Since our window is invalid, we will need to shift forward the left boundary of our window until there is only one `s` remaining (updating the collection of letters in the window as necessary).



Best window size encountered: 4
Letters currently in window: ***c, s, i***

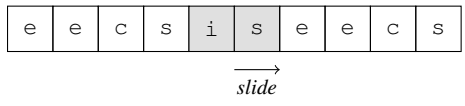


Best window size encountered: 4
Letters currently in window: ***s, i***



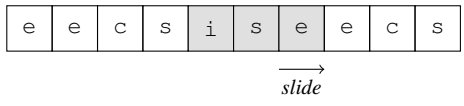
Best window size encountered: 4
Letters currently in window: ***i***

After removing the other `s` from the window, we can safely add the new `s` without introducing a duplicate letter. The window size now becomes 2, which is not better than the best known length encountered so far, so the best window size is not updated.



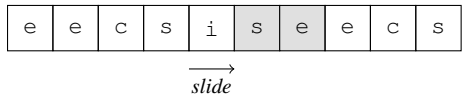
Best window size encountered: 4
Letters currently in window: ***i, s***

We slide our window forward and add `e`, which does not introduce a duplicate letter.

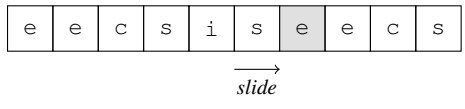


Best window size encountered: 4
Letters currently in window: ***i, s, e***

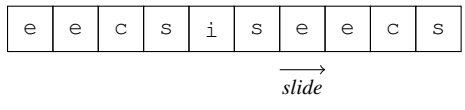
The next `e`, however, introduces a duplicate letter, so we increment the left boundary of our window until the other `e` is gone.



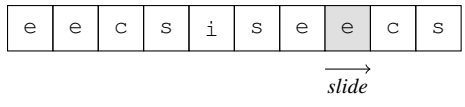
Best window size encountered: 4
Letters currently in window: ***s, e***



Best window size encountered: 4
Letters currently in window: ***e***

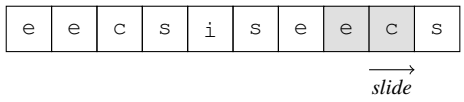


Best window size encountered: 4
Letters currently in window:



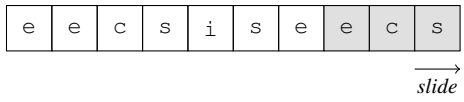
Best window size encountered: 4
Letters currently in window: ***e***

We slide our window forward and add `c`, which does not introduce a duplicate letter.



Best window size encountered: 4
Letters currently in window: ***e, c***

We slide our window forward and add `s`, which does not introduce a duplicate letter.



Best window size encountered: 4
Letters currently in window: ***e, c, s***

The window has now reached the end, so we are done iterating. At this point, the best window size encountered (in this case, 4) is the length of the longest substring without any duplicate letters, and thus the solution to our problem. An implementation of this solution is shown below:

```
1   int32_t length_of_longest_substring(const std::string& str) {
2     std::vector<int8_t> count(26);  // keeps track of whether each letter is in our window
3     size_t left_idx = 0, right_idx = 0;
4     int32_t best = 0;
5
6     while (right_idx < str.length()) {
7       ++count[str[right_idx] - 'a'];    // increment count for letter at right_idx
8       while (count[str[right_idx] - 'a'] > 1) {  // while loop runs if there is a duplicate
9         --count[str[left_idx] - 'a'];  // remove character at left_idx from count
10        ++left_idx;  // slide left_idx forward
11      } // while
12
13      int32_t current_length = right_idx - left_idx + 1;
14      best = std::max(best, current_length);  // update if current length is better than best so far
15      ++right_idx;
16    } // while
17
18    return best;
19  } // length_of_longest_substring()
```

What is the time complexity of this solution? There is a nested loop, so it may initially seem that this algorithm runs in $\Theta(n^2)$ time, where n is the length of the given string. However, this assumption is incorrect because there is a dependency involved between the two loops: the inner loop only runs as long as the count of the character at `right_idx` is greater than one, but the count itself is bounded by the number of times `right_idx` is incremented within the outer loop! Because the inner loop can only run at most once for each iteration of the outer loop (which runs n times), the combined time complexity of the two loops must also be $\Theta(n)$.

The auxiliary space used by this solution is constant, since we only declared a few variables and a vector of size 26, none of which depend on the size of the given string itself.

Example 15.8 You are given two strings, `str` and `target`, with lengths of s and t, respectively. Write a function that returns the shortest substring of `str` that contains every character in `target` (with the same frequency if there are duplicates). If there is no such substring in `str` such that every letter is in `target` with the same frequency, then return the empty string `""`. For simplicity, you may assume that the characters in the string can only contain the letters a-z, all in lowercase, and that there is only one unique solution.

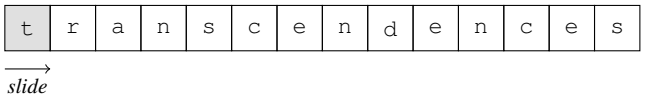
Example: Given `str` = `"transcendences"` and `target` = `"eecs"`, you would return `"ences"`, since this is the shortest substring in `"transcendences"` that contains all the letters in `"eecs"` with the same frequency (2 e's, 1 c, 1 s).

As with before, a simple solution would be to generate all possible substrings of the original string and find the shortest one that contains all of the characters in the target string. However, this approach is inefficient. Instead, we can use a sliding window approach to solve the problem with a time complexity that is linear on the lengths of the two given strings.

Similar to the previous problem, we will slide our window forward using two indices. The right index is responsible for expanding the window, while the left index is responsible for contracting the window. The movement of these indices depends on the contents of our window.

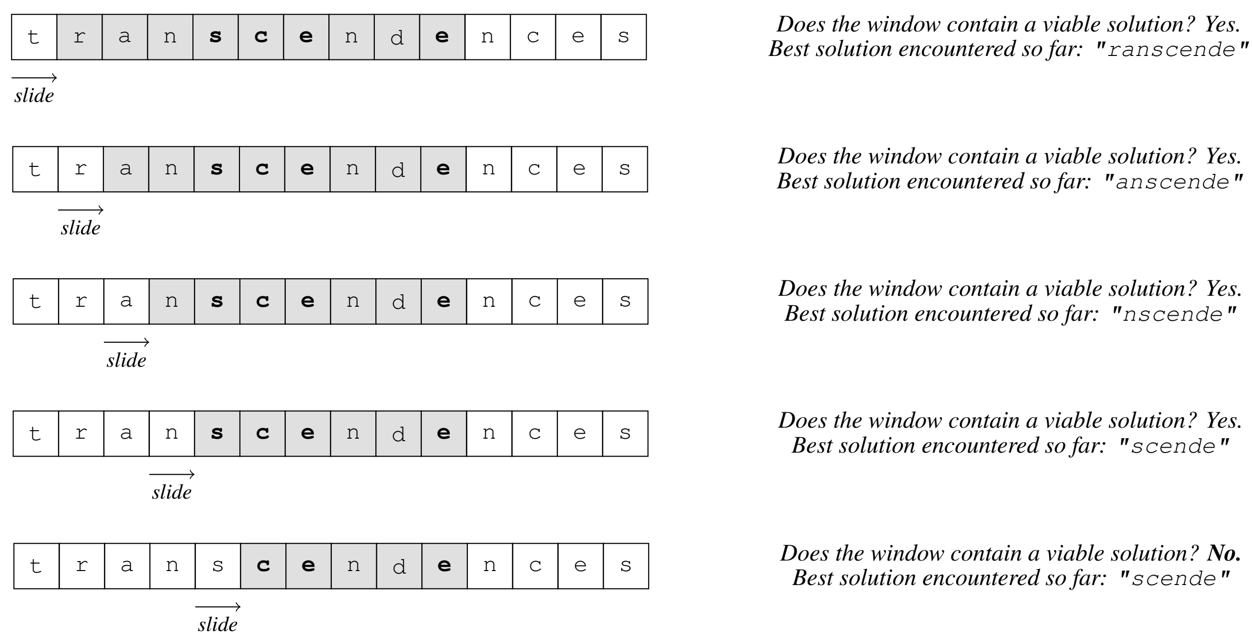
In our example, we will continuously increment the right index until we encounter a window that has all of the characters in `target` (with matching frequencies for duplicate letters), which we will define as a *viable window*. Once we find a viable window, we remember its contents as a possible solution. Then, we will move the left index forward to see if we can get a viable window with a smaller length. As long as our window remains viable, we will continuously move the left index forward and store the result if it is the best solution we have encountered so far. However, once our window is no longer viable, we stop incrementing the left index and begin incrementing the right index once again, expanding our window forward to find the next viable solution. This process continues until the right index reaches the end of the string, which indicates that there are no more windows to consider. At this point, we would return the best solution we encountered so far.

Let's look at this process visually using the example provided with the problem. First, we continuously increment the right index until our window stores a viable solution that contains all the letters in our target string.

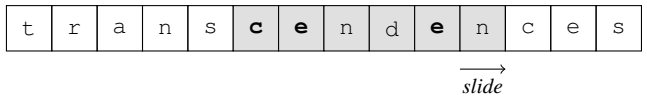


Does the window contain a viable solution? No.
Best solution encountered so far: ***""***

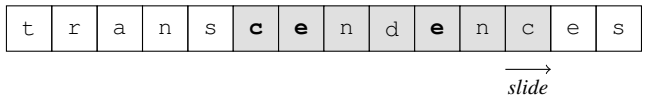
At this point, we have a viable solution in our window. We keep track of this solution, and then we continuously increment the left index until the window no longer stores a viable solution (updating the best known solution along the way).



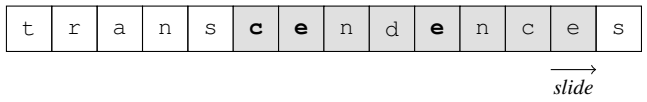
The window no longer holds a viable solution, so we will start incrementing the right index again until we encounter another viable solution.



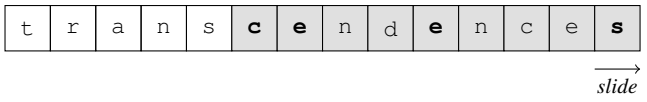
Does the window contain a viable solution? No.
Best solution encountered so far: "scende"



Does the window contain a viable solution? No.
Best solution encountered so far: "scende"

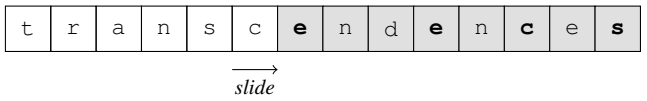


Does the window contain a viable solution? No.
Best solution encountered so far: "scende"

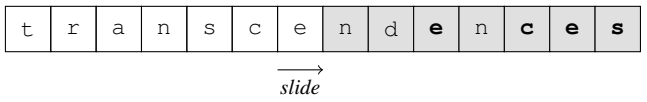


Does the window contain a viable solution? ***Yes.***
Best solution encountered so far: "scende"

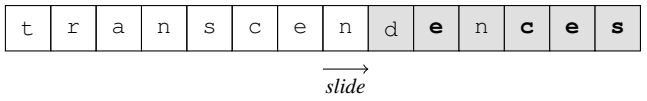
The window is viable again, so we will begin incrementing the left index to see how small we can make our window, updating our best solution if we ever encounter a viable window that is smaller.



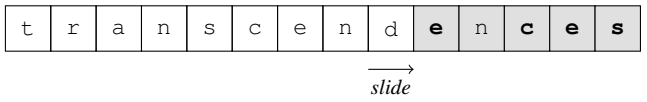
Does the window contain a viable solution? Yes.
Best solution encountered so far: "scende"



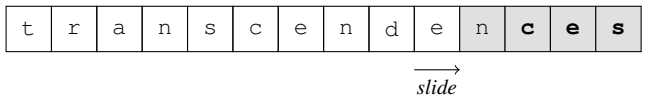
Does the window contain a viable solution? Yes.
Best solution encountered so far: "scende"



Does the window contain a viable solution? Yes.
Best solution encountered so far: "scende"



Does the window contain a viable solution? Yes.
Best solution encountered so far: "ences"



Does the window contain a viable solution? ***No.***
Best solution encountered so far: "ences"

The window is no longer viable, so we will resume incrementing the right index. However, this moves the right index off the end of the string. This indicates that we have checked all valid windows, so the current best solution of `"ences"` must be the solution to the entire problem. An implementation of this solution is provided below:

```
1   std::string min_window_substring(const std::string& str, const std::string& target) {
2     if (str.length() == 0 || target.length() == 0) {
3       return "";  // no solution if either string is empty
4     } // if
5
6     // vector keeps track of character frequency in window ('a' = 0, 'b' = 1, ..., 'z' = 25)
7     std::vector<int32_t> count_in_window(26);
8     // vector keeps track of character frequency in target ('a' = 0, 'b' = 1, ..., 'z' = 25)
9     std::vector<int32_t> count_in_target(26);
10    for (size_t i = 0; i < target.length(); ++i) {
11      ++count_in_target[target[i] - 'a'];
12    } // for i
13
14    // counts the number of characters that are matched in the window
15    // if this value ever equals the length of the target string, we have a viable window
16    size_t characters_matched = 0;
17    // left and right indices
18    size_t left_idx = 0, right_idx = 0;
19    // best solution
20    std::string best;
21
22    /* ... continued on next page ... */
```

```
23      // iterate while right_idx is not off the end
24      while (right_idx < str.length()) {
25        char right_idx_char = str[right_idx];
26        ++count_in_window[right_idx_char - 'a'];
27        // if character in target string added (and not over frequency), increment characters matched
28        if (count_in_window[right_idx_char - 'a'] <= count_in_target[right_idx_char - 'a']) {
29          ++characters_matched;
30        } // if
31
32        // if viable, try to contract the window (move left_idx forward) until window no longer viable
33        while (left_idx <= right_idx && characters_matched == target.length()) {
34          // update best known solution if needed
35          size_t window_length = right_idx - left_idx + 1;
36          if (best.empty() || window_length < best.length()) {
37            best = str.substr(left_idx, window_length);  // store window as substring
38          } // if
39
40          char left_idx_char = str[left_idx];
41          --count_in_window[left_idx_char - 'a'];
42          // if character removed was in target, decrement characters_matched
43          // this means the window is no longer viable, and would terminate this while loop
44          if (count_in_target[left_idx_char - 'a'] > 0
45              && count_in_window[left_idx_char - 'a'] < count_in_target[left_idx_char - 'a']) {
46            --characters_matched;
47          } // if
48
49          // move the left index forward, contracting the window
50          ++left_idx;
51        } // while
52
53        // window is not viable, so move the right index forward, expanding the window
54        ++right_idx;
55      } // while
56      return best;
57    } // min_window_substring()
```

Since the work done by the sliding window (lines 24-55) is linear on the size of the input string `str`, and we iterate over the characters of the `target` string once on lines 10-12, the overall time complexity of this solution is $\Theta(s+t)$, where s and t are the lengths of the two strings.

15.7 Monotonic Stacks and Queues (*)

A **monotonic stack/queue** is a stack or queue whose elements are stored in a strictly ascending or descending order. You can think of these containers as similar to their regular stack or queue counterparts, but with a key distinction for the push operation: before pushing an element into a monotonic stack or queue, you must first check if it breaks the monotonic condition (i.e., the strictly ascending or descending order of elements). If it does, you must pop out all elements that violate the monotonic condition before pushing the new element in. Monotonic containers can be useful in solving problems such as finding the next largest or smallest element in a list, or identifying the maximum or minimum value in a sliding window. We will go over some of these problems in this section.

Example 15.9 You are given a vector of integers `temps` that stores the daily temperature forecasts for the next few days. Implement a function that, for each index of the input vector, stores the number of days you would need to wait for a warmer temperature. If there is no future day where this is possible, a value of 0 should be stored.

Example: Given the vector `[25, 32, 16, 22, 21, 20, 21, 23, 33]`, you would return the solution `[1, 7, 1, 4, 3, 1, 1, 1, 0]`, since you would need to wait 1 day on day 0 for a warmer temperature (25 → 32), 7 days on day 1 for a warmer temperature (32 → 33), 1 day on day 2 for a warmer temperature (16 → 22), 4 days on day 3 for a warmer temperature (22 → 23), and so on.

A naïve brute-force force approach would iterate through the entire vector and, for each day, iterate again over all the remaining days to find a warmer temperature. This approach utilizes a nested loop that runs in $\Theta(n^2)$ time, where n is the number of days in the provided `temps` vector.

How can we do better? One key insight to notice is that, if there are multiple days where temperatures are in descending order, then those days will share the same "warmer temperature day," if there is one. Instead of looping over the entire vector for each day, we can "defer" finding the solution for days whose temperatures are in descending order until we encounter a warmer temperature, at which we can move backward and assign the solution for all these days at the same time. As an example, consider the following `temps` vector:

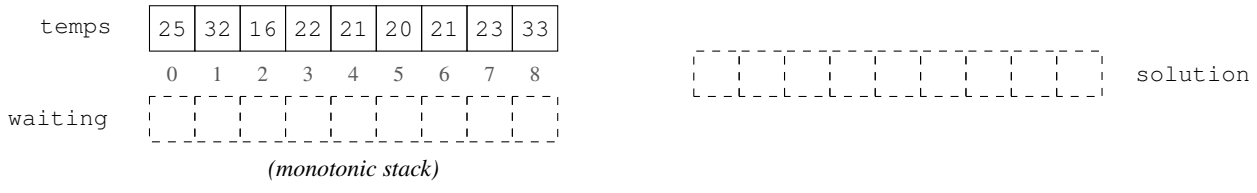
```
[18, 17, 16, 15, 14, 13, 12, 11, 19]
```

Since the first eight days have decreasing temperatures (18 → 11), the solution for all of these eight days is to wait until the ninth day. Instead of looping over the vector for each of the first eight days, we will wait until we encounter a warmer temperature (in this case, 19 on the ninth day) and then backfill this result to the previous eight days.

To accomplish this, we would need to store our data in a container that allows us to access *all the days that are still waiting on a warmer temperature* every time we consider a new temperature in the `temps` vector. Since we want to consider previous temperatures every time we consider a new temperature value, this container will need to support last-in, first-out (LIFO) behavior, which matches the functionality of a stack. In addition, we should only defer finding a solution if the temperatures are in descending order, as these days may end up waiting for the same higher temperature value. Because of this, our stack will need to keep track of the temperatures in *descending* order so that we can quickly

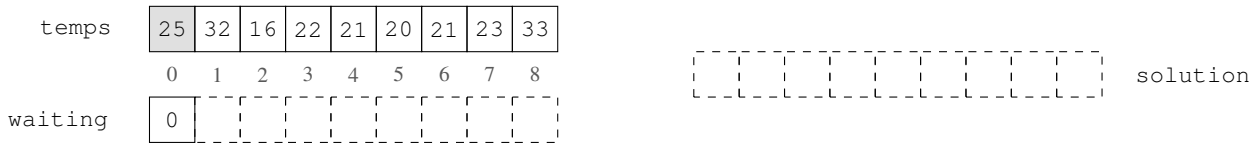
determine if the temperature value we are currently considering is higher than any of the temperatures waiting for a solution in our stack. This is where a monotonic stack can come into play.

As an example, consider the initial example provided in our problem: `[25, 32, 16, 22, 21, 20, 21, 23, 33]`. We will iterate over all the temperatures in our vector, pushing them into a descending monotonic stack (which we will call `waiting`) if we do not know of a warmer temperature. However, once we discover a warmer temperature, we can pop temperatures off the monotonic stack and assign them a solution. Since we want to know the number of days we need to wait, we will store the indices of the days in our stack rather than the temperatures themselves (which we can easily access by indexing into the temperatures vector). This allows us to more easily compute the number of days we need to wait to encounter a warmer temperature.



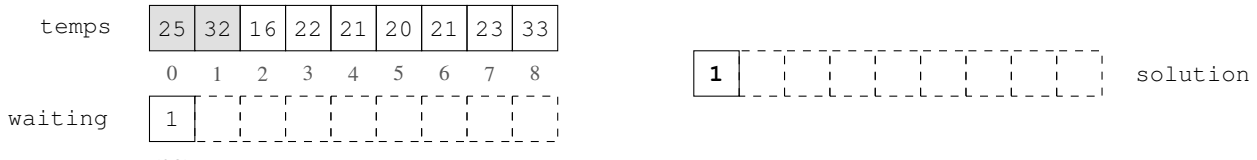
(monotonic stack)

We start off at day 0, with a temperature of 25. We do not yet know of a warmer temperature, so we push day 0 into the waiting stack.



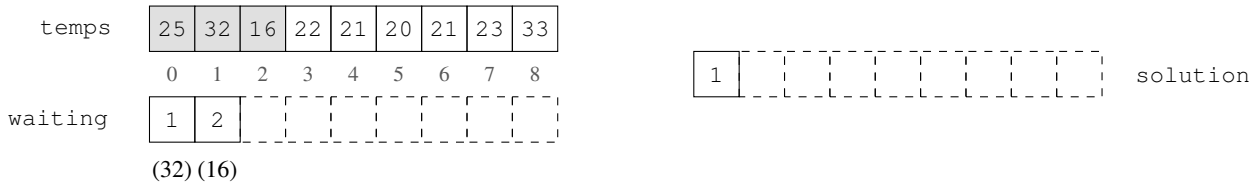
(25)

Next, we consider day 1, with a temperature of 32. This is warmer than the temperature on day 0, so we pop 0 out of the stack and update the solution of day 0 to 1 - 0 = 1 (the wait time between day 0 and day 1). Then, we push day 1 into the stack, since we have not yet discovered a warmer temperature for day 1.



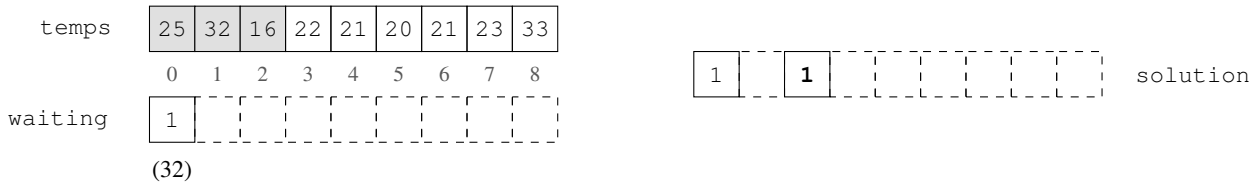
(32)

Next, we consider day 2, with a temperature of 16. This is not warmer than the day at the top of our stack, so we push 2 into the stack.



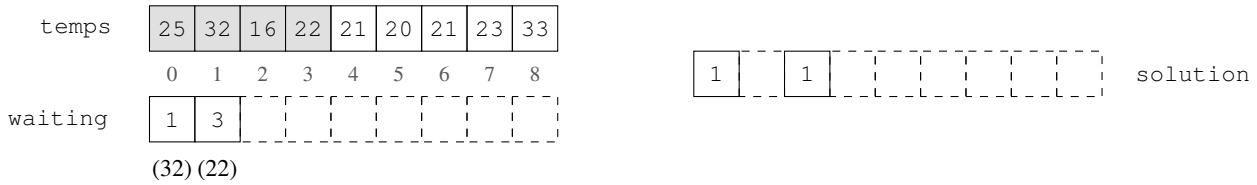
(32) (16)

Next, we consider day 3, with a temperature of 22. The day at the top of the stack, 2, has a temperature of 16. Since 22 is warmer than 16, we know that day 3 is the first day that is warmer than day 2, and that we have to wait 3 - 2 = 1 day to experience a warmer temperature starting from day 2. Therefore, we set the solution for day 2 to 1 and pop day 2 out of the stack.



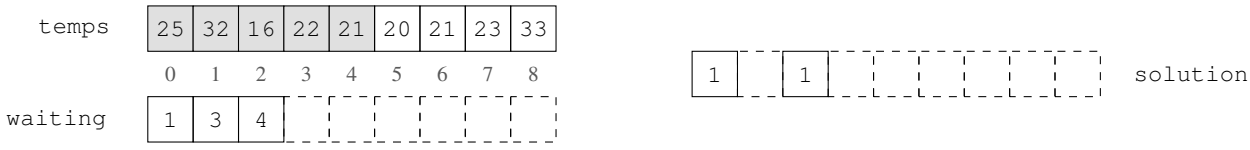
(32)

The day at the top of the stack, 1, has a temperature of 32. 22 is not warmer than 32, so we push day 3 into the stack.



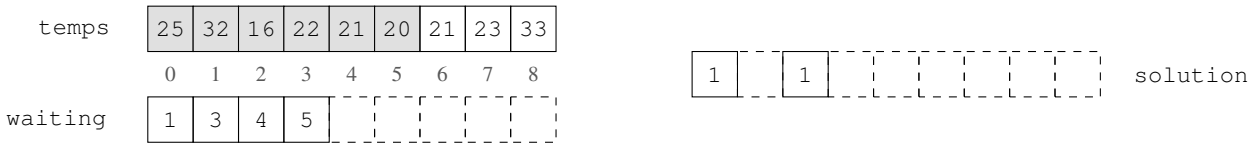
(32) (22)

Next, we consider day 4, with a temperature of 21. This is not warmer than the day at the top of the stack, so we push day 4 into the stack.



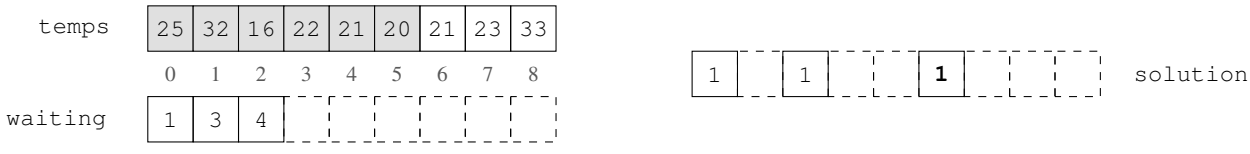
(32) (22) (21)

Next, we consider day 5, with a temperature of 20. This is not warmer than the day at the top of the stack, so we push day 5 into the stack.



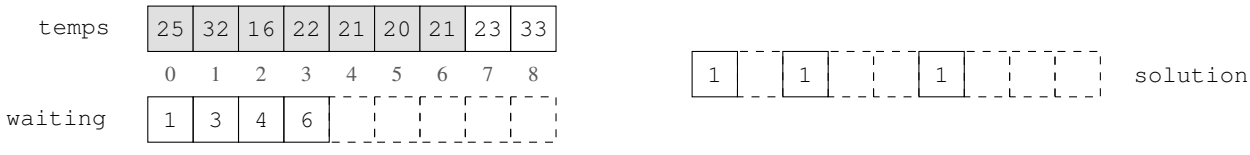
(32) (22) (21) (20)

Next, we consider day 6, with a temperature of 21. The day at the top of the stack, 5, has a temperature of 20. Since 21 is warmer than 20, we know that we have to wait 6 - 5 = 1 day to experience a warmer temperature starting from day 5. Therefore, we set the solution for day 5 to 1 and pop day 5 out of the stack.



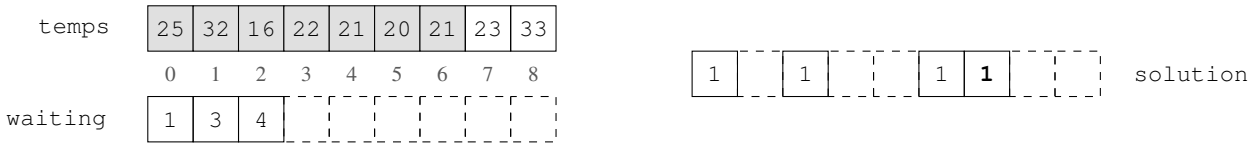
(32) (22) (21)

The day at the top of the stack, 4, has a temperature of 21. The current temperature of 21 is not warmer than 21, so day 6 is pushed into the stack.



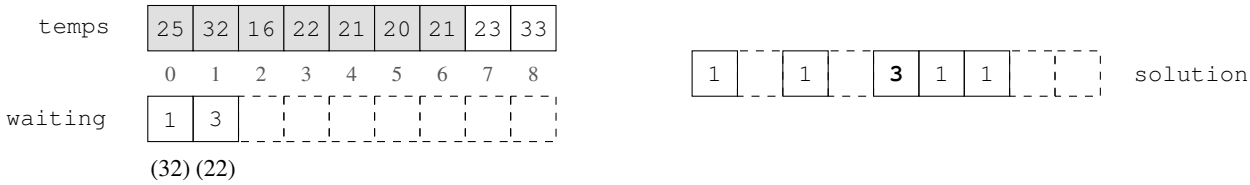
(32) (22) (21) (21)

Next, we consider day 7, with a temperature of 23. The day at the top of the stack, 6, has a temperature of 21. Since 23 is warmer than 21, we know that we have to wait 7 - 6 = 1 day to experience a warmer temperature starting from day 6. Therefore, we set the solution for day 6 to 1 and pop day 6 out of the stack.



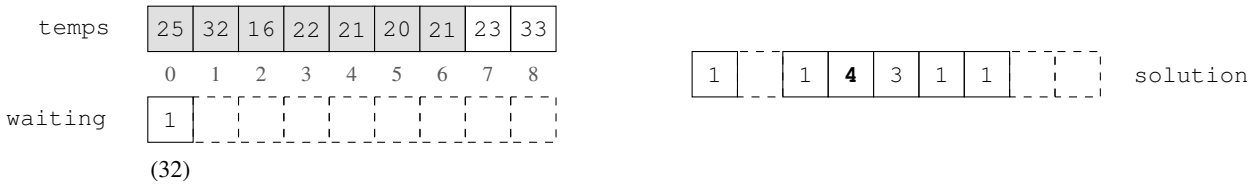
(32) (22) (21)

The current temperature of 23 is still warmer than the day at the top of the stack (day 4), which has a temperature of 21. This indicates that day 7 is also the first day with a warmer temperature starting from day 4, and that we had to wait 7 - 4 = 3 days for this warmer temperature. Therefore, we set the solution for day 4 to 3 and pop day 4 out of the stack.



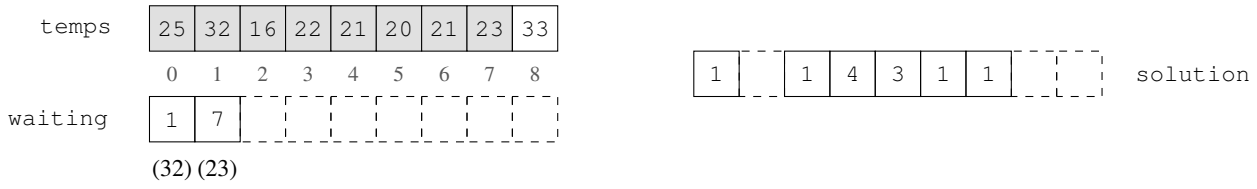
(32) (22)

The current temperature of 23 is still warmer than the day at the top of the stack (day 3), which has a temperature of 22. With similar reasoning as above, we know that we have to wait 7 - 3 = 4 days for a warmer temperature starting from day 3. Therefore, we set the solution for day 3 to 4 and pop day 3 out of the stack.

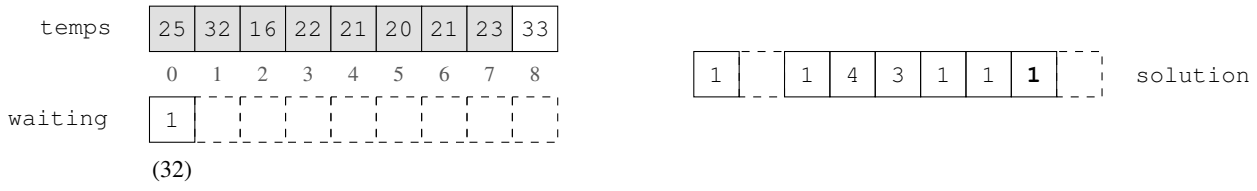


(32)

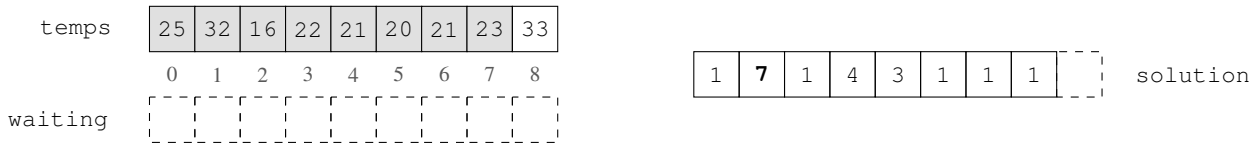
The current temperature of 23 is no longer warmer than the day waiting at the top of the stack, so we push day 7 into the stack.



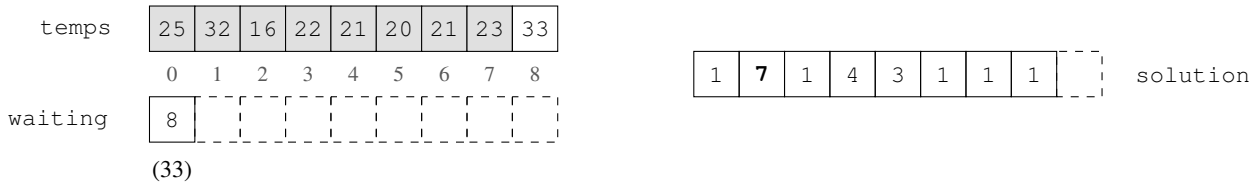
Next, we consider day 8, with a temperature of 33. The day at the top of the stack, 7, has a temperature of 23. Since 33 is warmer than 23, we know that day 8 is the first day with a warmer temperature after day 7, and that we would have to wait 8 - 7 = 1 day to experience a warmer temperature starting from day 7. Therefore, we set the solution for day 7 to 1 and pop day 7 out of the stack.



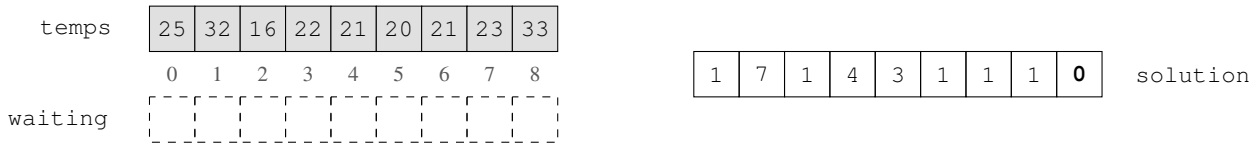
The current temperature of 33 is still warmer than the day at the top of the stack (day 1), which has a temperature of 32. We can therefore pop day 1 out of the stack and assign its solution to 8 - 1 = 7 days, since we need to wait 7 days for a warmer temperature starting from day 1.



The stack is empty, so we push day 8 into the stack.



We are now down iterating over the `temps` vector. At this point, there are no more warmer temperatures to consider, so all days remaining in the queue must have a solution 0 (the value to store if there is no solution).



This completes our algorithm, and our solution vector stores the number of days we need to wait for a warmer temperature on each day. An implementation of this problem is shown below:

```
1  std::vector<int32_t> warmer_temperatures(const std::vector<int32_t>& temps) {
2    std::vector<int32_t> solution(temps.size());
3    std::stack<int32_t> waiting;
4    for (size_t curr_day = 0; curr_day < temps.size(); ++curr_day) {
5      int32_t curr_temp = temps[curr_day];
6      // pop until the current day's temp is not warmer than the day at the top of the stack
7      while (!waiting.empty() && temps[waiting.top()] < curr_temp) {
8        // curr_temp is warmer, so we found a solution for prev_day
9        int32_t prev_day = waiting.top();
10       solution[prev_day] = curr_day - prev_day;
11       waiting.pop();
12     } // while
13     // after all lower temps are popped out of the monotonic stack, push in the current day
14     waiting.push(curr_day);
15   } // for curr_day
16   return solution;
17 } // warmer_temperatures()
```

What is the time complexity of this solution? It may initially seem that the time complexity should be $\Theta(n^2)$ given n temperatures, since there is a nested `while` loop instead a `for` loop. However, notice that each element can only be added to the stack *once*, which indicates that the stack only experiences n pops. Furthermore, each iteration of the inner `while` loop performs exactly one pop; since we know the stack only gets popped n times, this means the `while` loop must not perform more than n pops in total. Because each element only gets pushed and popped once, the time complexity of this solution is actually $\Theta(n)$. This is actually a common outcome when analyzing monotonic stack or queue problems: because of how elements are pushed or popped in, a solution may seem to have a quadratic time complexity when it is actually linear!

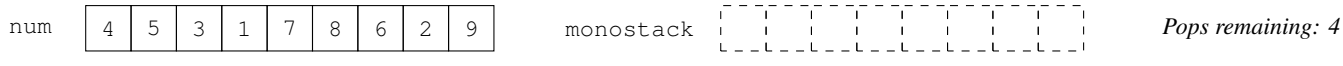
Example 15.10 You are given a string `num` that represents a non-negative integer, as well as an integer k. Implement a function that returns the smallest integer obtainable after removing exactly k digits from `num`.

Example: Given `num = "453178629"` and $k = 4$, you would return `"17629"`, since 17629 is the smallest integer that can be created after removing four digits from 453178629.

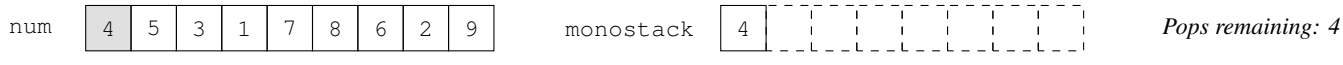
To obtain the smallest possible number, our goal is to remove as many big digits as possible in the most significant positions of the original number. This can be done by iterating over the string in order of decreasing significance position (i.e., from left to right) and tracking the digits encountered in a container that allows us to identify

- its significance position (i.e., the container must be to know the order in which its digits were considered)
- whether the digit should be removed (i.e., the digits must be stored in sorted order)

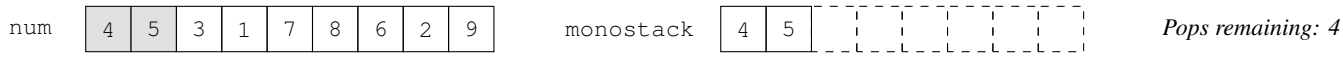
A monotonic stack fits the bill, since it provides both LIFO behavior while also maintaining its values in sorted order. To solve the problem, we will create an ascending monotonic stack that stores the digits we want to keep in our string. The largest element encountered will be at the top of the stack, which allows us to pop it off if we ever encounter a smaller digit that is more worthwhile to keep. Since we are limited in the number of digits we can remove, we will restrict the stack to exactly k pops — after the k^{th} pop, all remaining digits in the string are kept. If we end up visiting all digits without performing k pops, we will pop any excess values off the top of the stack so that we end up with exactly k pops (this is okay because the stack is sorted, so the larger digits are popped first). Furthermore, we will iterate over the string from left to right, ensuring that digits with higher significance positions are prioritized when determining which digits to remove (since a large digit in a higher significance position is worse than a large digit in a lower significance position). Let's walk through the process using the provided example.



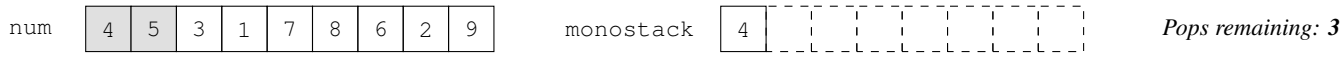
The first digit is 4. Our stack is currently empty, so there are no values to compare to, and we can push 4 into the stack.



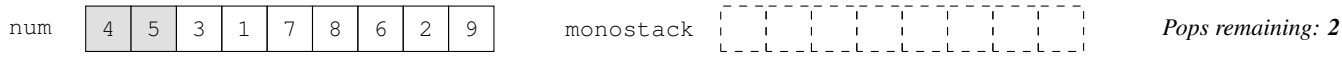
The next digit is 5. This is not better than the digit at the top of the stack, 4, so we do not need to pop 4 out. 5 is then pushed into the stack.



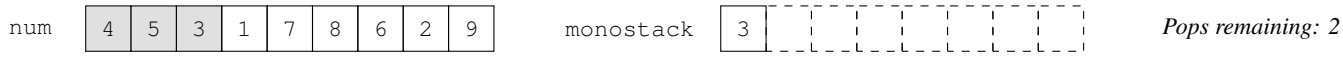
The next digit is 3. This is better than the digit at the top of the stack, 5, so we pop 5 out and decrement our counter.



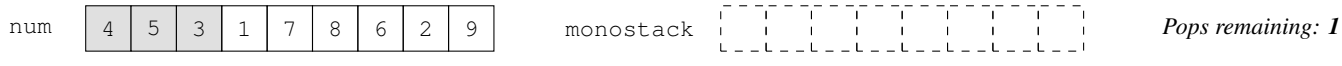
3 is still better than the digit at the top of the stack, 4, so we also pop 4 out an decrement our counter.



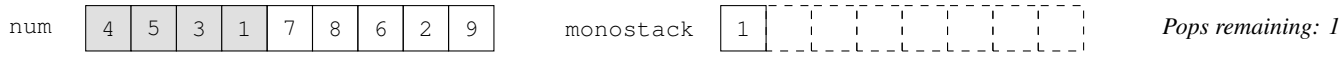
There are no more values in the stack, so we can push in 3.



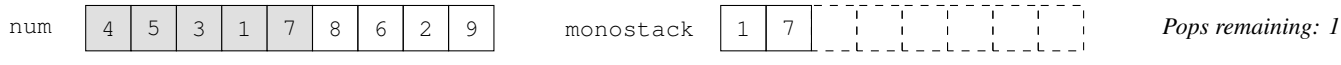
The next digit is 1. This is better than the digit at the top of the stack, 3, so we pop 3 out and decrement our counter.



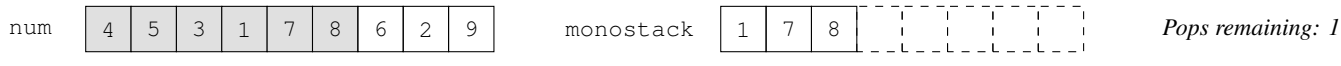
There are no more values in the stack, so we can push in 1.



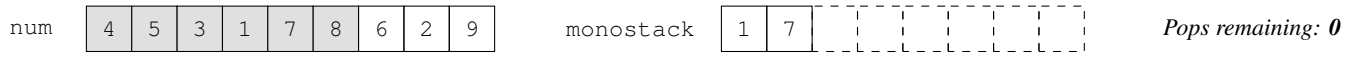
The next digit is 7. This is not better than the digit at the top of the stack, 1, so we do not need to pop 1 out. 7 is then pushed into the stack.



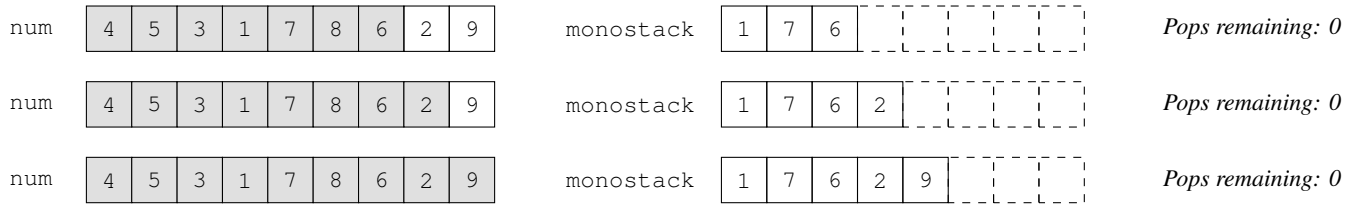
The next digit is 8. This is not better than the digit at the top of the stack, 7, so we do not need to pop 7 out. 8 is then pushed into the stack.



The next digit is 6. This is better than the digit at the top of the stack, 8, so we pop 8 out and decrement our counter.



We are now out of pops, since the counter has reached zero. This means that none of the remaining digits can be removed, so we will push them all into the stack.



The contents of our stack (`"17629"`) make up our final solution. An implementation is shown below:

```
1  std::string min_remove_k_digits(const std::string& num, int32_t k) {
2    int32_t counter = k;
3    // use vector for iterator support, but this behaves like a stack (.top() == .back())
4    std::vector<char> monostack;
5    for (char curr_digit : num) {
6      while (counter > 0 && !monostack.empty() && monostack.back() > curr_digit) {
7        monostack.pop_back();
8        counter -= 1;
9      } // while
10     // curr_digit != 0 check prevents leading zeros in the solution string
11     if (!monostack.empty() || curr_digit != '0') {
12       monostack.push_back(curr_digit);
13     } // if
14   } // for curr_digit
15   // if counter > 1, not all available digits are popped, so keep popping off the monotonic stack
16   // this is okay because stack is sorted, so the larger digits get popped first
17   while (!monostack.empty() && counter-- != 0) {
18     monostack.pop_back();
19   } // while
20   // contents of stack forms out final solution (if stack is empty, solution must be zero)
21   return monostack.empty() ? "0" : std::string{monostack.begin(), monostack.end()};
22 } // min_remove_k_digits()
```

It should also be noted that `std::string` supports `.push_back()` and `.pop_back()`, so you can use a string instead of a stack container (on line 4 below). This allows you to return the solution directly without a conversion; we will go over string operations in the next chapter.

```
1  std::string min_remove_k_digits(const std::string& num, int32_t k) {
2    int32_t counter = k;
3    // use a string to emulate a stack
4    std::string monostack;
5    for (char curr_digit : num) {
6      while (counter > 0 && !monostack.empty() && monostack.back() > curr_digit) {
7        monostack.pop_back();
8        counter -= 1;
9      } // while
10     // curr_digit != 0 check prevents leading zeros in the solution string
11     if (!monostack.empty() || curr_digit != '0') {
12       monostack.push_back(curr_digit);
13     } // if
14   } // for curr_digit
15   // if counter > 1, not all available digits are popped, so keep popping off the monotonic stack
16   // this is okay because stack is sorted, so the larger digits get popped first
17   while (!monostack.empty() && counter-- != 0) {
18     monostack.pop_back();
19   } // while
20   // contents of stack forms out final solution (if stack is empty, solution must be zero)
21   return monostack.empty() ? "0" : monostack;
22 } // min_remove_k_digits()
```

Similar to the previous problem, the nested loops in the solution may be deceiving. Each digit can only be pushed into the stack once, which indicates that the stack can only experience at most n pops, where n is the number of digits in the original string `num`. Since each iteration of the inner `while` loop performs exactly one pop, this means that the loop cannot perform more than n pops in total — and because each digit only gets pushed and popped from the stack at most once during the lifetime of the algorithm, the time complexity of the solution is actually $\Theta(n)$.

Example 15.11 You are given a vector of integers `nums`. A sliding window of size k is moving from the left of the vector to the right, and you are only able to see the k numbers in the window. Each time the sliding window moves right by one position. Return the maximum value of each sliding window in a separate vector. You may assume k is not greater than the size of the vector.

Example: Given a vector `nums = [1, 9, 8, 6, 7, 2, 4, 5]`, and $k = 3$, you would return the output vector `[9, 9, 8, 7, 7, 5]`, which stores the maximum value for each iteration of the sliding window of size 3.

Window Position	Maximum Window Value
[1, 9, 8], 6, 7, 2, 4, 5	9
1, **[9, 8, 6]**, 7, 2, 4, 5	9
1, 9, **[8, 6, 7]**, 2, 4, 5	8
1, 9, 8, **[6, 7, 2]**, 4, 5	7
1, 9, 8, 6, **[7, 2, 4]**, 5	7
1, 9, 8, 6, 7, **[2, 4, 5]**	5

The simplest solution would be to iterate over all possible sliding windows and find the maximum value for each window. However, this solution would end up taking $\Theta(nk)$ time, where n is the size of the vector `nums`, since you would have to iterate over all $n-k+1$ possible windows, each of size k. Is there a way to do better?

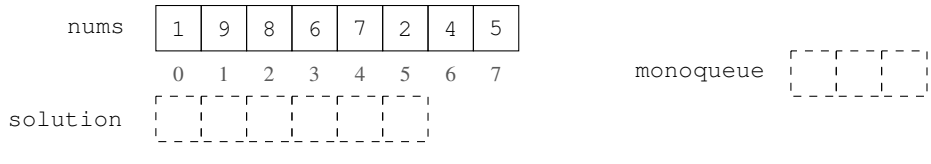
Another method for solving this problem is to use a priority queue that stores an integer pair containing the value and its index. This allows us to determine the largest value that is also at a valid index of our current window. This solution is shown below:

```
1  std::vector<int32_t> max_sliding_window(const std::vector<int32_t>& nums, int32_t k) {
2    std::priority_queue<std::pair<int32_t, int32_t>> pq;  // pair<element value, index>
3    std::vector<int32_t> solution;
4    solution.reserve(nums.size() - k + 1);
5    // push elements of first window into pq
6    for (size_t i = 0; i < k; ++i) {
7      pq.emplace(nums[i], i);
8    } // for i
9    // push largest value of first window in solution vector
10   solution.push_back(pq.top().first);
11   // repeat for the remaining windows
12   for (size_t j = k; j < nums.size(); ++j) {
13     pq.emplace(nums[j], j);
14     // remove elements that are no longer in the current window
15     while (!(pq.top().second > j - k)) {
16       pq.pop();
17     } // while
18     solution.push_back(pq.top().first);
19   } // for j
20   return solution;
21 } // max_sliding_window()
```

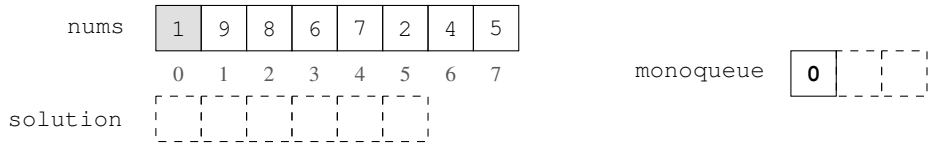
However, this particular solution is not the most ideal either. This is because pushing in new elements and removing older elements both take $\Theta(\log(n))$ time. Preferably, we want a container that allows us to insert new values and remove old values in constant time, while also allowing us to efficiently determine the largest value in each window. This is where a monotonic queue comes into play.

With a standard queue, we would push in new sliding window values to the back of the queue and pop out expired sliding window values out the front of the queue. However, a key insight to notice is that each new element gives us information on which values in the sliding window are promising and which ones are not. Whenever we want to push a new element in, we know that the values in the window that are smaller than this new element cannot ever be the sliding window maximum (since they will leave the window before this new element does). Because of this, it is safe to pop out all values that are smaller whenever we consider a new value to insert into our sliding window, ensuring that our queue stores its values in monotonically descending order. This allows us to identify the maximum value for any sliding window in $\Theta(1)$ time by simply querying the value at the front of the monotonic queue.

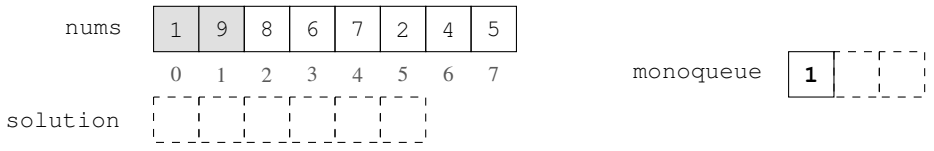
Let's illustrate this process using the provided example. Similar to the warmer temperatures problem, we will store indices in our monotonic queue rather than the values themselves, as this allows us to easily determine whether a value is no longer in the current sliding window (and the index also allows us to get its corresponding value by indexing into the `nums` vector).



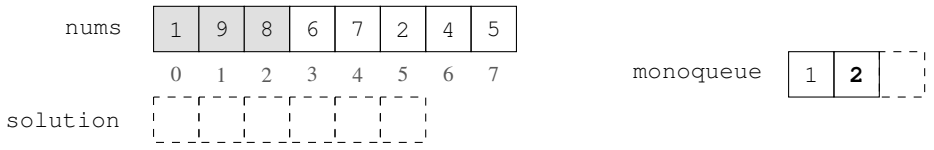
The value at index 0 is 1. The queue is currently empty, so we push index 0 in.



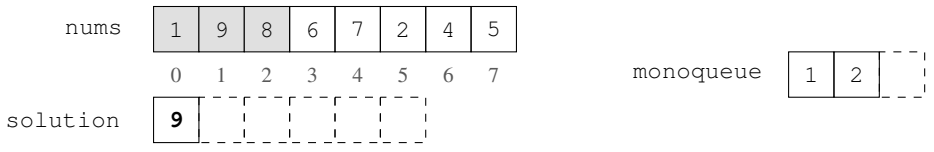
The value at index 1 is 9. The value at the back of the queue (index 0, with a value of 1) is smaller, so we will pop it out before pushing index 1 in.



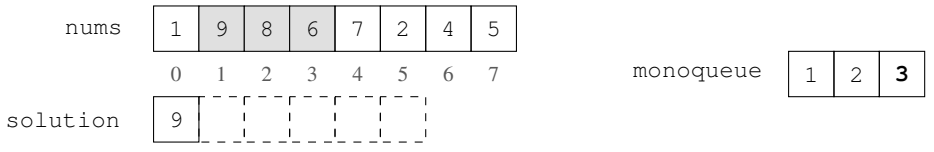
The value at index 2 is 8. The value at the back of the queue (index 1, with a value of 9) is larger, so we will push index 2 into the queue without removing anything.



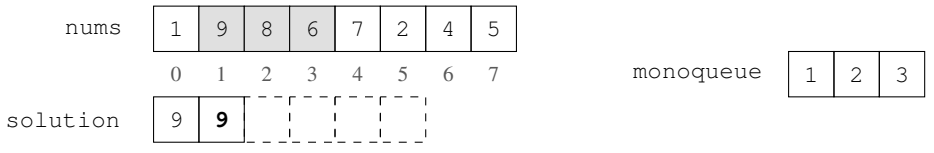
Since our sliding windows have size $k = 3$, our first window is ready to be processed. Since our monotonic queue stores the indices in order of decreasing value, we know that the value at index 1 of `nums` (or 9) must be the maximum value of this first window, since it is at the front of the queue. We therefore push 9 into our solution vector.



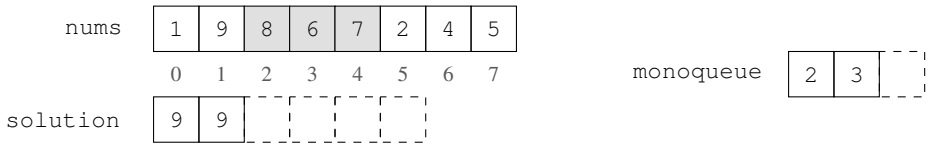
We now increment our window forward by one position. This requires us to remove index 0 from our queue; however, 0 has already been removed earlier (to keep our queue monotonic, since we determined index 0 cannot be the maximum of any remaining window). We will then insert index 3 (the new value in our window) into the queue after removing all values that are smaller. Since the other two indices in the queue have larger values of 9 and 8, nothing is added before index 3 is pushed in.

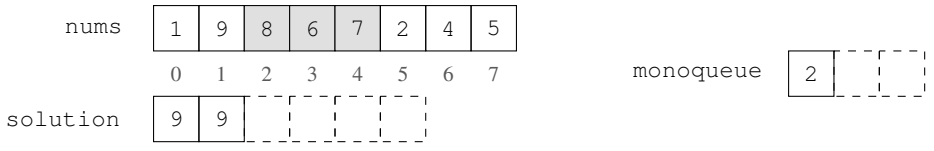


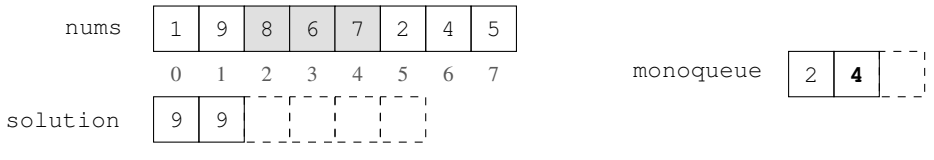
The maximum value of this sliding window must be the value at index 1 (or 9), since index 1 is at the front of the queue. We therefore push 9 into our solution vector.



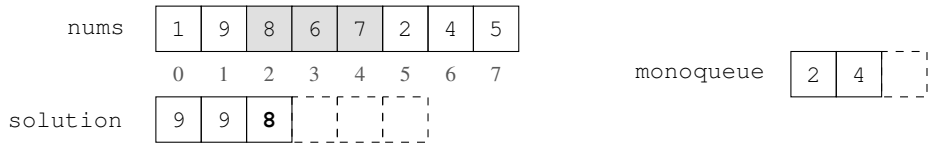
We now increment our window forward by one position. This requires us to remove index 1 from our queue, since it is no longer in our sliding window. We will then insert index 4 into the queue after removing all values that are smaller (in this case index 3, since $6 < 7$).



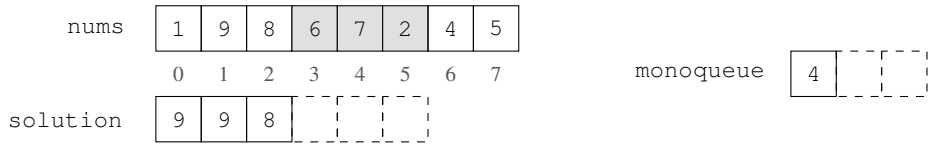


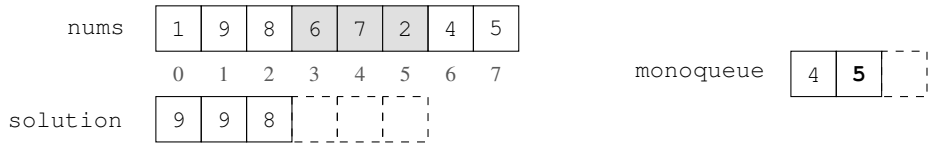


The maximum value of this sliding window is the value at index 2 (or 8), so we push 8 into our solution vector.

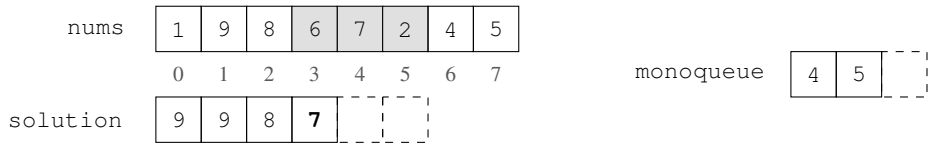


We now increment our window forward by one position. This requires us to remove index 2 from our queue. Since the new value at index 5 (or 2) is not larger than any of the values in the queue (index 4, which has a value of 7), we can push index 5 in without popping anything else out.

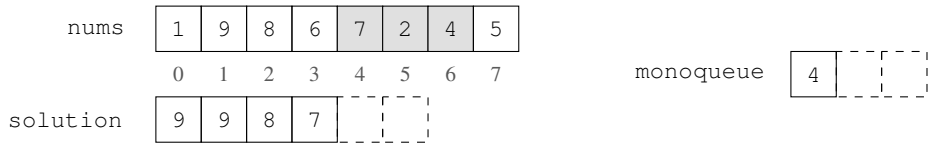


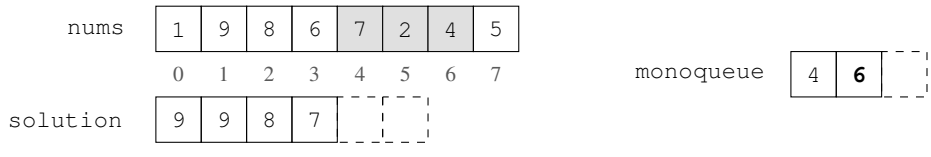


The maximum value of this sliding window is the value at index 4 (or 7), so we push 7 into our solution vector.

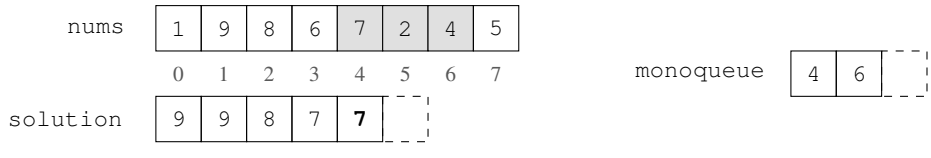


We now increment our window forward by one position. This requires us to remove index 3 from our queue (this step is trivial because index 3 had already been removed earlier). We then pop out all values that are smaller than the new value in our window (in this case, we pop out index 5 with a value of 2) before pushing index 6 in.

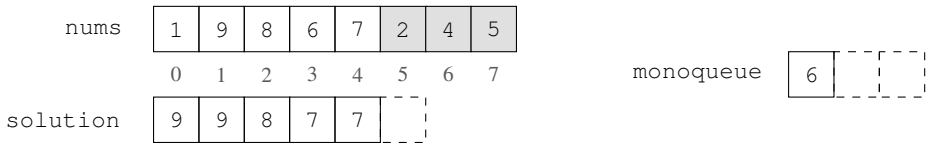


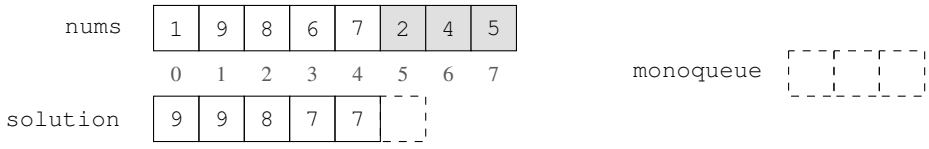


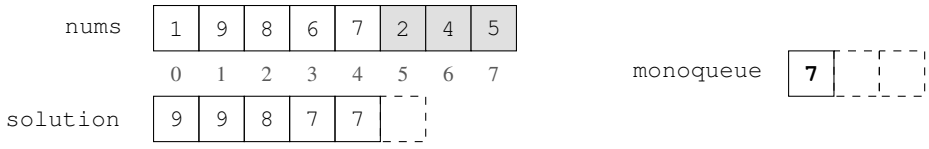
The maximum value of this sliding window is the value at index 4 (or 7), so we push 7 into our solution vector.



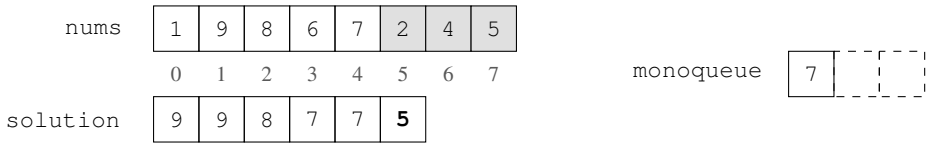
We now increment our window forward by one position. This requires us to remove index 4 from our queue. We then pop out all values that are smaller than the new value in our window (in this case, we pop out index 6 with a value of 4) before pushing index 7 in.







The maximum value of this sliding window is the value at index 7 (or 5), so we push 5 into our solution.



We have finished processing our final sliding window, so our algorithm has completed with a solution of `[9, 9, 8, 7, 7, 5]`. An implementation of this process is shown in the code below. Notice that we use a `std::deque<>` to implement our monotonic queue; this is because we need to be able to pop at both ends of the container (the front to discard values that exit the sliding window, and the back to discard values that are smaller than the new value that enters the sliding window).

```
1  std::vector<int32_t> max_sliding_window(const std::vector<int32_t>& nums, int32_t k) {
2    if (nums.empty()) {
3      return nums;
4    } // if
5    std::deque<int32_t> monoqueue;
6    std::vector<int32_t> solution;
7    solution.reserve(nums.size() - k + 1);
8    for (int32_t curr_idx = 0; curr_idx < nums.size(); ++curr_idx) {
9      // remove old values that are no longer in the sliding window
10     while (!monoqueue.empty() && monoqueue.front() < curr_idx - k + 1) {
11       monoqueue.pop_front();
12     } // while
13     // remove values that are smaller than the newest value in the window,
14     // since they cannot be the maximum value for any future windows
15     while (!monoqueue.empty() && nums[monoqueue.back()] < nums[curr_idx]) {
16       monoqueue.pop_back();
17     } // while
18     // push curr_idx into monotonic queue
19     monoqueue.push_back(curr_idx);
20     // write sliding window maximum to solution
21     if (curr_idx - k + 1 >= 0) {
22       solution.push_back(nums[monoqueue.front()]);
23     } // if
24   } // for curr_idx
25   return solution;
26 } // max_sliding_window()
```

Much like the previous two problems, the time complexity of this solution is $\Theta(n)$, where n is the size of the `nums` vector. This is because each element can only be pushed and popped out of the monotonic queue once, so the total work performed by the loop on line 8 ends up being linear on the size of the initial vector.

15.8 Median Finding Algorithms (⁎)

Finding the median of a given collection of values may seem like a trivial task, but trying to accomplish this in an asymptotically optimal manner is not entirely straightforward. The simplest way to find the median is to sort the container and take the value at index $n/2$ (if the container has an odd size) or the average of the values at indices $n/2$ and $\lfloor n/2 \rfloor - 1$ (if the container has an even size):

```
1  double find_median(const std::vector<double>& vec) {
2    std::vector<double> sorted{vec};
3    size_t sz = sorted.size();
4    if (sz == 0) {
5      std::cerr << "Vector cannot be empty!" << std::endl;
6      throw std::invalid_argument("Cannot find median of an empty vector");
7    } // if
8    std::sort(sorted.begin(), sorted.end());
9    if (sorted.size() % 2 == 1) {  // odd
10     return sorted[sz / 2];
11   } // if
12   else {  // even
13     return 0.5 * (sorted[sz / 2 - 1] + sorted[sz / 2]);
14   } // else
15 } // find_median()
```

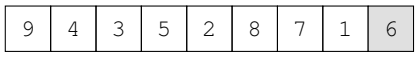
However, the time complexity of such an approach is $\Theta(n \log(n))$, since the sorting step is a bottleneck that dictates the overall performance. Is there a way to do better? In this section, we will discuss two different median finding algorithms designed to find the median in linear time.

※ 15.8.1 *Quickselect* (⁎)

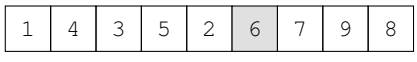
The **quickselect** algorithm can be used to find the k^{th} smallest element of a collection of elements in average-case $\Theta(n)$ time, and it is related to the partitioning step of the quicksort algorithm. Recall that, in the partitioning step:

1. A pivot element is selected.
2. The remaining elements in the array are partitioned so that all elements to the left of the pivot are lesser than the pivot, and all elements to the right of the pivot are greater than the pivot.

We can use the quickselect algorithm to help us find the median: after partitioning, one side of the pivot element must contain the median (if the median is not the pivot value itself). Suppose we know that the median is the k^{th} element among the sorted values. If there are more than k elements to the left of the pivot after partitioning, then the median must be located to the left of the pivot. Otherwise, if there are fewer than k elements to the left of the pivot after partitioning, then the median must be located to the right of the pivot. We can use this strategy to write an algorithm that finds the median by repeatedly recursing into the side that contains the median value. An example is illustrated below:

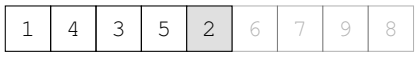


Select pivot.

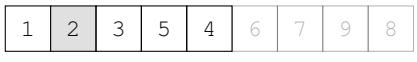


Partition the remaining values around this pivot.

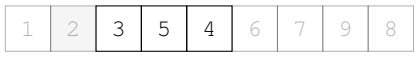
The median must be to the left of the pivot 6.



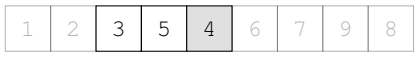
Recurse into the left side and select a pivot.



Partition the remaining values around this pivot.



The median must be to the right of the pivot 2.



Recurse into the right side and select a pivot.

Partition the remaining values around this pivot.

The median must be to the right of the pivot 4.

Recurse into the right side. 5 must be the median.

In summary, the quickselect algorithm can be used to find the median as follows:

1. Select a pivot element from the container of values.
2. Partition the remaining items into two groups: values less than the pivot, and values greater than the pivot.
3. One of these groups contains the median. Assuming the median should be at index k:
   - If k is less than the pivot index, recurse on the left side of the pivot.
   - If k is greater than the pivot index, recurse on the right side of the pivot.
   - If k is equal to the pivot, then the value at the pivot position must be the median.

A recursive implementation of quickselect is shown in the code below:

```
int32_t partition(std::vector<double>& vec, int32_t left, int32_t right) {
  int32_t pivot = --right;
  while (true) {
    while (vec[left] < vec[pivot]) {
      ++left;
    } // while
    while (left < right && vec[right - 1] >= vec[pivot]) {
      --right;
    } // while
    if (left >= right) {
      break;
    } // if
    std::swap(vec[left], vec[right - 1]);
  } // while
  std::swap(vec[left], vec[pivot]);
  return left;
} // partition()

double quickselect(std::vector<double>& vec, int32_t left, int32_t right, int32_t target_idx) {
  // handles case where there is only one element in the range
  if (left + 1 >= right) {
    return vec[left];
  } // if
  int32_t pivot_idx = partition(vec, left, right);
  // if target_idx (index of median) is equal to pivot_idx, return
  if (target_idx == pivot_idx) {
    return vec[target_idx];
  } // if
  // if target_idx < pivot_idx, recurse on left side
  else if (target_idx < pivot_idx) {
    return quickselect(vec, left, pivot_idx, target_idx);
  } // else if
  // if target_idx > pivot_idx, recurse on right side
  else {
    return quickselect(vec, pivot_idx + 1, right, target_idx);
  } // else
} // quicksort()

double find_median(std::vector<double>& vec) {
  if (vec.empty()) {
    std::cerr << "Vector cannot be empty!" << std::endl;
    throw std::invalid_argument("Cannot find median of an empty vector");
  } // if
  if (vec.size() % 2 == 1) {  // odd
    return quickselect(vec, 0, vec.size(), vec.size() / 2);
  } // if
  // even
  return 0.5 *
         (quickselect(vec, 0, vec.size(), vec.size() / 2) +
          quickselect(vec, 0, vec.size(), vec.size() / 2 - 1));
} // find_median()
```

The average-case time complexity of quickselect is $\Theta(n)$. To see why, suppose each pivot splits the list into two roughly equally sized partitions. Since we only need to recurse on the side with the median, the work we do at each iteration decreases by approximately half. This infinite geometric series yields us an $\Theta(n)$ complexity:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \ldots = 2n = \Theta(n)$$

That being said, this is obviously an optimistic assumption. Nonetheless, much like our analysis of quicksort's average-case time complexity in the previous chapter, we can use the formula for the sum of a geometric series to show that the time complexity would stilll be a constant multiple of n for any other constant factor split.

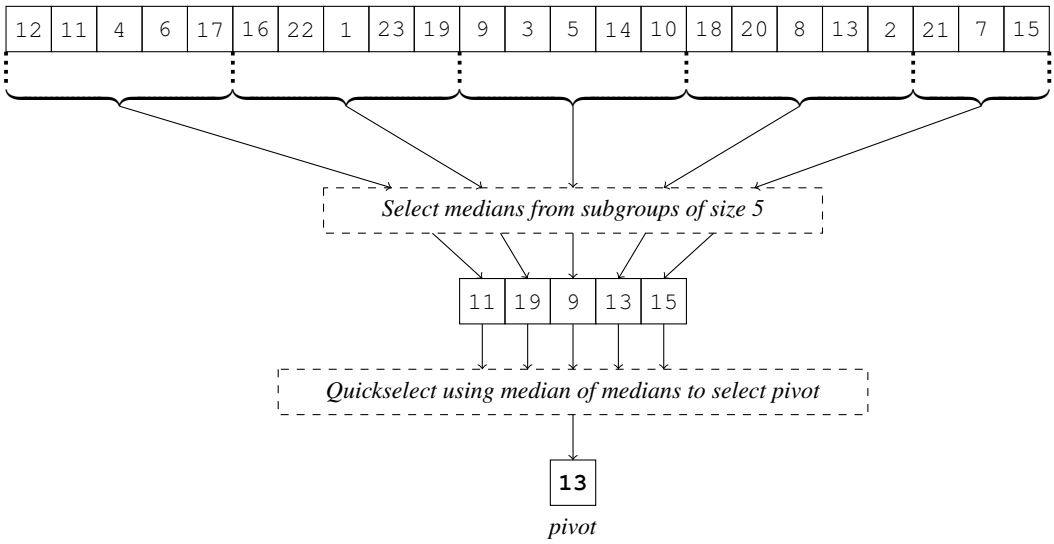
However, much like quicksort, the worst-case time complexity is $\Theta(n^2)$ if you get unlucky and pick the worst-case pivot at every iteration (which could end up decreasing the search range by only one instead of half). In practice, however, this scenario rarely happens, and quickselect exhibits a linear-time performance for most practical use cases. That being said, there is an approach that can be used to help us find the median of a collection of items in *worst-case* $\Theta(n)$ time known as the *median of medians* algorithm.

※ 15.8.2 *Median of Medians* (✻)

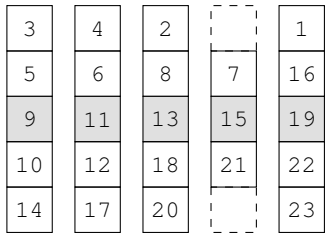
To avoid the worst-case time complexity of $\Theta(n^2)$, you must guarantee that each pivot value partitions the input into roughly equal halves. We could use randomization to reduce the likelihood of the worst-case scenario occuring (in fact, randomization is a very effective strategy in practice), but this does not prevent us from experiencing the worst case altogether. In this section, we will discuss the **median of medians algorithm**, which can be used to select optimal pivots that prevent the worst case from happening at all. The algorithm is as follows:

1. Divide the elements into smaller groups of size five (there can be fewer than five elements in the last group if the total number of elements is not divisible by five; you may also drop this last group from consideration completely, since this step is only used to find a good pivot).[2]
2. Determine the median of each of these smaller subgroups, either by sorting or brute force (since the groups have at most five elements, we can consider sorting each group as a constant time process).
3. Store the medians of the subgroups in a separate array.
4. Use the quickselect algorithm to find the median of this array, recursively using the median of medians approach to choose the pivot.
5. Use the pivot obtained from the previous step to partition the original container of values. If more than half the elements are to the left of this pivot after the partition, the median must be smaller, so recursively run this process again on the elements to the left. If more than half the elements are to the right of this pivot after the partition, the median must be larger, so recursively run this process again on the elements to the right. If the pivot ends up at the index of the median, return its value (with logic to handle an even number of elements).

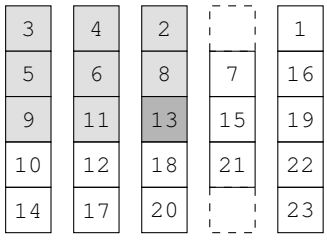
An illustration of this process is shown below. Here, 13 would be chosen as our pivot for partitioning.



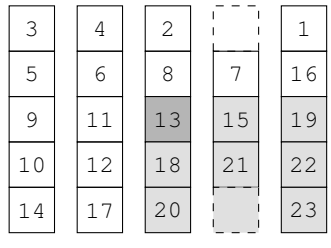
The median of medians approach allows us to select a reasonable pivot for each partition, but how from the worst case can this pivot be? It turns out that, if the median of medians strategy is used to select a pivot, we are guaranteed to remove 30% of the values with each partition. Let's take a look at why this is the case. Consider the example above, with the individual subgroups listed in order of their median values:



The values to the northwest of the median of medians cannot be larger, and the values to the southeast of the median of medians cannot be smaller.



These values cannot be larger than 13.



These values cannot be smaller than 13.

[2] The choice of five is intentional. We want an odd number, which makes median finding a lot more straightforward, and we also want the size to be as small as possible so that it is easier to find the median of each of the subgroups. If we divide the elements into groups of three, we do not end up with a worst-case time complexity of $\Theta(n)$. If we divide the elements into groups of seven or more, we do end up with a worst-case time complexity of $\Theta(n)$, but the coefficient overhead is larger since it is more expensive to process subgroups that are larger.

How many values must be in each of these two regions if there are n values in total? Since we are grouping elements into groups of five, there are a total of $n/5$ groups available. For half of these groups, three of its five values must be to the northwest or southeast of the median of medians. Therefore, the total number of values that are at least as large or small as the median of medians is:

$$\left(\frac{1}{2}n\right) \times \left(\frac{3}{5}\right) = \frac{3}{10}n$$

This indicates that at least $3/10$ of all the values *must be at least as extreme* as the median of medians. Thus, even in the worst case where the median of medians approach produces a pivot that is as far away from the median as possible, at least 30% of the values are still guaranteed to be even farther away (which allows them to be discarded after each partition).[3]

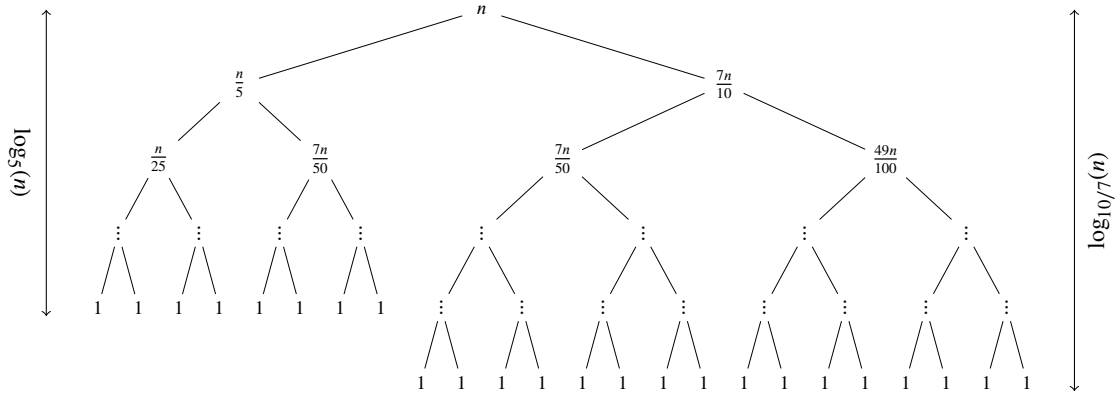
We can use this information to devise a recurrence relation that describes our median finding algorithm's performance if the worst-case pivot (30% away from the end) is chosen at every step. Notice that we perform the following steps at each iteration of the algorithm:

- Identify the median for each of the smaller groups of size five. Since each group has at most five elements, the complexity of finding the median for a single group is constant. There are a total of $n/5$ groups, so the total work done at this step is $n/5 \times \Theta(1) = \Theta(n)$.
- Store the medians of the subgroups into a separate array of size $n/5$ and recursively apply this algorithm to find the median of this array. This requires us to recursively solve a subproblem that is 1/5 the size of the original data range.
- Use the pivot obtained from the previous step to partition the original values. This takes $\Theta(n)$ time.
- After the pivot ends up in its correct position after partitioning, recurse into the side that contains the median. Since we are analyzing the worst case, this side includes 70% of all the values (since we proved that at least 30% of the values can be discarded at each partition). Thus, this step solves a subproblem that is 7/10 the size of the original data range.

The algorithm can therefore be expressed using the following recurrence relation:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$$

This recurrence relation ends up evaluating to $\Theta(n)$. We can see why by using a recursion tree to express the work done at each level of recursion:



On the top level, we do n work. On the next level, we do $\frac{n}{5} + \frac{7n}{10} = \frac{9n}{10}$ work. On the next level, we do $\frac{n}{25} + \frac{7n}{50} + \frac{7n}{50} + \frac{49n}{100} = \frac{81n}{100}$ work. In fact, this pattern can be used to show that the total work done cannot exceed:

$$T(n) = n + n\left(\frac{9}{10}\right) + n\left(\frac{9}{10}\right)^2 + n\left(\frac{9}{10}\right)^3 + \ldots + n\left(\frac{9}{10}\right)^{\log_{10/7}(n)}$$

This is because $n\left(\frac{9}{10}\right)^k$ work is done at the k^{th} level of recursion, and the longer recursive branch takes $\log_{10/7}(n)$ levels to reach the base case (notice that this solution is larger than the work actually done, since the left branch terminates after $\log_5(n)$ levels). Factoring out the n, we get

$$T(n) = n\left[\left(\frac{9}{10}\right)^0 + \left(\frac{9}{10}\right)^1 + \left(\frac{9}{10}\right)^2 + \left(\frac{9}{10}\right)^3 + \ldots + \left(\frac{9}{10}\right)^{\log_{10/7}(n)}\right]$$

Using the sum of an infinite geometric series, we know that the term multiplied by n must be a constant that is smaller than 10:

$$\sum_{i=0}^{\infty}\left(\frac{9}{10}\right)^i = \frac{1}{1-\frac{9}{10}} = 10$$

Therefore, $T(n)$ must be bounded by $\Theta(n)$. Since our analysis assumed that the worst possible pivot is chosen every time, we have successfully proved that the worst-case time complexity of finding the median using the median of medians approach is also $\Theta(n)$.

Even though the median of medians approach is asymptotically optimal, selecting a pivot randomly is almost always sufficient in practice. This is because computing the median of medians is not a trivial task, and the likelihood of ending up with a worst-case pivot every time is quite low, even for randomized pivot selections. Thus, for most use cases, the overhead involved in computing the median of medians is typically not a worthwhile tradeoff, even if it is asymptotically more efficient.

[3]Note that groups with fewer than five elements may affect this result, but only by a constant amount (since there are only at most five elements per group, and only one group may have fewer than five elements). Thus, this caveat does not actually change the outcome of our overall analysis.