Chapter 7 Practice Exercises

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

1. Which of the following statements is/are **TRUE**?
   - **I.** The actual elements in a `std::vector<>` are stored on the heap.
   - **II.** Inserting an element to the back of a `std::vector<>` always takes $\Theta(1)$ time.
   - **III.** Calling `.reserve()` on a `std::vector<>` never changes the vector's size.

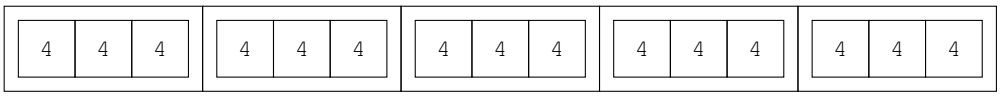
   A) I only
   B) II only
   C) I and II only
   D) I and III only
   E) I, II, and III

2. What is the worst-case time complexity of inserting an item at *any* position of a `std::vector<>`, assuming reallocation is needed, and assuming reallocation is not needed?
   A) $\Theta(1)$ if reallocation is needed, $\Theta(1)$ if reallocation not needed
   B) $\Theta(1)$ if reallocation is needed, $\Theta(n)$ if reallocation not needed
   C) $\Theta(n)$ if reallocation is needed, $\Theta(1)$ if reallocation not needed
   D) $\Theta(n)$ if reallocation is needed, $\Theta(n)$ if reallocation not needed
   E) None of the above

3. Suppose you are given a program that stores its primary data in a `std::vector<>`. The program executes correctly for small test datasets, but it runs out of memory while reading large datasets. What is a possible solution to overcome this problem?
   - **I.** Use `.resize()` with the exact number of elements, then use `.push_back()`.
   - **II.** Use `.resize()` with the exact number of elements, then use `operator[]`.
   - **III.** Use `.reserve()` with the exact number of elements, then use `.push_back()`.
   - **IV.** Use `.reserve()` with the exact number of elements, then use `operator[]`.

   A) I only
   B) II only
   C) I and IV only
   D) II and III only
   E) I, II, III, and IV

4. Suppose you wanted to instantiate a vector of vectors of integers. The outer vector holds five smaller vectors of integers, each of which individually holds three integer values all initialized to 4.



   Which one of the following expressions correctly initializes this vector of vectors?
   A) `std::vector<std::vector<int32_t>> vec(std::vector<int32_t>(5, 4), 3);`
   B) `std::vector<std::vector<int32_t>> vec(std::vector<int32_t>(3, 4), 5);`
   C) `std::vector<std::vector<int32_t>> vec(4, std::vector<int32_t>(5, 3));`
   D) `std::vector<std::vector<int32_t>> vec(5, std::vector<int32_t>(4, 3));`
   E) `std::vector<std::vector<int32_t>> vec(5, std::vector<int32_t>(3, 4));`

5. If the addition of a new element causes a vector to go over capacity, it has to reallocate. Which of the following is/are **TRUE** regarding the downsides of reallocation?
   - **I.** Reallocation can be expensive, especially if the elements to be copied are large.
   - **II.** Reallocation invalidates all existing pointers and iterators to elements in the old underlying array.
   - **III.** If you know the intended size of your container but do not reserve capacity beforehand, reallocation could result in wasted memory that is allocated but not used.

   A) I only
   B) III only
   C) I and II only
   D) I and III only
   E) I, II, and III

6. Suppose you have a container that doubles in capacity and copies all elements to a new location whenever an element is added while the container is at full capacity. You initialize the container's size to 12 and its capacity to 16. Which of the following operations would **NOT** result in a reallocation? Assume that all additional elements are inserted using `.push_back()`, and that no additional modifications to size or capacity are made.
   A) Inserting the 13th element into this container
   B) Inserting the 25th element into this container
   C) Inserting the 33rd element into this container
   D) Inserting the 49th element into this container
   E) More than one of the above

7. How many times does the vector allocate space in this code? Assume that, upon initialization, the vector's capacity is equal to its size.

```
1  int main () {
2    std::vector<int32_t> vec(2);
3    vec.reserve(4);
4    for (int32_t i = 0; i < 7; ++i) {
5      vec.push_back(i);
6    } // for
7  } // main()
```

   A) 2
   B) 3
   C) 4
   D) 5
   E) 6

8. Consider the following snippet of code:

```
1  int main () {
2    std::vector<int32_t> vec;
3    vec.reserve(10);
4    for (int32_t i = 0; i < 10; ++i) {
5      vec[i] = 281 + i;
6    } // for i
7    for (int32_t j = 0; j < vec.size(); ++j) {
8      std::cout << vec[j] << " ";
9    } // for j
10   std::cout << '\n';
11 } // main()
```

   Running the above code results in undefined behavior. Which line directly causes this behavior?
   A) Line 4
   B) Line 5
   C) Line 7
   D) Line 8
   E) None of the above

9. Consider the following snippet of code:

```
1  int main () {
2    std::vector<int32_t> vec;
3    for (int32_t i = 1; i < 10; ++i) {
4      vec.push_back(i);
5    } // for i
6    vec.resize(5);
7    vec.resize(8, 100);
8    vec.resize(9);
9    vec.reserve(10);
10   vec[9] = 10;
11   for (size_t i = 0; i < vec.size(); ++i) {
12     std::cout << vec[i] << ' ';
13   } // for i
14   std::cout << '\n';
15 } // main()
```

   Running the above code results in undefined behavior. Which line directly causes this behavior?
   A) Line 3
   B) Line 6
   C) Line 7
   D) Line 10
   E) None of the above

10. Consider the following snippet of code:

```
1  int main () {
2    std::vector<int32_t> vec = {1, 2, 3, 4, 5};
3    vec.reserve(6);
4    int32_t* ptr = &vec[1];
5    vec.push_back(6);
6    vec.push_back(7);
7    std::cout << *ptr + vec[3] << std::endl;
8  } // main()
```

What does this code print?

A) `4`
B) `5`
C) `6`
D) `7`
E) Impossible to determine, since this is undefined behavior

11. Consider the following snippet of code:

```
1  int main () {
2    std::vector<int32_t> v1;
3    std::vector<int32_t> v2;
4    for (int32_t i = 281; i < 381; i += 10) {
5      v1.push_back(i);
6    } // for i
7    v1.resize(6);
8    v2.resize(6);
9    for (int32_t j = 482; j < 494; ++j) {
10     v2.push_back(j);
11   } // for j
12   v1.resize(v2.size(), v2.size());
13   for (int32_t k = 1; k < val; k += 4) {
14     std::cout << v1[k] + v2[k] << ' ';
15   } // for k
16 } // main()
```

What does this code print?

A) `281 321 484 488 492`
B) `281 321 502 506 510`
C) `291 331 497 501 505`
D) `291 331 503 507 511`
E) `291 331 504 508 512`

12. Consider the following snippet of code:

```
1  int main () {
2    std::vector<int32_t> v1{2, 5, 1, 3, 4};
3    std::vector<int32_t> v2;
4    for (size_t i = 0; i < v1.size(); ++i) {
5      v2.resize(v1[i]);
6      v2.back() = i;
7    } // for i
8    for (auto num : v2) {
9      std::cout << num << ' ';
10   } // for num
11 } // main()
```

What does this code print?

A) `2 0 3 4`
B) `2 0 3 4 1`
C) `2 5 1 3`
D) `2 5 1 3 4`
E) `Segmentation fault`

13. Assume that you have a vector that, when its size is equal to its capacity and a new element is appended using `.push_back()`, all of the elements are copied to a new block of dynamic memory with a new capacity that is twice the previous capacity. When the capacity is 0 and a new element is added, the new container's capacity becomes 1. Suppose you push back n elements into a vector with initial size 0. What is the total number of times that an elements is copied from an old block of dynamic memory to a new one?

A) $n-1$
B) $2\times\sum_{i=0}^{\lfloor\log_2(n-1)\rfloor} i$
C) $\sum_{i=0}^{\lfloor\log_2(n-1)\rfloor} 2$
D) $2\times\sum_{i=0}^{\lfloor\log_2(n-1)\rfloor} i^2$
E) $\sum_{i=0}^{\lfloor\log_2(n-1)\rfloor} 2^i$

14. You are reading in a text file with 1,000,000 words. Because you know in advance how many words you are going to read in, you reserve a vector of strings with a capacity of 1,000,000. However, you make a mistake when reading the file and accidentally insert an additional empty string into the vector after pushing in all 1,000,000 words. Which of the following statements is **FALSE**? Assume that the implementation of this vector is the same as the vector detailed in the previous question.
    A) Adding in this empty string would cause the vector's memory usage to be roughly twice the memory it would have used had the empty string not been pushed in
    B) Adding in this empty string would require your program to copy or move 1,000,000 strings that you wouldn't have needed to copy or move otherwise
    C) Adding in this empty string would cause the vector's size to double to 2,000,000 at the end of the program, assuming that no additional elements are pushed in
    D) If you had initialized any iterators or pointers to the vector's data prior to pushing in the empty string, all these iterators/pointers would have been invalidated after pushing in the empty string
    E) None of the above

15. Which one of the following statements is **FALSE** regarding the `std::vector<>`?
    A) Inserting new elements into a vector can invalidate pointers and iterators pointing to existing elements in the vector
    B) Calling `.resize()` on a vector can invalidate pointers and iterators pointing to existing elements in the vector
    C) Calling `.reserve()` on a vector can invalidate pointers and iterators pointing to existing elements in the vector
    D) If `v` is a vector and `i` is a valid index in `v`, then `&v[i] == &v[0] + i` will always evaluate to `true`
    E) None of the above

16. Consider the following snippet of code:

```
1   class Thing {
2     int32_t* x;
3     Thing(int32_t x_in) : x{new int32_t{x_in}} {}
4     ~Thing() { delete x; }
5   };
6
7   int main() {
8     int32_t arr[5];
9     std::vector<int32_t> v1;
10    std::vector<Thing> v2;
11    std::vector<int32_t*> v3;
12    std::vector<Thing*> v4;
13    for (int32_t i = 0; i < 5; ++i) {
14      arr[i] = i;
15      v1.push_back(i);
16      v2.push_back(Thing{i});
17      v3.push_back(&arr[i]);
18      v4.push_back(new Thing{i});
19    } // for i
20  } // main()
```

    Does this code leak memory, and if so, what is the cause of the leak?
    A) Yes, the integers pointed to by `x` in the `Thing` objects of `v2` are leaked
    B) Yes, the integers pointed to by the elements of `v3` are leaked
    C) Yes, the `Thing` objects pointed to by the elements of `v4` are leaked
    D) More than one of the above
    E) No memory is leaked after the vector destructors are run

17. You want to store a four-dimensional data matrix in the form of a `std::vector<>`. You know that your data has dimensions of $203 \times 281 \times 183 \times 280$. If you want to minimize the amount of memory needed to create the 4-D vector (i.e., a vector of vector of vector of vectors), what should the dimension of the outermost vector be?
    A) 203
    B) 281
    C) 183
    D) 280
    E) It does not matter, since the memory required to create the vector will always be the same

18. Suppose you have a vector containing n elements. What is the overall time complexity of appending n more elements, if the vector always reallocates its underlying array with each new element?
    A) $\Theta(1)$
    B) $\Theta(n)$
    C) $\Theta(n\log(n))$
    D) $\Theta(n^2)$
    E) $\Theta(n^3)$

19. You are given a vector of integers `nums`. Implement a function that returns the smallest index `i` of `nums` such that the last digit of `nums[i]` matches the last digit of its index `i` (i.e., `nums[i]` mod 10 == `i` mod 10). If no such index exists, return `-1`. For example, given the following vector:

32	46	10	53	24	67	98
0	1	2	3	4	5	6

you would return `3`, since that is the first index whose last digit matches the last digit of its value (e.g., 5**3** at index **3**).

```
int32_t smallest_equal_index(const std::vector<int32_t>& nums);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(1)$ auxiliary space, where n is the size of the vector. Note that you can use `operator%` to calculate a modulus (e.g., `53 % 10` = `3`).

20. You are given an $n \times n$ square matrix of integers in the form of a vector of vectors. Implement a function that returns the sum of the values on the matrix diagonals (each value is only counted once, even if it is on two diagonals). For example, given the following matrix:

1	2	3
4	5	6
7	8	9

you would return `25`, since that is the sum of the diagonals ($1 + 3 + 5 + 7 + 9 = 25$).

1	2	**3**
4	**5**	6
7	8	**9**

```
int32_t matrix_diagonal_sum(const std::vector<std::vector<int32_t>>& matrix);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(1)$ auxiliary space, where n is the size of a matrix dimension.

21. In linear algebra, a Toeplitz matrix is a matrix where each descending diagonal from top-left to bottom-right contains the same value. For example, the following matrix is a Toeplitz matrix because each top-left to bottom-right diagonal consists of the same elements.

1	2	3	4
0	1	2	3
2	8	1	2

1	**2**	3	4
0	**1**	**2**	3
2	8	**1**	**2**

Given a $m \times n$ matrix in the form of a vector of vectors, implement a function that returns whether the matrix is a Toeplitz matrix or not. If a matrix is Toeplitz, return `true`; otherwise, return `false`.

```
bool is_toeplitz(const std::vector<std::vector<int32_t>>& matrix);
```

You should implement your solution in worst-case $\Theta(mn)$ time and $\Theta(1)$ auxiliary space, where m and n are the dimensions of the matrix.

22. You are given an unsorted vector of n integers, where n is an odd number. You are told that all but one of the values also have their additive inverse inside the vector (e.g., 5 and -5 are additive inverses). Implement a function that returns the value without an additive inverse in the vector. For example, given the following vector:

4	8	-3	-2	-8	5	-5	-4	3
0	1	2	3	4	5	6	7	8

you would return `-2`, since that is the only value in the vector whose additive inverse (in this case, 2) is also not in the vector.

```
int32_t value_without_additive_inverse(const std::vector<int32_t>& nums);
```

You should implement your solution in worst-case $\Theta(n)$ time and $\Theta(1)$ auxiliary space, where n is the size of the vector.

Chapter 7 Exercise Solutions

1. **The correct answer is (D).** Only statements I and III are true. Statement I is true because the data in a vector are stored in an underlying array that is allocated on the heap. Statement III is true because `.reserve()` only affects the vector's capacity, and not its size. Statement II is false because you may have to reallocate if the new element causes the vector to go over capacity, which would take linear time.

2. **The correct answer is (D).** The worst-case time complexity of insertion assuming reallocation is $\Theta(n)$, since you would have to move all the existing data to a new underlying array. Without reallocation, however, the worst-case time complexity could still be $\Theta(n)$ if the insertion takes place at the front of the vector, which would require all subsequent elements to be shifted over by one position.

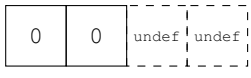
3. **The correct answer is (D).** When you call `.resize()` on a vector, you actually change the number of elements it contains. On the other hand, if you call `.reserve()`, you only reserve space for more elements, but you do not create them. Hence, you cannot use `.push_back()` on a vector that has been resized, since you would be adding elements on top of the elements you already created using `.resize()` (i.e., if you want 20 elements in your vector and nothing more, resizing the vector would initialize 20 elements in the vector, and using `.push_back()` would add a 21st element instead of modifying the 1st element). Meanwhile, after reserving space for a vector to hold more elements, you still must add the elements themselves. For instance, if you reserve a vector to hold 20 elements, it now has the capacity to hold 20 elements, but it does not actually have any elements yet (similar to how an empty 5-gallon bucket *can* hold 5 gallons of water, but it does not actually hold any water). Thus, you cannot use `operator[]` on a vector that only has space reserved, as the elements you want to create do not exist yet. The only ways to ensure that a vector has the number of elements you want it to have are to either resize the vector and modify each element using `operator[]` or reserve space for the vector and add the elements using `.push_back()`.

4. **The correct answer is (E).** When initializing a vector using a fill constructor, the first argument is the intended size of the vector, and the second argument is the value you want initialized for each of the elements in the vector. Only choice (E) works here: you are initializing a vector of size 5 (first argument) where each element is initialized to a vector of integers (second argument). This inner vector is given a size of 3 (first argument), where each element is initialized to 4 (second argument).

5. **The correct answer is (E).** All of the statements are true. Statement I is true because elements must be copied from one array to another during reallocation, which can be an expensive process. Statement II is true because all pointer or iterator references to the old array would be invalidated when the elements are copied to a new, larger array. Statement III is true because you may end up with a larger capacity than you need, which would waste memory (e.g., you only need to store 65 elements, but your vector ends up with a capacity of 128 since capacity is doubled with each reallocation, leaving 63 slots unused).

6. **The correct answer is (C).** The vector reallocates with double the capacity when you try to insert an element that exceeds its capacity. Currently, the capacity is 16, so a reallocation is triggered when you insert the 17th element, which doubles the capacity to 32. The next reallocation happens when you try to insert the 33rd element, which doubles the capacity to 64. The next reallocation happens when you try to insert the 65th element, which doubles the capacity to 128 (and so on).

7. **The correct answer is (C).** The vector allocates space four times. The first allocation occurs on line 2, where the vector's underlying array is initially set to a size of 2 (with the values initialized to zero). The second allocation occurs on line 3, since the requested capacity of 4 is larger than the existing capacity of 2. This is what the vector looks like after the second allocation:



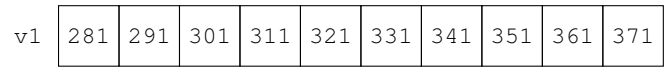
   Next, the loop on line 4 inserts 7 elements to the back of the vector. Pushing back the 3rd of these elements forces a reallocation since there would be 5 elements after the insertion, but only an existing capacity of 4. This reallocation doubles the vector's capacity to 8. Following this, the insertion of the 7th element in this loop increases the size of the vector to 9, which forces another reallocation of the vector from a capacity of 8 to a capacity of 16.

8. **The correct answer is (B).** The `.reserve()` call on line 2 reserves the vector's capacity to 10, but it does not actually create any elements. Thus, trying to index into the values on line 5 causes undefined behavior, since you cannot index a vector with a position that is larger than its size (which is still zero).

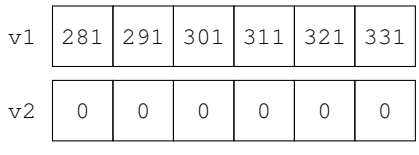
9. **The correct answer is (D).** On line 10, the vector only has a size of 9 (from the resize on line 8). Since vectors are zero-indexed, there is no value at index 9, so the use of `operator[]` on line 10 results in undefined behavior.

10. **The correct answer is (E).** It is impossible to determine what line 7 prints, since `ptr` is invalidated when the vector's underlying array is forced to reallocate on line 6 (since a 7th element is added when the capacity is 6).

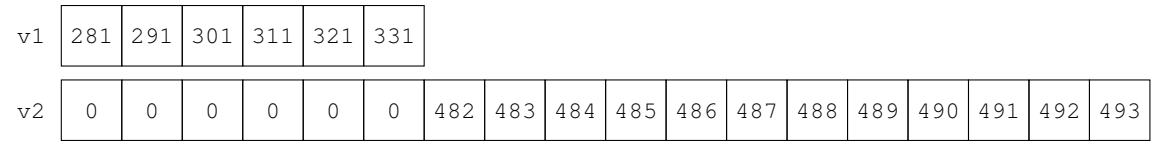
11. **The correct answer is (D).** The loop on line 4 pushes back all numbers from 281 to 381 (exclusive), incrementing by 10 each time. After the loop runs to completion, the contents of `v1` are:



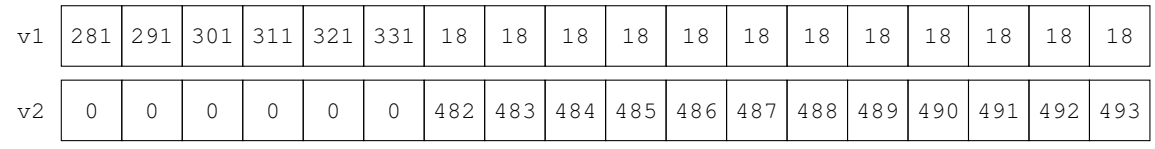
   On lines 7 and 8, both `v1` and `v2` are resized to a size of 6. This removes all but the first six elements in `v1` and initializes six elements in `v2` with a default value of zero.



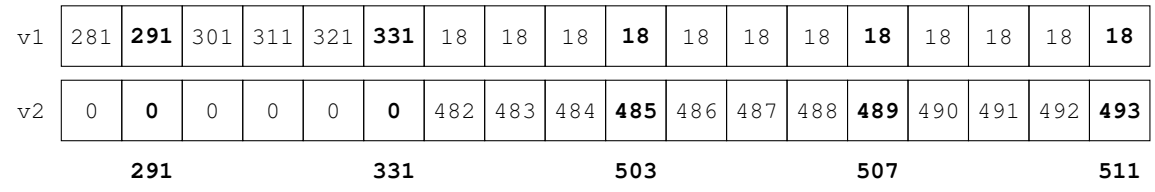
The loop on line 9 pushes back all values from 482 to 494 (exclusive) to the back of `v2`.



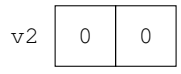
Line 12 resizes `v1` to the same size as `v2`, with values initialized to the size of `v2` (which is 18).



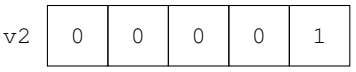
The loop on line 13 then prints out the sum of the two vectors at indices 1, 5, 9, 13, and 17.



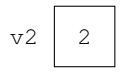
12. **The correct answer is (A).** For each index `i` in the loop on line 4, `v2` is resized to a value of `v1[i]` and has its last element assigned to `i`. On the first iteration, `v2` is resized to a size of `v1[0]` or 2, and the last element is assigned to a value of 0.



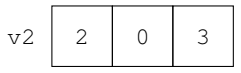
On the next iteration, `v2` is resized to a size of `v1[1]` or 5, and the last element is assigned a value of 1.



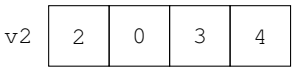
On the next iteration, `v2` is resized to a size of `v1[2]` or 1, and the last element is assigned with a value of 2.



On the next iteration, `v2` is resized to a size of `v1[3]` or 3, and the last element is assigned with a value of 3.



On the next and final iteration, `v2` is resized to a size of `v1[4]` or 4, and the last element is assigned with a value of 4.



The contents of this vector are then printed out.

13. **The correct answer is (E).** When the 2nd element is inserted, the vector reallocates and 1 element is copied over. When the 3rd element is inserted, the vector reallocates and 2 elements are copied over. When the 5th element is inserted, the vector reallocates and 4 elements are copied over. When the 9th element is inserted, the vector reallocates and 8 elements are copied over. Since $\log_2(n-1)$ reallocations are done if n elements are inserted without explicitly reserving capacity (since that is the number of times the capacity can double before you can support n elements), the total number of times an element is copied can be expressed using the summation $1+2+4+\ldots+2^{\lfloor \log_2(n-1) \rfloor}$, which matches option (E).

14. **The correct answer is (C).** Options (A), (B), and (D) are all true, since these are consequences of vector reallocation, which would happen if you pushed in a 1,000,001st string into a vector whose capacity is 1,000,000. Option (C) is incorrect because the size of the vector is not doubled with reallocation (as with capacity), so the size of the vector would end up being 1,000,001 instead of 2,000,000.

15. **The correct answer is (E).** All of the statements are true. Options (A), (B), and (C) could all result in a reallocation if the new requested size or capacity exceeds the existing capacity, which would result in pointer and iterator invalidation. Option (D) is true because elements in a vector are stored contiguously in memory, allowing us to use pointer arithmetic to identify the position of elements in the vector.

16. **The correct answer is (C).** Even though vectors store their values on the heap, they are responsible for cleaning up all memory that they own. Thus, the values in `v1` and `v2` are automatically cleaned up their respective vector is destructed. The pointers in `v3` refer to the values in `arr`, which are automatically cleaned up as well when `arr` goes out of scope. This is not the case for the values in `v4`, however, since they are explicitly allocated on the heap by the programmer. Since the programmer created these `Thing` entities using `new`, they are also responsible for cleaning them up using `delete`; this is not done, so the `Thing` objects in `v4` are leaked.

17. **The correct answer is (C).** If you have a four-dimensional vector that is declared in the order a, b, c, and d, the total number of bookkeeping pointers needed can be estimated using $3+3a+3ab+3abc$. Thus, to reduce the total number of pointers you need to keep track of, you should declare the outermost vector (in this case, the one with dimension a) using the size corresponding to the smallest dimension, which in this case is 183.

18. **The correct answer is (D).** If the vector always reallocates with each new element added, the fist new element added would force the existing n elements to be copied; the second new element added would force the existing $n+1$ elements to be copied; and so on. From this, you can see that each new insertion would take $\Theta(n)$ time for the old elements that need to be copied during reallocation. Since we are appending n new elements, this process is done a total of n times, for an overall time complexity of $n \times \Theta(n) = \Theta(n^2)$.

19. This problem can be solved using an iteration of the vector, checking each value to see if its last digit matches its index. Once we encounter such a value, we can return its index immediately. If we finish iterating the entire vector without encountering a value whose last digit matches its index, we would return `-1`. Since we only need to iterate over the vector of n elements once, and we perform a constant operation at each element (checking its last digit with its index), this solution takes $\Theta(n)$ time.

```
int32_t smallest_equal_index(const std::vector<int32_t>& nums) {
  for (size_t i = 0; i < nums.size(); ++i) {
    if (i % 10 == nums[i] % 10) {
      return i;
    } // if
  } // for i

  return -1;
} // smallest_equal_index()
```

20. One strategy for solving this problem is to use a nested `for` loop, one that iterates over the rows and one that iterates over the columns, and add the values where the row and column index identify a diagonal. However, that would take $\Theta(n^2)$ time, which exceeds the time complexity specified by the problem. Instead, we can implement a linear-time solution by only iterating over the rows and using arithmetic to determine which two values in that row are on a diagonal. For each row index i, the value on a diagonal must be located at column i and $n-i-1$. One thing to consider is the possibility of duplicating the center value if the dimension of the matrix is odd, so we will also need to subtract the center value in this scenario. One potential implementation is shown below:

```
int32_t matrix_diagonal_sum(const std::vector<std::vector<int32_t>>& matrix) {
  int32_t sum = 0;
  for (size_t i = 0; i < matrix.size(); ++i) {
    // value on top-left to bottom-right diagonal
    sum += matrix[i][i];
    // value on top-right to bottom-left diagonal
    sum += matrix[i][n - i - 1];
  } // for i

  // if dimension is odd, subtract center value so it is not double-counted
  if (matrix.size() % 2 == 1) {
    return sum - matrix[n / 2][n / 2];
  } // if

  return sum;
} // matrix_diagonal_sum()
```

21. One strategy for solving this problem is to use a nested `for` loop, one that iterates over the rows and one that iterates over the columns, and check if `matrix[i][j]` equals `matrix[i - 1][j - 1]` for all `i` and `j` from 1 to $n-1$. One implementation is shown below:

```
bool is_toeplitz(const std::vector<std::vector<int32_t>>& matrix) {
  for (size_t i = 1; i < matrix.size(); ++i) {
    for (size_t j = 1; j < matrix[i].size(); ++j) {
      if (matrix[i][j] != matrix[i - 1][j - 1]) {
        return false;
      } // if
    } // for j
  } // for i

  return true;
} // is_toeplitz()
```

22. One solution is to iterate over the vector and keep track of a set that stores the values for which an additive inverse has not been encountered. However, the easiest solution can be obtained with the following insight: since a number and its additive inverse must sum to zero, we can find the value without an additive inverse by just summing up the entire vector (since all other additive inverse pairs cancel each other out). Thus, a sum of the values would give us our solution, which takes $\Theta(n)$ time.

```
int32_t value_without_additive_inverse(const std::vector<int32_t>& nums) {
  int32_t sum = 0;
  for (auto num : nums) {
    sum += num;
  } // for num

  return sum;
} // value_without_additive_inverse()
```