Homework 2: LogNVector

Sarah Youngquist

October 6, 2021

Theoretical Big-O

Time Complexity for `LogNVector` Methods

Big-O of `LogNVector::push_back`

By design, in most cases, adding to the vector is $O(n)$ since copying is minimized in `LogNVector`. Copying only occurs when the container `std::vector` resizes. `std::vector` resizes when adding the $(2^i + 1)$th element. This means that `LogNVector`'s container `std::vector` is copied when adding the 2^{2^i+1}th element.

Since `array_` is an array of pointers to regular arrays, only the pointers are copied. This means the number of copies is equal to the length of `array_`, which is $\boxed{O(\log n).}$

Other Methods

The copy constructor and the list initializer constructor use `reserve()` for `array_`, preventing any copying from occurring, making the methods $O(1)$. All other methods are trivially $O(1)$.

Space Complexity for `LogNVector`

I may be confused for this question, but in general, the space complexity of a container is $O(n)$ since the biggest use of space is the elements being stored themselves.

The length of `arrays_` is $O(\log n)$. `arrays_` stores pointers, but the arrays pointed to still take up memory. There is never a level of copying that occurs that would make space anything other than $O(\log n)$.

Comparison to `std::vector`

The big-O of `std::vector::push_back` is $O(n)$ since copying occurs every time the array doubles. This happens whenever adding the $(2^i + 1)$th element. This means that theoretically, `LogNVector::push_back` is significantly more efficient.

The big-O for space and for all other methods is the same.

Analysis of `LogNVector` in Practice

I compared `LogNVector` to `std::vector` for all of my analysis.

Comparison for `push_back`

`push_back`'s time was nearly identical for both except when adding the 2^ith element and the $(2^i + 1$th element. For the 2^ith element, `LogNVector` is significantly slower because it needs to allocate memory for the new array. It is especially slow for the 2^{2^i+1}th elements since that's when copying occurs. For the $(2^i + 1)$th elements, `std::vector` is much slower because copying occurs.

When it is faster, `LogNVector` is often much, much faster than `std::vector`, and the ratio is roughly proportional to n. When `std::vector` is faster, it is not as much faster. This makes `LogNVector` generally faster.

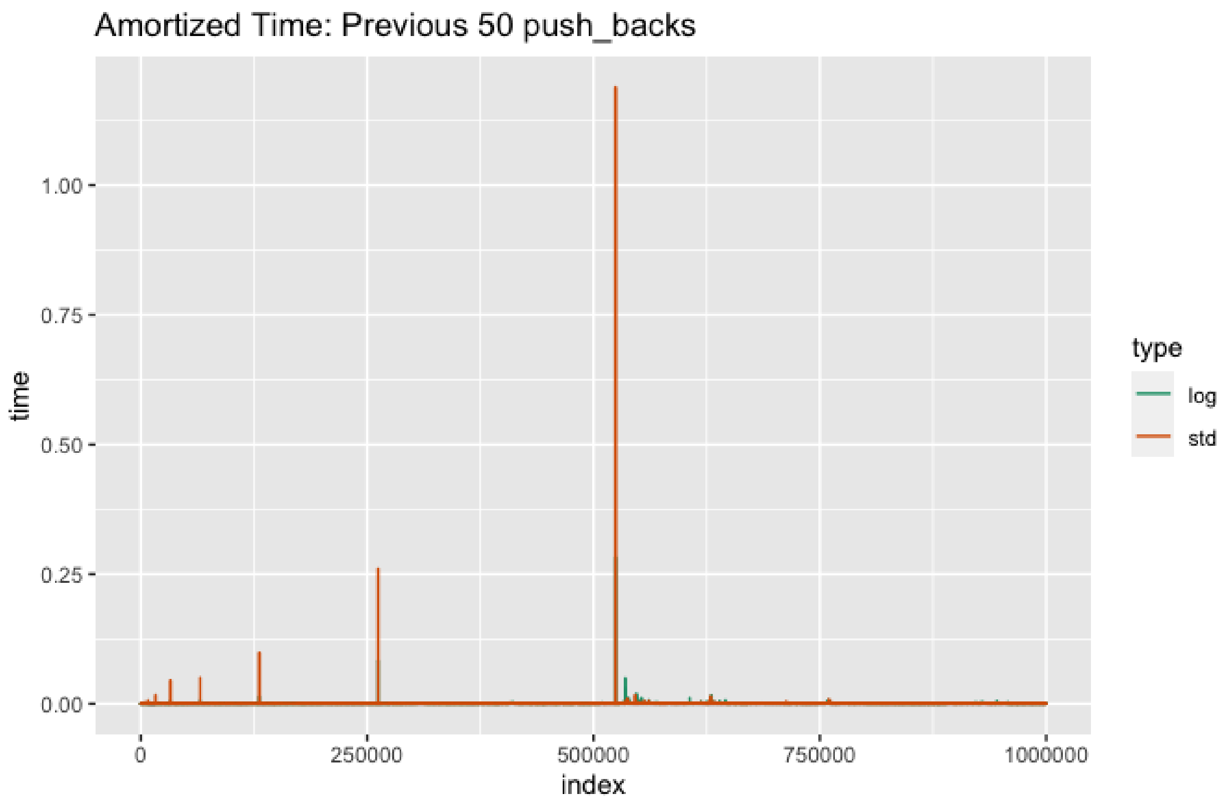

Figure 1: Time for the previous 50 `push_back` calls

This graph shown looks at the times for the previous 50 `push_back` calls. As can be seen, there are huge jumps every power of two for `std::vector`, and that is the main thing the graph shows. This is much less the case for `LogNVector`

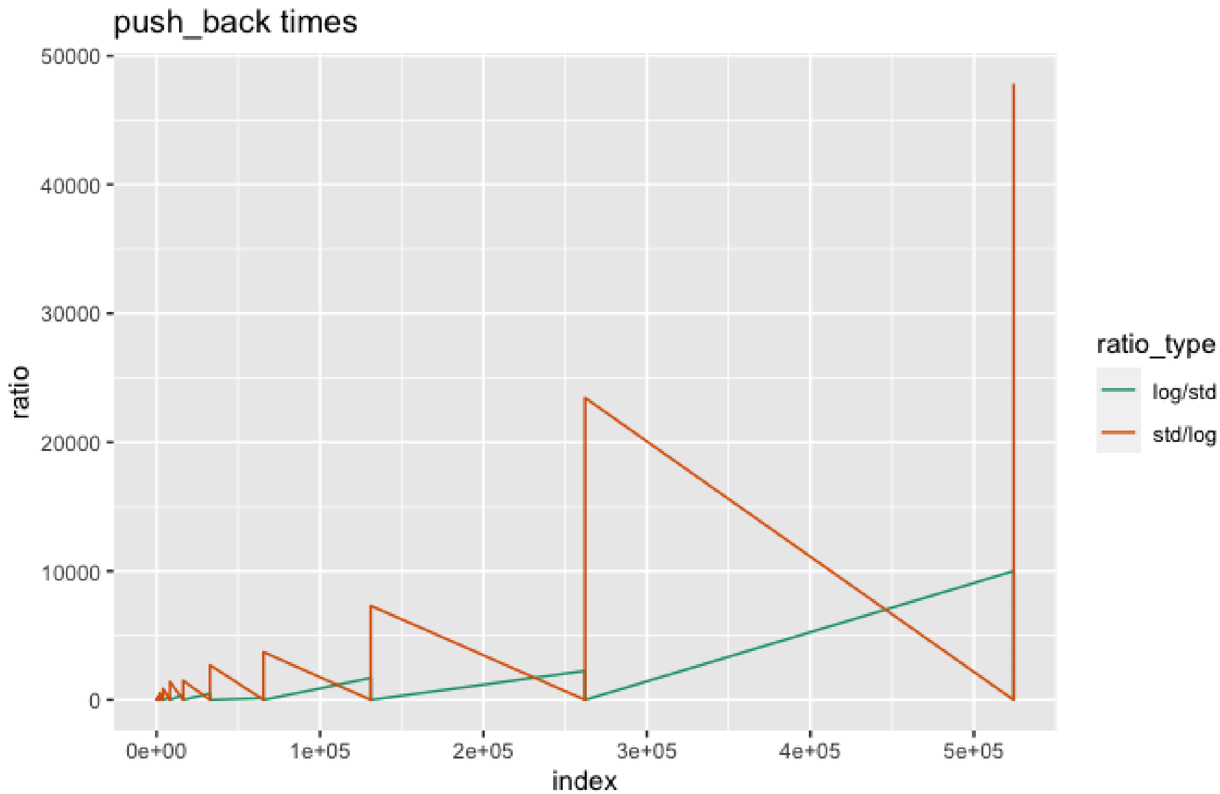

Figure 2: Comparing time for `push_back` for powers of 2

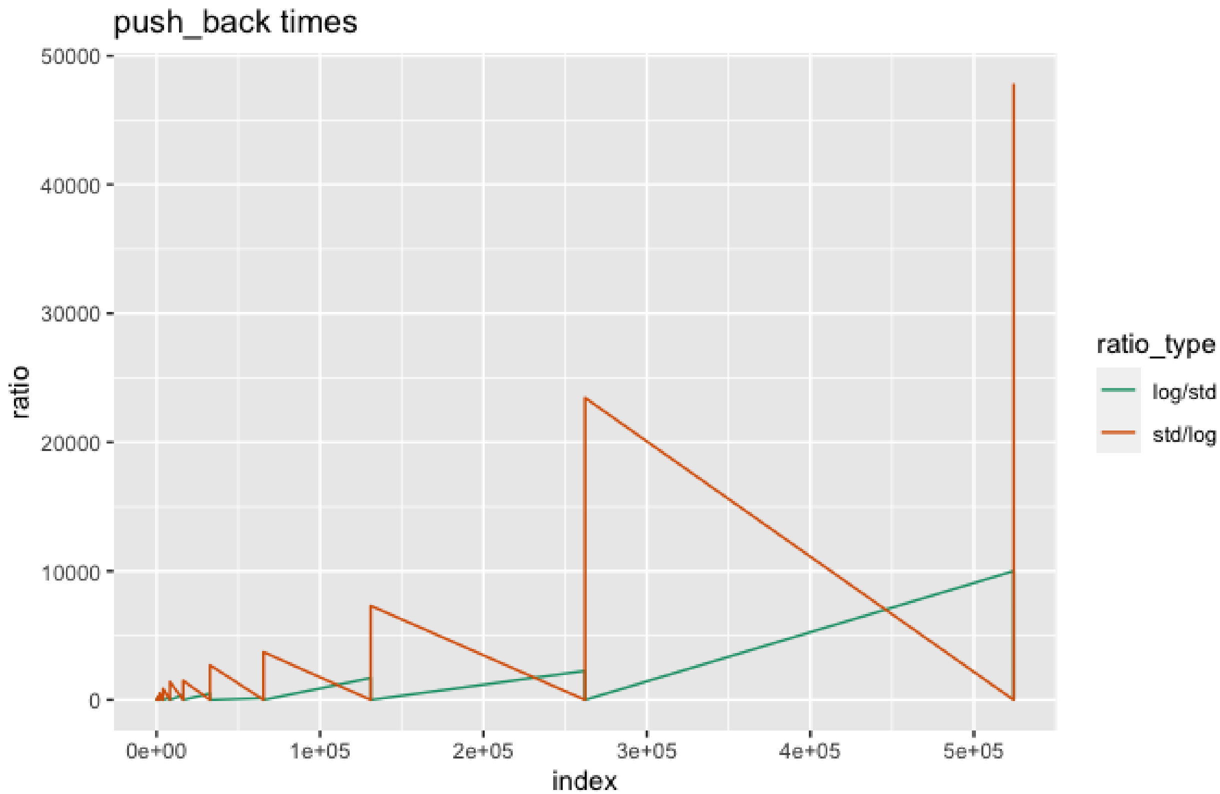

Figure 3: Above without largest time

This graph plots in blue the ratio of time for `LogNVector` to `std::vector` and in red the reciprocal, only for adding the 2^ith and $(2^i + 1)$th elements. These are the ones where the two differ greatly. These are the worst-case scenarios, so the graph allows seeing big-O more clearly.

The second graph is the same as the first but omits the one value with the huge time for better visibility.

Comparison for `operator[]`

There is no difference in big-O, but in practice, there should be a difference in performance because calculations are needed to get the index in `LogNVector::operator[]`. The margin should be a relatively constant factor.

I tested the access time to the ith element for `LogNVector` and `std::vector`. The data I analyzed was the time for `LogNVector` divided by the time for `std::vector` for index values 0 through 1,000,000.

Descriptive Statistics for `operator[]`

Mean	8.72
Std Dev	12.00
Q1	8.47
Median	8.58
Q3	8.71

There were several outliers on the high end. I could find no discernible pattern in the outlier indices, so I am guessing it is just random error. Most values fell in a very small range, as seen by the values for Q1 and Q3.

Random access in `LogNVector` is approximately a constant factor of 8.5 times slower than random access in `std::vector`. For reference, the median time for random access was 2.04e-06 for `LogNVector` and 2.39e-07 for `std::vector` on my (slow) computer.

Advantage of Bit-Based Calculations

The number of arrays needed for `LogNVector` is equal to $\lfloor \log_2 n \rfloor + 1$, which is equal to the number of bits needed to represent n. For some reason, `std::bit_width` did not work on my computer, so I used:
`std::numeric_limits<unsigned int>::digits - std::countl_zero((unsigned int) n)`
I also used left shifts to calculate powers of two.

I was interested to see how much time this actually saved, so I timed `push_back` using bit-shifts and bit width and without. I timed inserting 10,000, 100,000, and 1,000,000 elements with 500 repetitions (so all times are multiplied by 500).

Results for `push_back`

Size	Bit Operations	No Bit Operations
10000	0.376	0.744
100000	3.49	5.22
1000000	36.41	57.10

Without bit operations, it consistently takes 50% longer. This difference was significantly larger than I expected. (I did compare the speed of the operations using bits versus without, and the bit operations were usually around 10 times faster, but I was guessing that the calculation time was not such a significant fraction.)

These numbers also demonstrate that the time needed for n insertions is roughly $O(n)$, as expected given the small amortized big-O.

Instantiations

I made a simple class that kept track the number of times it had been instantiated. In `std::vector`, for n elements, the number of times was exactly n. However, for `LogNVector`, for n elements, the number of instantiations was around 2.3 times the number of elements.

This is because when my code adds the next array to `arrays_`, the array is populated with default objects that call the default constructor. These are then replaced by the actual added objects. This does not happen for `std::vector`, and I am not sure how to change my code to make that not happen.

Since those added elements quickly go out of scope when they are replaced, this is not much of a problem unless the objects being stored are really, really big, which is almost never the case.

Code and Data

Code Excerpt

Listing 1: Timing `push_back` by Index

```
std::vector<std::pair<double, double>> time_insert_by_index(int maximum, int
    reps)
{
  std::vector<std::pair<double, double>> times;

```

```cpp
  std::vector<LogNVector<int>> log_n_vectors;
  std::vector<std::vector<int>> std_vectors;
  // one vector for reach rep
  for (int i = 0; i < reps; ++i)
  {
    log_n_vectors.push_back(LogNVector<int>());
    std_vectors.push_back(std::vector<int>());
  }
  // find time for each index
  for (int index = 0; index < maximum; ++index)
  {
    double log_n_time, std_time;
    // LogNVector
    SimpleTimer timer;
    timer.start();
    for (int i = 0; i < reps; ++i)
    {
      log_n_vectors[i].push_back(index);
    }
    log_n_time = timer.elapsed_seconds();
    // std::vector
    timer.start();
    for (int i = 0; i < reps; ++i)
    {
      std_vectors[i].push_back(index);
    }
    std_time = timer.elapsed_seconds();
    times.push_back({log_n_time, std_time});
  }
  return times;
}
```

Full Code and Raw Data

Full code except for the actual `log_n_vector.h` because that is the answer to a homework assignment: https://github.com/sarah829/cs_2c-stats-analysis/tree/main/log_n_vector