Homework 1: Compromise Search

Sarah Youngquist

September 29, 2021

Big-O

Big-O of `std::find`

`std::find` is a linear search, as in it just iterates from the given start to the given end. Since in the worst case scenario, the iterator would need to traverse the whole list, `std::find` is $\boxed{0(n).}$

Big-O of `std::binary_search`

In a binary search, the range needed to be searched halves each time. In the worst case scenario, the value searched for is either not in the list or it is only selected as the middle item when it is the last item left.

Each iteration takes $O(1)$ steps since it does not depend on the size, so the number of iterations determines the order. Since the range is halved each time, the worst case scenario is $\log_2 n$ steps. So, `std::binary_search` is $\boxed{O(\log n).}$

Big-O of `compromise_search`

In compromise search, the range halves each time until it is smaller than `small_size` (which I will denote as s). In the worst case scenario, the number of times the range needs to be halved (k) is determined by:

$$\begin{aligned}
&\frac{n}{2^k} = s \\
&\frac{n}{s} = 2^k \\
&\log_2 \frac{n}{s} = k \\
&k = \log_2 n - \log_2 s
\end{aligned}$$

The number of steps per iteration is $O(1)$ since it does not depend on n or s.

After that part of the search, a linear search is done on the range of length at most s. This is $O(s)$.

Combining the two, `compromise_search` is $\boxed{O(s + \log n).}$ (Note that in this, meaningfully s maxes out at n since any higher is still the same linear search.) In n, it is $O(\log n)$ which is the case when s is small.

Analysis of Compromise Search

I generated results over several trials and different searches. I tested a bunch of sizes up to 200,000 and small sizes up to 250 (not all sizes or small sizes).

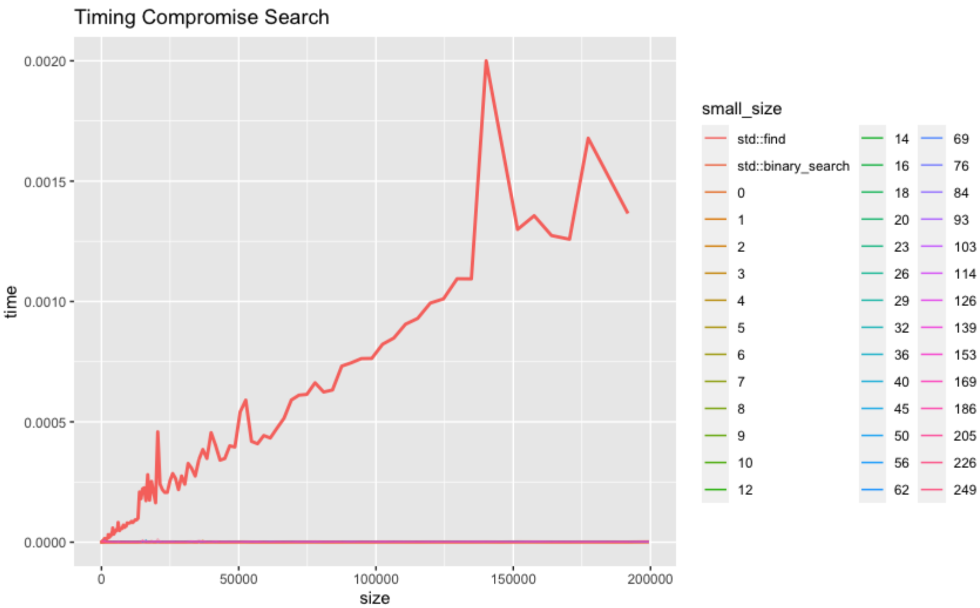

Figure 1: The linear search (`std::find`) is, as expected, the slowest

`std::find` makes the the graph pretty useless except for observing how comparatively slower $O(n)$ is versus $O(\log n)$. As such, the rest of the graphs exclude `std::find` .

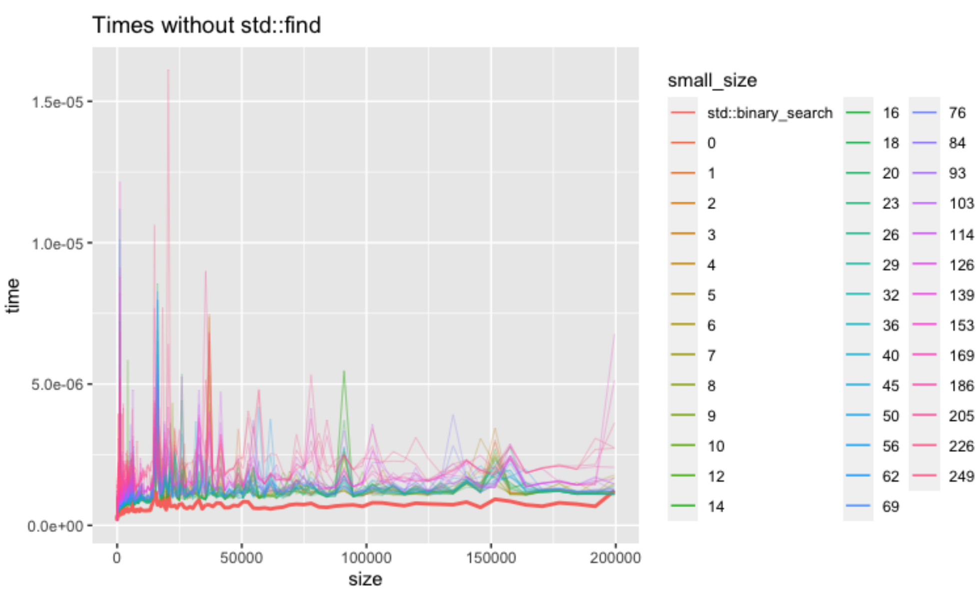

Figure 2: Times without linear search

Now that I can actually see both `std::binary_search` and `compromise_search` at the same time, it is clear they are of the same order. This means that all of the answers for the big-O of the searches were correct.

This also shows that binary search is better than compromise search for any `small_size` , including when `small_size = 0` , meaning that the C++ STL is more efficient than I am :(

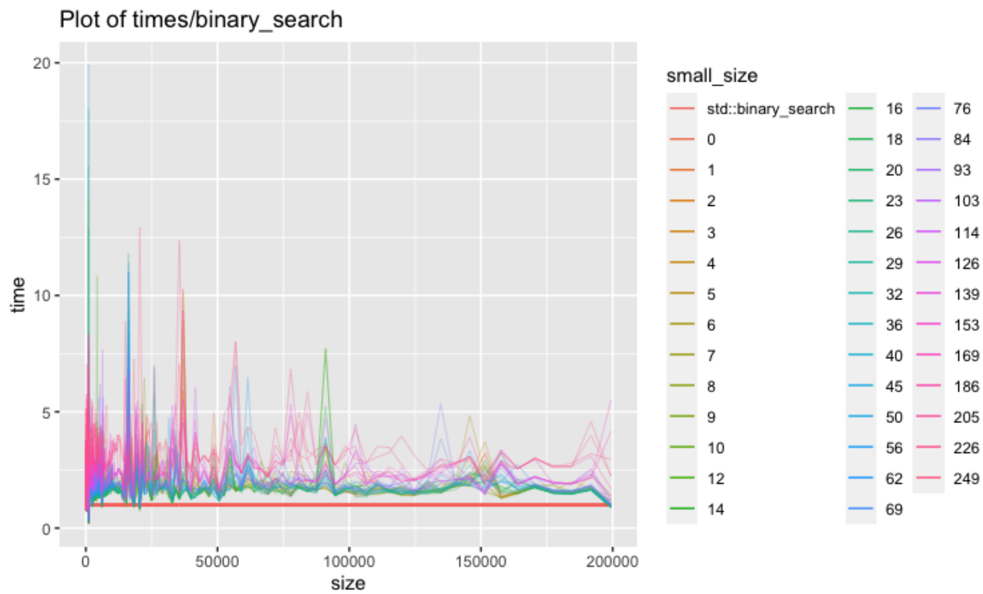

Figure 3: Times as a ratio to `std::binary_search`

Next, I compared `compromise_search` to a binary search by dividing search times by the binary search time. This clearly confirmed what the previous plot showed: `std::binary_search` is way better than `compromise_search` for any `small_size`. The best values of `small_size` seem to be in the late teens, occasionally slightly less than ten or in the early twenties. Notably, 0 is never the best value for `small_size` (see rank_of_0.csv). In fact, for small sizes, it is often the worst out of 40 values of `small_size` that I tested.

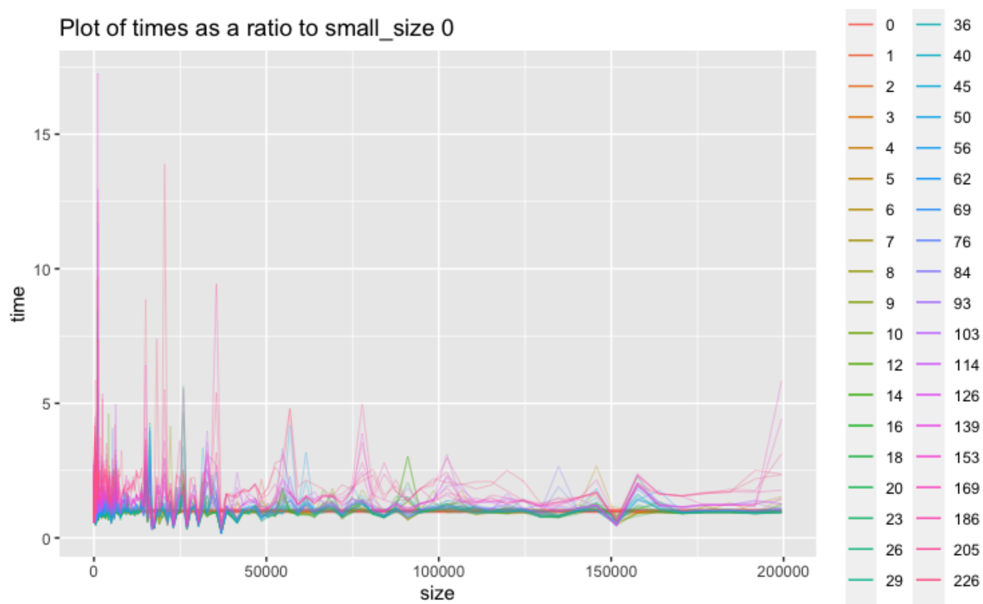

Figure 4: Times as a ratio to `small_size = 0`

I figure that it is likely not `compromise_search`'s fault that it is significantly slower than `std::binary_search`. There is one extra function call compared to a normally binary search in the worst case because `std::find` is called on an empty list, but I do not think the difference would be this significant. Somehow, the code in the STL is more efficient than mine.

Therefore, next, I compared `compromise_search` to the time of `compromise_search` for `small_size` 0, taking the ratio. Note that `std::binary_search` is no longer shown.

As expected, many values of `small_size` are better than the "binary search" generated by `compromise_search`.

Best Value for `small_size`

My guess was that above a `size` of a few hundred, the ideal values for `small_size` would not change very much. This is because the condition where `small_size` is ≥ the length left to be searched is only met once the remaining list is small, by definition.
For a large `size`, it just takes one more iteration than a list half its size. The length of the list is halved until the value for `small_size` matters. For large sizes, the best `small_size` value for size $2n$ is the same as the best value for `small_size` for size n.

The results I found supported this conclusion.

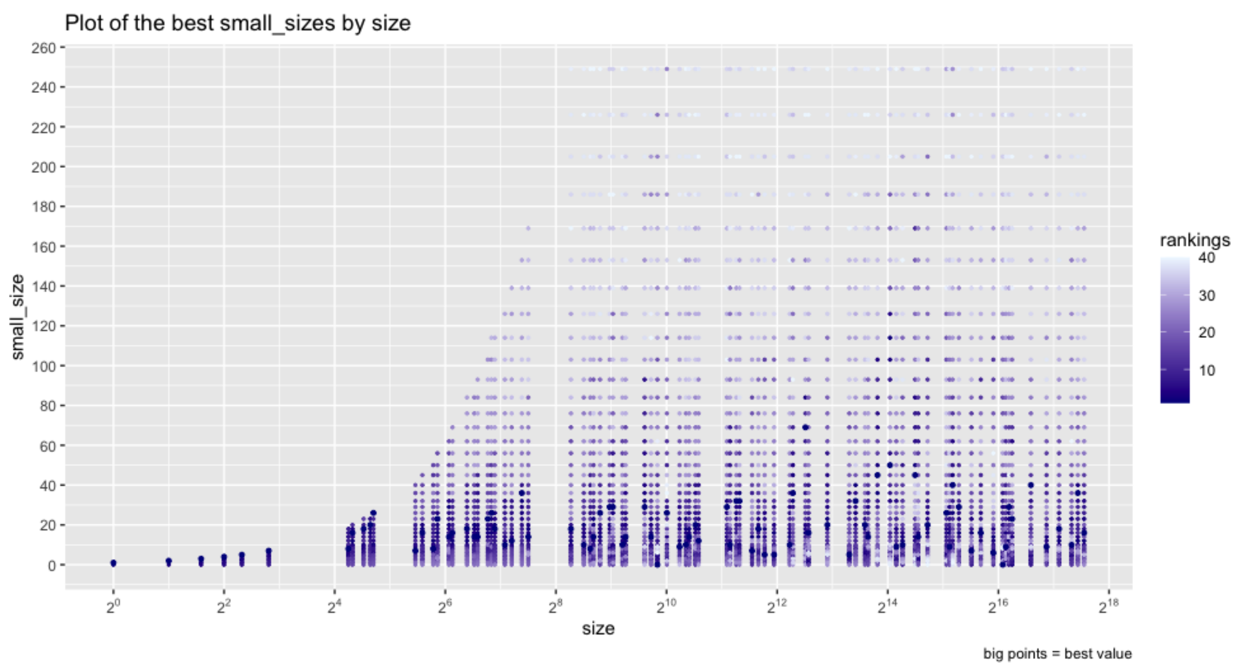

Figure 5: Which of the 40-or-so spaced-out values of `small_size` are best for many sizes

Darker colors signify better values. Time is not on the graph. Configurations were ordered from fastest to slowest, and fastest was assigned a ranking of 1, meaning a lower ranking is better. The axes have `size` and `small_size`. I tested a set of 40-or-so values for `small_size` and plotted the rankings.

This again supports that the best values are in the teens and sometimes slightly below ten or in the early twenties. It also shows that there are also larger values (into the 50s and 60s) that are also successful, but at least in my analysis, those were less frequent. Higher values are generally worse, regardless of `size`. This supports my initial hypothesis that the best values for `small_size` do not change after a certain size threshold.

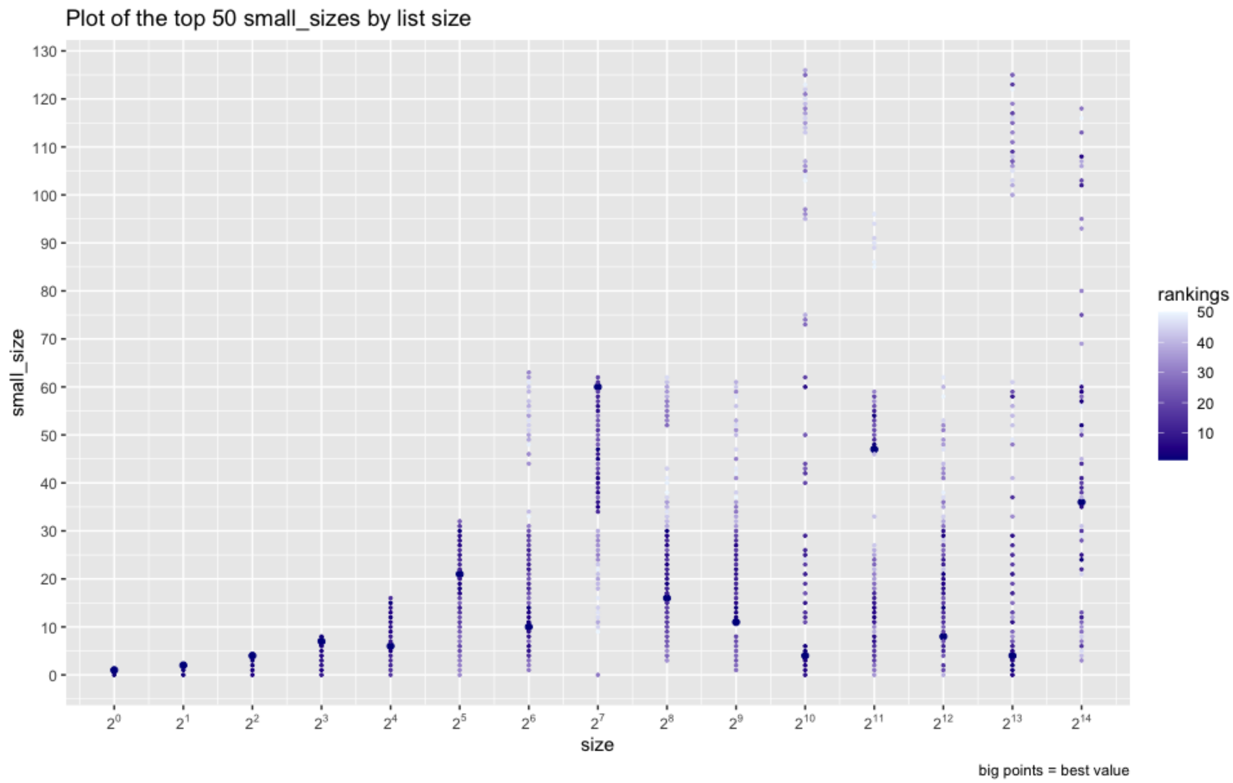

Figure 6: Top 50 small sizes for power of two sizes

To get this figure, I tested every value for `small_size` that was ≤ `size`. Due to the number of tests, I only tested sizes that were powers of two. I plotted the top 50 values, using a color ranking scale like above.

Looking at the color distribution and which points are present (those ranked worse than 50th are not graphed), again, the best small sizes range from around 5 to the early twenties (with some larger sizes), and the best values do not increase as `size` increases after a point.

Methods

Timing

I did benchmarking in two ways. In both, I made sure to use the same random samples for all tests on the same `size`.

The first way was trying a limited set of sizes (powers of two) and every value of `small_size` that was less than `size`. I found the best values for times. The best way to do this seemed to be with a priority queue of pairs with the `small_size` and the `time` for each value of `size` because I only needed the top n values. (I used top 50.)

The second way was testing many sizes and many small sizes. I did this by increasing `size` and `small_size` by a multiplier. I used `floor(1.1 * small_size + 1)` starting with 0 and `ceil(1.05 * size)` starting with 1.
I wanted the data to be sorted by `size` and then `small_size`, so I used a set of tuples containing the `size`, `small_size`, and `time`.

I printed all data and directed them to CSV files.

Analysis

I did all analysis and plotting in R. I did not include the R scripts that I used to reformat the data and preform calculations as well as do the actual plotting.

Raw Data and Code

Link to all code except the code for the actual search since that is an answer to a homework assignment. https://github.com/sarah829/cs_2c-stats-analysis/tree/main/compromise_search

Conclusion

To conclude, the main findings were:

- `compromise_search` is $O(\log n)$ for `size` (and $O(s)$ for `small_size`.
- STL binary search is faster than the `compromise_search` I programmed.
- The best value for `small_size` are frequently in the late teens, sometimes slightly below ten or over twenty. Larger values are successful for some sizes, but they are less frequent.
- The best value for `small_size` does not change with `size` very much after a point.

Code Excerpts

Listing 1: Testing many `size` values with some values for `small_size`

```cpp
std::set<std::tuple<int, int, double>> test_small_sizes_and_sizes(int trials, int reps,
        int max_size, int max_small_size, double size_multiplier, double small_size_multiplier)
{
  std::set<std::tuple<int, int, double>> results;
  int size = 1;
  std::vector<int> values;
  values.push_back(0);
  while (size <= max_size)
  {
    // get random samples
    int sample_size = size / SAMPLE_SIZE_FRACTION + 1;
    std::vector<std::vector<int>> samples;
    samples.reserve(trials);
    for (int i = 0; i < trials; ++i) {
      samples.push_back(random_sample(values, sample_size));
    }
    int small_size = 0;
    while (small_size <= max_small_size) {
      // iterate through samples
      double compromise_time = 0;
      for (const std::vector<int> &sample : samples) {
        // do each trial reps times
        for (int i = 0; i < reps; ++i) {
          compromise_time += time_compromise(values, sample, small_size);
        }
```

```
      }
      // avg time
      compromise_time /= (trials * reps * sample_size);
      results.insert(std::tuple<int, int, double>(small_size, size, compromise_time));
      small_size = floor(small_size_multiplier * small_size + 1);
    }

    // std::find and std::binary
    double std_find_time = 0, std_binary_time = 0;
    for (const std::vector<int> &sample : samples) {
      // do each trial reps times
      for (int i = 0; i < reps; ++i) {
        std_binary_time += time_std_binary(values, sample);
        std_find_time += time_std_find(values, sample);
      }
    }
    // avg time
    std_binary_time /= (trials * reps * sample_size);
    std_find_time /= (trials * reps * sample_size);
    results.insert(std::tuple<int, int, double>(-1, size, std_binary_time));
    results.insert(std::tuple<int, int, double>(-2, size, std_find_time));

    // add numbers to values
    int current_size = size;
    size = ceil(size_multiplier * size);
    for (int i = current_size; i < size; i++)
    {
      values.push_back(i);
    }
  }
  return results;
}
```

Listing 2: Testing all values for `small_size` for limited `size` values

```
template<class T>
std::vector<std::priority_queue<std::pair<int, double>, std::vector<std::pair<int, double>>,
    compare_pair_heap>> test_size(const std::vector<T> &values, int sample_size, int trials,
    int reps, int min_small_size, int max_small_size) {
  // store results
  std::priority_queue<std::pair<int, double>, std::vector<std::pair<int, double>>, compare_pair_heap>
      compromise_time_pairs;
  // these next two only have 1 element since they don't care about small_size
  // necessary b/c need to use the same random samples
  std::priority_queue<std::pair<int, double>, std::vector<std::pair<int, double>>,
      compare_pair_heap> std_find_time_pairs;
  std::priority_queue<std::pair<int, double>, std::vector<std::pair<int, double>>,
      compare_pair_heap> std_binary_time_pairs;

```

```
  std::vector<std::vector<T>>
      samples; // stores a vector of the random samples (size will be trials)

  // get the random samples
  samples.reserve(trials);
  for (int i = 0; i < trials; ++i) {
    samples.push_back(random_sample(values, sample_size));
  }

  // test_size compromise search
  // iterate through small_size values
  for (int small_size = min_small_size; small_size <= max_small_size; ++
      small_size) {
    double compromise_time = 0.0;
    // iterate through samples
    for (const std::vector<T> &sample : samples) {
      // do each trial reps times
      for (int i = 0; i < reps; ++i) {
        compromise_time += time_compromise(values, sample, small_size);
      }
    }
    // avg time
    compromise_time /= (trials * reps * sample_size);
    compromise_time_pairs.push(std::pair<int, double>(small_size,
        compromise_time));
  }

  // test_size std::find and std::binary_search
  double std_find_time = 0.0, std_binary_time = 0;
  // iterate through samples
  for (const std::vector<T> &sample : samples) {
    // do each trial reps times
    for (int i = 0; i < reps; ++i) {
      std_find_time += time_std_find(values, sample);
      std_binary_time += time_std_binary(values, sample);
    }
  }
  // avg time
  std_find_time /= (trials * reps * sample_size);
  std_binary_time /= (trials * reps * sample_size);
  std_find_time_pairs.push(std::pair<int, double>(-1, std_find_time));
  std_binary_time_pairs.push(std::pair<int, double>(-2, std_binary_time));

  // return results
  std::vector<std::priority_queue<std::pair<int, double>, std::vector<std::
      pair<int, double>>,
        compare_pair_heap>> results;
  results.push_back(compromise_time_pairs);
  results.push_back(std_find_time_pairs);
  results.push_back(std_binary_time_pairs);
  return results;
}
```