CompressionManager

Experiment Report

Team Members:

Sean Hinton

CSC316-002

11/19/2022

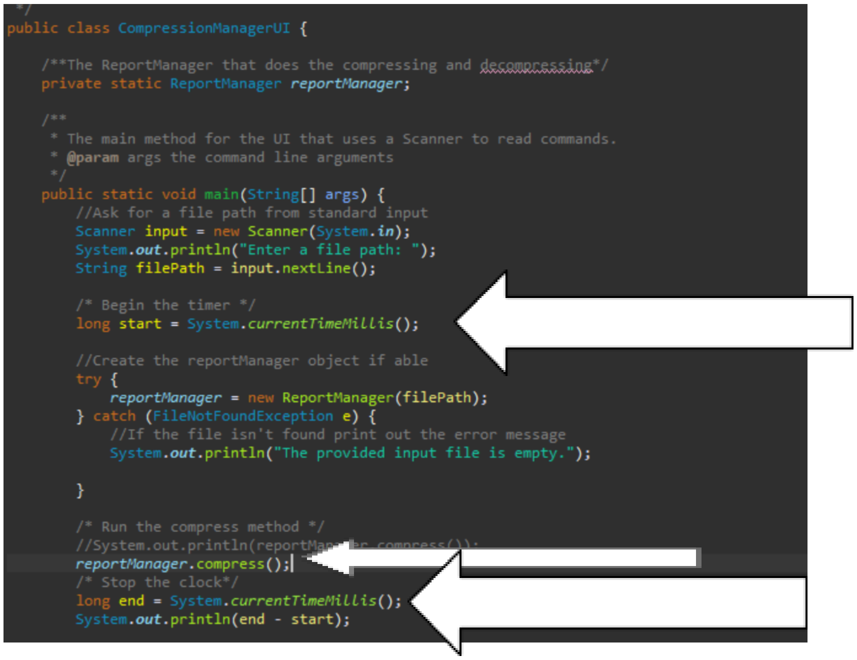
Implementation provided by:

Sean Hinton

Methodology

Measuring Runtimes

For the experiment it followed the UI’s normal behavior of asking for the file path of the file to compress. After the file path was input the program took the current time. The ReportManager was constructed then automatically went to compress. After returning the compressed output the timer was stopped and the run time was printed to the console after the output. The runtimes of different Map data structures will be measured for varying input sizes.



```
 */
public class CompressionManagerUI {

    /**The ReportManager that does the compressing and decompressing*/
    private static ReportManager reportManager;

    /**
     * The main method for the UI that uses a Scanner to read commands.
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        //Ask for a file path from standard input
        Scanner input = new Scanner(System.in);
        System.out.println("Enter a file path: ");
        String filePath = input.nextLine();

        /* Begin the timer */
        long start = System.currentTimeMillis();

        //Create the reportManager object if able
        try {
            reportManager = new ReportManager(filePath);
        } catch (FileNotFoundException e) {
            //If the file isn't found print out the error message
            System.out.println("The provided input file is empty.");

        }

        /* Run the compress method */
        //System.out.println(reportManager.compress());
        reportManager.compress();
        /* Stop the clock*/
        long end = System.currentTimeMillis();
        System.out.println(end - start);
```

Results

Hardware

We conducted the experiment using the following hardware:

- Operating system version: _Windows 10, 64 bit OS_
- Amount of RAM: _8.00 GB
- Processor Type & Speed: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz _

Table of Actual Runtimes for getDistancesReport()

> **You may delete this instruction box before submitting.** As you measure your actual runtimes, complete the table below. If any measured runtimes are longer than 30 minutes, indicate ">30 min" in the table cell. If your software produces an error, indicate "Error" in the table cell.

Input Size (2^x)	Unordered Linked Map (ms)	Search Table Map (ms)	Skip List Map (ms)	Splay Tree Map (ms)	Red-Black Tree Map (ms)	Linear Probing Hash Map (ms)
6	58	52	56	51	59	52
8	64	58	60	63	68	62
10	98	78	84	85	89	90
12	191	149	134	177	146	122
14	807	259	250	297	288	214
16	20058	1249	782	593	594	408
18	591914	11977	1664	1702	1146	1157
20	>30 min	265400	6230	5904	4442	2879
22	>30 min	679988	24288	30261	16288	9286

Chart 1: Log-Log Chart of Actual Runtimes

You may delete this instruction box before submitting. Insert an image of your log-log chart that compares the runtime of the data structures required for this experiment.

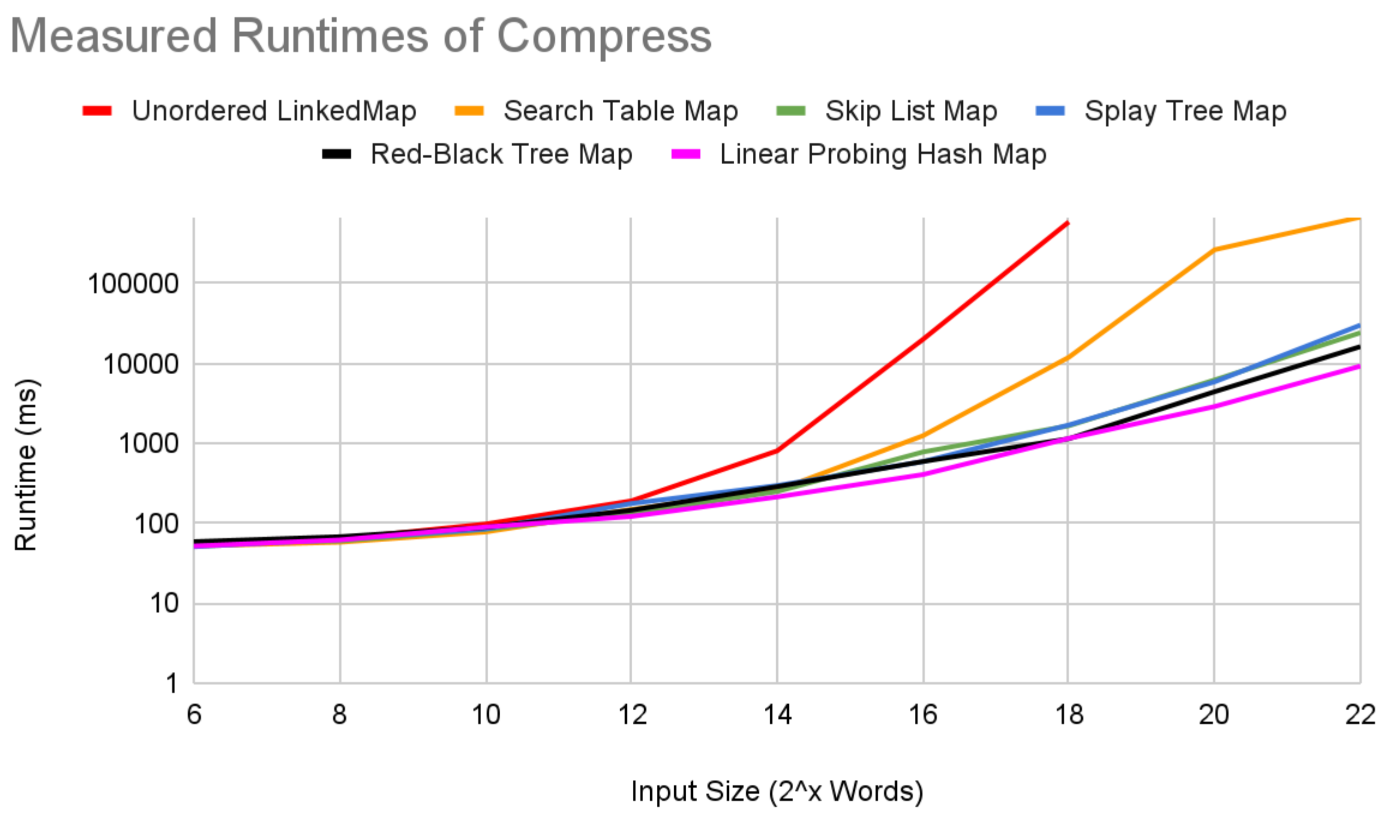


Chart 2: Log-Log Chart of Actual Runtimes Unordered Linked List-based Map

You may delete this instruction box before submitting. Insert an image of your log-log chart that shows:

- Actual runtime (as a solid line) of the software when using unordered linked list-based map
- Normalized growth rates (as dotted lines) for the following functions:
  - O(log n)
  - O(n)
  - O(n log n)
  - $O(n^2)$
  - $O(n^3)$
  - $O(n^4)$

Chart Reminders

- Include a chart title
- Label axes (include units, if appropriate)
- Include a legend to clearly indicate what each line represents

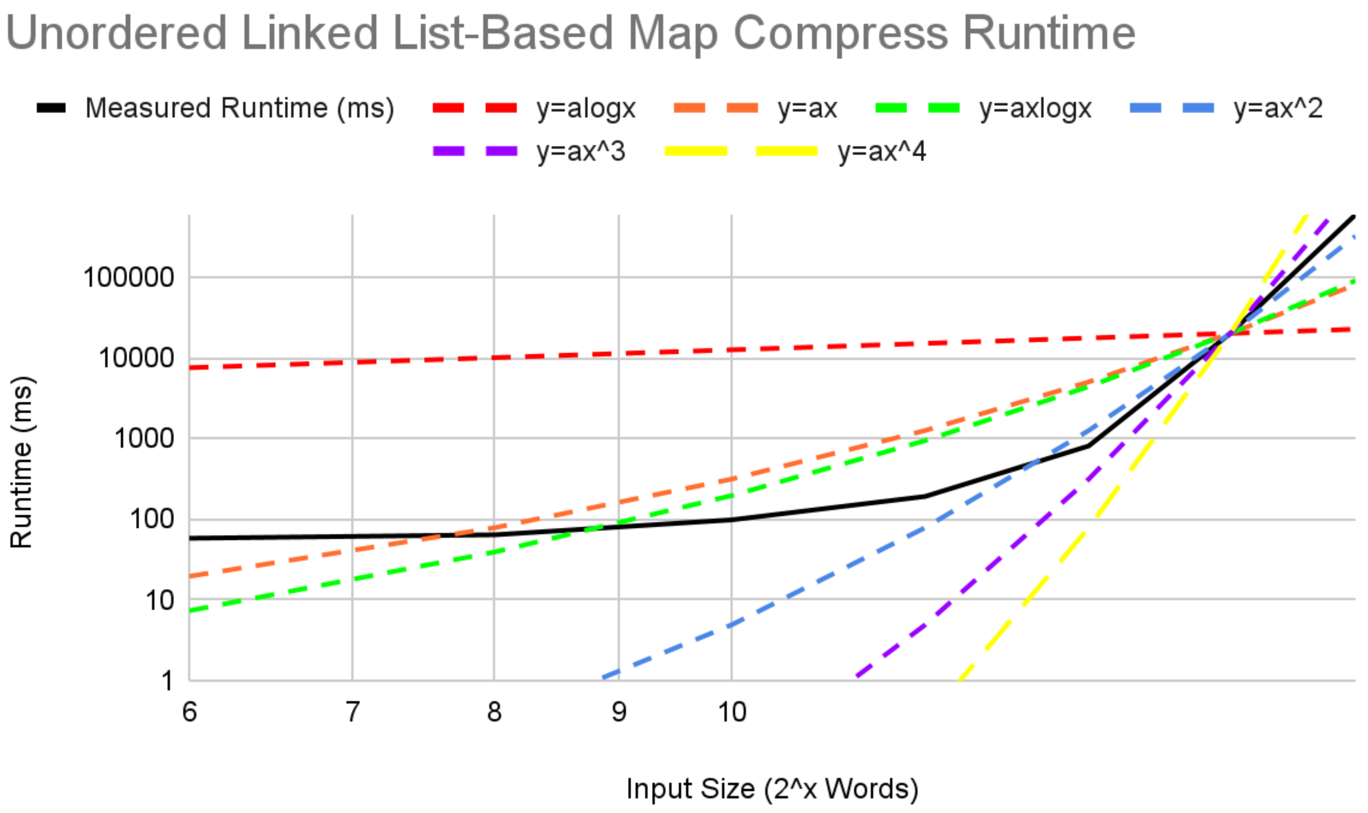


Chart 3: Log-Log Chart of Actual Runtimes for Search Table Map

You may delete this instruction box before submitting. Insert an image of your log-log chart that shows:

- Actual runtime (as a solid line) of the software when using your search table map
- Normalized growth rates (as dotted lines) for the following functions:
    - O(log n)
    - O(n)
    - O(n log n)
    - O(n^2)
    - O(n^3)
    - O(n^4)

Chart Reminders

- Include a chart title
- Label axes (include units, if appropriate)
- Include a legend to clearly indicate what each line represents

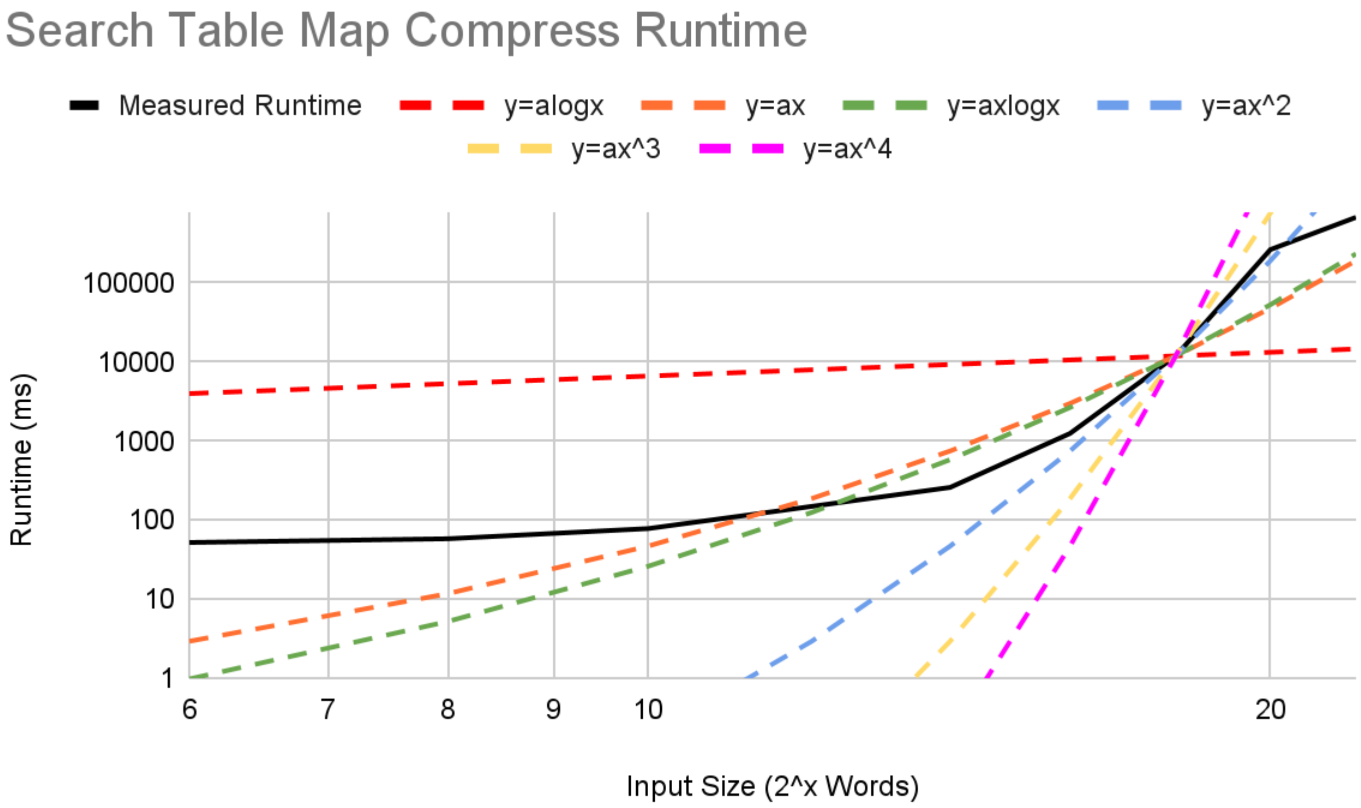


Chart 4: Log-Log Chart of Actual Runtimes for Skip List Map

You may delete this instruction box before submitting. Insert an image of your log-log chart that shows:

- Actual runtime (as a solid line) of the software when using your skip list map
- Normalized growth rates (as dotted lines) for the following functions:
  - O(log n)
  - O(n)
  - O(n log n)
  - O(n^2)
  - O(n^3)
  - O(n^4)

Chart Reminders

- Include a chart title
- Label axes (include units, if appropriate)
- Include a legend to clearly indicate what each line represents

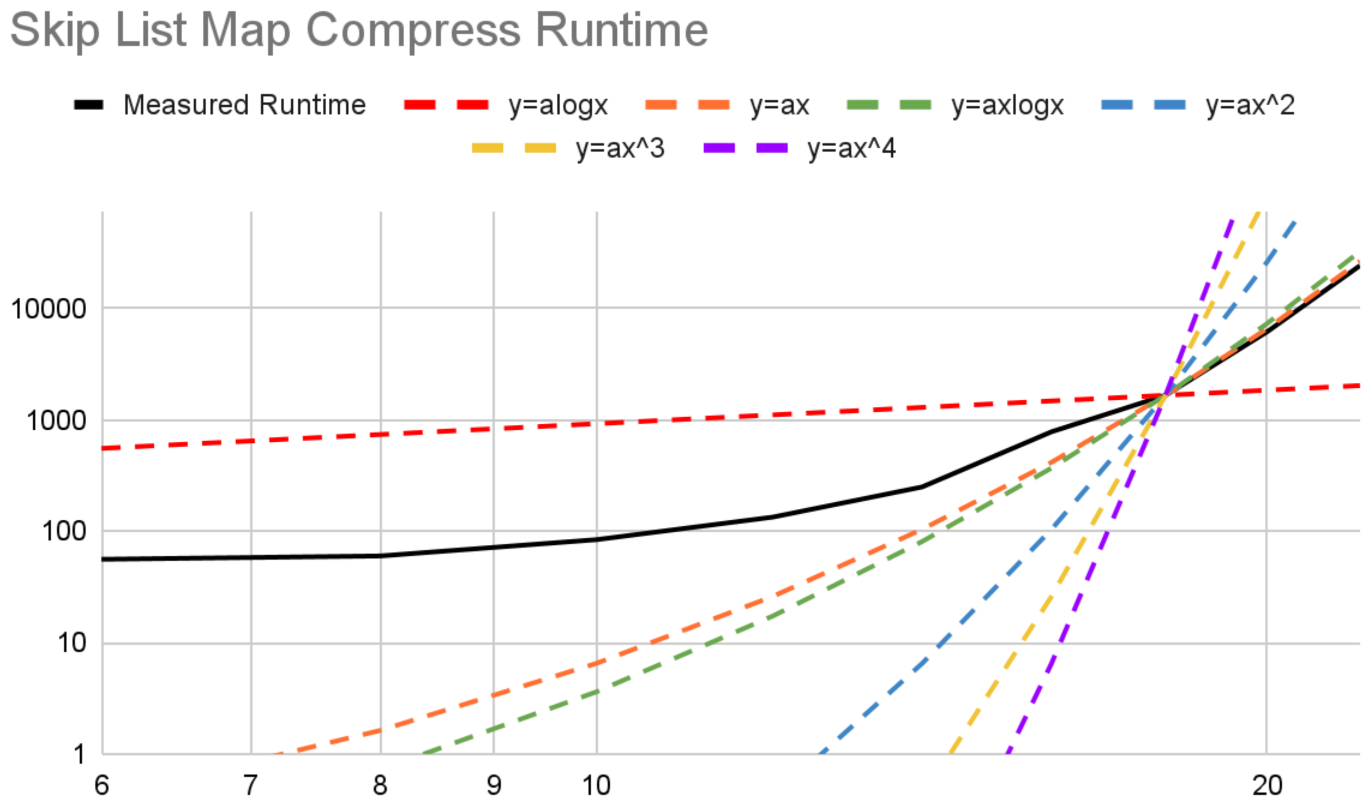


Chart 5: Log-Log Chart of Actual Runtimes for Splay Tree Map

You may delete this instruction box before submitting. Insert an image of your log-log chart that shows:

- Actual runtime (as a solid line) of the software when using your splay tree map
- Normalized growth rates (as dotted lines) for the following functions:
  - O(log n)
  - O(n)
  - O(n log n)
  - O(n^2)
  - O(n^3)
  - O(n^4)

Chart Reminders

- Include a chart title
- Label axes (include units, if appropriate)
- Include a legend to clearly indicate what each line represents

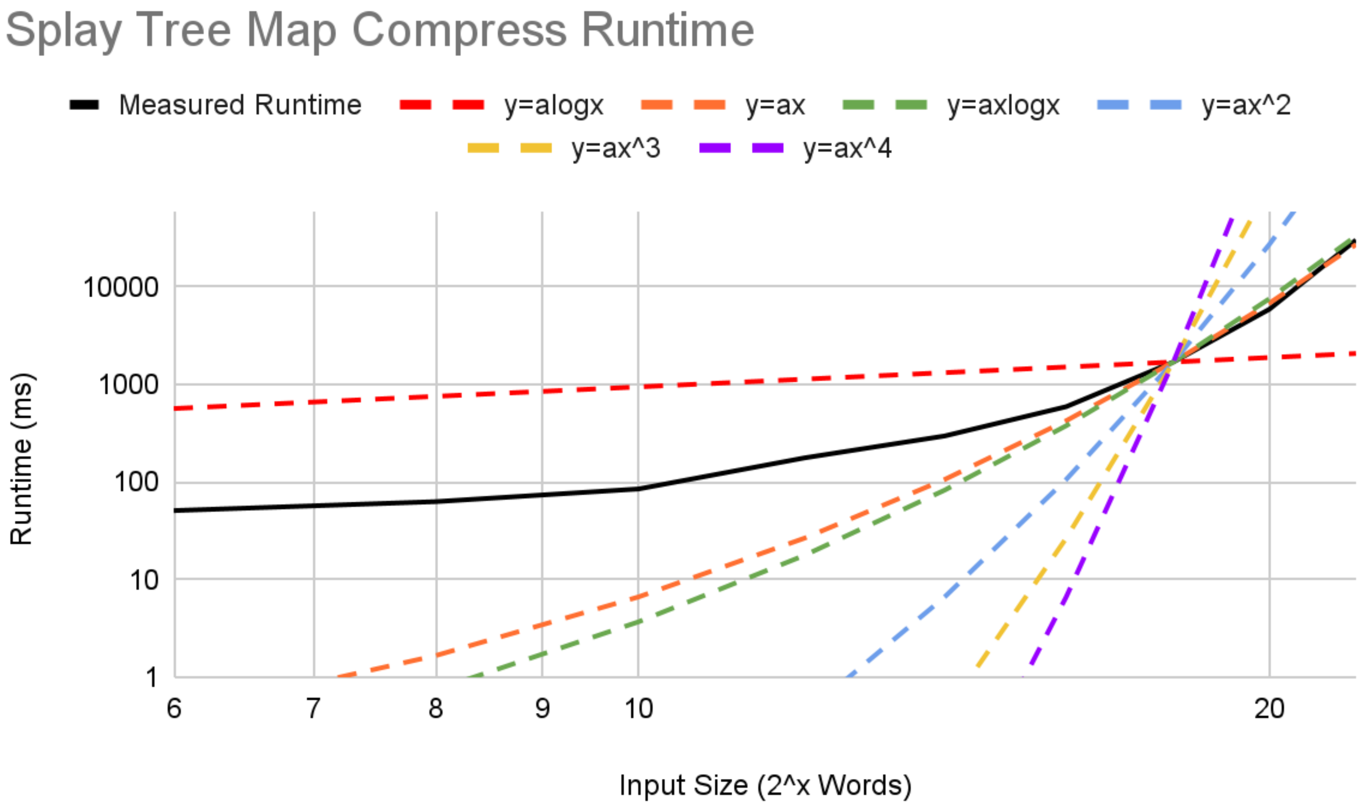


Chart 6: Log-Log Chart of Actual Runtimes for Red-Black Tree Map

You may delete this instruction box before submitting. Insert an image of your log-log chart that shows:

- Actual runtime (as a solid line) of the software when using your red-black tree map
- Normalized growth rates (as dotted lines) for the following functions:
    - O(log n)
    - O(n)
    - O(n log n)
    - O(n^2)
    - O(n^3)
    - O(n^4)

Chart Reminders

- Include a chart title
- Label axes (include units, if appropriate)
- Include a legend to clearly indicate what each line represents

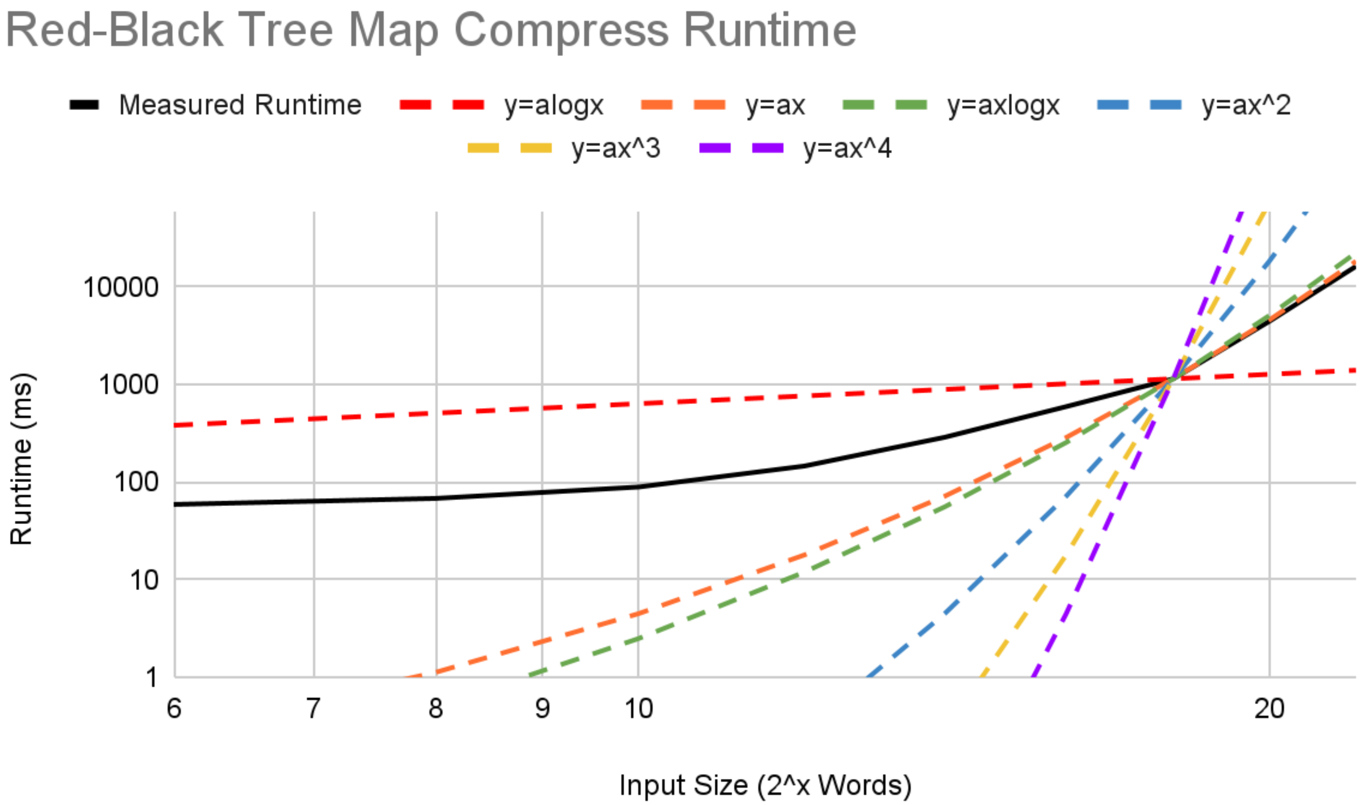


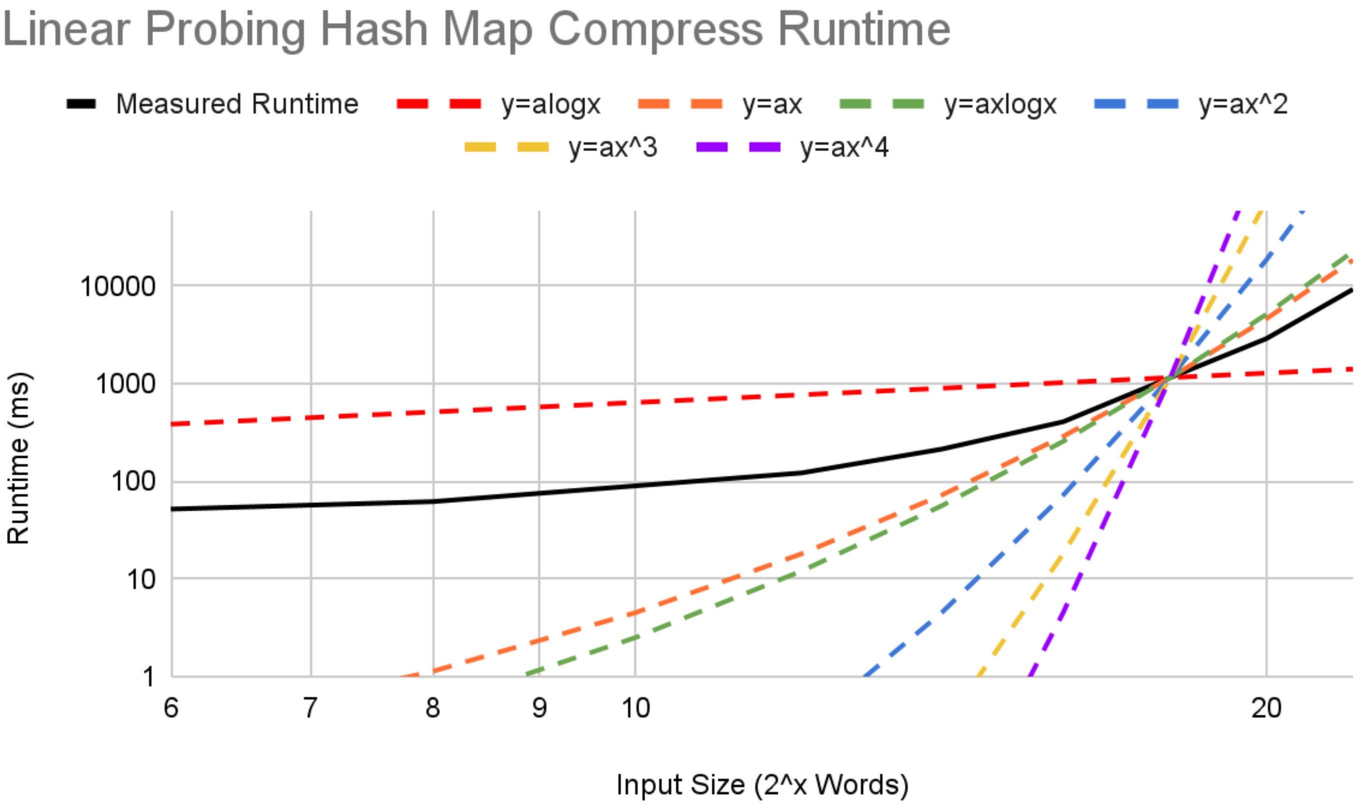
Chart 7: Log-Log Chart of Actual Runtimes for Linear Probing Hash Map

You may delete this instruction box before submitting. Insert an image of your log-log chart that shows:

- Actual runtime (as a solid line) of the software when using your linear probing hash map
- Normalized growth rates (as dotted lines) for the following functions:
  - O(log n)
  - O(n)
  - O(n log n)
  - O(n^2)
  - O(n^3)
  - O(n^4)

Chart Reminders

- Include a chart title
- Label axes (include units, if appropriate)
- Include a legend to clearly indicate what each line represents



Discussion

Reflection on Theoretical Analysis

Include a detailed, insightful discussion (no more than 1/2 page) of how your actual results compare to your theoretical analysis from your project proposal. Do the experiment results support the theoretical analysis? Why or why not?

The theoretical results of the pseudo code measured the theoretical runtime to be O(mn log mn) where m is the number of words in the current List and n is the number of Lists in the input file. If we change the runtime to just be the total number of words it would be, Words = m * n, so the theoretical runtime would be O(nlogn) where n is the number of words in the input. This gets close to the measured runtime of the compress method for the Skip List which was shown to stay around O(n) and O(nlogn) runtime. These measurements however support that the Skip List was running faster than the theoretical runtime of O(nlogn). This can probably be attributed to randomization providing something closer to O(logn) operations and possibly making bigger inputs run faster and smaller ones run slower, thus causing the measured runtime to match O(n) more closely compared to O(nlogn) even though we expected it to run at O(nlogn).

Reflection on Data Structure Selection

Include a detailed, insightful discussion (no more than 1/2 page) of why/how the originally selected data structure was appropriate to implement the software. Did your selected data structure actually perform the best? Why or why not? Remember that we started the project when we had covered only list-based maps.

The selected Skip List Map was most appropriate at the time for the provided Map data structures. As the graph of measured run times for all graph data structures shows the Skip List was the fastest of the 3 Map implementations provided. However, with the ability to use faster data structures like Red-Black Tree Map with O(logn) runtime, Splay Tree Map O(logn) amortized runtime, and Hash Map with O(1) expected runtime these newers maps provided a better expected runtime than the O(logn) average runtime. The Linear Probing Hash Map and Red-Black Tree both compressed text quicker, but the Skip List was better than the Splay Tree Map since compress never takes advantage of the Splay Tree's move to front heuristic since we are repeatedly getting lines in linear order rather than a random order. This might have caused the Splay Tree to run slower than the Skip List since the tree became unbalanced as lines were gotten. The Skip List did manage to be the best compared to Unordered Linked List Based Map and SearchTable Map, but with faster Maps being usable a Red-Black Tree Map or Hash Map would be the best for the software with the Linear Porbing Hash Map being the best of the tested Map types.

Improving Efficiency

Include a brief discussion (no more than 1/2 page) of how the software could be improved to further reduce actual runtimes. For example, could you use a different sorting algorithm? Could you improve your algorithm to stop “early” if some condition is met? Could you pre-sort the input file before processing any of the file contents?

To improve the software one of the major changes to be made should be to change the Map used to a Hash Map implementation. The Hash Map provide the quickest operations on put() and get() which are the primary methods used for the compress methods. The expected O(1) runtimes has shown to be quicker than the previous Skip List implementation. Another way that the compress algorithm could be improved is by changing the sorting method used. Although mergesort seems obviously fastest with its worst case O(nlogn) runtime counting sort can provide a faster run time. By using a wrapper class that has an Entry and is identifiable the software can use the Key of an uncompressed entry to be the id. Since we always know the range of the id’s, range = n - 1, the theoretical runtime become O(n) for counting sort. Another way to try and nullify the compress method’s runtime is by moving it into when the Map is read. Whenever a token for a String is found that value is checked with a map for a repeat and prints its corresponding number if it’s repeated. You could also create a wrapper class for each List of words with a boolean value. When each line is being read in check if any of the Strings in the List are a repeat, if there are no repeats the boolean value is set to false and no compression is needed. The InputReader could also take a Map<String, Integer> as a parameter and put unique values into that Map so we don’t have to compress for those false lines or add anything during the compress method.

Lessons Learned

Include a detailed, insightful discussion (no more than 1/2 page) of at least 2 lessons learned about data structures and algorithms based on the results of the experiment.

The biggest lesson learned was that keeping references and sacrificing some memory on local, stack variables can cause run time to be quicker. When input files are extremely large every single one of those extra get methods requires another search of a Tree, Skip List, or Search Table. Even though these data structures have quicker operations keeping references reduces runtime significantly when there are millions of entries. Another lesson learned was to use your iterators for things like Maps when available. Instead of using a regular for loop which requires a get() on every loop iteration to get an entry using an iterator allows constant access to Map entries in their sorted order. Sorting the Map was also helpful for compress and decompress methods before doing any checks so we don’t have re-read the input file every time. It also helps for multiple compress or decompress calls since a heuristic Map may take entries out of order and presorting the unprocessed Map is required. Some other lessons learned were counting sort can run faster than quicksort or mergesort if you know that your range is close to the number of entries. Also heuristic data structures are useful but maybe not for a situation like compression since the algorithm gets every entry in a Map, and all those splay operations become useless when every element is gotten. This can cause Splay Tree Map to run slower than another Tree Map lik Red-Black tree.