It needs to be REALLY fast!



A space-exploration Java coding challenge sought a rapid algorithm to assess the amount of radiation impinging on a square sensor. The sensor would “white out” if exposed to a critical amount of radiation, and we need a rapid exposed-area determination in order to invoke a protective filter in real time.

In this article, we will learn how to:

* Create a performance validation mechanism.
* Organize algorithm variants by trade-offs to optimize section.

By reviewing the code associated with this article, you will learn how to apply:

* OO techniques (Strategy and Factory patterns, interface-based programming.)
* Modularity (group together related items, separate out disparate items.)
* Use Test Driven Development (**TDD**) to increase speed and reliability of development.

Let’s start down the path of discovery as we work toward space exploration. A *reference* section is included below, with URLs to all code and run logs.

## The Sensor Problem

We have a square radiation sensor with an overlaid grid of exposure regions, with the lower left grid as the origin. The positive X direction, to the right of the origin, is labeled 0 through 1000. Similarly, the positive Y direction is upward from the origin, and is also labeled 0 through 1000. The value 1000 is symbolically named **XY\_UPPER\_BOUND**. The grid uses arbitrary length units (e.g., microns) and defines a logical partition of the physical sensor area. An exposed area is a rectangle defined by the coordinates of its lower left and upper right corners {(x1, y1) to (x2, y2)}.

We validate the sensor using a radiation generator that emits rapid, random bursts of radiation falling on a rectangular region of the sensor. A sensor used in a mission is expected to receive no more than a million bursts a second. The measurement period will encounter **N** total radiation bursts. If the accumulated bursts reach or exceed a threshold of **K** bursts over a monitoring period, then the sensor will “white out” for a period of time. For example, more than two bursts per millisecond in a grid square temporarily “blinds” that square.

Ideally, we are able to compute the exposure level by area (bursts impinging on a sensor region) in sufficient time for the filter to be deployed. For example, we need to deploy the filter if we detect that 60% (600K squares) of the exposed sensor area has reached a critical value. That is areas exceeding the threshold of **K** bursts of radiation after **N** exposures.

## Expected Deliverables of Investigation

Space agencies have two important concepts: *verification* and *validation*. For purposes of this problem, verification is showing functional requirements are met. That is, we are able to correctly assess exposed areas in the critical region of impinging radiation. Validation is the effort to show that the algorithm is both fast enough, and uses acceptable levels of memory, to solve the problem during an actual mission.

We will deliver:

1. Algorithms implemented as a Java function accepting a list of exposed rectangles (length **N**) and radiation burst count thresholds (**K** bursts in a grid square.)
2. A “Big Oh” analysis of time complexity for each algorithm.
3. A validation of the algorithm in a test environment.

## Verification and Validation Tools

We will use unit tests with specific test cases, each with a known result, to verify our algorithms. We will also create a timing mechanism to validate the execution time (wall clock.) Java execution time is sensitive to memory management (i.e., garbage collection) and operating system background needs. We mitigate this by running tests under continuous garbage collection, and on a “quiet” system with no other foreground tasks (see the timer code.) See reference #5 for the timer code, which uses each algorithm as a strategy to time.

## First Attempt – Use Java Maps

We model an exposed portion of the grid as a rectangle where a grid location is converted to an index based on the lower left corner coordinates. That is, a grid index is computed as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **h - 1** | **(h - 1) \*w** | **(h - 1) \*w + 1** | **(h - 1) \*w + 2** | **. . .** | **h\*w - 1** | | **. . .** | **. . .** | **. . .** | **. . .** | **. . .** | **. . .** | | **2** | **2\*w** | **2\*w + 1** | **2\*w + 2** | **. . .** | **3\*w - 1** | | **1** | **w** | **w + 1** | **w + 2** | **. . .** | **2\*w - 1** | | **0** | **0** | **1** | **2** | **. . .** | **w - 1** | |  | **0** | **1** | **2** | **. . .** | **w - 1** | |  | **x = horizontal position of LL corner.**  **y = vertical positon of lower left corner.**  **Index = y \* w + x** |

We increment each grid cell when it is exposed to a radiation burst; each burst is encoded as a rectangle passed into the **findArea** function. The code for recording bursts in a region looks like this:

//

// Apply radiation per exposure: O(n \* a)

**for** (Rectangle r : exposures) {

**for** (**int** y = r.y1; y < r.y2; y++) {

**int** ypos = y - bbox.lowerLeftY;

**int** yposR = bbox.height - ypos - 1;

**for** (**int** x = r.x1; x < r.x2; x++) {

**int** xpos = x - bbox.lowerLeftX;

**int** idx = yposR \* bbox.width + xpos;

sensorRegions.merge(idx, 1, (a, b) -> a + b);

}

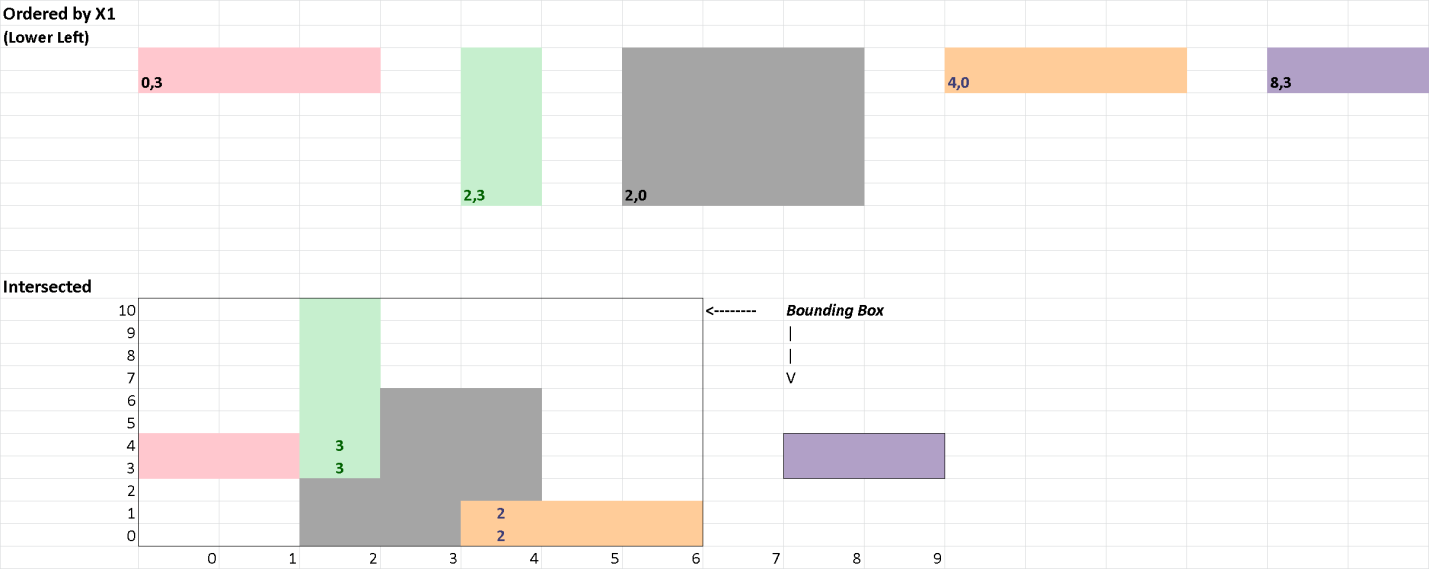
}

}

The Java Map implementation validation was run for up to a million bursts, and it required 1.55 seconds to process the million bursts. That does not leave a lot of time to deploy the protective filter as the bursts accumulate.

## Hybrid Approaches

Looking to speed up the algorithm, it did not appear that we could improve on the O(**n**) aspect of processing a list of **n** bursts. We could however, lower the effort of recording bursts as each rectangle was encountered. We could do this by lowering the number of recording regions that were needed for each burst exposure analysis. This new approach, named *hybrid*, would require sorting the input to identify the sub-regions needing analysis. This next diagram illustrates the approach to creating smaller sub-regions for analyzing; these sub-regions are called *holdings* in the code:



The “bounding box” for the intersected region is much smaller than the bounding box around both the intersected region and the extra (purple) exposed region beyond the overlap. The counts in the cells show the number of bursts to which that cell was exposed.

The required sort step introduced an O(n \* log(n)) operation. However, since we only need ordering in the X1 dimension, a counting sort, using X1 as a key, is fast (that is O(n + 1000).) As an implementation exploration issue, we also explored using the Java “parallel sort”. A parallel sort uses multiple treads to speed sorting. The radiation burst recording code now looked like this:

@Override

**public** **int** findArea(List<? **extends** Rectangle> exposures, **final** **int** k) {

**if** (exposures.isEmpty()) {

**return** 0;

}

List<Rectangle> regions = orderRectangles(exposures);

regions.add(***ender***);

State state = **new** State();

List<Rectangle> holding = **new** ArrayList<Rectangle>();

Iterator<Rectangle> itr = regions.iterator();

Rectangle reg = itr.next();

state = mergeIntoHoldings(state, reg, holding);

**while** (itr.hasNext()) {

reg = itr.next();

**if** (reg == ***ender***) {

state = flushHolding(state, k, holding);

holding = **null**;

**break**;

}

//

// Accumulate overlapping rectangles, process on flush

**if** (isNonOverlapping(reg, state.rgtHoldingBound)) {

state = flushHolding(state, k, holding);

holding = **new** ArrayList<Rectangle>();

}

//

// Merge the new rectangle into the current holding

state = mergeIntoHoldings(state, reg, holding);

}

**if** (reg != ***ender*** || holding != **null**) {

**throw** **new** IllegalStateException("Did not encounter ending record, or did not clear holdings");

}

**return** state.area;

}

We changed the sort implementation (the **orderRectangles** method) and timed the results for each sort implementation to compare sorting techniques. For the first million validation records, we got:

* Parallel Sort: 0.396 seconds.
* Standard Sort: 0.363 seconds.
* Counting Sort: 0.213 seconds.

This was a significant improvement over the initial Map-based approach. However, JVM based code is always subjected to garbage collection and operating system threading issues, making timing erratic. We wondered if using “*brute force*”, following the KISS principle, might be faster still without JVM and threading overhead.

## Simple Area Only

We only have one million grid cells in the logical sensor grid – this is not a large amount of memory to represent our exposure counts. In addition, modern machines can do basic operations, like addition in an array, in a few nanosecond clock cycles. As a result of this processing speed, the time required to deal with a million items is really very short. We next try ignoring the complex segmentation analysis of the Hybrid. With the direct area (KISS) approach, our code simplified to:

@Override

**public** **int** findArea(List<? **extends** Rectangle> exposures, **final** **int** k) {

**if** (exposures.isEmpty()) {

**return** 0;

}

//

// Get exposure session bounding box: O(n)

BoundingBox bbox = SensorMonitoring.*findBoundingBox*(exposures);

//

// Allocate bounding box of exposed Sensor region: O(1)

**int**[] sensorRegions = **new** **int**[bbox.width \* bbox.height];

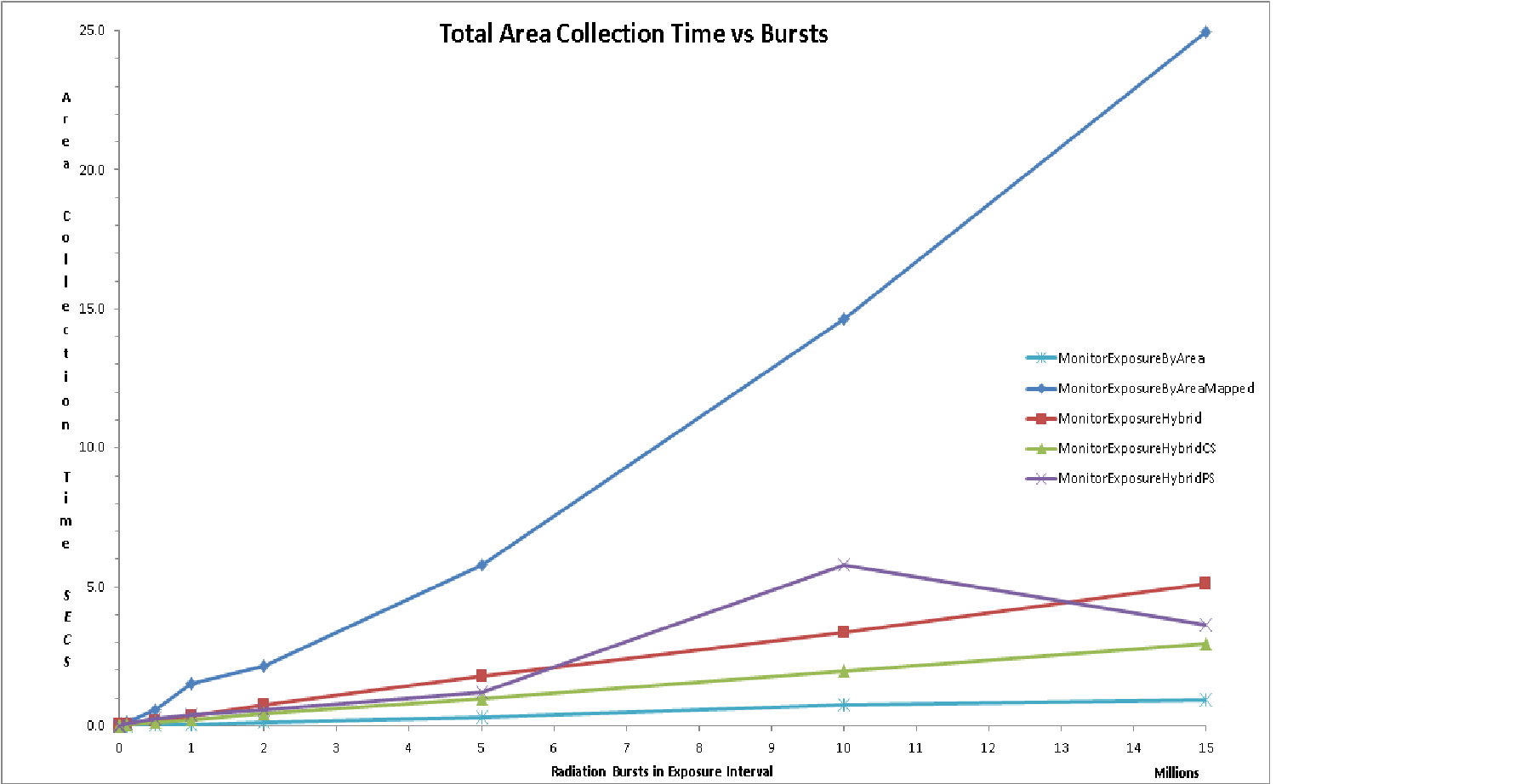
//

// Apply radiation per exposure: O(n \* a)

**return** SensorMonitoring.*exposeSensor*(sensorRegions, bbox, exposures, k);

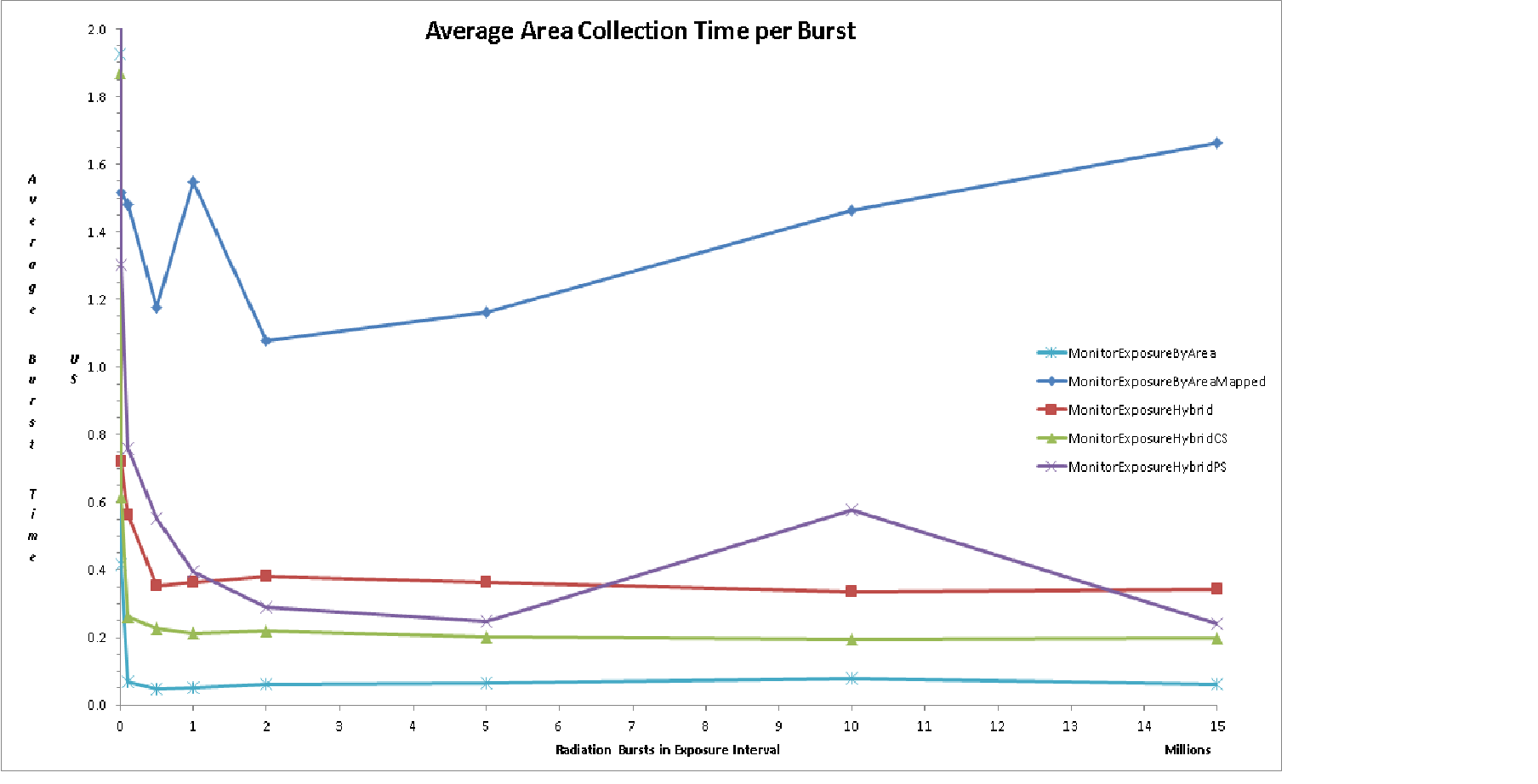
}

The KISS algorithm used a phenomenal 0.053 seconds to analyze the million busts. With these results, we extended the test range to include much larger burst counts to make sure our algorithms are well tested in both the specified and potential ranges of interest. The resulting combined performance view was surprising:



The map based approach suffers from rehashing the added burst areas as more and more are added to the map. The parallel sort version of the Hybrid algorithm shows its timing sensitivity of operating system random actions in the background. The KISS approach is relatively insensitive to these issues.

We can also view the average time required to process a burst of radiation to see another view algorithm behavior over the region of interest:



With this information, we can now create a trade-off analysis:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algo** | **Average Complexity** | | **Average Microseconds per Burst** | | **Comment** |
| **Time** | **Space** | **Validation** | **Asymptote** |
| Map by Area | O(n\*a) | O(n+a) | 1.55 | 1.66 | Minimizes space; time increases due to map rehash (add map capacity, lower time.) |
| Hybrid Parallel Sort | O(n\*b) | O(n+b) | 0.396 | 0.241 | Lowers space requirement by creating collections of overlapping exposures (holdings); each with a smaller bounding box than the total sensor area. Introduces a sort step. |
| Hybrid | O(n\*b) | O(n+b) | 0.363 | 0.341 |
| Hybrid Counting Sort | O(n\*b) | O(n+b) | 0.213 | 0.199 |
| Array by Area | O(n\*a) | O(n+a) | 0.053 | 0.063 | Simple to code, very fast, and **a** is 10^6 |

In the table above, **n** represents the number of bursts (exposures) monitored, **a** represents the average size of the area exposed over the whole sensor, and **b** is the average size of the area associated with the bounding box of holdings. We expect **b** << **a.**

## References

The Java code, run-logs, and documentation are all located in the ***DemoDev*** GitHub repo at <https://github.com/DonaldET/DemoDev>.

1. A Maven project for the code and article is located in the repo at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-nasa-sensor>.
2. Algorithm source at: <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-nasa-sensor/src/main/java/demo/algo/sensor> .
3. Verification tests: <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-nasa-sensor/src/test/java/demo/algo/sensor/test>.
4. Analysis of runs: <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-nasa-sensor/analysis>.
5. Validation timing tool: <https://github.com/DonaldET/DemoDev/blob/master/dev-topics-algorithms/dev-topics-nasa-sensor/src/test/java/demo/algo/sensor/test/TimeExposures.java>.
6. This document and supporting information: <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-nasa-sensor/documentation>.
7. “Big O” complexity explained: <https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/> and <https://www.bigocheatsheet.com/>.