# 

# Space verses Time Trade-offs and Algorithm Analysis

I was recently asked to create and compare two solutions to a telephone call analysis problem. As often happens between engineers, a preference develops for a particular solution. We often select the “fastest” solution, based on “Big Oh” analysis. However, as discussed in a previous article on the vagaries of such an analysis, we should consider the actual performance in the expected area of application (please see <https://www.linkedin.com/pulse/lies-damn-algorithm-analysis-donald-trummell/>.) Additionally, algorithms often trade off space for time (performance.) These trade-offs need to be considered as well. Let’s explore one interesting example of space-time trade-off, and some guidelines in making the trade-offs.

## The Interviewer’s Problem

We have a log file of telephone calls. The log has a sequence of unordered call times, each call having a start and end time recorded to the nearest minute. The log sequence initially spans a day of calls. We need to find the maximum number of operators required to answer those calls during the peak portion of a day. Assuming one operator per active call, then the number of required operators corresponds to the maximum number of simultaneously active calls.

## Solution Exercise

We explored two general algorithms: “Binning” and “Event monitoring”. Binning uses the time interval of interest, a day in this case, divided into one minute intervals, to create counters (bins) that span the interval at the desired granularity. Binning records the count of active calls during each minute of the day in the 1,440 bins. The maximum call count of this bin collection is the desired maximum active call value. There are two variants of Binning: dense (an array structure for the bins) and sparse (a Map/Dictionary for bins.)

Event monitoring tracks the active calls for the entire data set and records the maximum call count during processing. Even monitoring assumes a new call starts an “event” and records all the in-progress calls. Any call that ended prior to the start time of the new call is evicted from the active call list, and the new call is added to the list. Tracking the maximum size of the adjusted active call list will yield the maximum number of calls during the day.

Naive Event monitoring using an array list to record active calls has severe performance issues. This is because the entire list has to be examined with each call, and the list entries must be reshuffled after each removal. This performance problem is addressed by using a Java priority queue. *Note that event monitoring requires ordered input (ascending)!*

You may view examples of all four coding approaches in the ***DemoDev*** repository (see references below):

* Sparse Binning: ***SolutionBins.java***
* Dense Binning: ***SolutionBinsLinear.java***
* Event Monitoring: ***SolutionEvents.java***
* Queue-based Event Monitoring: ***SolutionEventsQueue.java***

For the initial problem conditions, all four approaches are nominally of time complexity ***O(n),*** where n is the number of calls in the log. After examining the Binning approach, which has a dependency on a small, fixed number of bins, the interviewer altered the problem. He proposed a much longer time interval, significantly exceeding a day, and therefore greatly increasing the number of bins. He wanted an algorithm not impacted by the time interval of interest. This prompted examining the Event Monitoring approach. These conditions offered an interesting opportunity to illustrate a space-time trade-off.

## Testing Data

Analytical Performance analysis is a very useful tool, and a great exercise to understand the algorithm, but should be backed by actual performance evaluation in the context of expected production usage.

Unfortunately, testing of JVM based applications often run into complexities of garbage collection, threading, and operating system load variation. This means that the predicted and actual performance results may vary for different inputs of the same scale. For example, running performance tests in the Eclipse IDE will be influenced by the memory consumed by Eclipse, as well as background threads performing Eclipse actions.

The first performance challenge was to generate representative data for testing (see ***CallGenerator.java*** in the *DemoDev* repository.) The generated telephone calls where between one and twenty-five minutes in duration, spread over many days, with each day having three peak call periods. This mock test data was created to study the behavior of the code under expected production conditions. A sample of call count distribution for a single day of mock is:

This mock telephone call distribution was replicated over the required multi-day test intervals by advancing the start-end times as needed.

## Testing Protocol

Timed runs were recorded for 1, 10, 20, 40, 50, 60, 70, 80, 90, 100, 110, and 120 days with a call volume of 250,000 calls a day. Both Binning and Event Queue were timed in random order. Timing was run outside the Eclipse IDE as suggested above to minimize uncontrolled variation in results.

## Testing Results Summary

For unsorted input:

* Sparse Binning and Event Monitoring had similar performance characteristics up to 120 days.
* The Sparse Binning algorithm often outperformed Event Monitoring over that interval of interest.

For sorted input however, the Event Monitoring algorithm outperformed the Sparse Binning algorithm (***findTm*** refers to Event search time after sort.)

The fastest approach overall was Linear Binning, and it was faster for both sorted and unsorted input data. The space complexity trade-off is:

* Binning: ***O(m)***, where m = number of time intervals spanning the input date range.
* Event Monitoring: ***O(a)***, where a = maximum number of active calls.

The algorithms offer similar space requirements over the range of interest. The graph below summarizes the performance findings. Note the variations in MAP\_BIN due to object creation and garbage collection.

### Legend: *findTm* refers to the time required by Event Monitoring after the sort, and *sortTm* refers to the time required to sort the unordered input.

## Algorithm Selection

If real-time performance is required, then using sufficient memory to implement dense (Linear) Binning would be justified. If calls are sparsely placed over a large interval, then the Map-based sparse Binning would be a good choice, as it offers better average performance then Event Monitoring. Note the influence of object creation and garbage collection on performance for the map-based approach.

If one can ensure ordered data, or afford the time to sort the input, and space is at a premium, then Even Monitoring may be the best choice. In particular, if the maximum number of active calls is much less than the number of bins required to store call counts, then space is conserved by Event Monitoring.

## Final Observation

Binning is a popular technique that can return much more information than just the maximum number of calls. A study of this nature would try an answer the question: “what is the probability of a missed call given N operators”, and binning would return that information. Also, rarely can log data be relied on for correct ordering. This is particularly true in a distributed environment where logs from multiple servers are merged. Usually a sort is required, so the binning approach is less complicated, performs well, and returns more information from a pass over the data.

## Reference Information

This exercise is located in my GitHub repository ***DemoDev*** at <https://github.com/DonaldET/DemoDev>. The interview coding exercise for this example is in ***DemoDev*** at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-codingexams/dev-topics-evernote-max-operators>. The *analysis* directory has testing output and the code is located in the *src* directory.