Implementing a Collection using Mergesort

Tobias Boege ⟨tobias@gambas-buch.de⟩

Abstract

We will develop the core of a special-purpose Collection class in pure Gambas which holds newly inserted (Key, Value) objects in a fixed number of small buckets. Whenever a search is requested or the bucket sizes reach a threshold, they will be merged into the larger data array that is the backing storage of the Collection. The class is based on the assumption that insertions and searches will happen in great quantities but *separately*. Then, the lazy merge will increase performance compared to gradual Insertionsort. Search, insertion and removal are each $\mathcal{O}(\log n)$ with the intended use pattern.

1 Preliminaries

In this article, we will discover a way to write our own `Collection` (`gb`) class using the Mergesort and Binary search algorithms. Both will be motivated and described in the following two sections.

We will soon need a means to compare the performance of algorithms with respect to the length n of their input – mostly for motivational reasons. Generally, only the magnitude of performance matters, i.e. if the function needs constant time, or linear, polynomial, exponential, etc. time in n. Our focus will be on the average-case performance. If a task is performed in, say, quadratic time in n, we say that the function which performs the task is $\mathcal{O}(n^2)$ without worrying much about this notation.

2 Binary search

Recall what a Collection does. It identifies a Variant value by using a String key. Collection is a typical data container class, so the critical operations are insertion, search and removal.

If we were to search a key String K in a String array, i.e. we want the index[1] of K in the String array, the intuitive approach is to look at all the values until we either found the key or we reached the end of the array in which case the search was unsuccessful. This method clearly needs to evaluate a number of array elements which is proportional to the total number of elements which is $\mathcal{O}(n)$ and, in the context of searching, usually considered bad performance.

Imagine you are searching your socks. $\mathcal{O}(n)$ means that you also have to look in your fridge to be sure to find them. But we can't do better without any more information about the places we could search. However, if we once

[1] *The* index provided that K occurs exactly once in the array.

in our life decided never to put our socks into the fridge, we can be sure that they are not in there and save the time when we search them. Based on the conventions we restrain ourselves to as where we put things, we can also rule out other places and save even more time.

Leaving the metaphor now completely, if we required the array to be sorted in ascending order, we could pick the middle element M and compare the key to it. If $M < K$, we can forget all the elements L in the left half of the array because the array is sorted, so: $L \leq M < K$. Therefore, no L could be equal to K. The same applies for the right half if the key was smaller than M. Of course, if $K = M$ by fortune, we are done.

Binary search divides a sorted array into two halves and decides based on the middle element in which half the searched item lies. This procedure is repeated recursively until the item is found or no elements remain to be searched.

This process is repeated with the remaining half of the array until we found the key or the then-remaining part is empty in which case we must have found two *successive* array elements A, B for which $A < K < B$. Since the array is sorted, K cannot be found.

A Gambas function can readily be written from this point. Exemplarily, we search a value in an Integer array:

```
'' This function returns an index where iFind can be found in aArray. If
'' iFind cannot be found, we return -1 by convention.
Public Function BinarySearch(aArray As Integer[], iFind As Integer) As Integer
  Dim iStart As Integer = 0, iEnd As Integer = aArray.Count
  Dim iMid As Integer

  While iStart < iEnd ' Interval not empty?
    iMid = (iStart + iEnd) / 2
    If aArray[iMid] = iFind Then Return iMid
    If aArray[iMid] < iFind Then
      iStart = iMid + 1
    Else
      iEnd = iMid
    Endif
  Wend
  Return -1
End
```

We call this method *Binary search* because we split the search range into two halves at each iteration. It is easy to see that Binary search is $\mathcal{O}(\log n)$ – quite good. To use this method, however, we need a sorted array.

3 Mergesort

Sorting an array of data is in general a non-trivial task. At least if you want to do it well: we could, naively, compare *any two* elements and so determine the correct ordering of the array; but this is $\mathcal{O}(n^2)$. We can do better.

A good strategy to solve a problem is "divide and conquer": we split our problem into smaller ones, which are easier to solve, and then try to combine the partial solutions to solve the initial, big problem.

If sorting an array is difficult for us, we can suppose that we already have two non-empty sorted partial arrays. "Partial arrays" means that both arrays, when combined, contain exactly the same elements as our target array. The partial arrays, because both are non-empty, are certainly shorter than the array we actually want to sort. All we need to do now is to *merge* them. This part is easy: we begin to iterate over both arrays, let us call them A and B, simultaneously using indices i and j. If $A[i] \leq B[j]$, we need to put $A[i]$ into the joint array and advance i because that element has been consumed. If $B[j] < A[i]$, we take $B[j]$ and advance j. If one of the arrays is at its end, we can put in the entire other array and we are done. By induction, we see that the resulting array must be sorted because A and B were sorted.

The question remains of how we get to these two sorted partial arrays. The answer is *recursion*. If it is still too difficult for us to deal with these partial arrays, we demand partial arrays of second order which are again shorter, and so forth. Because the size of the partial arrays decreases at every step we will eventually reach single-element arrays. These are easy to deal with because they are trivially sorted. From this point, we can move back up the recursion merging our partial arrays until we get the big array sorted. As you might have guessed, this is the *Mergesort* algorithm. It can be proven that it runs in $\mathcal{O}(n \log n)$ and is thus superior to the naive approach.

"The queue's WAAAAAY too long to have everything printed (and sorted) by the time I told them, so I kill all the small jobs so there's only 2 left and I can sort them in no time." — Bastard Operator From Hell

The Gambas code, sorting an Integer array, follows the description straightforwardly:

```
Public Function MergeSort(aArray As Integer[]) As Integer[]
  Dim aA, aB As Integer[]
  Dim iI As Integer = 0, iJ As Integer = 0, iK As Integer = 0
  Dim iMid As Integer

  ' Trivial case
  If aArray.Count = 1 Then Return aArray
  ' Split
  iMid = aArray.Count / 2
  aA = MergeSort(aArray.Copy(0, iMid))
  aB = MergeSort(aArray.Copy(iMid, aArray.Count - iMid))
  ' Merge. We reuse the space in aArray to save the sorted array
  While iI < aA.Count And iJ < aB.Count
    If aA[iI] <= aB[iJ] Then
      aArray[iK] = aA[iI]
      Inc iI
    Else
      aArray[iK] = aB[iJ]
      Inc iJ
    Endif
    Inc iK
  Wend
  If iI = aA.Count Then ' Put the non-consumed array into aA
    aA = aB
    iI = iJ
  Endif
  While iI < aA.Count
    aArray[iK] = aA[iI]
    Inc iI
    Inc iK
  Wend
  Return aArray
End
```

4 Sorting arbitrary objects

Now that we know how to search efficiently and how to prepare our data for the search, we come back to the Collection. The data we want to store is actually (Key, Value) pairs, i.e. objects of a class we will call _Entry. The leading underscore is a Gambas naming convention which denotes that the class is for internal use only. _Entry is essentially only little more than a data structure:

```
Public Key As String
Public Value As Variant

'' Create a new entry
Public Sub _new(Key As String, Value As Variant)
  Me.Key = Key
  Me.Value = Value
End
```

To apply our thoughts from the previous section, we need a way to sort _Entry objects. Let us pause for a moment and ask ourselves what sorting actually is.

By some means we, as a programmer, impose a *total order relation* $\preceq$ on a class of objects which dictates the sequence we have to put a concrete array of objects into when we want to sort it. Because we work towards implementing a Collection – which does not allow equal keys to address different values – we require the total order to be *strict* and use the symbol $\prec$ for this. An array A is said to be *sorted by* $\prec$ when for any two indices $i < j$ over the array the statement $A[i] \prec A[j]$ is true.

Sorting an array means to realise a total order on its elements. A total order is a very natural thing for humans: if the relations of objects are visualised, a total order results in a single continuous line that contains all objects. Thus, if the order is additionally strict, they are ideal to sort an array – there is no ambiguity for the position of any element. Total orders are also called “linear orders”.

As far as it concerns Mergesort, defining a total order[2] boils down to the ability to *compare* objects. Gambas provides a special method `_compare()` for precisely this purpose. This method is given another object of the same class which is to be compared to the current object. The relation is encoded in the return value: 0 means that both objects are equal, -1 that the current object is less and $+1$ means that the current object is greater than the other one. Since the Collection is searched for keys, we need to sort the array by keys. This, in turn, requires us to compare objects by their keys. So this is the natural way of sorting _Entry objects. The _compare() method of the _Entry class is a one-liner then:

[2]The strictness of the relation is not Mergesort's business but our own. In accordance with gb's Collection, we ensure that no duplicate keys are inserted by simply overwriting the old value by the new one when a key is already present.

```
'' Compare two entries' Keys
Public Sub _compare(hOther As _Entry) As Integer
  Return Comp(Key, hOther.Key)
End
```

The Sort() methods of various array classes use exactly this interface to perform their operation.

Someone might wonder now, why we do not simply use these Sort() methods and save the time to write our own Mergesort function. The reason will be given in the next section when we finally design our custom Collection's internals. We will enforce a strong structural property on our data. The internal Sort() implementations use a different sorting algorithm, Quicksort, which cannot take advantage of this property, but Mergesort will.

5 A merging Collection

You may have noticed that Mergesort scales very well when merging large partial arrays into an even larger array. But if we look at the opposite direction – the recursion down, splitting arrays – we see that it is too "small-minded" requiring single-element arrays. Few-elements arrays are already easy and fast to sort using even the naive technique of pairwise comparison, relative to Mergesort's immense overhead of recursive function calls for such small arrays.

A common solution to this problem is to define a lower threshold for the application of Mergesort. If arrays get smaller in the splitting process, another sorting algorithm, such as Insertionsort, kicks in and gets the job done quickly.

Our Collection, named `MergeCollection`, will make use of this strategy and another noteworthy heuristic: we assume that the user of our collection will have temporally separate heavy insertion and search/removal phases in their application. This makes MergeCollection a special-purpose Collection only but enables us to use a system of "bucket" arrays which act like a waiting hall: new objects are dropped there in the order they arrive (i.e. unsorted) and when the system of buckets reaches a certain size, they are Mergesort'd into the very bucket that is the MergeCollection's real backing storage. The same must happen, of course, if data is to be searched or deleted. So the waiting hall strategy will be effective only if insertions and searches are not mixed.

5.1 Thoughts about the merge step

We call the single bucket, that we merge into, M and the waiting hall buckets $B_1, B_2, \ldots, B_m$. M is already sorted and contains $\#M = n$ entries. Assuming the user works with MergeCollection as we intended, then we have insertion in $\mathcal{O}(1)$, *unless* a merge is performed because the buckets' threshold is reached. It makes sense to define this threshold as n/m. So that a merge is performed when all waiting hall buckets together are as big as the already sorted data. The merge step has to sort the m waiting hall buckets which needs, using Mergesort, $\mathcal{O}(m * (n/m \log(n/m))) = \mathcal{O}(n \log(n/m))$. We could eliminate any m because it is a constant in our implementation and a constant is irrelevant to the $\mathcal{O}$ notation because it cannot change the magnitude.

However, for the sake of the little closeness to computing reality we can have when using $\mathcal{O}$ notation, we keep it. Since it *will* be a difference for the man with the stopwatch if we fill up 1000 or just 10 buckets. With the buckets sorted, we can merge them all together. This is done by taking two buckets and merging them, then merging the next bucket with the result and so forth. We need to pay attention to a subtle point here: the waiting hall may contain duplicate keys and we want only the last value to survive. So we cannot just throw entries randomly into the waiting hall and merge them.

The solution is to fill up each bucket B_i until the threshold before we move to B_{i+1}. When we stick to the above merge algorithm, the last inserted instance of a key can overwrite any previous one. A little thought shows that the waiting hall merge is $\mathcal{O}\left(\sum_{i=1}^{m-1} \frac{(i+1)n}{m}\right) = \mathcal{O}\left(\frac{n}{m} * \frac{(m+2)(m-1)}{2}\right) = \mathcal{O}(mn)$. And then we merge that array into M which both contain n elements. So the complexity is $\mathcal{O}(n)$.

The complexity of these steps is sums up to $\mathcal{O}(n+mn+n\log(n/m)) = \mathcal{O}(n(1+m+\log(n/m)))$. But this merge step is performed only at every n-th insertion so actually, on average, we have $\mathcal{O}(1+m+\log(n/m))$ time at each insertion. An asymptotic estimate yields $\mathcal{O}(\log n)$. This is not any worse than if we did a binary search in M and inserted the new entry at the appropriate position – which is called Insertionsort. But in fact, our strategy will prove to be superior in the correct use case because it processes the input elements block-wise, where the waiting hall is the block. The block size is even *growing* as $n = \#M$ grows. And the computer likes block-wise operations better than small scattered ones.[3]

MergeCollection keeps newly inserted entries in an unsorted pile where they wait to be actually merged in. The lazy, block-wise merge strategy has high insertion time peaks – when a merge is due but this is not the case most of the time. The average insertion is efficient.

This also shows why we are not using the built-in Sort() method which uses Quicksort. Our estimates desperately rely on Mergesort's ability to efficiently combine sorted partial arrays. Quicksort was not made for that.

5.2 The MergeCollection class

So that is the idea and here is the relevant[4] code for the MergeCollection class – with inline comments:

```
' Number of buckets in the waiting hall
Private NrBuckets As Integer = 3

' Waiting hall, B_1, ..., B_m
Private $aHall As _Bucket[]
' The collection bucket M
Private $aCollection As _Bucket
' Where to drop the next inserted element
Private $iNextBucket As Integer
```

[3]Besides all the time complexity considerations, we also want to note that the maximum waiting hall size doubles after each merge. This means that the number of merges heavily decreases over time (if no search is requested) but the *space complexity* of the waiting hall, i.e. the memory used by *unsorted* data, grows ad infinitum and duplicate keys may accumulate in there effectively wasting space. This issue is, however, not addressed here.

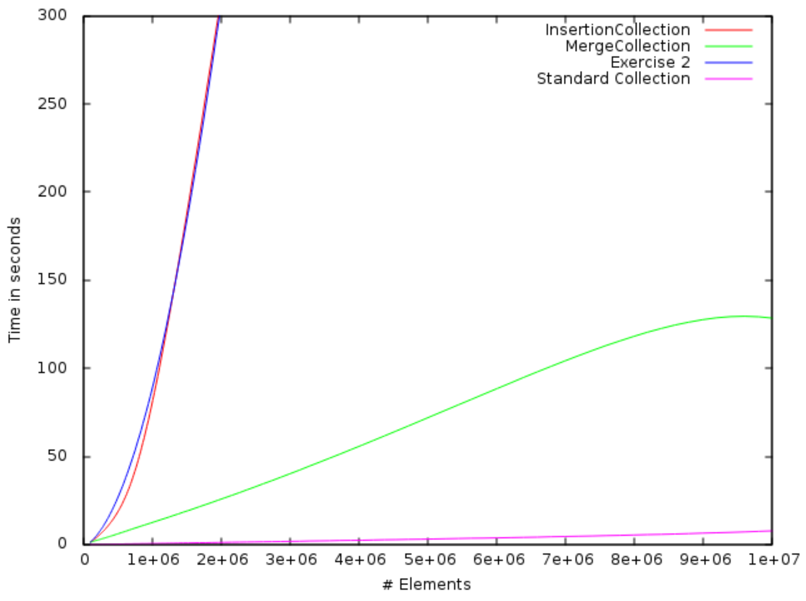
[4]The project accompanying this article contains a fully-featured MergeCollection class as a drop-in replacement for the standard Collection. In this article, the interface compliance is no issue.

```
Public Sub _new()
  ' Allocate our bucket system
  Clear()
End

Public Sub Clear()
  Dim iInd As Integer

  $aCollection = New _Bucket
  $aHall = New _Bucket[](NrBuckets)
  For iInd = 0 To $aHall.Max
    $aHall[iInd] = New _Bucket
  Next
  $iNextBucket = 0
End

'' Merge in all waiting hall buckets
Private Sub Merge()
  Dim hBucket As _Bucket

  If Not $aHall[0].Count Then Return
  For Each hBucket In $aHall
    hBucket.Sort()
    $aCollection.Merge(hBucket)
    hBucket.Clear()
  Next
End

Public Sub _get(Key As String) As Variant
  Dim iInd As Integer

  Merge()
  iInd = $aCollection.Find(Key)
  If iInd = -1 Then Return Null
  Return $aCollection[iInd].Value
End

Public Sub _put(Value As Variant, Key As String)
  If IsNull(Value) Then
    Remove(Key)
    Return
  Endif

  $aHall[$iNextBucket].Add(Key, Value)
  ' If we filled this bucket, move to the next. It is crucial that
  ' the bucket choice is aware of the Merge() algorithm to treat
  ' duplicate keys right.
  If $aHall[$iNextBucket].Count * NrBuckets >= $aCollection.Count Then
    $iNextBucket = ($iNextBucket + 1) Mod NrBuckets
    ' If we filled all the buckets up to the threshold, merge them in
    If Not $iNextBucket Then Merge()
  Endif
End

Public Sub Remove(Key As String)
  Merge()
  $aCollection.Remove(Key)
End
```

This class is pretty easy in terms of the _Bucket class where most of the complexity lies. You will find the Mergesort, Insertionsort and Binary search algorithms in _Bucket and we will not print it here.

5.3 Benchmark

Using the intended use pattern for the MergeCollection, the following benchmark was created. You can find the code in the `MergeCollectionFull` project. The figures were interpolated from at least seven sampling points.



Shown are the times needed to insert a specific amount of elements into a Collection working solely with Insertionsort, the MergeCollection and the standard Collection from gb (which is the near 0 pink line). The blue figure relates to Exercise 2 below. We cannot expect to beat a natively implemented Collection, especially because the standard Collection uses a hash table. We see a solid factor of 20 to 25 between MergeCollection and Collection. But we beat the pure Insertionsort approach.

And MergeCollection does have something over Collection: being a hash table, gb's Collection cannot naturally enumerate the elements in ascending key order. Sometimes this can be useful and we get this property for free with our Mergesort-based collection.

Exercise 1. Prove that the Sort() method found in _Bucket satisfies all criteria about the treatment of duplicate keys. (Hint: induction)

Exercise 2. There is a point in the bucket merge algorithm where Mergesort is not theoretically superior to Array.Sort()'s Quicksort (cf. the last paragraph of "Sorting arbitrary objects").

(a) Where is it?

(b) Assume we would want to change that piece of code to use Array.Sort(). What do we need to take care of when using the (also unstable) Quicksort algorithm?

(c) Despite the priority we should give the *natively implemented* Array.Sort(), why should we stick to our initial implementation (cf. the plot of the changed code's performance). Give your answer using $\mathcal{O}$ notation. (There is a time *and* space complexity penalty.)