Constant-Time Array Initialization in Little Space

Gonzalo Navarro
Dept. of Computer Science, University of Chile, Chile. gnavarro@dcc.uchile.cl

Abstract

We improve a folklore data structure for initializing an array of n elements in constant time, by reducing its space requirement from $2n \lg n$ to $n + o(n)$ bits.

1. Introduction

A problem frequently arising in programming and algorithm design is that one has to initialize a large array to possibly write a small (unknown) fraction of it, and then the cost of initializing the array may outweight that of writing the cells that are really needed. This situation arises, for example, when using a hash table and one has not sufficient knowledge of the amount of insertions that will occur. One may have enough memory to allocate a large enough table, but the price of initializing it may be too high. Another case arises when one marks arbitrary positions in a large array in order to spot duplicates in a large universe, and then the array must be cleaned up for the next iteration (one can go again over the elements to clear their positions, but this may double the processing time).

In his foundational trilogy, Mehlhorn describes a folklore technique to initialize an array in constant time. However, it has the severe drawback of requiring $2n \lg n$ extra bits of space for an array of n elements. Such a space overhead can easily triple the memory footprint of the array, and may be unacceptable when the array is very large (that is, when the initialization problem is relevant). The situation can be even worse if A stores element on a smaller universe. For example, if A is a bit array holding $n = 10^6$ entries, the extra space will be 40 times that required by A!

We obtain a succinct version of this technique. We show that an array can be initialized, read, and written in constant time by using only $n + o(n)$ bits. This result is interesting both from theoretical and practical standpoints, and makes the classical solution much more appealing in practical scenarios.

2. Problem Definition and Our Result

A folklore solution, well-known at least since the seventies, permits initializing an array $A[1, n]$ in constant time (see Aho et al. [1, Ex. 2.12, page 71], or a complete description by Mehlhorn [2, Section III.8.1]).

Definition 1 *An* initializable array *is a data structure $A[1, n]$ that supports the operations $init(A, n, v)$, $read(A, i)$ and $write(A, i, v)$. The first operation initializes $A[i] \leftarrow v$ for $1 \leq i \leq n$; the second obtains $A[i]$ and the third sets $A[i] \leftarrow v$. The size n is fixed at initialization time.*

Theorem 1 *(folklore). An initializable array $A[1, n]$ can be enhanced with structures using $2n\lceil \lg n \rceil$ additional bits of space, so that all its operations take constant time.*

In this paper we largely reduce the extra memory required to initialize A in constant time. More precisely, we obtain the following result.

Theorem 2 *On a RAM machine with word size $w = \Omega(\lg n)$, an initializable array $A[1, n]$ can be enhanced with structures using $n+o(n)$ bits of extra space, so that all its operations init, read and write, take constant time.*

3. The Original Technique

The original folklore technique is as follows. Let $A[1, n]$ be the array we wish to initialize in constant time. We use a second array $B[1, n]$ and a stack $S[1, n]$, both storing indices in $[n]$. An additional variable $0 \leq t \leq n$ tells the current size of S, and variable V stores the initialization value.

Initialization of the whole structure, $init(A, n, v)$, consists of setting $t \leftarrow 0$ and $V \leftarrow v$. The invariant maintained by the structures is as follows:

$$A[i] \text{ is initialized } \iff (1 \leq B[i] \leq t \ \wedge \ S[B[i]] = i)\,,$$

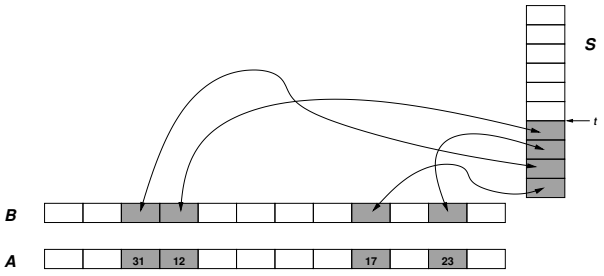

Figure 1. The classical scheme, tripling the space.

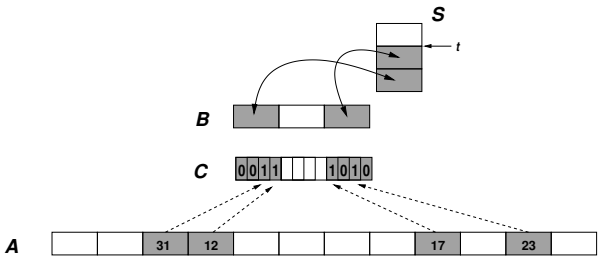

Figure 2. Our compact scheme with one level.

which is immediately correct once we set $t = 0$. The idea is to distinguish initialized entries $A[i]$ because $B[i] = j$ and there is a back pointer $S[j] = i$. Let us take an uninitialized entry $A[i]$. Value $B[i]$ is not initialized either. If $B[i] < 1$ or $B[i] > t$, we know for sure that $A[i]$ is not initialized. Yet it could be that $1 \leq B[i] \leq t$. But then it is not possible that $S[B[i]] = i$ because entry $B[i]$ in S has been used to initialize another entry $A[i']$ and then $S[B[i]] = i' \neq i$.

Operation $read(A, i)$ is as follows. If $A[i]$ is initialized, then it returns $A[i]$, otherwise it returns V. Operation $write(A, i, v)$ is as follows. If $A[i]$ is not initialized, it first sets $t \leftarrow t+1$, $B[i] \leftarrow t$, and $S[t] \leftarrow i$. After the possible initialization, it sets $A[i] \leftarrow v$. It is interesting that one can even uninitialize $A[i]$, by setting $S[B[i]] \leftarrow S[t]$, $B[S[t]] \leftarrow B[i]$, $t \leftarrow t - 1$.

Figure 1 illustrates the basic technique.

4. Reducing Space

We use, instead of array B and stack S, a bit vector $C[1, n]$ so that $C[i] = 1$ iff $A[i]$ has been initialized. This way we require only n bits in addition to A. Of course the problem translates into initializing $C[i] \leftarrow 0$ for all i. We now take advantage of the unit-cost RAM model of computation, where $w \geq \lg n$ must hold because we store numbers up to n in computer words.

As C is stored as a contiguous sequence of bits, let us interpret this sequence as an array $C'[1, n']$ of $n' = \lceil n/w \rceil$ entries, each entry holding a computer word of w bits of C. We can apply now the original solution of Section 3 to C', so that C' can be initialized in constant time (at value $C'[i] = 0$). The extra space on top of C' is $2n' \lg n' \leq 2n$ bits. Together with C', the space overhead of the solution is $3n$ bits. Now, in order to determine whether $A[i]$ is initialized, we just check $C[i]$: We compute $q = i$ div w and $r = i$ mod w and check the bit number $r + 1$ of $C'[q + 1]$. If $C'[q + 1]$ is not yet initialized, we know $C[i] = 0$ and thus $A[i]$ is not yet initialized. To initialize $A[i]$ we set the $(r + 1)$-th bit of $C'[q + 1]$, previously initializing $C'[q + 1] \leftarrow 0$ if needed.

This can be carried further. Instead of directly applying the solution of Section 3 to C', we could recursively use a bitmap of n' bits telling which entries of C' are initialized, compact them into a second array $C''[1, n'']$, $n'' = \lceil n'/w \rceil \approx n/w^2$, and then apply the solution of Section 3 to C''. This time the space overhead is $n + 3n/\lg n$ bits. This can be repeated a constant number of times, to achieve any extra space of the form $\left(\sum_{0 \leq d < h} n/\lg^d n\right) + 3n/\lg^h n$. The price in practice is that the time to access A grows linearly with h. Yet, the space is $n+o(n)$ bits already for $h = 1$.

Figure 2 illustrates our solution using $3n$ bits.

5. Practical Notes

Empirical tests run on an Intel machine using an array of n random integers in $[n]$ show that the classical solution is faster than explicitly running the initialization as long as the number of operations over the array does not exceed 8% of its size. Our $3n$-bit solution ($h = 0$) is faster only up to 2.5%, due to the increased complexity of the operatory. Yet, the classical solution crashes when the array is so large that the extra structures (200% extra space) require swapping, whereas the extra space of our solution (less than 10%) goes unnoticed.

Acknowledgments

We thank Alberto Apostolico for pointing us the earlier reference [1] for the folklore technique of constant-time array initialization.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.