duplicate detection problem

Restated from stackoverflow:

Given:

- array a[1:N] consisting of integer elements in the range 1:N

Is there a way to detect whether the array is a permutation (no duplicates) or whether it has any duplicate elements, in O(N) steps and O(1) space without modifying the original array?

clarification: the array takes space of size N but is a given input, you are allowed a fixed amount of additional space to use.

The stackoverflow crowd has dithered around enough to make me think this is nontrivial. I did find a few papers on it citing a problem originally stated by Berlekamp and Buhler (see "The Duplicate Detection Problem", S. Kamal Abdali, 2003)

permutations algorithms

edited May 23 at 12:37
Community ♦
1 ● 2 ● 3

asked May 20 '10 at 14:51
Jason S
346 ● 1 ● 4 ● 14

What is your model of computation? Such an array (and in particular such a permutation) takes $\mathcal{O}(N \log N)$ space to store, so it would take that much time just to read it. – Noah Stein May 20 '10 at 15:09

So you want this to be faster than sorting (which can be done in $O(N\log(N))$ steps if I am not mistaken)? – Roland Bacher May 20 '10 at 15:18

3 Sorting is not an O(n log n) solution, because it uses much more than O(1) space and/or modifies the array. And this input could be sorted in O(n) time anyway because it's all small integers. – David Eppstein May 20 '10 at 17:26

1 By small I meant polynomially bounded in the input size. Integers in the range 1..n can be sorted by bucket sort in linear time and integers in the range 1..polynomial can be sorted by radix sort in linear time. It's not a question of what's realistically large, it's a question of whether you allow your inputs to be used as array indexes or you artificially pretend your computer can only access them via pairwise comparisons. – David Eppstein May 20 '10 at 20:50

2 Stupid observation: with only O(1) space, you can't actually address the whole array. So you probably want something like "O(1) space, but pointers count as constant space." – David Speyer Jun 2 '10 at 18:27

|

5 Answers

It's at least possible to test whether the input is a permutation with a randomized algorithm that uses O(1) space, always answers "yes" when it is a permutation, and answers "yes" incorrectly when it is not a permutation only with very small probability.

Simply pick a hash function $h(x)$, compute $\sum_{i=1}^n h(i)$, compute $\sum_{i=1}^n h(a[i])$, and compare the two sums.

Ok, some care needs to be used in defining and choosing among an appropriate family of hash functions if you want a rigorous solution (and I suppose we do want one, since we're on mathoverflow not stackoverflow). Probably the simplest way is just to fill another array H with random numbers and let $h(x) = H[x]$, but that is unacceptable because it uses too much space. I'll leave this part as unsolved and state this as a partial answer rather than claiming full rigor at this point.

See also my paper Space-Efficient Straggler Identification in Round-Trip Data Streams via Newton's Identitities and Invertible Bloom Filters which solves a more general problem (if there are O(1) duplicates, say which ones are duplicated, using only O(1) space) with the same lacuna in how the hash functions are defined. It also contains a proof that an algorithm that makes only a single pass over the data cannot solve the problem exactly and deterministically, but of course that doesn't apply to algorithms with random access to the input array.

edited May 20 '10 at 17:43

answered May 20 '10 at 16:05
David Eppstein
15k ● 2 ● 36 ● 83

Why use a hash function when you can use the identity and have correct output every time? Just compute $\sum a[i]$ and compare with $n(n+1)/2$. – Dror Speiser May 20 '10 at 18:12

@Dror: Because sum and compare with n(n+1)/2 does not correctly check for duplicates, e.g. a valid permutation 1,2,3,4,5,6...N vs. 2,2,2,4,5,6,...,N – Jason S May 20 '10 at 18:19

David: +1. I assume this is like primality testing where you can use multiple passes to increase probability? – Jason S May 20 '10 at 20:52

Yes, or just use a hash function with a bigger range. – David Eppstein May 20 '10 at 21:01

3 One hash function that takes $O(1)$ storage and obviously works is $h(x) = 2^x \bmod p$ where p is a large random prime. But that leads to an $O(n \log n)$ time algorithm because it takes $O(\log n)$ time to evaluate $h(x)$ (for integer arguments in the range 1..n) in any reasonable model of machine arithmetic. I'm pretty sure that taking polynomials modulo a prime doesn't work (there are specific inputs that will trick all low-degree polynomials), but that doesn't exhaust the possibilities. – David Eppstein May 23 '10 at 0:01

|

In the complexity theory literature there is a related problem known as the **element distinctness problem**: given a list of n numbers, determine if they are all distinct.

Of course this problem isn't quite the same; one might expect that if you assume all numbers are in the range $\{1, \dots, n\}$ that you might solve the problem more efficiently.

The wikipedia article http://en.wikipedia.org/wiki/Element_distinctness_problem mentions the linear time bucket sort solution for the special case of $\{1, \dots, n\}$. The purpose of my answer is to let you know of a common name for the problem so that maybe your web searches will fare better. Much is known about element distinctness and I am sure that your special case has been studied to death.

answered May 20 '10 at 23:37

Ryan Williams
3,944 ● 17 ● 34

1 thank you -- seems like many of the problems in math / CS have already been solved so much of the problem is just figuring out what they are called – Jason S May 21 '10 at 0:38

This is still an open and interesting problem. The best deterministic algorithm that I know of takes $O(n \log n)$ time and $O(\log n)$ words of space by Munro, Fich and Poblete in Permuting in place. This paper doesn't explicitly mention the problem of detecting if there is a duplicate but the method they develop for permuting in place is directly applicable. It is still possible that there is a true linear time and $O(1)$ words of space solution (either randomised or deterministic).

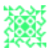
If you simply increase the alphabet size from n the situation changes drastically. Even if you change it to $2n$ the complexity of finding if there is a duplicate is unknown and in particular no near linear time solution is known for small space. The most obvious randomised approach is to hash the elements down to the range $[1, \dots, n]$. You are then left with the problem of trying to distinguish real duplicates from ones created by hash collisions. With full independence it seems you can most likely do this in something like $O(n^{3/2})$ time but I am not sure if this has ever been formally analyzed in published work. However, we can't actually use a hash function with full independence without also using linear space so the problem as before is to show that a hash function family whose members can be represented in small space and which has the desired properties actually exists.

For even larger alphabets of size n^2 there is an existing lower bound for small space algorithms given in Time-space trade-off lower bounds for randomized computation of decision problems. With space $O(\log n)$ bits (or $O(1)$ words) it simplifies to approximately $\Omega(n\sqrt{\log n/ \log\log n})$. This means that no linear time solution is possible in this case.

COMMENT: This should be a comment to David Eppstein's answer but I don't have the points for that. The function $h(x) = 2^x \bmod p$ with p a prime with $O(\log n)$ bits is very interesting. Although it is clear that it takes $\Theta(\log n)$ time to evaluate the hash function once (by repeated squaring, assuming constant time operations on words), is it obvious that it can't be done faster on average when evaluating at n points by some clever method? Consider, for example, an array with the elements in increasing order. In this case it takes only $O(n)$ time to compute all the hash values.

edited Jul 24 '12 at 18:52

answered Jul 16 '12 at 22:59


Raphael
71 ● 1 ● 3

EDIT: This turns out to be not an answer for practical N as well. Given two numbers and the right N, one can play games with the modulo pattern of the two numbers and create two other numbers such that their contribution to the modulo counts replicates that of the two given numbers. Thus a practical solution based on modular arithmetic may need space O(ln N) and a multiplicative time factor of O(ln N). Oops. END EDIT

For arbitrary N, this is not an answer, but for practical N, say N < 2^64, one approach is to consider the residues mod p of the array entries for primes p from 2 up to a sufficient limit, say 60.

If the counts match the expected distribution, then (I think) the list is a permutation if no element lies outside the range [1,P], where P > 2^64 and is the product of the primes from 2 up to 60. In general, the algorithm uses space Q * B and time O(Pi(Q)*N), where Q is the largest prime used, B is the size of N (or of an array element), and Pi(Q) is the number of primes less than or equal to Q. Additionally, pi(Q) is significantly less than ln(N) and Q is not much larger (with respect to N) than pi(Q). For practical N, this approach should suffice.

Gerhard "Ask Me About System Design" Paseman, 2010.05.20

edited May 20 '10 at 22:43

answered May 20 '10 at 20:27

Gerhard Paseman
940 ● 1 ● 7 ● 7

Basically David's approach: we fix M = number of bits storage, and compute the indicator $h = XOR(\text{hash}_M(a[i]))$ where hash_M is a hash function to M bits (eg MD5 masked to M bits). We decide that it is a permutation without repetitions by comparing with the same indicator for the ordered array (1..N). This is order N. And there is a probability of error which should be around $1/2^M$... if I'm not mistaken.

edited Aug 13 '13 at 14:40

Michael Albanese
4,871 ● 3 ● 36 ● 68

answered May 20 '10 at 20:54

leonbloy
216 ● 3 ● 8
