How to tell if an array is a permutation in O(n)?

Input: A *read-only* array of N elements containing integer values from 1 to N (some integer values can appear more than once!). And a memory zone of a **fixed** size (10, 100, 1000 etc - **not** depending on N).

How to tell **in O(n)** if the array represents a permutation?

--What I achieved so far (an answer proved that this was **not** good):--

1. ~~I use the limited memory area to store the sum and the product of the array.~~
2. ~~I compare the sum with **N*(N+1)/2** and the product with **N!**~~

~~I know that if condition (2) is true I **might** have a permutation. I'm wondering if there's a way to prove that condition (2) is sufficient to tell if I have a permutation. So far I haven't figured this out ...~~

arrays algorithm permutation

edited Jul 22 '11 at 22:26
Gilles
65.1k ● 17 ● 141 ● 193

asked May 20 '10 at 10:44
INS
7,099 ● 3 ● 43 ● 79

1 homework? ... .. – Mitch Wheat May 20 '10 at 10:47

3 no, it is purely for fun – INS May 20 '10 at 10:54

3 The storage required for the product `N!`, strictly speaking, depends on `N`. And strictly speaking, you can't multiply `N` numbers in `O(N)`. – polygenelubricants May 20 '10 at 11:01

1 I believe this would be a solution: aperiodic.net/phil/archives/Geekery/… – INS May 20 '10 at 11:03

3 Almost duplicate: stackoverflow.com/questions/177118/… – Eric Bainville May 20 '10 at 11:14

|

16 Answers

I'm very slightly skeptical that there is a solution. Your problem seems to be very close to one posed several years ago in the mathematical literature, with a summary given here ("The Duplicate Detection Problem", S. Kamal Abdali, 2003) that uses cycle-detection -- the idea being the following:

If there is a duplicate, there exists a number `j` between 1 and N such that the following would lead to an infinite loop:

```
x := j;
do
{
   x := a[x];
}
while (x != j);
```

because a permutation consists of one or more subsets S of distinct elements s_0, s_1, ... s_{k-1} where $s_j = a[s_{j-1}]$ for all j between 1 and k-1, and $s_0 = a[s_{k-1}]$, so all elements are involved in cycles -- one of the duplicates would not be part of such a subset.

e.g. if the array = [2, 1, 4, 6, **8**, 7, 9, 3, 8]

then the element in bold at position 5 is a duplicate because all the other elements form cycles: { 2 -> 1, 4 -> 6 -> 7 -> 9 -> 8 -> 3}. Whereas the arrays [2, 1, 4, 6, 5, 7, 9, 3, 8] and [2, 1, 4, 6, 3, 7, 9, 5, 8] are valid permutations (with cycles { 2 -> 1, 4 -> 6 -> 7 -> 9 -> 8 -> 3, 5 } and { 2 -> 1, 4 -> 6 -> 7 -> 9 -> 8 -> 5 -> 3 } respectively).

Abdali goes into a way of finding duplicates. Basically the following algorithm (using Floyd's cycle-finding algorithm) works if you happen across one of the duplicates in question:

```
function is_duplicate(a, N, j)
{
     /* assume we've already scanned the array to make sure all elements
        are integers between 1 and N */
     x1 := j;
     x2 := j;
     do
     {
         x1 := a[x1];
         x2 := a[x2];
         x2 := a[x2];
     } while (x1 != x2);
```

```
    /* stops when it finds a cycle; x2 has gone around it twice,
       x1 has gone around it once.
       If j is part of that cycle, both will be equal to j. */
    return (x1 != j);
}
```

The difficulty is I'm not sure your problem as stated matches the one in his paper, and I'm also not sure if the method he describes runs in O(N) or uses a fixed amount of space. A potential counterexample is the following array:

[3, 4, 5, 6, 7, 8, 9, 10, ... N-10, N-9, N-8, N-7, N-2, N-5, N-5, N-3, N-5, N-1, N, 1, 2]

which is basically the identity permutation shifted by 2, with the elements [N-6, N-4, and N-2] replaced by [N-2, N-5, N-5]. This has the correct sum (not the correct product, but I reject taking the product as a possible detection method since the space requirements for computing N! with arbitrary precision arithmetic are O(N) which violates the spirit of the "fixed memory space" requirement), and if you try to find cycles, you will get cycles { 3 -> 5 -> 7 -> 9 -> ... N-7 -> N-5 -> N-1 } and { 4 -> 6 -> 8 -> ... N-10 -> N-8 -> N-2 -> N -> 2}. The problem is that there could be up to N cycles, (identity permutation has N cycles) each taking up to O(N) to find a duplicate, and you have to keep track somehow of which cycles have been traced and which have not. I'm skeptical that it is possible to do this in a fixed amount of space. But maybe it is.

This is a heavy enough problem that it's worth asking on mathoverflow.net (despite the fact that most of the time mathoverflow.net is cited on stackoverflow it's for problems which are too easy)

---

edit: I did ask on mathoverflow, there's some interesting discussion there.

edited Apr 13 at 12:57
Community ♦
1 ● 1

answered May 20 '10 at 14:26
Jason S
96.1k ● 125 ● 444 ● 760

This algorithm in the paper requires an array of size n+1, so that it always contains at least one duplicate. This is not the same problem as the OP. Perhaps the algorithm can be adapted, but it cannot be used verbatim. – Jules May 20 '10 at 16:15

This is impossible to do in O(1) space, at least with a single-scan algorithm.

Proof

Suppose you have processed N/2 of the N elements. Assuming the sequence is a permutation then, given the state of the algorithm, you should be able to figure out the set of N/2 remaining elements. If you can't figure out the remaining elements, then the algorithm can be fooled by repeating some of the old elements.

There are N choose N/2 possible remaining sets. Each of them must be represented by a distinct internal state of the algorithm, because otherwise you couldn't figure out the remaining elements. However, it takes logarithmic space to store X states, so it takes BigTheta(log(N choose N/2)) space to store N choose N/2 states. That values grows with N, and therefore the algorithm's internal state *can not* fit in O(1) space.

More Formal Proof

You want to create a program P which, given the final N/2 elements and the internal state of the linear-time-constant-space algorithm after it has processed N/2 elements, determines if the entire sequence is a permutation of 1..N. There is no time or space bound on this secondary program.

Assuming P exists we can create a program Q, taking only the internal state of the linear-time-constant-space algorithm, which determines the necessary final N/2 elements of the sequence (if it was a permutation). Q works by passing P every possible final N/2 elements and returning the set for which P returns true.

However, because Q has N choose N/2 possible outputs, it must have at least N choose N/2 possible inputs. That means the internal state of the original algorithm must store at least N choose N/2 states, requiring BigTheta(log N choose N/2), which is greater than constant size.

Therefore the original algorithm, which does have time and space bounds, also can't work correctly if it has constant-size internal state.

[I think this idea can be generalized, but thinking isn't proving.]

Consequences

BigTheta(log(N choose N/2)) is equal to BigTheta(N). Therefore just using a boolean array and ticking values as you encounter them is (probably) space-optimal, and time-optimal too since it takes linear time.

edited May 20 '10 at 18:14

answered May 20 '10 at 14:35
Craig Gidney
11.5k ● 2 ● 42 ● 102

I disagree with your approach. The phrases "you should be able to figure out the set of N/2 remaining elements" and "the algorithm can be fooled by repeating some of the old elements." are vague... if by the former you mean producing a set of the N/2 remaining elements, that is not a requirement of the problem. – Jason S May 20 '10 at 14:45

Why should you be able to figure out the set of N/2 remaining elements? All you need to say is that you have membership in the set of permutations (at the end) within the set of {1..N}^N. – Rex Kerr May 20 '10 at 14:47

What I meant is that, given the internal state of the algorithm, a program without bounded time and space must be able to determine the final N/2 elements. Equivalently, some program given the internal state and the final N/2 elements of the sequence must be able to determine if the entire sequence forms a permutation. [I removed the bounds to create that equivalence.] If an unbounded program can't do it when given the constant-sized internal state, then clearly the original bounded program also can't do it. – Craig Gidney May 20 '10 at 14:57

@JasonS I clarified the post. – Craig Gidney May 20 '10 at 15:12

2 You have proven that the problem is *not subdividable*, but not that it can't be solved in `O(N)` time. How do you know that there exists no strategy where at `N/2` of the way through the list, you might still need to revisit the earlier part of the list to process the rest? As long as you do it rarely enough, it could still be `O(N)`. – Rex Kerr May 20 '10 at 15:20

|

I doubt you would be able to prove that ;)

```
(1, 2, 4, 4, 4, 5, 7, 9, 9)
```

I think that more generally, this problem isn't solvable by processing the numbers in order. Suppose you are processing the elements in order and you are halfway the array. Now the state of your program has to somehow reflect which numbers you've encountered so far. This requires at least O(n) bits to store.

edited May 20 '10 at 12:17

answered May 20 '10 at 11:09

Jules
4,645 ● 18 ● 36

Thanks! Rules that solution out. – INS May 20 '10 at 11:11

2 This is more for a comment than an answer, as it doesn't actually answer the question. – Joren May 20 '10 at 11:29

1 I agree, but it does rule out half of the "answers" further down as well as the approach that the OP was taking. So I believe it does solve part of the problem: you don't have to keep looking for a way to solve it by processing the elements in order. – Jules May 20 '10 at 15:48

This isn't going to work due to the complexity being given as a function of N rather than M, implying that N >> M

This was my shot at it, but for a bloom filter to be useful, you need a big M, at which point you may as well use simple bit toggling for something like integers

http://en.wikipedia.org/wiki/Bloom_filter

For each element in the array Run the k hash functions Check for inclusion in the bloom filter If it is there, there is a probability you've seen the element before If it isn't, add it

When you are done, you may as well compare it to the results of a 1..N array in order, as that'll only cost you another N.

Now if I haven't put enough caveats in. It isn't 100%, or even close since you specified complexity in N, which implies that N >> M, so fundamentally it won't work as you have specified it.

BTW, the false positive rate for an individual item should be e = 2^(-m/(n*sqrt(2)))

Which monkeying around with will give you an idea how big M would need to be to be acceptable.

answered May 20 '10 at 12:58

McBeth
1,032 ● 7 ● 12

Wouldn't that be O(n^2)? You say 'For each element ... compare it to the results ... that'll only cost you another N'. So N elements and then an additional N cost per element, N^2? – samoz May 20 '10 at 13:05

You skipped the "When you are done" bit. The final check is totally optional and would occur after the loop – McBeth May 21 '10 at 0:41

You might be able to do this in randomized `O(n)` time and constant space by computing `sum(x_i)` and `product(x_i)` modulo a bunch of different randomly chosen constants C of size `O(n)`. This basically gets you around the problem that `product(x_i)` gets too large.

There's still a lot of open questions, though, like if `sum(x_i)=N(N+1)/2` and `product(x_i)=N!` are sufficient conditions to guarantee a permutation, and what is the chance that a non-permutation generates a false positive (I would hope ~1/C for each C you try, but maybe not).

answered May 20 '10 at 19:57
Keith Randall
20.6k ● 1 ● 24 ● 47

I don't know how to do it in O(N), or even if it can be done in O(N). I know that it can be done in O(N log N) if you (use an appropriate) sort and compare.

That being said, there are many O(N) techniques that can be done to show that one is NOT a permutation of the other.

1. Check the length. If unequal, obviously not a permutation.
2. Create an XOR fingerprint. If the value of all the elements XOR'ed together does not match, then it can not be a permutation. A match would however be inconclusive.
3. Find the sum of all elements. Although the result may overflow, that should not be a worry when matching this 'fingerprint'. If however, you did a checksum that involved multiplying then overflow would be an issue.

Hope this helps.

answered May 20 '10 at 12:48
Sparky
8,216 ● 3 ● 18 ● 24

---

it's a permutation if and only if there are no duplicate values in the array, should be easy to check that in O(N)

answered May 20 '10 at 10:50
Chris Card
2,516 ● 12 ● 14

And how do I do that in O(n) with the restrictions above?:) – INS May 20 '10 at 10:51

4 This reminds me of Fermet's Last Theorem :P – Rubys May 20 '10 at 13:13

sorry, I missed the space restriction – Chris Card May 20 '10 at 13:24

Depending on how much space you have, relative to N, you might try using hashing and buckets.

That is, iterate over the entire list, hash each element, and store it in a bucket. You'll need to find a way to reduce bucket collisions from the hashes, but that is a solved problem.

If an element tries to go into a bucket with an item identical to it, it is a permutation.

This type of solution would be O(N) as you touch each element only once.

However, the problem with this is whether space M is larger than N or not. If M > N, this solution will be fine, but if M < N, then you will not be able to solve the problem with 100% accuracy.

answered May 20 '10 at 11:32
samoz
24.2k ● 46 ● 119 ● 174

Given that the question is O(N) time complexity with O(1) space complexity, there is by definition a big enough N where M < N. – Ants Aasma May 20 '10 at 11:48

@Ants Agreed, but maybe that O(1) space is on the order of gigabytes and N is much smaller. If this is known, he could use my solution. But agreed, this does require knowing a lot of information at the start of things. – samoz May 20 '10 at 13:03

1 The whole definition of the big-O concept is that N is large enough that the complexity class dominates everything else. Big O is always a theoretical exercise, practical considerations like how many gigabytes is available matters when solving real instances of a problem. – Ants Aasma May 20 '10 at 21:58

```
int solution(int A[], int N) {
  int i,j,count=0, d=0, temp=0,max;
  for(i=0;i<N-1;i++) {
    for(j=0;j<N-i-1;j++) {
      if(A[j]>A[j+1]) {
        temp = A[j+1];
        A[j+1] = A[j];
        A[j] = temp;
      }
    }
  }
  max = A[N-1];
  for(i=N-1;i>=0;i--) {
    if(A[i]==max) {
      count++;
    }
    else {
      d++;
    }
    max = max-1;
  }
  if(d!=0) {
    return 0;
  }
  else {
    return 1;
  }
}
```

edited Sep 5 '15 at 20:55
YoungHobbit
9,349 ●7 ●28 ●50

answered Sep 5 '15 at 19:45
shubh_
1

Alright, this is different, but it appears to work!

I ran this test program (C#):

```
    static void Main(string[] args) {
        for (int j = 3; j < 100; j++) {
            int x = 0;
            for (int i = 1; i <= j; i++) {
                x ^= i;
            }
            Console.WriteLine("j: " + j + "\tx: " + x + "\tj%4: " + (j % 4));
        }
    }
```

Short explanation: x is the result of all the XORs for a single list, i is the element in a particular list, and j is the size of the list. Since all I'm doing is XOR, the order of the elements don't matter. But I'm looking at what correct permutations look like when this is applied.

If you look at j%4, you can do a switch on that value and get something like this:

```
    bool IsPermutation = false;
    switch (j % 4) {
        case 0:
            IsPermutation = (x == j);
            break;
        case 1:
            IsPermutation = (x == 1);
            break;
        case 2:
            IsPermutation = (x == j + 1);
            break;
        case 3:
            IsPermutation = (x == 0);
            break;
    }
```

Now I acknowledge that this probably requires some fine tuning. It's not 100%, but it's a good easy way to get started. Maybe with some small checks running throughout the XOR loop, this could be perfected. Try starting somewhere around there.

answered Oct 31 '10 at 17:48
Corey Ogburn
9,793 ●15 ●71 ●141

Thanks, I will have a closer look at this. – INS Nov 3 '10 at 20:25

Java solution below answers question partly. Time complexity I believe is O(n). (This belief based on the fact that solution doesn't contains nested loops.) About memory -- not sure. Question appears first on relevant requests in google, so it probably can be useful for somebody.

```
public static boolean isPermutation(int[] array) {
    boolean result = true;
    array = removeDuplicates(array);
    int startValue = 1;
    for (int i = 0; i < array.length; i++) {
        if (startValue + i  != array[i]){
            return false;
        }
    }
    return result;
}
public static int[] removeDuplicates(int[] input){
    Arrays.sort(input);
    List<Integer> result = new ArrayList<Integer>();
    int current = input[0];
    boolean found = false;

    for (int i = 0; i < input.length; i++) {
        if (current == input[i] && !found) {
            found = true;
        } else if (current != input[i]) {
            result.add(current);
            current = input[i];
            found = false;
        }
    }
    result.add(current);
    int[] array = new int[result.size()];
    for (int i = 0; i < array.length ; i ++){
        array[i] = result.get(i);
    }
    return array;
}
public static void main (String ... args){
    int[] input = new int[] { 4,2,3,4,1};
    System.out.println(isPermutation(input));
    //output true
    input = new int[] { 4,2,4,1};
    System.out.println(isPermutation(input));
    //output false
}
```

answered Apr 18 '15 at 14:30

yurin
1,523 ● 19 ● 20

---

it looks like asking to find duplicate in array with stack machine.

it sounds impossible to know the full history of the stack , while you extract each number and have limited knowledge of the numbers that were taken out.

answered Nov 14 '10 at 12:19

none
1,729 ● 10 ● 41 ● 86

Here's **proof** it can't be done:

Suppose by some artifice you have detected no duplicates in all but the last cell. Then the problem reduces to checking if that last cell contains a duplicate.

If you have **no** structured representation of the problem state so far, then you are reduced to performing a linear search over the entire previous input, for EACH cell. It's easy to see how this leaves you with a quadratic-time algorithm.

Now, suppose through some clever data structure that you actually know which number you expect to see last. Then certainly that knowledge takes at least enough bits to store the number you seek -- perhaps one memory cell? But there is a second-to-last number and a second-to-last sub-problem: then you must similarly represent a set of two possible numbers yet-to-be-seen. This certainly requires more storage than encoding only for one remaining number. By a progression of similar arguments, the size of the state must grow with the size of the problem, unless you're willing to accept a quadratic-time worst-case.

This is the time-space trade-off. You can have quadratic time and constant space, or linear time and linear space. You cannot have linear time and constant space.

answered Aug 14 '11 at 7:26

Ian
3,146 ● 1 ● 11 ● 15

First, an information theoretic reason why this may be possible. We can trivially check that the numbers in the array are in bounds in O(N) time and O(1) space. To specify any such array of in-bounds numbers requires `N log N` bits of information. But to specify a permutation requires approximately `(N log N) - N` bits of information (Stirling's approximation). Thus, if we could acquire `N` bits of information during testing, we might be able to know the answer. This is trivial to do in `N` time (in fact, with `M` static space we can pretty easily acquire `log M` information per step, and under special circumstances we can acquire `log N` information).

On the other hand, we only get to store something like `M log N` bits of information in our static storage space, which is presumably much less than `N`, so it depends greatly what the shape of the decision surface is between "permutation" and "not".

I think that this is *almost* possible but not quite given the problem setup. I think one is "supposed" to use the cycling trick (as in the link that Iulian mentioned), but the key assumption of having a tail in hand fails here because you *can* index the last element of the array with a permutation.

answered May 20 '10 at 15:14
Rex Kerr
140k ● 18 ● 258 ● 363

The sum and the product will not guarantee the correct answer, since these hashes are subject to collisions, i.e. different inputs might potentially produce identical results. If you want a perfect hash, a single-number result that actually fully describes the numerical composition of the array, it might be the following.

Imagine that for any number `i` in `[1, N]` range you can produce a unique prime number `P(i)` (for example, `P(i)` is the i-th prime number). Now all you need to do is calculate the product of all `P(i)` for all numbers in your array. The product will fully and unambiguously describe the composition of your array, disregarding the ordering of values in it. All you need to do is to precalculate the "perfect" value (for a permutation) and compare it with the result for a given input :)

Of course, the algorithm like this does not immediately satisfy the posted requirements. But at the same time it is intuitively too generic: it allows you to detect a permutation of absolutely *any* numerical combination in an array. In your case you need to detect a permutation of a specific combination `1, 2, ..., N`. Maybe this can somehow be used to simplify things... Probably not.

answered May 20 '10 at 22:00
AnT
232k ● 29 ● 363 ● 596

Check out the following solution. It uses O(1) *additional* space. It alters the array during the checking process, but returns it back to its initial state at the end.

The idea is:

1. Check if any of the elements is out of the range [1, n] => O(n).
2. Go over the numbers in order (all of them are now assured to be in the range [1, n]), and for each number x (e.g. 3):
   - go to the x'th cell (e.g. a[3]), if it's negative, then someone already visited it before you => Not permutation. Otherwise (a[3] is positive), multiply it by -1. => O(n).
3. Go over the array and negate all negative numbers.

This way, we know for sure that all elements are in the range [1, n], and that there are no duplicates => The array is a permutation.

```
int is_permutation_linear(int a[], int n) {
    int i, is_permutation = 1;

    // Step 1.
    for (i = 0; i < n; ++i) {
        if (a[i] < 1 || a[i] > n) {
            return 0;
        }
    }

    // Step 2.
    for (i = 0; i < n; ++i) {
        if (a[abs(a[i]) - 1] < 0) {
            is_permutation = 0;
            break;
        }
        a[i] *= -1;
    }

    // Step 3.
    for (i = 0; i < n; ++i) {
        if (a[i] < 0) {
            a[i] *= -1;
        }
    }

    return is_permutation;
}
```

Here is the complete program that tests it:

```
/*
 * is_permutation_linear.c
 *
 *  Created on: Dec 27, 2011
 *      Author: Anis
 */

#include <stdio.h>

int abs(int x) {
    return x >= 0 ? x : -x;
}
```

```
int is_permutation_linear(int a[], int n) {
    int i, is_permutation = 1;

    for (i = 0; i < n; ++i) {
        if (a[i] < 1 || a[i] > n) {
            return 0;
        }
    }

    for (i = 0; i < n; ++i) {
        if (a[abs(a[i]) - 1] < 0) {
            is_permutation = 0;
            break;
        }
        a[abs(a[i]) - 1] *= -1;
    }

    for (i = 0; i < n; ++i) {
        if (a[i] < 0) {
            a[i] *= -1;
        }
    }

    return is_permutation;
}

void print_array(int a[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%2d ", a[i]);
    }
}

int main() {
    int arrays[9][8] = { { 1, 2, 3, 4, 5, 6, 7, 8 },
                         { 8, 6, 7, 2, 5, 4, 1, 3 },
                         { 0, 1, 2, 3, 4, 5, 6, 7 },
                         { 1, 1, 2, 3, 4, 5, 6, 7 },
                         { 8, 7, 6, 5, 4, 3, 2, 1 },
                         { 3, 5, 1, 6, 8, 4, 7, 2 },
                         { 8, 3, 2, 1, 4, 5, 6, 7 },
                         { 1, 1, 1, 1, 1, 1, 1, 1 },
                         { 1, 8, 4, 2, 1, 3, 5, 6 } };
    int i;

    for (i = 0; i < 9; i++) {
        printf("array: ");
        print_array(arrays[i], 8);
        printf("is %spermutation.\n",
               is_permutation_linear(arrays[i], 8) ? "" : "not ");
        printf("after: ");
        print_array(arrays[i], 8);
        printf("\n\n");

    }

    return 0;
}
```

And its output:

```
array:  1  2  3  4  5  6  7  8 is permutation.
after:  1  2  3  4  5  6  7  8

array:  8  6  7  2  5  4  1  3 is permutation.
after:  8  6  7  2  5  4  1  3

array:  0  1  2  3  4  5  6  7 is not permutation.
after:  0  1  2  3  4  5  6  7

array:  1  1  2  3  4  5  6  7 is not permutation.
after:  1  1  2  3  4  5  6  7

array:  8  7  6  5  4  3  2  1 is permutation.
after:  8  7  6  5  4  3  2  1

array:  3  5  1  6  8  4  7  2 is permutation.
after:  3  5  1  6  8  4  7  2

array:  8  3  2  1  4  5  6  7 is permutation.
after:  8  3  2  1  4  5  6  7

array:  1  1  1  1  1  1  1  1 is not permutation.
after:  1  1  1  1  1  1  1  1

array:  1  8  4  2  1  3  5  6 is not permutation.
after:  1  8  4  2  1  3  5  6
```

edited Jan 18 '12 at 12:07

answered Dec 26 '11 at 23:04

Anis Abboud

551 ● 1 ● 6 ● 16