# Easy interview question got harder: given numbers 1..100, find the missing number(s)



I had an interesting job interview experience a while back. The question started really easy:

> **Q1**: We have a bag containing numbers `1`, `2`, `3`, …, `100`. Each number appears exactly once, so there are 100 numbers. Now one number is randomly picked out of the bag. Find the missing number.

I've heard this interview question before, of course, so I very quickly answered along the lines of:

> **A1**: Well, the sum of the numbers `1 + 2 + 3 + … + N` is `(N+1)(N/2)` (see Wikipedia: sum of arithmetic series). For `N = 100`, the sum is `5050`.
>
> Thus, if all numbers are present in the bag, the sum will be exactly `5050`. Since one number is missing, the sum will be less than this, and the difference is that number. So we can find that missing number in `O(N)` time and `O(1)` space.

At this point I thought I had done well, but all of a sudden the question took an unexpected turn:

> **Q2**: That is correct, but now how would you do this if *TWO* numbers are missing?

I had never seen/heard/considered this variation before, so I panicked and couldn't answer the question. The interviewer insisted on knowing my thought process, so I mentioned that perhaps we can get more information by comparing against the expected product, or perhaps doing a second pass after having gathered some information from the first pass, etc, but I really was just shooting in the dark rather than actually having a clear path to the solution.

The interviewer did try to encourage me by saying that having a second equation is indeed one way to solve the problem. At this point I was kind of upset (for not knowing the answer before hand), and asked if this is a general (read: "useful") programming technique, or if it's just a trick/gotcha answer.

The interviewer's answer surprised me: you can generalize the technique to find 3 missing numbers. In fact, you can generalize it to find *k* missing numbers.

> **Qk**: If exactly *k* numbers are missing from the bag, how would you find it efficiently?

This was a few months ago, and I still couldn't figure out what this technique is. Obviously there's a `Ω(N)` time lower bound since we must scan all the numbers at least once, but the interviewer insisted that the *TIME* and *SPACE* complexity of the solving technique (minus the `O(N)` time input scan) is defined in *k* not *N*.

So the question here is simple:

- How would you solve **Q2**?
- How would you solve **Q3**?
- How would you solve **Qk**?

### Clarifications

- Generally there are *N* numbers from 1..*N*, not just 1..100.
- I'm not looking for the obvious set-based solution, e.g. using a bit set, encoding the presence/absence each number by the value of a designated bit, therefore using `O(N)` bits in additional space. We can't afford any additional space proportional to *N*.

- I'm also not looking for the obvious sort-first approach. This and the set-based approach are worth mentioning in an interview (they are easy to implement, and depending on *N*, can be very practical). I'm looking for the Holy Grail solution (which may or may not be practical to implement, but has the desired asymptotic characteristics nevertheless).

So again, of course you must scan the input in `O(N)`, but you can only capture small amount of information (defined in terms of *k* not *N*), and must then find the *k* missing numbers somehow.

algorithm | math

edited Aug 16 '10 at 11:46

asked Aug 16 '10 at 10:26

polygenelubricants
**150k** 48 349 505

**95** @polygenelubricants - so what you are saying here, that you expected to know beforehand **any** problem that they might pose to you?? And panicked because this wasn't so? Ok, does this sound a realistic expectation to you? – Dimitris Andreou Aug 16 '10 at 11:35

**52** Did you get the job? :) – Erik B Aug 16 '10 at 12:58

**29** @Erik: This was a screening question at the job fair, I got an invite for an on-site interview afterward. @Dimitris: I have dreams, and regardless of whether or not they're realistic, I must work toward them. – polygenelubricants Aug 16 '10 at 13:08

**77** Panicking, getting upset and finally criticizing the worth of the question, when you realize you can't answer it, has no doubt given your interviewer exactly the sort of insight they wanted. The reason they insisted on knowing your thought process is because that is *far* more important in judging your suitability for a job than getting the question right. – Ash Aug 28 '10 at 6:31

**4** The solution of summing the numbers requires log(N) space unless you consider the space requirement for an unbounded integer to be O(1). But if you allow for unbounded integers, then you have as much space as you want with just one integer. – Udo Klein Apr 10 '13 at 5:53

show **17** more comments

## 30 Answers

Here's a summary of Dimitris Andreou's link.

Remember sum of i-th powers, where i=1,2,..,k. This reduces the problem to solving the system of equations

$a_1 + a_2 + ... + a_k = b_1$

$a_1^2 + a_2^2 + ... + a_k^2 = b_2$

...

$a_1^k + a_2^k + ... + a_k^k = b_k$

Using Newton's identities, knowing $b_i$ allows to compute

$c_1 = a_1 + a_2 + ... a_k$

$c_2 = a_1a_2 + a_1a_3 + ... + a_{k-1}a_k$

...

$c_k = a_1a_2 ... a_k$

If you expand the polynomial $(x-a_1)...(x-a_k)$ the coefficients will be exactly $c_1$, ..., $c_k$ - see Viète's formulas. Since every polynomial factors uniquely (ring of polynomials is an Euclidean domain), this means $a_i$ are uniquely determined, up to permutation.

This ends a proof that remembering powers is enough to recover the numbers. For constant k, this is a good approach.

However, when k is varying, the direct approach of computing $c_1,...,c_k$ is prohibitely expensive, since e.g. $c_k$ is the product of all missing numbers, magnitude n!/(n-k)!. To overcome this, perform computations in $Z_q$ field, where q is a prime such that n <= q < 2n - it exists by Bertrand's postulate. The proof doesn't need to be changed, since the formulas still hold, and factorization of polynomials is still unique. You also need an algorithm for factorization over finite fields, for example the one by Berlekamp or Cantor-Zassenhaus.

High level pseudocode for constant k:

- Compute i-th powers of given numbers
- Subtract to get sums of i-th powers of unknown numbers. Call the sums $b_i$.
- Use Newton's identities to compute coefficients from $b_i$; call them $c_i$. Basically, $c_1 = b_1$; $c_2 = (c_1b_1 - b_2)/2$; see Wikipedia for exact formulas
- Factor the polynomial $x^k-c_1x^{k-1} + ... + c_k$.

- The roots of the polynomial are the needed numbers $a_1$, ..., $a_k$.

For varying k, find a prime n <= q < 2n using e.g. Miller-Rabin, and perform the steps with all numbers reduced modulo q.

As Heinrich Apfelmus commented, instead of a prime q you can use q=2 ⌈log n⌉ and perform arithmetic in finite field.

edited Aug 16 '10 at 14:11

answered Aug 16 '10 at 12:13

sdcvvc
**15.8k** 3 39 77

**13** Thanks for doing this! – Dimitris Andreou Aug 16 '10 at 12:38

**2** You don't have to use a prime field, you can also use `q = 2^(log n)` . (How did you make the super- and subscripts?!) – Heinrich Apfelmus Aug 16 '10 at 12:45

**3** Also, you can calculate the c_k on the fly, without using the power sums, thanks to the formula $c^{k+1}_m = c^k_{m+1} + c^k_m x_{k+1}$ where the superscript $k$ denotes the number of variables and $m$ the degree of the symmetric polynomial. – Heinrich Apfelmus Aug 16 '10 at 12:50

**18** +1 This is really, really clever. At the same time, it's questionable, whether it's really worth the effort, or whether (parts of) this solution to a quite artificial problem can be reused in another way. And even if this were a real world problem, on many platforms the most trivial `O(N^2)` solution will probably possibly outperform this beauty for even reasonably high `N` . Makes me think of this: tinyurl.com/c8fwgw Nonetheless, great work! I wouldn't have had the patience to crawl through all the math :) – back2dos Aug 16 '10 at 13:52

**36** I think this is a wonderful answer. I think this also illustrates how poor of an interview question it would be to extend the missing numbers beyond one. Even the first is kind of a gotchya, but it's common enough that it basically shows "you did some interview prep." But to expect a CS major to know go beyond k=1 (especially "on the spot" in an interview) is a bit silly. – corsiKa Mar 25 '11 at 21:03

show **9** more comments



You will find it by reading the couple of pages of *Muthukrishnan - Data Stream Algorithms: Puzzle 1: Finding Missing Numbers*. **It shows exactly the generalization you are looking for**. Probably this is what your interviewer read and why he posed these questions.

Now, if only people would start deleting the answers that are subsumed or superseded by Muthukrishnan's treatment, and make this text easier to find. :)

**Also see sdcvvc's directly related answer**, which also includes pseudocode (hurray! no need to read those tricky math formulations :)) (thanks, great work!).

edited May 29 '14 at 12:43

Peter Mortensen
**7,808** 8 55 90

answered Aug 16 '10 at 11:26

Dimitris Andreou
**5,681** 1 17 28

**6** How do you translate *that* into code?!? – Eldelshell Aug 16 '10 at 12:05

Oooh... That's interesting. I have to admit I got a bit confused by the maths but I was jsut skimming it. Might leave it open to look at more later. :) And +1 to get this link more findable. ;-) – Chris Aug 16 '10 at 12:21

**2** The google books link doesn't work for me. Here a better version [PostScript File]. – Heinrich Apfelmus Aug 16 '10 at 12:31

**4** Wow. I didn't expect this to get upvoted! Last time I posted a reference to the solution (Knuth's, in that case) instead of trying to solve it myself, it was actually downvoted: stackoverflow.com/questions/3060104/… The librarian inside me rejoices, thanks :) – Dimitris Andreou Aug 16 '10 at 12:33

**2** Please read the following as the answers provided here are ridiculous: stackoverflow.com/questions/4406110/… – Matthieu N. Dec 26 '10 at 9:12

show **1** more comment

---

We can solve Q2 by summing both the numbers themselves, and the *squares* of the numbers.

We can then reduce the problem to

```
k1 + k2 = x
k1^2 + k2^2 = y
```

Where `x` and `y` are how far the sums are below the expected values.

Substituting gives us:

```
(x-k2)^2 + k2^2 = y
```

Which we can then solve to determine our missing numbers.

answered Aug 16 '10 at 10:37

Anon.
**27.8k** 1 46 73

**1** +1; I've tried the formula in Maple for select numbers and it works. I still couldn't convince myself WHY it works, though. – polygenelubricants Aug 16 '10 at 11:12

@polygenelubricants: Rearranging the first equation gives `k1 = x - k2`, and we can substitute that into the second equation. You can also generalize this to higher k - though it becomes computationally expensive quite rapidly. – Anon. Aug 16 '10 at 11:24

**2** @polygenelubricants: If you wanted to prove correctness, you would first show that it always provides *a* correct solution (that is, it always produces a pair of numbers which, when removing them from the set, would result in the remainder of the set having the observed sum and sum-of-squares). From there, proving uniqueness is as simple as showing that it only produces one such pair of numbers. – Anon. Aug 16 '10 at 11:50

**3** The nature of the equations means that you will get two values of k2 from that equation. However, from teh first equation that you use to generate k1 you can see that these two values of k2 will mean that k1 is the other value so you have two solutions that are the same numbers the opposite way around. If you abitrarily declared that k1>k2 then you'd only have one solution to the quadratic equation and thus one solution overall. And clearly by the nature of the question an answer always exists so it always works. – Chris Aug 16 '10 at 12:06

**3** For a given sum k1+k2, there are many pairs. We can write these pairs as K1=a+b and K2 = a-b where a = (K1+k2/2). a is unique for a given sum. The sum of the squares (a+b)**2 + (a-b)**2 = 2*(a**2 + b**2). For a given sum K1+K2, the a**2 term is fixed and we see that the sum of the squares will be unique due to the b**2 term. Therefore, the values x and y are unique for a pair of integers. – phkahler Aug 16 '10 at 14:31

show **2** more comments

---

As @j_random_hacker pointed out, this is quite similar to *Finding duplicates in O(n) time and O(1) space*, and an adaptation of my answer there works here too.

Assuming that the "bag" is represented by a 1-based array `A[]` of size `N - k`, we can solve Qk in `O(N)` time and `O(k)` additional space.

First, we extend our array `A[]` by `k` elements, so that it is now of size `N`. This is the `O(k)` additional space. We then run the following pseudo-code algorithm:

```
for i := n - k + 1 to n
    A[i] := A[1]
end for

for i := 1 to n - k
    while A[A[i]] != A[i]
        swap(A[i], A[A[i]])
    end while
end for

for i := 1 to n
    if A[i] != i then
        print i
    end if
end for
```

The first loop initialises the `k` extra entries to the same as the first entry in the array (this is just a convenient value that we know is already present in the array - after this step, any entries that were missing in the initial array of size `N-k` are still missing in the extended array).

The second loop permutes the extended array so that if element `x` is present at least once, then one of those entries will be at position `A[x]`.

Note that although it has a nested loop, it still runs in `O(N)` time - a swap only occurs if there is an `i` such that `A[i] != i`, and each swap sets at least one element such that `A[i] == i`, where that wasn't true before. This means that the total number of swaps (and thus the total number of executions of the `while` loop body) is at most `N-1`.

The third loop prints those indexes of the array `i` that are not occupied by the value `i` - this means that `i` must have been missing.

edited Dec 20 '14 at 4:13

Pavan Manjunath
**11.4k** 2 41 72

answered Apr 22 '11 at 4:32

caf
**122k** 8 137 269

**1** I wonder why so few people vote this answer up and even did not mark it as a correct answer. Here is the code in Python. It runs in O(n) time and need extra space O(k). pastebin.com/9jZqnTzV – wall-e Oct 22 '12 at 4:03

**1** @caf this is quite similar to setting the bits and counting the places where the bit is 0. And I think as you are creating an integer array more memory is occupied. – Fox Apr 22 '13 at 6:41

**1** "Setting the bits and counting the places where the bit is 0" requires O(n) extra space, this solution shows how to use O(k) extra space. – caf Dec 12 '13 at 23:19

**1** Doesn't work with streams as input and modifies the input array (though I like it very much and the idea is fruitful). – comco Jan 30 '14 at 14:07

**1** @v.oddou: Nope, it's fine. The swap will change `A[i]`, which means that the next iteration won't be comparing the same two values as the previous one. The new `A[i]` will be the same as the last loop's `A[A[i]]`, but the new `A[A[i]]` will be a *new* value. Try it and see. – caf Apr 3 '14 at 10:55

show **5** more comments

---

I asked a 4-year-old to solve this problem. He sorted the numbers and then counted along. This has a space requirement of O(kitchen floor), and it works just as easy however many balls are missing.

edited May 29 '14 at 12:53

Peter Mortensen
**7,808** 8 55 90

answered Apr 12 '13 at 18:59

Colonel Panic
**28.8k** 15 123 183

**4** Haha, priceless – Mihai Danila Aug 14 '13 at 21:57

**2** ;) your 4 year old must be approaching 5 or/and is a genius. my 4 year old daughter cannot even count properly to 4 yet. well to be fair let's say she just barely finally integrated the "4"'s existence. otherwise until now she would always skip it. "1,2,3,5,6,7" was her usual counting sequence. I asked her to add pencils together and she would manage 1+2=3 by denumbering all again from scratch. I'm worried actually... :'( meh.. – v.oddou Apr 3 '14 at 8:07

add a comment

---

I haven't checked the maths, but I suspect that computing `Σ(n^2)` in the same pass as we compute `Σ(n)` would provide enough info to get two missing numbers, Do `Σ(n^3)` as well if there are three, and so on.

answered Aug 16 '10 at 10:38

AakashM
**35k** 5 72 127

add a comment

---

Not sure, if it's the most efficient solution, but I would loop over all entries, and use a bitset to remember, which numbers are set, and then test for 0 bits.

I like simple solutions - and I even believe, that it might be faster than calculating the sum, or the sum of squares etc.

answered Aug 16 '10 at 10:38

Chris Lercher
**22.7k** 4 54 92

**4** I did propose this obvious answer, but this is not what the interviewer wanted. I explicitly said in the question that this is not the answer I'm looking for. Another obvious answer: sort first. Neither the `O(N)` counting sort nor `O(N log N)` comparison sort is what I'm looking for, although they are both very simple solutions. – polygenelubricants Aug 16 '10 at 11:14

@polygenelubricants: I can't find where you said that in your question. If you consider the bitset to be the result, then there is no second pass. The complexity is (if we consider N to be constant, as the interviewer suggests by saying, that the complexity is "defined in *k* not N") O(1), and if you need to construct a more "clean" result, you get O(k), which is the best you can get, because you always need O(k) to create the clean result. – Chris Lercher Aug 16 '10 at 11:20

"Note that I'm not looking for the obvious set-based solution (e.g. using a bit set,". The second last paragraph from the original question. – hrnt Aug 16 '10 at 11:24

**3** @hmt: Yes, the question was edited a few minutes ago. I'm just giving the answer, that I would expect from an interviewee... Artificially constructing a sub-optimal solution (you can't beat O(n) + O(k) time, no matter what

you do) doesn't make sense to me - except if you can't afford O(n) additional space, but the question isn't explicit on that. – Chris Lercher Aug 16 '10 at 11:30

**2** I've edited the question again to further clarify. I do appreciate the feedback/answer. – polygenelubricants Aug 16 '10 at 11:42

add a comment

---

Wait a minute. As the question is stated, there are 100 numbers in the bag. No matter how big k is, the problem can be solved in constant time because you can use a set and remove numbers from the set in at most 100 - k iterations of a loop. 100 is constant. The set of remaining numbers is your answer.

If we generalise the solution to the numbers from 1 to N, nothing changes except N is not a constant, so we are in O(N - k) = O(N) time. For instance, if we use a bit set, we set the bits to 1 in O(N) time, iterate through the numbers, setting the bits to 0 as we go (O(N-k) = O(N)) and then we have the answer.

It seems to me that the interviewer was asking you how to *print out* the contents of the final set in O(k) time rather than O(N) time. Clearly, with a bit set, you have to iterate through all N bits to determine whether you should print the number or not. However, if you change the way the set is implemented you can print out the numbers in k iterations. This is done by putting the numbers into an object to be stored in both a hash set and a doubly linked list. When you remove an object from the hash set, you also remove it from the list. The answers will be left in the list which is now of length k.

answered Aug 16 '10 at 11:25

JeremyP
**49.1k** 6 60 97

**3** This answer is too simple, and we all know that simple answers don't work! ;) Seriously though, original question should probably emphasize O(k) space requirement. – DK. Sep 2 '10 at 20:48

The problem is not that is simple but that you'll have to use O(n) additional memory for the map. The problem bust me solved in constant time and constant memory – Mojo Risin Mar 14 '11 at 14:58

**1** I bet you can prove the minimal solution is at least O(N). because less, would mean that you didn't even LOOK at some numbers, and since there is no ordering specified, looking at ALL numbers is mandatory. – v.oddou Apr 3 '14 at 8:12

add a comment

---

The problem with solutions based on sums of numbers is they don't take into account the cost of storing and working with numbers with large exponents... in practice, for it to work for very large n, a big numbers library would be used. We can analyse the space utilisation for these algorithms.

We can analyse the time and space complexity of sdcvvc and Dimitris Andreou's algorithms.

Storage:

```
l_j = ceil (log_2 (sum_{i=1}^n i^j))
l_j > log_2 n^j  (assuming n >= 0, k >= 0)
l_j > j log_2 n \in \Omega(j log n)

l_j < log_2 ((sum_{i=1}^n i)^j) + 1
l_j < j log_2 (n) + j log_2 (n + 1) - j log_2 (2) + 1
l_j < j log_2 n + j + c \in O(j log n)`
```

So `l_j \in \Theta(j log n)`

Total storage used: `\sum_{j=1}^k l_j \in \Theta(k^2 log n)`

Space used: assuming that computing `a^j` takes `ceil(log_2 j)` time, total time:

```
t = k ceil(\sum_i=1^n log_2 (i)) = k ceil(log_2 (\prod_i=1^n (i)))
t > k log_2 (n^n + O(n^(n-1)))
t > k log_2 (n^n) = kn log_2 (n)  \in \Omega(kn log n)
t < k log_2 (\prod_i=1^n i^i) + 1
t < kn log_2 (n) + 1 \in O(kn log n)
```

Total time used: `\Theta(kn log n)`

If this time and space is satisfactory, you can use a simple recursive algorithm. Let b!i be the ith entry in the bag, n the number of numbers before removals, and k the number of removals. In Haskell syntax...

```
let
  -- O(1)
  isInRange low high v = (v >= low) && (v <= high)
  -- O(n - k)
  countInRange low high = sum $ map (fromEnum . isInRange low high . (!)b) [1..(n-k)]
  findMissing l low high krange
    -- O(1) if there is nothing to find.
    | krange=0 = l
    -- O(1) if there is only one possibility.
    | low=high = low:l
    -- Otherwise total of O(knlog(n)) time
    | otherwise =
       let
         mid = (low + high) `div` 2
         klow = countInRange low mid
         khigh = krange - klow
       in
         findMissing (findMissing low mid klow) (mid + 1) high khigh
in
  findMising 1 (n - k) k
```

Storage used: `O(k)` for list, `O(log(n))` for stack: `O(k + log(n))` This algorithm is more intuitive, has the same time complexity, and uses less space.

answered Sep 2 '10 at 11:41

a1kmm
**656** 4 8

**1** +1, looks nice but you lost me going from line 4 to line 5 in snippet #1 -- could you explain that further? Thanks! – j_random_hacker Oct 28 '10 at 8:16

add a comment

---

Can you check if every number exists? If yes you may try this:

S = sum of all numbers in the bag (S < 5050)
Z = sum of the missing numbers 5050 - S

if the missing numbers are `x` and `y` then:

x = Z - y and
max(x) = Z - 1

So you check the range from `1` to `max(x)` and find the number

edited Nov 21 '12 at 17:57

Nakilon
**13.2k** 5 44 63

answered Aug 16 '10 at 10:37

Ilian Iliev
**1,938** 1 11 42

add a comment

---

This might sound stupid, but, in the first problem presented to you, you would have to see all the remaining numbers in the bag to actually add them up to find the missing number using that equation.

So, since you get to see all the numbers, just look for the number that's missing. The same goes for when two numbers are missing. Pretty simple I think. No point in using an equation when you get to see the numbers remaining in the bag.

edited May 29 '14 at 12:46

Peter Mortensen
**7,808** 8 55 90

answered Sep 2 '10 at 3:27

Stephan M
**29** 1

**2** I think the benefit of summing them up is that you don't have to remember which numbers you've already seen (e.g., there's no extra memory requirement). Otherwise the only option is to retain a set of all the values seen and then iterate over that set again to find the one that's missing. – Dan Tao Sep 2 '10 at 23:00

**3** This question is usually asked with the stipulation of O(1) space complexity. – Matthieu N. Sep 14 '10 at 21:38

The sum of the first N numbers is N(N+1)/2. For N=100, Sum=100*(101)/2=5050 ; – tmarthal Apr 29 '11 at 1:39

add a comment

---

May be this algorithm can work for question 1:

1. Precompute xor of first 100 integers(val=1^2^3^4....100)
2. xor the elements as they keep coming from input stream ( val1=val1^next_input)
3. final answer=val^val1

Or even better:

```
for i=1 to 100
 do
   val=val^i
 done
while end of input
   do
     val=val^input()
   done
return val
```

edited May 29 '14 at 12:49

Peter Mortensen
**7,808** 8 55 90

answered Dec 6 '11 at 12:03

bashrc
**741** 1 4 18

add a comment

---

You can solve Q2 if you have the sum of both lists and the product of both lists.

(l1 is the original, l2 is the modified list)

```
d = sum(l1) - sum(l2)
m = mul(l1) / mul(l2)
```

We can optimise this since the sum of an arithmetic series is n times the average of the first and last terms:

```
n = len(l1)
d = (n/2)*(n+1) - sum(l2)
```

Now we know that (if a and b are the removed numbers):

```
a + b = d
a * b = m
```

So we can rearrange to:

```
a = s - b
b * (s - b) = m
```

And multiply out:

```
-b^2 + s*b = m
```

And rearrange so the right side is zero:

```
-b^2 + s*b - m = 0
```

Then we can solve with the quadratic formula:

```
b = (-s + sqrt(s^2 - (4*-1*-m)))/-2
a = s - b
```

Sample Python 3 code:

```
from functools import reduce
import operator
import math
x = list(range(1,21))
sx = (len(x)/2)*(len(x)+1)
x.remove(15)
x.remove(5)
mul = lambda l: reduce(operator.mul,l)
s = sx - sum(x)
m = mul(range(1,21)) / mul(x)
b = (-s + math.sqrt(s**2 - (-4*(-m))))/-2
a = s - b
print(a,b) #15,5
```

I do not know the complexity of the sqrt, reduce and sum functions so I cannot work out the complexity of this solution (if anyone does know please comment below.)

answered Nov 16 '14 at 16:14

Tuomas Laakkonen

**69** 6

add a comment

---

I think this can be done without any complex mathematical equations and theories. Below is a proposal for an in place and O(2n) time complexity solution:

Input form assumptions :

# of numbers in bag = n

# of missing numbers = k

The numbers in the bag are represented by an array of length n

Length of input array for the algo = n

Missing entries in the array (numbers taken out of the bag) are replaced by the value of the first element in the array.

Eg. Initially bag looks like [2,9,3,7,8,6,4,5,1,10]. If 4 is taken out, value of 4 will become 2 (the first element of the array). Therefore after taking 4 out the bag will look like [2,9,3,7,8,6,2,5,1,10]

The key to this solution is to tag the INDEX of a visited number by negating the value at that INDEX as the array is traversed.

```
    IEnumerable<int> GetMissingNumbers(int[] arrayOfNumbers)
    {
        List<int> missingNumbers = new List<int>();
        int arrayLength = arrayOfNumbers.Length;

        //First Pass
        for (int i = 0; i < arrayLength; i++)
        {
            int index = Math.Abs(arrayOfNumbers[i]) - 1;
            if (index > -1)
            {
                arrayOfNumbers[index] = Math.Abs(arrayOfNumbers[index]) * -1;
//Marking the visited indexes
            }
        }

        //Second Pass to get missing numbers
        for (int i = 0; i < arrayLength; i++)
        {
            //If this index is unvisited, means this is a missing number
            if (arrayOfNumbers[i] > 0)
            {
                missingNumbers.Add(i + 1);
            }
        }

        return missingNumbers;
    }
```

answered Dec 12 '12 at 18:57

pickhunter

**110** 9

add a comment

---

Here's a solution that uses k bits of extra storage, without any clever tricks and just straightforward. Execution time O (n), extra space O (k). Just to prove that this can be solved without reading up on the solution first or being a genius:

```
void puzzle (int* data, int n, bool* extra, int k)
{
    // data contains n distinct numbers from 1 to n + k, extra provides
    // space for k extra bits.

    // Rearrange the array so there are (even) even numbers at the start
    // and (odd) odd numbers at the end.
    int even = 0, odd = 0;
    while (even + odd < n)
    {
        if (data [even] % 2 == 0) ++even;
        else if (data [n - 1 - odd] % 2 == 1) ++odd;
        else { int tmp = data [even]; data [even] = data [n - 1 - odd];
               data [n - 1 - odd] = tmp; ++even; ++odd; }
    }

    // Erase the lowest bits of all numbers and set the extra bits to 0.
    for (int i = even; i < n; ++i) data [i] -= 1;
    for (int i = 0; i < k; ++i) extra [i] = false;

    // Set a bit for every number that is present
    for (int i = 0; i < n; ++i)
    {
        int tmp = data [i];
        tmp -= (tmp % 2);
        if (i >= odd) ++tmp;
        if (tmp <= n) data [tmp - 1] += 1; else extra [tmp - n - 1] = true;
    }

    // Print out the missing ones
    for (int i = 1; i <= n; ++i)
        if (data [i - 1] % 2 == 0) printf ("Number %d is missing\n", i);
    for (int i = n + 1; i <= n + k; ++i)
        if (! extra [i - n - 1]) printf ("Number %d is missing\n", i);

    // Restore the lowest bits again.
```

edited Apr 14 '14 at 10:29

answered Apr 7 '14 at 18:53

gnasher729

**10.8k** 1 9 19

Did you want `(data [n - 1 - odd] % 2 == 1) ++odd;` ? – Charles Apr 12 '14 at 13:36

Thanks, fixed it. – gnasher729 Apr 14 '14 at 10:29

Could you explain how this works? I don't understand. – Teepeemm Sep 26 '14 at 14:03

The solution would be very, very, simple if I could use an array of (n + k) booleans for temporary storage, but that is not allowed. So I rearrange the data, putting the even numbers at the beginning, and the odd numbers at the end of the array. Now the lowest bits of those n numbers can be used for temporary storage, because I know how many even and odd numbers there are and can reconstruct the lowest bits! These n bits and the k extra bits are exactly the (n + k) booleans that I needed. – gnasher729 Oct 15 '14 at 16:40

add a comment

---

I don't know whether this is efficient or not but I would like to suggest this solution.

1. Compute xor of the 100 elements
2. Compute xor of the 98 elements (after the 2 elements are removed)
3. Now (result of 1) XOR (result of 2) gives you the xor of the two missing nos i..e a XOR b if a and b are the missing elements
   4.Get the sum of the missing Nos with your usual approach of the sum formula diff and lets say the diff is d.

Now run a loop to get the possible pairs (p,q) both of which lies in [1 , 100] and sum to d.

When a pair is obtained check whether (result of 3) XOR p = q and if yes we are done.

Please correct me if I am wrong and also comment on time complexity if this is correct

answered Aug 4 '14 at 21:49

user2221214
**57** 8

I don't think the sum and xor uniquely define two numbers. Running a loop to get all possible k-tuples that sum to d takes time O(C(n,k-1))=O(n<sup>k-1</sup>), which, for k>2, is bad. – Teepeemm Sep 26 '14 at 13:57

add a comment

---

Very nice problem. I'd go for using a set difference for Qk. A lot of programming languages even have support for it, like in Ruby:

```
missing = (1..100).to_a - bag
```

It's probably not the most efficient solution but it's one I would use in real life if I was faced with such a task in this case (known boundaries, low boundaries). If the set of number would be very large then I would consider a more efficient algorithm, of course, but until then the simple solution would be enough for me.

answered Aug 16 '10 at 11:18

DarkDust
**50.8k** 9 84 129

add a comment

---

You could try using a Bloom Filter. Insert each number in the bag into the bloom, then iterate over the complete 1-k set until reporting each one not found. This may not find the answer in all scenarios, but might be a good enough solution.

edited Sep 2 '10 at 22:22

answered Sep 2 '10 at 16:29

jdizzle
**1,979** 1 12 21

add a comment

---

I'd take a different approach to that question and probe the interviewer for more details about the larger problem he's trying to solve. Depending on the problem and the requirements surrounding it, the obvious set-based solution might be the right thing and the generate-a-list-and-pick-through-it-afterward approach might not.

For example, it might be that the interviewer is going to dispatch `n` messages and needs to know the `k` that didn't result in a reply and needs to know it in as little wall clock time as possible after the `n-k` th reply arrives. Let's also say that the message channel's nature is such that even running at full bore, there's enough time to do some processing between messages without having any impact on how long it takes to produce the end result after the last reply arrives. That time can be put to use inserting some identifying facet of each sent message into a set and deleting it as each corresponding reply arrives. Once the last reply has arrived, the only thing to be done is to remove its identifier from the set, which in typical implementations takes `O(log k+1)` . After that, the set contains the list of `k` missing elements and there's no additional processing to be done.

This certainly isn't the fastest approach for batch processing pre-generated bags of numbers because the whole thing runs `O((log 1 + log 2 + ... + log n) + (log n + log n-1 + ... + log k))` . But it does work for any value of `k` (even if it's not known ahead of time) and in the example above it was applied in a way that minimizes the most critical interval.

answered Sep 3 '10 at 2:57

Blrfl
**4,424** 12 19

add a comment

---

I believe I have a `O(k)` time and `O(log(k))` space algorithm, given that you have the `floor(x)` and `log2(x)` functions for arbitrarily big integers available:

You have an `k` -bit long integer (hence the `log8(k)` space) where you add the `x^2` , where x is the next number you find in the bag: `s=1^2+2^2+...` This takes `O(N)` time (which is not a problem for the interviewer). At the end you get `j=floor(log2(s))` which is the biggest number you're looking for. Then `s=s-j` and you do again the above:

```
for (i = 0 ; i < k ; i++)
{
  j = floor(log2(s));
  missing[i] = j;
  s -= j;
}
```

Now, you usually don't have floor and log2 functions for `2756` -bit integers but instead for doubles. So? Simply, for each 2 bytes (or 1, or 3, or 4) you can use these functions to get the desired numbers, but this adds an `O(N)` factor to time complexity

edited Nov 21 '12 at 17:58

Nakilon
**13.2k** 5 44 63
add a comment

answered Sep 7 '10 at 14:43

CostasGR43
**1**

---

Try to find the product of numbers from 1 to 50:

Let product, P1 = 1 x 2 x 3 x ..... 50

When you take out numbers one by one, multiply them so that you get the product P2. But two numbers are missing here, hence P2 < P1.

The product of the two mising terms, a x b = P1 - P2.

You already know the sum, a + b = S1.

From the above two equations, solve for a and b through a quadratic equation. a and b are your missing numbers.

edited May 29 '14 at 12:45

Peter Mortensen
**7,808** 8 55 90
add a comment

answered Aug 25 '10 at 13:56

Manjunath
**9** 1

---

I think this can be generalized like this:

Denote S, M as the initial values for the sum of arithmetic series and multiplication.

```
S = 1 + 2 + 3 + 4 + ... n=(n+1)*n/2
M = 1 * 2 * 3 * 4 * .... * n
```

I should think about a formula to calculate this, but that is not the point. Anyway, if one number is missing, you already provided the solution. However, if two numbers are missing then, let's denote the new sum and total multiple by S1 and M1, which will be as follows:

```
S1 = S - (a + b).....................(1)

Where a and b are the missing numbers.

M1 = M - (a * b).....................(2)
```

Since you know S1, M1, M and S, the above equation is solvable to find a and b, the missing numbers.

Now for the three numbers missing:

```
S2 = S - ( a + b + c).....................(1)

Where a and b are the missing numbers.

M2 = M - (a * b * c).....................(2)
```

Now your unknown is 3 while you just have two equations you can solve from.

edited May 29 '14 at 12:59

Peter Mortensen
**7,808** 8 55 90
add a comment

answered Aug 16 '13 at 14:26

Jack_of_All_Trades
**2,311** 6 25 46

---

We can use the following simple code to find repeating and missing values:

```
int size = 8;
int arr[] = {1, 2, 3, 5, 1, 3};
int result[] = new int[size];

for(int i =0; i < arr.length; i++)
{
    if(result[arr[i]-1] == 1)
    {
        System.out.println("repeating: " + (arr[i]));
    }
    result[arr[i]-1]++;
}

for(int i =0; i < result.length; i++)
{
    if(result[i] == 0)
    {
        System.out.println("missing: " + (i+1));
    }
}
```

answered Apr 7 '13 at 4:11

user2253712
1

add a comment

---

A possible solution:

```
public class MissingNumber {
    public static void main(String[] args) {
        // 0-20
        int [] a = {1,4,3,6,7,9,8,11,10,12,15,18,14};
        printMissingNumbers(a,20);
    }

    public static void printMissingNumbers(int [] a, int upperLimit){
        int b [] = new int[upperLimit];
        for(int i = 0; i < a.length; i++){
            b[a[i]] = 1;
        }
        for(int k = 0; k < upperLimit; k++){
            if(b[k] == 0)
                System.out.println(k);
        }
    }
}
```

edited Jul 15 '13 at 14:34

Rubens
6,661 6 20 48

answered Jul 15 '13 at 14:16

sharma31vipul
1

It should be `System.out.println(k + 1)` since `0` is not counted to as missing number. Also, this doesn't work in an unsorted list of numbers. – mr5 Dec 12 '13 at 2:29

add a comment

---

```
// Size of numbers
def n=100;

// A list of numbers that is missing k numbers.
def list;

// A map
def map = [:];

// Populate the map so that it contains all numbers.
for(int index=0; index<n; index++)
{
  map[index+1] = index+1;
}

// Get size of list that is missing k numbers.
def size = list.size();

// Remove all numbers, that exists in list, from the map.
for(int index=0; index<size; index++)
{
  map.remove(list.get(index));
}

// Content of map is missing numbers
println("Missing numbers: " + map);
```

answered Jul 19 '13 at 22:53

Bae Cheol Shin

**1**

add a comment

---

If a number only appears exactly one time, it is pretty easy to tell in the following way:

Create a boolean array, `boolArray` , of size of the given number; here it is 100.

Loop through the input numbers and set an element to true according the number value. For example, if 45 found, then set `boolArray[45-1] = true` ;

That will be an O(N) operation.

Then loop through `boolArray` . If an element is staying false, then the index of element + 1 is the missing number. For example, if `boolArray[44]` is false, we know number 45 is missing.

That is an O(n) operation. Space complexity is O(1).

So this solution can work to find any missing number from a given continuous number set.

edited May 29 '14 at 12:51

Peter Mortensen

**7,808** 8 55 90

answered Aug 17 '12 at 18:25

Boveyking

**45** 2

No, space complexity of this approach is O(n). Futhermore, this way is already mentioned in the question. – sdcvvc Nov 23 '12 at 19:13

add a comment

---

This is a very easy question

```
void findMissing(){
    bool record[N] = {0};
    for(int i = 0; i < N; i++){
        record[bag[i]-1] = 1;
    }
    for(int i = 0; i < N; i++){
        if(!record[i]) cout << i+1 << endl;
    }
}
```

O(n) time and space complexity

answered May 30 '14 at 17:39

user3692106

**3** 3

1 We're specifically not looking for the O(n) space solution of writing everything down. – Teepeemm Sep 26 '14 at 13:57

Further Optimizations 1) Use bits instead an array of bools 2) Use a link list full of numbers 1-N and remove the ones you find Also, your smart equations still equate to my solution when you boil it down. – user3692106 Dec 16 '14 at 16:36

add a comment

---

For different values of k, the approach will be different so there won't be a generic answer in terms of k. For example, for k=1 one can take advantage of the sum of natural numbers, but for k = n/2, one has to use some kind of bitset. The same way for k=n-1, one can simply compare the only number in the bag with the rest.

edited May 29 '14 at 12:40
Peter Mortensen
7,808 8 55 90

answered Aug 16 '10 at 10:46
bhups
6,378 3 27 44

add a comment

---

Let us assume it is an array from 1 to N and its elements are a1, a2, ...., aN:

```
1+N=N+1;
2+N-1=N+1;
```

..... So the sum here is unique. We can scan the array from the start and from the end to add both the elements. If the sum is N+1; then okay, otherwise they are missing.

```
for (I <= N/2) {
    temp = a[I] + a[n-I];
    if (temp != N+1) then
        Find the missing number or numbers
}
```

Iterate this loop, and you get the answer easily.

edited May 29 '14 at 12:55
Peter Mortensen
7,808 8 55 90

answered Aug 4 '13 at 19:43
corneliusfelix
337 1 13

add a comment

---

Have I missed something? Can't you just put the missing numbers in a list?

```
List missingList = {};
For i=1 to 100 do
    if(i is missing) missing.add(i);
EndFor
```

missingList will have space complexity O(k), and if List is a Vector, then adding a number is O(1) in time.

answered Aug 16 '10 at 11:23
Per Alexandersson
1,037 7 19

2 What's the implementation of your "is missing" function? – JeremyP Aug 16 '10 at 11:27

5 your algorithm complexity is O(n^2), I is missing is O(N) and for loop is O(N) – ArsenMkrt Aug 16 '10 at 11:36

@ArsenMkrt: as many proposed, you could use a set. Transforming whatever-a-bag-is to a set is O(N), and the second pass (the loop) is O(N), yielding O(N). – back2dos Aug 16 '10 at 12:10

add a comment

---

## protected by Juhana Nov 1 '13 at 8:01

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?

**Not the answer you're looking for? Browse other questions tagged algorithm math or ask your own question.**