Find an integer not among four billion given ones

It is an interview question:

> Given an input file with four billion integers, provide an algorithm to generate an integer which is not contained in the file. Assume you have 1 GiB memory. Follow up with what you would do if you have only 10 MiB of memory.

My analysis:

The size of the file is 4 * 10^9 * 4 bytes = 16 GiB.

We can do external sorting, thus we get to know the range of the integers. My question is what is the best way to detect the missing integer in the sorted big integer sets?

My understanding(after reading all answers):

Assuming we are talking about 32-bit integers. There are 2^32 = 4*10^9 distinct integers.

Case 1: we have 1 GiB = 1 * 10^9 bytes * 8 bits/byte = 8 billion bits memory. Solution: if we use one bit representing one distinct integer, it is enough. we don't need sort. Implementation:

```
int radix = 8;
byte[] bitfield = new byte[0xffffffff/radix];
void F() throws FileNotFoundException{
    Scanner in = new Scanner(new FileReader("a.txt"));
    while(in.hasNextInt()){
        int n = in.nextInt();
        bitfield[n/radix] |= (1 << (n%radix));
    }

    for(int i = 0; i< bitfield.lenght; i++){
        for(int j =0; j<radix; j++){
            if( (bitfield[i] & (1<<j)) == 0) System.out.print(i*radix+j);
        }
    }
}
```

Case 2: 10 MB memory = 10 * 10^6 * 8 bits = 80 million bits

```
Solution: For all possible 16-bit prefixes, there are 2^16 number of
integers = 65536, we need 2^16 * 4 * 8 = 2 million bits. We need build
65536 buckets. For each bucket, we need 4 bytes holding all possibilities because
 the worst case is all the 4 billion integers belong to the same bucket.

Step 1: Build the counter of each bucket through the first pass through the file.
Step 2: Scan the buckets, find the first one who has less than 65536 hit.
Step 3: Build new buckets whose high 16-bit prefixes are we found in step2
through second pass of the file
Step 4: Scan the buckets built in step3, find the first bucket which doesnt
have a hit.

The code is very similar to above one.
```

Conclusion: We decrease memory through increasing file pass.

*A clarification for those arriving late: The question, as asked, does not say that there is exactly one integer that is not contained in the file -- at least that's not how most people interpret it. Many comments in the comment thread **are** about that variation of the task, though. Unfortunately the comment that **introduced** it to the comment thread was later deleted by its author, so now it looks like the orphaned replies to it just misunderstood everything. It's very confusing. Sorry.*

algorithm

edited Jun 26 '14 at 5:20

community wiki
19 revs, 11 users 61%
SecureFish

29 @trashgod, wrong. For 4294967295 unique integers you'll have 1 integer remaining. To find it, you should summ all integers and substract it from precalculated summ of all possible integers. – Nakilon Aug 23 '11 at 1:22

52 This is the second "pearl" from "Programming Pearls", and I would suggest you read the whole discussion in the book. See books.google.com/… – Alok Singhal Aug 23 '11 at 2:29

6 @Richard a 64 bit int would be more than large enough. – cftarnas Aug 23 '11 at 17:24

54 `int getMissingNumber(File inputFile) { return 4; }` (reference) – johnny Aug 23 '11 at 17:28

11 It doesn't matter that you can't store the sum of all integers from 1 to 2^32 because the integer type in languages like C/C++ ALWAYS preserves properties like associativity and communicativity. What this means is that although the sum won't be the right answer, if you calculate the expected with overflow, the actual sum with overflow, and then subtract, the result will still be correct (provided it itself does not overflow). – thedayturns Aug 23 '11 at 22:48

show **29** more comments

37 Answers

1 2 next

Assuming that "integer" means 32 bits: Having 10 MB of space is more than enough for you to count how many numbers there are in the input file with any given 16-bit prefix, for all possible 16-bit prefixes in one pass through the input file. At least one of the buckets will have be hit less than 2^16 times. Do a second pass to find of which of the possible numbers in that bucket are used already.

If it means more than 32 bits, but still of bounded size: Do as above, ignoring all input numbers that happen to fall outside the (signed or unsigned; your choice) 32-bit range.

If "integer" means mathematical integer: Read through the input once and keep track of the ~~largest number~~ length of the longest number you've ever seen. When you're done, output ~~the maximum plus one~~ a random number that has one more digit. (One of the numbers in the file may be a bignum that takes more than 10 MB to represent exactly, but if the input is a file, then you can at least represent the *length* of anything that fits in it).

edited Aug 22 '11 at 22:02

answered Aug 22 '11 at 21:28
Henning Makholm
13.8k 1 21 50

21 Perfect. Your first answer requires only 2 passes through the file! – corsiKa Aug 22 '11 at 21:54

35 A 10 MB bignum? That's pretty extreme. – Mark Ransom Aug 22 '11 at 22:44

8 @Legate, just skip overlarge numbers and do nothing about them. Since you're not going to output an overlarge number anyway, there's no need to keep track of which of them you've seen. – Henning Makholm Aug 23 '11 at 11:36

10 The good thing about Solution 1, is that you can decrease the memory by increasing passes. – Yousf Aug 23 '11 at 14:57

9 @Barry: The question above does not indicate that there is exactly one number missing. It doesn't say the numbers in the file don't repeat, either. (Following the question actually asked is probably a good idea in an interview, right? ;-)) – Christopher Creutzig Aug 24 '11 at 6:24

show **25** more comments

Statistically informed algorithms solve this problem using fewer passes than deterministic approaches.

If very large integers are allowed then one can generate a number that is likely to be unique in O(1) time. A pseudo-random 128-bit integer like a GUID will only collide with one of the existing four billion integers in the set in less than one out of every 64 billion billion billion cases.

If integers are limited to 32 bits then one can generate a number that is likely to be unique in a single pass using much less than 10 MB. The odds that a pseudo-random 32-bit integer will collide with one of the 4 billion existing integers is about 93% (4e9 / 2^32). The odds that 1000 pseudo-random integers will all collide is less than one in 12,000 billion billion billion (odds-of-one-collision ^ 1000). So if a program maintains a data structure containing 1000 pseudo-random candidates and iterates through the known integers, eliminating matches from the candidates, it is all but certain to find at least one integer that is not in the file.

edited Dec 28 '14 at 9:54
psmears
8,261 1 21 26

answered Aug 23 '11 at 5:04
Ben Haley
1,413 1 5 14

13 A wonderfully practical answer. – Thomas Padron-McCarthy Aug 23 '11 at 5:34

20 I'm pretty sure the integers are bounded. If they weren't, then even a beginner programmer would think of the algorithm "take one pass through the data to find the maximum number, and add 1 to it" – Adrian Petrescu Aug 23 '11 at 6:50

7 Literally guessing a random output probably won't get you many points on an interview – Brian Gordon Aug 23 '11 at 15:44

3 @Adrian, your solution seems obvious (and it was to me, I used it in my own answer) but it's not obvious to everybody. It's a good test to see if you can spot obvious solutions or if you're going to over-complicate everything you touch. – Mark Ransom Aug 23 '11 at 17:00

11 @Brian: I think this solution is both imaginitive and practical. I for one would give much kudos for this answer. – Richard H Aug 24 '11 at 10:55

show **7** more comments

A detailed discussion on this problem has been discussed in Jon Bentley "Column 1. Cracking the Oyster" *Programming Pearls* Addison-Wesley pp.3-10

Bentley discusses several approaches, including external sort, Merge Sort using several external files etc., But the best method Bentley suggests is a single pass algorithm using bit fields, which he humorously calls "Wonder Sort" :) Coming to the problem, 4 billion numbers can be represented in :

```
4 billion bits = (4000000000 / 8) bytes = about 0.466 GB
```

The code to implement the bitset is simple: (taken from solutions page)

```
#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
#define N 10000000
int a[1 + N/BITSPERWORD];

void set(int i) {        a[i>>SHIFT] |=  (1<<(i & MASK)); }
void clr(int i) {        a[i>>SHIFT] &= ~(1<<(i & MASK)); }
int  test(int i){ return a[i>>SHIFT] &   (1<<(i & MASK)); }
```

Bentley's algorithm makes a single pass over the file, `set` ting the appropriate bit in the array and then examines this array using `test` macro above to find the missing number.

If the available memory is less than 0.466 GB, Bentley suggests a k-pass algorithm, which divides the input into ranges depending on available memory. To take a very simple example, if only 1 byte (i.e memory to handle 8 numbers) was available and the range was from 0 to 31, we divide this into ranges of 0 to 7, 8-15, 16-22 and so on and handle this range in each of `32/8 = 4` passes.

HTH.

edited Aug 24 '11 at 10:38

answered Aug 23 '11 at 4:20

nominolo
4,035 13 26

vine'th
2,339 2 11 16

8 I dont know the book, but no reason to call it "Wonder Sort", as it is just a bucketsort, with a 1-bit counter. – flolo Aug 23 '11 at 6:03

2 Although more portable, this code will be *annihilated* by code written to utilize hardware-supported vector instructions. I think gcc can automatically convert code to using vector operations in some cases though. – Brian Gordon Aug 23 '11 at 15:57

2 @brian I don't think Jon Bentley was allowing such things into his book on algorithms. – David Heffernan Aug 23 '11 at 18:20

5 @BrianGordon, the time spent in ram will be negligible compared to the time spent reading the file. Forget about optimizing it. – Ian Jun 24 '12 at 6:45

show **9** more comments

Since the problem does not specify that we have to find the smallest possible number that is not in the file we could just generate a number that is longer than the input file itself. :)

edited Apr 16 '12 at 10:42
Saeed Amiri
16.5k 4 19 60

answered Aug 23 '11 at 11:07
Andris
1,161 5 12

6 Well done, very creative thinking – TrojanName Aug 25 '11 at 9:10

1 Unless the largest number in the file is max int then you will just overflow – KBusc Mar 11 '14 at 15:35

show **1** more comment

For the 1 GB RAM variant you can use a bit vector. You need to allocate 4 billion bits == 500 MB byte array. For each number you read from the input, set the corresponding bit to '1'. Once you done, iterate over the bits, find the first one that is still '0'. Its index is the answer.

edited Aug 27 '11 at 7:29
Peter Mortensen
7,808 8 55 90

answered Aug 22 '11 at 21:17
Itay Maman
16.3k 4 40 80

4 The range of numbers in the input isn't specified. How does this algorithm work if the input consists of all the even numbers between 8 billion and 16 billion? – Mark Ransom Aug 22 '11 at 21:35

21 @Mark, just ignore inputs that are outside the 0..2^32 range. You're not going to output any of them anyway, so there's no need to remember which of them to avoid. – Henning Makholm Aug 22 '11 at 21:40

4 Instead of iterating yourself you can use `bitSet.nextClearBit(0)` : download.oracle.com/javase/6/docs/api/java/util/… – starblue Aug 23 '11 at 10:32

3 It would be useful to mention that, regardless of the range of the integers, at least one bit is guaranteed to be 0 at the end of the pass. This is due to the pigeonhole principle. – Rafał Dowgird Aug 23 '11 at 12:53

show **3** more comments

If they are 32-bit integers (likely from the choice of ~4 billion numbers close to 2^32), your list of 4 billion numbers will take up at most 93% of the possible integers (4 * 10^9 / (2^32)). So if you create a bit-array of 2^32 bits with each bit initialized to zero (which will take up 2^29 bytes ~ 500 MB of RAM; remember a byte = 2^3 bits = 8 bits), read through your integer list and for each int set the corresponding bit-array element from 0 to 1; and then read through your bit-array and return the first bit that's still 0.

In the case where you have less RAM (~10 MB), this solution needs to be slightly modified. 10 MB ~ 83886080 bits is still enough to do a bit-array for all numbers between 0 and 83886079. So you could read through your list of ints; and only record #s that are between 0 and 83886079 in your bit array. If the numbers are randomly distributed; with overwhelming probability (it differs by 100% by about 10^-2592069) you will find a missing int). In fact, if you only choose numbers 1 to 2048 (with only 256 bytes of RAM) you'd still find a missing number an overwhelming percentage (99.9999999999999999999999999999999999999999999999999999999999999999995%) of the time.

But let's say instead of having about 4 billion numbers; you had something like 2^32 - 1 numbers and less than 10 MB of RAM; so any small range of ints only has a small possibility of not containing the number.

If you were guaranteed that each int in the list was unique, you could sum the numbers and subtract the sum with one # missing to the full sum (1/2)*(2^32)*(2^32 - 1) = 9223372034707292160 to find the missing int. However, if an int occurred twice this method will fail.

However, you can always divide and conquer. A naive method, would be to read through the array and count the number of numbers that are in the first half (0 to 2^31-1) and second half (2^31, 2^32). Then pick the range with fewer numbers and repeat dividing that range in half. (Say if there were two less number in (2^31, 2^32) then your next search would count the numbers in the range (2^31, 3*2^30-1), (3*2^30, 2^32). Keep repeating until you find a range with zero numbers and you have your answer. Should take O(lg N) ~ 32 reads through the array.

That method was inefficient. We are only using two integers in each step (or about 8 bytes of RAM with a 4 byte (32-bit) integer). A better method would be to divide into sqrt(2^32) = 2^16 = 65536 bins, each with 65536 numbers in a bin. Each bin requires 4 bytes to store its count, so you need 2^18 bytes = 256 kB. So bin 0 is (0 to 65535=2^16-1), bin 1 is (2^16=65536 to 2*2^16-1=131071), bin 2 is (2*2^16=131072 to 3*2^16-1=196607). In python you'd have something like:

```
import numpy as np
nums_in_bin = np.zeros(65536, dtype=np.uint32)
for N in four_billion_int_array:
    nums_in_bin[N // 65536] += 1
for bin_num, bin_count in enumerate(nums_in_bin):
    if bin_count < 65536:
        break # we have found an incomplete bin with missing ints (bin_num)
```

Read through the ~4 billion integer list; and count how many ints fall in each of the 2^16 bins and find an incomplete_bin that doesn't have all 65536 numbers. Then you read through the 4 billion integer list again; but this time only notice when integers are in that range; flipping a bit when you find them.

```
del nums_in_bin # allow gc to free old 256kB array
from bitarray import bitarray
my_bit_array = bitarray(65536) # 32 kB
my_bit_array.setall(0)
for N in four_billion_int_array:
    if N // 65536 == bin_num:
        my_bit_array[N % 65536] = 1
for i, bit in enumerate(my_bit_array):
    if not bit:
        print bin_num*65536 + i
        break
```

edited Jul 10 '13 at 16:59

answered Aug 23 '11 at 5:58
dr jimbob
7,372 23 53

3 Such an awesome answer. This would actually work; and has guaranteed results. – Jonathan Dickinson Aug 23 '11 at 13:14

show **2** more comments

Why make it so complicated? You ask for an integer not present in the file?

According to the rules specified, the only thing you need to store is the largest integer that you encountered so far in the file. Once the entire file has been read, return a number 1 greater than that.

There is no risk of hitting maxint or anything, because according to the rules, there is no restriction to the size of the integer or the number returned by the algorithm.

answered Aug 23 '11 at 14:38
Pete
5,589 4 26 46

4 This would work unless the max int was in the file, which is entirely possible... – PearsonArtPhoto Aug 23 '11 at 15:04

10 The rules does not specify that it is 32bit or 64bit or anything, so according to the rules specified, there is no max int. Integer is not a computer term, it is a math term identifying positive or negative whole numbers. – Pete Aug 23 '11 at 19:30

17 The entire notion of "max int" is not valid in the context if no programming language has been specified. e.g. look at Python's definition of a long integer. It is limitless. There is no roof. You can always add one. You are assuming it is being implemented in a language that has a maximum allowed value for an integer. – Pete Aug 23 '11 at 19:47

show **1** more comment

This can be solved in very little space using a variant of binary search.

1. Start off with the allowed range of numbers, `0` to `4294967295` .

2. Calculate the midpoint.

3. Loop through the file, counting how many numbers were equal, less than or higher than the midpoint value.
4. If no numbers were equal, you're done. The midpoint number is the answer.
5. Otherwise, choose the range that had the fewest numbers and repeat from step 2 with this new range.

This will require up to 32 linear scans through the file, but it will only use a few bytes of memory for storing the range and the counts.

This is essentially the same as Henning's solution, except it uses two bins instead of 16k.

edited Aug 23 '11 at 20:28 answered Aug 23 '11 at 20:17
hammar
94.7k 8 181 294

1 It's what I started with, before I began optimizing for the given parameters. – Henning Makholm Aug 24 '11 at 15:06

show **2** more comments

EDIT Ok, this wasn't quite thought through as it assumes the integers in the file follow some static distribution. Apparently they don't need to, but even then one should try this:

There are ≈4.3 billion 32-bit integers. We don't know how they are distributed in the file, but the worst case is the one with the highest Shannon entropy: an equal distribution. In this case, the probablity for any one integer to not occur in the file is

$((2^{32}-1)/2^{32})^{4\ 000\ 000\ 000} \approx .4$

The lower the Shannon entropy, the higher this probability gets on the average, but even for this worst case we have a chance of 90% to find a nonoccurring number after 5 guesses with random integers. Just create such numbers with a pseudorandom generator, store them in a list. Then read int after int and compare it to all of your guesses. When there's a match, remove this list entry. After having been through all of the file, chances are you will have more than one guess left. Use any of them. In the rare (10% even at worst case) event of no guess remaining, get a new set of random integers, perhaps more this time (10->99%).

Memory consumption: a few dozen bytes, complexity: O(n), overhead: neclectable as most of the time will be spent in the unavoidable hard disk accesses rather than comparing ints anyway.

The actual worst case, when we do *not* assume a static distribution, is that every integer occurs max. once, because then only $1 - 4000000000/2^{32} \approx 6\%$ of all integers don't occur in the file. So you'll need some more guesses, but that still won't cost hurtful amounts of memory.

edited Aug 23 '11 at 20:12 answered Aug 23 '11 at 1:49
leftaroundabout
27.9k 3 39 92

4 I'm glad to see someone else thought of this, but why is it way down here at the bottom? This is a 1-pass algo… 10 MB is enough for 2.5 M guesses, and 93%^2.5M ≈ 10^-79000 is truly a negligible chance of needing a second scan. Due to the overhead of binary searching, it goes faster if you use fewer guesses! This is optimal in both time and space. – Potatoswatter Aug 24 '11 at 6:46

1 @Potatoswatter: good you mentioned binary search. That's probably not worth the overhead when using only 5 guesses, but it certainly is at 10 or more. You could even do the 2 M guesses, but then you should store them in a hash set to get O(1) for the searching. – leftaroundabout Aug 24 '11 at 10:31

1 @Potatoswatter Ben Haley's equivalent answer is up near the top – Brian Gordon Aug 24 '11 at 14:52

show **3** more comments

If you have one integer missing from the range [0, 2^***x*** - 1] then just xor them all together. For example:

```
>>> 0 ^ 1 ^ 3
2
>>> 0 ^ 1 ^ 2 ^ 3 ^ 4 ^ 6 ^ 7
5
```

(I know this doesn't answer the question *exactly*, but it's a good answer to a very similar question.)

answered Aug 24 '11 at 2:43
rfrankel
432 3 19

1 Yes, it's easy to prove[] *that works when one integer is missing, but it frequently fails if more than one is missing. For example,* `0 ^ 1 ^ 3 ^ 4 ^ 6 ^ 7` *is 0. [* Writing 2**x for 2 to x'th power, and a^b for a xor b, the xor of all k<2**x is zero -- k ^ ~k = (2^x)-1 for k < 2^(x-1), and k ^ ~k ^ j ^ ~j = 0 when j=k+2**(x-2) -- so the xor of all but one number is the value of the missing one] – jwpat7 Aug 24 '11 at 15:19

1 As I mention in a comment on ircmaxell's reply: The problem does not say "one number is missing", it says to find a number not included in the 4 billion numbers in the file. If we assume 32-bit integers, then about 300 million numbers may be missing from the file. The likelihood of the xor of the numbers present matching a missing number is only about 7%. – jwpat7 Aug 24 '11 at 15:44

show **1** more comment

They may be looking to see if you have heard of a probabilistic Bloom Filter which can very efficiently determine absolutely if a value is not part of a large set, (but can only determine with high probability it is a member of the set.)

answered Aug 23 '11 at 21:20
Paul
14.1k 3 36 76

4 With probably over 90% of the possible values set, your Bloom Filter would probably need to degenerate to the bitfield so many answers already use. Otherwise, you'll just end up with a useless completely filled bitstring. – Christopher Creutzig Aug 24 '11 at 6:59

show **4** more comments

Based on the current wording in the original question, the simplest solution is:

Find the maximum value in the file, then add 1 to it.

answered Aug 23 '11 at 3:04
oosterwal
1,122 5 15

5 What if the MAXINT is included in the file? – Petr Peller Aug 23 '11 at 9:55

1 @oosterwal, if this answer was allowed, than you don't even need to read the file – just print as big number as you can. – Nakilon Aug 24 '11 at 0:50

1 @oosterwal, if your random huge number was the largest you could print, and it was in file, then this task couldn't be solved. – Nakilon Aug 24 '11 at 1:21

3 @Nakilon: +1 Your point is taken. It's roughly equivalent to figuring the total number of digits in the file and printing a number with that many digits. – oosterwal Aug 24 '11 at 1:53

show **3** more comments

Use a `BitSet`. 4 billion integers (assuming up to 2^32 integers) packed into a BitSet at 8 per byte is 2^32 / 2^3 = 2^29 = approx 0.5 Gb.

To add a bit more detail - every time you read a number, set the corresponding bit in the BitSet. Then, do a pass over the BitSet to find the first number that's not present. In fact, you could do this just as effectively by repeatedly picking a random number and testing if it's present.

Actually BitSet.nextClearBit(0) will tell you the first non-set bit.

Looking at the BitSet API, it appears to only support 0..MAX_INT, so you may need 2 BitSets - one for +'ve numbers and one for -'ve numbers - but the memory requirements don't change.

edited Aug 22 '11 at 21:22

answered Aug 22 '11 at 21:17
dty
11.6k 1 23 59

show **1** more comment

If there is no size limit, the quickest way is to take the length of the file, and generate the length of the file+1 number of random digits (or just "11111..." s). Advantage: you don't even need to read the file, and you can minimize memory use nearly to zero. Disadvantage: You will print billions of digits.

However, if the only factor was minimizing memory usage, and nothing else is important, this would be the optimal solution. It might even get you a "worst misuse of the rules" award.

edited Aug 24 '11 at 13:33

answered Aug 24 '11 at 6:09
vsz
1,836 1 14 35

add a comment

- The simplest approach is to find the minimum number in the file, and return 1 less than that. This uses O(1) storage, and O(n) time for a file of n numbers. However, it will fail if number range is limited, which could make min-1 not-a-number.
- The simple and straightforward method of using a bitmap has already been mentioned. That method uses O(n) time and storage.
- A 2-pass method with 2^16 counting-buckets has also been mentioned. It reads 2*n integers, so uses O(n) time and O(1) storage, but it cannot handle datasets with more than 2^16 numbers. However, it's easily extended to (eg) 2^60 64-bit integers by running 4 passes instead of 2, and easily adapted to using tiny memory by using only as many bins as fit in memory and increasing the number of passes correspondingly, in which case run time is no longer O(n) but instead is O(n*log n).
- The method of XOR'ing all the numbers together, mentioned so far by rfrankel and at length by ircmaxell answers the question asked in stackoverflow#35185, as ltn100 pointed out. It uses O(1) storage and O(n) run time. If for the moment we assume 32-bit integers, XOR has a 7% probability of producing a distinct number. Rationale: given ~ 4G distinct numbers XOR'd together, and ca. 300M not in file, the number of set bits in each bit position has equal chance of being odd or even. Thus, 2^32 numbers have equal likelihood of arising as the XOR result, of which 93% are already in file. Note that if the numbers in file aren't all distinct, the XOR method's probability of success rises.

edited Aug 24 '11 at 18:43 answered Aug 22 '11 at 21:35

jwpat7
5,632 1 9 25

add a comment

If we assume that the range of numbers will always be 2^n (an even power of 2), then exclusive-or will work (as shown by another poster). As far as why, let's prove it:

The Theory

Given any 0 based range of integers that has `2^n` elements with one element missing, you can find that missing element by simply xor-ing the known values together to yield the missing number.

The Proof

Let's look at n = 2. For n=2, we can represent 4 unique integers: 0, 1, 2, 3. They have a bit pattern of:

- 0 - 00
- 1 - 01
- 2 - 10
- 3 - 11

Now, if we look, each and every bit is set exactly twice. Therefore, since it is set an even number of times, and exclusive-or of the numbers will yield 0. If a single number is missing, the exclusive-or will yield a number that when exclusive-ored with the missing number will result in 0. Therefore, the missing number, and the resulting exclusive-ored number are exactly the same. If we remove 2, the resulting xor will be `10` (or 2).

Now, let's look at n+1. Let's call the number of times each bit is set in `n`, `x` and the number of times each bit is set in `n+1` `y`. The value of `y` will be equal to `y = x * 2` because there are `x` elements with the `n+1` bit set to 0, and `x` elements with the `n+1` bit set to 1. And since `2x` will always be even, `n+1` will always have each bit set an even number of times.

Therefore, since `n=2` works, and `n+1` works, the xor method will work for all values of `n>=2`.

The Algorithm For 0 Based Ranges

This is quite simple. It uses 2*n bits of memory, so for any range <= 32, 2 32 bit integers will work (ignoring any memory consumed by the file descriptor). And it makes a single pass of the file.

```
long supplied = 0;
long result = 0;
while (supplied = read_int_from_file()) {
    result = result ^ supplied;
}
return result;
```

The Algorithm For Arbitrary Based Ranges

This algorithm will work for ranges of any starting number to any ending number, as long as the total range is equal to 2^n... This basically re-bases the range to have the minimum at 0. But it does require 2 passes through the file (the first to grab the minimum, the second to compute the missing int).

```
long supplied = 0;
long result = 0;
long offset = INT_MAX;
while (supplied = read_int_from_file()) {
    if (supplied < offset) {
        offset = supplied;
    }
}
reset_file_pointer();
while (supplied = read_int_from_file()) {
    result = result ^ (supplied - offset);
}
return result + offset;
```

Arbitrary Ranges

We can apply this modified method to a set of arbitrary ranges, since all ranges will cross a power of 2^n at least once. This works only if there is a single missing bit. It takes 2 passes of an unsorted file, but it will find the single missing number every time:

```
long supplied = 0;
long result = 0;
long offset = INT_MAX;
long n = 0;
double temp;
while (supplied = read_int_from_file()) {
    if (supplied < offset) {
        offset = supplied;
    }
}
reset_file_pointer();
while (supplied = read_int_from_file()) {
    n++;
    result = result ^ (supplied - offset);
}
// We need to increment n one value so that we take care of the missing
// int value
n++
while (n == 1 || 0 != (n & (n - 1))) {
    result = result ^ (n++);
}
return result + offset;
```

Basically, re-bases the range around 0. Then, it counts the number of unsorted values to append as it computes the exclusive-or. Then, it adds 1 to the count of unsorted values to take care of the missing value (count the missing one). Then, keep xoring the n value, incremented by 1 each time until n is a power of 2. The result is then re-based back to the original base. Done.

Here's the algorithm I tested in PHP (using an array instead of a file, but same concept):

```
function find($array) {
    $offset = min($array);
    $n = 0;
    $result = 0;
    foreach ($array as $value) {
        $result = $result ^ ($value - $offset);
        $n++;
    }
    $n++; // This takes care of the missing value
    while ($n == 1 || 0 != ($n & ($n - 1))) {
        $result = $result ^ ($n++);
    }
    return $result + $offset;
}
```

Fed in an array with any range of values (I tested including negatives) with one inside that range which is missing, it found the correct value each time.

Another Approach

Since we can use external sorting, why not just check for a gap? If we assume the file is sorted prior to the running of this algorithm:

```
long supplied = 0;
long last = read_int_from_file();
while (supplied = read_int_from_file()) {
    if (supplied != last + 1) {
        return last + 1;
    }
    last = supplied;
}
// The range is contiguous, so what do we do here?  Let's return last + 1:
return last + 1;
```

edited Nov 2 '11 at 12:27
Stécy
3,164 5 31 67

answered Aug 24 '11 at 13:56
ircmaxell
81.7k 17 152 218

3 The problem does not say "one number is missing", it says to find a number not included in the 4 billion numbers in the file. If we assume 32-bit integers, then about 300 million numbers may be missing from the file. The likelihood of the xor of the numbers present matching a missing number is only about 7%. – jwpat7 Aug 24 '11 at 15:42

add a comment

Check the size of the input file, then output *any* number which is **too large to be represented by a file that size.** This may seem like a cheap trick, but it's a creative solution to an interview problem, it neatly sidesteps the memory issue, and it's technically O(n).

```
void maxNum(ulong filesize)
{
    ulong bitcount = filesize * 8; //number of bits in file

    for (ulong i = 0; i < bitcount; i++)
    {
        Console.Write(9);
    }
}
```

Should print **10^{bitcount} - 1**, which will always be greater than **2^{bitcount}**. Technically, the number you have to beat is **2^{bitcount} - (4 * 10^9 - 1)**, since you know there are (4 billion - 1) other integers in the file, and even with perfect compression they'll take up at least one bit each.

edited Nov 2 '11 at 12:37
Stécy
3,164 5 31 67

answered Aug 24 '11 at 4:16
Justin Morgan
13.7k 5 34 72

show **2** more comments

Trick question, unless it's been quoted improperly. Just read through the file once to get the maximum integer `n` , and return `n+1` .

Of course you'd need a backup plan in case `n+1` causes an integer overflow.

answered Aug 22 '11 at 21:37
Mark Ransom
144k 12 129 306

1 Here's a solution that works... except when it doesn't. Useful! :-) – dty Aug 22 '11 at 21:57

show **1** more comment

You can use bit flags to mark whether an integer is present or not.

After traversing the entire file, scan each bit to determine if the number exists or not.

Assuming each integer is 32 bit, they will conveniently fit in 1 GB of RAM if bit flagging is done.

answered Aug 22 '11 at 21:18
Shamim Hafiz
8,769 14 57 115

2 @dty I think he means "comfortably", as in there will be lots of room in the 1Gb. – corsiKa Aug 22 '11 at 21:52

show **1** more comment

For some reason, as soon as I read this problem I thought of diagonalization. I'm assuming arbitrarily large integers.

Read the first number. Left-pad it with zero bits until you have 4 billion bits. If the first (high-order) bit is 0, output 1; else output 0. (You don't really have to left-pad: you just output a 1 if there are not enough bits in the number.) Do the same with the second number, except use its second bit. Continue through the file in this way. You will output a 4-billion bit number one bit at a time, and that number will not be the same as any in the file. Proof: it were the same as the nth number, then they would agree on the nth bit, but they don't by construction.

answered Aug 24 '11 at 14:49 community wiki
Jonathan Amsterdam

1 @Henning The problem doesn't make sense for arbitrarily large integers because, as many people have pointed out, it's trivial to just find the largest number and add one, or construct a very long number out of the file itself. This diagonalization solution is particularly inappropriate because rather than branching on the `i` th bit you could just output 1 bits 4 billion times and throw an extra 1 on the end. I'm OK with having arbitrarily large integers *in the algorithm* but I think the problem is to output a missing 32-bit integer. It just doesn't make sense any other way. – Brian Gordon Aug 24 '11 at 15:27

show **6** more comments

I will answer the 1 GB version:

There is not enough information in the question, so I will state some assumptions first:

The integer is 32 bits with range -2,147,483,648 to 2,147,483,647.

Pseudo-code:

```
var bitArray = new bit[4294967296];  // 0.5 GB, initialized to all 0s.

foreach (var number in file) {
    bitArray[number + 2147483648] = 1;   // Shift all numbers so they start at 0.
}

for (var i = 0; i < 4294967296; i++) {
    if (bitArray[i] == 0) {
        return i - 2147483648;
    }
}
```

edited Aug 27 '11 at 7:33
Peter Mortensen
7,808 8 55 90

answered Aug 23 '11 at 12:39
BobTurbo
190 1 5

add a comment

Just for the sake of completeness, here is another very simple solution, which will most likely take a very long time to run, but uses very little memory.

Let all possible integers be the range from `int_min` to `int_max`, and `bool isNotInFile(integer)` a function which returns true if the file does not contain a certain integer and false else (by comparing that certain integer with each integer in the file)

```
for (integer i = int_min; i <= int_max; ++i)
{
    if (isNotInFile(i)) {
        return i;
    }
}
```

edited Aug 27 '11 at 7:35
Peter Mortensen
7,808 8 55 90

answered Aug 24 '11 at 11:51
deg
61 2

2 no, the question was "which integer is not in the file", not "is integer x in the file". a function to determine the answer to the latter question could for example just compare every integer in the file to the integer in question and return true on a match. – deg Aug 24 '11 at 14:33

1 @Aleks G - Right. I never said you said that either. We just say IsNotInFile can be trivially implemented using a loop on the file: Open;While Not Eof{Read Integer;Return False if Integer=i;Else Continue;}. It needs only 4 bytes of memory. – Simon Mourier Aug 24 '11 at 15:30

show **4** more comments

Bit Elimination

One way is to eliminate bits, however this might not actually yield a result (chances are it won't). Psuedocode:

```
long val = 0xFFFFFFFFFFFFFFFF; // (all bits set)
foreach long fileVal in file
{
    val = val & ~fileVal;
    if (val == 0) error;
}
```

Bit Counts

Keep track of the bit counts; and use the bits with the least amounts to generate a value. Again this has no guarantee of generating a correct value.

Range Logic

Keep track of a list ordered ranges (ordered by start). A range is defined by the structure:

```
struct Range
{
  long Start, End; // Inclusive.
}
Range startRange = new Range { Start = 0x0, End = 0xFFFFFFFFFFFFFFFF };
```

Go through each value in the file and try and remove it from the current range. This method has no memory guarantees, but it should do pretty well.

answered Aug 23 '11 at 12:26
Jonathan Dickinson
4,866 13 35

add a comment

For the 10 MB memory constraint:

1. Convert the number to its binary representation.
2. Create a binary tree where left = 0 and right = 1.
3. Insert each number in the tree using its binary representation.
4. If a number has already been inserted, the leafs will already have been created.

When finished, just take a path that has not been created before to create the requested number.

4 billion number = 2^32, meaning 10 MB might not be sufficient.

EDIT

An optimization is possible, if two ends leafs have been created and have a common parent, then they can be removed and the parent flagged as not a solution. This cuts branches and reduces the need for memory.

EDIT II

There is no need to build the tree completely too. You only need to build deep branches if numbers are similar. If we cut branches too, then this solution might work in fact.

edited Aug 23 '11 at 18:25

answered Aug 22 '11 at 21:38
JVerstry
13.6k 17 114 217

6 ... and how will that fit into 10 MB? – Henning Makholm Aug 22 '11 at 21:42

show **1** more comment

$2^{128*10^{18}}$ + 1 (which is $(2^8)^{16*10^{18}}$ + 1) - cannot it be a universal answer for today? This represents a number that cannot be held in 16 EB file, which is the maximum file size in any current file system.

answered Aug 24 '11 at 9:57
Michael Sagalovich
1,679 7 16

show **3** more comments

Strip the white space and non numeric characters from the file and append 1. Your file now contains a single number not listed in the original file.

From Reddit by Carbonetc.

answered Aug 24 '11 at 12:39
Ashley
31 1

show **1** more comment

As Ryan said it basically, sort the file and then go over the integers and when a value is skipped there you have it :)

EDIT at downvoters: the OP mentioned that the file could be sorted so this is a valid method.

edited Aug 27 '11 at 7:28
Peter Mortensen
7,808 8 55 90

answered Aug 22 '11 at 21:15
ratchet freak
25.3k 1 16 57

4 @klas merge sort is designed for that – ratchet freak Aug 24 '11 at 13:18

show **4** more comments

As long as we're doing creative answers, here is another one.

Use the external sort program to sort the input file numerically. This will work for any amount of memory you may have (it will use file storage if needed). Read through the sorted file and output the first number that is missing.

answered Jul 12 '13 at 19:38

community wiki
Rhialto

add a comment

I think this is a solved problem (see above), but there's an interesting side case to keep in mind because it might get asked:

If there are exactly 4,294,967,295 (2^32 - 1) 32-bit integers with no repeats, and therefore only one is missing, there is a simple solution.

Start a running total at zero, and for each integer in the file, add that integer with 32-bit overflow (effectively, runningTotal = (runningTotal + nextInteger) % 4294967296). Once complete, add 4294967296/2 to the running total, again with 32-bit overflow. Subtract this from 4294967296, and the result is the missing integer.

The "only one missing integer" problem is solvable with only one run, and only 64 bits of RAM dedicated to the data (32 for the running total, 32 to read in the next integer).

Corollary: The more general specification is extremely simple to match if we aren't concerned with how many bits the integer result must have. We just generate a big enough integer that it cannot be contained in the file we're given. Again, this takes up absolutely minimal RAM. See the pseudocode.

```
# Grab the file size
fseek(fp, 0L, SEEK_END);
sz = ftell(fp);
# Print a '2' for every bit of the file.
for (c=0; c<sz; c++) {
  for (b=0; b<4; b++) {
    print "2";
  }
}
```

edited Aug 24 '11 at 10:48

answered Aug 24 '11 at 10:37
Syntaera
21 3

show **1** more comment

If you don't assume the 32-bit constraint, just return a randomly generated 64-bit number (or 128-bit if you're a pessimist). The chance of collision is `1 in 2^64/(4*10^9) = 4611686018.4` (roughly 1 in 4 billion). You'd be right most of the time!

(Joking... kind of.)

edited Aug 27 '11 at 7:34
Peter Mortensen
7,808 8 55 90

answered Aug 24 '11 at 8:12
Peter Gibson
6,321 1 18 24

show **1** more comment

1 2 next

Answer This Question

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