Sorting 1 million 8-decimal-digit numbers with 1 MB of RAM

Asked 12 years, 2 months ago Modified 2 years, 10 months ago

Viewed 214k times

761

I have a computer with 1 MB of RAM and no other local storage. I must use it to accept 1 million 8-digit decimal numbers over a TCP connection, sort them, and then send the sorted list out over another TCP connection.

The list of numbers may contain duplicates, which I must not discard. The code will be placed in ROM, so I need not subtract the size of my code from the 1 MB. I already have code to drive the Ethernet port and handle TCP/IP connections, and it requires 2 KB for its state data, including a 1 KB buffer via which the code will read and write data. Is there a solution to this problem?

Sources Of Question And Answer:

slashdot.org

cleaton.net

algorithm sorting embedded ram

edited Mar 30, 2020 at 6:49

Share

Improve this question

Follow

community wiki
11 revs, 10 users 38%
phuclv

50 Ehm, a million times 8-digit decimal number (min. 27-bit integer binary) > 1MB ram – Mr47 Oct 5, 2012 at 14:19

16 1M of RAM means 2^20 bytes? And how many bits are in a byte on this architecture? And is the "million" in "1 million 8 digit decimal numbers" a SI million (10^6)? What is a 8 digit decimal number, a natural number < 10^8, a rational number whose decimal representation takes 8 digits excluding the decimal point, or something else? – user395760 Oct 5, 2012 at 14:20

15 1 million 8 decimal digit numbers or 1 million 8 bit numbers? – Patrick White Oct 5, 2012 at 14:20

15 it reminds me of an article in "Dr Dobb's Journal" (somewhere between 1998-2001), where the author used an insertion sort to sort phone numbers as he was reading them: that was the first time i realized that, sometimes, a slower algorithm may be faster... – Adrien Plisson Oct 21, 2012 at 21:29

111 There's another solution nobody has mentioned yet: buy hardware with 2MB RAM. It shouldn't be much more expensive, and it will make the problem much, *much* easier to solve. – Daniel Wagner Oct 21, 2012 at 23:48

35 Answers

Sorted by: Highest score (default)

1 2 Next

789

+200

There is one rather sneaky trick not mentioned here so far. We assume that you have no extra way to store data, but that is not strictly true.

One way around your problem is to do the following horrible thing, which should not be attempted by anyone under any circumstances: Use the network traffic to store data. And no, I don't mean NAS.

You can sort the numbers with only a few bytes of RAM in the following way:

- First take 2 variables: `COUNTER` and `VALUE`.
- First set all registers to `0`;
- Every time you receive an integer `I`, increment `COUNTER` and set `VALUE` to `max(VALUE, I)`;
- Then send an ICMP echo request packet with data set to `I` to the router. Erase `I` and repeat.
- Every time you receive the returned ICMP packet, you simply extract the integer and send it back out again in another echo request. This produces a huge number of ICMP requests scuttling backward and forward containing the integers.

Once `COUNTER` reaches `1000000`, you have all of the values stored in the incessant stream of ICMP requests, and `VALUE` now contains the maximum integer. Pick some `threshold T >> 1000000`. Set `COUNTER` to zero. Every time you receive an ICMP packet, increment `COUNTER` and send the contained integer `I` back out in another echo request, unless `I=VALUE`, in which case transmit it to the destination for the sorted integers. Once `COUNTER=T`, decrement `VALUE` by `1`, reset `COUNTER` to zero and repeat. Once `VALUE` reaches zero you should have transmitted all integers in order from largest to smallest to the destination, and have only used about 47 bits of RAM for the two persistent variables (and whatever small amount you need for the temporary values).

I know this is horrible, and I know there can be all sorts of practical issues, but I thought it might give some of you a laugh or at least horrify you.

Share Improve this answer Follow

edited Mar 30, 2020 at 8:47

community wiki
7 revs, 6 users 61%
Joe Fitzsimons

43 So you're basically leveraging network latency and turning your router into a sort of que? – Eric R. Oct 21, 2012 at 17:39

23 This isn't exactly practical. Any sysadmin with half a brain would just drop all traffic to/from that device until it stopped behaving maliciously. And yes, 100 trillion pings in a short time is malicious. – MDMarra Oct 21, 2012 at 20:48

40 @MDMarra: You'll notice right at the top I say "One way around your problem is to do the following horrible thing, which should not be attempted by anyone under any circumstances". There was a reason I said this. – Joe Fitzsimons Oct 22, 2012 at 7:38

11 There's a fundmental flaw in this solution: the 1 Million numbers *need* to be stored somewhere. Either on your device or on the router. If you're just saying, that "you send an ICMP" request, the data is either held on your local stack or on the router's stack. If the memory of your device and the router together isn't big enough to hold all the data (I'm not sure about typical memory sizes of routers...), the approach simply won't work. – MartinStettner Oct 23, 2012 at 8:27

15 @JoeFitzsimons: This approach is pretty much how memory was implemented in some historical computing systems: Delay line memory. – JPvdMerwe Oct 23, 2012 at 14:20

429

Here's some working C++ code which solves the problem.

~~Proof that the memory constraints are satisfied:~~

Editor: There is no proof of the maximum memory requirements offered by the author either in this post or in his blogs. Since the number of bits necessary to encode a value depends on the values previously encoded, such a proof is likely non-trivial. The author notes that the largest encoded size he could stumble upon empirically was `1011732`, and chose the buffer size `1013000` arbitrarily.

```
typedef unsigned int u32;

namespace WorkArea
{
    static const u32 circularSize = 253250;
    u32 circular[circularSize] = { 0 };         // con

    static const u32 stageSize = 8000;
    u32 stage[stageSize];                       // con

    ...
```

Together, these two arrays take 1045000 bytes of storage. That leaves 1048576 - 1045000 - 2×1024 = 1528 bytes for remaining variables and stack space.

It runs in about 23 seconds on my Xeon W3520. You can verify that the program works using the following Python script, assuming a program name of `sort1mb.exe`.

```
from subprocess import *
import random

sequence = [random.randint(0, 99999999) for i in xrang

sorter = Popen('sort1mb.exe', stdin=PIPE, stdout=PIPE)
for value in sequence:
    sorter.stdin.write('%08d\n' % value)
sorter.stdin.close()

result = [int(line) for line in sorter.stdout]
print('OK!' if result == sorted(sequence) else 'Error!
```

A detailed explanation of the algorithm can be found in the following series of posts:

- 1MB Sorting Explained
- Arithmetic Coding and the 1MB Sorting Problem
- Arithmetic Encoding Using Fixed-Point Math

Share Improve this answer Follow

edited Feb 25, 2015 at 9:24

community wiki
8 revs, 3 users 91%
preshing

What about the libstdc++ memory size ? :D – Gui13 Oct 25, 2012 at 12:47

---

33 I think the key observation is that an 8-digit number has about 26.6 bits of information and one million is 19.9 bits. If you delta compress the list (store the differences of adjacent values) the differences range from 0 (0 bits) to 99999999 (26.6 bits) but you can't have the maximum delta between *every* pair. The worst case should actually be one million evenly distributed values, requiring deltas of (26.6-19.9) or about 6.7 bits per delta. Storing one million values of 6.7 bits easily fits in 1M. Delta compression requires continuous merge sorting so you almost get that for free. – Ben Jackson Oct 25, 2012 at 19:31

---

4 sweet solution. y'all should visit his blog for the explanation preshing.com/20121025/… – davec Oct 26, 2012 at 6:26

---

10 @BenJackson: There is an error somewhere in your maths. There are 2.265 x 10^2436455 unique possible outputs (ordered sets of 10^6 8-digit integers) which takes 8.094 x

10^6 bits to store (i.e. a hair under a megabyte). No clever scheme can compress beyond this information theoretic limit without loss. Your explanation implies you need much less space, and is hence wrong. Indeed, "circular" in the above solution is just large enough to hold the needed information, so preshing seems to have taken this into account, but you are missing it. – Joe Fitzsimons Oct 28, 2012 at 11:03

5 @JoeFitzsimons: I had not worked out the recursion (unique sorted sets of n numbers from 0..m is `(n+m)!/(n!m!)`) so you must be right. Probably it's my estimate that a delta of b bits takes b bits to store -- clearly deltas of 0 do not take 0 bits to store. – Ben Jackson Oct 28, 2012 at 20:34

374

+50

Please see the first correct answer or the later answer with arithmetic encoding. Below you may find some fun, but not a 100% bullet-proof solution.

This is quite an interesting task and here is an another solution. I hope somebody would find the result useful (or at least interesting).

Stage 1: Initial data structure, rough compression approach, basic results

Let's do some simple math: we have 1M (1048576 bytes) of RAM initially available to store 10^6 8 digit decimal numbers. [0;99999999]. So to store one number 27 bits are needed (taking the assumption that unsigned numbers will be used). Thus, to store a raw stream ~3.5M of RAM will be needed. Somebody already said it doesn't seem to be feasible, but I would say the task can be solved if the input is "good enough". Basically, the idea is to compress the input data with compression factor 0.29 or higher and do sorting in a proper manner.

Let's solve the compression issue first. There are some relevant tests already available:

http://www.theeggeadventure.com/wikimedia/index.php/Java_Data_Compression

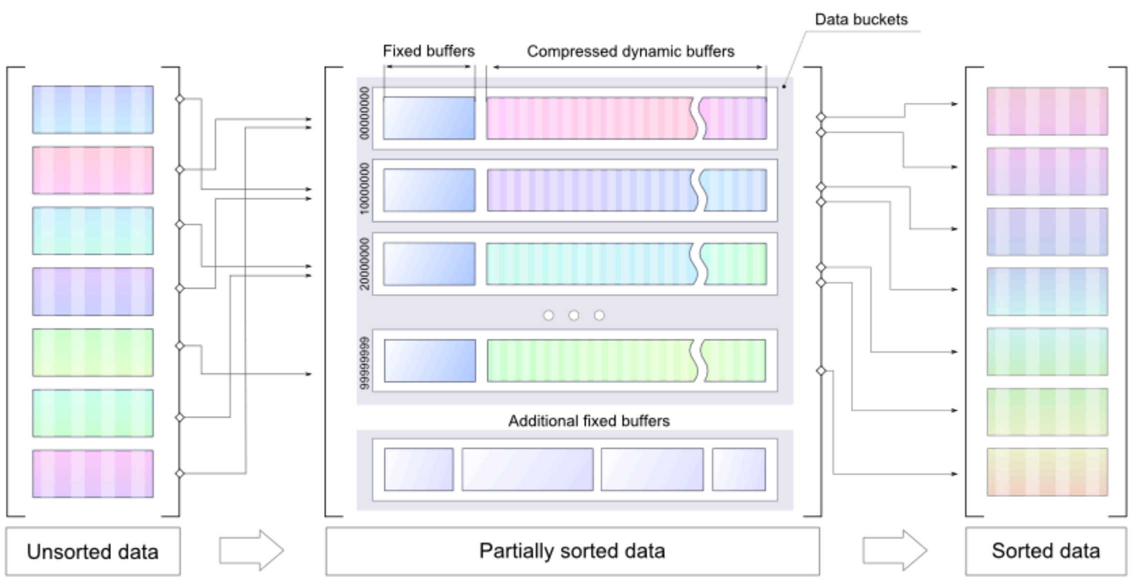
> "I ran a test to compress one million consecutive integers using various forms of compression. The results are as follows:"

```
None     4000027
Deflate  2006803
Filtered 1391833
BZip2    427067
Lzma     255040
```

It looks like LZMA (Lempel–Ziv–Markov chain algorithm) is a good choice to continue with. I've prepared a simple PoC, but there are still some details to be highlighted:

1. Memory is limited so the idea is to presort numbers and use compressed buckets (dynamic size) as temporary storage
2. It is easier to achieve a better compression factor with presorted data, so there is a static buffer for each bucket (numbers from the buffer are to be sorted before LZMA)

3. Each bucket holds a specific range, so the final sort can be done for each bucket separately
4. Bucket's size can be properly set, so there will be enough memory to decompress stored data and do the final sort for each bucket separately



Please note, attached code is a POC, it can't be used as a final solution, it just demonstrates the idea of using several smaller buffers to store presorted numbers in some optimal way (possibly compressed). LZMA is not proposed as a final solution. It is used as a fastest possible way to introduce a compression to this PoC.

See the PoC code below (please note it just a demo, to compile it LZMA-Java will be needed):

```
public class MemorySortDemo {

static final int NUM_COUNT = 1000000;
static final int NUM_MAX   = 100000000;

static final int BUCKETS      = 5;
static final int DICT_SIZE    = 16 * 1024; // LZMA dic
static final int BUCKET_SIZE  = 1024;
static final int BUFFER_SIZE  = 10 * 1024;
static final int BUCKET_RANGE = NUM_MAX / BUCKETS;

static class Producer {
    private Random random = new Random();
    public int produce() { return random.nextInt(NUM_M
}

static class Bucket {
    public int size, pointer;
    public int[] buffer = new int[BUFFER_SIZE];

    public ByteArrayOutputStream tempOut = new ByteArr
    public DataOutputStream tempDataOut = new DataOutp
    public ByteArrayOutputStream compressedOut = new B

    public void submitBuffer() throws IOException {
        Arrays.sort(buffer, 0, pointer);

        for (int j = 0; j < pointer; j++) {
            tempDataOut.writeInt(buffer[j]);
            size++;
        }
        pointer = 0;
    }

    public void write(int value) throws IOException {
        if (isBufferFull()) {
            submitBuffer();
        }
        buffer[pointer++] = value;
    }

    public boolean isBufferFull() {
        return pointer == BUFFER_SIZE;
    }

    public byte[] compressData() throws IOException {
        tempDataOut.close();
        return compress(tempOut.toByteArray());
    }
    private byte[] compress(byte[] input) throws IOExc
        final BufferedInputStream in = new BufferedInp
ByteArrayInputStream(input));
        final DataOutputStream out = new DataOutputStr
BufferedOutputStream(compressedOut));

        final Encoder encoder = new Encoder();
        encoder.setEndMarkerMode(true);
        encoder.setNumFastBytes(0x20);
        encoder.setDictionarySize(DICT_SIZE);
        encoder.setMatchFinder(Encoder.EMatchFinderTyp

        ByteArrayOutputStream encoderPrperties = new B
        encoder.writeCoderProperties(encoderPrperties)
        encoderPrperties.flush();
        encoderPrperties.close();

        encoder.code(in, out, -1, -1, null);
        out.flush();
        out.close();
        in.close();

        return encoderPrperties.toByteArray();
    }

    public int[] decompress(byte[] properties) throws
        InputStream in = new ByteArrayInputStream(comp
        ByteArrayOutputStream data = new ByteArrayOutp
        BufferedOutputStream out = new BufferedOutputS

        Decoder decoder = new Decoder();
        decoder.setDecoderProperties(properties);
        decoder.code(in, out, 4 * size);

        out.flush();
        out.close();
        in.close();

        DataInputStream input = new DataInputStream(ne
ByteArrayInputStream(data.toByteArray()));
        int[] array = new int[size];
        for (int k = 0; k < size; k++) {
            array[k] = input.readInt();
        }
        return array;
    }
}

static class Sorter {
    private Bucket[] bucket = new Bucket[BUCKETS];

    public void doSort(Producer p, Consumer c) throws

        for (int i = 0; i < bucket.length; i++) {  //
            bucket[i] = new Bucket();
        }

        for(int i=0; i< NUM_COUNT; i++) {         // p
            int value = p.produce();
            int bucketId = value/BUCKET_RANGE;
            bucket[bucketId].write(value);
            c.register(value);
        }

        for (int i = 0; i < bucket.length; i++) { // s
            bucket[i].submitBuffer();
        }

        byte[] compressProperties = null;
        for (int i = 0; i < bucket.length; i++) { // c
            compressProperties = bucket[i].compressDat
        }

        printStatistics();

        for (int i = 0; i < bucket.length; i++) { // d
by one
            int[] array = bucket[i].decompress(compres
            Arrays.sort(array);

            for(int v : array) {
                c.consume(v);
            }
        }
        c.finalCheck();
    }
    public void printStatistics() {
        int size = 0;
        int sizeCompressed = 0;

        for (int i = 0; i < BUCKETS; i++) {
            int bucketSize = 4*bucket[i].size;
            size += bucketSize;
            sizeCompressed += bucket[i].compressedOut.

            System.out.println("  bucket[" + i
                    + "] contains: " + bucket[i].size
                    + " numbers, compressed size: " +
bucket[i].compressedOut.size()
                    + String.format(" compression fact
((double)bucket[i].compressedOut.size())/bucketSize));
        }

        System.out.println(String.format("Data size: %
(double)size/(1014*1024))
                + String.format(" compressed %.2fM",
(double)sizeCompressed/(1014*1024))
                + String.format(" compression factor %
(double)sizeCompressed/size));
    }
}

static class Consumer {
    private Set<Integer> values = new HashSet<>();

    int v = -1;
    public void consume(int value) {
        if(v < 0) v = value;

        if(v > value) {
            throw new IllegalArgumentException("Curren
previous: " + v + " > " + value);
        }else{
            v = value;
            values.remove(value);
        }
    }

    public void register(int value) {
        values.add(value);
    }

    public void finalCheck() {
        System.out.println(values.size() > 0 ? "NOT OK
"OK!");
    }
}

public static void main(String[] args) throws IOExcept
    Producer p = new Producer();
    Consumer c = new Consumer();
    Sorter sorter = new Sorter();

    sorter.doSort(p, c);
}
}
```

With random numbers it produces the following:

```
bucket[0] contains: 200357 numbers, compressed
size: 353679 compression factor: 0.44
bucket[1] contains: 199465 numbers, compressed
size: 352127 compression factor: 0.44
bucket[2] contains: 199682 numbers, compressed
size: 352464 compression factor: 0.44
bucket[3] contains: 199949 numbers, compressed
size: 352947 compression factor: 0.44
bucket[4] contains: 200547 numbers, compressed
size: 353914 compression factor: 0.44
Data size: 3.85M compressed 1.70M compression
factor 0.44
```

For a simple ascending sequence (one bucket is used) it produces:

```
bucket[0] contains: 1000000 numbers, compressed
size: 256700 compression factor: 0.06
Data size: 3.85M compressed 0.25M compression
factor 0.06
```

EDIT

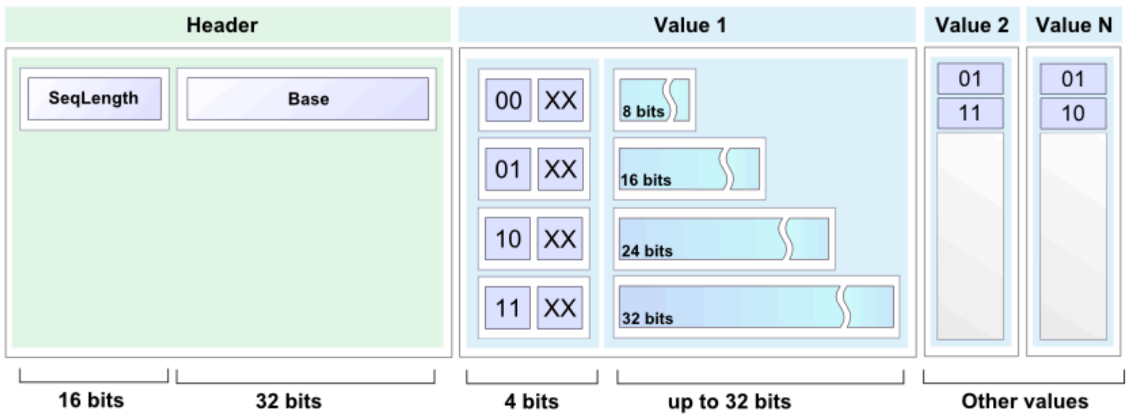
Conclusion:

1. Don't try to fool the Nature
2. Use simpler compression with lower memory footprint
3. Some additional clues are really needed. Common bullet-proof solution does not seem to be feasible.

Stage 2: Enhanced compression, final conclusion

As was already mentioned in the previous section, any suitable compression technique can be used. So let's get rid of LZMA in favor of simpler and better (if possible) approach. There are a lot of good solutions including Arithmetic coding, Radix tree etc.

Anyway, simple but useful encoding scheme will be more illustrative than yet another external library, providing some nifty algorithm. The actual solution is pretty straightforward: since there are buckets with partially sorted data, deltas can be used instead of numbers.



Random input test shows slightly better results:

```
bucket[0] contains: 10103 numbers, compressed
size: 13683 compression factor: 0.34
bucket[1] contains: 9885 numbers, compressed size:
13479 compression factor: 0.34
...
bucket[98] contains: 10026 numbers, compressed
size: 13612 compression factor: 0.34
bucket[99] contains: 10058 numbers, compressed
size: 13701 compression factor: 0.34
Data size: 3.85M compressed 1.31M compression
factor 0.34
```

Sample code

```
  public static void encode(int[] buffer, int length,
    short size = (short)(length & 0x7FFF);

    output.write(size);
    output.write(buffer[0]);

    for(int i=1; i< size; i++) {
        int next = buffer[i] - buffer[i-1];
        int bits = getBinarySize(next);

        int len = bits;

        if(bits > 24) {
          output.write(3, 2);
          len = bits - 24;
        }else if(bits > 16) {
          output.write(2, 2);
          len = bits-16;
        }else if(bits > 8) {
          output.write(1, 2);
          len = bits - 8;
        }else{
          output.write(0, 2);
        }
        if (len > 0) {
            if ((len % 2) > 0) {
                len = len / 2;
                output.write(len, 2);
                output.write(false);
            } else {
                len = len / 2 - 1;
                output.write(len, 2);
            }

            output.write(next, bits);
        }
    }
}

public static short decode(BinaryIn input, int[] buffe
    short length = input.readShort();
    int value = input.readInt();
    buffer[offset] = value;

    for (int i = 1; i < length; i++) {
        int flag = input.readInt(2);

        int bits;
        int next = 0;
        switch (flag) {
            case 0:
                bits = 2 * input.readInt(2) + 2;
                next = input.readInt(bits);
                break;
            case 1:
                bits = 8 + 2 * input.readInt(2) +2;
                next = input.readInt(bits);
                break;
            case 2:
                bits = 16 + 2 * input.readInt(2) +2;
                next = input.readInt(bits);
                break;
            case 3:
                bits = 24 + 2 * input.readInt(2) +2;
                next = input.readInt(bits);
                break;
        }
```

```
            buffer[offset + i] = buffer[offset + i - 1] +
        }

        return length;
    }
```

Please note, this approach:

1. does not consume a lot of memory
2. works with streams
3. provides not so bad results

Full code can be found here, BinaryInput and BinaryOutput implementations can be found here

Final conclusion

No final conclusion :) Sometimes it is really good idea to move one level up and review the task from a meta-level point of view.

It was fun to spend some time with this task. BTW, there are a lot of interesting answers below. Thank you for your attention and happy codding.

Share Improve this answer Follow

edited May 23, 2017 at 12:10

community wiki
11 revs, 4 users 98%
Renat Gilmanov

17 I used Inkscape. Great tool by the way. You can use this diagram source as an example. – Renat Gilmanov Oct 21, 2012 at 10:44

22 Surely LZMA requires too much memory to be useful in this case? As an algorithm it's meant to minimise the amount of data that has to be stored or transmitted, rather than be efficient in memory. – Mjiig Oct 21, 2012 at 14:11

68 This is nonsense... Get 1 million random 27 bit integers, sort them, compress with 7zip, xz, whatever LZMA you want. Result is over 1MB. The premise on top is compression of sequential numbers. Delta encoding of that with 0bit would be just the number, e.g. 1000000 (say in 4 bytes). With sequential and duplicates (no gaps) , the number 1000000 and 1000000 bits = 128KB, with 0 for duplicate number and 1 to mark next. When you have random gaps, even small, LZMA is ridiculous. It's not designed for this. – alecco Oct 21, 2012 at 21:15

31 This won't actually work. I ran a simulation and while the compressed data is more than 1MB (about 1.5MB), it still uses over 100MB of RAM to compress the data. So even the compressed integers don't fit the problem not to mention run time RAM usage. Awarding you the bounty is my biggest error on stackoverflow. – Favourite Onwuemene Oct 22, 2012 at 9:59

10 This answer is upvoted so much because a lot of programmers like shiny ideas rather than proven code. If this idea worked, you'd see an actual compression algorithm chosen and proven rather than a mere assertion that surely there's one out there that can do it...when it's quite possible that there isn't one out there that can do it. – Olathe Oct 22, 2012 at 12:28

188

A solution is possible only because of the difference between 1 megabyte and 1 million bytes. There are about 2 to the power 8093729.5 different ways to choose 1 million 8-digit numbers with duplicates allowed and order unimportant, so a machine with only 1 million bytes of RAM doesn't have enough states to represent all the possibilities. But 1M (less 2k for TCP/IP) is 1022*1024*8 = 8372224 bits, so a solution is possible.

Part 1, initial solution

This approach needs a little more than 1M, I'll refine it to fit into 1M later.

I'll store a compact sorted list of numbers in the range 0 to 99999999 as a sequence of sublists of 7-bit numbers. The first sublist holds numbers from 0 to 127, the second sublist holds numbers from 128 to 255, etc. 100000000/128 is exactly 781250, so 781250 such sublists will be needed.

Each sublist consists of a 2-bit sublist header followed by a sublist body. The sublist body takes up 7 bits per sublist entry. The sublists are all concatenated together, and the format makes it possible to tell where one sublist ends and the next begins. The total storage required for a fully populated list is 2*781250 + 7*1000000 = 8562500 bits, which is about 1.021 M-bytes.

The 4 possible sublist header values are:

00 Empty sublist, nothing follows.

01 Singleton, there is only one entry in the sublist and and next 7 bits hold it.

10 The sublist holds at least 2 distinct numbers. The entries are stored in non-decreasing order, except that the last entry is less than or equal to the first. This allows the end of the sublist to be identified. For example, the numbers 2,4,6 would be stored as (4,6,2). The numbers 2,2,3,4,4 would be stored as (2,3,4,4,2).

11 The sublist holds 2 or more repetitions of a single number. The next 7 bits give the number. Then come zero or more 7-bit entries with the value 1, followed by a 7-bit entry with the value 0. The length of the sublist body dictates the number of repetitions. For example, the numbers 12,12 would be stored as (12,0), the numbers 12,12,12 would be stored as (12,1,0), 12,12,12,12 would be (12,1,1,0) and so on.

I start off with an empty list, read a bunch of numbers in and store them as 32 bit integers, sort the new numbers in place (using heapsort, probably) and then merge them into a new compact sorted list. Repeat until there are no more numbers to read, then walk the compact list once more to generate the output.

The line below represents memory just before the start of the list merge operation. The "O"s are the region that hold the sorted 32-bit integers. The "X"s are the region that hold the old compact list. The "=" signs are the expansion room for the compact list, 7 bits for each integer in the "O"s. The "Z"s are other random overhead.

```
ZZZOOOOOOOOOOOOOOOOOOOOOOOOOO==========XXXXXXXXXXXX
```

The merge routine starts reading at the leftmost "O" and at the leftmost "X", and starts writing at the leftmost "=". The write pointer doesn't catch the compact list read pointer until all of the new integers are merged, because both pointers advance 2 bits for each sublist and 7 bits for each entry in the old compact list, and there is enough extra room for the 7-bit entries for the new numbers.

Part 2, cramming it into 1M

To Squeeze the solution above into 1M, I need to make the compact list format a bit more compact. I'll get rid of one of the sublist types, so that there will be just 3 different possible sublist header values. Then I can use "00", "01" and "1" as the sublist header values and save a few bits. The sublist types are:

A Empty sublist, nothing follows.

B Singleton, there is only one entry in the sublist and and next 7 bits hold it.

C The sublist holds at least 2 distinct numbers. The entries are stored in non-decreasing order, except that the last entry is less than or equal to the first. This allows the end of the sublist to be identified. For example, the numbers 2,4,6 would be stored as (4,6,2). The numbers 2,2,3,4,4 would be stored as (2,3,4,4,2).

D The sublist consists of 2 or more repetitions of a single number.

My 3 sublist header values will be "A", "B" and "C", so I need a way to represent D-type sublists.

Suppose I have the C-type sublist header followed by 3 entries, such as "C[17][101][58]". This can't be part of a valid C-type sublist as described above, since the third entry is less than the second but more than the first. I can use this type of construct to represent a D-type sublist. In bit terms, anywhere I have "C{00?????}{1??????}{01?????}" is an impossible C-type sublist. I'll use this to represent a sublist consisting of 3 or more repetitions of a single number. The first two 7-bit words encode the number (the "N" bits below) and are followed by zero or more {0100001} words followed by a {0100000} word.

```
For example, 3 repetitions: "C{00NNNNN}{1NN0000}
{0100000}", 4 repetitions: "C{00NNNNN}{1NN0000}
{0100001}{0100000}", and so on.
```

That just leaves lists that hold exactly 2 repetitions of a single number. I'll represent those with another impossible C-type sublist pattern: "C{0??????}{11?????}{10?????}". There's plenty of room for the 7 bits of the number in the first 2 words, but this pattern is longer than the sublist that it represents, which makes things a bit more complex. The five question-marks at the end can be considered not part of the pattern, so I have: "C{0NNNNNN}{11N????}10" as my pattern, with the number to be repeated stored in the "N"s. That's 2 bits too long.

I'll have to borrow 2 bits and pay them back from the 4 unused bits in this pattern. When reading, on encountering "C{0NNNNNN}{11N00AB}10", output 2 instances of the number in the "N"s, overwrite the "10" at the end with bits A and B, and rewind the read pointer by 2 bits. Destructive reads are ok for this algorithm, since each compact list gets walked only once.

When writing a sublist of 2 repetitions of a single number, write "C{0NNNNNN}11N00" and set the borrowed bits counter to 2. At every write where the borrowed bits counter is non-zero, it is decremented for each bit written and "10" is written when the counter hits zero. So the next 2 bits written will go into slots A and B, and then the "10" will get dropped onto the end.

With 3 sublist header values represented by "00", "01" and "1", I can assign "1" to the most popular sublist type. I'll need a small table to map sublist header values to sublist types, and I'll need an occurrence counter for each sublist type so that I know what the best sublist header mapping is.

The worst case minimal representation of a fully populated compact list occurs when all the sublist types are equally popular. In that case I save 1 bit for every 3 sublist headers, so the list size is 2*781250 + 7*1000000 - 781250/3 = 8302083.3 bits. Rounding up to a 32 bit word boundary, thats 8302112 bits, or 1037764 bytes.

1M minus the 2k for TCP/IP state and buffers is 1022*1024 = 1046528 bytes, leaving me 8764 bytes to play with.

But what about the process of changing the sublist header mapping ? In the memory map below, "Z" is random overhead, "=" is free space, "X" is the compact list.

```
ZZZ=====XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Start reading at the leftmost "X" and start writing at the leftmost "=" and work right. When it's done the compact list will be a little shorter and it will be at the wrong end of memory:

```
ZZZXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

So then I'll need to shunt it to the right:

```
ZZZ=======XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

In the header mapping change process, up to 1/3 of the sublist headers will be changing from 1-bit to 2-bit. In the worst case these will all be at the head of the list, so I'll need at least 781250/3 bits of free storage before I start, which takes me back to the memory requirements of the previous version of the compact list :(

To get around that, I'll split the 781250 sublists into 10 sublist groups of 78125 sublists each. Each group has its own independent sublist header mapping. Using the letters A to J for the groups:

```
ZZZ=====AAAAAABBCCCCDDDDDEEEFFFGGGGGGGGGGGHHIJJJJJJJ
```

Each sublist group shrinks or stays the same during a sublist header mapping change:

```
ZZZ=====AAAAAABBCCCCDDDDDEEEFFFGGGGGGGGGGGHHIJJJJJJJ
ZZZAAAAAA=====BBCCCCDDDDDEEEFFFGGGGGGGGGGGHHIJJJJJJJ
ZZZAAAAAABB=====CCCCDDDDDEEEFFFGGGGGGGGGGGHHIJJJJJJJ
ZZZAAAAAABBCCC======DDDDDEEEFFFGGGGGGGGGGGHHIJJJJJJJ
ZZZAAAAAABBCCCDDDDD======EEEFFFGGGGGGGGGGGHHIJJJJJJJ
ZZZAAAAAABBCCCDDDDDEEE======FFFGGGGGGGGGGGHHIJJJJJJJ
ZZZAAAAAABBCCCDDDDDEEEFFF======GGGGGGGGGGGHHIJJJJJJJ
ZZZAAAAAABBCCCDDDDDEEEFFFGGGGGGGGGG=======HHIJJJJJJJ
ZZZAAAAAABBCCCDDDDDEEEFFFGGGGGGGGGGHH=======IJJJJJJJ
ZZZAAAAAABBCCCDDDDDEEEFFFGGGGGGGGGGHHI=======JJJJJJJ
ZZZAAAAAABBCCCDDDDDEEEFFFGGGGGGGGGGHHIJJJJJJJJJJJJJJ
ZZZ=======AAAAAABBCCCDDDDDEEEFFFGGGGGGGGGGHHIJJJJJJJ
```

The worst case temporary expansion of a sublist group during a mapping change is 78125/3 = 26042 bits, under 4k. If I allow 4k plus the 1037764 bytes for a fully populated compact list, that leaves me 8764 - 4096 = 4668 bytes for the "Z"s in the memory map.

That should be plenty for the 10 sublist header mapping tables, 30 sublist header occurrence counts and the other few counters, pointers and small buffers I'll need, and space I've used without noticing, like stack space for function call return addresses and local variables.

Part 3, how long would it take to run?

With an empty compact list the 1-bit list header will be used for an empty sublist, and the starting size of the list will be 781250 bits. In the worst case the list grows 8 bits for each number added, so 32 + 8 = 40 bits of free space are needed for each of the 32-bit numbers to be placed at the top of the list buffer and then sorted and merged. In the worst case, changing the sublist header mapping results in a space usage of 2*781250 + 7*entries - 781250/3 bits.

With a policy of changing the sublist header mapping after every fifth merge once there are at least 800000 numbers in the list, a worst case run would involve a total of about 30M of compact list reading and writing activity.

Source:

http://nick.cleaton.net/ramsortsol.html

Share Improve this answer Follow

edited Oct 11, 2015 at 15:05

community wiki
4 revs, 3 users 93%
Favourite Chigozie Onwuemene

---

15 I don't think any better solution is possible (in case we need to work with any incompressible values). But this one may be a little improved. It's not necessary to change sublist headers between 1-bit and 2-bit representations. Instead you can use

arithmetic coding, which simplifies the algorithm and also decreases worst-case number of bits per header from 1.67 to 1.58. And you don't need to move compact list in memory; instead use circular buffer and change only pointers. – Evgeny Kluev Oct 21, 2012 at 15:48

5 So, finally, was that an interview question? – mlvljr Oct 21, 2012 at 20:59

2 Other possible improvement is to use 100-element sublists instead of 128-element sublists (because we get most compact representation when number of sublists is equal to the number of elements in data set). Each value of the sublist to be encoded with arithmetic coding (with equal frequency of 1/100 for each value). This can save about 10000 bits, much less than compression of sublist headers. – Evgeny Kluev Oct 22, 2012 at 11:30

1 He has a special case for when a list is just a list of one or more repetitions of a single number. – aronchick Nov 2, 2012 at 23:25

1 A simpler solution of sublist header encoding is possible with the same compression ratio 1.67 bits per subheader without complicated switching the mapping. You can combine every 3 consecutive subheaders together, which can be easy encoded into 5 bits because `3 * 3 * 3 = 27 < 32`. You combine them `combined_subheader = subheader1 + 3 * subheader2 + 9 * subheader3`. – hynekcer Nov 25, 2012 at 17:27

57

Gilmanov's answer is very wrong in its assumptions. It starts speculating based in a *pointless* measure of a million **consecutive** integers. That means no gaps. Those random gaps, however small, really makes it a poor idea.

Try it yourself. Get 1 million random 27-bit integers, sort them, compress with 7-Zip, xz, whatever LZMA you want. The result is over 1.5 MB. The premise on top is the compression of sequential numbers. Even *delta encoding* of that is **over 1.1 MB**. And never mind, this is using over 100 MB of RAM for compression. So even the compressed integers don't fit the problem *and never mind run time RAM usage*.

It's saddens me how people just upvote pretty graphics and rationalization.

```
#include <stdint.h>
#include <stdlib.h>
#include <time.h>

int32_t ints[1000000]; // Random 27-bit integers

int cmpi32(const void *a, const void *b) {
    return ( *(int32_t *)a - *(int32_t *)b );
}

int main() {
    int32_t *pi = ints; // Pointer to input ints (REPL

    // Fill pseudo-random integers of 27 bits
    srand(time(NULL));
    for (int i = 0; i < 1000000; i++)
        ints[i] = rand() & ((1<<27) - 1); // Random 32
    qsort(ints, 1000000, sizeof (ints[0]), cmpi32); //

    // Now delta encode, optional, store differences t
    for (int i = 1, prev = ints[0]; i < 1000000; i++)
        ints[i] -= prev;
        prev    += ints[i];
    }

    FILE *f = fopen("ints.bin", "w");
    fwrite(ints, 4, 1000000, f);
    fclose(f);
    exit(0);

}
```

Now compress `ints.bin` with LZMA...

```
    $ xz -f --keep ints.bin       # 100 MB RAM
    $ 7z a ints.bin.7z ints.bin   # 130 MB RAM
    $ ls -lh ints.bin*
        3.8M ints.bin
        1.1M ints.bin.7z
        1.2M ints.bin.xz
```

Share Improve this answer Follow

edited Jan 26, 2022 at 11:39

community wiki
3 revs, 3 users 67%
alecco

---

7 any algorithm involving dictionary based compression is just beyond retarded, i have coded a few custom ones and all they take *quite* a bit of memory just to place their own hash tables (and no HashMap in java as it's extra hungry on resources). The closest solution would be delta encoding w/ variable bit length and bouncing back the TCP packets you

don't like. The peer will retransmit, still whacky at best. – bestsss Oct 21, 2012 at 23:04

@bestsss yeah! check out my last in-progress answer. I think it *might* be possible. – alecco Oct 21, 2012 at 23:14

6 Sorry but this doesn't seem to *answer the question* either, actually. – n611x007 Jan 9, 2013 at 17:09

@naxa yes it answers: it cannot be done within the original question's parameters. It can only be done if the distribution of the numbers has very low entropy. – alecco Oct 6, 2015 at 15:12

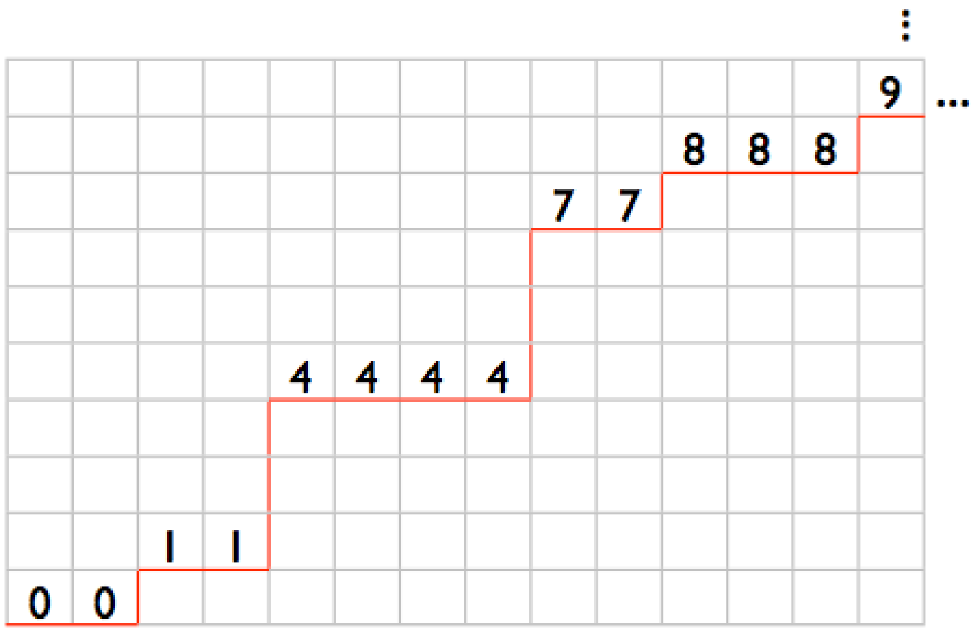
1 All this answer shows is that standard compression routines have difficulty compressing the data below 1MB. There may or may not be an encoding scheme that can compress the data to require less than 1MB, but this answer does not prove that there is no encoding scheme that will compress the data this much. – Itsme2003 Dec 26, 2018 at 19:52

42

I think one way to think about this is from a combinatorics viewpoint: how many possible combinations of sorted number orderings are there? If we give the combination 0,0,0,....,0 the code 0, and 0,0,0,...,1 the code 1, and 99999999, 99999999, ... 99999999 the code N, what is N? In other words, how big is the result space?

Well, one way to think about this is noticing that this is a bijection of the problem of finding the number of monotonic paths in an N x M grid, where N = 1,000,000 and M = 100,000,000. In other words, if you have a grid that is 1,000,000 wide and 100,000,000 tall, how many shortest paths from the bottom left to the top right are there? Shortest paths of course require you only ever either move right or up (if you were to move down or left you would be undoing previously accomplished progress). To see how this is a bijection of our number sorting problem, observe the following:

You can imagine any horizontal leg in our path as a number in our ordering, where the Y location of the leg represents the value.



This path represents the numbers 0,0,1,1,4,4,4,4,7,7,8,8,8,9, ...

So if the path simply moves to the right all the way to the end, then jumps all the way to the top, that is equivalent to the ordering 0,0,0,...,0. if it instead begins by jumping all the way to the top and then moves to the right 1,000,000 times, that is equivalent to 99999999,99999999,..., 99999999. A path where it moves right once, then up once, then right one, then up once, etc to the very end (then necessarily jumps all the way to the top), is equivalent to 0,1,2,3,...,999999.

Luckily for us this problem has already been solved, such a grid has (N + M) Choose (M) paths:

(1,000,000 + 100,000,000) Choose (100,000,000) ~= 2.27 * 10^2436455

N thus equals 2.27 * 10^2436455, and so the code 0 represents 0,0,0,...,0 and the code 2.27 * 10^2436455 and some change represents 99999999,99999999,..., 99999999.

In order to store all the numbers from 0 to 2.27 * 10^2436455 you need lg2 (2.27 * 10^2436455) = 8.0937 * 10^6 bits.

1 megabyte = 8388608 bits > 8093700 bits

So it appears that we at least actually have enough room to store the result! Now of course the interesting bit is doing the sorting as the numbers stream in. Not sure the best approach to this is given we have 294908 bits remaining. I imagine an interesting technique would be to at each point assume that that is is the entire ordering, finding the code for that ordering, and then as you receive a new number going back and updating the previous code. Hand wave hand wave.

Share Improve this answer Follow

edited Oct 21, 2012 at 23:06

community wiki

3 revs
Francisco Ryan Tolmasky I

---

1 This is really a lot of hand waving. On the one hand, theoretically this is the solution because we can just write a big -- but still finite -- state machine; on the other hand, the size of the instruction pointer for that big state machine might be more than one megabyte, rendering this a non-starter. It really does require rather a bit more thought than this to actually solve the given problem. We need to not only represent all the states, but also all the transitional states needed to compute what to do on any given next input number. – Daniel Wagner Oct 22, 2012 at 1:18

---

5 I think the other answers are just more subtle about their hand waving. Given that we now know the size of the result space, we know how much space we absolutely need. No other answer will be able to store every possible answer in anything smaller than 8093700 bits, since thats how many final states there can be. Doing compress(final-state) can at best *sometimes* reduce the space, but there will always be some answer that requires the full space (no compression algorithm can compress every input).
– Francisco Ryan Tolmasky I Oct 22, 2012 at 1:22

Several other answers have already mentioned the hard lower bound anyway (e.g. the second sentence of the original question-asker's answer), so I'm not sure I see what this answer is adding to the gestalt. – Daniel Wagner Oct 22, 2012 at 1:27

---

1 "There are about 2 to the power 8093729.5 different ways to choose 1 million 8-digit numbers with duplicates allowed and order unimportant" <- from the original question-asker's answer. Don't know how to be any more clear about what bound I'm talking about. I referred pretty specifically to this sentence in my last comment. – Daniel Wagner Oct 22, 2012 at 5:01

1 My apologies thats why I asked. Anyways simply knowing the lower bound still lacks the additional insight that that size could perhaps in some way be considered the answer itself. The goal of my answer was to show how we arrived at the lower bound and what that bound actually told us about the problem. I guess the point of my post was more along the lines of "for any solution to exist, the rest must be doable in the remaining bits". I think this helps get a deeper understanding of whats going on vs. directly jumping into list compression implementations. Im sorry if you don't find that useful. – Francisco Ryan Tolmasky I Oct 22, 2012 at 6:34

20

My suggestions here owe a lot to Dan's solution

First off I assume the solution must handle *all* possible input lists. I think the popular answers do not make this assumption (which IMO is a huge mistake).

It is known that no form of lossless compression will reduce the size of all inputs.

All the popular answers assume they will be able to apply compression effective enough to allow them extra space. In fact, a chunk of extra space large enough to hold some portion of their partially completed list in an uncompressed form and allow them to perform their sorting operations. This is just a bad assumption.

For such a solution, anyone with knowledge of how they do their compression will be able to design some input data that does not compress well for this scheme, and the "solution" will most likely then break due to running out of space.

Instead I take a mathematical approach. Our possible outputs are all the lists of length LEN consisting of elements in the range 0..MAX. Here the LEN is 1,000,000 and our MAX is 100,000,000.

For arbitrary LEN and MAX, the amount of bits needed to encode this state is:

Log2(MAX Multichoose LEN)

So for our numbers, once we have completed recieving and sorting, we will need at least Log2(100,000,000 MC 1,000,000) bits to store our result in a way that can uniquely distinguish all possible outputs.

This is ~= 988kb. So we actually have enough space to hold our result. From this point of view, it is possible.

[Deleted pointless rambling now that better examples exist...]

Best answer is here.

Another good answer is here and basically uses insertion sort as the function to expand the list by one element (buffers a few elements and pre-sorts, to allow insertion of more than one at a time, saves a bit of time). uses a nice compact state encoding too, buckets of seven bit deltas

Share Improve this answer Follow

edited May 23, 2017 at 12:10

community wiki
8 revs
davec

Always fun to re-read your own answer the next day... So while the top answer is wrong, the accepted one stackoverflow.com/a/12978097/1763801 is pretty good. Basically uses insertion sort as the function to take list LEN-1 and return LEN. Capitalizes on the fact that if you presort a small set you can insert them all in one pass, to increase efficiency. The state representation is pretty compact (buckets of 7 bit numbers) better than my hand-wavy suggestion and more intuitive. my comp geo thoughts were bollocks, sorry about that – davec Oct 22, 2012 at 15:02

1 I think your arithmetic is a little off. I get lg2(100999999!/(99999999! * 1000000!)) = 1011718.55 – NovaDenizen Oct 22, 2012 at 17:29

Yes thanks it was 988kb not 965. I was sloppy in terms of 1024 versus 1000. We are still left with about 35kb to play around with. I added a link to math calculation in the answer. – davec Oct 22, 2012 at 17:59

19

Suppose this task is possible. Just prior to output, there will be an in-memory representation of the million sorted numbers. How many different such representations are there? Since there may be repeated numbers we can't use nCr (choose), but there is an operation called multichoose that works on multisets.

- There are 2.2e2436455 ways to choose a million numbers in range 0..99,999,999.
- That requires 8,093,730 bits to represent every possible combination, or 1,011,717 bytes.

So theoretically it may be possible, if you can come up with a sane (enough) representation of the sorted list of numbers. For example, an insane representation might require a 10MB lookup table or thousands of lines of code.

However, if "1M RAM" means one million bytes, then clearly there is not enough space. The fact that 5% more memory makes it theoretically possible suggests to me that the representation will have to be VERY efficient and probably not sane.

Share Improve this answer Follow

edited Oct 22, 2012 at 17:55

community wiki
2 revs
Dan

The number of ways to choose a million numbers (2.2e2436455) happens to be close to (256 ^ (1024 * 988)), which is (2.0e2436445). Ergo, if you take away about 32 KB of memory from the 1M, the problem cannot be solved. Also keep in mind at least 3 KB of memory was reserved.
– johnwbyrd Oct 21, 2012 at 23:20

This of course assumes the data is completely random. As far as we know, it is, but I'm just saying :) – Thorarin Oct 22, 2012 at 5:46

The conventional way to represent this number of possible states is by taking the log base 2 and reporting the number of bits required to represent them. – NovaDenizen Oct 22, 2012 at 17:31

@Thorarin, yup, I see no point in a "solution" that only works for some inputs. – Dan Oct 22, 2012 at 17:56

12

(My original answer was wrong, sorry for the bad math, see below the break.)

How about this?

The first 27 bits store the lowest number you have seen, then the difference to the next number seen, encoded as follows: 5 bits to store the number of bits used in storing the difference, then the difference. Use 00000 to indicate that you saw that number again.

This works because as more numbers are inserted, the average difference between numbers goes down, so you use less bits to store the difference as you add more numbers. I believe this is called a delta list.

The worst case I can think of is all numbers evenly spaced (by 100), e.g. Assuming 0 is the first number:

```
000000000000000000000000000 00111 1100100
                            ^^^^^^^^^^^^^
                            a million times
```

```
27 + 1,000,000 * (5+7) bits = ~ 427k
```

Reddit to the rescue!

If all you had to do was sort them, this problem would be easy. It takes 122k (1 million bits) to store which numbers you have seen (0th bit on if 0 was seen, 2300th bit on if 2300 was seen, etc.

You read the numbers, store them in the bit field, and then shift the bits out while keeping a count.

BUT, you have to remember how many you have seen. I was inspired by the sublist answer above to come up with this scheme:

Instead of using one bit, use either 2 or 27 bits:

- 00 means you did not see the number.
- 01 means you saw it once
- 1 means you saw it, and the next 26 bits are the count of how many times.

I think this works: if there are no duplicates, you have a 244k list. In the worst case you see each number twice (if you see one number three times, it shortens the rest of the list for you), that means you have seen 50,000 more than once, and you have seen 950,000 items 0 or 1 times.

50,000 * 27 + 950,000 * 2 = 396.7k.

You can make further improvements if you use the following encoding:

0 means you did not see the number 10 means you saw it once 11 is how you keep count

Which will, on average, result in 280.7k of storage.

EDIT: my Sunday morning math was wrong.

The worst case is we see 500,000 numbers twice, so the math becomes:

500,000 *27 + 500,000 *2 = 1.77M

The alternate encoding results in an average storage of

500,000 * 27 + 500,000 = 1.70M

: (

Share Improve this answer Follow

edited Oct 21, 2012 at 19:25

community wiki
5 revs, 2 users 87%
jfernand

1 Well, no, since the second number would be 500000.
– jfernand Oct 21, 2012 at 14:55

Maybe add some intermediate, like where 11 means you've seen the number up to 64 times (using the next 6 bits), and 11000000 means use another 32 bits to store the number of times you've seen it. – τεκ Oct 21, 2012 at 15:23

10 Where did you get the "1 million bits" number? You said the 2300th bit represents whether 2300 was seen. (I think you actually meant 2301st.) Which bit represents whether 99,999,999 was seen (the largest 8-digit number)? Presumably, it would be the 100-millionth bit. – user94559 Oct 21, 2012 at 15:24

You got your one million and your hundred million backwards. The most times a value can possibly occur is 1 million, and you only need 20 bits to represent the number of occurrences of a value. Likewise you need 100,000,000 bit fields (not 1 million), one for each possible value. – Tim R. Oct 21, 2012 at 15:52

Uh, 27+1000000*(5+7) = 12000027 bits = 1.43M, not 427K. – Daniel Wagner Oct 21, 2012 at 18:27

9

There is one solution to this problem across all possible inputs. Cheat.

1. Read m values over TCP, where m is near the max that can be sorted in memory, maybe n/4.
2. Sort the 250,000 (or so) numbers and output them.
3. Repeat for the other 3 quarters.
4. Let the receiver merge the 4 lists of numbers it has received as it processes them. (It's not much slower than using a single list.)

Share Improve this answer Follow

answered Oct 21, 2012 at 19:39

community wiki
xpda

9

What kind of computer are you using? It may not have any other "normal" local storage, but does it have video RAM, for example? 1 megapixel x 32 bits per pixel (say) is pretty close to your required data input size.

(I largely ask in memory of the old Acorn RISC PC, which could 'borrow' VRAM to expand the available system RAM, if you chose a low resolution or low colour-depth screen mode!). This was rather useful on a machine with only a few MB of normal RAM.

Share Improve this answer Follow

edited Oct 21, 2012 at 20:22

community wiki
2 revs
DNA

1 There may be no computer at all, as relevant thread on hacker news mentions this once was a Google interview question. – mlvljr Oct 21, 2012 at 20:57

2 Yes - I answered before the question was edited to indicate that it's an interview question! – DNA Oct 28, 2012 at 8:51

7

I would try a Radix Tree. If you could store the data in a tree, you could then do an in-order traverse to transmit the data.

I'm not sure you could fit this into 1MB, but I think it's worth a try.

Share Improve this answer Follow

answered Oct 21, 2012 at 16:33

community wiki
Alex Chamberlain

6

A radix tree representation would come close to handling this problem, since the radix tree takes advantage of "prefix compression". But it's hard to conceive of a radix tree representation that could represent a single node in one byte -- two is probably about the limit.

But, regardless of how the data is represented, once it is sorted it can be stored in prefix-compressed form, where the numbers 10, 11, and 12 would be represented by, say 001b, 001b, 001b, indicating an increment of 1 from the previous number. Perhaps, then, 10101b would represent an increment of 5, 1101001b an increment of 9, etc.

Share Improve this answer Follow

answered Oct 21, 2012 at 13:24

community wiki
Hot Licks

6

There are 10^6 values in a range of 10^8, so there's one value per hundred code points on average. Store the distance from the Nth point to the (N+1)th. Duplicate values have a skip of 0. This means that the skip needs an average of just under 7 bits to store, so a million of them will happily fit into our 8 million bits of storage.

These skips need to be encoded into a bitstream, say by Huffman encoding. Insertion is by iterating through the bitstream and rewriting after the new value. Output by iterating through and writing out the implied values. For practicality, it probably wants to be done as, say, 10^4 lists covering 10^4 code points (and an average of 100 values) each.

A good Huffman tree for random data can be built a priori by assuming a Poisson distribution (mean=variance=100) on the length of the skips, but real statistics can be kept on the input and used to generate an optimal tree to deal with pathological cases.

Share Improve this answer Follow

answered Oct 21, 2012 at 20:54

community wiki
Russ Williams

5

> I have a computer with 1M of RAM and no other local storage

Another way to cheat: you could use non-local (networked) storage instead (your question does not preclude this) and call a networked service that could use straightforward disk-based mergesort (or just enough RAM to sort in-memory, since you only need to accept 1M numbers), without needing the (admittedly extremely ingenious) solutions already given.

This might be cheating, but it's not clear whether you are looking for a solution to a real-world problem, or a puzzle that invites bending of the rules... if the latter, then a simple cheat may get better results than a complex but "genuine" solution (which as others have pointed out, can only work for compressible inputs).

Share Improve this answer Follow

edited Oct 21, 2012 at 20:10

community wiki
2 revs
DNA

5

Google's (bad) approach, from HN thread. Store RLE-style counts.

> Your initial data structure is '99999999:0' (all zeros, haven't seen any numbers) and then lets say you see the number 3,866,344 so your data structure becomes '3866343:0,1:1,96133654:0' as you can see the numbers will always alternate between number of zero bits and number of '1' bits so you can just assume the odd numbers represent 0 bits and the even numbers 1 bits. This becomes (3866343,1,96133654)

Their problem doesn't seem to cover duplicates, but let's say they use "0:1" for duplicates.

Big problem #1: insertions for 1M integers *would take ages*.

Big problem #2: like all plain delta encoding solutions, some distributions can't be covered this way. For example, 1m integers with distances 0:99 (e.g. +99 each one). Now think the same but with **random distance** in the **range of 0:99**. (Note: 99999999/1000000 = 99.99)

Google's approach is both unworthy (slow) and incorrect. But to their defense, their problem might have been slightly different.

Share Improve this answer Follow

edited Oct 21, 2012 at 22:34

community wiki

2 revs
alecco

5

I think the solution is to combine techniques from video encoding, namely the discrete cosine transformation. In digital video, rather recording the changing the brightness or colour of video as regular values such as 110 112 115 116, each is subtracted from the last (similar to run length encoding). 110 112 115 116 becomes 110 2 3 1. The values, 2 3 1 require less bits than the originals.

So lets say we create a list of the input values as they arrive on the socket. We are storing in each element, not the value, but the offset of the one before it. We sort as we go, so the offsets are only going to be positive. But the offset could be 8 decimal digits wide which this fits in 3 bytes. Each element can't be 3 bytes, so we need to pack these. We could use the top bit of each byte as a "continue bit", indicating that the next byte is part of the number and the lower 7 bits of each byte need to be combined. zero is valid for duplicates.

As the list fills up, the numbers should be get closer together, meaning on average only 1 byte is used to determine the distance to the next value. 7 bits of value and 1 bit of offset if convenient, but there may be a sweet spot that requires less than 8 bits for a "continue" value.

Anyway, I did some experiment. I use a random number generator and I can fit a million sorted 8 digit decimal numbers into about 1279000 bytes. The average space between each number is consistently 99...

```
public class Test {
    public static void main(String[] args) throws IOEx
        // 1 million values
        int[] values = new int[1000000];

        // create random values up to 8 digits lrong
        Random random = new Random();
        for (int x=0;x<values.length;x++) {
            values[x] = random.nextInt(100000000);
        }
        Arrays.sort(values);

        ByteArrayOutputStream baos = new ByteArrayOutp

        int av = 0;
        writeCompact(baos, values[0]);     // first va
        for (int x=1;x<values.length;x++) {
            int v = values[x] - values[x-1];  // diffe
            av += v;
            System.out.println(values[x] + " diff " +
            writeCompact(baos, v);
        }

        System.out.println("Average offset " + (av/val
        System.out.println("Fits in " + baos.toByteArr
    }

    public static void writeCompact(OutputStream os, l
IOException {
        do {
            int b = (int) value & 0x7f;
            value = (value & 0x7fffffffffffffffl) >> 7
            os.write(value == 0 ? b : (b | 0x80));
        } while (value != 0);
    }
}
```

Share Improve this answer   edited Oct 25, 2020 at 6:53

Follow

community wiki
6 revs, 2 users 97%
slipperyseal

4

We could play with the networking stack to send the numbers in sorted order before we have all the numbers. If you send 1M of data, TCP/IP will break it into 1500 byte packets and stream them in order to the target. Each packet will be given a sequence number.

We can do this by hand. Just before we fill our RAM we can sort what we have and send the list to our target but leave holes in our sequence around each number. Then process the 2nd 1/2 of the numbers the same way using those holes in the sequence.

The networking stack on the far end will assemble the resulting data stream in order of sequence before handing it up to the application.

It's using the network to perform a merge sort. This is a total hack, but I was inspired by the other networking hack listed previously.

Share Improve this answer Follow

answered Oct 21, 2012 at 21:27

community wiki
Kevin Marquette

4

I would exploit the retransmission behaviour of TCP.

1. Make the TCP component create a large receive window.
2. Receive some amount of packets without sending an ACK for them.
   - Process those in passes creating some (prefix) compressed data structure
   - Send duplicate ack for last packet that is not needed anymore/wait for retransmission timeout
   - Goto 2
3. All packets were accepted

This assumes some kind of benefit of buckets or multiple passes.

Probably by sorting the batches/buckets and merging them. -> radix trees

Use this technique to accept and sort the first 80% then read the last 20%, verify that the last 20% do not contain numbers that would land in the first 20% of the lowest numbers. Then send the 20% lowest numbers, remove from memory, accept the remaining 20% of new numbers and merge.**

Share Improve this answer Follow

edited Oct 11, 2015 at 15:14

community wiki
2 revs, 2 users 74%
sleeplessnerd

3

To represent the sorted array one can just store the first element and the difference between adjacent elements. In this way we are concerned with encoding 10^6 elements that can sum up to at most 10^8. Let's call this **D**. To encode the elements of **D** one can use a Huffman code. The dictionary for the Huffman code can be created on the go and the array updated every time a new item is inserted in the sorted array (insertion sort). Note that when the dictionary changes because of a new item the whole array should be updated to match the new encoding.

The average number of bits for encoding each element of **D** is maximized if we have equal number of each unique element. Say elements *d1*, *d2*, ..., *dN* in **D** each appear *F* times. In that case (in worst case we have both 0 and 10^8 in input sequence) we have

sum(1<=*i*<=*N*) *F*. *di* = 10^8

where

sum(1<=*i*<=*N*) *F* = 10^6, or *F*=10^6/*N* and the normalized frequency will be *p*= *F*/10^=1/*N*

The average number of bits will be -log2(1/*P*) = log2(*N*). Under these circumstances we should find a case that maximizes *N*. This happens if we have consecutive numbers for *di* starting from 0, or, *di*= *i*-1, therefore

10^8=**sum(1<=*i*<=*N*)** *F*. *di* = **sum(1<=*i*<=*N*)** (10^6/*N*) (i-1) = (10^6/*N*) *N* (*N*-1)/2

i.e.

N <= 201. And for this case average number of bits is log2(201)=7.6511 which means we will need around 1 byte per input element for saving the sorted array. Note that this doesn't mean **D** in general cannot have more than 201 elements. It just sows that if elements of **D** are uniformly distributed, it cannot have more than 201 unique values.

Share Improve this answer Follow

answered Oct 22, 2012 at 1:59

community wiki
Mohsen Nosratinia

1 I think you have forgotten that number can be duplicate.
– bestsss Oct 23, 2012 at 9:32

For duplicate numbers the difference between adjacent numbers will be zero. Doesn't create any problem. Huffman code doesn't require nonzero values. – Mohsen Nosratinia Oct 30, 2012 at 19:30

3

Here is a generalized solution to this kind of problem:

General procedure

The taken approach is as follows. The algorithm operates on a single buffer of 32-bit words. It performs the following procedure in a loop:

- We start with a buffer filled with compressed data from the last iteration. The buffer looks like this

  ```
  |compressed sorted|empty|
  ```

- Calculate the maximum amount of numbers that can be stored in this buffer, both compressed and uncompressed. Split the buffer into these two sections, beginning with the space for compressed data, ending with the uncompressed data. The buffer looks like

  ```
  |compressed sorted|empty|empty|
  ```

- Fill the uncompressed section with numbers to be sorted. The buffer looks like

  ```
  |compressed sorted|empty|uncompressed unsorted|
  ```

- Sort the new numbers with an in-place sort. The buffer looks like

    `|compressed sorted|empty|uncompressed sorted|`

- Right-align any already compressed data from the previous iteration in the compressed section. At this point the buffer is partitioned

  `|empty|compressed sorted|uncompressed sorted|`

- Perform a streaming decompression-recompression on the compressed section, merging in the sorted data in the uncompressed section. The old compressed section is consumed as the new compressed section grows. The buffer looks like

  `|compressed sorted|empty|`

This procedure is performed until all numbers have been sorted.

Compression

This algorithm of course only works when it's possible to calculate the final compressed size of the new sorting buffer before actually knowing what will actually be compressed. Next to that, the compression algorithm needs to be good enough to solve the actual problem.

The used approach uses three steps. First, the algorithm will always store sorted sequences, therefore we can instead store purely the differences between consecutive entries. Each difference is in the range [0, 99999999].

These differences are then encoded as a unary bitstream. A 1 in this stream means "Add 1 to the accumulator, A 0 means "Emit the accumulator as an entry, and reset". So difference N will be represented by N 1's and one 0.

The sum of all differences will approach the maximum value that the algorithm supports, and the count of all differences will approach the amount of values inserted in the algorithm. This means we expect the stream to, at the end, contain max value 1's and count 0's. This allows us to calculate the expected probability of a 0 and 1 in the stream. Namely, the probability of a 0 is `count/(count+maxval)` and the probability of a 1 is `maxval/(count+maxval)`.

We use these probabilities to define an arithmetic coding model over this bitstream. This arithmetic code will encode exactly this amounts of 1's and 0's in optimal space. We can calculate the space used by this model for any intermediate bitstream as: `bits = encoded * log2(1 + amount / maxval) + maxval * log2(1 + maxval / amount)`. To calculate the total required space for the algorithm, set `encoded` equal to amount.

To not require a ridiculous amount of iterations, a small overhead can be added to the buffer. This will ensure that the algorithm will at least operate on the amount of numbers that fit in this overhead, as by far the largest time cost of the algorithm is the arithmetic coding compression and decompression each cycle.

Next to that, some overhead is necessary to store bookkeeping data and to handle slight inaccuracies in the fixed-point approximation of the arithmetic coding algorithm, but in total the algorithm is able to fit in 1MiB of space even with an extra buffer that can contain 8000 numbers, for a total of 1043916 bytes of space.

Optimality

Outside of reducing the (small) overhead of the algorithm it should be theoretically impossible to get a smaller result. To just contain the entropy of the final result, 1011717 bytes would be necessary. If we subtract the extra buffer added for efficiency this algorithm used 1011916 bytes to store the final result + overhead.

Share Improve this answer Follow

edited Apr 1, 2020 at 22:14

community wiki
3 revs, 2 users 87%
CensoredUsername

2

If the input stream could be received few times this would be much easier (no information about that, idea and time-performance problem).

Then, we could count the decimal values. With counted values it would be easy to make the output stream.

Compress by counting the values. It depends what would be in the input stream.

Share Improve this answer Follow

edited Oct 11, 2015 at 15:22

community wiki
2 revs, 2 users 67%
Baronth

1

If the input stream could be received few times this would be much easier (no info about that, idea and time-performance problem). Then, we could count the decimal values. With counted values it would be easy to make the output stream. Compress by counting the values. It depends what would be in the input stream.

Share Improve this answer Follow

answered Oct 20, 2012 at 22:33

community wiki
pbies

1

Sorting is a secondary problem here. As other said, just storing the integers is hard, and **cannot work on all inputs**, since 27 bits per number would be necessary.

My take on this is: store only the differences between the consecutive (sorted) integers, as they will be most likely small. Then use a compression scheme, e.g. with 2 additional bits per input number, to encode how many bits the number is stored on. Something like:

```
00 -> 5 bits
01 -> 11 bits
10 -> 19 bits
11 -> 27 bits
```

It should be possible to store a fair number of possible input lists within the given memory constraint. The maths of how to pick the compression scheme to have it work on the maximum number of inputs, are beyond me.

I hope you may be able to exploit **domain-specific knowledge of your input** to find a good enough **integer compression scheme** based on this.

Oh and then, you do an insertion sort on that sorted list as you receive data.

Share Improve this answer Follow

edited Oct 21, 2012 at 21:57

community wiki
2 revs
Eldritch Conundrum

1

Now aiming to an actual solution, covering all possible cases of input in the 8 digit range with only 1MB of RAM. NOTE: work in progress, tomorrow will continue. Using arithmetic coding of deltas of the sorted ints, worst case for 1M sorted ints would cost about 7bits per entry (since 99999999/1000000 is 99, and log2(99) is almost 7 bits).

But you need the 1m integers sorted to get to 7 or 8 bits! Shorter series would have bigger deltas, therefore more bits per element.

I'm working on taking as many as possible and compressing (almost) in-place. First batch of close to 250K ints would need about 9 bits each at best. So result would take about 275KB. Repeat with remaining free memory a few times. Then decompress-merge-in-place-compress those compressed chunks. This is **quite hard**, but possible. I think.

The merged lists would get closer and closer to the 7bit per integer target. But I don't know how many iterations it would take of the merge loop. Perhaps 3.

But the imprecision of the arithmetic coding implementation might make it impossible. If this problem is possible at all, it would be extremely tight.

Any volunteers?

answered Oct 21, 2012 at 23:12

community wiki
alecco

Arithmetic coding is workable. It might help to notice that each successive delta is drawn from a negative binomial distribution. – crowding Oct 23, 2012 at 0:36

1

You just need to store the differences between the numbers in sequence, and use an encoding to compress these sequence numbers. We have 2^23 bits. We shall divide it into 6bit chunks, and let the last bit indicate whether the number extends to another 6 bits (5bits plus extending chunk).

Thus, 000010 is 1, and 000100 is 2. 000001100000 is 128. Now, we consider the worst cast in representing differences in sequence of a numbers up to 10,000,000. There can be 10,000,000/2^5 differences greater than 2^5, 10,000,000/2^10 differences greater than 2^10, and 10,000,000/2^15 differences greater than 2^15, etc.

So, we add how many bits it will take to represent our the sequence. We have 1,000,000*6 + roundup(10,000,000/2^5)*6+roundup(10,000,000/2^10)*6 +roundup(10,000,000/2^15)*6+roundup(10,000,000/2^20)*4=7935479.

2^24 = 8388608. Since 8388608 > 7935479, we should easily have enough memory. We will probably need another little bit of memory to store the sum of where are when we insert new numbers. We then go through the sequence, and find where to insert our new number, decrease the next difference if necessary, and shift everything after it right.

Share Improve this answer Follow

answered Oct 22, 2012 at 4:50

community wiki
gersh

I *believe* my analysis here shows that this scheme doesn't work (and can't even if we choose another size than five bits). – Daniel Wagner Oct 22, 2012 at 6:14

@Daniel Wagner -- You don't have to use the a uniform number of bits per chunk, and you don't even have to use an integer number of bits per chunk. – crowding Oct 23, 2012 at 1:06

@crowding If you have a concrete proposal, I'd like to hear it. =) – Daniel Wagner Oct 23, 2012 at 4:14

@crowding Do the math on how much space arithmetic coding would take. Cry a bit. Then think harder. – Daniel Wagner Oct 23, 2012 at 7:20

Learn more. A complete conditional distribution of symbols in the right intermediate representation (Francisco has the simplest intermediate representation, as does Strilanc) is easy to compute. Thus the encoding model can be literally perfect and can come within one bit of the entropic limit. Finite precision arithmetic might add a few bits. – crowding Oct 23, 2012 at 7:53

If we don't know anything about those numbers, we are limited by the following constraints:

1

- we need to load all numbers before we can sort them them,
- the set of numbers is not compressible.

If these assumptions hold, there is no way to carry out your task, as you will need at least 26,575,425 bits of storage (3,321,929 bytes).

What can you tell us about your data ?

Share Improve this answer Follow

answered Oct 22, 2012 at 9:30

community wiki
user1196549

1 You read them in and sort them as you go. It theoretically requires lg2(100999999!/(99999999! * 1000000!)) bits to store 1M indistinguishable items in 100M distinguished boxes, which works out to 96.4% of 1MB. – NovaDenizen Oct 22, 2012 at 17:20

1

The trick is to represent the algorithms state, which is an integer multi-set, as a compressed stream of "increment counter"="+" and "output counter"="!" characters. For example, the set {0,3,3,4} would be represented as "!+++!!+!", followed by any number of "+" characters. To modify the multi-set you stream out the characters, keeping only a constant amount decompressed at a time, and make changes inplace before streaming them back in compressed form.

Details

We know there are exactly 10^6 numbers in the final set, so there are at most 10^6 "!" characters. We also know that our range has size 10^8, meaning there are at most 10^8 "+" characters. The number of ways we can arrange 10^6 "!"s amongst 10^8 "+"s is `(10^8 + 10^6) choose 10^6`, and so specifying some particular arrangement takes ~0.965 MiB` of data. That'll be a tight fit.

We can treat each character as independent without exceeding our quota. There are exactly 100 times more "+" characters than "!" characters, which simplifies to 100:1 odds of each character being a "+" if we forget that they are dependent. Odds of 100:101 corresponds to ~0.08 bits per character, for an almost identical total of ~0.965 MiB (ignoring the dependency has a cost of only ~12 bits in this case!).

The simplest technique for storing independent characters with known prior probability is Huffman coding. Note that we need an impractically large tree (A huffman tree for blocks of 10 characters has an average cost per block of about 2.4 bits, for a total of ~2.9 Mib. A huffman tree for blocks of 20 characters has an average cost per block of about 3 bits, which is a total of ~1.8 MiB. We're probably going to need a block of size on the order of a hundred, implying more nodes in our tree than all the computer equipment that has ever existed can store.).

However, ROM is technically "free" according to the problem and practical solutions that take advantage of the regularity in the tree will look essentially the same.

Pseudo-code

- Have a sufficiently large huffman tree (or similar block-by-block compression data) stored in ROM
- Start with a compressed string of 10^8 "+" characters.
- To insert the number N, stream out the compressed string until N "+" characters have gone past then insert a "!". Stream the recompressed string back over the previous one as you go, keeping a constant amount of buffered blocks to avoid over/under-runs.
- Repeat one million times: [input, stream decompress>insert>compress], then decompress to output

Share Improve this answer Follow

edited Oct 24, 2012 at 4:43

community wiki
2 revs
Strilanc

1 So far, this is the only answer I see that actually answers the problem! I think arithmetic coding is a simpler fit than Huffman coding though, as it obviates the needs to store a codebook and worry about symbol boundaries. You can

account for the dependency too. – crowding Oct 23, 2012 at 0:51

The input integers are NOT sorted. You need to sort first. – alecco Oct 24, 2012 at 2:13

1 @alecco The algorithm sorts them as it progresses. They're never stored unsorted. – Craig Gidney Oct 24, 2012 at 4:43

1

We have 1 MB - 3 KB RAM = 2^23 - 3*2^13 bits = 8388608 - 24576 = 8364032 bits available.

We are given 10^6 numbers in a 10^8 range. This gives an average gap of ~100 < 2^7 = 128

Let's first consider the simpler problem of fairly evenly spaced numbers when all gaps are < 128. This is easy. Just store the first number and the 7-bit gaps:

(27 bits) + 10^6 7-bit gap numbers = 7000027 bits required

Note repeated numbers have gaps of 0.

But what if we have gaps larger than 127?

OK, let's say a gap size < 127 is represented directly, but a gap size of 127 is followed by a continuous 8-bit encoding for the actual gap length:

```
 10xxxxxx xxxxxxxx                       = 127 ..
16,383
```

```
 110xxxxx xxxxxxxx xxxxxxxx              = 16384
.. 2,097,151
```

etc.

Note this number representation describes its own length so we know when the next gap number starts.

With just small gaps < 127, this still requires 7000027 bits.

There can be up to (10^8)/(2^7) = 781250 23-bit gap number, requiring an extra 16*781,250 = 12,500,000 bits which is too much. We need a more compact and slowly increasing representation of gaps.

The average gap size is 100 so if we reorder them as [100, 99, 101, 98, 102, ..., 2, 198, 1, 199, 0, 200, 201, 202, ...] and index this with a dense binary Fibonacci base encoding with no pairs of zeros (for example, 11011=8+5+2+1=16) with numbers delimited by '00' then I think we can keep the gap representation short enough, but it needs more analysis.

Share Improve this answer Follow

edited Oct 11, 2015 at 15:20

community wiki
2 revs, 2 users 85%
Toby Kelsey

0

While receiving the stream do these steps.

1st set some reasonable chunk size

Pseudo Code idea:

1. The first step would be to find all the duplicates and stick them in a dictionary with its count and remove them.
2. The third step would be to place number that exist in sequence of their algorithmic steps and place them in counters special dictionaries with the first number and their step like n, n+1..., n+2, 2n, 2n+1, 2n+2...
3. Begin to compress in chunks some reasonable ranges of number like every 1000 or ever 10000 the remaining numbers that appear less often to repeat.
4. Uncompress that range if a number is found and add it to the range and leave it uncompressed for a while longer.
5. Otherwise just add that number to a byte[chunkSize]

Continue the first 4 steps while receiving the stream. The final step would be to either fail if you exceeded memory or start outputting the result once all the data is collected by beginning to sort the ranges and spit out the results in order and uncompressing those in order that need to be uncompressed and sort them when you get to them.

Share Improve this answer   edited Oct 25, 2020 at 6:54

Follow

community wiki
2 revs, 2 users 76%
RetroCoder

1 2 Next

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.