Efficiently get sorted sums of a sorted list

Asked 16 years, 4 months ago Modified 4 years, 4 months ago

Viewed 5k times

20

You have an ascending list of numbers, what is the most efficient algorithm you can think of to get the ascending list of sums of every two numbers in that list. Duplicates in the resulting list are irrelevant, you can remove them or avoid them if you like.

To be clear, I'm interested in the algorithm. Feel free to post code in any language and paradigm that you like.

algorithm language-agnostic

Share

Improve this question

Follow

edited Aug 3, 2008 at 21:38

asked Aug 3, 2008 at 21:08

Peter Burns

45.3k ● 7 ● 40 ● 56

8 Answers

Sorted by: Highest score (default)

13

Edit as of 2018: You should probably stop reading this. (But I can't delete it as it is accepted.)

If you write out the sums like this:

```
1 4  5  6  8  9
---------------
2 5  6  7  9 10
  8  9 10 12 13
    10 11 13 14
       12 14 15
          16 17
             18
```

You'll notice that since M[i,j] <= M[i,j+1] and M[i,j] <= M[i+1,j], then you only need to examine the top left "corners" and choose the lowest one.

e.g.

- only 1 top left corner, pick 2
- only 1, pick 5
- 6 or 8, pick 6
- 7 or 8, pick 7
- 9 or 8, pick 8
- 9 or 9, pick both :)
- 10 or 10 or 10, pick all
- 12 or 11, pick 11
- 12 or 12, pick both

- 13 or 13, pick both
- 14 or 14, pick both
- 15 or 16, pick 15
- only 1, pick 16
- only 1, pick 17
- only 1, pick 18

Of course, when you have *lots* of top left corners then this solution devolves.

I'm pretty sure this problem is $\Omega(n^2)$, because you have to calculate the sums for each M[i,j] -- unless someone has a better algorithm for the summation :)

Share Improve this answer Follow

edited Feb 8, 2018 at 21:40

answered Sep 18, 2008 at 21:41

porges

30.5k ● 4 ● 87 ● 115

1 I think this is O(n^3) since there are n potential 'top left corners' at each stage. – user97370 Feb 5, 2010 at 2:56

2 You can implement this algorithm in O(n^2 log n) time by storing the first unpicked entry in each row in a priority queue, but asymptotically this is no better than just generating all sums and sorting. – Reid Barton Oct 3, 2010 at 19:08

If you have two lists instead of one, of length m and n with m<n, then the k-way merge-based approach gives O(mn log m) rather than O(mn log(mn)). Furthermore, if m is fairly

small (up to a few hundred elements, I would guess), the merge-based approach will make far better use of processor cache (because the priority queue or merge tree will fit in L1 cache), which could easily improve the speed by a factor of 100 or more, and if it's written very carefully can make good use of the processor pipeline, in which case it should really scream. – dfeuer Aug 25, 2012 at 23:10

1 On reflection 4 years later, this seems like a really bad version of merge sort :) – porges Sep 6, 2012 at 21:53

4

Rather than coding this out, I figure I'll pseudo-code it in steps and explain my logic, so that better programmers can poke holes in my logic if necessary.

On the first step we start out with a list of numbers length n. For each number we need to create a list of length n-1 becuase we aren't adding a number to itself. By the end we have a list of about n sorted lists that was generated in O(n^2) time.

```
step 1 (startinglist)
for each number num1 in startinglist
   for each number num2 in startinglist
      add num1 plus num2 into templist
   add templist to sumlist
return sumlist
```

In step 2 because the lists were sorted by design (add a number to each element in a sorted list and the list will still be sorted) we can simply do a mergesort by merging each list together rather than mergesorting the whole lot. In the end this should take O(n^2) time.

```
step 2 (sumlist)
create an empty list mergedlist
for each list templist in sumlist
   set mergelist equal to:
merge(mergedlist,templist)
return mergedlist
```

The merge method would be then the normal merge step with a check to make sure that there are no duplicate sums. I won't write this out because anyone can look up mergesort.

So there's my solution. The entire algorithm is O(n^2) time. Feel free to point out any mistakes or improvements.

Share Improve this answer

Follow

edited Apr 17, 2015 at 10:17

Stefan Steiger

81.9k ● 69 ● 399 ● 454

answered Aug 3, 2008 at 23:06

Holtorf

1,491 ● 4 ● 22 ● 42

I think this is O(N^3) since there are n comparisons at each stage in step 2. – user97370 Feb 5, 2010 at 2:56

2

You can do this in two lines in python with

```
allSums = set(a+b for a in X for b in X)
allSums = sorted(allSums)
```

The cost of this is `n^2` (maybe an extra log factor for the set?) for the iteration and s * log(s) for the sorting where s is the size of the set.

The size of the set could be as big as `n*(n-1)/2` for example if `X = [1,2,4,...,2^n]`. So if you want to generate this list it will take at least `n^2/2` in the worst case since this is the size of the output.

However if you want to select the first k elements of the result you can do this in O(kn) using a selection algorithm for sorted `X+Y` matrices by Frederickson and Johnson (see here for gory details). Although this can probably be modified to generate them online by reusing computation and get an efficient generator for this set.

@deuseldorf, Peter There is some confusion about `(n!)` I seriously doubt deuseldorf meant "n factorial" but simply "n, (very excited)!"

Share Improve this answer

Follow

edited Jul 27, 2020 at 1:15

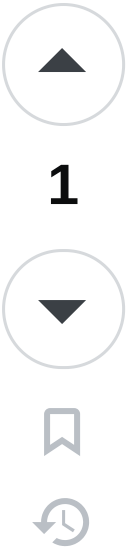
linkonabe

846 • 9 • 26

answered Aug 11, 2008 at 14:47

Pall Melsted

This has better complexity than all of the other solutions I think! O(n^2.log(n)). It's also the most readable and the shortest. – user97370 Feb 5, 2010 at 3:00



The best I could come up with is to produce a matrix of sums of each pair, and then merge the rows together, a-la merge sort. I feel like I'm missing some simple insight that will reveal a much more efficient solution.

My algorithm, in Haskell:

```
matrixOfSums list = [[a+b | b <- list, b >= a] | a
<- list]

sortedSums = foldl merge [] matrixOfSums

--A normal merge, save that we remove duplicates
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) = case compare x y of
    LT -> x:(merge xs (y:ys))
    EQ -> x:(merge xs (dropWhile (==x) ys))
    GT -> y:(merge (x:xs) ys)
```

I found a minor improvement, one that's more amenable to lazy stream-based coding. Instead of merging the columns pair-wise, merge all of them at once. The advantage being that you start getting elements of the list immediately.

```
-- wide-merge does a standard merge (ala merge-
sort) across an arbitrary number of lists
-- wideNubMerge does this while eliminating
duplicates
wideNubMerge :: Ord a => [[a]] -> [a]
wideNubMerge ls = wideNubMerge1 $ filter (/= [])
ls
wideNubMerge1 [] = []
wideNubMerge1 ls = mini:(wideNubMerge rest)
    where mini = minimum $ map head ls
```

```
            rest = map (dropWhile (== mini)) ls

betterSortedSums = wideNubMerge matrixOfSums
```

However, if you know you're going to use all of the sums, and there's no advantage to getting some of them earlier, go with '`foldl merge []`', as it's faster.

Share Improve this answer Follow

edited Aug 6, 2008 at 18:56

answered Aug 3, 2008 at 21:36

Peter Burns

45.3k ● 7 ● 40 ● 56

I think (my haskell is rusty) this is O(N^3) since there are O(n) comparisons done for each thing in the result.
– user97370 Feb 5, 2010 at 2:55

1

In SQL:

```
create table numbers(n int not null)
insert into numbers(n) values(1),(1), (2), (2),
(3), (4)


select distinct num1.n+num2.n sum2n
from numbers num1
inner join numbers num2
    on num1.n<>num2.n
order by sum2n
```

C# LINQ:

```
List<int> num = new List<int>{ 1, 1, 2, 2, 3, 4};
var uNum = num.Distinct().ToList();
var sums=(from num1 in uNum
        from num2 in uNum
        where num1!=num2
        select num1+num2).Distinct();
foreach (var s in sums)
{
    Console.WriteLine(s);
}
```

Share Improve this answer

Follow

edited Aug 8, 2008 at 23:31

answered Aug 8, 2008 at 23:05

vzczc

9,380 ● 5 ● 55 ● 62

1

No matter what you do, without additional constraints on the input values, you cannot do better than O(n^2), simply because you have to iterate through all pairs of numbers. The iteration will dominate sorting (which you can do in O(n log n) or faster).

Share Improve this answer

Follow

answered Sep 18, 2008 at 22:15

florin

14.3k ● 6 ● 49 ● 47

1 Yes, but to sort n^2 things takes O(n^2 log n) so sorting never dominates. – user97370 Feb 5, 2010 at 3:01

This question has been wracking my brain for about a day now. Awesome.

Anyways, you can't get away from the n^2 nature of it easily, but you can do slightly better with the merge since you can bound the range to insert each element in.

If you look at all the lists you generate, they have the following form:

```
(a[i], a[j]) | j>=i
```

If you flip it 90 degrees, you get:

```
(a[i], a[j]) | i<=j
```

Now, the merge process should be taking two lists `i` and `i+1` (which correspond to lists where the first member is always `a[i]` and `a[i+1]`), you can bound the range to insert element `(a[i + 1], a[j])` into list `i` by the location of `(a[i], a[j])` and the location of `(a[i + 1], a[j + 1])`.

This means that you should merge in reverse in terms of `j`. I don't know (yet) if you can leverage this across `j` as well, but it seems possible.

edited Jul 3, 2012 at 14:49

user142162

answered Aug 21, 2008 at 18:16

Mat Noguchi
1,080 • 6 • 7

-4

If you are looking for a truly language agnostic solution then you will be sorely disappointed in my opinion because you'll be stuck with a for loop and some conditionals. However if you opened it up to functional languages or functional language features (I'm looking at you LINQ) then my colleagues here can fill this page with elegant examples in Ruby, Lisp, Erlang, and others.

Share Improve this answer

Follow

answered Aug 3, 2008 at 21:24

John Downey
14.1k • 5 • 38 • 33