Searching a Sorted Matrix Faster

posted by Craig Gidney on August 13, 2013

In this post: Craig solves a problem that's already been solved, except most people get the solution wrong or claim non-optimal solutions are optimal. Hopefully he makes up for any retreading by providing visualizations and working code and explaining real good.

The Problem

Three years ago, I came across an interview puzzle question on stack overflow: create an efficient algorithm that finds the position of a target item within a matrix where each row and column happens to be sorted.

For example:

$$M = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 5 & 7 \\ 1 & 4 & 9 & 16 & 25 \end{pmatrix}$$

$find(M, 0) = None$
$find(M, 1) = (0, 0)$ or alternatively $(0, 1)$ or $(0, 2)$
$find(M, 9) = (2, 2)$

At that time, and at the time of this writing, there were three notable answers to the question. Each is representative of a class of commonly claimed solutions, which you'll find in many places if you search around.

I remember wondering at the time if the proposed solutions were optimal. Recently, I revisited and realized: nope.

Non-Optimal Solution #1: Line at a Time

The first notable solution is elegantly simple: always query in the bottom left of the not-eliminated area, eliminating one row or column at a time. It takes $O(w + h)$ time, where h is the height of the matrix and w is the width of the matrix. This is fine when the matrix is approximately square, but not optimal when the matrix is much wider than it is tall, or vice versa.

For example, consider what happens when $h = 1$: we just have a normal sorted list, and can apply binary search to solve the problem in $O(lg(w))$ time, but line-at-a-time will take $O(w)$ time. Line-at-a-time is not optimal when the matrix is very tall or very wide.

Non-Optimal Solution #2: Divide and Conquer

The second notable solution is a divide-and-conquer approach. It queries the center of a rectangular area to be searched, eliminating either the upper left or bottom right quadrant, then divides the remaining valid area into two rectangles, and recurses on those rectangles.

This answer has an interesting recurrence relation for its running time, assuming the asymmetry of the search areas doesn't affect the running time: $T(a) = 1 + T(a/2) + T(a/4)$. To me this looks like a nice efficient relation, but when you actually solve it (no, you can't apply the master theorem) you see that it's not.

The recurrence relation solves to $T(a) = O\left(Fibonacci(lg_2(a))\right) = O\left(\phi^{lg_2(a)}\right) = O(a^{\frac{lg(\phi)}{lg(2)}}) \approx a^{0.7}$. So the divide and conquer algorithm takes $O\left((w \cdot h)^{0.7}\right)$ time, which is $O\left((w+h)^{1.4}\right)$ when $w \approx h$, which is more than line-at-a-time's $O(w+h)$.

Note: A variant of this algorithm divides the remaining area into three rectangles. It also requires more-than-linear time, although its recurrence relation is easier to solve.

Update: A more careful analysis shows that this approach is actually optimal. Whoops. See the comments below.

Non-Optimal Solution #3: Multi Binary Search

The final notable solution is to do a binary search of each row or each column, depending on which is longer.

This approach takes time $O(b \cdot lg(b \cdot t))$, where $b = min(w, h)$ and $t = \frac{max(w,h)}{b}$. This is not optimal when $h \approx w$, (that is to say, $t \approx 1$). In that case the running time reduces to $O((w+h) \cdot lg(w+h))$, which is less efficient than line-at-a-time.

(Actually, it turns out that this algorithm is not optimal even when $w >> h$ or $h >> w$. That's too bad, since otherwise an optimal solution would be to just switch between multi-binary-search and line-at-a-time based on how tall/wide the matrix was.)

None are Optimal

The takeaway so far is: everyone keeps giving non-optimal answers to this question. The answers I linked to don't do claim or imply their answers are optimal, but if you search around you can find plenty of equivalent ones that do.

Clearly this problem is a bit more complicated than it appears.

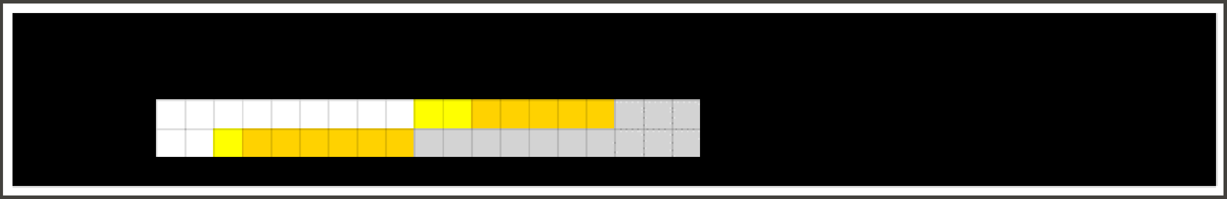
To settle the issue, I'm going to solve the problem and I'm going to prove my solution is asymptotically optimal. I'll do this in the usual way: prove some lower bound on the amount of required time, then find an algorithm that achieves that time bound.

Lower Bounding: Adverse Diagonal

Imagine you are an adversary who controls the entries of the matrix as it is being searched. You can delay choosing the value of a cell until the algorithm actually queries it. You're not allowed to violate the sorted-ness of rows or columns, but you can make sure the correct result is in the last place the algorithm looks. How many queries can you force an arbitrary algorithm to do?

When imagining this scenario, it's useful to forget about the fact that the matrix contains numbers. Instead, imagine the algorithm picking not-yet-eliminated positions, and the adversary deciding if the upper left or lower right quadrant defined by that position gets eliminated. How long can the adversary keep the game going? How quickly can the algorithm eliminate the entire grid?

One trick the adversary can use, to keep the game going for awhile, is to totally ignore everything except the main diagonal. Here's a diagram of an adversary forcing an algorithm to search on the diagonal:



Legend: white cells are smaller items, gray cells are larger items, yellow cells are smaller-or-equal items and orange cells are larger-or-equal items. The adversary forces the solution to be whichever yellow or orange cell the algorithm queries last.

The above diagram shows a main diagonal made up of b contiguous sections of length t. Remember that $b = min(w, h)$ and $t = \frac{max(w,h)}{b}$. When the algorithm queries off of the diagonal it makes no progress, because either the upper-left or lower-right quadrant won't include the diagonal and the adversary will pick that quadrant as the one to eliminate. Also, when the algorithm queries inside one section, it makes no progress on the other sections. Thus the algorithm is forced to make $\Omega(lg(t))$ queries to eliminate each of the b sections (with a binary search).

This gives us a lower bound on the worst case running time: $\Omega(b \cdot lg(t))$. In terms of the width and height, that's $\Omega\left(min(w,h) \cdot lg\left(\frac{max(w,h)}{min(w,h)}\right)\right)$. An adversary can force any algorithm to perform that many queries, using only the main diagonal.

Notice that when $w = h$, this lower bound reduces to $\Omega(w \cdot lg(1)) = \Omega(w) = \Omega(w + h)$ (note: we're using the magical computer science logarithm that's clamped to a minimum result of one), which is tight against the running time of the line-at-a-time algorithm. That's a good sign.

Also notice that, when $w >> h$ or $h >> w$, this lower bound is below the complexity achieved by multi-binary-search. Multi-binary-search takes $O(b \cdot lg(b \cdot t))$ time, which is not tight against $\Omega(b \cdot lg(t))$ time.

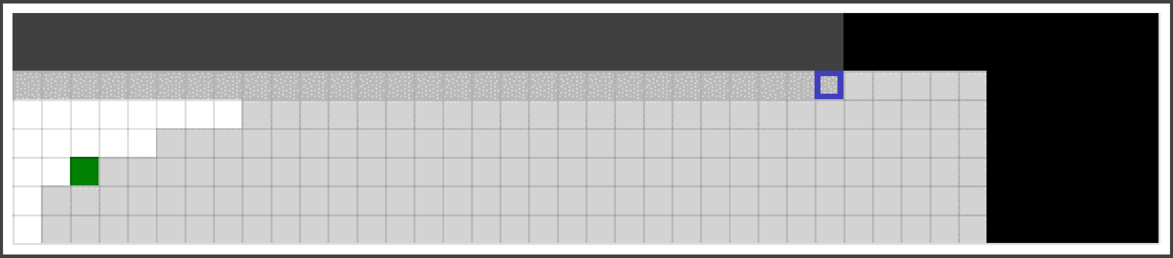
Now the question is: can we find an algorithm that meets this lower bound?

Upper Bounding: Algorithm

After far too much thought, I discovered the trick to making an algorithm that meets the lower bound: you have to use parts of both line-at-a-time and multi-binary-search.

The optimal algorithm works almost like line-at-a-time, except it advances in steps of size t instead of size 1 when moving along the long axis. This allows the algorithm to either immediately eliminate t short lines, or to spend $O(lg(t))$ more operations eliminating the rest of a long line with a binary search. Since there's only b long lines and $b \cdot t$ short lines, the algorithm has running time $O(b \cdot lg(t) + \frac{b \cdot t}{t}) = O(b \cdot lg(t))$. Hey, that's tight against our lower bound!

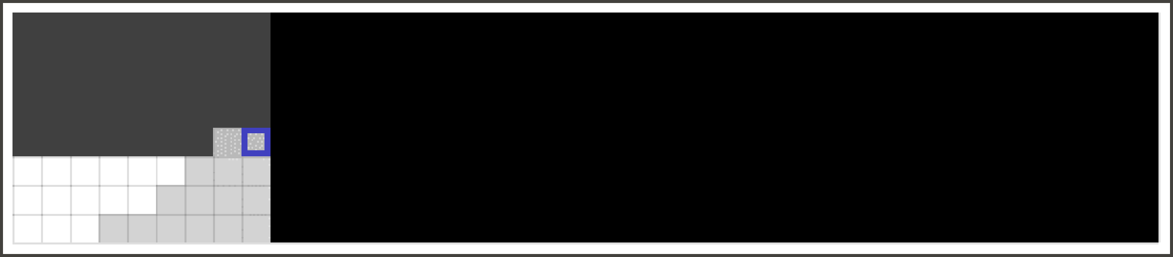
Here's an animation of the algorithm finding an item:



Legend: white cells are smaller items, gray cells are larger items, and the green cell is an equal item.

In the above diagram, where $w = 40$ and $h = 8$ meaning $b = 8$ and $t = 5$, you can see how the algorithm tracks the boundary of the search space. It keeps querying 5 cells to the left of the top right of the current valid area. When the top right is well inside the gray area, past the boundary between smaller and larger items, this eliminates $5 + 1$ columns at a time. When the top right is well inside the white area, most of a row is eliminated instead. Three or four more queries are needed to completely finish off the row, via a binary search. When straddling the boundary, a combination of these two things happens.

Here's another animation, where the algorithm determines no matching item is present:



Since this algorithm takes $O(b \cdot lg(t))$ time, and we have a lower bound of $\Omega(b \cdot lg(t))$, searching a sorted matrix is in $\Theta(b \cdot lg(t))$. Expanded to be in terms of the width and height, that's $\Theta\left(min(w,h) \cdot lg\left(\frac{max(w,h)}{min(w,h)}\right)\right)$.

Code

I implemented the algorithm in C#, and tested it. Here is the relevant code:

```
public static Tuple<int, int> TryFindItemInSortedMatrix< >(this IReadOnlyList<IReadOnlyList< >>      ,
      , IComparer< >             = null) {
    if (     == null) throw new ArgumentNullException("grid");
             =          ?? Comparer< >.Default;

    // check size
    var      =     .Count;
    if (     == 0) return null;
    var       =     [0].Count;
    if (      <      ) {
        // note: LazyTranspose uses LinqToCollections
        var        =     .LazyTranspose().TryFindItemInSortedMatrix(    ,         );
        if (       == null) return null;
        return Tuple.Create(      .Item2,       .Item1);
    }

    // search
    var       = 0;
    var       =        - 1;
    var   =        /      ;
    while (     <       &&        >= 0) {
        // query the item in the minimum column, t above the maximum row
        var         = Math.Max(       -  , 0);
        var              =         .Compare(    ,     [      ][        ]);
        if (             == 0) return Tuple.Create(      ,         );

        // did we eliminate t rows from the bottom?
        if (             < 0) {
                   =          - 1;
            continue;
        }

        // we eliminated most of the current minimum column
        // spend lg(t) time eliminating rest of column
        var           =          + 1;
        var           =       ;
        while (          <=            ) {
            var     =            + (              -             + 1) / 2;
            var              =         .Compare(    ,     [       ][   ]);
            if (             == 0) return Tuple.Create(      ,    );
            if (             > 0) {
                            =     + 1;
            } else {
                            =     - 1;
                       =      - 1;
            }
        }

               += 1;
    }

    return null;
}
```

Feel free to use the above code anytime you have to solve the searching a sorted matrix problem in production (uhh… yeah… about that…).

Summary

Searching a sorted matrix takes $\Theta(b \cdot lg(t))$ time, where $b = min(w, h)$ and $t = \frac{max(w,h)}{b}$, in the worst case.

Be wary that the solutions you'll find online are mostly not optimal, taking either $O(b \cdot t) = O(w + h)$ or $O(b \cdot lg(b \cdot t))$ time. After I solved the problem, I went looking but only found one person who found the right bound: +1 competence points for Evgeny Kluev.

—

Discuss on Reddit

—

My Twitter: @CraigGidney

—

4 Responses to "Searching a Sorted Matrix Faster"

1. *utopcell* says:
   August 30, 2013 at 9:45 pm

   Hi Craig,

   this is a very entertaining article.

   I believe that the "Divide and Conquer" approach can afford a more refined analysis however.

   For a matrix of w x h dimensions, with w <= h and w, h powers of 2, the following holds:
   T(w, h) = 1 + T(w/2, h/2) + T(w/2, h).
   Binary search is performed at the base cases: T(1, h) = log(h), T(w, 1) = log(w).

   One can inductively verify that the solution to this recurrence is: T(w, h) = w*(log(h) – 1/2*log(w) + 1) – 1, which is ?(wlog(h/w)).

   — Stergios

   - *CraigGidney* says:
     September 1, 2013 at 3:12 pm

     I checked your solution and it is indeed maintained by expanding T inside the recursive step, has the right complexity when w = 1 or h = 1, and has the same complexity in general. Good catch.

   - *CraigGidney* says:
     September 1, 2013 at 3:21 pm

Wait, hold on, when h=1 the function goes negative instead of solving out to log(w):

w*(log(h) – 1/2*log(w) + 1) – 1
@= w*(1 – 1/2*log(w) + 1) – 1
@= w – w log(w)
@= -w lg w

- *utopcell* says:
  September 1, 2013 at 4:00 pm

  This is where w <= h becomes relevant in a non-trivial way, and also the reason the large submatrix has size w/2 x h instead of w x h/2.

Twisted Oak Studios offers consulting and development on high-tech interactive projects. Check out our portfolio, or Give us a shout if you have anything you think some really rad engineers should help you with.

Archive

- strilanc
- What Quantum Computers Do Faster, with Caveats
- Naming Things: Fail-Useful
- Detecting Simple Cycles Forming, Faster
- Third Party Bit Commitment
- Angular Velocity is Simple
- Collection Equality is Hard
- Deadlocks in Practice: Don’t Hold Locks While Notifying
- Brute Force Parallelization
- A Year’s Worth of Opinions about Objective-C
- Referencing Substrings Faster, without Leaking Memory
- Not Crying Over Old Code
- Exploring Universal Ternary Gates
- Impractical Experiments #2: Securing Peer to Peer Fog of War against Map Hacks
- Achieving Exponential Slowdown by Enumerating Twice
- Using Immortality to Kill Accidental Callback Cycles
- Cancellation Tokens (and Collapsing Futures) for Objective-C
- Visualizing the Eigenvectors of a Rotation
- Collapsing Futures in Objective-C
- Bug Hunting #1: Garbled Audio from End to End
- Impractical Experiments #1: Representing Numbers as Polynomials
- Implementing Quantum Pseudo-Telepathy
- Turn On Your Damn Warnings
- Big-O Made Trivial
- Unfathomable Bugs #7: The Broken Oven
- Solomonoff’s Mad Scientist
- Yearly Blogging Roundup #1
- What isn’t a Monad
- Searching a Sorted Matrix Faster
- How to Read Nested Ternary Operators
- Making Sublime Text 2 Jump to the Correct Line with Unity on OS X

- My Bug, My Bad #4: Reading Concurrently
- Whole API Testing with Reflection
- Optimizing a Parser Combinator into a memcpy
- Don’t Treat Paths Like Strings
- Breaking a Toy Hash Function
- Counting Iterators Lazily
- Unfathomable Bugs #6: Pretend Precision
- My Bug, My Bad #3: Accidentally Attacking WarCraft 3
- Collapsing Types vs Monads (followup)
- Collapsing Futures: Easy to Use, Hard to Represent
- Eventual Exceptions vs Programming in a Minimal Functional Style
- The Mystery of Flunf
- Explain it like I’m Five: The Socialist Millionaire Problem and Secure Multi-Party Computation
- Computer Science Blows My Mind
- A visit to Execution Labs in Montréal
- Transmuting Dice, Conserving Entropy
- Rule of Thumb: Ask for the Clock
- Rule of Thumb: Use Purposefully Weakened Methods
- Rule of thumb: Preconditions Should be Checked Explicitly
- Intersecting Linked Lists Faster
- Mouse Path Smoothing for Jack Lumber
- My Bug, My Bad #2: Sunk by Float
- Repeat Yourself Differently
- Grover’s Quantum Search Algorithm
- Followup to Non-Nullable Types vs C#
- Optimizing Just in Time with Expression Trees
- When One-Way Latency Doesn’t Matter
- Determining exactly if/when/where a moving line intersected a moving point
- Emulating Actors in C# with Async/Await
- Making an immutable queue with guaranteed constant time operations
- Improving Checked Exceptions
- Perishable Collections: The Benefits of Removal-by-Lifetime
- Decoupling shared control
- Decoupling inlined UI code
- Linq to Collections: Beyond IEnumerable<T>
- Publish your .Net library as a NuGet package
- When null is not enough: an option type for C#
- Unfathomable Bugs #5: Readonly or not
- Minkowski sums: examples
- My Bug, My Bad #1: Fractal Spheres
- Working around the brittle UI Virtualization in Windows 8
- Encapsulating Angles
- Unfathomable Bugs #4: Keys that aren’t
- How would I even use a monad (in C#)?
- Useful/Interesting Methods #1: Observable.WhenEach
- Unfathomable Bugs #3: Stringing you along
- Anonymous Implementation Classes – A Design Pattern for C#
- Tasks for ActionScript 3 – Improving on Event-Driven Programming
- Minkowski sums and differences
- Non-Nullable Types vs C#: Fixing the Billion Dollar Mistake
- Unfathomable Bugs #2: Slashing Out
- Script templates and base classes
- Unity font extraction
- Abusing “Phantom Types” to Encode List Lengths Into Their Type

- Constructive Criticism of the Reactive Extensions API
- Quaternions part 3
- Quaternions part 2
- Quaternions part 1
- Unfathomable Bugs #1: You can have things! You can have things IN things! You can have …
- Coroutines – More than you want to know
- Asset Bundle Helper
- The Visual Studio goes away
- .Net’s time traveling StopWatch
- Introducing Catalyst

Twisted Oak Studios

2050 Gottingen Street
Halifax NS B3K 3A9

contact@twistedoakstudios.com

Twisted Oak Studios