[**Comparing Sequences Without Sorting**](https://0fps.net/2013/01/24/comparing-sequences-without-sorting/)

Posted on [January 24, 2013](https://0fps.net/2013/01/24/comparing-sequences-without-sorting/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

This is a post about a silly (and mostly pointless) optimization.  To motivate this, consider the following problem which you see all the time in mesh processing:

Given a collection of triangles represented as ordered tuples of indices, find all topological duplicates.

By a topological duplicate, we mean that the two faces have the same vertices.  Now you aren’t allowed to mutate the faces themselves (since orientation matters), but changing their order in the array is ok.  We could also consider a related problem for edges in a graph, or for tetrahedra in mesh.

There are of course many easy ways to solve this, but in the end it basically boils down to finding some function or other for comparing faces up to permutation, and then using either sorting or hashing to check for duplicates.  It is that comparison step that I want to talk about today, or in other words:

How do we check if two faces are the same?

In fact, what I really want to talk about is the following more general question where we let the lengths of our faces become arbitrarily large:

Given a pair of arrays, *A* and *B*, find a [total order](http://en.wikipedia.org/wiki/Total_order) \leq such A \leq B \: \mathrm{and} B \leq A if and only if A = \pi(B) up to some [permutation](http://en.wikipedia.org/wiki/Permutation) \pi.

For example, we should require that:

[0, 1, 2] \leq [2, 0, 1]

[2, 0, 1] \leq [0, 1, 2]

And for two sequences which are not equal, like [0,1,2] and [3, 4, 5] we want the ordering to be invariant under permutation.

Obvious Solution

An obvious solution to this problem is to sort the arrays element-wise and return the result:

function compareSimple(a, b) {

if(a.length !== b.length) {

return a.length - b.length;

}

var as = a.slice(0)

, bs = b.slice(0);

as.sort();

bs.sort();

for(var i=0; i<a.length; ++i) {

var d = as[i] - bs[i];

if(d) {

return d;

}

}

return 0;

}

This follows the standard C/JavaScript convention for comparison operators, where it returns -1 if a comes before b, +1 if a comes after b and 0 if they are equal.  For large enough sequences, this isn’t a bad solution.  It takes O(n \log(n)) time and is pretty straightforward to implement.  However, for smaller sequences it is somewhat suboptimal.  Most notably, it makes a copy of a and b, which introduces some overhead into the computation and in JavaScript may even trigger a garbage collection event (bad news).  If we need to do this on a large mesh, it could slow things down a lot.

An obvious way to fix this would be to try inlining the sorting function for small values of n (which is all we really care about), but doing this yourself is pure punishment.  Here is an `optimized` version for length 2 sets:

function compare2Brutal(a, b) {

if(a[0] < a[1]) {

if(b[0] < b[1]) {

if(a[0] === b[0]) {

return a[1] - b[1];

}

return a[0] - b[0];

} else {

if(a[0] === b[1]) {

return a[1] - b[0];

}

return a[0] - b[1];

}

} else {

if(b[0] < b[1]) {

if(a[1] === b[0]) {

return a[0] - b[1];

}

return a[1] - b[0];

} else {

if(a[1] === b[1]) {

return a[0] - b[0];

}

return a[1] - b[1];

}

}

}

If you have any aesthetic sensibility at all, then that code probably makes you want to vomit.   And that’s just for length two arrays!  You really don’t want to see what the version for length 3 arrays looks like.  But it is faster by a wide margin.  I ran a benchmark to try comparing how fast these two approaches were at sorting an array of 1000 randomly generated tuples, and here are the results:

* compareSimple:  5864 µs
* compareBrutal: 404 µs

That is a pretty good speed up, but it would be nice if there was a prettier way to do this.

Symmetry

The starting point for our next approach is the following screwy idea:  what if we could find a hash function for each face that was invariant under permutations?  Or even better, if the function was injective up to permutations, then we could just use the `[symmetrized hash](http://en.wikipedia.org/wiki/Symmetric_function)` compare two sets.  At first this may seem like a tall order, but if you know a little algebra then there is an obvious way to do this; namely you can use the [(elementary) symmetric polynomials](http://en.wikipedia.org/wiki/Elementary_symmetric_polynomial).  Here is how they are defined:

Given a collection of n numbers, a_0, a_1, ..., a_{n-1} the kth elementary symmetric polynomial, S_{n,k} is the coefficient of x^{n-k-1} in the polynomial:

(x + a_0) (x + a_1) ... (x + a_{n-1}) = S_{n,n-1} + S_{n,n-2} x + ... + S_{n,0} x^{n-1} + x^n

For example, if n = 2, then the symmetric polynomials are just:

S_{2,0} = a_0 + a_1

S_{2,1} = a_0 a_1

And for n = 3 we get:

S_{3,0} = a_0 + a_1 + a_2

S_{3,1} = a_0 a_1 + a_0 a_2 + a_1 a_2

S_{3,2} = a_0 a_1 a_2

The neat thing about these functions is that they contain enough data to uniquely determine a_0, a_1, ..., a_{n-1} up to permutation.  This is a consequence of the [fundamental theorem for elementary symmetric polynomials](http://en.wikipedia.org/wiki/Elementary_symmetric_polynomial#The_fundamental_theorem_of_symmetric_polynomials), which basically says that these S_{n,k} polynomials form a complete independent basis for the ring of all symmetric polynomials.  Using this trick, we can formulate a simplified version of the sequence comparison for n=2:

function compare2Sym(a, b) {

var d = a[0] + a[1] - b[0] - b[1];

if(d) {

return d;

}

return a[0] \* a[1] - b[0] \* b[1];

}

Not only is this way shorter than the brutal method, but it also turns out to be a little bit faster.  On the same test, I got:

* compare2Sym: 336 µs

Which is about a 25% improvement over brute force inlining.  The same idea extends to higher n, for example here is the result for n=3:

function compare3Sym(a, b) {

var d = a[0] + a[1] + a[2] - b[0] - b[1] - b[2];

if(d) {

return d;

}

d = a[0] \* a[1] + a[0] \* a[2] + a[1] \* a[2] - b[0] \* b[1] - b[0] \* b[2] - b[1] \* b[2];

if(d) {

return d;

}

return a[0] \* a[1] \* a[2] - b[0] \* b[1] \* b[2];

}

Running the sort-1000-items test against the simple method gives the following results:

* compareSimple: 7637 µs
* compare3Sym:  352 µs

Computing Symmetric Polynomials

This is all well and good, and it avoids making a copy of the arrays like we used in the basic sorting method.  However, it is also not very efficient.  If one were to compute the coefficients of a symmetric polynomial directly using the naive method we just wrote, then you would quickly end up with O(2^n) terms!  That is because the number of terms in \# S_{n,k} = { n \choose k+1 }, and so the binomial theorem tells us that:

\#S_{n,0} + \#S_{n,1} + \#S_{n,2} + ... + \#S_{n,n-1} = 2^n - 1

A slightly better way to compute them is to use the polynomial formula and apply the FOIL method.  That is, we just expand the symmetric polynomials using multiplication.  This dynamic programming algorithm speeds up the time complexity to O(n^2).  For example, here is an optimized version of the n=3 case:

function compare3SymOpt(a,b) {

var l0 = a[0] + a[1]

, m0 = b[0] + b[1]

, d = l0 + a[2] - m0 - b[2];

if(d) {

return d;

}

var l1 = a[0] \* a[1]

, m1 = b[0] \* b[1];

d = l1 \* a[2] - m1 \* b[2];

if(d) {

return d;

}

return l0 \* a[2] + l1 - m0 \* b[2] - m1;

}

For comparison, the first version of compare3 used 11 adds and 10 multiplies, while this new version only uses 9 adds and 6 multiplies, and also has the option to early out more often.  This may seem like an improvement, but it turns out that in practice the difference isn’t so great.  Based on my experiments, the reordered version ends up taking about the same amount of time overall, more or less:

* compare3SymOpt: 356 µs

Which isn’t very good.  Part of the reason for the discrepancy most likely has something to do with the way compare3Sym gets optimized.  One possible explanation is that the expressions in compare3Sym might be easier to vectorize than those in compare3SymOpt, though I must admit this is pretty speculative.

But there is also a deeper question of can we do better than O(n^2) asumptotically?  It turns out the answer is yes, and it requires the following observation:

Polynomial multiplication is [convolution](http://en.wikipedia.org/wiki/Convolution).

Using a fast convolution algorithm, we can multiply two polynomials together in O(n \log(n)) time.  Combined with a basic divide and conquer strategy, this gives an O(n \log^2(n)) algorithm for computing all the elementary symmetric functions. However, this is still worse than sorting the sequences!  It feels like we ought to be able to do better, but further progress escapes me.  I’ll pose the following question to you readers:

**Question:**Can we compute the n elementary symmetric polynomials in O(n \log(n)) time or better?

Overflow

Now there is still a small problem with using symmetric polynomials for this purpose: namely integer overflow.  If you have any indices in your tuple that go over 1000 then you are going to run into this problem once you start multiplying them together.  Fortunately, we can fix this problem by just working in a ring large enough to contain all our elements.  In languages with unsigned 32-bit integers, the natural choice is \mathbb{Z}_{2^{32}}, and we get these operations for free just using ordinary arithmetic.

But life is not so simple in the weird world of JavaScript!  It turns out for reasons that can charitably be described as “arbitrary” JavaScript does not support a proper integer data type, and so every number type in the language gets promoted to a double when it overflows whether you like it or not (mostly not).  The net result:  the above approach messes up.  One way to fix this is to try applying a modulus operator at each step, but the results are pretty bad.  Here are the timings for a modified version of compare2Sym that enforces bounds:

* compare2Masked: 1750 µs

That’s more than a 5-fold increase in running time, all because we added a few innocent bit masks!  How frustrating!

Semirings

The weirdness of JavaScript suggests that we need to find a better approach.  But in the world of commutative rings, it doesn’t seem like there are any immediately good alternatives.  And so we must cast the net wider.  One interesting possibility is to extend our approach to include [semirings](http://en.wikipedia.org/wiki/Semiring).  A semiring is basically a ring where we don’t require addition to be invertible, hence they are sometimes called “rigs” (which is short for a “ri**n**g without **n**egatives”, get it? haha).

Just like a ring, a semiring is basically a set S with a pair of operators \oplus, \otimes that act as generalized addition and multiplication.  You also have a pair of elements, 0,1 \in S which act as the additive and multiplicative identity.  These things then have to satisfy a list of axioms which are directly analogous to those of the [natural numbers](http://en.wikipedia.org/wiki/Natural_number) (ie nonnegative integers).  Here are a few examples of semirings, which you may or may not be familiar with:

* The [complex numbers](http://en.wikipedia.org/wiki/Complex_number) are semiring (and more generally, so is every [field](http://en.wikipedia.org/wiki/Field_(mathematics)))
* The [integers](http://en.wikipedia.org/wiki/Integer) are a semiring (and so is every other [ring](http://en.wikipedia.org/wiki/Ring_(mathematics)))
* The [natural numbers](http://en.wikipedia.org/wiki/Natural_number) are a semiring (but not a ring)
* The set of Boolean truth values, \mathbb{B} = \{ \mathrm{true}, \mathrm{false} \} under the operations OR, AND is a semiring (but is definitely not a ring)
* The set of reals under the operations min,max is a semiring (and more generally so is any [distributive lattice](http://en.wikipedia.org/wiki/Distributive_lattice))
* The [tropical semiring](http://en.wikipedia.org/wiki/Tropical_geometry#Basic_definitions), which is the set of reals under (max, +) is a semiring.

In many ways, semirings are more fundamental than rings.  After all, we learn to count stuff using the natural numbers long before we learn about integers.  But they are also much more diverse, and some of our favorite definitions from ring theory don’t necessarily translate well.  For those who are familiar with algebra, probably the most shocking thing is that the concept of an [ideal](http://en.wikipedia.org/wiki/Ideal_(ring_theory)) in a ring does not really have a good analogue in the language of semirings, much like how [monoids](http://en.wikipedia.org/wiki/Monoid) don’t really have any workable generalization of a [normal subgroup](http://en.wikipedia.org/wiki/Normal_subgroup).

Symmetric Polynomials in Semirings

However, for the modest purposes of this blog post, the most important thing is that we can define polynomials in a semiring (just like we do in a ring) and that we therefore have a good notion of an elementary symmetric polynomial.  The way this works is pretty much the same as before:

Let R be a semiring under \oplus, \otimes; then we have the symmetric functions.  Then for two variables, we have the symmetric functions:

S_{2,0} = a_0 \oplus a_1

S_{2,1} = a_0 \otimes a_1

And for n=3 we get:

S_{3,0} = a_0 \oplus a_1 \oplus a_2

S_{3,1} = (a_0 \otimes a_1) \oplus (a_0 \otimes a_2) \oplus (a_1 \otimes a_2)

S_{3,2} = a_0 \otimes a_1 \otimes a_2

And so on on…

Rank Functions and (min,max)

Let’s look at what happens if we fix a particular semiring, say the (min,max) lattice.  This is a semiring over the [extended real line](http://en.wikipedia.org/wiki/Extended_real_number_line) \mathbb{R} \cup \{ - \infty, \infty \} where:

* \oplus \mapsto \mathrm{min}
* \otimes \mapsto \mathrm{max}
* 0 \mapsto \infty
* 1 \mapsto -\infty

Now, here is a neat puzzle:

**Question:** What are the elementary symmetric polynomials in this semiring?

Here is a hint:

S_{2,0} = \min(a_0, a_1)

S_{2,1} = \max(a_0, a_1)

And…

S_{3,0} = \min(a_0, a_1, a_2)

S_{3,1} = \min( \max(a_0, a_1), \max(a_0, a_2), \max(a_1, a_2) )

S_{3,2} = \max(a_0, a_1, a_2)

Give up?  These are the rank functions!

**Theorem:**Over the min,max semiring, S_{n,k} is the kth element of the sorted sequence a_0, a_1, ...

In other words, evaluating the symmetric polynomials over the min/max semiring is the same thing as sorting.  It also suggests a more streamlined way to do the brutal inlining of a sort:

function compare2Rank(a, b) {

var d = Math.min(a[0],a[1]) - Math.min(b[0],b[1]);

if(d) {

return d;

}

return Math.max(a[0],a[1]) - Math.max(b[0],b[1]);

}

Slick!  We went from 25 lines down to just 5.  Unfortunately, this approach is a bit less efficient since it does the comparison between a and b twice, a fact which is reflected in the timing measurements:

* compare2Rank: 495 µs

And we can also easily extend this technique to triples as well:

function compare3Rank(a,b) {

var l0 = Math.min(a[0], a[1])

, m0 = Math.min(b[0], b[1])

, d = Math.min(l0, a[2]) - Math.min(m0, b[2]);

if(d) {

return d;

}

var l1 = Math.max(a[0], a[1])

, m1 = Math.max(b[0], b[1]);

d = Math.max(l1, a[2]) - Math.max(m1, b[2]);

if(d) {

return d;

}

return Math.min(Math.max(l0, a[2]), l1) - Math.min(Math.max(m0, b[2]), m1);

}

* compare3Rank: 618.71 microseconds

The Tropical Alternative

Using rank functions to sort the elements turns out to be much simpler than doing a selection sort, and it is way faster than calling the native JS sort on small arrays while avoiding the overflow problems of integer symmetric functions.  However, it is still quite a bit slower than the integer approach.

The final method which I will now describe (and the one which I believe to be best suited to the vagaries of JS) is to compute the symmetric functions over the (max,+), or tropical semiring.  It is basically a semiring over the half-extended real line, \mathbb{R} \cup \{ - \infty \} with the following operators:

* \oplus \mapsto \max
* \otimes \mapsto +
* 0 \mapsto -\infty
* 1 \mapsto 0

There is a cute story for why the tropical semiring has such a colorful name, which is that it was popularized at an algebraic geometry conference in Brazil.  Of course people have known about (max,+) for quite some time before that and most of the pioneering research into it was done by the mathematician [Victor P. Maslov](http://en.wikipedia.org/wiki/Victor_Pavlovich_Maslov) in the 50s.  The (max,+) semiring is actually quite interesting and plays an important role in the algebra of distance transforms, numerical optimization and the transition from quantum systems to classical systems.

This is because the (max,+) semiring works in many ways like the large scale limit of the complex numbers under addition and multiplication.  For example, we all know that:

\log(a b) = \log(a) + \log(b)

But did you also know that:

\log(a + b) = \max(\log(a), \log(b)) + O(1)

This basically a formalization of that time honored engineering philosophy that once your numbers get big enough, you can start to think about them on a log scale.  If you do this systematically, then you eventually will end up doing arithmetic in the (max,+)-semiring.  Masolv asserts that this is essentially what happens in quantum mechanics when we go from small isolated systems to very big things.  A brief overview of this philosophy that has been translated to English can be found here:

Litvinov, G. “[The Maslov dequantization, idempotent and tropical mathematics: a very brief introduction](http://arxiv.org/abs/math/0501038)” (2006) arXiv:math/0501038

The more detailed explanations of this are unfortunately all store in thick, inscrutable Russian text books (but you can find an English translation if you look hard enough):

Maslov, V. P.; Fedoriuk, M. V. “Semiclassical approximation in quantum mechanics”

But enough of that digression!  Let’s apply (max,+) to the problem at hand of comparing sequences.  If we expand the symmetric polynomials in the (max,+) world, here is what we get:

S_{2,0} = \max(a_0, a_1)

S_{2,1} = a_0 + a_1

And for n = 3:

S_{3,0} = \max(a_0, a_1, a_2)

S_{3,1} = \max(a_0+a_1, a_0+a_2, a_1+a_2)

S_{3,2} = a_0+a_1+a_2

If you stare at this enough, I am sure you can spot the pattern:

**Theorem:** The elementary symmetric polynomials on the (max,+) semiring are the [partial sums](http://en.wikipedia.org/wiki/Series_(mathematics)#Calculus_and_partial_summation_as_an_operation_on_sequences) of the sorted sequence.

This means that if we want to compute the (max,+) symmetric polynomials, we can do it in O(n \log(n)) by sorting and folding.

Working the (max,+) solves most of our woes about overflow, since adding numbers is much less likely to go beyond INT\_MAX.  However, we will tweak just one thing: instead of using max, we’ll flip it around and use min so that the values stay small.  Both theoretically and practically, it doesn’t save much but it gives us a maybe a fraction of a bit of extra address space to use for indices.  Here is an implementation for pairs:

function compare2MinP(a, b) {

var d = a[0]+a[1]-b[0]-b[1];

if(d) {

return d;

}

return Math.min(a[0],a[1]) - Math.min(b[0],b[1]);

}

And it clocks in at:

* compare2MinP: 354 µs

Which is a bit slower than the symmetric functions, but still way faster than ranking.  We can also play the same game for n=3:

function compare3MinP(a, b) {

var l1 = a[0]+a[1]

, m1 = b[0]+b[1];

d = l1+a[2] - (m1+b[2]);

if(d) {

return d;

}

var l0 = Math.min(a[0], a[1])

, m0 = Math.min(b[0], b[1])

, d = Math.min(l0, a[2]) - Math.min(m0, b[2]);

if(d) {

return d;

}

return Math.min(l0+a[2], l1) - Math.min(m0+b[2], m1);

}

Which hits:

* compare3MinP: 382 µs

Again, not quite as fast as integers, but pretty good for JavaScript.

Summary

You can get all the code to run these experiments on github:

<https://github.com/mikolalysenko/set-compare>

And here are the results that I got from running the experiment, all collated for easy reading:

Dimension = 2

* compareSimple: 5982 µs
* compare2Brutal: 415 µs
* compare2Sym: 352 µs
* compare2SymMasked: 1737 µs
* compare2Rank: 498 µs
* compare2MinP: 369 µs

Dimension = 3

* compareSimple: 7737 µs
* compare3Sym: 361 µs
* compare3Sym\_opt: 362 µs
* compare3Rank: 612 µs
* compare3MinP: 377 µs

As you can see, the (min,+) solution is nearly as fast as the symmetric version without having the same overflow problems.

I hope you enjoyed reading this as much as I enjoyed tinkering around!  Of course I still don’t know of an optimal way to compare two lists.  As a final puzzle, I leave you with the following:

**Question:**Is there any method which can test if two unordered sequences are equal in linear time and at most log space?

Frankly, I don’t know the answer to this question and it may very well be impossible.  If you have any thoughts or opinions on this, please leave a comment!