NAME

Algorithm::Diff − Compute ‘intelligent’ differences between two files / lists

SYNOPSIS

```
require Algorithm::Diff;

# This example produces traditional 'diff' output:

my $diff = Algorithm::Diff->new( \@seq1, \@seq2 );

$diff->Base( 1 );   # Return line numbers, not indices
while(  $diff->Next()  ) {
    next   if  $diff->Same();
    my $sep = '';
    if(  ! $diff->Items(2)  ) {
        printf "%d,%dd%d\n",
            $diff->Get(qw( Min1 Max1 Max2 ));
    } elsif(  ! $diff->Items(1)  ) {
        printf "%da%d,%d\n",
            $diff->Get(qw( Max1 Min2 Max2 ));
    } else {
        $sep = "---\n";
        printf "%d,%dc%d,%d\n",
            $diff->Get(qw( Min1 Max1 Min2 Max2 ));
    }
    print "< $_"   for  $diff->Items(1);
    print $sep;
    print "> $_"   for  $diff->Items(2);
}


# Alternate interfaces:

use Algorithm::Diff qw(
    LCS LCS_length LCSidx
    diff sdiff compact_diff
    traverse_sequences traverse_balanced );

@lcs    = LCS( \@seq1, \@seq2 );
$lcsref = LCS( \@seq1, \@seq2 );
$count  = LCS_length( \@seq1, \@seq2 );

( $seq1idxref, $seq2idxref ) = LCSidx( \@seq1, \@seq2 );


# Complicated interfaces:

@diffs  = diff( \@seq1, \@seq2 );

@sdiffs = sdiff( \@seq1, \@seq2 );

@cdiffs = compact_diff( \@seq1, \@seq2 );

traverse_sequences(
    \@seq1,
```

```
        \@seq2,
        {   MATCH     => \&callback1,
            DISCARD_A => \&callback2,
            DISCARD_B => \&callback3,
        },
        \&key_generator,
        @extra_args,
    );

    traverse_balanced(
        \@seq1,
        \@seq2,
        {   MATCH     => \&callback1,
            DISCARD_A => \&callback2,
            DISCARD_B => \&callback3,
            CHANGE    => \&callback4,
        },
        \&key_generator,
        @extra_args,
    );
```

INTRODUCTION

(by Mark-Jason Dominus)

I once read an article written by the authors of `diff`; they said that they worked very hard on the algorithm until they found the right one.

I think what they ended up using (and I hope someone will correct me, because I am not very confident about this) was the 'longest common subsequence' method. In the LCS problem, you have two sequences of items:

```
a b c d f g h j q z

a b c d e f g i j k r x y z
```

and you want to find the longest sequence of items that is present in both original sequences in the same order. That is, you want to find a new sequence *S* which can be obtained from the first sequence by deleting some items, and from the second sequence by deleting other items. You also want *S* to be as long as possible. In this case*S* is

```
a b c d f g j z
```

From there it's only a small step to get diff-like output:

```
e   h i   k   q r x y
+   - +   +   - + + +
```

This module solves the LCS problem. It also includes a canned function to generate `diff`-like output.

It might seem from the example above that the LCS of two sequences is always pretty obvious, but that's not always the case, especially when the two sequences have many repeated elements. For example, consider

```
a x b y c z p d q
a b c a x b y c z
```

A naive approach might start by matching up the `a` and `b` that appear at the beginning of each sequence, like this:

```
a x b y c         z p d q
a   b   c a b y c z
```

This finds the common subsequence `a b c z`. But actually, the LCS is `a x b y c z`:

```
          a x b y c z p d q
    a b c a x b y c z
```

or

```
    a       x b y c z p d q
    a b c a x b y c z
```

USAGE

(See also the README file and several example scripts include with this module.)

This module now provides an object-oriented interface that uses less memory and is easier to use than most of the previous procedural interfaces. It also still provides several exportable functions. We'll deal with these in ascending order of difficulty: `LCS`, `LCS_length`, `LCSidx`, OO interface, `prepare`, `diff`, `sdiff`, `traverse_sequences`, and `traverse_balanced`.

LCS

Given references to two lists of items, LCS returns an array containing their longest common subsequence. In scalar context, it returns a reference to such a list.

```
    @lcs    = LCS( \@seq1, \@seq2 );
    $lcsref = LCS( \@seq1, \@seq2 );
```

`LCS` may be passed an optional third parameter; this is a CODE reference to a key generation function. See ''KEY GENERATION FUNCTIONS''.

```
    @lcs    = LCS( \@seq1, \@seq2, \&keyGen, @args );
    $lcsref = LCS( \@seq1, \@seq2, \&keyGen, @args );
```

Additional parameters, if any, will be passed to the key generation routine.

LCS_length

This is just like `LCS` except it only returns the length of the longest common subsequence. This provides a performance gain of about 9% compared to `LCS`.

LCSidx

Like `LCS` except it returns references to two arrays. The first array contains the indices into `@seq1` where the LCS items are located. The second array contains the indices into `@seq2` where the LCS items are located.

Therefore, the following three lists will contain the same values:

```
    my( $idx1, $idx2 ) = LCSidx( \@seq1, \@seq2 );
    my @list1 = @seq1[ @$idx1 ];
    my @list2 = @seq2[ @$idx2 ];
    my @list3 = LCS( \@seq1, \@seq2 );
```

new

```
    $diff = Algorithm::Diff->new( \@seq1, \@seq2 );
    $diff = Algorithm::Diff->new( \@seq1, \@seq2, \%opts );
```

`new` computes the smallest set of additions and deletions necessary to turn the first sequence into the second and compactly records them in the object.

You use the object to iterate over *hunks*, where each hunk represents a contiguous section of items which should be added, deleted, replaced, or left unchanged.

The following summary of all of the methods looks a lot like Perl code but some of the symbols have different meanings:

```
    [ ]     Encloses optional arguments
    :       Is followed by the default value for an optional argument
    |       Separates alternate return results
```

Method summary:

```
    $obj        = Algorithm::Diff->new( \@seq1, \@seq2, [ \%opts ] );
    $pos        = $obj->Next(  [ $count : 1 ] );
    $revPos     = $obj->Prev(  [ $count : 1 ] );
    $obj        = $obj->Reset( [ $pos : 0 ] );
    $copy       = $obj->Copy(  [ $pos, [ $newBase ] ] );
    $oldBase    = $obj->Base(  [ $newBase ] );
```

Note that all of the following methods `die` if used on an object that is ‘‘reset’’ (not currently pointing at any hunk).

```
    $bits       = $obj->Diff(  );
    @items|$cnt = $obj->Same(  );
    @items|$cnt = $obj->Items( $seqNum );
    @idxs |$cnt = $obj->Range( $seqNum, [ $base ] );
    $minIdx     = $obj->Min(   $seqNum, [ $base ] );
    $maxIdx     = $obj->Max(   $seqNum, [ $base ] );
    @values     = $obj->Get(   @names );
```

Passing in `undef` for an optional argument is always treated the same as if no argument were passed in.

`Next`

```
        $pos = $diff->Next();    # Move forward 1 hunk
        $pos = $diff->Next( 2 ); # Move forward 2 hunks
        $pos = $diff->Next(-5);  # Move backward 5 hunks
```

`Next` moves the object to point at the next hunk. The object starts out ‘‘reset’’, which means it isn’t pointing at any hunk. If the object is reset, then `Next()` moves to the first hunk.

`Next` returns a true value iff the move didn’t go past the last hunk. So `Next(0)` will return true iff the object is not reset.

Actually, `Next` returns the object’s new position, which is a number between 1 and the number of hunks (inclusive), or returns a false value.

`Prev`

`Prev($N)` is almost identical to `Next(-$N)`; it moves to the `$Nth` previous hunk. On a ’reset’ object, `Prev()` [and `Next(-1)`] move to the last hunk.

The position returned by `Prev` is relative to the *end* of the hunks; −1 for the last hunk, −2 for the second-to-last, etc.

`Reset`

```
        $diff->Reset();     # Reset the object's position
        $diff->Reset($pos); # Move to the specified hunk
        $diff->Reset(1);    # Move to the first hunk
        $diff->Reset(-1);   # Move to the last hunk
```

`Reset` returns the object, so, for example, you could use `$diff->Reset()->Next(-1)` to get the number of hunks.

`Copy`

```
        $copy = $diff->Copy( $newPos, $newBase );
```

`Copy` returns a copy of the object. The copy and the original object share most of their data, so making copies takes very little memory. The copy maintains its own position (separate from the original), which is the main purpose of copies. It also maintains its own base.

By default, the copy’s position starts out the same as the original object’s position. But `Copy` takes an optional first argument to set the new position, so the following three snippets are equivalent:

```
    $copy = $diff->Copy($pos);

    $copy = $diff->Copy();
```

```
    $copy->Reset($pos);

    $copy = $diff->Copy()->Reset($pos);
```

`Copy` takes an optional second argument to set the base for the copy. If you wish to change the base of the copy but leave the position the same as in the original, here are two equivalent ways:

```
    $copy = $diff->Copy();
    $copy->Base( 0 );

    $copy = $diff->Copy(undef,0);
```

Here are two equivalent way to get a "reset" copy:

```
    $copy = $diff->Copy(0);

    $copy = $diff->Copy()->Reset();
```

`Diff`

```
    $bits = $obj->Diff();
```

`Diff` returns a true value iff the current hunk contains items that are different between the two sequences. It actually returns one of the follow 4 values:

3 `3==(1|2)`. This hunk contains items from`@seq1` and the items from `@seq2` that should replace them. Both sequence 1 and 2 contain changed items so both the 1 and 2 bits are set.

2 This hunk only contains items from `@seq2` that should be inserted (not items from `@seq1`). Only sequence 2 contains changed items so only the 2 bit is set.

1 This hunk only contains items from `@seq1` that should be deleted (not items from `@seq2`). Only sequence 1 contains changed items so only the 1 bit is set.

0 This means that the items in this hunk are the same in both sequences. Neither sequence 1 nor 2 contain changed items so neither the 1 nor the 2 bits are set.

`Same`

`Same` returns a true value iff the current hunk contains items that are the same in both sequences. It actually returns the list of items if they are the same or an empty list if they aren't. In a scalar context, it returns the size of the list.

`Items`

```
    $count = $diff->Items(2);
    @items = $diff->Items($seqNum);
```

`Items` returns the (number of) items from the specified sequence that are part of the current hunk.

If the current hunk contains only insertions, then `$diff->Items(1)` will return an empty list (0 in a scalar context). If the current hunk contains only deletions, then`$diff->Items(2)` will return an empty list (0 in a scalar context).

If the hunk contains replacements, then both `$diff->Items(1)` and `$diff->Items(2)` will return different, non-empty lists.

Otherwise, the hunk contains identical items and all of the following will return the same lists:

```
    @items = $diff->Items(1);
    @items = $diff->Items(2);
    @items = $diff->Same();
```

`Range`

```
    $count = $diff->Range( $seqNum );
    @indices = $diff->Range( $seqNum );
    @indices = $diff->Range( $seqNum, $base );
```

`Range` is like `Items` except that it returns a list of *indices* to the items rather than the items themselves. By default, the index of the first item (in each sequence) is 0 but this can be changed by calling the `Base` method. So, by default, the following two snippets return the same lists:

```
    @list = $diff->Items(2);
    @list = @seq2[ $diff->Range(2) ];
```

You can also specify the base to use as the second argument. So the following two snippets *always* return the same lists:

```
    @list = $diff->Items(1);
    @list = @seq1[ $diff->Range(1,0) ];
```

`Base`

```
    $curBase = $diff->Base();
    $oldBase = $diff->Base($newBase);
```

`Base` sets and/or returns the current base (usually 0 or 1) that is used when you request range information. The base defaults to 0 so that range information is returned as array indices. You can set the base to 1 if you want to report traditional line numbers instead.

`Min`

```
    $min1 = $diff->Min(1);
    $min = $diff->Min( $seqNum, $base );
```

`Min` returns the first value that `Range` would return (given the same arguments) or returns `undef` if `Range` would return an empty list.

`Max`

`Max` returns the last value that `Range` would return or `undef`.

`Get`

```
    ( $n, $x, $r ) = $diff->Get(qw( min1 max1 range1 ));
    @values = $diff->Get(qw( 0min2 1max2 range2 same base ));
```

`Get` returns one or more scalar values. You pass in a list of the names of the values you want returned. Each name must match one of the following regexes:

```
    /^(-?\d+)?(min|max)[12]$/i
    /^(range[12]|same|diff|base)$/i
```

The 1 or 2 after a name says which sequence you want the information for (and where allowed, it is required). The optional number before ‘‘min’’ or ‘‘max’’ is the base to use. So the following equalities hold:

```
    $diff->Get('min1') == $diff->Min(1)
    $diff->Get('0min2') == $diff->Min(2,0)
```

Using `Get` in a scalar context when you’ve passed in more than one name is a fatal error (`die` is called).

`prepare`

Given a reference to a list of items, `prepare` returns a reference to a hash which can be used when comparing this sequence to other sequences with `LCS` or `LCS_length`.

```
    $prep = prepare( \@seq1 );
    for $i ( 0 .. 10_000 )
    {
        @lcs = LCS( $prep, $seq[$i] );
        # do something useful with @lcs
    }
```

`prepare` may be passed an optional third parameter; this is a CODE reference to a key generation function. See ''KEY GENERATION FUNCTIONS''.

```
    $prep = prepare( \@seq1, \&keyGen );
    for $i ( 0 .. 10_000 )
    {
        @lcs = LCS( $seq[$i], $prep, \&keyGen );
        # do something useful with @lcs
    }
```

Using `prepare` provides a performance gain of about 50% when calling LCS many times compared with not preparing.

`diff`

```
    @diffs     = diff( \@seq1, \@seq2 );
    $diffs_ref = diff( \@seq1, \@seq2 );
```

`diff` computes the smallest set of additions and deletions necessary to turn the first sequence into the second, and returns a description of these changes. The description is a list of *hunks*; each hunk represents a contiguous section of items which should be added, deleted, or replaced. (Hunks containing unchanged items are not included.)

The return value of `diff` is a list of hunks, or, in scalar context, a reference to such a list. If there are no differences, the list will be empty.

Here is an example. Calling`diff` for the follo wing two sequences:

```
    a b c e h j l m n p
    b c d e f j k l m r s t
```

would produce the following list:

```
    (
      [ [ '-', 0, 'a' ] ],

      [ [ '+', 2, 'd' ] ],

      [ [ '-', 4, 'h' ],
        [ '+', 4, 'f' ] ],

      [ [ '+', 6, 'k' ] ],

      [ [ '-',  8, 'n' ],
        [ '-',  9, 'p' ],
        [ '+',  9, 'r' ],
        [ '+', 10, 's' ],
        [ '+', 11, 't' ] ],
    )
```

There are five hunks here. The first hunk says that the `a` at position 0 of the first sequence should be deleted (`-`). The second hunk says that the`d` at position 2 of the second sequence should be inserted (`+`). The third hunk says that the `h` at position 4 of the first sequence should be removed and replaced with the `f` from position 4 of the second sequence. And so on.

`diff` may be passed an optional third parameter; this is a CODE reference to a key generation function.

See "KEY GENERATION FUNCTIONS".

Additional parameters, if any, will be passed to the key generation routine.

`sdiff`

```
    @sdiffs     = sdiff( \@seq1, \@seq2 );
    $sdiffs_ref = sdiff( \@seq1, \@seq2 );
```

`sdiff` computes all necessary components to show two sequences and their minimized differences side by side, just like the Unix-utility *sdiff* does:

```
    same             same
    before     |     after
    old        <     -
    -          >     new
```

It returns a list of array refs, each pointing to an array of display instructions. In scalar context it returns a reference to such a list. If there are no differences, the list will have one entry per item, each indicating that the item was unchanged.

Display instructions consist of three elements: A modifier indicator (`+`: Element added, `-`: Element removed, `u`: Element unmodified, `c`: Element changed) and the value of the old and new elements, to be displayed side-by-side.

An `sdiff` of the following two sequences:

```
    a b c e h j l m n p
    b c d e f j k l m r s t
```

results in

```
    ( [ '-', 'a', ''  ],
      [ 'u', 'b', 'b' ],
      [ 'u', 'c', 'c' ],
      [ '+', '',  'd' ],
      [ 'u', 'e', 'e' ],
      [ 'c', 'h', 'f' ],
      [ 'u', 'j', 'j' ],
      [ '+', '',  'k' ],
      [ 'u', 'l', 'l' ],
      [ 'u', 'm', 'm' ],
      [ 'c', 'n', 'r' ],
      [ 'c', 'p', 's' ],
      [ '+', '',  't' ],
    )
```

`sdiff` may be passed an optional third parameter; this is a CODE reference to a key generation function. See "KEY GENERATION FUNCTIONS".

Additional parameters, if any, will be passed to the key generation routine.

`compact_diff`

`compact_diff` is much like `sdiff` except it returns a much more compact description consisting of just one flat list of indices. An example helps explain the format:

```
my @a = qw( a b c   e  h j   l m n p      );
my @b = qw(   b c d e f  j k l m    r s t );
@cdiff = compact_diff( \@a, \@b );
# Returns:
#   @a      @b       @a       @b
#  start   start   values   values
(    0,      0,   #       =
     0,      0,   #    a  !
     1,      0,   #  b c  =  b c
     3,      2,   #       !  d
     3,      3,   #    e  =  e
     4,      4,   #    f  !  h
     5,      5,   #    j  =  j
     6,      6,   #       !  k
     6,      7,   #  l m  =  l m
     8,      9,   #  n p  !  r s t
    10,     12,   #
);
```

The 0th, 2nd, 4th, etc. entries are all indices into `@seq1` (@a in the above example) indicating where a hunk begins. The 1st, 3rd, 5th, etc. entries are all indices into`@seq2` (@b in the abo ve example) indicating where the same hunk begins.

So each pair of indices (except the last pair) describes where a hunk begins (in each sequence). Since each hunk must end at the item just before the item that starts the next hunk, the next pair of indices can be used to determine where the hunk ends.

So, the first 4 entries (0..3) describe the first hunk. Entries 0 and 1 describe where the first hunk begins (and so are always both 0). Entries 2 and 3 describe where the next hunk begins, so subtracting 1 from each tells us where the first hunk ends. That is, the first hunk contains items `$diff[0]` through `$diff[2] - 1` of the first sequence and contains items `$diff[1]` through `$diff[3] - 1` of the second sequence.

In other words, the first hunk consists of the following two lists of items:

```
            #  1st pair     2nd pair
            # of indices   of indices
@list1 = @a[ $cdiff[0] .. $cdiff[2]-1 ];
@list2 = @b[ $cdiff[1] .. $cdiff[3]-1 ];
            # Hunk start   Hunk end
```

Note that the hunks will always alternate between those that are part of the LCS (those that contain unchanged items) and those that contain changes. This means that all we need to be told is whether the first hunk is a 'same' or 'diff' hunk and we can determine which of the other hunks contain 'same' items or 'diff' items.

By convention, we always make the first hunk contain unchanged items. So the 1st, 3rd, 5th, etc. hunks (all odd-numbered hunks if you start counting from 1) all contain unchanged items. And the 2nd, 4th, 6th, etc. hunks (all even-numbered hunks if you start counting from 1) all contain changed items.

Since `@a` and `@b` don't begin with the same value, the first hunk in our example is empty (otherwise we'd violate the above convention). Note that the first 4 index values in our example are all zero. Plug these values into our previous code block and we get:

```
@hunk1a = @a[ 0 .. 0-1 ];
@hunk1b = @b[ 0 .. 0-1 ];
```

And `0..-1` returns the empty list.

Move down one pair of indices (2..5) and we get the offset ranges for the second hunk, which contains changed items.

Since `@diff[2..5]` contains (0,0,1,0) in our example, the second hunk consists of these two lists of

items:

```
        @hunk2a = @a[ $cdiff[2] .. $cdiff[4]-1 ];
        @hunk2b = @b[ $cdiff[3] .. $cdiff[5]-1 ];
    # or
        @hunk2a = @a[ 0 .. 1-1 ];
        @hunk2b = @b[ 0 .. 0-1 ];
    # or
        @hunk2a = @a[ 0 .. 0 ];
        @hunk2b = @b[ 0 .. -1 ];
    # or
        @hunk2a = ( 'a' );
        @hunk2b = ( );
```

That is, we would delete item 0 ('a') from `@a`.

Since `@diff[4..7]` contains (1,0,3,2) in our example, the third hunk consists of these two lists of items:

```
        @hunk3a = @a[ $cdiff[4] .. $cdiff[6]-1 ];
        @hunk3a = @b[ $cdiff[5] .. $cdiff[7]-1 ];
    # or
        @hunk3a = @a[ 1 .. 3-1 ];
        @hunk3a = @b[ 0 .. 2-1 ];
    # or
        @hunk3a = @a[ 1 .. 2 ];
        @hunk3a = @b[ 0 .. 1 ];
    # or
        @hunk3a = qw( b c );
        @hunk3a = qw( b c );
```

Note that this third hunk contains unchanged items as our convention demands.

You can continue this process until you reach the last two indices, which will always be the number of items in each sequence. This is required so that subtracting one from each will give you the indices to the last items in each sequence.

`traverse_sequences`

`traverse_sequences` used to be the most general facility provided by this module (the new OO interface is more powerful and much easier to use).

Imagine that there are two arrows. Arrow A points to an element of sequence A, and arrow B points to an element of the sequence B. Initially, the arrows point to the first elements of the respective sequences. `traverse_sequences` will advance the arrows through the sequences one element at a time, calling an appropriate user-specified callback function before each advance. It will advance the arrows in such a way that if there are equal elements `$A[$i]` and `$B[$j]` which are equal and which are part of the LCS, there will be some moment during the execution of `traverse_sequences` when arrow A is pointing to `$A[$i]` and arrow B is pointing to `$B[$j]`. When this happens,`traverse_sequences` will call the `MATCH` callback function and then it will advance both arrows.

Otherwise, one of the arrows is pointing to an element of its sequence that is not part of the LCS. `traverse_sequences` will advance that arrow and will call the `DISCARD_A` or the `DISCARD_B` callback, depending on which arrow it advanced. If both arrows point to elements that are not part of the LCS, then `traverse_sequences` will advance one of them and call the appropriate callback, but it is not specified which it will call.

The arguments to `traverse_sequences` are the two sequences to traverse, and a hash which specifies the callback functions, like this:

```
    traverse_sequences(
        \@seq1, \@seq2,
        {   MATCH => $callback_1,
            DISCARD_A => $callback_2,
            DISCARD_B => $callback_3,
        }
    );
```

Callbacks for MATCH, DISCARD_A, and DISCARD_B are invoked with at least the indices of the two arrows as their arguments. They are not expected to return any values. If a callback is omitted from the table, it is not called.

Callbacks for A_FINISHED and B_FINISHED are invoked with at least the corresponding index in A or B.

If arrow A reaches the end of its sequence, before arrow B does, `traverse_sequences` will call the `A_FINISHED` callback when it advances arrow B, if there is such a function; if not it will call `DISCARD_B` instead. Similarly if arrow B finishes first. `traverse_sequences` returns when both arrows are at the ends of their respective sequences. It returns true on success and false on failure. At present there is no way to fail.

`traverse_sequences` may be passed an optional fourth parameter; this is a CODE reference to a key generation function. See ''KEY GENERATION FUNCTIONS''.

Additional parameters, if any, will be passed to the key generation function.

If you want to pass additional parameters to your callbacks, but don't need a custom key generation function, you can get the default by passing undef:

```
    traverse_sequences(
        \@seq1, \@seq2,
        {   MATCH => $callback_1,
            DISCARD_A => $callback_2,
            DISCARD_B => $callback_3,
        },
        undef,      # default key-gen
        $myArgument1,
        $myArgument2,
        $myArgument3,
    );
```

`traverse_sequences` does not have a useful return value; you are expected to plug in the appropriate behavior with the callback functions.

`traverse_balanced`

`traverse_balanced` is an alternative to `traverse_sequences`. It uses a different algorithm to iterate through the entries in the computed LCS. Instead of sticking to one side and showing element changes as insertions and deletions only, it will jump back and forth between the two sequences and report *changes* occurring as deletions on one side followed immediately by an insertion on the other side.

In addition to the `DISCARD_A`, `DISCARD_B`, and `MATCH` callbacks supported by `traverse_sequences`, `traverse_balanced` supports a `CHANGE` callback indicating that one element got `replaced` by another:

```
traverse_balanced(
    \@seq1, \@seq2,
    {   MATCH => $callback_1,
        DISCARD_A => $callback_2,
        DISCARD_B => $callback_3,
        CHANGE    => $callback_4,
    }
);
```

If no `CHANGE` callback is specified, `traverse_balanced` will map `CHANGE` events to `DISCARD_A` and `DISCARD_B` actions, therefore resulting in a similar behaviour as `traverse_sequences` with different order of events.

`traverse_balanced` might be a bit slower than `traverse_sequences`, noticeable only while processing huge amounts of data.

The `sdiff` function of this module is implemented as call to `traverse_balanced`.

`traverse_balanced` does not have a useful return value; you are expected to plug in the appropriate behavior with the callback functions.

KEY GENERATION FUNCTIONS

Most of the functions accept an optional extra parameter. This is a CODE reference to a key generating (hashing) function that should return a string that uniquely identifies a given element. It should be the case that if two elements are to be considered equal, their keys should be the same (and the other way around). If no key generation function is provided, the key will be the element as a string.

By default, comparisons will use ‘‘eq’’ and elements will be turned into keys using the default stringizing operator ’""’.

Where this is important is when you’re comparing something other than strings. If it is the case that you have multiple different objects that should be considered to be equal, you should supply a key generation function. Otherwise, you have to make sure that your arrays contain unique references.

For instance, consider this example:

```
package Person;

sub new
{
    my $package = shift;
    return bless { name => '', ssn => '', @_ }, $package;
}

sub clone
{
    my $old = shift;
    my $new = bless { %$old }, ref($old);
}

sub hash
{
    return shift()->{'ssn'};
}

my $person1 = Person->new( name => 'Joe', ssn => '123-45-6789' );
my $person2 = Person->new( name => 'Mary', ssn => '123-47-0000' );
my $person3 = Person->new( name => 'Pete', ssn => '999-45-2222' );
my $person4 = Person->new( name => 'Peggy', ssn => '123-45-9999' );
my $person5 = Person->new( name => 'Frank', ssn => '000-45-9999' );
```

If you did this:

```
my $array1 = [ $person1, $person2, $person4 ];
my $array2 = [ $person1, $person3, $person4, $person5 ];
Algorithm::Diff::diff( $array1, $array2 );
```

everything would work out OK (each of the objects would be converted into a string like ''Person=HASH(0x82425b0)'' for comparison).

But if you did this:

```
my $array1 = [ $person1, $person2, $person4 ];
my $array2 = [ $person1, $person3, $person4->clone(), $person5 ];
Algorithm::Diff::diff( $array1, $array2 );
```

`$person4` and `$person4->`**clone()** (which have the same name and SSN) would be seen as different objects. If you wanted them to be considered equivalent, you would have to pass in a key generation function:

```
my $array1 = [ $person1, $person2, $person4 ];
my $array2 = [ $person1, $person3, $person4->clone(), $person5 ];
Algorithm::Diff::diff( $array1, $array2, \&Person::hash );
```

This would use the 'ssn' field in each Person as a comparison key, and so would consider `$person4` and `$person4->`**clone()** as equal.

You may also pass additional parameters to the key generation function if you wish.

ERROR CHECKING

If you pass these routines a non-reference and they expect a reference, they will die with a message.

AUTHOR

This version released by Tye McQueen (http://perlmonks.org/?node=tye).

LICENSE

MAILING LIST

Mark-Jason still maintains a mailing list. To join a low-volume mailing list for announcements related to diff and Algorithm::Diff, send an empty mail message to mjd-perl-diff-request@plover.com.

CREDITS

Versions through 0.59 (and much of this documentation) were written by:

Mark-Jason Dominus

This version borrows some documentation and routine names from Mark-Jason's, but Diff.pm's code was completely replaced.

This code was adapted from the Smalltalk code of Mario Wolczko <mario@wolczko.com>, which is available at ftp://st.cs.uiuc.edu/pub/Smalltalk/MANCHESTER/manchester/4.0/diff.st

`sdiff` and `traverse_balanced` were written by Mike Schilli <m@perlmeister.com>.

The algorithm is that described in *A Fast Algorithm for Computing Longest Common Subsequences*, CACM, vol.20, no.5, pp.350-353, May 1977, with a few minor improvements to improve the speed.

Much work was done by Ned Konz (perl@bike-nomad.com).

The OO interface and some other changes are by Tye McQueen.