Example

1) Load all the definitions in this notebook (**Evaluation -> Evaluate Initialization Cells**).

2) Make sure MAFFT is installed in /usr/local/bin.

2) Evaluate the following example:

```
SplitTssFile[NotebookDirectory[] <> "sample.tss", 500, 1.5,
 0.0, "--genafpair --maxiterate 1000 --thread 2", 0.25, 3, True]
```

Phase detection

Split a TSS file into phases, show some visualizations, and possibly export intervals as JSON.

Parameters:
- tssPathname: path to the TSS file
- subsetSize: How many interactions to consider (“all of them” can take a long time if too many)
- mafftOP: Gap-opening penalty for MAFFT. If in doubt, use 1.5
- mafftEP: Gap-extending penalty for MAFFT. If in doubt, use 0.0
- mafftOtherArgs: Other command-line arguments to pass to MAFFT. We use “--genafpair --maxiterate 1000”.
- maxSpreadStable: Maximum variance in a column to be considered “stable”. We typically use 0.25 or so
- minColsStable: Minimum number of adjacent stable columns to constitute a stable phase. We typically use 3
- exportToJSON: use True to export intervals generated to foo.tss as foo.tss.json, and any other value not to export

```
SplitTssFile[tssPathname_, subsetSize_, mafftOP_, mafftEP_,
  mafftOtherArgs_, maxSpreadStable_, minColsStable_, exportToJSON_] :=
 Module[{uniqueValues, minVal, maxVal, colorFunction, exampleSeqs,
   plotExampleSeqs, plotWithoutAlignment, plotWithAlignment,
   plotPhases, plotPhasesShuffled, plotAlignmentConsensus},

  Print[Style[tssPathname, 16, Bold]];
  Print["Loading"];
  Print[DateString[]];
  DoLoadDataset[tssPathname];  (* this sets ias *)
  DoSubDataset[Min[{subsetSize, Length[ias]}]];
  seqs = sdSizes;
  allseqs = SizesIA /@ ias;

  (* Distinct values used anywhere in the seqs *)
```

```
uniqueValues = DeleteDuplicates@Flatten[allseqs];
{minVal, maxVal} = {Min[uniqueValues], Max[uniqueValues]};
(* Color function based on all values used by seqs *)
colorFunction = ValuesToColors[uniqueValues];

(* BEFORE ALIGNMENT *)
plotWithoutAlignment = ArrayPlot[
  Map[colorFunction, seqs, {2}],
  ImageSize → Large, Mesh → All
 ];

(* AFTER ALIGNMENT *)
Print["Aligning..."];
Print[DateString[]];
{v2a, a2vs} =
 MakeEncoding[DeleteDuplicates@Flatten@seqs, MafftaApparentlySafeChars[]];
{lbs, mtx} = RunMafft[seqs, v2a, mafftOP, mafftEP, mafftOtherArgs];
msa = SeqsUnderMtx[seqs, mtx];
plotWithAlignment = ArrayPlot[
  Map[colorFunction, msa, {2}],
  ImageSize → Large, Mesh → All
 ];

Print["Splitting into phases..."];
Print[DateString[]];
{phinds, phases} =
 MsaSplitSeqsIntoPhases[msa, allseqs, maxSpreadStable, minColsStable];
phases = DeleteEmptyPhases[phases];
Print["Splitting fun succeeded for " <>
  ToString[Length@phinds] <> " out of " <> ToString[Length@allseqs] <>
  " sequences (" <> ToString[N[100 Length@phinds/Length@allseqs]] <> "%)"];
Print[ToString[Length@allseqs - Length@phinds] <> " sequences were discarded."];
Print[DateString[]];

plotPhases = ShowPhases[phases, 100]; (* show up to 300 initial ones *)
plotPhasesShuffled = ShowPhases[Shuffle@phases, 100];
(* show up to 300 random ones *)

If[exportToJSON === True,
  jsonPathname = tssPathname <> ".json";
  Print["Exporting JSON to " <> jsonPathname];
  fromToRanges = PhasesToListsOfFromToRanges[phases];
  ExportIAsToJSON[ias, phinds, fromToRanges, jsonPathname];
```

```
   ]

   Print["Done!"];

  (* OUTPUT *)
  Column[{
    Row[{plotWithoutAlignment,
      plotWithAlignment}],
    plotPhases,
    plotPhasesShuffled
   }]
 ]
```

Parse and cache TSS files

Load interactions and their secrets from a TSS file into the **ias** and **secs** variables.

```
DoLoadDataset[tssPathname_] := Module[{},
  Print["Parsing TSS file"];
  {secs, ias} = ParseTssFileCaching[tssPathname];
  Print["Removing markers"];
  ias = Map[RemoveMarkersIA, ias];
  Print["Removing zero-byte packets"];
  ias = Map[RemoveZeroBytePacketsIA, ias];
  firstSrc = ias[[1]][[1]]["src"];
  Print["PosNeg-ing size using src=" <> firstSrc];
  ias = Map[PosNegIA[#, firstSrc] &, ias];
  Print["Done!"];
 ]
```

Parse a TSS file and return {secrets, interactions} — a list of strings and a list of Datasets.

```
ParseTssFile[pathname_] := Module[{text, chunklist, tuples, secrets, interactions},
  text = Import[pathname, "Text"];
  chunklist = StringSplit[text, "\n\n"];
  tuples = ParseTssChunk /@ chunklist;
  secrets = First /@ tuples;
  interactions = Last /@ tuples;
  Print@DescribeSecsIAs[secrets, interactions];
  {secrets, interactions}
 ]
```

This uses the global variable **tssCache** (an Association) as a tss-parsing cache that associates already-parsed TSS file names with {secs, ias} pairs. Be aware that this can use up a lot of memory if a lot of

large TSS files are parsed.

```
ParseTssFileCaching[pathname_] :=
 Module[{text, chunklist, tuples, secrets, interactions},
  (* Create an empty cache if it doesn't exist yet. *)
  If[Not@AssociationQ[tssCache], tssCache = Association[]];
  (* Have we already parsed this filename? *)
  If[KeyExistsQ[tssCache, pathname],
   {secrets, interactions} = tssCache[[pathname]];
   Print@DescribeSecsIAs[secrets, interactions];,
   (* else *)
   {secrets, interactions} = ParseTssFile[pathname];
   AssociateTo[tssCache, pathname → {secrets, interactions}];
  ];
  Return[{secrets, interactions}];
 ]
```

Clear the TSS cache.

```
ClearTssCache[] := Clear[tssCache];
```

Parse one chunk of a TSS file.

The first line contains SECRET: followed by the secret.
The second line contains comma-separated column labels.
The rest of the lines contain comma-separated values.

```
ParseTssChunk[chunk_] :=
 Module[{secretline, headerline, restcsv, secret, columnnames, matrix, packets},
  {secretline, headerline, restcsv} = StringSplit[chunk, "\n", 3];
  secret = Last@StringSplit[secretline, "SECRET:", 2];
  columnnames = StringSplit[headerline, ","];
  matrix = ImportString[restcsv, "CSV"];
  packets = Dataset[AssociationThread[columnnames, #] & /@ matrix];
  <|"secret" → secret, "packets" → packets|>
 ]
```

Describe a {secrets, interactions} pair.

```
DescribeSecsIAs[secs_, ias_] := Column[{
   "Interactions" → Length[ias],
   "Secrets" → Length[secs],
   "Unique secrets" → Length[DeleteDuplicates[secs]],
   "Min time" → MinTimeInIAs[ias],
   "Max time" → MaxTimeInIAs[ias]
  }]
```

Maximum and minimum size and time over many interactions.

```
MinTimeInIAs[ias_] := Min[Map[#[Min, "time"] &, ias]];
MaxTimeInIAs[ias_] := Max[Map[#[Max, "time"] &, ias]];

MinSizeInIAs[ias_] := Min[Map[#[Min, "size"] &, ias]];
MaxSizeInIAs[ias_] := Max[Map[#[Max, "size"] &, ias]];
```

Filter markers from an interaction

Test whether a row of an interaction is a marker.

```
MarkerQ[iaRow_] := iaRow[["sport"]] == iaRow[["dport"]] == 55 555
NonMarkerQ[iaRow_] := ¬ MarkerQ[iaRow]
```

Select the rows that are markers.

```
MarkersIA[ia_] := ia[Select[MarkerQ]]
```

Select the rows that aren't markers.

```
RemoveMarkersIA[ia_] := ia[Select[NonMarkerQ]]
```

Testing negative/positive size representation; this incorporates some direction information into the size feature.
Packets whose source IP address ends with the provided suffix become negative.

```
PosNegIA[ia_, senderSuffix_] :=
 ia[All,
  ReplacePart[#, {"size" → If[StringEndsQ[#src, senderSuffix], -#size, #size]}] &]
```

Remove zero-byte packets from an interaction.

```
RemoveZeroBytePacketsIA[ia_] := Select[ia, #["size"] > 0 &]
```

Take a random sample from a loaded dataset.
The dataset is read from the **ias** and **secs** variables.

The subdataset is written into the **sdIndices, sdIas, sdSecs, sdSizes, sdTimes** variables.

```
DoSubDataset[desiredSampleSize_] := Module[{sampleSize},
  sampleSize = Min[desiredSampleSize, Length@ias]; (* if not enough, use all *)
  sdIndices = RandomSample[Range[Length[ias]], sampleSize];
  sdIas = ias[[sdIndices]];
  sdSecs = secs[[sdIndices]];
  sdSizes = SizesIA /@ sdIas;
  sdTimes = TimesIA /@ sdIas;
  Print[ToString[sampleSize] <> " of " <> ToString[Length[ias]] <>
    " traces copied to sdIas/sdSecs/sdSizes/sdTimes"];
 ]
```

Same as above but taking a sample of adjacent interactions instead of a random sample.

```
DoSubDatasetInOrder[from_, to_] := Module[{},
  sdIndices = Range[from, to];
  sdIas = ias[[sdIndices]];
  sdSecs = secs[[sdIndices]];
  sdSizes = SizesIA /@ sdIas;
  sdTimes = TimesIA /@ sdIas;
  Print[ToString[Length[sdIndices]] <> " of " <>
    ToString[Length[ias]] <> " traces copied to sdIas/sdSecs/sdSizes/sdTimes"];
 ]
```

Shorthands to project only the vector of times or the vector of sizes of an interaction.

```
TimesIA[ia_] := Normal[ia[[All, "time"]]]
SizesIA[ia_] := Normal[ia[[All, "size"]]]
```

Multiple sequence alignment (MSA) using MAFFT

The mafft tool provides a --text mode that can handle up to 248 characters with no special (bio-specific) meaning. As of 2017 it's also a pretty robust and well-maintained tool, and the authors have been very responsive. We encode numeric values into characters and run the tool to obtain an alignment.

Given all the distinct values used, and an alphabet, return a tuple {v2a, a2vs} of associations. Each symbol in the alphabet is associated with as many values as needed to cover all values. If there are more symbols than distinct values, a bijection is created (and only the first N symbols of the alphabet are used).

```
MakeEncoding[values_, alphabet_] := Module[{
   partition, uniqueValues, nValuesPerChar, v2a, a2vs
  },
  uniqueValues = DeleteDuplicates[values];
  If[Length@uniqueValues > Length@alphabet,
   (*not enough symbols in alphabet, need to quantize*)
   nValuesPerChar = Ceiling[Length@uniqueValues / Length@alphabet];
   Print["Using " <> ToString[nValuesPerChar] <> " values per symbol"];
   partition = Partition[uniqueValues, UpTo[nValuesPerChar]],
   (*enough symbols*)
   Print["Using 1 value per symbol"];
   (*make singleton lists*)
   partition = Map[List, uniqueValues];
  ];
  (* Print["Partition: " <> ToString@partition]; *)
  a2vs = Association@Table[
     alphabet[[i]] → partition[[i]],
     {i, Length@partition}
    ];
  v2a = Association[
    Rule @@@ Flatten[
      Tuples /@ Transpose[{Values[a2vs], Map[List, Keys[a2vs]]}],
      1
     ]
   ];
  Return[{v2a, a2vs}];
 ]
```

The 248 values that mafft can supposedly handle (255 ASCII except a few ones that are illegal to use.)

```
MafftaChars[] := Module[{forbiddenCharCodes, validCharCodes, validChars},
  (* Perhaps these should be global constants? *)
  forbiddenCharCodes = {
    0 (* NULL (0x00) *),
    10 (* Line Feed (0x0a) *),
    13 (* Carriage Return (0x0d) *),
    32 (* Space (0x20) *),
    45 (* -(0x2D) *),
    60 (* <(0x3C) *),
    61 (* =(0x3D) *),
    62 (* >(0x3E) *)
   };
  validCharCodes = Complement[Range[0, 255], forbiddenCharCodes];
  validChars = Map[FromCharacterCode, validCharCodes];
  Return[validChars];
 ]
```

A subset of 187 characters that mafft should be able to handle and are printable, sorted so that regular letters appear first.

```
MafftaPrintableChars[] :=
 Join[
  CharacterRange["A", "~"],
  CharacterRange["!", ","],
  CharacterRange[".", ";"],
  CharacterRange["?", "@"],
  {"□"},
  CharacterRange["¡", "ÿ"],
  {"□"}
 ]
```

A subset of 89 characters that should almost certainly be completely safe (using this while debugging a potential encoding quirk).

```
MafftaTotallySafeChars[] :=
 Join[
  CharacterRange["!", ","],
  CharacterRange[".", ";"],
  CharacterRange["@", "~"]
 ]
```

A subset of characters that apparently are safe? (using this while debugging a potential encoding quirk).

```
MafftaApparentlySafeChars[] :=
 Join[
  CharacterRange["!", ","],
  CharacterRange[".", ";"],
  CharacterRange["@", "~"],
  CharacterRange["®", "ö"]
 ]
```

```
MakeEncoding248[seqs_] := Module[{validChars, uniqueValues},
  validChars = Chars248[];

  uniqueValues = DeleteDuplicates@Flatten@seqs;
  {v2a, a2vs} = MakeEncoding[Sort[uniqueValues], validChars];
  Return[{v2a, a2vs}];
 ]
```

Return list of encoded strings.

```
SeqsToStrings[seqs_, v2a_] := Module[{strings},
  strings = Map[StringJoin[Map[v2a, #]] &, seqs];
  Return[strings];
 ]
```

```
StringsToFASTA[strings_] := StringJoin[
  Table[
   StringJoin[">" <> ToString[i] <> "\n" <> strings[[i]] <> "\n\n"],
   {i, 1, Length@strings}
  ]
 ]

SeqsToFASTA[seqs_] := StringsToFASTA@SeqsToStrings@seqs
```

Parse standard output from MAFFT and return {labels, strings}.

NOTE: Both exporting the input and parsing the output need to use ISO Latin-1 for consistency. (No Unicode!)

```
ParseMafftOutput[pathname_] :=
 Module[{chunks, labStrs, labStrPairs, labels, strings},
  chunks = Map[StringTrim,
    StringSplit[Import[pathname, "Text", CharacterEncoding → "ISOLatin1"], ">"]
   ];
  labStrs = Map[StringSplit, chunks];
  labStrPairs = Map[
    {First[Take[#, 1]],
      StringJoin[StringTrim /@ Drop[#, 1]]} &,
    labStrs
   ];
  {labels, strings} = Transpose@labStrPairs;
  Return[{labels, strings}];
 ]
```

```
(* default without moreArgs *)
RunMafft[seqs_, v2a_, op_, ep_] := RunMafft[seqs, v2a, op, ep, ""]

(* more general version *)
RunMafft[seqs_, v2a_, op_, ep_, moreArgs_] := Module[{
    exepath = "/usr/local/bin/mafft",
    strings, fastaString,
    tempFileIn, tempFileOut, command,
    logText, resultLabels, resultStrings, resultMatrix
  },
  strings = SeqsToStrings[seqs, v2a];
  fastaString = StringsToFASTA[strings];
  tempFileIn = CreateFile[];
  tempFileOut = CreateFile[];
  Export[tempFileIn, fastaString, "Text", CharacterEncoding → "ISOLatin1"];
  command = exepath <> " "
    <> "--text" <> " "
    <> If[op === None, "", "--op " <> ToString[op] <> " "]
    <> If[ep === None, "", "--ep " <> ToString[ep] <> " "]
    <> moreArgs <> " "
    <> tempFileIn <> " "
    <> ">" <> tempFileOut;
  Print["Command: " <> command];
  Print[tempFileIn];
  logText = RunProcess[$SystemShell, "StandardError", command];
  Print[tempFileOut];
  (* Print["MAFFT log: "<>logText]; *)
  {resultLabels, resultStrings} = ParseMafftOutput[tempFileOut];
  resultMatrix = Map[Characters, resultStrings];
  (* DeleteFile[tempFileIn];
  DeleteFile[tempFileOut]; *)
  Return[{resultLabels, resultMatrix}];
 ]
```

Given a matrix of chars, return a matrix of char codes.

```
MtxToCharCodes[mtx_] := Map[Flatten@Map[ToCharacterCode, #] &, mtx]
```

Build a consensus vector based on an MSA matrix.

PEND: This version still uses the Mtx of encoded CHARS. Should we be using the MSA with the actual values?

```
MtxConsensus[mtx_] := Module[{
    height = Length@mtx,
    width = Length@First@mtx,
    column, freqs, normalizedFreqs, largestNormalizedFreq
   },
   If[DeleteDuplicates[Map[Length, mtx]] ≠ {width},
    (* Ragged matrix? Should not happen! *)
    Print["Aligned matrix is ragged! Row lengths: " <> ToString[Map[Length, mtx]]],
    Print["Aligned matrix is nice and rectangular."]
   ];
   Table[
    column = mtx[[All, colIndex]];
    freqs = Transpose[Tally[column]][[2]];
    normalizedFreqs = freqs/height;
    largestNormalizedFreq = Max[normalizedFreqs];
    largestNormalizedFreq,
    {colIndex, 1, width}
   ]
  ]
```

Given a list of sequences of values and the corresponding alignment matrix OF CHARS, return a new matrix where successive non-gap values are replaced by the corresponding sequence values (and gaps are left untouched).

This allows to convert the MSA (of encoded chars) returned by MAFFT into an MSA with the original numeric values.

NOTE: We will always get the exact original values because we remember them. But some precision may have been lost in transit (if multiple values had to be encoded as the same char). In other words, the values are correct but the MSA alignment could have been affected by the quantization of values. (Try using an extremely small alphabet and you should see the effect.)

NOTE: This assumes that gaps are the string “-“ (dash char), and replaces them with Null.

```
SeqsUnderMtx[seqs_, mtx_] := Module[{msaWithDashes},
  msaWithDashes = MapThread[SeqUnderMtxRow, {seqs, mtx}];
  Return[ReplaceAll[msaWithDashes, "-" → Null]]
 ]

SeqUnderMtxRow[seq_, mtxRow_] := Module[{rules},
  rules = Rule @@@ Transpose[{
      PositionsThatAreNot[mtxRow, "-"],
      seq
     }];
  Return[ReplacePart[mtxRow, rules]];
 ]

PositionsThatAreNot[list_, elem_] := Complement[
  Range[Length[list]], Flatten@Position[list, elem]
 ]
```

Given an MSA with original values and Nulls, compute a few additional rows of interesting stats that are relevant toward finding consensus / good splitting points.

```
MsaConsensus[msa_] := Module[{
   rows, resultVecs,
   rowWithoutGaps,
   densities, spreads
  },
  (* Let's work on columns as if they were rows (easier to index). *)
  rows = Transpose[msa];
  (* Compute a vector of interesting aggregations per row. *)
  resultVecs = Table[
    rowWithoutGaps = DeleteCases[row, Null];
    dens = Length[rowWithoutGaps] / Length[row] // N;
    spread = If[
      Length[rowWithoutGaps] > 2,
      StandardDeviation[rowWithoutGaps] // N,
      0
     ];
    {dens, spread},
    {row, rows}
   ];

  {densities, spreads} = Transpose[resultVecs];
  Return[{densities, Rescale@spreads}];
 ]
```

Given a set of values, return an association that can be applied as a function that maps values to colors.

NOTE: The value Null is treated specially, and mapped to the color white.

```
ValuesToColors[values_] := Module[{uniqueValues, colors, rules},
  uniqueValues = DeleteDuplicates[values];

  (* Special case: remove "-" if it's there *)
  uniqueValues = DeleteCases[values, Null];

  colors = Map[ColorData[54], Range[Length[uniqueValues]]];
  rules = Rule @@@ Transpose[{uniqueValues, colors}];

  (* Special case: map "-" to White *)
  rules = Append[rules, {Null → White}];

  Return[Association[rules]];
 ]
```

Which seqs do not match? (It would be good to amortize this time by returning both the matches and the non-matching indices.)

```
MsaTryToSplitSeqsIntoPhases[msa_,
  seqs_, maxSpread_Real, minLength_Integer] := Module[{
   patt, indicesOfSeqsThatDontMatch
  },
  patt = MsaToPattern[msa, maxSpread, minLength];
  Print[patt];
  Print["Running MatchQ on " <> ToString[Length@seqs] <> " seqs ..."];
  indicesOfSeqsThatDontMatch = IndicesOfElemsThatDontMatch[patt, seqs];
  Print[ToString[Length[indicesOfSeqsThatDontMatch]] <>
    " of " <> ToString[Length[seqs]] <> " seqs don't match."];
  Return[indicesOfSeqsThatDontMatch];
 ]
```

This fails if any seqs do not match (could be improved by merging with the above).

```
MsaSplitSeqsIntoPhasesRequireAllToMatch[
  msa_, seqs_, maxSpread_Real, minLength_Integer] := Module[{
   patt, splittingFun, ruleLists, listOfListsOfSubseqs
  },
  patt = MsaToPattern[msa, maxSpread, minLength];
  splittingFun = MsaPatternToSplittingFunction[patt];
  ruleLists = Map[splittingFun, seqs];
  (* hmm *)
  listOfListsOfSubseqs = Map[Last, ruleLists, {2}];
  Return[listOfListsOfSubseqs];
 ]
```

This version tolerates failures. It returns a list of indices of the seqs that matched, and a list of lists of subseqs only for those seqs that matched.

```
MsaSplitSeqsIntoPhases[msa_, seqs_, maxSpread_Real, minLength_Integer] := Module[{
    patt, splittingFun, splittingFunResults,
    listOfPairsOfIndexAndListOfListsOfSubseqs,
    listOfIndicesThatMatched, listOfListsOfSubseqs
   },
   patt = MsaToPattern[msa, maxSpread, minLength];
   splittingFun = MsaPatternToSplittingFunction[patt];
   (* These are either rule lists (if success) or integer lists (if failure) *)
   splittingFunResults = Map[splittingFun, seqs];
   listOfPairsOfIndexAndListOfListsOfSubseqs =
    MapIndexed[ProcessSplitFunnedLineAndIndex, splittingFunResults];
   (* Remove the ones that did not match *)
   listOfPairsOfIndexAndListOfListsOfSubseqs =
    DeleteCases[listOfPairsOfIndexAndListOfListsOfSubseqs, {__, None}];
   (* Transpose and return *)
   {listOfIndicesThatMatched, listOfListsOfSubseqs} =
    Transpose[listOfPairsOfIndexAndListOfListsOfSubseqs];
   Return[ {listOfIndicesThatMatched, listOfListsOfSubseqs}];
  ]

IsFirstElemRuleQ[x_List] := Head[First[x]] === Rule;
DidSplittingFunSucceed[x_List] := IsFirstElemRuleQ[x];
(* Given a result of splittingFun and the partspec of the splitfunned elem,
return a tuple {i,result} where
   i is the index and
   result is either Map[Last] onto the result of splittingFun,
if splittingFun was successful, or None if it wasn't. *)
ProcessSplitFunnedLineAndIndex[splittingFunResult_List, indexPartSpec_] := {
   First[indexPartSpec],
   If[DidSplittingFunSucceed[splittingFunResult],
    Map[Last, splittingFunResult], None]
  }
```

Given an MSA, a maximum value (between 0 and 1) for the spread (stddev) of a stable column to be considered “not too diverse”, and a minimum length for a run of stable columns to be considered a phase, return a list of lists of contiguous indices corresponding to stable phases over the MSA. For instance, something like {{4,5,6}, {10,11}, {21,22,23,24}}.

```
MsaStableParts[msa_, maxStableColumnSpread_, minStablePhaseLength_] := Module[{
   densities, spreads, listOfPairs,
   listsOfPairs, listsOfTruePairs, listsOfContiguousIndices
  },
  {densities, spreads} = MsaConsensus[msa];
  listOfPairs = Table[
    {i, densities[[i]] == 1.0 && spreads[[i]] <= maxStableColumnSpread},
    {i, Range@Length@densities}
   ];
  listsOfPairs = SplitBy[listOfPairs, Last];
  listsOfTruePairs = DeleteCases[listsOfPairs, {Repeated@{_, False}}];
  listsOfContiguousIndices = Map[First, listsOfTruePairs, {2}];
  Return[Select[listsOfContiguousIndices, Length[#] ≥ minStablePhaseLength &]];
 ]
```

Given the MSA and one list of contiguous column indices corresponding to a stable phase, return a pattern that will match it. When the column contains more than one value (assuming the spread tolerance was high enough to allow this), an Alternatives pattern object is used for that column.

```
MsaColumnsToPattern[msa_, contiguousColIndices_] := Module[{possibleValues},
  Table[
   (* We assume there can be no Nulls in these columns *)
   possibleValues = DeleteDuplicates@msa[[All, i]];
   If[Length[possibleValues] > 1,
    Alternatives @@ possibleValues, First@possibleValues],
   {i, contiguousColIndices}
  ]
 ]
```

Given an MSA, a maximum value (between 0 and 1) for the spread (stddev) of a stable column to be considered “not too diverse”, and a minimum length for a run of stable columns to be considered a phase, return one big pattern that can match a sequence similar to the ones that were aligned, where the subpatterns for stable and variable parts are named v1, s1, v2, s2, v3, etc.

This assumes that the subset of traces used in the alignment was representative and diverse enough to cover most stable phenomena (which should be pretty easy if the stable phenomena are fairly stable), and that the variable parts in non-aligned traces won’t cause too many accidental or ambiguous matches (this could become problematic if the variable parts are hugely diverse, the stable parts are very small, and thus the odds of the latter occurring accidentally within the former become too high). We should have a retry criterion for this (e.g., if it’s just a few traces, ignore them, and if it’s a lot of them, realign with the problematic traces included in the alignment).

```
MsaToPattern[msa_, maxStableColumnSpread_, minStablePhaseLength_] := Module[{
   stablePatterns,
```

```
  numStable, stableNames, namedStablePatterns,
  numWildcards, wildcardNames, namedWildcardPatterns,
  fullPattern
 },
 (* obtain an unnamed pattern for each stable part *)
 stablePatterns = Map[
   MsaColumnsToPattern[msa, #] &,
   MsaStableParts[msa, maxStableColumnSpread, minStablePhaseLength]
  ];

 (* create stable-part patterns with names like s1, s2, etc *)
 (* caution:
  this clears any global definitions to ensure these names are fresh! *)
 numStable = Length[stablePatterns];
 stableNames = Table["s" <> ToString[i], {i, numStable}];
 Apply[ClearAll, stableNames];
 stableNames = Symbol /@ stableNames;
 namedStablePatterns = MapThread[
   Pattern[#1, Apply[PatternSequence, #2]] &,
   {stableNames, stablePatterns}
  ];

 (* create wildcard patterns with names like v1, v2, etc *)
 (* caution:
  this clears any global definitions to ensure these names are fresh! *)
 (* PEND: Use ___ or __? Before and after? In between? Think about this,
 and the phase-emptiness issue. *)
 numWildcards = Length[stablePatterns] + 1;
 wildcardNames = Table["v" <> ToString[i], {i, numWildcards}];
 Apply[ClearAll, wildcardNames];
 wildcardNames = Symbol /@ wildcardNames;
 namedWildcardPatterns = Map[
   (* Pattern[#,___Integer]&, *)
   (* testing to see if we can use real numbers *)
   Pattern[#, ___] &,
   wildcardNames
  ];

 (* riffle variable and stable-phase patterns *)
 fullPattern = Riffle[namedWildcardPatterns, namedStablePatterns];
 Return[fullPattern];
]
```

```
MsaPatternToSplittingFunction[fullPattern_] :=
 Block[{partNames, outputTemplate, splittingFun},
  partNames = Map[First, fullPattern];
  outputTemplate = Map[(ToString[#] → {#}) &, partNames];
  splittingFun = ReplaceAll[fullPattern → outputTemplate];
  Return[splittingFun];
 ]
```

Given a list of lists of subseqs, remove any “column” subseq such that the whole column consists of empty subseqs. Note that when doing this, the information may be lost about which phases are variable and which are constant, so we better keep note of that elsewhere.
NOTE: To check whether all members of a list are an empty list we’re using DeleteDuplicates, which has no short-circuit and is thus inefficient for the other (non-empty) lists.

```
DeleteEmptyPhases[listOfListsOfSubseqs_] := Module[{},
  Table[
     column = listOfListsOfSubseqs[[All, i]];
     If[DeleteDuplicates[column] === {{}},
      None,
      column
     ],
     {i, 1, Length@First@listOfListsOfSubseqs}
    ] // DeleteCases[None]
   // Transpose
 ]
```

Show phases graphically.

```
ShowPhases[listOfListsOfSubseqs_, upTohowManyLines_Integer: 100] := Module[{
   howManyCols, howManyLines, uniqueValues, colorFunction
  },
  howManyCols = Length[First[listOfListsOfSubseqs]];
  howManyLines = Min[{upTohowManyLines, Length[listOfListsOfSubseqs]}];
  uniqueValues = DeleteDuplicates@Flatten[listOfListsOfSubseqs];
  colorFunction = ValuesToColors[uniqueValues];
  GraphicsRow[
   Table[
    ArrayPlot[
     Map[colorFunction,
      listOfListsOfSubseqs[[1 ;; howManyLines, i]],
      {2}
     ],
     Mesh → All,
     ImageSize → {Automatic, 500},
     ImagePadding → None
    ],
    {i, 1, howManyCols}
   ]
  ]
 ]
```

Aux function to shuffle a list.

```
Shuffle[elems_List] := RandomSample[elems, Length[elems]]
```

Given a list of lists of subseqs (i.e., a list of seqs, each of which is split into chunks), return a list of lists of {from, to} ranges.

```
PhasesToListsOfFromToRanges[phases_] := Map[OnePartitionToFromToRanges, phases]
```

Given a list of lists of values, return a list of {from, to} index ranges such that extracting each range from the concatenation of the list of lists would yield that list of lists. For example, {{50, 70}, {90, 40, 10, 22}, {44}} yields {{1, 2}, {3, 6}, {7, 7}}.

Whenever one of the lists is empty, it yields a {-1,1} range.

```
OnePartitionToFromToRanges[listOfLists_] := Module[{
   lengths = Length /@ listOfLists, pairs
  },
  pairs =
   Transpose[{Drop[Accumulate[Join[{1}, lengths]], -1], Accumulate[lengths]}];
  (* replace any {a,b} pairs where b>a with {-1,1} *)
  Return[Map[
    If[#[[1]] ≤ #[[2]], #, {-1, 1}] &,
    pairs
   ]];
 ]
```

JSON export based on {from, to} index pairs

Given a list of interaction numbers and a list of lists of {idxFrom, idxTo} pairs (one list of pairs per interaction), return a JSON string in the format expected by the Python side.

```
MakeJSONString[iaNumbers_, idxFromToLists_] := Module[{iaNumberIdxFromToListPairs},
  (* we subtract 1 from all iaNumbers
    because the Python side expects 0-based indices *)
  iaNumberIdxFromToListPairs = Transpose[{iaNumbers - 1, idxFromToLists}];
  ExportString[
   Map[
    <|"interaction_num" → First[#], "interval_list" → Last[#]|> &,
    iaNumberIdxFromToListPairs
   ],
   "RawJSON", "Compact" → 2
  ]
 ]
```

Given an interaction and a {from, to} range, return a list of idxs (original-pcap indices) of the events within the given indices.

```
SelectIdxsIA[ia_, {from_, to_}] := ia[[from ;; to, "idx"]] // Normal
```

Given a list (assumed to be sorted), return {first, last} if nonempty, and {-1, 1} if empty.

```
FirstLastOrMinusOneOne[elems_] :=
 If[Length[elems] > 0, {First@elems, Last@elems}, {-1, 1}]
```

Given:
- a list of interactions
- a list of the relevant interaction indices in some order (1-based)
- a list of lists of {from, to} tuples indicating the chosen ranges for each relevant interaction (1-based)

(must have same number of tuples for all interactions)

return a JSON string in the format expected by the Python side, with 0-based interaction numbers and the ranges expressed using idxs (original pcap-indices) instead of normal indices.

```
IAsToJSONString[allIas_, relevantIaNumbers_, rangesForEach_] :=
 Module[{listOfListsOfIdxTuples, relevantIa},
  listOfListsOfIdxTuples = Table[
    (* double indirection:
      we use the relevant IA index to look up the right ia from ias *)
    relevantIa = allIas[[relevantIaNumbers[[i]]]];
    Map[
     FirstLastOrMinusOneOne[SelectIdxsIA[relevantIa, #]] &,
     rangesForEach[[i]]
    ],
    {i, 1, Length[relevantIaNumbers]}
   ];
  Return[MakeJSONString[relevantIaNumbers, listOfListsOfIdxTuples]];
 ]
```

Same as above, exporting the JSON to a file.

```
ExportIAsToJSON[allIas_, relevantIaNumbers_, rangesForEach_, pathname_] := Export[
   pathname, IAsToJSONString[allIas, relevantIaNumbers, rangesForEach], "Text"];
```