4.0.2.1.1

Sets and Spans

- handle coordinate elements
  - clones
  - contigs
  - alignments
- easily form intersections and unions of elements
- learn about index sets

Sets, Lists and Spans

A **set** is a finite or infinite collection of objects in which order is of no significance and multiplicity is usually ignored.

{1,2,5,10}

Common operations are membership ($\in$), intersection ($\cap$), union ($\cup$), or complement ($\overline{S}$). The empty set is $\varnothing$.

A **multiset** is a set in which multiplicity is explicitly ignored.

{1,1,2,5,10}

A multiset has the additional operation of multiplicity. A **list** is an **ordered** set of elements in which an object may be another set or multiset.

A **span of a set**, S, is defined as

$$\max S - \min S$$

A **span of elements**, E, is a set of consecutive objects

$$E(a,b) = \{x \mid a \le x \le b\}$$

A **window** on the integer line, for example.

Union of multiple sets is written as

$$\bigcup_{i=1,2,3} S_i$$

An intersection of multiple sets is written as

$$\bigcap_{i,1,2,3} S_i$$

An **index set** is a set whose elements label those of another set. Here K is the index set of S.

$$S = \bigcup_{k \in K} S_k$$

CPAN's offerings

- a large number of modules implement various aspects of sets, lists, etc.
  - do not write your own implementation – use these excellent resources

> we will focus on these

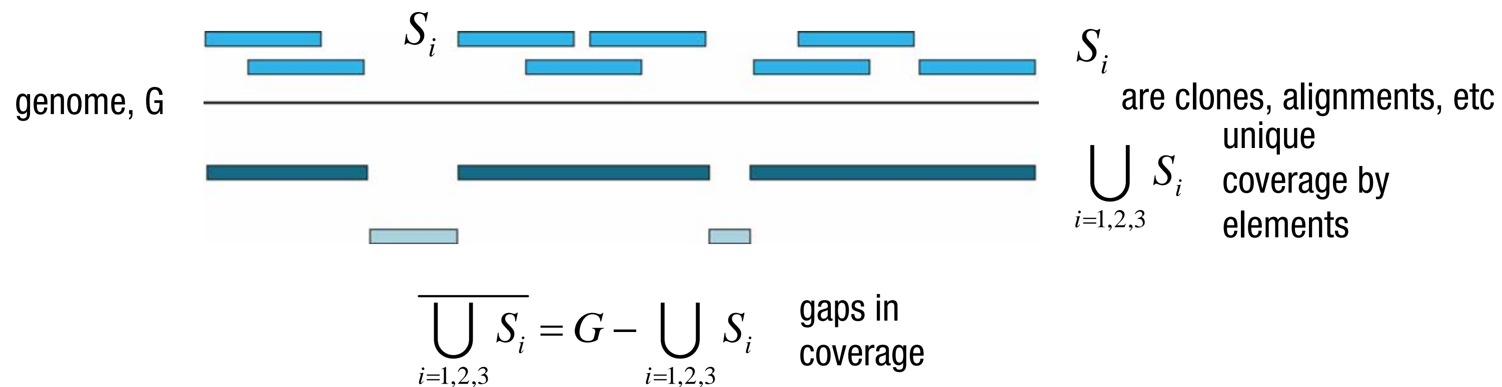
> and briefly look at these

	Module	DSLI	Description
	Set::Array	Rpd0p	Arrays as objects, with set methods
>	Set::Bag	Rpd0	Bag (multiset) class
	Set::CheckList	apdh	Maintain a list of "to-do" items
	Set::Crontab	Rpd0a	Expand crontab(5)-style integer lists
	Set::CrossProduct	Rpd0p	interact with the cartesian product of sets
	Set::Hash	Rpd0p	Hashes as objects, including set methods
	Set::Infinite	bpd0p	Infinite Set Theory module, with Date, Time
>	Set::IntRange	Rcd0p	Set of integers (arbitrary intervals, fast)
>	Set::IntSpan	Rpd0p	Set of integers newsrc style '1,5-9,11' etc
	Set::NestedGroups	Rpd0	Grouped data eg ACL's, city/state/country
	Set::Object	bcd0	Set of Objects (smalltalkish: IdentitySet)
	Set::Scalar	Mpd0p	Set of scalars (inc references)
	Set::String	Rpd0p	Strings as sets of characters
>	Set::Window	Rpd0p	Manages an interval on the integer line

search.cpan.org/modlist/Data_and_Data_Types/Set

Why You Should Care – Part I

- you work with objects that have **spatial coordinates** (alignments, clones, contigs, etc)
  - manipulate objects – intersection, union, difference
  - compute coverage, redundancy, gaps



$$\overline{\bigcup_{i=1,2,3} S_i} = G - \bigcup_{i=1,2,3} S_i$$ gaps in coverage

Why You Should Care – Part II

- you work with **indexed objects** (array probes), which may have spatial coordinates, and are interested in consecutive runs that exhibit a certain characteristic (experimental result)
  - 5 consecutive deleted array probes = putative deletion

probe index set 1 2 3 4 . . .
probe coordinate set P_1 P_2 P_3 P_4 . . .

run of 5 in index set of deleted probes

run of 6 in index set of amplified probes

- identify runs in index sets
- identify probes in runs
- extract coordinates of probes
- map runs to positions

```
D      := index set of deleted probes
R(D)   := all runs in index set
R(D,N) := all runs in index set of length N or greater
for r in R(D,N)
  # coordinate of first probe in run
  p = P(r->min)
  # coordinate of last probe in run
  q = P(r->max)
  # left position of probe run
  p->min
  # right position of probe run
  q->max
```

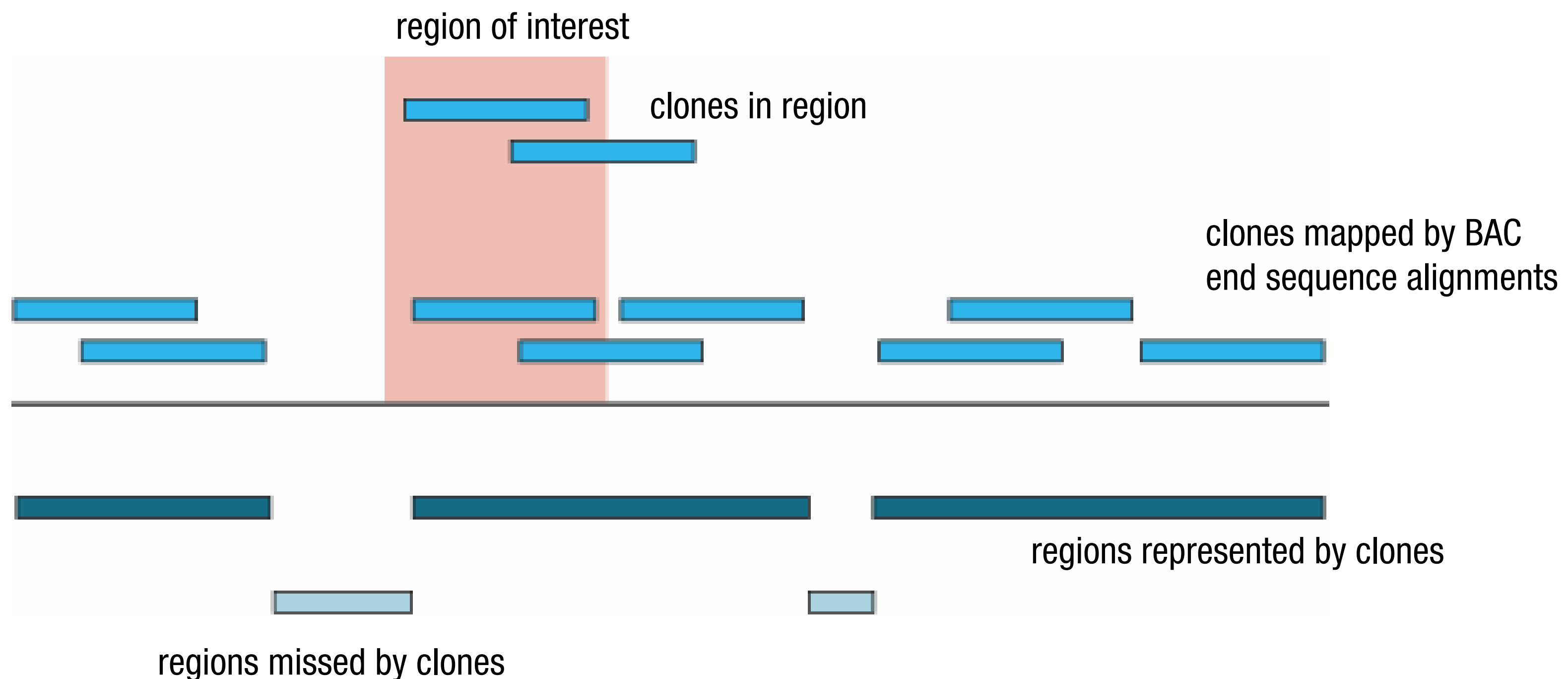
Set::IntSpan

- v1.08, Steven McDougall
- manages sets of integers, optimized for sets that have long runs of consecutive integers
  - supports infinite forms
    - (-5
    - 10-)
    - (-)
- spans operator is extremely useful in extracting runs from unions or intersections
- supports for iterators (first, last, next), comparisons (equal, equivalent, superset, subset)
- *very clean* API

```
$S = Set::IntSpan->new("1,5,10-15,20-50");
$T = Set::IntSpan->new("2-6,8-16,30-40,45");

$S->cardinality # 39
for $span ($S->spans) {
  $span->run_list     # 1  5  10-15  20-50
  $span->min          # 1  5  10     20
  $span->max          # 1  5  15     50
  $span->cardinality  # 1  1  6      31
}

$U = $S->union($T)
$U->run_list    # 1-6,8-16,20-50
$U->cardinality # 46

$V = $S->intersect($T)
$V->run_list    # 5,10-15,30-40,45
$V->cardinality # 19

$W = $S->diff($T)
$W->run_list    # 1,20-29,41-44,46-50
$W->cardinality # 20

$X = $S->union($T)->complement
$X->run_list    # (-0,7,17-19,51-)
$X->cardinality # -1
```

Set::IntSpan in Action

- I have some clones with end sequence coordinates and want to know
  - what parts of the genome to these clones represent?
  - given a genomic region, which clones lie entirely within this region? partially within the region?
  - are what are the largest “holes” in which no clones with coordinates can be found?

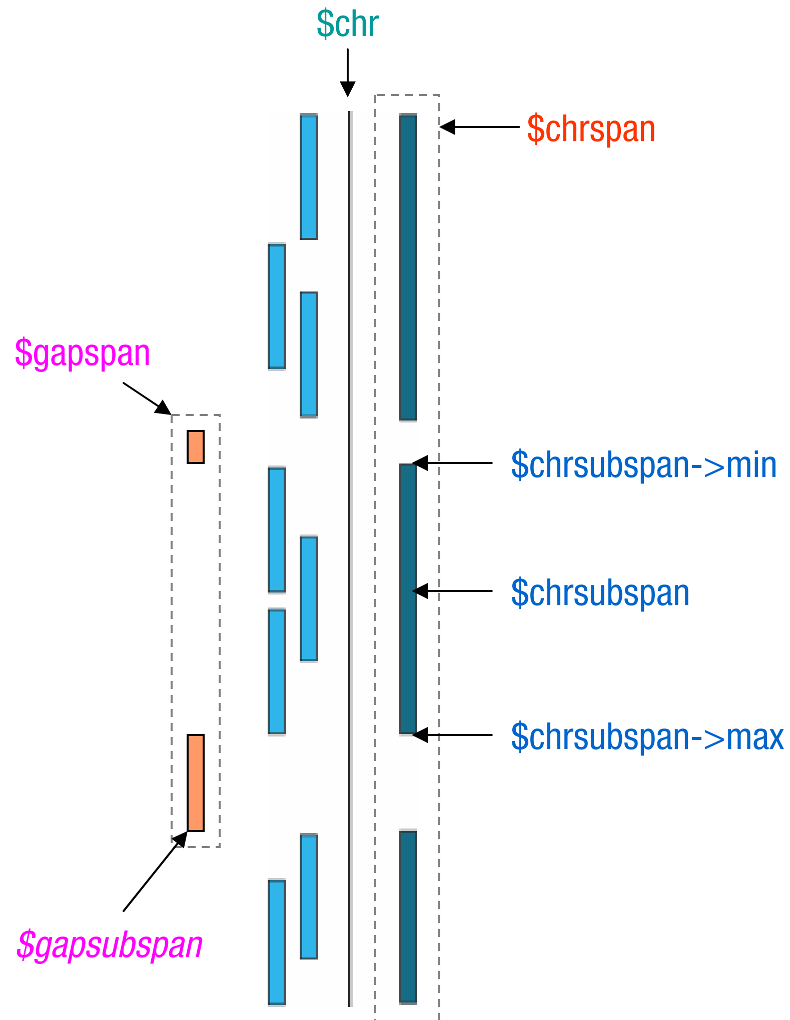


Constructing Spans from File Coordinates

- read coordinates from a file
  - construct a span for each clone
  - save the clone spans in an hash of arrays
  - construct a union of spans for each chromosome – on the fly
- $clonespans{$chr} reference to list of hashes
  - each hash stores clone name and clone span
- $chrspans{$chr} stores the union of all clone spans for a given chromosome

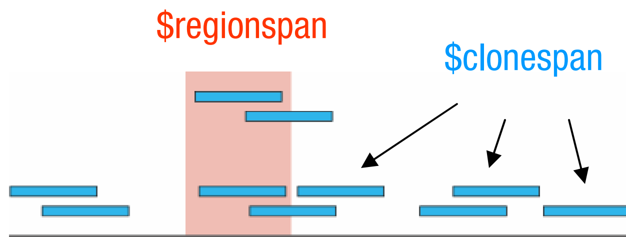
```
# clones.txt
#
# name chr start end
# RP11-2K22 1 238603586 238769410
# RP11-2K23 1 200117141 200294916
# RP11-2K24 1 63415083 63586024
#

open(F,"clones.txt");
my %chrspans;
my %clonespans;

while(<F>) {
  chomp;
  my ($clone,$chr,$start,$end) = split;

  my $clonespan = Set::IntSpan->new("$start-$end");

  $chrspans{$chr} ||= Set::IntSpan->new();

  $chrspans{$chr} = $chrspans{$chr}->union($clonespan);

  push(@{$clonespans{$chr}},
            {clone=>$clone,span=>$clonespan});
}
```

Determining Coverage by Coordinates




```
for my $chr (keys %chrspans) {
  my $chrspan = $chrspans{$chr};

  # total coverage on this chromosome
  $chrspan->cardinality;

  for my $chrsubspan ($chrspan->spans) {
    # contiguous regions of coverage
    $chrsubspan->cardinality;
    $chrsubspan->run_list;
    $chrsubspan->min;
    $chrsubspan->max;
  }

  my $entirechr = Set::IntSpan->new("1-$chrlength");
  my $gapspan = $entirechr->diff($chrspan);

  for my $gapsubspan ($gapspan->spans) {
    # regions missed by clone coverage
    $gapsubspan->cardinality;
    . . .
  }

}
```

Finding Overlapping Elements

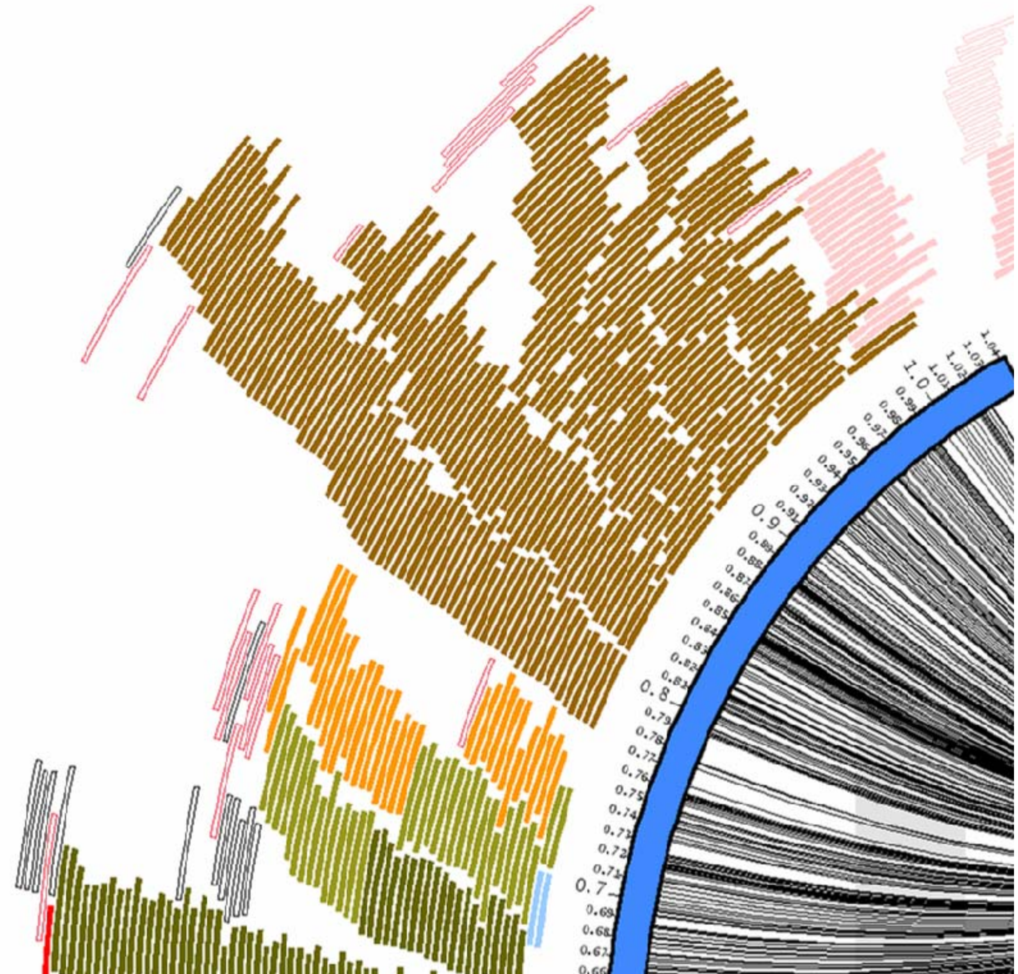


- do not test for non-empty intersection by using
  - if $a->intersect($b)
  - a span is always returned by intersect!
  - remember, you get a span object (therefore evaluates to TRUE) not the size of the span (which may be 0)
  - use
    - if $a->intersect($b)->cardinality
    - if not $a->intersect($b)->empty

```
my $regionspan = Set::IntSpan->new("$mystart-$myend");
my $regionchr = $mychr;

# do we have coverage on this chromosome?
if(exists $chrspans{$regionchr}) {

  # cycle through the clones on this chromosome
  for $clonespandata (@{$clonespans{$regionchr}}) {

    my ($clone,$clonespan)
         = @{$clonespandata}{qw(clone clonespan)};

    # intersect clone with region
    my $intersection =
       $clonespan->intersect($regionspan);

    # is the intersection non-empty?
    next unless $intersection->cardinality;

    # what fraction of the clone intersects the region?
    my $fraction = $intersection->cardinality /
                   $clonespan->cardinality;

    if ($fraction == 1) {
       # clone falls within region span
    } elsif ($fraction >= 0.5) {
       # most of clone falls within region span
    } else {
       # less than half of clone overlaps with region
    }

  }
}
```

Drawing Tilings

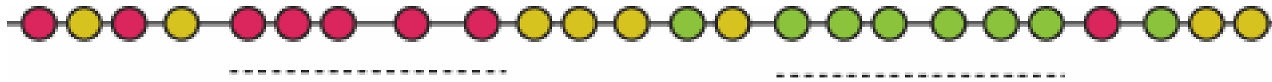
- did you ever wonder how tilings are drawn in genome browsers?
  - elements are drawn in layers, as not to overlap with one another in a given layer
- use Set::IntSpan
- set up N spans, one for each layer
- for each element to draw, find the first span, n, that does not overlap with the element
  - draw the element in layer n
  - add the element to the span,
    - span(n)->union(element)
  - you may want to pad the element to get small spacing



Index Sets

- sometimes intersect won't help you because your individual objects don't intersect (e.g. SNPs – single base pair positions)

- you are interested in consecutive runs of objects with a given characteristic



- suppose I have a collection of positions (e.g. SNPs from array)
  - each SNP has some identifier (name) and a value associated with it (-1, 0 or 1), for example.
  - let each SNP be represented by a HASH, keyed by **id**, **pos** and **value**.
  - assume all SNPs are on the same chromosome
    - if not, use a hash to store SNPs for each chromosome

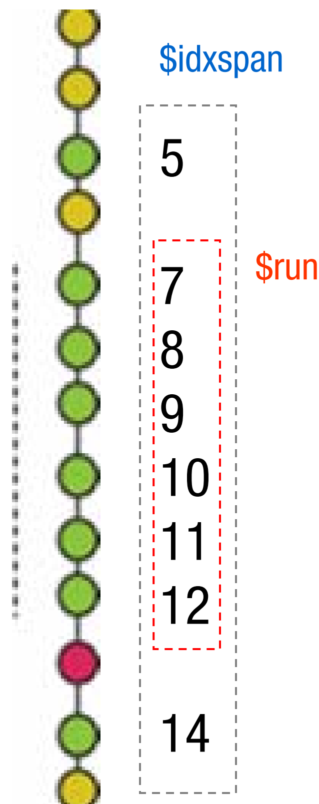
```
$snp = {id=>ID, pos=>POS, value=>VALUE}

$snp->{id}    # SNP_123
$snp->{pos}   # 23523829
$snp->{value} # 1
```

Associate Index with Each SNP

- we can’t intersect two SNP positions, since they’re single base pair coordinates
  - base pairs don’t overlap!
- neighbouring SNPs will have adjacent indeces
  - (i, i+1)
- runs of neighbouring SNPs with a given value will form a span
  - -1 SNPs
    - {1,5,6,7,8,9,20,25,28}
    - 1,5-9,20,25,28
- runs are identified by using the spans functions and testing the size of the span

```
# associate an index with each SNP,
# in order of appearance
my $idx=0;
for my $snp ( sort {$a->{pos} ⇔ $b->{pos}} @snp ) {
  $snp->{idx} = $idx++;
}

# let's make a idx-to-snp lookup table

my %idxtosnp;
map { $idxtosnp{$_->{idx}} = $_ } @snp;

# create three spans which will store index sets,
# one for each value of SNP

my @values = (-1,0,1);
my %idxspan;

map { $idxspan{$_} = Set::IntSpan->new() };

# populate each span with indexes of SNPs
# of a given value

for my $snp (@snp) {
  $idxspan{$snp->{value}}->insert($snp->{idx});
}
```

Identifying Runs of SNPs




```
# find runs of snps

for my $value (keys %idxspan) {

  # index set for a given SNP value (-1, 0, 1)
  my $idxspan = $idxspan{$value};

  # spans within index set (runs)
  for my $run ($idxspan->spans) {

    # test run size, make sure it's big enough
    my $runsize = $run->cardinality;
    next unless $runsize > 5;

    # what are the indexes in this run?
    my @runindexes = $run->elements;

    # recover SNPs in run
    my @runsnps = map { $idxtosnp{$_} } @runindexes;

    # SNP ids in run
    my @snpids = map { $_->{id} } @runsnps;

    # left and right most SNP positions
    my $leftpos = min ( map { $_->{pos} } @runsnps );
    my $rightpos = max ( map { $_->{pos} } @runsnps );

  }

}
```

Set::IntRange

- v5.1, Steffen Beyer

- this module is similar to Set::IntSpan, with additional features
  - you specify the maximum extent of your range
  - you "fill" elements with Bit_On/Bit_Off or Interval_Fill
  - overloaded operators
    - $U = $S * $T # intersection
    - $S *= $T # in-place intersection
    - $U = $S + $T # union
  - constructor takes a list, not a string
  - Norm instead of cardinality

Multiset – Grab Your Set::Bag

- v1.009, Jarkko Hietaniemi
- implements **multiset** – a set in which objects may appear more than once
- supports overloading
- use this when you want to keep track of multiplicity of elements of a given kind

```
$bag_1 = Set::Bag->new(sheep=>5,pigs=>3);
$bag_2 = Set::Bag->new(chickens=>2);

# add a sheep to bag 1
$bag_1->insert(sheep=>1);

# what animals are in bag 1?
@animals = $bag_1->elements;

# how many sheep?
$numsheep = $bag_1->grab("sheep");

# what's in the bag?
$bag_1->grab  # (sheep=>5, pigs=>3);

# eat a pig
$bag_1->delete(pig=>1);

# combine bags
$bag_1->insert($bag_2);
```

Window – Set::Window

- useful for implementing sliding windows
  - calculate GC content in 20kb sliding (by 5kb) windows

- Set::Window works similarly to Set::IntSpan, but represents a single run of consecutive integers
  - create a window using left/right position
  - move the window ($w->offset)
  - shrink the window ($w->inset(1000))
  - intersect windows ($w->intersect(@w))
    - largest window contained in $w and @w
  - union window ($w->cover(@w))
    - smallest window containing $w and @w
  - find windows inside a window ($w->series(5000))
    - get all unique windows of length 5000 within $w

Want More Data Types?

Home · Authors · Recent · News · Mirrors · FAQ · Feedback

in All CPAN Search

Archiving Compression Conversion	File Name Systems Locking	Option Parameter Config Processing
Bundles (and SDKs)	Graphics	Perl6
Commercial Software Interfaces	Internationalization Locale	Pragmas
Control Flow Utilities	Language Extensions	Security
Data and Data Types	Language Interfaces	Server Daemon Utilities
Database Interfaces	Mail and Usenet News	String Language Text Processing
Development Support	Miscellaneous	User Interfaces
Documentation	Networking Devices IPC	World Wide Web
File Handle Input/Output	Operating System Interfaces	

search. cpan. org

4.0.2.1.1

Sets and Spans

- Set::IntSpan – get to know it
- explore CPAN's **Data and Data Types** section