# Biostrings

October 5, 2010

---

`AAString-class` *AAString objects*

---

## Description

An AAString object allows efficient storage and manipulation of a long amino acid sequence.

## Details

The AAString class is a direct XString subclass (with no additional slot). Therefore all functions and methods described in the XString man page also work with an AAString object (inheritance).

Unlike the BString container that allows storage of any single string (based on a single-byte character set) the AAString container can only store a string based on the Amino Acid alphabet (see below).

## The Amino Acid alphabet

This alphabet contains all letters from the Single-Letter Amino Acid Code (see `?AMINO_ACID_CODE`) + the stop (`"*"`), the gap (`"-"`) and the hard masking (`"+"`) letters. It is stored in the `AA_ALPHABET` constant (character vector). The `alphabet` method also returns `AA_ALPHABET` when applied to an AAString object and is provided for convenience only.

## Constructor-like functions and generics

In the code snippet below, `x` can be a single string (character vector of length 1) or a BString object.

`AAString(x="", start=1, nchar=NA)`: Tries to convert `x` into an AAString object by reading `nchar` letters starting at position `start` in `x`.

## Accessor methods

In the code snippet below, `x` is an AAString object.

`alphabet(x)`: If `x` is an AAString object, then return the Amino Acid alphabet (see above). See the corresponding man pages when `x` is a BString, DNAString or RNAString object.

## Author(s)

H. Pages

### See Also

AMINO_ACID_CODE, letter, XString-class, alphabetFrequency

### Examples

```
AA_ALPHABET
a <- AAString("MARKSLEMSIR*")
length(a)
alphabet(a)
```

---

## AMINO_ACID_CODE *The Single-Letter Amino Acid Code*

---

### Description

Named character vector mapping single-letter amino acid representations to 3-letter amino acid representations.

### See Also

AAString, GENETIC_CODE

### Examples

```
## See all the 3-letter codes
AMINO_ACID_CODE

## Convert an AAString object to a vector of 3-letter amino acid codes
aa <- AAString("LANDEECQW")
AMINO_ACID_CODE[strsplit(as.character(aa), NULL)[[1]]]
```

---

## AlignedXStringSet-class *AlignedXStringSet and QualityAlignedXStringSet objects*

---

### Description

The `AlignedXStringSet` and `QualityAlignedXStringSet` classes are containers for storing an aligned `XStringSet`.

### Details

Before we define the notion of alignment, we introduce the notion of "filled-with-gaps subsequence". A "filled-with-gaps subsequence" of a string string1 is obtained by inserting 0 or any number of gaps in a subsequence of s1. For example L-A–ND and A–N-D are "filled-with-gaps subsequences" of LAND. An alignment between two strings string1 and string2 results in two strings (align1 and align2) that have the same length and are "filled-with-gaps subsequences" of string1 and string2.

For example, this is an alignment between LAND and LEAVES:

```
L-A
LEA
```

An alignment can be seen as a compact representation of one set of basic operations that transforms string1 into align1. There are 3 different kinds of basic operations: "insertions" (gaps in align1), "deletions" (gaps in align2), "replacements". The above alignment represents the following basic operations:

```
insert E at pos 2
insert V at pos 4
insert E at pos 5
replace by S at pos 6 (N is replaced by S)
delete at pos 7 (D is deleted)
```

Note that "insert X at pos i" means that all letters at a position >= i are moved 1 place to the right before X is actually inserted.

There are many possible alignments between two given strings string1 and string2 and a common problem is to find the one (or those ones) with the highest score, i.e. with the lower total cost in terms of basic operations.

## Accessor methods

In the code snippets below, `x` is a `AlignedXStringSet` or `QualityAlignedXStringSet` object.

`unaligned(x)`: The original string.

`aligned(x, degap = FALSE)`: If `degap = FALSE`, the "filled-with-gaps subsequence" representing the aligned substring. If `degap = TRUE`, the "gap-less subsequence" representing the aligned substring.

`start(x)`: The start of the aligned substring.

`end(x)`: The end of the aligned substring.

`width(x)`: The width of the aligned substring, ignoring gaps.

`indel(x)`: The positions, in the form of an `IRanges` object, of the insertions or deletions (depending on what `x` represents).

`nindel(x)`: A two-column matrix containing the length and sum of the widths for each of the elements returned by `indel`.

`length(x)`: The length of the `aligned(x)`.

`nchar(x)`: The nchar of the `aligned(x)`.

`alphabet(x)`: Equivalent to `alphabet(unaligned(x))`.

`as.character(x)`: Converts `aligned(x)` to a character vector.

`toString(x)`: Equivalent to `toString(as.character(x))`.

## Subsetting methods

`x[i]`: Returns a new `AlignedXStringSet` or `QualityAlignedXStringSet` object made of the selected elements.

`rep(x, times)`: Returns a new `AlignedXStringSet` or `QualityAlignedXStringSet` object made of the repeated elements.

### Author(s)

P. Aboyoun and H. Pages

### See Also

`pairwiseAlignment`, `PairwiseAlignedXStringSet-class`, `XStringSet-class`

### Examples

```
pattern <- AAString("LAND")
subject <- AAString("LEAVES")
nw1 <- pairwiseAlignment(pattern, subject, substitutionMatrix = "BLOSUM50", gapOpening
alignedPattern <- pattern(nw1)
unaligned(alignedPattern)
aligned(alignedPattern)
as.character(alignedPattern)
nchar(alignedPattern)
```

---

## BOC_SubjectString-class *BOC_SubjectString and BOC2_SubjectString objects*

---

### Description

The BOC_SubjectString and BOC2_SubjectString classes are experimental and might not work properly.

Please DO NOT TRY TO USE them for now. Thanks for your comprehension!

### Author(s)

H. Pages

---

## DNAString-class *DNAString objects*

---

### Description

A DNAString object allows efficient storage and manipulation of a long DNA sequence.

### Details

The DNAString class is a direct XString subclass (with no additional slot). Therefore all functions and methods described in the XString man page also work with a DNAString object (inheritance).

Unlike the BString container that allows storage of any single string (based on a single-byte character set) the DNAString container can only store a string based on the DNA alphabet (see below). In addition, the letters stored in a DNAString object are encoded in a way that optimizes fast search algorithms.

## The DNA alphabet

This alphabet contains all letters from the IUPAC Extended Genetic Alphabet (see `?IUPAC_CODE_MAP`) + the gap (`"-"`) and the hard masking (`"+"`) letters. It is stored in the `DNA_ALPHABET` constant (character vector). The `alphabet` method also returns `DNA_ALPHABET` when applied to a DNAString object and is provided for convenience only.

## Constructor-like functions and generics

In the code snippet below, `x` can be a single string (character vector of length 1), a BString object or an RNAString object.

`DNAString(x="", start=1, nchar=NA)`: Tries to convert `x` into a DNAString object by reading `nchar` letters starting at position `start` in `x`.

## Accessor methods

In the code snippet below, `x` is a DNAString object.

`alphabet(x, baseOnly=FALSE)`: If `x` is a DNAString object, then return the DNA alphabet (see above). See the corresponding man pages when `x` is a BString, RNAString or AAString object.

## Author(s)

H. Pages

## See Also

`IUPAC_CODE_MAP`, `letter`, XString-class, RNAString-class, `reverseComplement`, `alphabetFrequency`

## Examples

```
DNA_BASES
DNA_ALPHABET
d <- DNAString("TTGAAAA-CTC-N")
length(d)
alphabet(d)                 # DNA_ALPHABET
alphabet(d, baseOnly=TRUE)  # DNA_BASES
```

---

`GENETIC_CODE` *The Standard Genetic Code*

---

## Description

Two predefined objects (`GENETIC_CODE` and `RNA_GENETIC_CODE`) that represent The Standard Genetic Code.

## Usage

```
GENETIC_CODE
RNA_GENETIC_CODE
```

## Details

Formally, a genetic code is a mapping between tri-nucleotide sequences called codons, and amino acids.

The Standard Genetic Code (aka The Canonical Genetic Code, or simply The Genetic Code) is the particular mapping that encodes the vast majority of genes in nature.

`GENETIC_CODE` and `RNA_GENETIC_CODE` are predefined named character vectors that represent this mapping.

## Value

`GENETIC_CODE` and `RNA_GENETIC_CODE` are both named character vectors of length 64 (the number of all possible tri-nucleotide sequences) where each element is a single letter representing either an amino acid or the stop codon `"*"` (aka termination codon).

The names of the `GENETIC_CODE` vector are the DNA codons i.e. the tri-nucleotide sequences (directed 5' to 3') that are assumed to belong to the "coding DNA strand" (aka "sense DNA strand" or "non-template DNA strand") of the gene.

The names of the `RNA_GENETIC_CODE` are the RNA codons i.e. the tri-nucleotide sequences (directed 5' to 3') that are assumed to belong to the mRNA of the gene.

Note that the values in the `GENETIC_CODE` and `RNA_GENETIC_CODE` vectors are the same, only their names are different. The names of the latter are those of the former where all occurrences of T (thymine) have been replaced by U (uracil).

## Author(s)

H. Pages

## References

http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi

## See Also

`AA_ALPHABET`, `AMINO_ACID_CODE`, `translate`, `trinucleotideFrequency`, `DNAString`, `RNAString`, `AAString`

## Examples

```
GENETIC_CODE
GENETIC_CODE[["ATG"]]  # codon ATG is translated into M (Methionine)
sort(table(GENETIC_CODE))  # the same amino acid can be encoded by 1
                           # to 6 different codons

RNA_GENETIC_CODE
all(GENETIC_CODE == RNA_GENETIC_CODE)  # TRUE
```

## HNF4alpha *Known HNF4alpha binding sequences*

### Description

Seventy one known HNF4alpha binding sequences

### Details

A `DNAStringSet` containing 71 known binding sequences for HNF4alpha.

### Author(s)

P. Aboyoun

### References

Ellrott, K., Yang, C., Sladek, F.M., Jiang, T. (2002) "Identifying transcription factor binding sites through Markov chain optimations", Bioinformatics, 18 (Suppl. 2), S100-S109.

### Examples

```
data(HNF4alpha)
HNF4alpha
```

## IUPAC_CODE_MAP *The IUPAC Extended Genetic Alphabet*

### Description

The `IUPAC_CODE_MAP` named character vector contains the mapping from the IUPAC nucleotide ambiguity codes to their meaning.

The `mergeIUPACLetters` function provides the reverse mapping.

### Usage

```
IUPAC_CODE_MAP
mergeIUPACLetters(x)
```

### Arguments

| | |
|---|---|
| `x` | A vector of non-empty character strings made of IUPAC letters. |

### Details

IUPAC nucleotide ambiguity codes are used for representing sequences of nucleotides where the exact nucleotides that occur at some given positions are not known with certainty.

## Value

`IUPAC_CODE_MAP` is a named character vector where the names are the IUPAC nucleotide ambiguity codes and the values are their corresponding meanings. The meaning of each code is described by a string that enumarates the base letters (`"A"`, `"C"`, `"G"` or `"T"`) associated with the code.

The value returned by `mergeIUPACLetters` is an unnamed character vector of the same length as its argument `x` where each element is an IUPAC nucleotide ambiguity code.

## Author(s)

H. Pages

## References

http://www.chick.manchester.ac.uk/SiteSeer/IUPAC_codes.html

IUPAC-IUB SYMBOLS FOR NUCLEOTIDE NOMENCLATURE: Cornish-Bowden (1985) *Nucl. Acids Res.* 13: 3021-3030.

## See Also

`DNAString`, `RNAString`

## Examples

```
IUPAC_CODE_MAP
some_iupac_codes <- c("R", "M", "G", "N", "V")
IUPAC_CODE_MAP[some_iupac_codes]
mergeIUPACLetters(IUPAC_CODE_MAP[some_iupac_codes])

mergeIUPACLetters(c("Ca", "Acc", "aA", "MAAmC", "gM", "AB", "bS", "mk"))
```

---

`InDel-class` *InDel objects*

---

## Description

The `InDel` class is a container for storing insertion and deletion information.

## Details

This is a generic class that stores any insertion and deletion information.

## Accessor methods

In the code snippets below, `x` is a `InDel` object.

`insertion(x)`: The insertion information.

`deletion(x)`: The deletion information.

## Author(s)

P. Aboyoun

### See Also

`pairwiseAlignment`, `PairwiseAlignedXStringSet-class`

---

`MIndex-class` *MIndex objects*

---

### Description

The MIndex class is the basic container for storing the matches of a set of patterns in a subject sequence.

### Details

An MIndex object contains the matches (start/end locations) of a set of patterns found in an XString object called "the subject string" or "the subject sequence" or simply "the subject".

`matchPDict` function returns an MIndex object.

### Accessor methods

In the code snippets below, `x` is an MIndex object.

`length(x)`: The number of patterns that matches are stored for.

`names(x)`: The names of the patterns that matches are stored for.

`startIndex(x)`: A list containing the starting positions of the matches for each pattern.

`endIndex(x)`: A list containing the ending positions of the matches for each pattern.

`countIndex(x)`: An integer vector containing the number of matches for each pattern. Equivalent to `elementLengths(x)`.

### Subsetting methods

In the code snippets below, `x` is an MIndex object.

`x[[i]]`: Extract the matches for the i-th pattern as an IRanges object.

### Coercion

In the code snippets below, `x` is an MIndex object.

`as(x, "CompressedIRangesList")`: Turns `x` into an CompressedIRangesList object. This coercion changes `x` from one RangesList subtype to another with the underlying Ranges values remaining unchanged.

### Other utility methods and functions

In the code snippets below, `x` and `mindex` are MIndex objects and `subject` is the XString object containing the sequence in which the matches were found.

`unlist(x, recursive=TRUE, use.names=TRUE)`: Return all the matches in a single IRanges object. `recursive` and `use.names` are ignored.

`extractAllMatches(subject, mindex)`: Return all the matches in a single XStringViews object.

## Author(s)

H. Pages

## See Also

`matchPDict`, PDict-class, IRanges-class, XStringViews-class

## Examples

```
## See ?matchPDict and ?`matchPDict-inexact` for some examples.
```

---

## `MaskedXString-class` *MaskedXString objects*

---

## Description

The MaskedBString, MaskedDNAString, MaskedRNAString and MaskedAAString classes are containers for storing masked sequences.

All those containers derive directly (and with no additional slots) from the MaskedXString virtual class.

## Details

In Biostrings, a pile of masks can be put on top of a sequence. A pile of masks is represented by a MaskCollection object and the sequence by an XString object. A MaskedXString object is the result of bundling them together in a single object.

Note that, no matter what masks are put on top of it, the original sequence is always stored unmodified in a MaskedXString object. This allows the user to activate/deactivate masks without having to worry about losing the information stored in the masked/unmasked regions. Also this allows efficient memory management since the original sequence never needs to be copied (modifying it would require to make a copy of it first - sequences cannot and should never be modified in place in Biostrings), even when the set of active/inactive masks changes.

## Accessor methods

In the code snippets below, `x` is a MaskedXString object. For `masks(x)` and `masks(x) <- y`, it can also be an XString object and `y` must be `NULL` or a MaskCollection object.

`unmasked(x)`: Turns `x` into an XString object by dropping the masks.

`masks(x)`: Turns `x` into a MaskCollection object by dropping the sequence.

`masks(x) <- y`: If `x` is an XString object and `y` is `NULL`, then this doesn't do anything.

If `x` is an XString object and `y` is a MaskCollection object, then this turns `x` into a MaskedXString object by putting the masks in `y` on top of it.

If `x` is a MaskedXString object and `y` is `NULL`, then this is equivalent to `x <- unmasked(x)`.

If `x` is a MaskedXString object and `y` is a MaskCollection object, then this replaces the masks currently on top of `x` by the masks in `y`.

`alphabet(x)`: Equivalent to `alphabet(unmasked(x))`. See `?alphabet` for more information.

`length(x)`: Equivalent to `length(unmasked(x))`. See ¿`length,XString-method`‘ for more information.

## "maskedwidth" and related methods

In the code snippets below, `x` is a MaskedXString object.

`maskedwidth(x)`: Get the number of masked letters in `x`. A letter is considered masked iff it's masked by at least one active mask.

`maskedratio(x)`: Equivalent to `maskedwidth(x) / length(x)`.

`nchar(x)`: Equivalent to `length(x) - maskedwidth(x)`.

## Coercion

In the code snippets below, `x` is a MaskedXString object.

`as(x, "XStringViews")`: Turns `x` into an XStringViews object where the views are the unmasked regions of the original sequence ("unmasked" means not masked by at least one active mask).

## Other methods

In the code snippets below, `x` is a MaskedXString object.

`collapse(x)`: Collapses the set of masks in `x` into a single mask made of all active masks.

`gaps(x)`: Reverses all the masks i.e. each mask is replaced by a mask where previously unmasked regions are now masked and previously masked regions are now unmasked.

## Author(s)

H. Pages

## See Also

`maskMotif`, `injectHardMask`, `alphabetFrequency`, `reverse,MaskedXString-method`, XString-class, MaskCollection-class, XStringViews-class, Ranges-utils

## Examples

```
## ---------------------------------------------------------------------
## A. MASKING BY POSITION
## ---------------------------------------------------------------------
mask0 <- Mask(mask.width=29, start=c(3, 10, 25), width=c(6, 8, 5))
x <- DNAString("ACACAACTAGATAGNACTNNGAGAGACGC")
length(x)  # same as width(mask0)
nchar(x)   # same as length(x)
masks(x) <- mask0
x
length(x)  # has not changed
nchar(x)   # has changed
gaps(x)

## Prepare a MaskCollection object of 3 masks ('mymasks') by running the
## examples in the man page for these objects:
example(MaskCollection, package="IRanges")

## Put it on 'x':
masks(x) <- mymasks
```

```
x
alphabetFrequency(x)

## Deactivate all masks:
active(masks(x)) <- FALSE
x

## Activate mask "C":
active(masks(x))["C"] <- TRUE
x

## Turn MaskedXString object into an XStringViews object:
as(x, "XStringViews")

## Drop the masks:
masks(x) <- NULL
x
alphabetFrequency(x)


## ---------------------------------------------------------------------
## B. MASKING BY CONTENT
## ---------------------------------------------------------------------
## See ?maskMotif for masking by content
```

---

## PDict-class *PDict objects*

---

### Description

The PDict class is a container for storing a preprocessed dictionary of DNA patterns that can later be passed to the `matchPDict` function for fast matching against a reference sequence (the subject).

`PDict` is the constructor function for creating new PDict objects.

### Usage

```
PDict(x, max.mismatch=NA, tb.start=NA, tb.end=NA, tb.width=NA,
         algorithm="ACtree2", skip.invalid.patterns=FALSE)
```

### Arguments

| | |
|---|---|
| `x` | A character vector, a DNAStringSet object or an XStringViews object with a DNAString subject. |
| `max.mismatch` | A single non-negative integer or `NA`. See the "Allowing a small number of mismatching letters" section below. |
| `tb.start,tb.end,tb.width` | A single integer or `NA`. See the "Trusted Band" section below. |
| `algorithm` | `"ACtree2"` (the default) or `"Twobit"`. |
| `skip.invalid.patterns` | This argument is not supported yet (and might in fact be replaced by the `filter` argument very soon). |

## Details

THIS IS STILL WORK IN PROGRESS!

If the original dictionary `x` is a character vector or an XStringViews object with a DNAString subject, then the `PDict` constructor will first try to turn it into a DNAStringSet object.

By default (i.e. if `PDict` is called with `max.mismatch=NA`, `tb.start=NA`, `tb.end=NA` and `tb.width=NA`) the following limitations apply: (1) the original dictionary can only contain base letters (i.e. only As, Cs, Gs and Ts), therefore IUPAC ambiguity codes are not allowed; (2) all the patterns in the dictionary must have the same length ("constant width" dictionary); and (3) later `matchPdict` can only be used with `max.mismatch=0`.

A Trusted Band can be used in order to relax these limitations (see the "Trusted Band" section below).

If you are planning to use the resulting `PDict` object in order to do inexact matching where valid hits are allowed to have a small number of mismatching letters, then see the "Allowing a small number of mismatching letters" section below.

Two preprocessing algorithms are currently supported: `algorithm="ACtree2"` (the default) and `algorithm="Twobit"`. With the `"ACtree2"` algorithm, all the oligonucleotides in the Trusted Band are stored in a 4-ary Aho-Corasick tree. With the `"Twobit"` algorithm, the 2-bit-per-letter signatures of all the oligonucleotides in the Trusted Band are computed and the mapping from these signatures to the 1-based position of the corresponding oligonucleotide in the Trusted Band is stored in a way that allows very fast lookup. Only PDict objects preprocessed with the `"ACtree2"` algo can then be used with `matchPdict` (and family) and with `fixed="pattern"` (instead of `fixed=TRUE`, the default), so that IUPAC ambiguity codes in the subject are treated as ambiguities. PDict objects obtained with the `"Twobit"` algo don't allow this. See ¿`matchPDict-inexact`‘ for more information about support of IUPAC ambiguity codes in the subject.

## Trusted Band

What's a Trusted Band?

A Trusted Band is a region defined in the original dictionary where the limitations described above will apply.

Why use a Trusted Band?

Because the limitations described above will apply to the Trusted Band only! For example the Trusted Band cannot contain IUPAC ambiguity codes but the "head" and the "tail" can (see below for what those are). Also with a Trusted Band, if `matchPdict` is called with a non-null `max.mismatch` value then mismatching letters will be allowed in the head and the tail. Or, if `matchPdict` is called with `fixed="subject"`, then IUPAC ambiguity codes in the head and the tail will be treated as ambiguities.

How to specify a Trusted Band?

Use the `tb.start`, `tb.end` and `tb.width` arguments of the `PDict` constructor in order to specify a Trusted Band. This will divide each pattern in the original dictionary into three parts: a left part, a middle part and a right part. The middle part is defined by its starting and ending nucleotide positions given relatively to each pattern thru the `tb.start`, `tb.end` and `tb.width` arguments. It must have the same length for all patterns (this common length is called the width of the Trusted Band). The left and right parts are defined implicitely: they are the parts that remain before (prefix) and after (suffix) the middle part, respectively. Therefore three DNAStringSet objects result from this division: the first one is made of all the left parts and forms the head of the PDict object, the second one is made of all the middle parts and forms the Trusted Band of the PDict object, and the third one is made of all the right parts and forms the tail of the PDict object.

In other words you can think of the process of specifying a Trusted Band as drawing 2 vertical lines on the original dictionary (note that these 2 lines are not necessarily straight lines but the horizontal space between them must be constant). When doing this, you are dividing the dictionary into three regions (from left to right): the head, the Trusted Band and the tail. Each of them is a DNAStringSet object with the same number of elements than the original dictionary and the original dictionary could easily be reconstructed from those three regions.

The width of the Trusted Band must be >= 1 because Trusted Bands of width 0 are not supported.

Finally note that calling `PDict` with `tb.start=NA`, `tb.end=NA` and `tb.width=NA` (the default) is equivalent to calling it with `tb.start=1`, `tb.end=-1` and `tb.width=NA`, which results in a full-width Trusted Band i.e. a Trusted Band that covers the entire dictionary (no head and no tail).

## Allowing a small number of mismatching letters

TODO

## Accessor methods

In the code snippets below, `x` is a PDict object.

`length(x)`: The number of patterns in `x`.

`width(x)`: A vector of non-negative integers containing the number of letters for each pattern in `x`.

`names(x)`: The names of the patterns in `x`.

`head(x)`: The head of `x` or `NULL` if `x` has no head.

`tb(x)`: The Trusted Band defined on `x`.

`tb.width(x)`: The width of the Trusted Band defined on `x`. Note that, unlike `width(tb(x))`, this is a single integer. And because the Trusted Band has a constant width, `tb.width(x)` is in fact equivalent to `unique(width(tb(x)))`, or to `width(tb(x))[1]`.

`tail(x)`: The tail of `x` or `NULL` if `x` has no tail.

## Subsetting methods

In the code snippets below, `x` is a PDict object.

`x[[i]]`: Extract the i-th pattern from `x` as a DNAString object.

## Other methods

In the code snippet below, `x` is a PDict object.

`duplicated(x)`: [TODO]

`patternFrequency(x)`: [TODO]

## Author(s)

H. Pages

## References

Aho, Alfred V.; Margaret J. Corasick (June 1975). "Efficient string matching: An aid to bibliographic search". Communications of the ACM 18 (6): 333-340.

### See Also

matchPDict, DNA_ALPHABET, IUPAC_CODE_MAP, DNAStringSet-class, XStringViews-class

### Examples

```
## ---------------------------------------------------------------------
## A. NO HEAD AND NO TAIL (THE DEFAULT)
## ---------------------------------------------------------------------
library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe)
dict0                                # The original dictionary.
length(dict0)                        # Hundreds of thousands of patterns.
unique(nchar(dict0))                 # Patterns are 25-mers.

pdict0 <- PDict(dict0)               # Store the original dictionary in
                                     # a PDict object (preprocessing).
pdict0
class(pdict0)
length(pdict0)                       # Same as length(dict0).
tb.width(pdict0)                     # The width of the (implicit)
                                     # Trusted Band.
sum(duplicated(pdict0))
table(patternFrequency(pdict0))      # 9 patterns are repeated 3 times.
pdict0[[1]]
pdict0[[5]]

## ---------------------------------------------------------------------
## B. NO HEAD AND A TAIL
## ---------------------------------------------------------------------
dict1 <- c("ACNG", "GT", "CGT", "AC")
pdict1 <- PDict(dict1, tb.end=2)
pdict1
class(pdict1)
length(pdict1)
width(pdict1)
head(pdict1)
tb(pdict1)
tb.width(pdict1)
width(tb(pdict1))
tail(pdict1)
pdict1[[3]]
```

---

## PairwiseAlignedXStringSet-class
*PairwiseAlignedXStringSet, PairwiseAlignedFixedSubject, and PairwiseAlignedFixedSubjectSummary objects*

---

### Description

The `PairwiseAlignedXStringSet` class is a container for storing an elementwise pairwise alignment. The `PairwiseAlignedFixedSubject` class is a container for storing a pairwise alignment with a single subject. The `PairwiseAlignedFixedSubjectSummary` class is a container for storing the summary of an alignment.

## Usage

```
  ## Constructors:
  ## When subject is missing, pattern must be of length 2
  ## S4 method for signature 'XString,XString':
PairwiseAlignedXStringSet(pattern, subject,
    type = "global", substitutionMatrix = NULL, gapOpening = 0, gapExtension = -
  ## S4 method for signature 'XStringSet,missing':
PairwiseAlignedXStringSet(pattern, subject,
    type = "global", substitutionMatrix = NULL, gapOpening = 0, gapExtension = -
  ## S4 method for signature 'character,character':
PairwiseAlignedXStringSet(pattern, subject,
    type = "global", substitutionMatrix = NULL, gapOpening = 0, gapExtension = -
    baseClass = "BString")
  ## S4 method for signature 'character,missing':
PairwiseAlignedXStringSet(pattern, subject,
    type = "global", substitutionMatrix = NULL, gapOpening = 0, gapExtension = -
    baseClass = "BString")
```

## Arguments

| | |
|---|---|
| `pattern` | a character vector of length 1 or 2, an `XString`, or an `XStringSet` object of length 1 or 2. |
| `subject` | a character vector of length 1 or an `XString` object. |
| `type` | type of alignment. One of `"global"`, `"local"`, `"overlap"`, `"global-local"`, and `"local-global"` where `"global"` = align whole strings with end gap penalties, `"local"` = align string fragments, `"overlap"` = align whole strings without end gap penalties, `"global-local"` = align whole strings with end gap penalties on `pattern` and without end gap penalties on `subject`. `"local-global"` = align whole strings without end gap penalties on `pattern` and with end gap penalties on `subject`. |
| `substitutionMatrix` | substitution matrix for the alignment. If NULL, the diagonal values and off-diagonal values are set to 0 and 1 respectively. |
| `gapOpening` | the cost for opening a gap in the alignment. |
| `gapExtension` | the incremental cost incurred along the length of the gap in the alignment. |
| `baseClass` | the base `XString` class to use in the alignment. |

## Details

Before we define the notion of alignment, we introduce the notion of "filled-with-gaps subsequence". A "filled-with-gaps subsequence" of a string string1 is obtained by inserting 0 or any number of gaps in a subsequence of s1. For example L-A–ND and A–N-D are "filled-with-gaps subsequences" of LAND. An alignment between two strings string1 and string2 results in two strings (align1 and align2) that have the same length and are "filled-with-gaps subsequences" of string1 and string2.

For example, this is an alignment between LAND and LEAVES:

```
L-A
LEA
```

An alignment can be seen as a compact representation of one set of basic operations that transforms string1 into align1. There are 3 different kinds of basic operations: "insertions" (gaps in align1), "deletions" (gaps in align2), "replacements". The above alignment represents the following basic operations:

```
insert E at pos 2
insert V at pos 4
insert E at pos 5
replace by S at pos 6 (N is replaced by S)
delete at pos 7 (D is deleted)
```

Note that "insert X at pos i" means that all letters at a position >= i are moved 1 place to the right before X is actually inserted.

There are many possible alignments between two given strings string1 and string2 and a common problem is to find the one (or those ones) with the highest score, i.e. with the lower total cost in terms of basic operations.

## Object extraction methods

In the code snippets below, `x` is a `PairwiseAlignedXStringSet` object, except otherwise noted.

`pattern(x)`: The `AlignedXStringSet` object for the pattern.

`subject(x)`: The `AlignedXStringSet` object for the subject.

`summary(object, ...)`: Generates a summary for the `PairwiseAlignedXStringSet`.

## General information methods

In the code snippets below, `x` is a `PairwiseAlignedXStringSet` object, except otherwise noted.

`alphabet(x)`: Equivalent to `alphabet(unaligned(subject(x)))`.

`length(x)`: The length of the `aligned(pattern(x))` and `aligned(subject(x))`. There is a method for `PairwiseAlignedFixedSubjectSummary` as well.

`type(x)`: The type of the alignment (`"global"`, `"local"`, `"overlap"`, `"global-local"`, or `"local-global"`). There is a method for `PairwiseAlignedFixedSubjectSummary` as well.

## Aligned sequence methods

In the code snippets below, `x` is a `PairwiseAlignedFixedSubject` object, except otherwise noted.

`aligned(x, degap = FALSE, gapCode="-", endgapCode="-")`: If `degap = FALSE`, "align" the alignments by returning an `XStringSet` object containing the aligned patterns without insertions. If `degap = TRUE`, returns `aligned(pattern(x), degap=TRUE)`. The `gapCode` and `endgapCode` arguments denote the code in the appropriate `alphabet` to use for the internal and end gaps.

`as.character(x)`: Converts `aligned(x)` to a character vector.

`as.matrix(x)`: Returns an "exploded" character matrix representation of `aligned(x)`.

`toString(x)`: Equivalent to `toString(as.character(x))`.

## Subject position methods

In the code snippets below, `x` is a `PairwiseAlignedFixedSubject` object, except otherwise noted.

`consensusMatrix(x, as.prob=FALSE, baseOnly=FALSE, gapCode="-", endgapCode="-")` See ‘consensusMatrix‘ for more information.

`consensusString(x)` See ‘consensusString‘ for more information.

`coverage(x, shift=0L, width=NULL, weight=1L)` See ‘coverage,PairwiseAlignedFixedSubject-method‘ for more information.

`Views(subject, start=NULL, end=NULL, width=NULL, names=NULL)`: The `XStringViews` object that represents the pairwise alignments along `unaligned(subject(subject))`. The `start` and `end` arguments must be either `NULL`/`NA` or an integer vector of length 1 that denotes the offset from `start(subject(subject))`.

## Numeric summary methods

In the code snippets below, `x` is a `PairwiseAlignedXStringSet` object, except otherwise noted.

`nchar(x)`: The nchar of the `aligned(pattern(x))` and `aligned(subject(x))`. There is a method for `PairwiseAlignedFixedSubjectSummary` as well.

`insertion(x)`: An `CompressedIRangesList` object containing the locations of the insertions from the perspective of the `pattern`.

`deletion(x)`: An `CompressedIRangesList` object containing the locations of the deletions from the perspective of the `pattern`.

`indel(x)`: An `InDel` object containing the locations of the insertions and deletions from the perspective of the `pattern`.

`nindel(x)`: An `InDel` object containing the number of insertions and deletions.

`score(x)`: The score of the alignment. There is a method for `PairwiseAlignedFixedSubjectSummary` as well.

## Subsetting methods

`x[i]`: Returns a new `PairwiseAlignedXStringSet` object made of the selected elements.

`rep(x, times)`: Returns a new `PairwiseAlignedXStringSet` object made of the repeated elements.

## Author(s)

P. Aboyoun

## See Also

`pairwiseAlignment`, `AlignedXStringSet-class`, `XString-class`, `XStringViews-class`, `align-utils`, `pid`

### Examples

```
PairwiseAlignedXStringSet("-PA--W-HEAE", "HEAGAWGHE-E")
pattern <- AAStringSet(c("HLDNLKGTF", "HVDDMPNAL"))
subject <- AAString("SMDDTEKMSMKL")
nwl <- pairwiseAlignment(pattern, subject, substitutionMatrix = "BLOSUM50",
  gapOpening = -3, gapExtension = -1)
pattern(nw1)
subject(nw1)
aligned(nw1)
as.character(nw1)
as.matrix(nw1)
nchar(nw1)
score(nw1)
nw1
```

---

## QualityScaledXStringSet-class

*QualityScaledBStringSet, QualityScaledDNAStringSet, QualityScaledRNAStringSet and QualityScaledAAStringSet objects*

---

### Description

The QualityScaledBStringSet class is a container for storing a `BStringSet` object with an `XStringQuality` object.

Similarly, the QualityScaledDNAStringSet (or QualityScaledRNAStringSet, or QualityScaledAAStringSet) class is a container for storing a `DNAStringSet` (or `RNAStringSet`, or `AAStringSet`) objects with an `XStringQuality` object.

### Usage

```
## Constructors:
QualityScaledBStringSet(x, quality)
QualityScaledDNAStringSet(x, quality)
QualityScaledRNAStringSet(x, quality)
QualityScaledAAStringSet(x, quality)
```

### Arguments

| | |
|---|---|
| `x` | Either a character vector, or an XString, XStringSet or XStringViews object. |
| `quality` | An XStringQuality object. |

### Details

The `QualityScaledBStringSet`, `QualityScaledDNAStringSet`, `QualityScaledRNAStringSet` and `QualityScaledAAStringSet` functions are constructors that can be used to "naturally" turn `x` into an QualityScaledXStringSet object of the desired base type.

### Accessor methods

The QualityScaledXStringSet class derives from the XStringSet class hence all the accessor methods defined for an XStringSet object can also be used on an QualityScaledXStringSet object. Common methods include (in the code snippets below, `x` is an QualityScaledXStringSet object):

`length(x)`: The number of sequences in `x`.

`width(x)`: A vector of non-negative integers containing the number of letters for each element in `x`.

`nchar(x)`: The same as `width(x)`.

`names(x)`: `NULL` or a character vector of the same length as `x` containing a short user-provided description or comment for each element in `x`.

`quality(x)`: The quality of the strings.

### Subsetting and appending

In the code snippets below, `x` and `values` are XStringSet objects, and `i` should be an index specifying the elements to extract.

`x[i]`: Return a new QualityScaledXStringSet object made of the selected elements.

### Author(s)

P. Aboyoun

### See Also

BStringSet-class, DNAStringSet-class, RNAStringSet-class, AAStringSet-class, XStringQuality-class

### Examples

```
x1 <- DNAStringSet(c("TTGA", "CTCN"))
q1 <- PhredQuality(c("*+,-", "6789"))
qx1 <- QualityScaledDNAStringSet(x1, q1)
qx1
```

---

## `RNAString-class` *RNAString objects*

---

### Description

An RNAString object allows efficient storage and manipulation of a long RNA sequence.

### Details

The RNAString class is a direct XString subclass (with no additional slot). Therefore all functions and methods described in the XString man page also work with an RNAString object (inheritance).

Unlike the BString container that allows storage of any single string (based on a single-byte character set) the RNAString container can only store a string based on the RNA alphabet (see below). In addition, the letters stored in an RNAString object are encoded in a way that optimizes fast search algorithms.

### The RNA alphabet

This alphabet contains all letters from the IUPAC Extended Genetic Alphabet (see `?IUPAC_CODE_MAP`) where `"T"` is replaced by `"U"` + the gap (`"-"`) and the hard masking (`"+"`) letters. It is stored in the `RNA_ALPHABET` constant (character vector). The `alphabet` method also returns `RNA_ALPHABET` when applied to an RNAString object and is provided for convenience only.

### Constructor-like functions and generics

In the code snippet below, `x` can be a single string (character vector of length 1), a BString object or a DNAString object.

`RNAString(x="", start=1, nchar=NA)`: Tries to convert `x` into an RNAString object by reading `nchar` letters starting at position `start` in `x`.

### Accessor methods

In the code snippet below, `x` is an RNAString object.

`alphabet(x, baseOnly=FALSE)`: If `x` is an RNAString object, then return the RNA alphabet (see above). See the corresponding man pages when `x` is a BString, DNAString or AAString object.

### Author(s)

H. Pages

### See Also

`IUPAC_CODE_MAP`, `letter`, XString-class, DNAString-class, `reverseComplement`, `alphabetFrequency`

### Examples

```
RNA_BASES
RNA_ALPHABET
d <- DNAString("TTGAAAA-CTC-N")
r <- RNAString(d)
r
alphabet(r)                  # RNA_ALPHABET
alphabet(r, baseOnly=TRUE)   # RNA_BASES

## When comparing an RNAString object with a DNAString object,
## U and T are considered equals:
r == d  # TRUE
```

---

## `WCP` *Weighted Clustered Positions (WCP) objects*

---

### Description

The WCP class is a container for storing weighted clustered positions within XString-based strings.

### Author(s)

P. Aboyoun

### See Also

`matchWCP`, XString-class

---

## `XKeySortedData` *Data Dictionaries with XString-based Keys*

---

### Description

The XKeySortedData class is a container for storing a dictionary with XString-based keys and DataFrame (an IRanges class) values.

### Author(s)

P. Aboyoun

### See Also

XStringSet-class, DataFrame-class

---

## `XKeySortedDataList` *List of Data Dictionaries with XString-based Keys*

---

### Description

The XKeySortedDataList class is a container for storing a list of dictionaries with XString-based keys and DataFrame (an IRanges class) values.

### Author(s)

P. Aboyoun

### See Also

XKeySortedData-class, SimpleList-class

## `XString-class` *BString objects*

### Description

The BString class is a general container for storing a big string (a long sequence of characters) and for making its manipulation easy and efficient.

The DNAString, RNAString and AAString classes are similar containers but with the more biology-oriented purpose of storing a DNA sequence (DNAString), an RNA sequence (RNAString), or a sequence of amino acids (AAString).

All those containers derive directly (and with no additional slots) from the XString virtual class.

### Details

The 2 main differences between an XString object and a standard character vector are: (1) the data stored in an XString object are not copied on object duplication and (2) an XString object can only store a single string (see the XStringSet container for an efficient way to store a big collection of strings in a single object).

Unlike the DNAString, RNAString and AAString containers that accept only a predefined set of letters (the alphabet), a BString object can be used for storing any single string based on a single-byte character set.

### Constructor-like functions and generics

In the code snippet below, `x` can be a single string (character vector of length 1) or an XString object.

`BString(x="", start=1, nchar=NA)`: Tries to convert `x` into a BString object by reading `nchar` letters starting at position `start` in `x`.

### Accessor methods

In the code snippets below, `x` is an XString object.

`alphabet(x)`: `NULL` for a `BString` object. See the corresponding man pages when `x` is a DNAString, RNAString or AAString object.

`length(x)` or `nchar(x)`: Get the length of an XString object, i.e., its number of letters.

### Coercion

In the code snippets below, `x` is an XString object.

`as.character(x)`: Converts `x` to a character string.

`toString(x)`: Equivalent to `as.character(x)`.

## Subsetting

In the code snippets below, `x` is an XString object.

`x[i]`: Return a new XString object made of the selected letters (subscript `i` must be an NA-free numeric vector specifying the positions of the letters to select). The returned object belongs to the same class as `x`.

Note that, unlike `subseq`, `x[i]` does copy the sequence data and therefore will be very inefficient for extracting a big number of letters (e.g. when `i` contains millions of positions).

## Equality

In the code snippets below, `e1` and `e2` are XString objects.

`e1 == e2`: `TRUE` if `e1` is equal to `e2`. `FALSE` otherwise.

Comparison between two XString objects of different base types (e.g. a BString object and a DNAString object) is not supported with one exception: a DNAString object and an RNAString object can be compared (see RNAString-class for more details about this).

Comparison between a BString object and a character string is also supported (see examples below).

`e1 != e2`: Equivalent to `!(e1 == e2)`.

## Author(s)

H. Pages

## See Also

`subseq`, `letter`, DNAString-class, RNAString-class, AAString-class, XStringSet-class, XStringViews-class, `reverse,XString-method`

## Examples

```
b <- BString("I am a BString object")
b
length(b)

## Extracting a linear subsequence
subseq(b)
subseq(b, start=3)
subseq(b, start=-3)
subseq(b, end=-3)
subseq(b, end=-3, width=5)

## Subsetting
b2 <- b[length(b):1]       # better done with reverse(b)

as.character(b2)

b2 == b                    # FALSE
b2 == as.character(b2)     # TRUE

## b[1:length(b)] is equal but not identical to b!
b == b[1:length(b)]        # TRUE
identical(b, 1:length(b))  # FALSE
```

```
## This is because subsetting an XString object with [ makes a copy
## of part or all its sequence data. Hence, for the resulting object,
## the internal slot containing the memory address of the sequence
## data differs from the original. This is enough for identical() to
## see the 2 objects as different.
```

---

## `XStringPartialMatches-class` *XStringPartialMatches objects*

---

### Description

WARNING: This class is currently under development and might not work properly! Full documentation will come later.

Please DO NOT TRY TO USE it for now. Thanks for your comprehension!

### Accessor methods

In the code snippets below, `x` is an XStringPartialMatches object.

`subpatterns(x)`: Not ready yet.

`pattern(x)`: Not ready yet.

### Standard generic methods

In the code snippets below, `x` is an XStringPartialMatches objects, and `i` can be a numeric or logical vector.

`x[i]`: Return a new XStringPartialMatches object made of the selected views. `i` can be a numeric vector, a logical vector, `NULL` or missing. The returned object has the same subject as `x`.

### Author(s)

H. Pages

### See Also

XStringViews-class, XString-class, `letter`

`XStringQuality-class`

*PhredQuality and SolexaQuality objects*

### Description

Objects for storing string quality measures.

### Usage

```
## Constructors:
PhredQuality(x)
SolexaQuality(x)
```

### Arguments

| | |
|---|---|
| `x` | Either a character vector, BString, BStringSet, integer vector, or number vector of error probabilities. |

### Details

`PhredQuality` objects store characters that are interpreted as [0 - 99] quality measures by subtracting 33 from their ASCII decimal representation (e.g. ! = 0, " = 1, \# = 2, ...).

`SolexaQuality` objects store characters are interpreted as [-5 - 99] quality measures by subtracting 64 from their ASCII decimal representation (e.g. ; = -5, < = -4, = = -3, ...).

### Author(s)

P. Aboyoun

### See Also

`pairwiseAlignment`, PairwiseAlignedXStringSet-class, DNAString-class, BStringSet-class

### Examples

```
PhredQuality(0:40)
SolexaQuality(0:40)

PhredQuality(seq(1e-4,0.5,length=10))
SolexaQuality(seq(1e-4,0.5,length=10))
```

## `XStringSet-class` *XStringSet objects*

### Description

The BStringSet class is a container for storing a set of `BString` objects and for making its manipulation easy and efficient.

Similarly, the DNAStringSet (or RNAStringSet, or AAStringSet) class is a container for storing a set of `DNAString` (or `RNAString`, or `AAString`) objects.

All those containers derive directly (and with no additional slots) from the XStringSet virtual class.

### Usage

```
  ## Constructors:
  BStringSet(x=character(), start=NA, end=NA, width=NA, use.names=TRUE)
  DNAStringSet(x=character(), start=NA, end=NA, width=NA, use.names=TRUE)
  RNAStringSet(x=character(), start=NA, end=NA, width=NA, use.names=TRUE)
  AAStringSet(x=character(), start=NA, end=NA, width=NA, use.names=TRUE)

  ## Accessor-like methods:
  ## S4 method for signature 'character':
width(x)
  ## S4 method for signature 'XStringSet':
nchar(x, type="chars", allowNA=FALSE)

  ## ... and more (see below)
```

### Arguments

| | |
|---|---|
| `x` | Either a character vector (with no NAs), or an XString, XStringSet or XStringViews object. |
| `start,end,width` | Either `NA`, a single integer, or an integer vector of the same length as `x` specifying how `x` should be "narrowed" (see `?narrow` for the details). |
| `use.names` | `TRUE` or `FALSE`. Should names be preserved? |
| `type,allowNA` | Ignored. |

### Details

The `BStringSet`, `DNAStringSet`, `RNAStringSet` and `AAStringSet` functions are constructors that can be used to "naturally" turn `x` into an XStringSet object of the desired base type.

They also allow the user to "narrow" the sequences contained in `x` via proper use of the `start`, `end` and/or `width` arguments. In this context, "narrowing" means dropping a prefix or/and a suffix of each sequence in `x`. The "narrowing" capabilities of these constructors can be illustrated by the following property: if `x` is a character vector (with no NAs), or an XStringSet (or XStringViews) object, then the 3 following transformations are equivalent:

```
BStringSet(x, start=mystart, end=myend, width=mywidth)
subseq(BStringSet(x), start=mystart, end=myend, width=mywidth)
```

```
BStringSet(subseq(x, start=mystart, end=myend, width=mywidth))
```

Note that, besides being more convenient, the first form is also more efficient on character vectors.

### Accessor-like methods

In the code snippets below, `x` is an XStringSet object.

`length(x)`: The number of sequences in `x`.

`width(x)`: A vector of non-negative integers containing the number of letters for each element in `x`. Note that `width(x)` is also defined for a character vector with no NAs and is equivalent to `nchar(x, type="bytes")`.

`names(x)`: `NULL` or a character vector of the same length as `x` containing a short user-provided description or comment for each element in `x`. These are the only data in an XStringSet object that can safely be changed by the user. All the other data are immutable! As a general recommendation, the user should never try to modify an object by accessing its slots directly.

`alphabet(x)`: Return `NULL`, `DNA_ALPHABET`, `RNA_ALPHABET` or `AA_ALPHABET` depending on whether `x` is a BStringSet, DNAStringSet, RNAStringSet or AAStringSet object.

`nchar(x)`: The same as `width(x)`.

### Subsequence extraction and related transformations

In the code snippets below, `x` is a character vector (with no NAs), or an XStringSet (or XStringViews) object.

`subseq(x, start=NA, end=NA, width=NA)`: Applies `subseq` on each element in `x`. See `?subseq` for the details.

Note that this is similar to what `substr` does on a character vector. However there are some noticeable differences:

(1) the arguments are `start` and `stop` for `substr`;

(2) the SEW interface (start/end/width) interface of `subseq` is richer (e.g. support for negative start or end values); and (3) `subseq` checks that the specified start/end/width values are valid i.e., unlike `substr`, it throws an error if they define "out of limits" subsequences or subsequences with a negative width.

`narrow(x, start=NA, end=NA, width=NA, use.names=TRUE)`: Same as `subseq`. The only differences are: (1) `narrow` has a `use.names` argument; and (2) all the things `narrow` and `subseq` work on (IRanges, XStringSet or XStringViews objects for `narrow`, XVector or XStringSet objects for `subseq`). But they both work and do the same thing on an XStringSet object.

`threebands(x, start=NA, end=NA, width=NA)`: Like the method for IRanges objects, the `threebands` methods for character vectors and XStringSet objects extend the capability of `narrow` by returning the 3 set of subsequences (the left, middle and right subsequences) associated to the narrowing operation. See `?threebands` in the IRanges package for the details.

`subseq(x, start=NA, end=NA, width=NA) <- value`: A vectorized version of the `subseq<-` method for XVector objects. See ¿`subseq<-`' for the details.

### Compacting

In the code snippets below, `x` is an XStringSet object.

`compact(x, basetype=NULL)`: Makes a deep copy of `x` that reduces its memory footprint. Typically used before saving `x` to a file (serialization).

## Subsetting and appending

In the code snippets below, `x` and `values` are XStringSet objects, and `i` should be an index specifying the elements to extract.

`x[i]`: Return a new XStringSet object made of the selected elements.

`x[[i]]`: Extract the i-th `XString` object from `x`.

`append(x, values, after=length(x))`: Add sequences in `values` to `x`.

## Ordering and related methods

In the code snippets below, `x` is an XStringSet object.

`is.unsorted(x, strictly=FALSE)`: Return a logical values specifying if `x` is unsorted. The `strictly` argument takes logical value indicating if the check should be for _strictly_ increasing values.

`order(x)`: Return a permutation which rearranges `x` into ascending or descending order.

`sort(x)`: Sort `x` into ascending order (equivalent to `x[order(x)]`).

`rank(x)`: Rank `x` in ascending order.

## Duplicated and unique methods

In the code snippets below, `x` is an XStringSet object.

`duplicated(x)`: Return a logical vector whose elements denotes duplicates in `x`.

`unique(x)`: Return an XStringSet containing the unique values in `x`.

## Set operations

In the code snippets below, `x` and `y` are XStringSet objects

`union(x, y)`: Union of `x` and `y`.

`intersect(x, y)`: Intersection of `x` and `y`.

`setdiff(x, y)`: Asymmetric set difference of `x` and `y`.

`setequal(x, y)`: Set equality of `x` to `y`.

## Identical value matching

In the code snippets below, `x` is a character vector, XString, or XStringSet object and `table` is an XStringSet object.

`x %in% table`: Returns a logical vector indicating which elements in `x` match identically with an element in `table`.

`match(x, table, nomatch = NA_integer_, incomparables = NULL)`: Returns an integer vector containing the first positions of an identical match in `table` for the elements in `x`.

## Other methods

In the code snippets below, `x` is an XStringSet object.

`unlist(x)`: Turns `x` into an XString object by combining the sequences in `x` together. Fast equivalent to `do.call(c, as.list(x))`.

`as.character(x, use.names)`: Convert `x` to a character vector of the same length as `x`. `use.names` controls whether or not `names(x)` should be used to set the names of the returned vector (default is `TRUE`).

`as.matrix(x, use.names)`: Return a character matrix containing the "exploded" representation of the strings. This can only be used on an XStringSet object with equal-width strings. `use.names` controls whether or not `names(x)` should be used to set the row names of the returned matrix (default is `TRUE`).

`toString(x)`: Equivalent to `toString(as.character(x))`.

## Author(s)

H. Pages

## See Also

XString-class, XStringViews-class, XStringSetList-class, `subseq`, `narrow`, `substr`

## Examples

```
## ---------------------------------------------------------------------
## A. USING THE XStringSet CONSTRUCTORS ON A CHARACTER VECTOR
## ---------------------------------------------------------------------
## Note that there is no XStringSet() constructor, but an XStringSet
## family of constructors: BStringSet(), DNAStringSet(), RNAStringSet(),
## etc...
x0 <- c("#CTC-NACCAGTAT", "#TTGA", "TACCTAGAG")
width(x0)
x1 <- BStringSet(x0)
x1

## 3 equivalent ways to obtain the same BStringSet object:
BStringSet(x0, start=4, end=-3)
subseq(x1, start=4, end=-3)
BStringSet(subseq(x0, start=4, end=-3))

dna0 <- DNAStringSet(x0, start=4, end=-3)
dna0
names(dna0)
names(dna0)[2] <- "seqB"
dna0

## ---------------------------------------------------------------------
## B. USING THE XStringSet CONSTRUCTORS ON A SINGLE SEQUENCE (XString
##    OBJECT OR CHARACTER STRING)
## ---------------------------------------------------------------------
x2 <- "abcdefghij"
BStringSet(x2, start=2, end=6:2)  # behaves like 'substring(x2, 2, 6:2)'
BStringSet(x2, start=-(1:6))
x3 <- BString(x2)
```

```
BStringSet(x3, end=-(1:6), width=3)

## Randomly extract 1 million 40-mers from C. elegans chrI:
extractRandomReads <- function(subject, nread, readlength)
{
    if (!is.integer(readlength))
        readlength <- as.integer(readlength)
    start <- sample(length(subject) - readlength + 1L, nread,
                    replace=TRUE)
    DNAStringSet(subject, start=start, width=readlength)
}
library(BSgenome.Celegans.UCSC.ce2)
rndreads <- extractRandomReads(Celegans$chrI, 1000000, 40)
## Notes:
## - This takes only 2 or 3 seconds versus several hours for a solution
##   using substring() on a standard character string.
## - The short sequences in 'rndreads' can be seen as the result of a
##   simulated high-throughput sequencing experiment. A non-realistic
##   one though because:
##     (a) It assumes that the underlying technology is perfect (the
##         generated reads have no technology induced errors).
##     (b) It assumes that the sequenced genome is exactly the same as the
##         reference genome.
##     (c) The simulated reads can contain IUPAC ambiguity letters only
##         because the reference genome contains them. In a real
##         high-throughput sequencing experiment, the sequenced genome
##         of course doesn't contain those letters, but the sequencer
##         can introduce them in the generated reads to indicate ambiguous
##         base-calling.
##     (d) The simulated reads come from the plus strand only of a single
##         chromosome.
## - See the getSeq() function in the BSgenome package for how to
##   circumvent (d) i.e. how to generate reads that come from the whole
##   genome (plus and minus strands of all chromosomes).

## ---------------------------------------------------------------------
## C. USING THE XStringSet CONSTRUCTORS ON AN XStringSet OBJECT
## ---------------------------------------------------------------------
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
probes

RNAStringSet(probes, start=2, end=-5)  # does NOT copy the sequence data!

## ---------------------------------------------------------------------
## D. USING subseq() ON AN XStringSet OBJECT
## ---------------------------------------------------------------------
subseq(probes, start=2, end=-5)

subseq(probes, start=13, end=13) <- "N"
probes

## Add/remove a prefix:
subseq(probes, start=1, end=0) <- "--"
probes
subseq(probes, end=2) <- ""
probes
```

```
## Do more complicated things:
subseq(probes, start=4:7, end=7) <- c("YYYY", "YYY", "YY", "Y")
subseq(probes, start=4, end=6) <- subseq(probes, start=-2:-5)
probes

## ---------------------------------------------------------------------
## E. COMPACTING AN XStringSet OBJECT
## ---------------------------------------------------------------------
## Compacting is done typically before serialization.
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
object.size(probes)
y1 <- subseq(probes[1:12], start=5)
object.size(y1)
file1 <- file.path(tempdir(), "y1.rda")
save(y1, file=file1)
file.info(file1)$size
y2 <- compact(y1)
object.size(y2)  # much smaller!
file2 <- file.path(tempdir(), "y2.rda")
save(y2, file=file2)
file.info(file2)$size

## ---------------------------------------------------------------------
## F. UNLISTING AN XStringSet OBJECT
## ---------------------------------------------------------------------
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
unlist(probes)
```

---

`XStringSet-io` *Read/write an XStringSet or XStringViews object from/to a file*

---

## Description

Functions to read/write an XStringSet or XStringViews object from/to a file.

## Usage

```
## Read FASTA (or FASTQ) files in an XStringSet object:
read.BStringSet(filepath, format="fasta")
read.DNAStringSet(filepath, format="fasta")
read.RNAStringSet(filepath, format="fasta")
read.AAStringSet(filepath, format="fasta")

## Extract basic information about FASTA (or FASTQ) files
## without loading them:
fasta.info(filepath, use.descs=TRUE)
fastq.geometry(filepath)

## Write an XStringSet object to a FASTA (or FASTQ) file:
write.XStringSet(x, file="", append=FALSE, format="fasta", width=80)
```

```
## Serialize an XStringSet object:
save.XStringSet(x, objname, dirpath=".", save.dups=FALSE, verbose=TRUE)

## Some legacy stuff:
read.XStringViews(filepath, format="fasta", subjectClass, collapse="")
write.XStringViews(x, file="", append=FALSE, format="fasta", width=80)
FASTArecordsToCharacter(FASTArecs, use.names=TRUE)
CharacterToFASTArecords(x)
FASTArecordsToXStringViews(FASTArecs, subjectClass, collapse="")
XStringSetToFASTArecords(x)
```

## Arguments

| | |
|---|---|
| `filepath` | A character vector containing the paths to the input files. |
| `format` | Either `"fasta"` (the default) or `"fastq"`. Note that `write.XStringSet` and `write.XStringViews` only support `"fasta"` for now. |
| `use.descs` | Should the returned vector be named with the description lines found in the FASTA records? |
| `x` | For `write.XStringSet` and `write.XStringViews`, the object to write to `file`. For `CharacterToFASTArecords`, the (possibly named) character vector to be converted to a list of FASTA records as one returned by `readFASTA`. For `XStringSetToFASTArecords`, the XStringSet object to be converted to a list of FASTA records as one returned by `readFASTA`. |
| `file` | A connection, or a character string naming the file to write to. If `""` (the default), print to the standard output connection (generally the console) unless redirected by `sink`. |
| `append` | `TRUE` or `FALSE`. If `TRUE` output will be appended to `file`; otherwise, it will overwrite the contents of `file`. See `?cat` for the details. |
| `width` | Only relevant if `format` is `"fasta"`. The maximum number of letters per line of sequence. |
| `objname` | The name of the serialized object. |
| `dirpath` | The path to the directory where to save the serialized object. |
| `save.dups` | `TRUE` or `FALSE`. If `TRUE` then the `Dups` object describing how duplicated elements in `x` are related to each other is saved too. For advanced users only. |
| `verbose` | `TRUE` or `FALSE`. |
| `subjectClass` | The class to be given to the subject of the XStringViews object created and returned by the function. Must be the name of one of the direct XString subclasses i.e. `"BString"`, `"DNAString"`, `"RNAString"` or `"AAString"`. |
| `collapse` | An optional character string to be inserted between the views of the XStringViews object created and returned by the function. |
| `FASTArecs` | A list of FASTA records as one returned by `readFASTA`. |
| `use.names` | Whether or not the description line preceding each FASTA records should be used to set the names of the returned object. |

## Details

Only FASTA and FASTQ files are supported for now. The identifiers and qualities stored in the FASTQ records are ignored (only the sequences are returned).

Reading functions `read.BStringSet`, `read.DNAStringSet`, `read.RNAStringSet`, `read.AAStringSet` and `read.XStringViews` load sequences from an input file (or set of input files) into an XStringSet or XStringViews object. (Note that for now `read.XStringViews` can only read 1 FASTA file at a time but this will be addressed ASAP). When multiple input files are specified, they are read in the corresponding order and their data are stored in the returned object in that order. Note that when multiple input FASTQ files are specified, they must all have the same "width" (i.e. all their sequences must have the same length).

The `fasta.info` utility returns an integer vector with one element per FASTA record in the input files. Each element is the length of the sequence found in the corresponding record. If `use.descs` is `TRUE` (the default) then the returned vector is named with the description lines found in the FASTA records.

The `fastq.geometry` utility returns an integer vector describing the "geometry" of the FASTQ files i.e. a vector of length 2 where the first element is the total number of FASTQ records in the files and the second element the common "width" of these files (this width is `NA` if the files contain no FASTQ records or records with different "widths").

Writing functions `write.XStringSet` and `write.XStringViews` write an XStringSet or XStringViews object to a file or connection. They only support the FASTA format for now.

Serializing an XStringSet object with `save.XStringSet` is equivalent to using the standard `save` mechanism. But it will try to reduce the size of `x` in memory first before calling `save`. Most of the times this leads to a much reduced size on disk.

`FASTArecordsToCharacter`, `CharacterToFASTArecords`, `FASTArecordsToXStringViews` and `XStringSetToFASTArecords` are helper functions used internally by `write.XStringSet` and `read.XStringViews` for switching between different representations of the same object.

## See Also

`readFASTA`, `writeFASTA`, XStringSet-class, XStringViews-class, BString-class, DNAString-class, RNAString-class, AAString-class

## Examples

```
## ---------------------------------------------------------------------
## A. READ/WRITE FASTA FILES
## ---------------------------------------------------------------------
filepath <- system.file("extdata", "someORF.fa", package="Biostrings")
fasta.info(filepath)
x <- read.DNAStringSet(filepath)
x
write.XStringSet(x)  # writes to the console

## ---------------------------------------------------------------------
## B. READ FASTQ FILES
## ---------------------------------------------------------------------
filepath <- system.file("extdata", "s_1_sequence.txt", package="Biostrings")
fastq.geometry(filepath)
## Only the FASTQ sequences are returned (identifiers and qualities
## are dropped):
read.DNAStringSet(filepath, format="fastq")
```

```
## ---------------------------------------------------------------------
## C. SERIALIZATION
## ---------------------------------------------------------------------
library(BSgenome.Celegans.UCSC.ce2)
## Create a "sliding window" on chr I:
sw_start <- seq.int(1, length(Celegans$chrI)-50, by=50)
sw <- Views(Celegans$chrI, start=sw_start, width=10)
my_fake_shortreads <- as(sw, "XStringSet")
save.XStringSet(my_fake_shortreads, "my_fake_shortreads", dirpath=tempdir())

## ---------------------------------------------------------------------
## D. SOME RELATED HELPER FUNCTIONS
## ---------------------------------------------------------------------
## Converting 'x'...
## ... to a list of FASTA records (as one returned by the "readFASTA" function)
x1 <- XStringSetToFASTArecords(x)
## ... to a named character vector
x2 <- FASTArecordsToCharacter(x1) # same as 'as.character(x)'
```

---

`XStringSetList-class` *XStringSetList objects*

---

## Description

The XStringSetList class is a virtual container for storing a list of XStringSet objects.

## Details

Concrete flavors of the XStringSetList container are the BStringSetList, DNAStringSetList, RNAStringSetList and AAStringSetList containers for storing a list of BStringSet, DNAStringSet, RNAStringSet and AAStringSet objects, respectively. These four containers are direct subclasses of XStringSetList with no additional slots.

## Methods

TODO

## Author(s)

H. Pages

## See Also

XStringSet-class, Grouping-class, Sequence-class

## Examples

```
unlisted <- DNAStringSet(c("AAA", "AC", "GGATA"))
partitioning <- PartitioningByEnd(c(0, 2, 2, 3))
x <- new("DNAStringSetList",
         unlisted=unlisted,
         partitioning=partitioning)
x
```

```
length(x)
unlist(x)
x[[1]]
x[[2]]
as.list(x)

names(x) <- LETTERS[1:4]
x[["A"]]
x[["B"]]
as.list(x)  # named list

## Using the Grouping core API on 'partitioning(x)':
partitioning(x)
length(partitioning(x))
nobj(partitioning(x))
grouplength(partitioning(x))  # same as 'unname(sapply(x, length))'

## Using the Ranges core API on 'partitioning(x)':
start(partitioning(x))
end(partitioning(x))
width(partitioning(x))  # same as 'grouplength(partitioning(x))'
```

---

## `XStringViews-class` *The XStringViews class*

---

### Description

The XStringViews class is the basic container for storing a set of views (start/end locations) on the same sequence (an XString object).

### Details

An XStringViews object contains a set of views (start/end locations) on the same XString object called "the subject string" or "the subject sequence" or simply "the subject". Each view is defined by its start and end locations: both are integers such that start <= end. An XStringViews object is in fact a particular case of an Views object (the XStringViews class contains the Views class) so it can be manipulated in a similar manner: see `?Views` for more information. Note that two views can overlap and that a view can be "out of limits" i.e. it can start before the first letter of the subject or/and end after its last letter.

### Constructor

`Views(subject, start=NULL, end=NULL, width=NULL, names=NULL)`: See `?Views` in the IRanges package for the details.

### Accessor-like methods

All the accessor-like methods defined for `Views` objects work on XStringViews objects. In addition, the following accessors are defined for XStringViews objects:

`nchar(x)`: A vector of non-negative integers containing the number of letters in each view. Values in `nchar(x)` coincide with values in `width(x)` except for "out of limits" views where they are lower.

## Other methods

In the code snippets below, `x`, `object`, `e1` and `e2` are XStringViews objects, and `i` can be a numeric or logical vector.

`e1 == e2`: A vector of logicals indicating the result of the view by view comparison. The views in the shorter of the two XStringViews object being compared are recycled as necessary.

Like for comparison between XString objects, comparison between two XStringViews objects with subjects of different classes is not supported with one exception: when the subjects are DNAString and RNAString instances.

Also, like with XString objects, comparison between an XStringViews object with a BString subject and a character vector is supported (see examples below).

`e1 != e2`: Equivalent to `!(e1 == e2)`.

`as.character(x, use.names, check.limits)`: Convert `x` to a character vector of the same length as `x`. `use.names` controls whether or not `names(x)` should be used to set the names of the returned vector (default is `TRUE`). `check.limits` controls whether or not an error should be raised if `x` contains "out of limit" views (default is `TRUE`). With `check.limits=FALSE` then "out of limit" views are padded with spaces.

`as.matrix(x, mode, use.names, check.limits)`: Depending on what `mode` is chosen (`"integer"` or `"character"`), return either a 2-column integer matrix containing `start(x)` and `end(x)` or a character matrix containing the "exploded" representation of the views. `mode="character"` can only be used on an XStringViews object with equal-width views. Arguments `use.names` and `check.limits` are ignored with `mode="integer"`. With `mode="character"`, `use.names` controls whether or not `names(x)` should be used to set the row names of the returned matrix (default is `TRUE`), and `check.limits` controls whether or not an error should be raised if `x` contains "out of limit" views (default is `TRUE`). With `check.limits=FALSE` then "out of limit" views are padded with spaces.

`toString(x)`: Equivalent to `toString(as.character(x))`.

## Author(s)

H. Pages

## See Also

Views-class, `gaps`, XStringViews-constructors, XString-class, XStringSet-class, `letter`, MIndex-class

## Examples

```
## One standard way to create an XStringViews object is to use
## the Views() constructor.

## Views on a DNAString object:
s <- DNAString("-CTC-N")
v4 <- Views(s, start=3:0, end=5:8)
v4
subject(v4)
length(v4)
start(v4)
end(v4)
width(v4)
```

```
## Attach a comment to views #3 and #4:
names(v4)[3:4] <- "out of limits"
names(v4)

## A more programatical way to "tag" the "out of limits" views:
names(v4)[start(v4) < 1 | nchar(subject(v4)) < end(v4)] <- "out of limits"
## or just:
names(v4)[nchar(v4) < width(v4)] <- "out of limits"

## Two equivalent ways to extract a view as an XString object:
s2a <- v4[[2]]
s2b <- subseq(subject(v4), start=start(v4)[2], end=end(v4)[2])
identical(s2a, s2b) # TRUE

## It is an error to try to extract an "out of limits" view:
#v4[[3]] # Error!

v12 <- Views(DNAString("TAATAATG"), start=-2:9, end=0:11)
v12 == DNAString("TAA")
v12[v12 == v12[4]]
v12[v12 == v12[1]]
v12[3] == Views(RNAString("AU"), start=0, end=2)

## Here the first view doesn't even overlap with the subject:
Views(BString("aaa--b"), start=-3:4, end=-3:4 + c(3:6, 6:3))

## 'start' and 'end' are recycled:
subject <- "abcdefghij"
Views(subject, start=2:1, end=4)
Views(subject, start=5:7, end=nchar(subject))
Views(subject, start=1, end=5:7)

## Applying gaps() to an XStringViews object:
v2 <- Views("abCDefgHIJK", start=c(8, 3), end=c(14, 4))
gaps(v2)

## Coercion:
as(v12, "XStringSet")  # same as 'as(v12, "DNAStringSet")'
as(v12, "RNAStringSet")
```

---

## XStringViews-constructors

*Basic functions for creating or modifying XStringViews objects*

---

### Description

A set of basic functions for creating or modifying XStringViews objects.

### Usage

```
XStringViews(x, subjectClass, collapse="")
```

## Arguments

| | |
|---|---|
| `x` | An XString object or a character vector for `XStringViews`. |
| `subjectClass` | The class to be given to the subject of the XStringViews object created and returned by the function. Must be the name of one of the direct XString subclasses i.e. `"BString"`, `"DNAString"`, `"RNAString"` or `"AAString"`. |
| `collapse` | An optional character string to be inserted between the views of the XStringViews object created and returned by the function. |

## Details

The `XStringViews` constructor will try to create an XStringViews object from the value passed to its `x` argument. If `x` itself is an XStringViews object, the returned object is obtained by coercing its subject to the class specified by `subjectClass`. If `x` is an XString object, the returned object is made of a single view that starts at the first letter and ends at the last letter of `x` (in addition `x` itself is coerced to the class specified by `subjectClass` when specified). If `x` is a character vector, the returned object has one view per character string in `x` (and its subject is an instance of the class specified by `subjectClass`).

## Value

An XStringViews object `y`. `length(y)` (the number of views in `y`) is `1` when `x` is an XString object and `length(x)` otherwise.

## See Also

XStringViews-class, XString-class

## Examples

```
v12 <- Views(DNAString("TAATAATG"), start=-2:9, end=0:11)
XStringViews(v12, subjectClass="RNAString")
XStringViews(AAString("MARKSLEMSIR*"))
XStringViews("abcdefghij", subjectClass="BString")
```

---

`align-utils` *Utility functions related to sequence alignment*

---

## Description

A variety of different functions used to deal with sequence alignments.

## Usage

```
  nedit(x) # also nmatch and nmismatch

  mismatchTable(x, shiftLeft=0L, shiftRight=0L, ...)
  mismatchSummary(x, ...)
  ## S4 method for signature 'AlignedXStringSet0':
coverage(x, shift=0L, width=NULL, weight=1L)
  ## S4 method for signature 'PairwiseAlignedFixedSubject':
coverage(x, shift=0L, width=NULL, weight=1L)
```

```
  compareStrings(pattern, subject)

  ## S4 method for signature 'PairwiseAlignedFixedSubject':
consensusMatrix(x,
                  as.prob=FALSE, freq=FALSE, shift=0L, width=NULL,
                  baseOnly=FALSE, gapCode="-", endgapCode="-")
```

## Arguments

| | |
|---|---|
| `x` | A `character` vector or matrix, `XStringSet`, `XStringViews`, `PairwiseAlignedXStrin` or `list` of FASTA records containing the equal-length strings. |
| `shiftLeft, shiftRight` | Non-positive and non-negative integers respectively that specify how many preceding and succeeding characters to and from the mismatch position to include in the mismatch substrings. |
| `...` | Further arguments to be passed to or from other methods. |
| `shift, width` | See `?coverage`. |
| `weight` | An integer vector specifying how much each element in `x` counts. |
| `pattern, subject` | The strings to compare. Can be of type `character`, `XString`, `XStringSet`, `AlignedXStringSet`, or, in the case of `pattern`, `PairwiseAlignedXStringSet`. If `pattern` is a `PairwiseAlignedXStringSet` object, then `subject` must be missing. |
| `as.prob` | If `TRUE` then probabilities are reported, otherwise counts (the default). |
| `freq` | This argument is deprecated. Please use the `as.prob` argument instead. |
| `baseOnly` | `TRUE` or `FALSE`. If `TRUE`, the returned vector only contains frequencies for the letters in the "base" alphabet i.e. "A", "C", "G", "T" if `x` is a "DNA input", and "A", "C", "G", "U" if `x` is "RNA input". When `x` is a BString object (or an XStringViews object with a BString subject, or a BStringSet object), then the `baseOnly` argument is ignored. |
| `gapCode, endgapCode` | The codes in the appropriate `alphabet` to use for the internal and end gaps. |

## Details

`mismatchTable`: a data.frame containing the positions and substrings of the mismatches for the `AlignedXStringSet` or `PairwiseAlignedXStringSet` object.

`mismatchSummary`: a list of data.frame objects containing counts and frequencies of the mismatches for the `AlignedXStringSet` or `PairwiseAlignedFixedSubject` object.

`compareStrings` combines two equal-length strings that are assumed to be aligned into a single character string containing that replaces mismatches with `"?"`, insertions with `"+"`, and deletions with `"-"`.

## See Also

`pairwiseAlignment`, `consensusMatrix`, XString-class, XStringSet-class, XStringViews-class, AlignedXStringSet-class, PairwiseAlignedXStringSet-class, match-utils

## Examples

```
## Compare two globally aligned strings
string1 <- "ACTTCACCAGCTCCCTGGCGGTAAGTTGATC---AAAGG---AAACGCAAAGTTTTCAAG"
string2 <- "GTTTCACTACTTCCTTTCGGGTAAGTAAATATATAAATATATAAAAATATAATTTTCATC"
compareStrings(string1, string2)

## Create a consensus matrix
nw1 <-
  pairwiseAlignment(AAStringSet(c("HLDNLKGTF", "HVDDMPNAL")), AAString("SMDDTEKMSMKL"),
    substitutionMatrix = "BLOSUM50", gapOpening = -3, gapExtension = -1)
consensusMatrix(nw1)

## Examine the consensus between the bacteriophage phi X174 genomes
data(phiX174Phage)
phageConsmat <- consensusMatrix(phiX174Phage, baseOnly = TRUE)
phageDiffs <- which(apply(phageConsmat, 2, max) < length(phiX174Phage))
phageDiffs
phageConsmat[,phageDiffs]
```

---

`basecontent` *Obtain the ATCG content of a gene*

---

## Description

WARNING: Both `basecontent` and `countbases` have been deprecated in favor of `alphabetFrequency`.

These functions accept a character vector representing the nucleotide sequences and compute the frequencies of each base (A, C, G, T).

## Usage

```
basecontent(seq)
countbases(seq, dna = TRUE)
```

## Arguments

| | |
|---|---|
| `seq` | Character vector. |
| `dna` | Logical value indicating whether the sequence is DNA (`TRUE`) or RNA (`FALSE`) |

## Details

The base frequencies are calculated separately for each element of `x`. The elements of `x` can be in upper case, lower case or mixed.

## Value

A matrix with 4 columns and `length(x)` rows. The columns are named `A`, `C`, `T`, `G`, and the values in each column are the counts of the corresponding bases in the elements of `x`. When `dna=FALSE`, the `T` column is replaced with a `U` column.

## Author(s)

R. Gentleman, W. Huber, S. Falcon

## See Also

`alphabetFrequency`, `reverseComplement`

## Examples

```
v<-c("AAACT", "GGGTT", "ggAtT")

## Do not use these functions anymore:
if (interactive()) {
  basecontent(v)
  countbases(v)
}

## But use more efficient alphabetFrequency() instead:
v <- DNAStringSet(v)
alphabetFrequency(v, baseOnly=TRUE)

## Comparing efficiencies:
if (interactive()) {
  library(hgu95av2probe)
  system.time(y1 <- countbases(hgu95av2probe$sequence))
  x <- DNAStringSet(hgu95av2probe)
  system.time(y2 <- alphabetFrequency(x, baseOnly=TRUE))
}
```

---

`chartr` *Translating letters of a sequence*

---

## Description

Translate letters of a sequence.

## Usage

```
  ## S4 method for signature 'ANY,ANY,XString':
chartr(old, new, x)
```

## Arguments

| | |
|---|---|
| `old` | A character string specifying the characters to be translated. |
| `new` | A character string specifying the translations. |
| `x` | The sequence or set of sequences to translate. If `x` is an XString, XStringSet, XStringViews or MaskedXString object, then the appropriate `chartr` method is called, otherwise the standard `chartr` R function is called. |

## Details

See `?chartr` for the details.

Note that, unlike the standard `chartr` R function, the methods for XString, XStringSet, XStringViews and MaskedXString objects do NOT support character ranges in the specifications.

## Value

An object of the same class and length as the original object.

## See Also

`chartr`, `replaceLetterAt`, XString-class, XStringSet-class, XStringViews-class, MaskedXString-class, `alphabetFrequency`, `matchPattern`, `reverseComplement`

## Examples

```
x <- BString("MiXeD cAsE 123")
chartr("iXs", "why", x)

## ---------------------------------------------------------------------
## TRANSFORMING DNA WITH BISULFITE (AND SEARCHING IT...)
## ---------------------------------------------------------------------

library(BSgenome.Celegans.UCSC.ce2)
chrII <- Celegans[["chrII"]]
alphabetFrequency(chrII)
pattern <- DNAString("TGGGTGTATTTA")

## Transforming and searching the + strand
plus_strand <- chartr("C", "T", chrII)
alphabetFrequency(plus_strand)
matchPattern(pattern, plus_strand)
matchPattern(pattern, chrII)

## Transforming and searching the - strand
minus_strand <- chartr("G", "A", chrII)
alphabetFrequency(minus_strand)
matchPattern(reverseComplement(pattern), minus_strand)
matchPattern(reverseComplement(pattern), chrII)
```

---

## `complementSeq` *Complementary sequence.*

---

## Description

WARNING: `complementSeq` has been deprecated in favor of `complement`.

Function to obtain the complementary sequence.

## Usage

```
complementSeq(seq, start=1, stop=0)
```

## Arguments

| | |
|---|---|
| `seq` | Character vector consisting of the letters A, C, G and T. |
| `start` | Numeric scalar: the sequence position at which to start complementing. If 1, start from the beginning. |
| `stop` | Numeric scalar: the sequence position at which to stop complementing. If 0, go until the end. |

## Details

The complemented sequence for each element of the input is computed and returned. The complement is given by the mapping: A -> T, C -> G, G -> C, T -> A.

An important special case is `start=13`, `stop=13`: If `seq` is a vector of 25mer sequences on an Affymetrix GeneChip, `complementSeq(seq, start=13, stop=13)` calculates the so-called *mismatch* sequences.

The function deals only with sequences that represent DNA. These can consist only of the letters `A`, `C`, `T` or `G`. Upper, lower or mixed case is allowed and honored.

## Value

A character vector of the same length as `seq` is returned. Each component represents the transformed sequence for the input value.

## Author(s)

R. Gentleman, W. Huber

## See Also

`alphabetFrequency`, `reverseComplement`

## Examples

```
## ----------------------------------------------------------------------
## EXAMPLE 1
## ----------------------------------------------------------------------
seq <- c("AAACT", "GGGTT")

## Don't do this anymore (deprecated):
if (interactive()) {
  complementSeq(seq)  # inefficient on large vectors
}
## But do this instead:
complement(DNAStringSet(seq))  # more efficient

## ----------------------------------------------------------------------
## EXAMPLE 2
## ----------------------------------------------------------------------
seq <- c("CGACTGAGACCAAGACCTACAACAG", "CCCGCATCATCTTTCCTGTGCTCTT")

## Don't do this anymore (deprecated):
if (interactive()) {
  complementSeq(seq, start=13, stop=13)
}
## But do this instead:
pm2mm <- function(probes)
{
    probes <- DNAStringSet(probes)
    subseq(probes, start=13, end=13) <- complement(subseq(probes, start=13, end=13))
    probes
}
pm2mm(seq)

## ----------------------------------------------------------------------
## SPEED OF complementSeq() VS complement()
## ------------------------------------------------------------------------
if (interactive()) {
  library(hgu95av2probe)
  system.time(y1 <- complementSeq(hgu95av2probe$sequence))
  probes <- DNAStringSet(hgu95av2probe)
  system.time(y2 <- complement(probes))
}
```

---

`dinucleotideFrequencyTest`
*Pearson's chi-squared Test and G-tests for String Position Dependence*

---

## Description

Performs Person's chi-squared test, G-test, or William's corrected G-test to determine dependence between two nucleotide positions.

## Usage

```
dinucleotideFrequencyTest(x, i, j, test = c("chisq", "G", "adjG"),
                          simulate.p.value = FALSE, B = 2000)
```

## Arguments

| | |
|---|---|
| `x` | A DNAStringSet or RNAStringSet object. |
| `i, j` | Single integer values for positions to test for dependence. |
| `test` | One of `"chisq"` (Person's chi-squared test), `"G"` (G-test), or `"adjG"` (William's corrected G-test). See Details section. |
| `simulate.p.value` | a logical indicating whether to compute p-values by Monte Carlo simulation. |
| `B` | an integer specifying the number of replicates used in the Monte Carlo test. |

## Details

The null and alternative hypotheses for this function are:

**H0:** positions `i` and `j` are independent

**H1:** otherwise

Let O and E be the observed and expected probabilities for base pair combinations at positions `i` and `j` respectively. Then the test statistics are calculated as:

`test="chisq"`: stat = sum(abs(O - E)^2/E)

`test="G"`: stat = 2 * sum(O * log(O/E))

`test="adjG"`: stat = 2 * sum(O * log(O/E))/q, where q = 1 + ((df - 1)^2 - 1)/(6*length(x)*(df - 2))

Under the null hypothesis, these test statistics are approximately distributed chi-squared(df = ((distinct bases at i) - 1) * ((distinct bases at j) - 1)).

### Value

An htest object. See help(chisq.test) for more details.

### Author(s)

P. Aboyoun

### References

Ellrott, K., Yang, C., Sladek, F.M., Jiang, T. (2002) "Identifying transcription factor binding sites through Markov chain optimations", Bioinformatics, 18 (Suppl. 2), S100-S109.

Sokal, R.R., Rohlf, F.J. (2003) "Biometry: The Principle and Practice of Statistics in Biological Research", W.H. Freeman and Company, New York.

Tomovic, A., Oakeley, E. (2007) "Position dependencies in transcription factor binding sites", Bioinformatics, 23, 933-941.

Williams, D.A. (1976) "Improved Likelihood ratio tests for complete contingency tables", Biometrika, 63, 33-37.

### See Also

`nucleotideFrequencyAt`, XStringSet-class, `chisq.test`

### Examples

```
data(HNF4alpha)
dinucleotideFrequencyTest(HNF4alpha, 1, 2)
dinucleotideFrequencyTest(HNF4alpha, 1, 2, test = "G")
dinucleotideFrequencyTest(HNF4alpha, 1, 2, test = "adjG")
```

---

## `extractTranscripts` *Extract a set of transcripts*

---

### Description

`extractTranscripts` allows the user to extract a set of transcripts specified by the starts and ends of their exons as well as the strand from which the transcript is coming.

`transcriptWidths` only returns the lengths of the transcripts (called the "widths" in this context) specified by the starts and ends of their exons.

`transcriptLocs2refLocs` converts transcript-based locations into reference-based locations.

### Usage

```
extractTranscripts(x, exonStarts=list(), exonEnds=list(),
                   strand=character(0), reorder.exons.on.minus.strand=FALSE)

transcriptWidths(exonStarts=list(), exonEnds=list())

transcriptLocs2refLocs(tlocs, exonStarts=list(), exonEnds=list(),
                       strand=character(0),
                       reorder.exons.on.minus.strand=FALSE)
```

## Arguments

| | |
|---|---|
| `x` | A DNAString or MaskedDNAString object. |
| `exonStarts, exonEnds` | The starts and ends of the exons, respectively.<br>Each argument can be a list of integer vectors, an IntegerList object, or a character vector where each element is a comma-separated list of integers. In addition, the lists represented by `exonStarts` and `exonEnds` must have the same shape i.e. have the same lengths and have elements of the same lengths. The length of `exonStarts` and `exonEnds` is the number of transcripts. |
| `strand` | A character vector of the same length as `exonStarts` and `exonEnds` specifying the strand (`"+"` or `"-"`) from which the transcript is coming. |
| `reorder.exons.on.minus.strand` | `TRUE` or `FALSE`. Should the order of exons for transcripts coming from the minus strand be reversed? |
| `tlocs` | A list of integer vectors of the same length as `exonStarts` and `exonEnds`. Each element in `tlocs` must contain transcript-based locations. |

## Details

`extractTranscripts` allows the user to extract a set of transcripts specified by the starts and ends of their exons as well as the strand from which the transcript is coming. See `extractTranscriptsFromGenom` in the GenomicFeatures package for extracting transcripts from a genome.

## Value

A DNAStringSet object for `extractTranscripts`.

An integer vector for `transcriptWidths`.

A list of integer vectors of the same shape as `tlocs` for `transcriptLocs2refLocs`.

## See Also

`extractTranscriptsFromGenome`, `reverseComplement`, DNAString-class, DNAStringSet-class

## Examples

```
## ---------------------------------------------------------------------
## A. EXTRACTING WORM TRANSCRIPTS ZC101.3 AND F37B1.1
## ---------------------------------------------------------------------

## Transcript ZC101.3 (is on + strand):
##   Exons starts/ends relative to transcript:
rstarts1 <- c(1, 488, 654, 996, 1365, 1712, 2163, 2453)
rends1 <- c(137, 578, 889, 1277, 1662, 1870, 2410, 2561)
##   Exons starts/ends relative to chromosome:
starts1 <- 14678410 + rstarts1
ends1 <- 14678410 + rends1

## Transcript F37B1.1 (is on - strand):
##   Exons starts/ends relative to transcript:
rstarts2 <- c(1, 325)
rends2 <- c(139, 815)
```

```
##   Exons starts/ends relative to chromosome:
starts2 <- 13611188 - rends2
ends2 <- 13611188 - rstarts2

exon_starts <- list(as.integer(starts1), as.integer(starts2))
exon_ends <- list(as.integer(ends1), as.integer(ends2))

library(BSgenome.Celegans.UCSC.ce2)
## Both transcripts are on chrII:
chrII <- Celegans$chrII
transcripts <- extractTranscripts(chrII,
                  exonStarts=exon_starts,
                  exonEnds=exon_ends,
                  strand=c("+","-"))

## Same as 'width(transcripts)':
transcriptWidths(exonStarts=exon_starts, exonEnds=exon_ends)

transcriptLocs2refLocs(list(c(1:6, 135:140, 1555:1560), c(1:6, 137:142, 625:630)),
                  exonStarts=exon_starts,
                  exonEnds=exon_ends,
                  strand=c("+","-"))

## A sanity check:
ref_locs <- transcriptLocs2refLocs(list(1:1560, 1:630),
                  exonStarts=exon_starts,
                  exonEnds=exon_ends,
                  strand=c("+","-"))
stopifnot(chrII[ref_locs[[1]]] == transcripts[[1]])
stopifnot(complement(chrII)[ref_locs[[2]]] == transcripts[[2]])
```

## findPalindromes *Searching a sequence for palindromes or complemented palindromes*

### Description

The `findPalindromes` and `findComplementedPalindromes` functions can be used to find palindromic or complemented palindromic regions in a sequence.

`palindromeArmLength`, `palindromeLeftArm`, `palindromeRightArm`, `complementedPalindromeAr`
`complementedPalindromeLeftArm` and `complementedPalindromeRightArm` are utility functions for operating on palindromic or complemented palindromic sequences.

### Usage

```
findPalindromes(subject, min.armlength=4, max.looplength=1, min.looplength=0,
palindromeArmLength(x, max.mismatch=0, ...)
palindromeLeftArm(x, max.mismatch=0, ...)
palindromeRightArm(x, max.mismatch=0, ...)

findComplementedPalindromes(subject, min.armlength=4, max.looplength=1, min.lo
complementedPalindromeArmLength(x, max.mismatch=0, ...)
complementedPalindromeLeftArm(x, max.mismatch=0, ...)
complementedPalindromeRightArm(x, max.mismatch=0, ...)
```

## Arguments

| | |
|---|---|
| `subject` | An XString object containing the subject string, or an XStringViews object. |
| `min.armlength` | An integer giving the minimum length of the arms of the palindromes (or complemented palindromes) to search for. |
| `max.looplength` | An integer giving the maximum length of "the loop" (i.e the sequence separating the 2 arms) of the palindromes (or complemented palindromes) to search for. Note that by default (`max.looplength=1`), `findPalindromes` will search for strict palindromes (or complemented palindromes) only. |
| `min.looplength` | An integer giving the minimum length of "the loop" of the palindromes (or complemented palindromes) to search for. |
| `max.mismatch` | The maximum number of mismatching letters allowed between the 2 arms of the palindromes (or complemented palindromes) to search for. |
| `x` | An XString object containing a 2-arm palindrome or complemented palindrome, or an XStringViews object containing a set of 2-arm palindromes or complemented palindromes. |
| `...` | Additional arguments to be passed to or from methods. |

## Details

The `findPalindromes` function finds palindromic substrings in a subject string. The palindromes that can be searched for are either strict palindromes or 2-arm palindromes (the former being a particular case of the latter) i.e. palindromes where the 2 arms are separated by an arbitrary sequence called "the loop".

Use the `findComplementedPalindromes` function to find complemented palindromic substrings in a DNAString subject (in a complemented palindrome the 2 arms are reverse-complementary sequences).

## Value

`findPalindromes` and `findComplementedPalindromes` return an XStringViews object containing all palindromes (or complemented palindromes) found in `subject` (one view per palindromic substring found).

`palindromeArmLength` and `complementedPalindromeArmLength` return the arm length (integer) of the 2-arm palindrome (or complemented palindrome) `x`. It will raise an error if `x` has no arms. Note that any sequence could be considered a 2-arm palindrome if we were OK with arms of length 0 but we are not: `x` must have arms of length greater or equal to 1 in order to be considered a 2-arm palindrome. The same apply to 2-arm complemented palindromes. When applied to an XStringViews object `x`, `palindromeArmLength` and `complementedPalindromeArmLength` behave in a vectorized fashion by returning an integer vector of the same length as `x`.

`palindromeLeftArm` and `complementedPalindromeLeftArm` return an object of the same class as the original object `x` and containing the left arm of `x`.

`palindromeRightArm` does the same as `palindromeLeftArm` but on the right arm of `x`.

Like `palindromeArmLength`, both `palindromeLeftArm` and `palindromeRightArm` will raise an error if `x` has no arms. Also, when applied to an XStringViews object `x`, both behave in a vectorized fashion by returning an XStringViews object of the same length as `x`.

## Author(s)

H. Pages

## See Also

maskMotif, matchPattern, matchLRPatterns, matchProbePair, XStringViews-class, DNAString-class

## Examples

```
## Note that complemented palindromes (like palindromes) can be nested
findComplementedPalindromes(DNAString("ACGTTNAACGT-ACGTTNAACGT"))

## A real use case
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- Dmelanogaster$chrX
chrX_pals <- findComplementedPalindromes(chrX, min.armlength=50, max.looplength=20)
complementedPalindromeArmLength(chrX_pals)  # 251

## Of course, whitespaces matter
palindromeArmLength(BString("was it a car or a cat I saw"))

## Note that the 2 arms of a strict palindrome (or strict complemented
## palindrome) are equal to the full sequence.
palindromeLeftArm(BString("Delia saw I was aileD"))
complementedPalindromeLeftArm(DNAString("N-ACGTT-AACGT-N"))
palindromeLeftArm(DNAString("N-AAA-N-N-TTT-N"))
```

---

# gregexpr2 — *A replacement for R standard gregexpr function*

---

## Description

This is a replacement for the standard gregexpr function that does exact matching only. Standard gregexpr() misses matches when they are overlapping. The gregexpr2 function finds all matches but it only works in "fixed" mode i.e. for exact matching (regular expressions are not supported).

## Usage

```
gregexpr2(pattern, text)
```

## Arguments

| | |
|---|---|
| `pattern` | character string to be matched in the given character vector |
| `text` | a character vector where matches are sought |

## Value

A list of the same length as `text` each element of which is an integer vector as in `gregexpr`, except that the starting positions of all (even overlapping) matches are given. Note that, unlike `gregexpr`, `gregexpr2` doesn't attach a "match.length" attribute to each element of the returned list because, since it only works in "fixed" mode, then all the matches have the length of the pattern. Another difference with `gregexpr` is that with `gregexpr2`, the `pattern` argument must be a single (non-NA, non-empty) string.

### Author(s)

H. Pages

### See Also

`gregexpr`, `matchPattern`

### Examples

```
gregexpr("aa", c("XaaaYaa", "a"), fixed=TRUE)
gregexpr2("aa", c("XaaaYaa", "a"))
```

---

## `injectHardMask` *Injecting a hard mask in a sequence*

---

### Description

`injectHardMask` allows the user to "fill" the masked regions of a sequence with an arbitrary letter (typically the `"+"` letter).

### Usage

```
injectHardMask(x, letter="+")
```

### Arguments

| | |
|---|---|
| `x` | A MaskedXString or XStringViews object. |
| `letter` | A single letter. |

### Details

The name of the `injectHardMask` function was chosen because of the primary use that it is intended for: converting a pile of active "soft masks" into a "hard mask". Here the pile of active "soft masks" refers to the active masks that have been put on top of a sequence. In Biostrings, the original sequence and the masks defined on top of it are bundled together in one of the dedicated containers for this: the MaskedBString, MaskedDNAString, MaskedRNAString and MaskedAAString containers (this is the MaskedXString family of containers). The original sequence is always stored unmodified in a MaskedXString object so no information is lost. This allows the user to activate/deactivate masks without having to worry about losing the letters that are in the regions that are masked/unmasked. Also this allows better memory management since the original sequence never needs to be copied, even when the set of active/inactive masks changes.

However, there are situations where the user might want to *really* get rid of the letters that are in some particular regions by replacing them with a junk letter (e.g. `"+"`) that is guaranteed to not interfer with the analysis that s/he is currently doing. For example, it's very likely that a set of motifs or short reads will not contain the `"+"` letter (this could easily be checked) so they will never hit the regions filled with `"+"`. In a way, it's like the regions filled with `"+"` were masked but we call this kind of masking "hard masking".

Some important differences between "soft" and "hard" masking:

`injectHardMask` creates a (modified) copy of the original sequence. Using "soft masking" does not.

A function that is "mask aware" like `alphabetFrequency` or `matchPattern` will really skip the masked regions when "soft masking" is used i.e. they will not walk thru the regions that are under active masks. This might lead to some speed improvements when a high percentage of the original sequence is masked. With "hard masking", the entire sequence is walked thru.

Matches cannot span over masked regions with "soft masking". With "hard masking" they can.

### Value

An XString object of the same length as the orignal object `x` if `x` is a MaskedXString object, or of the same length as `subject(x)` if it's an XStringViews object.

### Author(s)

H. Pages

### See Also

`maskMotif`, MaskedXString-class, `replaceLetterAt`, `chartr`, XString, XStringViews-class

### Examples

```
## ---------------------------------------------------------------------
## A. WITH AN XStringViews OBJECT
## ---------------------------------------------------------------------
v2 <- Views("abCDefgHIJK", start=c(8, 3), end=c(14, 4))
injectHardMask(v2)
injectHardMask(v2, letter="=")

## ---------------------------------------------------------------------
## B. WITH A MaskedXString OBJECT
## ---------------------------------------------------------------------
mask0 <- Mask(mask.width=29, start=c(3, 10, 25), width=c(6, 8, 5))
x <- DNAString("ACACAACTAGATAGNACTNNGAGAGACGC")
masks(x) <- mask0
x
subject <- injectHardMask(x)

## Matches can span over masked regions with "hard masking":
matchPattern("ACggggggA", subject, max.mismatch=6)
## but not with "soft masking":
matchPattern("ACggggggA", x, max.mismatch=6)
```

---

`letter` *Subsetting a string*

---

### Description

Extract a substring from a string by picking up individual letters by their position.

### Usage

```
letter(x, i)
```

## Arguments

| | |
|---|---|
| x | A character vector, or an XString, XStringViews or MaskedXString object. |
| i | An integer vector with no NAs. |

## Details

Unlike with the `substr` or `substring` functions, `i` must contain valid positions.

## Value

A character vector of length 1 when `x` is an XString or MaskedXString object (the masks are ignored for the latter).

A character vector of the same length as `x` when `x` is a character vector or an XStringViews object.

Note that, because `i` must contain valid positions, all non-NA elements in the result are guaranteed to have exactly `length(i)` characters.

## See Also

`subseq`, XString-class, XStringViews-class, MaskedXString-class

## Examples

```
x <- c("abcd", "ABC")
i <- c(3, 1, 1, 2, 1)

## With a character vector:
letter(x[1], 3:1)
letter(x, 3)
letter(x, i)
#letter(x, 4)             # Error!

## With a BString object:
letter(BString(x[1]), i)  # returns a character vector
BString(x[1])[i]          # returns a BString object

## With an XStringViews object:
x2 <- XStringViews(x, "BString")
letter(x2, i)
```

---

`letterFrequency` *Calculate the frequency of letters in a biological sequence, or the consensus matrix of a set of sequences*

---

## Description

Given a biological sequence (or a set of biological sequences), the `alphabetFrequency` function computes the frequency of each letter in the (base) alphabet.

The `letterFrequencyInSlidingView` function is a more specialized version of `alphabetFrequency` that computes the frequencies of a set of letters in a view (or window) that is conceptually sliding along the input sequence.

The `consensusMatrix` function computes the consensus matrix of a set of sequences, and the `consensusString` function creates the consensus sequence from the consensus matrix based upon specified criteria.

In this man page we call "DNA input" (or "RNA input") an XString, XStringSet, XStringViews or MaskedXString object of base type DNA (or RNA).

## Usage

```
  alphabetFrequency(x, as.prob=FALSE, freq=FALSE, ...)
  hasOnlyBaseLetters(x)
  uniqueLetters(x)

  letterFrequencyInSlidingView(x, view.width, letters, OR="|")

  consensusMatrix(x, as.prob=FALSE, freq=FALSE, shift=0L, width=NULL, ...)

  ## S4 method for signature 'matrix':
consensusString(x, ambiguityMap="?", threshold=0.5)
  ## S4 method for signature 'DNAStringSet':
consensusString(x, ambiguityMap=IUPAC_CODE_MAP,
               threshold=0.25, shift=0L, width=NULL)
  ## S4 method for signature 'RNAStringSet':
consensusString(x,
               ambiguityMap=
               structure(as.character(RNAStringSet(DNAStringSet(IUPAC_CODE_MAP))
                         names=
                         as.character(RNAStringSet(DNAStringSet(names(IUPAC_CODE
               threshold=0.25, shift=0L, width=NULL)
```

## Arguments

| | |
|---|---|
| `x` | An XString, XStringSet, XStringViews or MaskedXString object for `alphabetFrequency` or `uniqueLetters`. DNA or RNA input for `hasOnlyBaseLetters`. An XString object for `letterFrequencyInSlidingView`. A character vector, or an XStringSet or XStringViews object for `consensusMatrix`. A consensus matrix (as returned by `consensusMatrix`), or an XStringSet or XStringViews object for `consensusString`. |
| `as.prob` | If `TRUE` then probabilities are reported, otherwise counts (the default). |
| `freq` | This argument is deprecated. Please use the `as.prob` argument instead. |
| `view.width` | For letterFrequencyInSlidingView, the constant (e.g. 35, 48, 1000) size of the "window" to slide along `x`. The specified `letters` are tabulated in each window of length `view.width`. The rows of the result (see value) correspond to the various windows. |
| `letters` | For letterFrequencyInSlidingView, a character vector (e.g. "C", "CG", c("C", "G")) giving the letters to tabulate. Except with `OR=0`, multi-character elements of letters ('nchar' > 1) are taken as groupings of letters into subsets, to be tabulated in common ("or"'d), as if their alphabetFrequency's were added (Arithmetic). The columns of the result (see value) correspond to the individual and sets of letters which are counted separately. Unrelated (and, with some post-processing, related) counts may of course be obtained in separate calls. |

| | |
|---|---|
| `OR` | For `letterFrequencyInSlidingView`, the string (default `\|`) to use as a separator in forming names for the "grouped" columns, e.g. "C\|G". The otherwise exceptional value `0` (zero) disables or'ing and is provided for convenience, allowing a single multi-character string (or several strings) of letters that should be counted separately. If some but not all letters are to be counted separately, they must reside in separate elements of letters (with 'nchar' 1 unless they are to be grouped with other letters), and `OR` cannot be 0. |
| `ambiguityMap` | Either a single character to use when agreement is not reached or a named character vector where the names are the ambiguity characters and the values are the combinations of letters that comprise the ambiguity (e.g. `link{IUPAC_CODE_MAP}`). When `ambiguityMap` is a named character vector, occurrences of ambiguous letters in `x` are replaced with their base alphabet letters that have been equally weighted to sum to 1. (See Details for some examples.) |
| `threshold` | The minimum probability threshold for an agreement to be declared. When `ambiguityMap` is a single character, `threshold` is a single number in (0, 1]. When `ambiguityMap` is a named character vector (e.g. `link{IUPAC_CODE_MAP}`), `threshold` is a single number in (0, 1/sum(nchar(ambiguityMap) == 1)]. |
| `...` | Further arguments to be passed to or from other methods. For the XStringViews and XStringSet methods, the `collapse` argument is accepted. For DNA or RNA input, the `baseOnly` argument is accepted. If `baseOnly` is `TRUE`, the returned vector (or matrix) only contains the frequencies of the letters that belong to the "base" alphabet of `x` i.e. to the alphabet returned by `alphabet(x, baseOnly=TRUE)`. |
| `shift` | An integer vector (recycled to the length of `x`) specifying how each sequence in `x` should be (horizontally) shifted with respect to the first column of the consensus matrix to be returned. By default (`shift=0`), each sequence in `x` has its first letter aligned with the first column of the matrix. A positive `shift` value means that the corresponding sequence must be shifted to the right, and a negative `shift` value that it must be shifted to the left. For example, a shift of 5 means that it must be shifted 5 positions to the right (i.e. the first letter in the sequence must be aligned with the 6th column of the matrix), and a shift of -3 means that it must be shifted 3 positions to the left (i.e. the 4th letter in the sequence must be aligned with the first column of the matrix). |
| `width` | The number of columns of the returned matrix for the `consensusMatrix` method for XStringSet objects. When `width=NULL` (the default), then this method returns a matrix that has just enough columns to have its last column aligned with the rightmost letter of all the sequences in `x` after those sequences have been shifted (see the `shift` argument above). This ensures that any wider consensus matrix would be a "padded with zeros" version of the matrix returned when `width=NULL`. The length of the returned sequence for the `consensusString` method for XStringSet objects. |

## Details

`alphabetFrequency` and `letterFrequencyInSlidingView` are generic functions defined in the Biostrings package.

`letterFrequencyInSlidingView` is a much lighter alternative to `alphabetFrequency`, without `collapse`, of the hypothetical XStringViews object consisting of every interval of length view.width on x. If x is masked (MaskedXString), it is treated as the XStringSet of its visible segments. To include the masked regions (as well as the intervals of length view.width-1 which immediately precede them), use unmasked(x) or DNAString(x) as the subject.

When consensusString is executed with a named character ambiguityMap argument, it weights each input string equally and assigns an equal probability to each of the base letters represented by an ambiguity letter. So for DNA and a threshold of 0.25, a "G" and an "R" would result in an "R" since 1/2 "G" + 1/2 "R" = 3/4 "G" + 1/4 "A" => "R"; two "G"'s and one "R" would result in a "G" since 2/3 "G" + 1/3 "R" = 5/6 "G" + 1/6 "A" => "G"; and one "A" and one "N" would result in an "N" since 1/2 "A" + 1/2 "N" = 5/8 "A" + 1/8 "C" + 1/8 "G" + 1/8 "T" => "N".

## Value

alphabetFrequency returns an integer vector when x is an XString or MaskedXString object. When x is an XStringSet or XStringViews object, then it returns an integer matrix with length(x) rows where the i-th row contains the frequencies for x[[i]]. If x is a DNA or RNA input, then the returned vector is named with the letters in the alphabet. If the baseOnly argument is TRUE, then the returned vector has only 5 elements: 4 elements corresponding to the 4 nucleotides + the 'other' element.

letterFrequencyInSlidingView returns for each XString element of x, say s of length L, an integer matrix with L-view.width+1 rows, the i-th of which holds the various letter frequencies in the i-th "window along s", i.e. substring(s, i, i+view.width-1).

hasOnlyBaseLetters returns TRUE or FALSE indicating whether or not x contains only base letters (i.e. As, Cs, Gs and Ts for DNA input and As, Cs, Gs and Us for RNA input).

uniqueLetters returns a vector of 1-letter or empty strings. The empty string is used to represent the nul character if x happens to contain any. Note that this can only happen if the base class of x is BString.

An integer matrix with letters as row names for consensusMatrix.

A standard character string for consensusString.

## Author(s)

H. Pages and P. Aboyoun; H. Jaffee for letterFrequencyInSlidingView

## See Also

alphabet, coverage, oligonucleotideFrequency, countPDict, XString-class, XStringSet-class, XStringViews-class, MaskedXString-class, strsplit

## Examples

```
## ---------------------------------------------------------------------
## alphabetFrequency()
## ---------------------------------------------------------------------
data(yeastSEQCHR1)
yeast1 <- DNAString(yeastSEQCHR1)

alphabetFrequency(yeast1)
alphabetFrequency(yeast1, baseOnly=TRUE)

hasOnlyBaseLetters(yeast1)
uniqueLetters(yeast1)

## With input made of multiple sequences:
```

```
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
alphabetFrequency(probes[1:50], baseOnly=TRUE)
alphabetFrequency(probes, baseOnly=TRUE, collapse=TRUE)

## ---------------------------------------------------------------------
## letterFrequencyInSlidingView()
## ---------------------------------------------------------------------
data(yeastSEQCHR1)
x <- DNAString(yeastSEQCHR1)
view.width <- 48
letters <- c("A", "CG")
two_columns <- letterFrequencyInSlidingView(x, view.width, letters)
head(two_columns)
tail(two_columns)
three_columns <- letterFrequencyInSlidingView(x, view.width, letters, OR=0)
head(three_columns)
tail(three_columns)
stopifnot(identical(two_columns[ , "C|G"],
                    three_columns[ , "C"] + three_columns[ , "G"]))

## Note that, alternatively, 'three_columns' can also be obtained by
## creating the views on 'x' (as a Views object) and by calling
## alphabetFrequency() on it. But, of course, that is be *much* less
## efficient (both, in terms of memory and speed) than using
## letterFrequencyInSlidingView():
v <- Views(x, start=seq_len(length(x) - view.width + 1), width=view.width)
v
three_columns2 <- alphabetFrequency(v, baseOnly=TRUE)[ , c("A", "C", "G")]
stopifnot(identical(three_columns2, three_columns))

## Set the width of the view to length(x) to get the global frequencies:
letterFrequencyInSlidingView(x, letters="ACGTN", view.width=length(x), OR=0)

## ---------------------------------------------------------------------
## consensus*()
## ---------------------------------------------------------------------
## Read in ORF data:
file <- system.file("extdata", "someORF.fa", package="Biostrings")
orf <- read.DNAStringSet(file)

## To illustrate, the following example assumes the ORF data
## to be aligned for the first 10 positions (patently false):
orf10 <- DNAStringSet(orf, end=10)
consensusMatrix(orf10, baseOnly=TRUE)

## The following example assumes the first 10 positions to be aligned
## after some incremental shifting to the right (patently false):
consensusMatrix(orf10, baseOnly=TRUE, shift=0:6)
consensusMatrix(orf10, baseOnly=TRUE, shift=0:6, width=10)

## For the character matrix containing the "exploded" representation
## of the strings, do:
as.matrix(orf10, use.names=FALSE)

## consensusMatrix() can be used to just compute the alphabet frequency
## for each position in the input sequences:
```

```
consensusMatrix(probes, baseOnly=TRUE)

## After sorting, the first 5 probes might look similar (at least on
## their first bases):
consensusString(sort(probes)[1:5])
consensusString(sort(probes)[1:5], ambiguityMap = "N", threshold = 0.5)

## Consensus involving ambiguity letters in the input strings
consensusString(DNAStringSet(c("NNNN","ACTG")))
consensusString(DNAStringSet(c("AANN","ACTG")))
consensusString(DNAStringSet(c("ACAG","ACAR")))
consensusString(DNAStringSet(c("ACAG","ACAR", "ACAG")))

## ---------------------------------------------------------------------
## C. RELATIONSHIP BETWEEN consensusMatrix() AND coverage()
## ---------------------------------------------------------------------
## Applying colSums() on a consensus matrix gives the coverage that
## would be obtained by piling up (after shifting) the input sequences
## on top of an (imaginary) reference sequence:
cm <- consensusMatrix(orf10, shift=0:6, width=10)
colSums(cm)

## Note that this coverage can also be obtained with:
as.integer(coverage(IRanges(rep(1, length(orf)), width(orf)), shift=0:6, width=10))
```

## `longestConsecutive` *Obtain the length of the longest substring containing only 'letter'*

### Description

This function accepts a character vector and computes the length of the longest substring containing only `letter` for each element of `x`.

### Usage

```
longestConsecutive(seq, letter)
```

### Arguments

| | |
|---|---|
| `seq` | Character vector. |
| `letter` | Character vector of length 1, containing one single character. |

### Details

The elements of `x` can be in upper case, lower case or mixed. NAs are handled.

### Value

An integer vector of the same length as `x`.

### Author(s)

W. Huber

## See Also

complementSeq,basecontent,reverseSeq

## Examples

```
v = c("AAACTGTGFG", "GGGAATT", "CCAAAAAAAAAATT")
longestConsecutive(v, "A")
```

---

lowlevel-matching *Low-level matching functions*

---

## Description

In this man page we define precisely and illustrate what a "match" of a pattern P in a subject S is in the context of the Biostrings package. This definition of a "match" is central to most pattern matching functions available in this package: unless specified otherwise, most of them will adhere to the definition provided here.

`hasLetterAt` checks whether a sequence or set of sequences has the specified letters at the specified positions.

`neditAt`, `isMatchingAt` and `which.isMatchingAt` are low-level matching functions that only look for matches at the specified positions in the subject.

## Usage

```
hasLetterAt(x, letter, at, fixed=TRUE)

## neditAt() and related utils:
neditAt(pattern, subject, at=1,
        with.indels=FALSE, fixed=TRUE)
neditStartingAt(pattern, subject, starting.at=1,
        with.indels=FALSE, fixed=TRUE)
neditEndingAt(pattern, subject, ending.at=1,
        with.indels=FALSE, fixed=TRUE)

## isMatchingAt() and related utils:
isMatchingAt(pattern, subject, at=1,
        max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE)
isMatchingStartingAt(pattern, subject, starting.at=1,
        max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE)
isMatchingEndingAt(pattern, subject, ending.at=1,
        max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE)

## which.isMatchingAt() and related utils:
which.isMatchingAt(pattern, subject, at=1,
        max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
        follow.index=FALSE, auto.reduce.pattern=FALSE)
which.isMatchingStartingAt(pattern, subject, starting.at=1,
        max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
        follow.index=FALSE, auto.reduce.pattern=FALSE)
which.isMatchingEndingAt(pattern, subject, ending.at=1,
        max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
        follow.index=FALSE, auto.reduce.pattern=FALSE)
```

## Arguments

| | |
|---|---|
| `x` | A character vector, or an XString or XStringSet object. |
| `letter` | A character string or an XString object containing the letters to check. |
| `at, starting.at, ending.at` | An integer vector specifying the starting (for `starting.at` and `at`) or ending (for `ending.at`) positions of the pattern relatively to the subject. With `auto.reduce.pattern` (below), either a single integer or a constant vector of length `nchar(pattern)` (below), to which the former is immediately converted.<br>For the `hasLetterAt` function, `letter` and `at` must have the same length. |
| `pattern` | The pattern string (but see `auto.reduce.pattern`, below). |
| `subject` | A character vector, or an XString or XStringSet object containing the subject sequence(s). |
| `max.mismatch, min.mismatch` | Integer vectors of length >= 1 recycled to the length of the `at` (or `starting.at`, or `ending.at`) argument. More details below. |
| `with.indels` | See details below. |
| `fixed` | Only with a DNAString or RNAString-based subject can a `fixed` value other than the default (`TRUE`) be used.<br>If `TRUE` (the default), an IUPAC ambiguity code in the pattern can only match the same code in the subject, and vice versa. If `FALSE`, an IUPAC ambiguity code in the pattern can match any letter in the subject that is associated with the code, and vice versa. See `IUPAC_CODE_MAP` for more information about the IUPAC Extended Genetic Alphabet.<br>`fixed` can also be a character vector, a subset of `c("pattern", "subject")`. `fixed=c("pattern", "subject")` is equivalent to `fixed=TRUE` (the default). An empty vector is equivalent to `fixed=FALSE`. With `fixed="subject"`, ambiguities in the pattern only are interpreted as wildcards. With `fixed="pattern"`, ambiguities in the subject only are interpreted as wildcards. |
| `follow.index` | Whether the single integer returned by `which.isMatchingAt` (and related utils) should be the first *value* in `at` for which a match occurred, or its *index* in `at` (the default). |
| `auto.reduce.pattern` | Whether `pattern` should be effectively shortened by 1 letter, from its beginning for `which.isMatchingStartingAt` and from its end for `which.isMatchingEnding` for each successive `(at, max.mismatch)` "pair". |

## Details

A "match" of pattern P in subject S is a substring S' of S that is considered similar enough to P according to some distance (or metric) specified by the user. 2 distances are supported by most pattern matching functions in the Biostrings package. The first (and simplest) one is the "number of mismatching letters". It is defined only when the 2 strings to compare have the same length, so when this distance is used, only matches that have the same number of letters as P are considered. The second one is the "edit distance" (aka Levenshtein distance): it's the minimum number of operations needed to transform P into S', where an operation is an insertion, deletion, or substitution of a single letter. When this metric is used, matches can have a different number of letters than P.

The `neditAt` function implements these 2 distances. If `with.indels` is `FALSE` (the default), then the first distance is used i.e. `neditAt` returns the "number of mismatching letters" between the pattern P and the substring S' of S starting at the positions specified in `at` (note that `neditAt` is vectorized so a long vector of integers can be passed thru the `at` argument). If `with.indels` is `TRUE`, then the "edit distance" distance is used: for each position specified in `at`, P is compared to all the substrings S' of S starting at this position and the smallest distance is returned. Note that this distance is guaranteed to be reached for a substring of length < 2*length(P) so, of course, in practice, P only needs to be compared to a small number of substrings for every starting position.

## Value

`hasLetterAt`: A logical matrix with one row per element in `x` and one column per letter/position to check. When a specified position is invalid with respect to an element in `x` then the corresponding matrix element is set to NA.

`neditAt`: If `subject` is an XString object, then return an integer vector of the same length as `at`. If `subject` is an XStringSet object, then return the integer matrix with `length(at)` rows and `length(subject)` columns defined by:

```
sapply(unname(subject),
       function(x) neditAt(pattern, x, ...))
```

`neditStartingAt` is identical to `neditAt` except that the `at` argument is now called `starting.at`. `neditEndingAt` is similar to `neditAt` except that the `at` argument is now called `ending.at` and must contain the ending positions of the pattern relatively to the subject.

`isMatchingAt`: If `subject` is an XString object, then return the logical vector defined by:

```
min.mismatch <= neditAt(...) <= max.mismatch
```

If `subject` is an XStringSet object, then return the logical matrix with `length(at)` rows and `length(subject)` columns defined by:

```
sapply(unname(subject),
       function(x) isMatchingAt(pattern, x, ...))
```

`isMatchingStartingAt` is identical to `isMatchingAt` except that the `at` argument is now called `starting.at`. `isMatchingEndingAt` is similar to `isMatchingAt` except that the `at` argument is now called `ending.at` and must contain the ending positions of the pattern relatively to the subject.

`which.isMatchingAt`: The default behavior (`follow.index=FALSE`) is as follow. If `subject` is an XString object, then return the single integer defined by:

```
which(isMatchingAt(...))[1]
```

If `subject` is an XStringSet object, then return the integer vector defined by:

```
sapply(unname(subject),
       function(x) which.isMatchingAt(pattern, x, ...))
```

If `follow.index=TRUE`, then the returned value is defined by:

```
at[which.isMatchingAt(..., follow.index=FALSE)]
```

`which.isMatchingStartingAt` is identical to `which.isMatchingAt` except that the `at` argument is now called `starting.at`. `which.isMatchingEndingAt` is similar to `which.isMatchingAt` except that the `at` argument is now called `ending.at` and must contain the ending positions of the pattern relatively to the subject.

## See Also

`nucleotideFrequencyAt`, `matchPattern`, `matchPDict`, `matchLRPatterns`, `trimLRPatterns`, `IUPAC_CODE_MAP`, XString-class, align-utils

## Examples

```
## ---------------------------------------------------------------------
## hasLetterAt()
## ---------------------------------------------------------------------
x <- DNAStringSet(c("AAACGT", "AACGT", "ACGT", "TAGGA"))
hasLetterAt(x, "AAAAAA", 1:6)

## hasLetterAt() can be used to answer questions like: "which elements
## in 'x' have an A at position 2 and a G at position 4?"
q1 <- hasLetterAt(x, "AG", c(2, 4))
which(rowSums(q1) == 2)

## or "how many probes in the drosophila2 chip have T, G, T, A at
## position 2, 4, 13 and 20, respectively?"
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
q2 <- hasLetterAt(probes, "TGTA", c(2, 4, 13, 20))
sum(rowSums(q2) == 4)
## or "what's the probability to have an A at position 25 if there is
## one at position 13?"
q3 <- hasLetterAt(probes, "AACGT", c(13, 25, 25, 25, 25))
sum(q3[ , 1] & q3[ , 2]) / sum(q3[ , 1])
## Probabilities to have other bases at position 25 if there is an A
## at position 13:
sum(q3[ , 1] & q3[ , 3]) / sum(q3[ , 1])  # C
sum(q3[ , 1] & q3[ , 4]) / sum(q3[ , 1])  # G
sum(q3[ , 1] & q3[ , 5]) / sum(q3[ , 1])  # T

## See ?nucleotideFrequencyAt for another way to get those results.

## ---------------------------------------------------------------------
## neditAt() / isMatchingAt() / which.isMatchingAt()
## ---------------------------------------------------------------------
subject <- DNAString("GTATA")

## Pattern "AT" matches subject "GTATA" at position 3 (exact match)
```

```
neditAt("AT", subject, at=3)
isMatchingAt("AT", subject, at=3)

## ... but not at position 1
neditAt("AT", subject)
isMatchingAt("AT", subject)

## ... unless we allow 1 mismatching letter (inexact match)
isMatchingAt("AT", subject, max.mismatch=1)

## Here we look at 6 different starting positions and find 3 matches if
## we allow 1 mismatching letter
isMatchingAt("AT", subject, at=0:5, max.mismatch=1)

## No match
neditAt("NT", subject, at=1:4)
isMatchingAt("NT", subject, at=1:4)

## 2 matches if N is interpreted as an ambiguity (fixed=FALSE)
neditAt("NT", subject, at=1:4, fixed=FALSE)
isMatchingAt("NT", subject, at=1:4, fixed=FALSE)

## max.mismatch != 0 and fixed=FALSE can be used together
neditAt("NCA", subject, at=0:5, fixed=FALSE)
isMatchingAt("NCA", subject, at=0:5, max.mismatch=1, fixed=FALSE)

some_starts <- c(10:-10, NA, 6)
subject <- DNAString("ACGTGCA")
is_matching <- isMatchingAt("CAT", subject, at=some_starts, max.mismatch=1)
some_starts[is_matching]

which.isMatchingAt("CAT", subject, at=some_starts, max.mismatch=1)
which.isMatchingAt("CAT", subject, at=some_starts, max.mismatch=1,
                   follow.index=TRUE)

## ---------------------------------------------------------------------
## WITH INDELS
## ---------------------------------------------------------------------
subject <- BString("ABCDEFxxxCDEFxxxABBCDE")

neditAt("ABCDEF", subject, at=9)
neditAt("ABCDEF", subject, at=9, with.indels=TRUE)
isMatchingAt("ABCDEF", subject, at=9, max.mismatch=1, with.indels=TRUE)
isMatchingAt("ABCDEF", subject, at=9, max.mismatch=2, with.indels=TRUE)
neditAt("ABCDEF", subject, at=17)
neditAt("ABCDEF", subject, at=17, with.indels=TRUE)
neditEndingAt("ABCDEF", subject, ending.at=22)
neditEndingAt("ABCDEF", subject, ending.at=22, with.indels=TRUE)
```

---

## `maskMotif` *Masking by content (or by position)*

---

### Description

Functions for masking a sequence by content (or by position).

### Usage

```
maskMotif(x, motif, min.block.width=1)
mask(x, start=NA, end=NA, pattern)
```

### Arguments

| | |
|---|---|
| `x` | The sequence to mask. |
| `motif` | The motif to mask in the sequence. |
| `min.block.width` | The minimum width of the blocks to mask. |
| `start` | An integer vector containing the starting positions of the regions to mask. |
| `end` | An integer vector containing the ending positions of the regions to mask. |
| `pattern` | The motif to mask in the sequence. |

### Value

A MaskedXString object for `maskMotif` and an XStringViews object for `mask`.

### Author(s)

H. Pages

### See Also

`read.Mask`, XString-class, MaskedXString-class, XStringViews-class, MaskCollection-class

### Examples

```
## ---------------------------------------------------------------------
## EXAMPLE 1
## ---------------------------------------------------------------------

maskMotif(BString("AbcbbcbEEE"), "bcb")
maskMotif(BString("AbcbcbEEE"), "bcb")

## maskMotif() can be used in an incremental way to mask more than 1
## motif. Note that maskMotif() does not try to mask again what's
## already masked (i.e. the new mask will never overlaps with the
## previous masks) so the order in which the motifs are masked actually
## matters as it will affect the total set of masked positions.
x0 <- BString("AbcbEEEEbcbbEEEcbbcbc")
x1 <- maskMotif(x0, "E")
x1
x2 <- maskMotif(x1, "bcb")
x2
x3 <- maskMotif(x2, "b")
x3
## Note that inverting the order in which "b" and "bcb" are masked would
## lead to a different final set of masked positions.
## Also note that the order doesn't matter if the motifs to mask don't
## overlap (we assume that the motifs are unique) i.e. if the prefix of
## each motif is not the suffix of any other motif. This is of course
## the case when all the motifs have only 1 letter.
## ---------------------------------------------------------------------
## EXAMPLE 2
## ---------------------------------------------------------------------

x <- DNAString("ACACAACTAGATAGNACTNNGAGAGACGC")

## Mask the N-blocks
x1 <- maskMotif(x, "N")
x1
as(x1, "XStringViews")
gaps(x1)
as(gaps(x1), "XStringViews")

## Mask the AC-blocks
x2 <- maskMotif(x1, "AC")
x2
gaps(x2)

## Mask the GA-blocks
x3 <- maskMotif(x2, "GA", min.block.width=5)
x3  # masks 2 and 3 overlap
gaps(x3)

## ---------------------------------------------------------------------
## EXAMPLE 3
## ---------------------------------------------------------------------

library(BSgenome.Dmelanogaster.UCSC.dm3)
chrU <- Dmelanogaster$chrU
chrU
alphabetFrequency(chrU)
chrU <- maskMotif(chrU, "N")
chrU
alphabetFrequency(chrU)
as(chrU, "XStringViews")
as(gaps(chrU), "XStringViews")

mask2 <- Mask(mask.width=length(chrU), start=c(50000, 350000, 543900), width=25000)
names(mask2) <- "some ugly regions"
masks(chrU) <- append(masks(chrU), mask2)
chrU
as(chrU, "XStringViews")
as(gaps(chrU), "XStringViews")

## ---------------------------------------------------------------------
## EXAMPLE 4
## ---------------------------------------------------------------------
## Note that unlike maskMotif(), mask() returns an XStringViews object!

## masking "by position"
mask("AxyxyxBC", 2, 6)

## masking "by content"
mask("AxyxyxBC", "xyx")
noN_chrU <- mask(chrU, "N")
noN_chrU
alphabetFrequency(noN_chrU, collapse=TRUE)
```

## match-utils — *Utility functions operating on the matches returned by a high-level matching function*

### Description

Miscellaneous utility functions operating on the matches returned by a high-level matching function like `matchPattern`, `matchPDict`, etc...

### Usage

```
  mismatch(pattern, x, fixed=TRUE)
  nmatch(pattern, x, fixed=TRUE)
  nmismatch(pattern, x, fixed=TRUE)
  ## S4 method for signature 'MIndex':
coverage(x, shift=0L, width=NULL, weight=1L)
  ## S4 method for signature 'MaskedXString':
coverage(x, shift=0L, width=NULL, weight=1L)
```

### Arguments

| | |
|---|---|
| `pattern` | The pattern string. |
| `x` | An XStringViews object for `mismatch` (typically, one returned by `matchPattern(pattern, subject)`).<br>An MIndex object for `coverage`, or any object for which a `coverage` method is defined. See `?coverage`. |
| `fixed` | See `¿lowlevel-matching`'. |
| `shift, width` | See `?coverage`. |
| `weight` | An integer vector specifying how much each element in `x` counts. |

### Details

The `mismatch` function gives the positions of the mismatching letters of a given pattern relatively to its matches in a given subject.

The `nmatch` and `nmismatch` functions give the number of matching and mismatching letters produced by the `mismatch` function.

The `coverage` function computes the "coverage" of a subject by a given pattern or set of patterns.

### Value

`mismatch`: a list of integer vectors.

`nmismatch`: an integer vector containing the length of the vectors produced by `mismatch`.

`coverage`: an Rle object indicating the coverage of `x`. See `?coverage` for the details. If `x` is an MIndex object, the coverage of a given position in the underlying sequence (typically the subject used during the search that returned `x`) is the number of matches (or hits) it belongs to.

### See Also

lowlevel-matching, `matchPattern`, `matchPDict`, XString-class, XStringViews-class, MIndex-class, coverage, align-utils

## Examples

```
## ---------------------------------------------------------------------
## mismatch() / nmismatch()
## ---------------------------------------------------------------------
subject <- DNAString("ACGTGCA")
m <- matchPattern("NCA", subject, max.mismatch=1, fixed=FALSE)
mismatch("NCA", m)
nmismatch("NCA", m)

## ---------------------------------------------------------------------
## coverage()
## ---------------------------------------------------------------------
coverage(m)

## See ?matchPDict for examples of using coverage() on an MIndex object...
```

---

## `matchLRPatterns` *Find paired matches in a sequence*

---

### Description

The `matchLRPatterns` function finds paired matches in a sequence i.e. matches specified by a left pattern, a right pattern and a maximum distance between the left pattern and the right pattern.

### Usage

```
matchLRPatterns(Lpattern, Rpattern, max.gaplength, subject,
                max.Lmismatch=0, max.Rmismatch=0,
                with.Lindels=FALSE, with.Rindels=FALSE,
                Lfixed=TRUE, Rfixed=TRUE)
```

### Arguments

| | |
|---|---|
| `Lpattern` | The left part of the pattern. |
| `Rpattern` | The right part of the pattern. |
| `max.gaplength` | The max length of the gap in the middle i.e the max distance between the left and right parts of the pattern. |
| `subject` | An XString, XStringViews or MaskedXString object containing the target sequence. |
| `max.Lmismatch` | The maximum number of mismatching letters allowed in the left part of the pattern. If non-zero, an inexact matching algorithm is used (see the `matchPattern` function for more information). |
| `max.Rmismatch` | Same as `max.Lmismatch` but for the right part of the pattern. |
| `with.Lindels` | If `TRUE` then indels are allowed in the left part of the pattern. In that case `max.Lmismatch` is interpreted as the maximum "edit distance" allowed in the left part of the pattern. See the `with.indels` argument of the `matchPattern` function for more information. |

| | |
|---|---|
| `with.Rindels` | Same as `with.Lindels` but for the right part of the pattern. |
| `Lfixed` | Only with a DNAString or RNAString subject can a `Lfixed` value other than the default (`TRUE`) be used. With `Lfixed=FALSE`, ambiguities (i.e. letters from the IUPAC Extended Genetic Alphabet (see `IUPAC_CODE_MAP`) that are not from the base alphabet) in the left pattern _and_ in the subject are interpreted as wildcards i.e. they match any letter that they stand for. `Lfixed` can also be a character vector, a subset of `c("pattern", "subject")`. `Lfixed=c("pattern", "subject")` is equivalent to `Lfixed=TRUE` (the default). An empty vector is equivalent to `Lfixed=FALSE`. With `Lfixed="subject"`, ambiguities in the pattern only are interpreted as wildcards. With `Lfixed="pattern"`, ambiguities in the subject only are interpreted as wildcards. |
| `Rfixed` | Same as `Lfixed` but for the right part of the pattern. |

## Value

An XStringViews object containing all the matches, even when they are overlapping (see the examples below), and where the matches are ordered from left to right (i.e. by ascending starting position).

## Author(s)

H. Pages

## See Also

`matchPattern`, `matchProbePair`, `trimLRPatterns`, `findPalindromes`, `reverseComplement`, XString-class, XStringViews-class, MaskedXString-class

## Examples

```
library(BSgenome.Dmelanogaster.UCSC.dm3)
subject <- Dmelanogaster$chr3R
Lpattern <- "AGCTCCGAG"
Rpattern <- "TTGTTCACA"
matchLRPatterns(Lpattern, Rpattern, 500, subject) # 1 match

## Note that matchLRPatterns() will return all matches, even when they are
## overlapping:
subject <- DNAString("AAATTAACCCTT")
matchLRPatterns("AA", "TT", 0, subject) # 1 match
matchLRPatterns("AA", "TT", 1, subject) # 2 matches
matchLRPatterns("AA", "TT", 3, subject) # 3 matches
matchLRPatterns("AA", "TT", 7, subject) # 4 matches
```

---

`matchPDict` *Matching a dictionary of patterns against a reference*

---

## Description

A set of functions for finding all the occurrences (aka "matches" or "hits") of a set of patterns (aka the dictionary) in a reference sequence or set of reference sequences (aka the subject)

The following functions differ in what they return: `matchPDict` returns the "where" information i.e. the positions in the subject of all the occurrences of every pattern; `countPDict` returns the "how many times" information i.e. the number of occurrences for each pattern; and `whichPDict` returns the "who" information i.e. which patterns in the input dictionary have at least one match.

`vcountPDict` and `vwhichPDict` are vectorized versions of `countPDict` and `whichPDict`, respectively, that is, they work on a set of reference sequences in a vectorized fashion.

This man page shows how to use these functions (aka the `*PDict` functions) for exact matching of a constant width dictionary i.e. a dictionary where all the patterns have the same length (same number of nucleotides).

See ¿`matchPDict-inexact`‘ for how to use these functions for inexact matching or when the original dictionary has a variable width.

## Usage

```
matchPDict(pdict, subject,
           max.mismatch=0, min.mismatch=0, fixed=TRUE,
           algorithm="auto", verbose=FALSE)
countPDict(pdict, subject,
           max.mismatch=0, min.mismatch=0, fixed=TRUE,
           algorithm="auto", verbose=FALSE)
whichPDict(pdict, subject,
           max.mismatch=0, min.mismatch=0, fixed=TRUE,
           algorithm="auto", verbose=FALSE)

vcountPDict(pdict, subject,
            max.mismatch=0, min.mismatch=0, fixed=TRUE,
            algorithm="auto", collapse=FALSE, weight=1L,
            verbose=FALSE, ...)
vwhichPDict(pdict, subject,
            max.mismatch=0, min.mismatch=0, fixed=TRUE,
            algorithm="auto", verbose=FALSE)
```

## Arguments

`pdict` A PDict object containing the preprocessed dictionary.

All these functions also work with a dictionary that has not been preprocessed (in other words, the `pdict` argument can receive an XStringSet object). Of course, it won't be as fast as with a preprocessed dictionary, but it will generally be slightly faster than using `matchPattern`/`countPattern` or `vmatchPattern`/`vcountPatte`
in a "lapply/sapply loop", because, here, looping is done at the C-level.

`subject` An XString or MaskedXString object containing the subject sequence for `matchPDict`, `countPDict` and `whichPDict`.

An XStringSet object containing the subject sequences for `vcountPDict` and `vwhichPDict`.

For now, only subjects of base class DNAString are supported.

`max.mismatch, min.mismatch`

The maximum and minimum number of mismatching letters allowed (see `?isMatching` for the details). This man page focuses on exact matching of a constant width

| | |
|---|---|
| | dictionary so `max.mismatch=0` in the examples below. See ¿`matchPDict-inexact` ‘ for inexact matching. |
| `fixed` | Whether IUPAC ambiguity codes should be interpreted literally or not (see `?isMatching` for more information). This man page focuses on exact matching of a constant width dictionary so `fixed=TRUE` in the examples below. See ¿`matchPDict-inexact` ‘ for inexact matching. |
| `algorithm` | Ignored if `pdict` is a preprocessed dictionary (i.e. a PDict object). Otherwise, can be one of the following: `"auto"`, `"naive-exact"`, `"naive-inexact"`, `"boyer-moore"` or `"shift-or"`. See `?matchPattern` for more information. Note that `"indels"` is not supported for now. |
| `verbose` | `TRUE` or `FALSE`. |
| `collapse, weight` | `collapse` must be `FALSE`, `1`, or `2`.<br>If `collapse=FALSE` (the default), then `weight` is ignored and `vcountPDict` returns the full matrix of counts (`M0`). If `collapse=1`, then `M0` is collapsed "horizontally" i.e. it is turned into a vector with `length` equal to `length(pdict)`. If `weight=1L` (the default), then this vector is defined by `rowSums(M0)`. If `collapse=2`, then `M0` is collapsed "vertically" i.e. it is turned into a vector with `length` equal to `length(subject)`. If `weight=1L` (the default), then this vector is defined by `colSums(M0)`.<br>If `collapse=1` or `collapse=2`, then the elements in `subject` (`collapse=1`) or in `pdict` (`collapse=2`) can be weighted thru the `weight` argument. In that case, the returned vector is defined by `M0 %*% rep(weight, length.out=length(su` and `rep(weight, length.out=length(pdict)) %*% M0`, respectively. |
| `...` | Additional arguments for methods. |

## Details

In this man page, we assume that you know how to preprocess a dictionary of DNA patterns that can then be used with any of the `*PDict` functions described here. Please see `?PDict` if you don't.

When using the `*PDict` functions for exact matching of a constant width dictionary, the standard way to preprocess the original dictionary is by calling the `PDict` constructor on it with no extra arguments. This returns the preprocessed dictionary in a PDict object that can be used with any of the `*PDict` functions.

## Value

If `M` denotes the number of patterns in the `pdict` argument (`M <- length(pdict)`), then `matchPDict` returns an MIndex object of length `M`, and `countPDict` an integer vector of length `M`.

`whichPDict` returns an integer vector made of the indices of the patterns in the `pdict` argument that have at least one match.

If `N` denotes the number of sequences in the `subject` argument (`N <- length(subject)`), then `vcountPDict` returns an integer matrix with `M` rows and `N` columns, unless the `collapse` argument is used. In that case, depending on the type of `weight`, an integer or numeric vector is returned (see above for the details).

`vwhichPDict` returns a list of `N` integer vectors.

### Author(s)

H. Pages

### References

Aho, Alfred V.; Margaret J. Corasick (June 1975). "Efficient string matching: An aid to bibliographic search". Communications of the ACM 18 (6): 333-340.

### See Also

PDict-class, MIndex-class, matchPDict-inexact, `isMatching`, `coverage,MIndex-method`, `matchPattern`, `alphabetFrequency`, DNAStringSet-class, XStringViews-class, MaskedDNAString-class

### Examples

```
## ---------------------------------------------------------------------
## A. A SIMPLE EXAMPLE OF EXACT MATCHING
## ---------------------------------------------------------------------

## Creating the pattern dictionary:
library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe)
dict0                                # The original dictionary.
length(dict0)                        # Hundreds of thousands of patterns.
pdict0 <- PDict(dict0)               # Store the original dictionary in
                                     # a PDict object (preprocessing).

## Using the pattern dictionary on chromosome 3R:
library(BSgenome.Dmelanogaster.UCSC.dm3)
chr3R <- Dmelanogaster$chr3R         # Load chromosome 3R
chr3R
mi0 <- matchPDict(pdict0, chr3R)     # Search...

## Looking at the matches:
start_index <- startIndex(mi0)       # Get the start index.
length(start_index)                  # Same as the original dictionary.
start_index[[8220]]                  # Starts of the 8220th pattern.
end_index <- endIndex(mi0)           # Get the end index.
end_index[[8220]]                    # Ends of the 8220th pattern.
count_index <- countIndex(mi0)       # Get the number of matches per pattern.
count_index[[8220]]
mi0[[8220]]                          # Get the matches for the 8220th pattern.
start(mi0[[8220]])                   # Equivalent to startIndex(mi0)[[8220]].
sum(count_index)                     # Total number of matches.
table(count_index)
i0 <- which(count_index == max(count_index))
pdict0[[i0]]                         # The pattern with most occurrences.
mi0[[i0]]                            # Its matches as an IRanges object.
Views(chr3R, mi0[[i0]])              # And as an XStringViews object.

## Get the coverage of the original subject:
cov3R <- as.integer(coverage(mi0, width=length(chr3R)))
max(cov3R)
mean(cov3R)
sum(cov3R != 0) / length(cov3R)      # Only 2.44% of chr3R is covered.
```

```
if (interactive()) {
  plotCoverage <- function(cx, start, end)
  {
    plot.new()
    plot.window(c(start, end), c(0, 20))
    axis(1)
    axis(2)
    axis(4)
    lines(start:end, cx[start:end], type="l")
  }
  plotCoverage(cov3R, 27600000, 27900000)
}

## ---------------------------------------------------------------------
## B. NAMING THE PATTERNS
## ---------------------------------------------------------------------

## The names of the original patterns, if any, are propagated to the
## PDict and MIndex objects:
names(dict0) <- mkAllStrings(letters, 4)[seq_len(length(dict0))]
dict0
dict0[["abcd"]]
pdict0n <- PDict(dict0)
names(pdict0n)[1:30]
pdict0n[["abcd"]]
mi0n <- matchPDict(pdict0n, chr3R)
names(mi0n)[1:30]
mi0n[["abcd"]]

## This is particularly useful when unlisting an MIndex object:
unlist(mi0)[1:10]
unlist(mi0n)[1:10]  # keep track of where the matches are coming from

## ---------------------------------------------------------------------
## C. PERFORMANCE
## ---------------------------------------------------------------------

## If getting the number of matches is what matters only (without
## regarding their positions), then countPDict() will be faster,
## especially when there is a high number of matches:

count_index0 <- countPDict(pdict0, chr3R)
stopifnot(identical(count_index0, count_index))

if (interactive()) {
  ## What's the impact of the dictionary width on performance?
  ## Below is some code that can be used to figure out (will take a long
  ## time to run). For different widths of the original dictionary, we
  ## look at:
  ##   o pptime: preprocessing time (in sec.) i.e. time needed for
  ##             building the PDict object from the truncated input
  ##             sequences;
  ##   o nnodes: nb of nodes in the resulting Aho-Corasick tree;
  ##   o nupatt: nb of unique truncated input sequences;
  ##   o matchtime: time (in sec.) needed to find all the matches;
  ##   o totalcount: total number of matches.
  getPDictStats <- function(dict, subject)
  {
    ans_width <- width(dict[1])
    ans_pptime <- system.time(pdict <- PDict(dict))[["elapsed"]]
    pptb <- pdict@threeparts@pptb
    ans_nnodes <- nnodes(pptb)
    ans_nupatt <- sum(!duplicated(pdict))
    ans_matchtime <- system.time(
                       mi0 <- matchPDict(pdict, subject)
                     )[["elapsed"]]
    ans_totalcount <- sum(countIndex(mi0))
    list(
      width=ans_width,
      pptime=ans_pptime,
      nnodes=ans_nnodes,
      nupatt=ans_nupatt,
      matchtime=ans_matchtime,
      totalcount=ans_totalcount
    )
  }
  stats <- lapply(8:25,
               function(width)
                   getPDictStats(DNAStringSet(dict0, end=width), chr3R))
  stats <- data.frame(do.call(rbind, stats))
  stats
}

## ---------------------------------------------------------------------
## D. USING A NON-PREPROCESSED DICTIONARY
## ---------------------------------------------------------------------

dict3 <- DNAStringSet(mkAllStrings(DNA_BASES, 3))  # all trinucleotides
dict3
pdict3 <- PDict(dict3)

## The 3 following calls are equivalent (from faster to slower):
res3a <- countPDict(pdict3, chr3R)
res3b <- countPDict(dict3, chr3R)
res3c <- sapply(dict3,
             function(pattern) countPattern(pattern, chr3R))
stopifnot(identical(res3a, res3b))
stopifnot(identical(res3a, res3c))

## One reason for using a non-preprocessed dictionary is to get rid of
## all the constraints associated with preprocessing, e.g., when
## preprocessing with \code{\link{PDict}}, the input dictionary must
## be DNA and a Trusted Band must be defined (explicitly or implicitly).
## See \code{?\link{PDict}} for more information about these constraints.
## In particular, using a non-preprocessed dictionary can be
## useful for the kind of inexact matching that can't be achieved
## with a \link{PDict} object (if performance is not an issue).
## See \code{?`\link{matchPDict-inexact}`} for more information about
## inexact matching.

dictD <- xscat(dict3, "N", reverseComplement(dict3))

## The 2 following calls are equivalent (from faster to slower):
resDa <- matchPDict(dictD, chr3R, fixed=FALSE)
```

```
resDb <- sapply(dictD,
             function(pattern) matchPattern(pattern, chr3R, fixed=FALSE))
stopifnot(all(sapply(seq_len(length(dictD)),
                  function(i) identical(resDa[[i]], IRanges(resDb[[i]])))))

## ---------------------------------------------------------------------
## E. vcountPDict()
## ---------------------------------------------------------------------
subject <- Dmelanogaster$upstream1000[1:100]
subject
mat1 <- vcountPDict(pdict0, subject)
dim(mat1)  # length(pdict0) x length(subject)
nhit_per_probe <- rowSums(mat1)
table(nhit_per_probe)

## Without vcountPDict(), 'mat1' could have been computed with:
mat2 <- sapply(unname(subject), function(x) countPDict(pdict0, x))
stopifnot(identical(mat1, mat2))
## but using vcountPDict() is faster (10x or more, depending of the
## average length of the sequences in 'subject').

if (interactive()) {
  ## This will fail (with message "allocMatrix: too many elements
  ## specified") because, on most platforms, vectors and matrices in R
  ## are limited to 2^31 elements:
  subject <- Dmelanogaster$upstream1000
  vcountPDict(pdict0, subject)
  length(pdict0) * length(Dmelanogaster$upstream1000)
  1 * length(pdict0) * length(Dmelanogaster$upstream1000)  # > 2^31
  ## But this will work:
  nhit_per_seq <- vcountPDict(pdict0, subject, collapse=2)
  sum(nhit_per_seq >= 1)  # nb of subject sequences with at least 1 hit
  table(nhit_per_seq)
  which(nhit_per_seq == 37)  # 603
  sum(countPDict(pdict0, subject[[603]]))  # 37
}

## ---------------------------------------------------------------------
## F. RELATIONSHIP BETWEEN vcountPDict(), countPDict() AND
## vcountPattern()
## ---------------------------------------------------------------------
pdict3 <- PDict(dict3)
subject <- Dmelanogaster$upstream1000
subject

## The 4 following calls are equivalent (from faster to slower):
mat3a <- vcountPDict(pdict3, subject)
mat3b <- vcountPDict(dict3, subject)
mat3c <- sapply(dict3,
             function(pattern) vcountPattern(pattern, subject))
mat3d <- sapply(unname(subject),
             function(x) countPDict(pdict3, x))
stopifnot(identical(mat3a, mat3b))
stopifnot(identical(mat3a, t(mat3c)))
stopifnot(identical(mat3a, mat3d))

## The 3 following calls are equivalent (from faster to slower):
```

```
nhitpp3a <- vcountPDict(pdict3, subject, collapse=1)  # rowSums(mat3a)
nhitpp3b <- vcountPDict(dict3, subject, collapse=1)
nhitpp3c <- sapply(dict3,
                function(pattern) sum(vcountPattern(pattern, subject)))
stopifnot(identical(nhitpp3a, nhitpp3b))
stopifnot(identical(nhitpp3a, nhitpp3c))

## The 3 following calls are equivalent (from faster to slower):
nhitps3a <- vcountPDict(pdict3, subject, collapse=2)  # colSums(mat3a)
nhitps3b <- vcountPDict(dict3, subject, collapse=2)
nhitps3c <- sapply(unname(subject),
                function(x) sum(countPDict(pdict3, x)))
stopifnot(identical(nhitps3a, nhitps3b))
stopifnot(identical(nhitps3a, nhitps3c))

## ---------------------------------------------------------------------
## G. vwhichPDict()
## ---------------------------------------------------------------------
## The 4 following calls are equivalent (from faster to slower):
vwp3a <- vwhichPDict(pdict3, subject)
vwp3b <- vwhichPDict(dict3, subject)
vwp3c <- lapply(seq_len(ncol(mat3a)), function(j) which(mat3a[ , j] != 0L))
vwp3d <- lapply(unname(subject), function(x) whichPDict(pdict3, x))
stopifnot(identical(vwp3a, vwp3b))
stopifnot(identical(vwp3a, vwp3c))
stopifnot(identical(vwp3a, vwp3d))

table(sapply(vwp3a, length))
which.min(sapply(vwp3a, length))
## Get the trinucleotides not represented in reference sequence 9181:
dict3[-vwp3a[[9181]]]  # 21 trinucleotides

## ---------------------------------------------------------------------
## H. MAPPING PROBE SET IDS BETWEEN CHIPS WITH vwhichPDict()
## ---------------------------------------------------------------------
## Here we show a simple (and very naive) algorithm for mapping probe
## set IDs between the hgu95av2 and hgu133a chips (Affymetrix).
## 2 probe set IDs are considered mapped iff they share at least one
## probe.
## WARNING: This example takes about 25 minutes to run.
if (interactive()) {

  library(hgu95av2probe)
  library(hgu133aprobe)
  probes1 <- DNAStringSet(hgu95av2probe)
  probes2 <- DNAStringSet(hgu133aprobe)
  pdict2 <- PDict(probes2)

  ## Get the mapping from probes1 to probes2 (based on exact matching):
  map1to2 <- vwhichPDict(pdict2, probes1)  # takes about 10 minutes

  ## The following helper function uses the probe level mapping to induce
  ## the mapping at the probe set IDs level (from hgu95av2 to hgu133a).
  ## To keep things simple, 2 probe set IDs are considered mapped iff
  ## each of them contains at least one probe mapped to one probe of
  ## the other:
  mapProbeSetIDs1to2 <- function(psID)
```

```
      unique(hgu133aprobe$Probe.Set.Name[unlist(
        map1to2[hgu95av2probe$Probe.Set.Name == psID]
      )])

    ## Use the helper function to build the complete mapping:
    psIDs1 <- unique(hgu95av2probe$Probe.Set.Name)
    mapPSIDs1to2 <- lapply(psIDs1, mapProbeSetIDs1to2)  # about 3 min.
    names(mapPSIDs1to2) <- psIDs1

    ## Do some basic stats:
    table(sapply(mapPSIDs1to2, length))

    ## [ADVANCED USERS ONLY]
    ## An alternative that is slightly faster is to put all the probes
    ## (hgu95av2 + hgu133a) in a single PDict object and then query its
    ## 'dups0' slot directly. This slot is a Dups object containing the
    ## mapping between duplicated patterns.
    ## Note that we can do this only because all the probes have the
    ## same length (25) and because we are doing exact matching:

    probes12 <- DNAStringSet(c(hgu95av2probe$sequence, hgu133aprobe$sequence))
    pdict12 <- PDict(probes12)
    dups0 <- pdict12@dups0

    mapProbeSetIDs1to2alt <- function(psID)
    {
      ii1 <- unique(togroup(dups0, which(hgu95av2probe$Probe.Set.Name == psID)))
      ii2 <- members(dups0, ii1) - length(probes1)
      ii2 <- ii2[ii2 >= 1L]
      unique(hgu133aprobe$Probe.Set.Name[ii2])
    }

    mapPSIDs1to2alt <- lapply(psIDs1, mapProbeSetIDs1to2alt)  # about 10 min.
    names(mapPSIDs1to2alt) <- psIDs1

    ## 'mapPSIDs1to2alt' and 'mapPSIDs1to2' contain the same mapping:
    stopifnot(identical(lapply(mapPSIDs1to2alt, sort),
                        lapply(mapPSIDs1to2, sort)))
  }
```

## `matchPDict-inexact` *Inexact matching with matchPDict()/countPDict()/whichPDict()*

### Description

The `matchPDict`, `countPDict` and `whichPDict` functions efficiently find the occurrences in a text (the subject) of all patterns stored in a preprocessed dictionary.

This man page shows how to use these functions for inexact (or fuzzy) matching or when the original dictionary has a variable width.

See `?matchPDict` for how to use these functions for exact matching of a constant width dictionary i.e. a dictionary where all the patterns have the same length (same number of nucleotides).

## Details

In this man page, we assume that you know how to preprocess a dictionary of DNA patterns that can then be used with `matchPDict`, `countPDict` or `whichPDict`. Please see `?PDict` if you don't.

`matchPDict` and family support different kinds of inexact matching but with some restrictions. Inexact matching is controlled via the definition of a Trusted Band during the preprocessing step and/or via the `max.mismatch`, `min.mismatch` and `fixed` arguments. Defining a Trusted Band is also required when the original dictionary is not rectangular (variable width), even for exact matching. See `?PDict` for how to define a Trusted Band.

Here is how `matchPDict` and family handle the Trusted Band defined on `pdict`:

- (1) Find all the exact matches of all the elements in the Trusted Band.
- (2) For each element in the Trusted Band that has at least one exact match, compare the head and the tail of this element with the flanking sequences of the matches found in (1).

Note that the number of exact matches found in (1) will decrease exponentially with the width of the Trusted Band. Here is a simple guideline in order to get reasonably good performance: if TBW is the width of the Trusted Band (`TBW <- tb.width(pdict)`) and L the number of letters in the subject (`L <- nchar(subject)`), then `L / (4^TBW)` should be kept as small as possible, typically < 10 or 20.

In addition, when a Trusted Band has been defined during preprocessing, then `matchPDict` and family can be called with `fixed=FALSE`. In this case, IUPAC ambiguity codes in the head or the tail of the PDict object are treated as ambiguities.

Finally, `fixed="pattern"` can be used to indicate that IUPAC ambiguity codes in the subject should be treated as ambiguities. It only works if the density of codes is not too high. It works whether or not a Trusted Band has been defined on `pdict`.

## Author(s)

H. Pages

## References

Aho, Alfred V.; Margaret J. Corasick (June 1975). "Efficient string matching: An aid to bibliographic search". Communications of the ACM 18 (6): 333-340.

## See Also

PDict-class, MIndex-class, matchPDict

## Examples

```
## ---------------------------------------------------------------------
## A. USING AN EXPLICIT TRUSTED BAND
## ---------------------------------------------------------------------

library(drosophila2probe)
dict0 <- DNAStringSet(drosophila2probe)
dict0  # the original dictionary

## Preprocess the original dictionary by defining a Trusted Band that
## spans nucleotides 1 to 9 of each pattern.
pdict9 <- PDict(dict0, tb.end=9)
```

```
  pdict9
  tail(pdict9)
  sum(duplicated(pdict9))
  table(patternFrequency(pdict9))

  library(BSgenome.Dmelanogaster.UCSC.dm3)
  chr3R <- Dmelanogaster$chr3R
  chr3R
  table(countPDict(pdict9, chr3R, max.mismatch=1))
  table(countPDict(pdict9, chr3R, max.mismatch=3))
  table(countPDict(pdict9, chr3R, max.mismatch=5))

  ## ---------------------------------------------------------------------
  ## B. COMPARISON WITH EXACT MATCHING
  ## ---------------------------------------------------------------------

  ## When the original dictionary is of constant width, exact matching
  ## (i.e. 'max.mismatch=0' and 'fixed=TRUE) will be more efficient with
  ## a full-width Trusted Band (i.e. a Trusted Band that covers the entire
  ## dictionary) than with a Trusted Band of width < width(dict0).
  pdict0 <- PDict(dict0)
  count0 <- countPDict(pdict0, chr3R)
  count0b <- countPDict(pdict9, chr3R, max.mismatch=0)
  identical(count0b, count0)  # TRUE

  ## ---------------------------------------------------------------------
  ## C. USING AN EXPLICIT TRUSTED BAND ON A VARIABLE WIDTH DICTIONARY
  ## ---------------------------------------------------------------------

  ## Here is a small variable width dictionary that contains IUPAC
  ## ambiguities (pattern 1 and 3 contain an N):
  dict0 <- DNAStringSet(c("TACCNG", "TAGT", "CGGNT", "AGTAG", "TAGT"))
  ## (Note that pattern 2 and 5 are identical.)

  ## If we only want to do exact matching, then it is recommended to use
  ## the widest possible Trusted Band i.e. to set its width to
  ## 'min(width(dict0))' because this is what will give the best
  ## performance. However, when 'dict0' contains IUPAC ambiguities (like
  ## in our case), it could be that one of them is falling into the
  ## Trusted Band so we get an error (only base letters can go in the
  ## Trusted Band for now):
  ## Not run:
    PDict(dict0, tb.end=min(width(dict0)))  # Error!

## End(Not run)

  ## In our case, the Trusted Band cannot be wider than 3:
  pdict <- PDict(dict0, tb.end=3)
  tail(pdict)

  subject <- DNAString("TAGTACCAGTTTCGGG")

  m <- matchPDict(pdict, subject)
  countIndex(m)  # pattern 2 and 5 have 1 exact match
  m[[2]]

  ## We can take advantage of the fact that our Trusted Band doesn't cover
  ## the entire dictionary to allow inexact matching on the uncovered parts
  ## (the tail in our case):

  m <- matchPDict(pdict, subject, fixed=FALSE)
  countIndex(m)  # now pattern 1 has 1 match too
  m[[1]]

  m <- matchPDict(pdict, subject, max.mismatch=1)
  countIndex(m)  # now pattern 4 has 1 match too
  m[[4]]

  m <- matchPDict(pdict, subject, max.mismatch=1, fixed=FALSE)
  countIndex(m)  # now pattern 3 has 1 match too
  m[[3]]  # note that this match is "out of limit"
  Views(subject, m[[3]])

  m <- matchPDict(pdict, subject, max.mismatch=2)
  countIndex(m)  # pattern 4 gets 1 additional match
  m[[4]]

  ## Unlist all matches:
  unlist(m)

  ## ---------------------------------------------------------------------
  ## D. WITH IUPAC AMBIGUITY CODES IN THE SUBJECT
  ## ---------------------------------------------------------------------
  pdict <- PDict(c("ACAC", "TCCG"))
  as.list(matchPDict(pdict, DNAString("ACNCCGT")))
  as.list(matchPDict(pdict, DNAString("ACNCCGT"), fixed="pattern"))
  as.list(matchPDict(pdict, DNAString("ACWCCGT"), fixed="pattern"))
  as.list(matchPDict(pdict, DNAString("ACRCCGT"), fixed="pattern"))
  as.list(matchPDict(pdict, DNAString("ACKCCGT"), fixed="pattern"))

  dict <- DNAStringSet(c("TTC", "CTT"))
  pdict <- PDict(dict)
  subject <- DNAString("CYTCACTTC")
  mi1 <- matchPDict(pdict, subject, fixed="pattern")
  mi2 <- matchPDict(dict, subject, fixed="pattern")
  stopifnot(identical(as.list(mi1), as.list(mi2)))
```

---

## matchPWM — *PWM creating, matching, and related utilities*

### Description

Position Weight Matrix (PWM) creating, matching, and related utilities for DNA data. (PWM for amino acid sequences are not supported.)

### Usage

```
PWM(x, type = c("log2probratio", "prob"),
    prior.params = c("A"=0.25, "C"=0.25, "G"=0.25, "T"=0.25))

matchPWM(pwm, subject, min.score="80%", ...)
```

```
  countPWM(pwm, subject, min.score="80%", ...)
  PWMscoreStartingAt(pwm, subject, starting.at=1)

  ## Utility functions for basic manipulation of the Position Weight Matrix
  maxWeights(x)
  minWeights(x)
  maxScore(x)
  minScore(x)
  unitScale(x)
  ## S4 method for signature 'matrix':
reverseComplement(x, ...)
```

## Arguments

| | |
|---|---|
| `x` | For `PWM` a character string or DNAStringSet whose elements all have the same number of characters.<br>For `maxWeights`, `minWeights`, `maxScore`, `minScore`, `unitScale` , and `reverseComplement` a numeric matrix with row names A, C, G and T representing a Position Weight Matrix. |
| `type` | The type of position weight matrix, either "log2probratio" or "prob". See Details section for more information. |
| `prior.params` | A positive numeric vector, which represents the parameters of the Dirichlet conjugate prior, with names A, C, G, and T. See Details section for more information. |
| `pwm` | A numeric matrix with row names A, C, G and T representing a Position Weight Matrix. |
| `subject` | An DNAString, XStringViews or MaskedDNAString object for `matchPWM` and `countPWM`.<br>A DNAString object containing the subject sequence. |
| `min.score` | The minimum score for counting a match. Can be given as a character string containing a percentage (e.g. `"85%"`) of the highest possible score or as a single number. |
| `starting.at` | An integer vector specifying the starting positions of the Position Weight Matrix relatively to the subject. |
| `...` | Additional arguments for methods. |

## Details

The `PWM` function uses a multinomial model with a Dirichlet conjugate prior to calculate the estimated probability of base b at position i. As mentioned in the Arguments section, `prior.params` supplies the parameters for the DNA bases A, C, G, and T in the Dirichlet prior. These values result in a position independent initial estimate of the probabilities for the bases to be `priorProbs = prior.params/sum(prior.params)` and the posterior (data infused) estimate for the probabilities for the bases in each of the positions to be `postProbs = (consensusMatrix(x) + prior.params)/(length(x) + sum(prior.params))`. When `type = "log2probratio"`, the PWM = `unitScale(log2(postProbs/priorProbs))`. When `type = "prob"`, the PWM = `unitScale(postProbs)`.

## Value

A numeric matrix representing the Position Weight Matrix for `PWM`.

A numeric vector containing the Position Weight Matrix-based scores for `PWMscoreStartingAt`.

An XStringViews object for `matchPWM`.

A single integer for `countPWM`.

A vector containing the max weight for each position in `pwm` for `maxWeights`.

A vector containing the min weight for each position in `pwm` for `minWeights`.

The highest possible score for a given Position Weight Matrix for `maxScore`.

The lowest possible score for a given Position Weight Matrix for `maxScore`.

The modified numeric matrix given by `(x - minScore(x)/ncol(x))/(maxScore(x) - minScore(x))` for `unitScale`.

A PWM obtained by reverting the column order in PWM `x` and by reassigning each row to its complementary nucleotide for `reverseComplement`.

## Author(s)

H. Pages and P. Aboyoun

## References

Wasserman, WW, Sandelin, A., (2004) Applied bioinformatics for the identification of regulatory elements, Nat Rev Genet., 5(4):276-87.

## See Also

`matchPattern`, `reverseComplement`, DNAString-class, XStringViews-class

## Examples

```
## Data setup
data(HNF4alpha)
library(BSgenome.Dmelanogaster.UCSC.dm3)
chr3R <- Dmelanogaster$chr3R
chr3R

## Create a PWM and perform some general routines
pwm <- PWM(HNF4alpha)
round(pwm, 2)
maxWeights(pwm)
maxScore(pwm)
reverseComplement(pwm)

## Score the first 5 positions
PWMscoreStartingAt(pwm, unmasked(chr3R), starting.at=1:5)

## Match the plus strand
matchPWM(pwm, chr3R)
countPWM(pwm, chr3R)

## Match the minus strand
matchPWM(reverseComplement(pwm), chr3R)
```

`matchPattern` *String searching functions*

## Description

A set of functions for finding all the occurrences (aka "matches" or "hits") of a given pattern (typically short) in a (typically long) reference sequence or set of reference sequences (aka the subject)

## Usage

```
matchPattern(pattern, subject,
             max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
             algorithm="auto")
countPattern(pattern, subject,
             max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
             algorithm="auto")
vmatchPattern(pattern, subject,
              max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
              algorithm="auto", ...)
vcountPattern(pattern, subject,
              max.mismatch=0, min.mismatch=0, with.indels=FALSE, fixed=TRUE,
              algorithm="auto", ...)
```

## Arguments

`pattern` The pattern string.

`subject` An XString, XStringViews or MaskedXString object for `matchPattern` and `countPattern`.

An XStringSet or XStringViews object for `vmatchPattern` and `vcountPattern`.

`max.mismatch, min.mismatch`

The maximum and minimum number of mismatching letters allowed (see ¿`lowlevel-matching`‘ for the details). If non-zero, an algorithm that supports inexact matching is used.

`with.indels` If `TRUE` then indels are allowed. In that case, `min.mismatch` must be `0` and `max.mismatch` is interpreted as the maximum "edit distance" allowed between the pattern and a match. Note that in order to avoid pollution by redundant matches, only the "best local matches" are returned. Roughly speaking, a "best local match" is a match that is locally both the closest (to the pattern P) and the shortest. More precisely, a substring S' of the subject S is a "best local match" iff:

```
(a) nedit(P, S') <= max.mismatch
(b) for every substring S1 of S':
        nedit(P, S1) > nedit(P, S')
(c) for every substring S2 of S that contains S':
        nedit(P, S2) >= nedit(P, S')
```

One nice property of "best local matches" is that their first and last letters are guaranteed to be aligned with letters in P (i.e. they match letters in P).

| | |
|---|---|
| `fixed` | If `TRUE` (the default), an IUPAC ambiguity code in the pattern can only match the same code in the subject, and vice versa. If `FALSE`, an IUPAC ambiguity code in the pattern can match any letter in the subject that is associated with the code, and vice versa. See ¿`lowlevel-matching`‘ for more information. |
| `algorithm` | One of the following: `"auto"`, `"naive-exact"`, `"naive-inexact"`, `"boyer-moore"`, `"shift-or"` or `"indels"`. |
| `...` | Additional arguments for methods. |

## Details

Available algorithms are: “naive exact”, “naive inexact”, “Boyer-Moore-like”, “shift-or” and “indels”. Not all of them can be used in all situations: restrictions apply depending on the "search criteria" i.e. on the values of the `pattern`, `subject`, `max.mismatch`, `min.mismatch`, `with.indels` and `fixed` arguments.

It is important to note that the `algorithm` argument is not part of the search criteria. This is because the supported algorithms are interchangeable, that is, if 2 different algorithms are compatible with a given search criteria, then choosing one or the other will not affect the result (but will most likely affect the performance). So there is no "wrong choice" of algorithm (strictly speaking).

Using `algorithm="auto"` (the default) is recommended because then the best suited algorithm will automatically be selected among the set of algorithms that are valid for the given search criteria.

## Value

An XStringViews object for `matchPattern`.

A single integer for `countPattern`.

An MIndex object for `vmatchPattern`.

An integer vector for `vcountPattern`, with each element in the vector corresponding to the number of matches in the corresponding element of `subject`.

## Note

Use `matchPDict` if you need to match a (big) set of patterns against a reference sequence.

Use `pairwiseAlignment` if you need to solve a (Needleman-Wunsch) global alignment, a (Smith-Waterman) local alignment, or an (ends-free) overlap alignment problem.

## See Also

lowlevel-matching, `matchPDict`, `pairwiseAlignment`, `mismatch`, `matchLRPatterns`, `matchProbePair`, `maskMotif`, `alphabetFrequency`, XStringViews-class, MIndex-class

## Examples

```
## ---------------------------------------------------------------------
## A. matchPattern()/countPattern()
## ---------------------------------------------------------------------

## A simple inexact matching example with a short subject:
x <- DNAString("AAGCGCGATATG")
m1 <- matchPattern("GCNNNAT", x)
m1
m2 <- matchPattern("GCNNNAT", x, fixed=FALSE)
m2
```

```
as.matrix(m2)

## With DNA sequence of yeast chromosome number 1:
data(yeastSEQCHR1)
yeast1 <- DNAString(yeastSEQCHR1)
PpiI <- "GAACNNNNNCTC" # a restriction enzyme pattern
match1.PpiI <- matchPattern(PpiI, yeast1, fixed=FALSE)
match2.PpiI <- matchPattern(PpiI, yeast1, max.mismatch=1, fixed=FALSE)

## With a genome containing isolated Ns:
library(BSgenome.Celegans.UCSC.ce2)
chrII <- Celegans[["chrII"]]
alphabetFrequency(chrII)
matchPattern("N", chrII)
matchPattern("TGGGTGTCTTT", chrII) # no match
matchPattern("TGGGTGTCTTT", chrII, fixed=FALSE) # 1 match

## Using wildcards ("N") in the pattern on a genome containing N-blocks:
library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- maskMotif(Dmelanogaster$chrX, "N")
as(chrX, "XStringViews") # 4 non masked regions
matchPattern("TTTATGNTTGGTA", chrX, fixed=FALSE)
## Can also be achieved with no mask:
masks(chrX) <- NULL
matchPattern("TTTATGNTTGGTA", chrX, fixed="subject")

## ---------------------------------------------------------------------
## B. vmatchPattern()/vcountPattern()
## ---------------------------------------------------------------------

Ebox <- DNAString("CANNTG")
subject <- Celegans$upstream5000
mindex <- vmatchPattern(Ebox, subject, fixed=FALSE)
count_index <- countIndex(mindex)  # Get the number of matches per
                                   # subject element.
sum(count_index)  # Total number of matches.
table(count_index)
i0 <- which(count_index == max(count_index))
subject[i0]  # The subject element with most matches.

## The matches in 'subject[i0]' as an IRanges object:
mindex[[i0]]
## The matches in 'subject[i0]' as an XStringViews object:
Views(subject[[i0]], mindex[[i0]])

## ---------------------------------------------------------------------
## C. WITH INDELS
## ---------------------------------------------------------------------
library(BSgenome.Celegans.UCSC.ce2)
pattern <- DNAString("ACGGACCTAATGTTATC")
subject <- Celegans$chrI

## Allowing up to 2 mismatching letters doesn't give any match:
matchPattern(pattern, subject, max.mismatch=2)

## But allowing up to 2 edit operations gives 3 matches:
system.time(m <- matchPattern(pattern, subject, max.mismatch=2, with.indels=TRUE))
```

```
m

## pairwiseAlignment() returns the (first) best match only:
if (interactive()) {
  mat <- nucleotideSubstitutionMatrix(match=1, mismatch=0, baseOnly=TRUE)
  ## Note that this call to pairwiseAlignment() will need to
  ## allocate 733.5 Mb of memory (i.e. length(pattern) * length(subject)
  ## * 3 bytes).
  system.time(pwa <- pairwiseAlignment(pattern, subject, type="local",
                                       substitutionMatrix=mat,
                                       gapOpening=0, gapExtension=1))
  pwa
}

## Only "best local matches" are reported:
  ## - with deletions in the subject
subject <- BString("ACDEFxxxCDEFxxxABCE")
matchPattern("ABCDEF", subject, max.mismatch=2, with.indels=TRUE)
matchPattern("ABCDEF", subject, max.mismatch=2)
  ## - with insertions in the subject
subject <- BString("AiBCDiEFxxxABCDiiFxxxAiBCDEFxxxABCiDEF")
matchPattern("ABCDEF", subject, max.mismatch=2, with.indels=TRUE)
matchPattern("ABCDEF", subject, max.mismatch=2)
  ## - with substitutions (note that the "best local matches" can introduce
  ##   indels and therefore be shorter than 6)
subject <- BString("AsCDEFxxxABDCEFxxxBACDEFxxxABCEDF")
matchPattern("ABCDEF", subject, max.mismatch=2, with.indels=TRUE)
matchPattern("ABCDEF", subject, max.mismatch=2)
```

## `matchProbePair` *Find "theoretical amplicons" mapped to a probe pair*

### Description

In the context of a computer-simulated PCR experiment, one wants to find the amplicons mapped to a given primer pair. The `matchProbePair` function can be used for this: given a forward and a reverse probe (i.e. the chromosome-specific sequences of the forward and reverse primers used for the experiment) and a target sequence (generally a chromosome sequence), the `matchProbePair` function will return all the "theoretical amplicons" mapped to this probe pair.

### Usage

```
matchProbePair(Fprobe, Rprobe, subject, algorithm="auto", logfile=NULL, verbos
```

### Arguments

| | |
|---|---|
| `Fprobe` | The forward probe. |
| `Rprobe` | The reverse probe. |
| `subject` | A `DNAString` object (or an `XStringViews` object with a `DNAString` subject) containing the target sequence. |
| `algorithm` | One of the following: `"auto"`, `"naive-exact"`, `"naive-inexact"`, `"boyer-moore"` or `"shift-or"`. See `matchPattern` for more information. |

| | |
|---|---|
| `logfile` | A file used for logging. |
| `verbose` | `TRUE` or `FALSE`. |

## Details

The `matchProbePair` function does the following: (1) find all the "plus hits" i.e. the Fprobe and Rprobe matches on the "plus" strand, (2) find all the "minus hits" i.e. the Fprobe and Rprobe matches on the "minus" strand and (3) from the set of all (plus_hit, minus_hit) pairs, extract and return the subset of "reduced matches" i.e. the (plus_hit, minus_hit) pairs such that (a) plus_hit <= minus_hit and (b) there are no hits (plus or minus) between plus_hit and minus_hit. This set of "reduced matches" is the set of "theoretical amplicons".

## Value

An XStringViews object containing the set of "theoretical amplicons".

## Author(s)

H. Pages

## See Also

`matchPattern`, `matchLRPatterns`, `findPalindromes`, `reverseComplement`, `XStringViews`

## Examples

```
library(BSgenome.Dmelanogaster.UCSC.dm3)
subject <- Dmelanogaster$chr3R

## With 20-nucleotide forward and reverse probes:
Fprobe <- "AGCTCCGAGTTCCTGCAATA"
Rprobe <- "CGTTGTTCACAAATATGCGG"
matchProbePair(Fprobe, Rprobe, subject) # 1 "theoretical amplicon"

## With shorter forward and reverse probes, the risk of having multiple
## "theoretical amplicons" increases:
Fprobe <- "AGCTCCGAGTTCC"
Rprobe <- "CGTTGTTCACAA"
matchProbePair(Fprobe, Rprobe, subject) # 2 "theoretical amplicons"
Fprobe <- "AGCTCCGAGTT"
Rprobe <- "CGTTGTTCACA"
matchProbePair(Fprobe, Rprobe, subject) # 9 "theoretical amplicons"
```

---

`matchWCP` *A simple WCP matching function and related utilities*

---

## Description

A function implementing a simple algorithm for matching a set of patterns represented by Weighted Clustered Positions (WCP) to an XString sequence.

## Usage

```
matchWCP(wcp, subject, min.score="80%")
countWCP(wcp, subject, min.score="80%")
WCPscoreStartingAt(wcp, subject, starting.at=1)
```

## Arguments

| | |
|---|---|
| `wcp` | A WCP object. |
| `subject` | An XString, XStringViews or MaskedXString object for `matchWCP` and `countWCP`. A XString object for `WCPscoreStartingAt`. |
| `min.score` | The minimum score for counting a match. Can be given as a character string containing a percentage (e.g. `"85%"`) of the highest possible score or as a single number. |
| `starting.at` | An integer vector specifying the starting positions of the Weighted Clustered Positions relatively to the subject. |

## Value

An XStringViews object for `matchWCP`.

A single integer for `countWCP`.

A numeric vector containing the Weighted Clustered Positions-based scores for `WCPscoreStartingAt`.

## Author(s)

P. Aboyoun

## See Also

`matchPWM`, `matchPattern`, WCP-class, XString-class, XStringViews-class

---

`matchprobes` *A function to match a query sequence to the sequences of a set of probes.*

---

## Description

The `query` sequence, a character string (probably representing a transcript of interest), is scanned for the presence of exact matches to the sequences in the character vector `records`. The indices of the set of matches are returned.

The function is inefficient: it works on R's character vectors, and the actual matching algorithm is of time complexity `length(query)` times `length(records)`!

See `matchPattern`, `vmatchPattern` and `matchPDict` for more efficient sequence matching functions.

## Usage

```
matchprobes(query, records, probepos=FALSE)
```

### Arguments

| | |
|---|---|
| `query` | A character vector. For example, each element may represent a gene (transcript) of interest. See Details. |
| `records` | A character vector. For example, each element may represent the probes on a DNA array. |
| `probepos` | A logical value. If TRUE, return also the start positions of the matches in the query sequence. |

### Details

`toupper` is applied to the arguments `query` and `records` before matching. The intention of this is to make the matching case-insensitive. The function is embarrassingly naive. The matching is done using the C library function `strstr`.

### Value

A list. Its first element is a list of the same length as the input vector. Each element of the list is a numeric vector containing the indices of the probes that have a perfect match in the query sequence.

If `probepos` is TRUE, the returned list has a second element: it is of the same shape as described above, and gives the respective positions of the matches.

### Author(s)

R. Gentleman, Laurent Gautier, Wolfgang Huber

### See Also

`matchPattern`, `vmatchPattern`, `matchPDict`

### Examples

```
if(require("hgu95av2probe")){
  data("hgu95av2probe")
  seq <- hgu95av2probe$sequence[1:20]
  target <- paste(seq, collapse="")
  matchprobes(target, seq, probepos=TRUE)
}
```

---

## `misc` *Some miscellaneous stuff*

---

### Description

Some miscellaneous stuff.

### Usage

```
N50(csizes)
```

### Arguments

| | |
|---|---|
| `csizes` | A vector containing the contig sizes. |

## Value

`N50`: The N50 value as an integer.

## The N50 contig size

**Definition** The N50 contig size of an assembly (aka the N50 value) is the size of the largest contig such that the contigs larger than that have at least 50% the bases of the assembly.

**How is it calculated?** It is calculated by adding the sizes of the biggest contigs until you reach half the total size of the contigs. The N50 value is then the size of the contig that was added last (i.e. the smallest of the big contigs covering 50% of the genome).

**What for?** The N50 value is a standard measure of the quality of a de novo assembly.

## Author(s)

Nicolas Delhomme <delhomme@embl.de>

## See Also

XStringSet-class

## Examples

```
## ---------------------------------------------------------------------
## A. The N50 contig size
## ---------------------------------------------------------------------

# generate 10 random contigs of sizes comprised between 100 and 10000
my.contig <- DNAStringSet(
               sapply(
                 sample(c(100:10000), 10),
                 function(size)
                     paste(sample(DNA_BASES, size, replace=TRUE), collapse="")
               )
             )

# get their sizes
my.size <- width(my.contig)

# calculate the N50 value of this set of contigs
my.contig.N50 <- N50(my.size)
```

---

`needwunsQS` *(Deprecated) Needleman-Wunsch Global Alignment*

---

## Description

Simple gap implementation of Needleman-Wunsch global alignment algorithm.

## Usage

```
needwunsQS(s1, s2, substmat, gappen = 8)
```

## Arguments

| | |
|---|---|
| `s1, s2` | an R character vector of length 1 or an XString object. |
| `substmat` | matrix of alignment score values. |
| `gappen` | penalty for introducing a gap in the alignment. |

## Details

Follows specification of Durbin, Eddy, Krogh, Mitchison (1998). This function has been deprecated and is being replaced by `pairwiseAlignment`.

## Value

An instance of class "`PairwiseAlignedXStringSet`".

## Author(s)

Vince Carey (`<stvjc@channing.harvard.edu>`) (original author) and H. Pages (current maintainer).

## References

R. Durbin, S. Eddy, A. Krogh, G. Mitchison, Biological Sequence Analysis, Cambridge UP 1998, sec 2.3.

## See Also

pairwiseAlignment, PairwiseAlignedXStringSet-class, substitution.matrices

## Examples

```
## Not run: 
  ## This function has been deprecated
  ## Use 'pairwiseAlignment' instead.

  ## nucleotide alignment
  mat <- matrix(-5L, nrow = 4, ncol = 4)
  for (i in seq_len(4)) mat[i, i] <- 0L
  rownames(mat) <- colnames(mat) <- DNA_ALPHABET[1:4]
  s1 <- DNAString(paste(sample(DNA_ALPHABET[1:4], 1000, replace=TRUE), collapse=""))
  s2 <- DNAString(paste(sample(DNA_ALPHABET[1:4], 1000, replace=TRUE), collapse=""))
  nw0 <- needwunsQS(s1, s2, mat, gappen = 0)
  nw1 <- needwunsQS(s1, s2, mat, gappen = 1)
  nw5 <- needwunsQS(s1, s2, mat, gappen = 5)

  ## amino acid alignment
  needwunsQS("PAWHEAE", "HEAGAWGHEE", substmat = "BLOSUM50")

## End(Not run)
```

`nucleotideFrequency` *Calculate the frequency of oligonucleotides in a DNA or RNA sequence (and other related functions)*

## Description

Given a DNA or RNA sequence (or a set of DNA or RNA sequences), the `oligonucleotideFrequency` function computes the frequency of all possible oligonucleotides of a given length (called the "width" in this particular context).

The `dinucleotideFrequency` and `trinucleotideFrequency` functions are convenient wrappers for calling `oligonucleotideFrequency` with `width=2` and `width=3`, respectively.

The `nucleotideFrequencyAt` function computes the frequency of the short sequences formed by extracting the nucleotides found at some fixed positions from each sequence of a set of DNA or RNA sequences.

In this man page we call "DNA input" (or "RNA input") an XString, XStringSet, XStringViews or MaskedXString object of base type DNA (or RNA).

## Usage

```
  oligonucleotideFrequency(x, width, as.prob=FALSE, freq=FALSE, as.array=FALSE,
                           fast.moving.side="right", with.labels=TRUE, ...)

  ## S4 method for signature 'XStringSet':
oligonucleotideFrequency(x,
      width, as.prob=FALSE, freq=FALSE, as.array=FALSE,
      fast.moving.side="right", with.labels=TRUE, simplify.as="matrix")

  dinucleotideFrequency(x, as.prob=FALSE, freq=FALSE, as.matrix=FALSE,
                        fast.moving.side="right", with.labels=TRUE, ...)
  trinucleotideFrequency(x, as.prob=FALSE, freq=FALSE, as.array=FALSE,
                         fast.moving.side="right", with.labels=TRUE, ...)

  nucleotideFrequencyAt(x, at, as.prob=FALSE, freq=FALSE, as.array=TRUE,
                        fast.moving.side="right", with.labels=TRUE, ...)

  ## Some related functions:
  oligonucleotideTransitions(x, left=1, right=1, as.prob=FALSE, freq=FALSE)
  mkAllStrings(alphabet, width, fast.moving.side="right")
```

## Arguments

| | |
|---|---|
| `x` | Any DNA or RNA input for the `*Frequency` and `oligonucleotideTransitions` functions.<br>An XStringSet or XStringViews object of base type DNA or RNA for `nucleotideFrequencyAt`. |
| `width` | The number of nucleotides per oligonucleotide for `oligonucleotideFrequency`.<br>The number of letters per string for `mkAllStrings`. |
| `at` | An integer vector containing the positions to look at in each element of `x`. |

| | |
|---|---|
| `as.prob` | If `TRUE` then probabilities are reported, otherwise counts (the default). |
| `freq` | This argument is deprecated. Please use the `as.prob` argument instead. |
| `as.array,as.matrix` | Controls the "shape" of the returned object. If `TRUE` (the default for `nucleotideFrequencyAt`) then it's a numeric matrix (or array), otherwise it's just a "flat" numeric vector i.e. a vector with no dim attribute (the default for the `*Frequency` functions). |
| `fast.moving.side` | Which side of the strings should move fastest? Note that, when `as.array` is TRUE, then the supplied value is ignored and the effective value is `"left"`. |
| `with.labels` | If `TRUE` then the returned object is named. |
| `...` | Further arguments to be passed to or from other methods. |
| `simplify.as` | Together with the `as.array` and `as.matrix` arguments, controls the "shape" of the returned object when the input `x` is an XStringSet or XStringViews object. Supported `simplify.as` values are `"matrix"` (the default), `"list"` and `"collapsed"`. If `simplify.as` is `"matrix"`, the returned object is a matrix with `length(x)` rows where the `i`-th row contains the frequencies for `x[[i]]`. If `simplify.as` is `"list"`, the returned object is a list of the same length as `length(x)` where the `i`-th element contains the frequencies for `x[[i]]`. If `simplify.as` is `"collapsed"`, then the the frequencies are computed for the entire object `x` as a whole (i.e. frequencies cumulated across all sequences in `x`). |
| `left, right` | The number of nucleotides per oligonucleotide for the rows and columns respectively in the transition matrix created by `oligonucleotideTransitions`. |
| `alphabet` | The alphabet to use to make the strings. |

## Value

If `x` is an XString or MaskedXString object, the `*Frequency` functions return a numeric vector of length `4^width`. If `as.array` (or `as.matrix`) is `TRUE`, then this vector is formatted as an array (or matrix). If `x` is an XStringSet or XStringViews object, the returned object has the shape specified by the `simplify.as` argument.

## Author(s)

H. Pages and P. Aboyoun

## See Also

`alphabetFrequency`, `alphabet`, `hasLetterAt`, XString-class, XStringSet-class, XStringViews-class, MaskedXString-class, `GENETIC_CODE`, `AMINO_ACID_CODE`, `reverse,XString-method`, `rev`

## Examples

```
## ---------------------------------------------------------------------
## A. BASIC *Frequency() EXAMPLES
## ---------------------------------------------------------------------
data(yeastSEQCHR1)
yeast1 <- DNAString(yeastSEQCHR1)

dinucleotideFrequency(yeast1)
trinucleotideFrequency(yeast1)
```

```
oligonucleotideFrequency(yeast1, 4)

## Get the less and most represented 6-mers:
f6 <- oligonucleotideFrequency(yeast1, 6)
f6[f6 == min(f6)]
f6[f6 == max(f6)]

## Get the result as an array:
tri <- trinucleotideFrequency(yeast1, as.array=TRUE)
tri["A", "A", "C"] # == trinucleotideFrequency(yeast1)["AAC"]
tri["T", , ] # frequencies of trinucleotides starting with a "T"

## With input made of multiple sequences:
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
dfmat <- dinucleotideFrequency(probes)  # a big matrix
dinucleotideFrequency(probes, simplify.as="collapsed")
dinucleotideFrequency(probes, simplify.as="collapsed", as.matrix=TRUE)

## ---------------------------------------------------------------------
## B. nucleotideFrequencyAt()
## ---------------------------------------------------------------------
nucleotideFrequencyAt(probes, 13)
nucleotideFrequencyAt(probes, c(13, 20))
nucleotideFrequencyAt(probes, c(13, 20), as.array=FALSE)

## nucleotideFrequencyAt() can be used to answer questions like: "how
## many probes in the drosophila2 chip have T, G, T, A at position
## 2, 4, 13 and 20, respectively?"
nucleotideFrequencyAt(probes, c(2, 4, 13, 20))["T", "G", "T", "A"]
## or "what's the probability to have an A at position 25 if there is
## one at position 13?"
nf <- nucleotideFrequencyAt(probes, c(13, 25))
sum(nf["A", "A"]) / sum(nf["A", ])
## Probabilities to have other bases at position 25 if there is an A
## at position 13:
sum(nf["A", "C"]) / sum(nf["A", ])  # C
sum(nf["A", "G"]) / sum(nf["A", ])  # G
sum(nf["A", "T"]) / sum(nf["A", ])  # T

## See ?hasLetterAt for another way to get those results.

## ---------------------------------------------------------------------
## C. oligonucleotideTransitions()
## ---------------------------------------------------------------------
## Get nucleotide transition matrices for yeast1
oligonucleotideTransitions(yeast1)
oligonucleotideTransitions(yeast1, 2, as.prob=TRUE)

## ---------------------------------------------------------------------
## D. ADVANCED *Frequency() EXAMPLES
## ---------------------------------------------------------------------
## Note that when dropping the dimensions of the 'tri' array, elements
## in the resulting vector are ordered as if they were obtained with
## 'fast.moving.side="left"':
triL <- trinucleotideFrequency(yeast1, fast.moving.side="left")
all(as.vector(tri) == triL) # TRUE
```

```
## Convert the trinucleotide frequency into the amino acid frequency
## based on translation:
tri1 <- trinucleotideFrequency(yeast1)
names(tri1) <- GENETIC_CODE[names(tri1)]
sapply(split(tri1, names(tri1)), sum) # 12512 occurrences of the stop codon

## When the returned vector is very long (e.g. width >= 10), using
## 'with.labels=FALSE' can improve performance significantly.
## Here for example, the observed speed up is between 25x and 500x:
f12 <- oligonucleotideFrequency(yeast1, 12, with.labels=FALSE) # very fast!

## Spome related functions:
dict1 <- mkAllStrings(LETTERS[1:3], 4)
dict2 <- mkAllStrings(LETTERS[1:3], 4, fast.moving.side="left")
identical(reverse(dict1), dict2) # TRUE
```

---

## `pairwiseAlignment` *Optimal Pairwise Alignment*

---

### Description

Solves (Needleman-Wunsch) global alignment, (Smith-Waterman) local alignment, and (ends-free) overlap alignment problems.

### Usage

```
pairwiseAlignment(pattern, subject, ...)
## S4 method for signature 'XStringSet,XStringSet':
pairwiseAlignment(pattern, subject,
                  patternQuality = PhredQuality(22L), subjectQuality = PhredQual
                  type = "global", substitutionMatrix = NULL, fuzzyMatrix = NULL
                  gapOpening = -10, gapExtension = -4, scoreOnly = FALSE)
## S4 method for signature 'QualityScaledXStringSet,QualityScaledXStringSet':
pairwiseAlignment(pattern, subject,
                  type = "global", substitutionMatrix = NULL, fuzzyMatrix = NULL
                  gapOpening = -10, gapExtension = -4, scoreOnly = FALSE)
```

### Arguments

| | |
|---|---|
| `pattern` | a character vector of any length, an `XString`, or an `XStringSet` object. |
| `subject` | a character vector of length 1 or an `XString` object. |
| `patternQuality, subjectQuality` | objects of class `XStringQuality` representing the respective quality scores for `pattern` and `subject` that are used in a quality-based method for generating a substitution matrix. These two arguments are ignored if `!is.null(substitutionMatr` or if its respective string set (`pattern`, `subject`) is of class `QualityScaledXStringSet`. |
| `type` | type of alignment. One of `"global"`, `"local"`, `"overlap"`, `"global-local"`, and `"local-global"` where `"global"` = align whole strings with end gap penalties, `"local"` = align string fragments, `"overlap"` = align whole strings without end gap penalties, `"global-local"` = align |

| | |
|---|---|
| | whole strings with end gap penalties on `pattern` and without end gap penalties on `subject` `"local-global"` = align whole strings without end gap penalties on `pattern` and with end gap penalties on `subject`. |
| `substitutionMatrix` | substitution matrix representing the fixed substitution scores for an alignment. It cannot be used in conjunction with `patternQuality` and `subjectQuality` arguments. |
| `fuzzyMatrix` | fuzzy match matrix for quality-based alignments. It takes values between 0 and 1; where 0 is an unambiguous mismatch, 1 is an unambiguous match, and values in between represent a fraction of "matchiness". (See details section below.) |
| `gapOpening` | the cost for opening a gap in the alignment. |
| `gapExtension` | the incremental cost incurred along the length of the gap in the alignment. |
| `scoreOnly` | logical to denote whether or not to return just the scores of the optimal pairwise alignment. |
| `...` | optional arguments to generic function to support additional methods. |

## Details

Quality-based alignments are based on the paper the Bioinformatics article by Ketil Malde listed in the Reference section below. Let $\epsilon_i$ be the probability of an error in the base read. For `"Phred"` quality measures $Q$ in $[0, 99]$, these error probabilities are given by $\epsilon_i = 10^{-Q/10}$. For `"Solexa"` quality measures $Q$ in $[-5, 99]$, they are given by $\epsilon_i = 1 - 1/(1 + 10^{-Q/10})$. Assuming independence within and between base reads, the combined error probability of a mismatch when the underlying bases do match is $\epsilon_c = \epsilon_1 + \epsilon_2 - (n/(n-1)) * \epsilon_1 * \epsilon_2$, where $n$ is the number of letters in the underlying alphabet. Using $\epsilon_c$, the substitution score is given by when two bases match is given by $b * \log_2(\gamma_{x,y} * (1 - \epsilon_c) * n + (1 - \gamma_{x,y}) * \epsilon_c * (n/(n-1)))$, where $b$ is the bit-scaling for the scoring and $\gamma_{x,y}$ is the probability that characters $x$ and $y$ represents the same underlying information (e.g. using IUPAC, $\gamma_{A,A} = 1$ and $\gamma_{A,N} = 1/4$. In the arguments listed above `fuzzyMatch` represents $\gamma_{x,y}$ and `patternQuality` and `subjectQuality` represents $\epsilon_1$ and $\epsilon_2$ respectively.

If `scoreOnly == FALSE`, a pairwise alignment with the maximum alignment score is returned. If more than one pairwise alignment produces the maximum alignment score, then the alignment with the smallest initial deletion whose mismatches occur before its insertions and deletions is chosen. For example, if `pattern = "AGTA"` and `subject = "AACTAACTA"`, then the alignment `pattern: [1] AG-TA; subject: [1] AACTA` is chosen over `pattern: [1] A-GTA; subject: [1] AACTA` or `pattern: [1] AG-TA; subject: [5] AACTA` if they all achieve the maximum alignment score.

## Value

If `scoreOnly == FALSE`, an instance of class `PairwiseAlignedXStringSet` or `PairwiseAlignedFixec` is returned. If `scoreOnly == TRUE`, a numeric vector containing the scores for the optimal pairwise alignments is returned.

## Note

Use `matchPattern` or `vmatchPattern` if you need to find all the occurrences (eventually with indels) of a given pattern in a reference sequence or set of sequences.

Use `matchPDict` if you need to match a (big) set of patterns against a reference sequence.

## Author(s)

P. Aboyoun and H. Pages

### References

R. Durbin, S. Eddy, A. Krogh, G. Mitchison, Biological Sequence Analysis, Cambridge UP 1998, sec 2.3.

B. Haubold, T. Wiehe, Introduction to Computational Biology, Birkhauser Verlag 2006, Chapter 2.

K. Malde, The effect of sequence quality on sequence alignment, Bioinformatics 2008 24(7):897-900.

### See Also

`stringDist`, PairwiseAlignedXStringSet-class, XStringQuality-class, substitution.matrices, `matchPattern`

### Examples

```
## Nucleotide global, local, and overlap alignments
s1 <-
  DNAString("ACTTCACCAGCTCCCTGGCGGTAAGTTGATCAAAGGAAACGCAAAGTTTTCAAG")
s2 <-
  DNAString("GTTTCACTACTTCCTTTCGGGTAAGTAAATATATAAATATATATAAAAATATAATTTTCATC")

# First use a fixed substitution matrix
mat <- nucleotideSubstitutionMatrix(match = 1, mismatch = -3, baseOnly = TRUE)
globalAlign <-
  pairwiseAlignment(s1, s2, substitutionMatrix = mat,
                    gapOpening = -5, gapExtension = -2)
localAlign <-
  pairwiseAlignment(s1, s2, type = "local", substitutionMatrix = mat,
                    gapOpening = -5, gapExtension = -2)
overlapAlign <-
  pairwiseAlignment(s1, s2, type = "overlap", substitutionMatrix = mat,
                    gapOpening = -5, gapExtension = -2)

# Then use quality-based method for generating a substitution matrix
pairwiseAlignment(s1, s2,
                  patternQuality = SolexaQuality(rep(c(22L, 12L), times = c(36, 18))),
                  subjectQuality = SolexaQuality(rep(c(22L, 12L), times = c(40, 20))),
                  scoreOnly = TRUE)

# Now assume can't distinguish between C/T and G/A
pairwiseAlignment(s1, s2,
                  patternQuality = SolexaQuality(rep(c(22L, 12L), times = c(36, 18))),
                  subjectQuality = SolexaQuality(rep(c(22L, 12L), times = c(40, 20))),
                  type = "local")
mapping <- diag(4)
dimnames(mapping) <- list(DNA_BASES, DNA_BASES)
mapping["C", "T"] <- mapping["T", "C"] <- 1
mapping["G", "A"] <- mapping["A", "G"] <- 1
pairwiseAlignment(s1, s2,
                  patternQuality = SolexaQuality(rep(c(22L, 12L), times = c(36, 18))),
                  subjectQuality = SolexaQuality(rep(c(22L, 12L), times = c(40, 20))),
                  fuzzyMatrix = mapping,
                  type = "local")

## Amino acid global alignment
pairwiseAlignment(AAString("PAWHEAE"), AAString("HEAGAWGHEE"),
                  substitutionMatrix = "BLOSUM50",
                         gapOpening = 0, gapExtension = -8)
```

---

`phiX174Phage` *Versions of bacteriophage phiX174 complete genome and sample short reads*

---

## Description

Six versions of the complete genome for bacteriophage $\phi$ X174 as well as a small number of Solexa short reads, qualities associated with those short reads, and counts for the number times those short reads occurred.

## Details

The `phiX174Phage` object is a `DNAStringSet` containing the following six naturally occurring versions of the bacteriophage $\phi$ X174 genome cited in Smith et al.:

**Genbank:** The version of the genome from GenBank (NC_001422.1, GI:9626372).

**RF70s:** A preparation of $\phi$ X double-stranded replicative form (RF) of DNA by Clyde A. Hutchison III from the late 1970s.

**SS78:** A preparation of $\phi$ X virion single-stranded DNA from 1978.

**Bull:** The sequence of wild-type $\phi$ X used by Bull et al.

**G'97:** The $\phi$ X replicative form (RF) of DNA from Bull et al.

**NEB'03:** A $\phi$ X replicative form (RF) of DNA from New England BioLabs (NEB).

The `srPhiX174` object is a `DNAStringSet` containing short reads from a Solexa machine.

The `quPhiX174` object is a `BStringSet` containing Solexa quality scores associated with `srPhiX174`.

The `wtPhiX174` object is an integer vector containing counts associated with `srPhiX174`.

## References

http://www.genome.jp/dbget-bin/www_bget?refseq+NC_001422

Bull, J. J., Badgett, M. R., Wichman, H. A., Huelsenbeck, Hillis, D. M., Gulati, A., Ho, C. & Molineux, J. (1997) Genetics 147, 1497-1507.

Smith, Hamilton O.; Clyde A. Hutchison, Cynthia Pfannkoch, J. Craig Venter (2003-12-23). "Generating a synthetic genome by whole genome assembly: {phi}X174 bacteriophage from synthetic oligonucleotides". Proceedings of the National Academy of Sciences 100 (26): 15440-15445. doi:10.1073/pnas.2237126100.

## Examples

```
data(phiX174Phage)
nchar(phiX174Phage)
genBankPhage <- phiX174Phage[[1]]
genBankSubstring <- substring(genBankPhage, 2793-34, 2811+34)

data(srPhiX174)
srPhiX174
quPhiX174
summary(wtPhiX174)
```

```
alignPhiX174 <-
  pairwiseAlignment(srPhiX174, genBankSubstring,
                    patternQuality = SolexaQuality(quPhiX174),
                    subjectQuality = SolexaQuality(99L),
                    type = "global-local")
summary(alignPhiX174, weight = wtPhiX174)
```

---

## pid *Percent Sequence Identity*

---

### Description

Calculates the percent sequence identity for a pairwise sequence alignment.

### Usage

```
pid(x, type="PID1")
```

### Arguments

| | |
|---|---|
| `x` | a `PairwiseAlignedXStringSet` object. |
| `type` | one of percent sequence identity. One of `"PID1"`, `"PID2"`, `"PID3"`, and `"PID4"`. See Details for more information. |

### Details

Since there is no universal definition of percent sequence identity, the `pid` function calculates this statistic in the following types:

`"PID1"`: 100 * (identical positions) / (aligned positions + internal gap positions)

`"PID2"`: 100 * (identical positions) / (aligned positions)

`"PID3"`: 100 * (identical positions) / (length shorter sequence)

`"PID4"`: 100 * (identical positions) / (average length of the two sequences)

### Value

A numeric vector containing the specified sequence identity measures.

### Author(s)

P. Aboyoun

### References

A. May, Percent Sequence Identity: The Need to Be Explicit, Structure 2004, 12(5):737.

G. Raghava and G. Barton, Quantification of the variation in percentage identity for protein sequence alignments, BMC Bioinformatics 2006, 7:415.

### See Also

pairwiseAlignment, PairwiseAlignedXStringSet-class, match-utils

## Examples

```
s1 <- DNAString("AGTATAGATGATAGAT")
s2 <- DNAString("AGTAGATAGATGGATGATAGATA")

palign1 <- pairwiseAlignment(s1, s2)
palign1
pid(palign1)

palign2 <-
  pairwiseAlignment(s1, s2,
    substitutionMatrix =
    nucleotideSubstitutionMatrix(match = 2, mismatch = 10, baseOnly = TRUE))
palign2
pid(palign2, type = "PID4")
```

---

`pmatchPattern` *Longest Common Prefix/Suffix/Substring searching functions*

---

## Description

Functions for searching the Longest Common Prefix/Suffix/Substring of two strings.

WARNING: These functions are experimental and might not work properly! Full documentation will come later.

Please send questions/comments to hpages@fhcrc.org

Thanks for your comprehension!

## Usage

```
lcprefix(s1, s2)
lcsuffix(s1, s2)
lcsubstr(s1, s2)
pmatchPattern(pattern, subject, maxlength.out=1L)
```

## Arguments

| | |
|---|---|
| `s1` | 1st string, a character string or an XString object. |
| `s2` | 2nd string, a character string or an XString object. |
| `pattern` | The pattern string. |
| `subject` | An XString object containing the subject string. |
| `maxlength.out` | The maximum length of the output i.e. the maximum number of views in the returned object. |

## See Also

matchPattern, XStringViews-class, XString-class `readFASTA` *Functions to read/write FASTA formatted files*

## Description

`readFASTA` and `writeFASTA` read from and write to a FASTA file. Note that the object returned by `readFASTA` or passed to `writeFASTA` is a standard list. For faster and more memory efficient alternatives that return/accept an XStringSet object, see the `read.DNAStringSet` function and family.

## Usage

```
readFASTA(file, checkComments=TRUE, strip.descs=TRUE)
writeFASTA(x, file="", desc=NULL, append=FALSE, width=80)
```

## Arguments

| | |
|---|---|
| `file` | Either a character string naming a file or a connection. If `""` (the default for `writeFASTA`), then the function writes to the standard output connection (the console) unless redirected by `sink`. |
| `checkComments` | Whether or not comments, lines beginning with a semi-colon should be found and removed. |
| `strip.descs` | Whether or not the ">" marking the beginning of the description lines should be removed. Note that this argument is new in Biostrings >= 2.8. In previous versions `readFASTA` was keeping the ">". |
| `x` | A list as one returned by `readFASTA` if `desc` is not specified (i.e. `NULL`). If `desc` is specified (see below) then `x` can also be a list-like object with XString elements (for example it can be an XStringSet, XStringViews or BSgenome object) or just a character vector. |
| `desc` | If `NULL` (the default) then `x` must be a list as one returned by `readFASTA` and all the sequences in `x` are written to the file. Otherwise `desc` must be a character vector no longer than the number of sequences in `x` containing the descriptions of the sequences in `x` that must be written to the file. |
| `append` | `TRUE` or `FALSE`. If `TRUE` output will be appended to `file`; otherwise, it will overwrite the contents of `file`. See `?cat` for the details. |
| `width` | The maximum number of letters per line of sequence. |

## Details

FASTA is a simple file format for biological sequence data. A file may contain one or more sequences, for each sequence there is a description line which begins with a >.

FASTA is a widely used format in biology. It is a relatively simple markup. I am not aware of a standard. It might be nice to check to see if the data that were parsed are sequences of some appropriate type, but without a standard that does not seem possible.

There are many other packages that provide similar, but different capabilities. The one in the package seqinr seems most similar but they separate the biological sequence into single character strings, which is too inefficient for large problems.

## Value

For `readFASTA`: A list with one element per FASTA record in the file. Each element is in two parts, one is the description of the record and the second a character string of the biological sequence.

## Author(s)

R. Gentleman, H. Pages. Improvements to writeFASTA by Kasper D. Hansen

## See Also

`read.DNAStringSet`, `fasta.info`, `write.XStringSet`, `read.table`, `scan`, `write.table`, `BSgenome-class`

## Examples

```
f1 <- system.file("extdata", "someORF.fa", package="Biostrings")
ff <- readFASTA(f1, strip.descs=TRUE)
desc <- sapply(ff, function(x) x$desc)
desc

## Keep the "reverse complement" sequences only:
ff2 <- ff[grep("reverse complement", desc, fixed=TRUE)]

## Write them to a FASTA file:
temp_file <- file.path(tempdir(), "temp.fa")
writeFASTA(ff2, file=temp_file)

## Write the first 2 to a FASTA file with a modified description:
writeFASTA(ff2, file=temp_file, desc=c("a", "b"))

## Write a genome to a FASTA file:
library(BSgenome.Celegans.UCSC.ce2)
writeFASTA(Celegans, file=temp_file, desc=seqnames(Celegans))
```

---

# `replaceLetterAt` *Replacing letters in a sequence (or set of sequences) at some specified locations*

---

## Description

`replaceLetterAt` first makes a copy of a sequence (or set of sequences) and then replaces some of the original letters by new letters at the specified locations.

`.inplaceReplaceLetterAt` is the IN PLACE version of `replaceLetterAt`: it will modify the original sequence in place i.e. without copying it first. Note that in place modification of a sequence is fundamentally dangerous because it alters all objects defined in your session that make reference to the modified sequence. NEVER use `.inplaceReplaceLetterAt`, unless you know what you are doing!

## Usage

```
replaceLetterAt(x, at, letter, if.not.extending="replace", verbose=FALSE)

## NEVER USE THIS FUNCTION!
.inplaceReplaceLetterAt(x, at, letter)
```

## Arguments

| | |
|---|---|
| `x` | A DNAString or rectangular DNAStringSet object. |
| `at` | The locations where the replacements must occur.<br>If `x` is a DNAString object, then `at` is typically an integer vector with no NAs but a logical vector or Rle object is valid too. Locations can be repeated and in this case the last replacement to occur at a given location prevails.<br>If `x` is a rectangular DNAStringSet object, then `at` must be a matrix of logicals with the same dimensions as `x`. |
| `letter` | The new letters.<br>If `x` is a DNAString object, then `letter` must be a DNAString object or a character vector (with no NAs) with a total number of letters (`sum(nchar(letter))`) equal to the number of locations specified in `at`.<br>If `x` is a rectangular DNAStringSet object, then `letter` must be a DNAStringSet object or a character vector of the same length as `x`. In addition, the number of letters in each element of `letter` must match the number of locations specified in the corresponding row of `at` (`all(width(letter) == rowSums(at))`). |
| `if.not.extending` | What to do if the new letter is not "extending" the old letter? The new letter "extends" the old letter if both are IUPAC letters and the new letter is as specific or less specific than the old one (e.g. M extends A, Y extends Y, but Y doesn't extend S). Possible values are `"replace"` (the default) for replacing in all cases, `"skip"` for not replacing when the new letter does not extend the old letter, `"merge"` for merging the new IUPAC letter with the old one, and `"error"` for raising an error.<br>Note that the gap (`"-"`) and hard masking (`"+"`) letters are not extending or extended by any other letter.<br>Also note that `"merge"` is the only value for the `if.not.extending` argument that guarantees the final result to be independent on the order the replacement is performed (although this is only relevant when `at` contains duplicated locations, otherwise the result is of course always independent on the order, whatever the value of `if.not.extending` is). |
| `verbose` | When `TRUE`, a warning will report the number of skipped or merged letters. |

## Details

`.inplaceReplaceLetterAt` semantic is equivalent to calling `replaceLetterAt` with `if.not.extending=` and `verbose=FALSE`.

Never use `.inplaceReplaceLetterAt`! It is used by the `injectSNPs` function in the BSgenome package, as part of the "lazy sequence loading" mechanism, for altering the original sequences of a BSgenome object at "sequence-load time". This alteration consists in injecting the IUPAC ambiguity letters representing the SNPs into the just loaded sequence, which is the only time where in place modification of the external data of an XString object is safe.

## Value

A DNAString or DNAStringSet object of the same shape (i.e. length and width) as the orignal object `x` for `replaceLetterAt`.

## Author(s)

H. Pages

## See Also

`IUPAC_CODE_MAP`, `chartr`, `injectHardMask`, DNAString, DNAStringSet, `injectSNPs`, BSgenome

## Examples

```
## Replace letters of a DNAString object:
replaceLetterAt(DNAString("AAMAA"), c(5, 1, 3, 1), "TYNC")
replaceLetterAt(DNAString("AAMAA"), c(5, 1, 3, 1), "TYNC", if.not.extending="merge")

## Replace letters of a DNAStringSet object (sorry for the totally
## artificial example with absolutely no biological meaning):
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
at <- matrix(c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE),
             nrow=length(probes), ncol=width(probes)[1],
             byrow=TRUE)
letter_subject <- DNAString(paste(rep.int("-", width(probes)[1]), collapse=""))
letter <- as(Views(letter_subject, start=1, end=rowSums(at)), "XStringSet")
replaceLetterAt(probes, at, letter)
```

---

`reverseComplement` *Sequence reversing and complementing*

---

## Description

Use these functions for reversing sequences and/or complementing DNA or RNA sequences.

## Usage

```
  ## S4 method for signature 'character':
reverse(x, ...)
  ## S4 method for signature 'XString':
reverse(x, ...)
  complement(x, ...)
  reverseComplement(x, ...)
```

## Arguments

| | |
|---|---|
| `x` | A character vector, or an XString, XStringSet, XStringViews or MaskedXString object for `reverse`.<br>A DNAString, RNAString, DNAStringSet, RNAStringSet, XStringViews (with DNAString or RNAString subject), MaskedDNAString or MaskedRNAString object for `complement` and `reverseComplement`. |
| `...` | Additional arguments to be passed to or from methods. |

## Details

Given an XString object x, reverse(x) returns an object of the same XString base type as x where letters in x have been reordered in the reverse order.

If x is a DNAString or RNAString object, complement(x) returns an object where each base in x is "complemented" i.e. A, C, G, T in a DNAString object are replaced by T, G, C, A respectively and A, C, G, U in a RNAString object are replaced by U, G, C, A respectively.

Letters belonging to the IUPAC Extended Genetic Alphabet are also replaced by their complement (M <-> K, R <-> Y, S <-> S, V <-> B, W <-> W, H <-> D, N <-> N) and the gap ("-") and hard masking ("+") letters are unchanged.

reverseComplement(x) is equivalent to reverse(complement(x)) but is faster and more memory efficient.

## Value

An object of the same class and length as the original object.

## See Also

DNAString-class, RNAString-class, DNAStringSet-class, RNAStringSet-class, XStringViews-class, MaskedXString-class, chartr, findPalindromes, IUPAC_CODE_MAP

## Examples

```
## ---------------------------------------------------------------------
## A. SOME SIMPLE EXAMPLES
## ---------------------------------------------------------------------

x <- DNAString("ACGT-YN-")
reverseComplement(x)

library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
probes
alphabetFrequency(probes, collapse=TRUE)
rcprobes <- reverseComplement(probes)
rcprobes
alphabetFrequency(rcprobes, collapse=TRUE)

## ---------------------------------------------------------------------
## B. OBTAINING THE MISMATCH PROBES OF A CHIP
## ---------------------------------------------------------------------

pm2mm <- function(probes)
{
    probes <- DNAStringSet(probes)
    subseq(probes, start=13, end=13) <- complement(subseq(probes, start=13, end=13))
    probes
}
mmprobes <- pm2mm(probes)
mmprobes
alphabetFrequency(mmprobes, collapse=TRUE)

## ---------------------------------------------------------------------
## C. SEARCHING THE MINUS STRAND OF A CHROMOSOME
## ---------------------------------------------------------------------
## Applying reverseComplement() to the pattern before calling
## matchPattern() is the recommended way of searching hits on the
## minus strand of a chromosome.

library(BSgenome.Dmelanogaster.UCSC.dm3)
chrX <- Dmelanogaster$chrX
pattern <- DNAString("ACCAACNNGGTTG")
matchPattern(pattern, chrX, fixed=FALSE)  # 3 hits on strand +
rcpattern <- reverseComplement(pattern)
rcpattern
m0 <- matchPattern(rcpattern, chrX, fixed=FALSE)
m0  # 5 hits on strand -

## Applying reverseComplement() to the subject instead of the pattern is not
## a good idea for 2 reasons:
## (1) Chromosome sequences are generally big and sometimes very big
##     so computing the reverse complement of the positive strand will
##     take time and memory proportional to its length.
chrXminus <- reverseComplement(chrX)  # needs to allocate 22M of memory!
chrXminus
## (2) Chromosome locations are generally given relatively to the positive
##     strand, even for features located in the negative strand, so after
##     doing this:
m1 <- matchPattern(pattern, chrXminus, fixed=FALSE)
##     the start/end of the matches are now relative to the negative strand.
##     You need to apply reverseComplement() again on the result if you want
##     them to be relative to the positive strand:
m2 <- reverseComplement(m1)  # allocates 22M of memory, again!
##     and finally to apply rev() to sort the matches from left to right
##     (5'3' direction) like in m0:
m3 <- rev(m2) # same as m0, finally!

## WARNING: Before you try the example below on human chromosome 1, be aware
## that it will require the allocation of about 500Mb of memory!
if (interactive()) {
  library(BSgenome.Hsapiens.UCSC.hg18)
  chr1 <- Hsapiens$chr1
  matchPattern(pattern, reverseComplement(chr1))  # DON'T DO THIS!
  matchPattern(reverseComplement(pattern), chr1)  # DO THIS INSTEAD
}
```

---

`reverseSeq` *Reverse Sequence*

---

## Description

WARNING: The functions described in this man page have been deprecated in favor of `reverse,XString-method` and `reverseComplement`.

Functions to obtain the reverse and reverse complement of a sequence

## Usage

```
reverseSeq(seq)
```

```
revcompDNA(seq)
revcompRNA(seq)
```

## Arguments

| | |
|---|---|
| `seq` | Character vector. For `revcompRNA` and `revcompDNA` the sequence should consist of appropriate letter codes: `[ACGUN]` and `ACGTN`, respectively. |

## Details

The function reverses the order of the constituent character strings of its argument.

## Value

A character vector of the same length as `seq`.

## Author(s)

R. Gentleman, W. Huber, S. Falcon

## See Also

`alphabetFrequency`, `reverseComplement`

## Examples

```
w <-  c("hey there", "you silly fool")
if (interactive()) {
  reverseSeq(w)  # deprecated (inefficient on large vectors)
}
reverse(BStringSet(w))  # more efficient

w <- "able was I ere I saw Elba"
if (interactive()) {
  reverseSeq(w)  # deprecated (inefficient on large vectors)
}
reverse(BStringSet(w))  # more efficient

rna1 <- "UGCA"
if (interactive()) {
  revcompRNA(rna1)  # deprecated (inefficient on large vectors)
}
reverseComplement(RNAString(rna1))  # more efficient

dna1 <- "TGCA"
if (interactive()) {
  revcompDNA(dna1)  # deprecated (inefficient on large vectors)
}
reverseComplement(DNAString(dna1))  # more efficient

## Comparing efficiencies:
if (interactive()) {
  library(hgu95av2probe)
  system.time(y1 <- reverseSeq(hgu95av2probe$sequence))
  x <- DNAStringSet(hgu95av2probe)
  system.time(y2 <- reverse(x))
```

```
    system.time(y3 <- revcompDNA(hgu95av2probe$sequence))
    system.time(y4 <- reverseComplement(x))
  }
```

---

stringDist *String Distance/Alignment Score Matrix*

---

## Description

Computes the Levenshtein edit distance or pairwise alignment score matrix for a set of strings.

## Usage

```
stringDist(x, method = "levenshtein", ignoreCase = FALSE, diag = FALSE, upper =
## S4 method for signature 'XStringSet':
stringDist(x, method = "levenshtein", ignoreCase = FALSE, diag = FALSE,
                    upper = FALSE, type = "global", quality = PhredQuality(22L),
                    substitutionMatrix = NULL, fuzzyMatrix = NULL, gapOpening = 0
                    gapExtension = -1)
## S4 method for signature 'QualityScaledXStringSet':
stringDist(x, method = "quality", ignoreCase = FALSE,
                    diag = FALSE, upper = FALSE, type = "global", substitutionMat
                    fuzzyMatrix = NULL, gapOpening = 0, gapExtension = -1)
```

## Arguments

| | |
|---|---|
| x | a character vector or an XStringSet object. |
| method | calculation method. One of "levenshtein", "hamming", "quality", or "substitutionMatrix". |
| ignoreCase | logical value indicating whether to ignore case during scoring. |
| diag | logical value indicating whether the diagonal of the matrix should be printed by print.dist. |
| upper | logical value indicating whether the upper triangle of the matrix should be printed by print.dist. |
| type | (applicable when method = "quality" or method = "substitutionMatrix"). type of alignment. One of "global", "local", and "overlap", where "global" = align whole strings with end gap penalties, "local" = align string fragments, "overlap" = align whole strings without end gap penalties. |
| quality | (applicable when method = "quality"). object of class XStringQuality representing the quality scores for x that are used in a quality-based method for generating a substitution matrix. |
| substitutionMatrix | (applicable when method = "substitutionMatrix"). symmetric matrix representing the fixed substitution scores in the alignment. |
| fuzzyMatrix | (applicable when method = "quality"). fuzzy match matrix for quality-based alignments. It takes values between 0 and 1; where 0 is an unambiguous mismatch, 1 is an unambiguous match, and values in between represent a fraction of "matchiness". |

| | |
|---|---|
| `gapOpening` | (applicable when `method = "quality"` or `method = "substitutionMatrix"`). penalty for opening a gap in the alignment. |
| `gapExtension` | (applicable when `method = "quality"` or `method = "substitutionMatrix"`). penalty for extending a gap in the alignment |
| `...` | optional arguments to generic function to support additional methods. |

## Details

When `method = "hamming"`, uses the underlying `neditStartingAt` code to calculate the distances, where the Hamming distance is defined as the number of substitutions between two strings of equal length. Otherwise, uses the underlying `pairwiseAlignment` code to compute the distance/alignment score matrix.

## Value

Returns an object of class `"dist"`.

## Author(s)

P. Aboyoun

## See Also

dist, agrep, pairwiseAlignment, substitution.matrices

## Examples

```
  stringDist(c("lazy", "HaZy", "crAzY"))
  stringDist(c("lazy", "HaZy", "crAzY"), ignoreCase = TRUE)

  data(phiX174Phage)
  plot(hclust(stringDist(phiX174Phage), method = "single"))

  data(srPhiX174)
  stringDist(srPhiX174[1:4])
  stringDist(srPhiX174[1:4], method = "quality",
             quality = SolexaQuality(quPhiX174[1:4]),
             gapOpening = -10, gapExtension = -4)
```

---

## `subXString` *Fast substring extraction*

---

## Description

Functions for fast substring extraction.

## Usage

```
  subXString(x, start=NA, end=NA, length=NA)
  ## S4 method for signature 'XString':
substr(x, start=NA, stop=NA)
  ## S4 method for signature 'XString':
substring(text, first=NA, last=NA)
```

### Arguments

| | |
|---|---|
| `x` | An XString object for `subXString`. A character vector, an XStringViews, XString, or MaskedXString object for `substr` or `substring`. |
| `start` | A numeric vector. |
| `end` | A numeric vector. |
| `length` | A numeric vector. |
| `stop` | A numeric vector. |
| `text` | A character vector, an XStringViews or an XString object. |
| `first` | A numeric vector. |
| `last` | A numeric vector. |

### Details

`subXString` is deprecated in favor of `subseq`.

### Value

An XString object of the same base type as `x` for `subXString`.

A character vector for `substr` and `substring`.

### See Also

`subseq`, `letter`, XString-class, XStringViews-class

---

## `substitution.matrices` *Scoring matrices*

---

### Description

Predefined substitution matrices for nucleotide and amino acid alignments.

### Usage

```
data(BLOSUM45)
data(BLOSUM50)
data(BLOSUM62)
data(BLOSUM80)
data(BLOSUM100)
data(PAM30)
data(PAM40)
data(PAM70)
data(PAM120)
data(PAM250)
nucleotideSubstitutionMatrix(match = 1, mismatch = 0, baseOnly = FALSE, type =
qualitySubstitutionMatrices(fuzzyMatch = c(0, 1), alphabetLength = 4L, quality
errorSubstitutionMatrices(errorProbability, fuzzyMatch = c(0, 1), alphabetLeng
```

## Arguments

| | |
|---|---|
| `match` | the scoring for a nucleotide match. |
| `mismatch` | the scoring for a nucleotide mismatch. |
| `baseOnly` | `TRUE` or `FALSE`. If `TRUE`, only uses the letters in the "base" alphabet i.e. "A", "C", "G", "T". |
| `type` | either "DNA" or "RNA". |
| `fuzzyMatch` | a named or unnamed numeric vector representing the base match probability. |
| `errorProbability` | a named or unnamed numeric vector representing the error probability. |
| `alphabetLength` | an integer representing the number of letters in the underlying string alphabet. For DNA and RNA, this would be 4L. For Amino Acids, this could be 20L. |
| `qualityClass` | a character string of either `"PhredQuality"` or `"SolexaQuality"`. |
| `bitScale` | a numeric value to scale the quality-based substitution matrices. By default, this is 1, representing bit-scale scoring. |

## Format

The BLOSUM and PAM matrices are square symmetric matrices with integer coefficients, whose row and column names are identical and unique: each name is a single letter representing a nucleotide or an amino acid.

`nucleotideSubstitutionMatrix` produces a substitution matrix for all IUPAC nucleic acid codes based upon match and mismatch parameters.

`errorSubstitutionMatrices` produces a two element list of numeric square symmetric matrices, one for matches and one for mismatches.

`qualitySubstitutionMatrices` produces the substitution matrices for Phred or Solexa quality-based reads.

## Details

The BLOSUM and PAM matrices are not unique. For example, the definition of the widely used BLOSUM62 matrix varies depending on the source, and even a given source can provide different versions of "BLOSUM62" without keeping track of the changes over time. NCBI provides many matrices here ftp://ftp.ncbi.nih.gov/blast/matrices/ but their definitions don't match those of the matrices bundled with their stand-alone BLAST software available here ftp://ftp.ncbi.nih.gov/blast/

The BLOSUM45, BLOSUM62, BLOSUM80, PAM30 and PAM70 matrices were taken from NCBI stand-alone BLAST software.

The BLOSUM50, BLOSUM100, PAM40, PAM120 and PAM250 matrices were taken from ftp://ftp.ncbi.nih.gov/blast/m

The quality matrices computed in `qualitySubstitutionMatrices` are based on the paper by Ketil Malde. Let $\epsilon_i$ be the probability of an error in the base read. For `"Phred"` quality measures $Q$ in $[0, 99]$, these error probabilities are given by $\epsilon_i = 10^{-Q/10}$. For `"Solexa"` quality measures $Q$ in $[-5, 99]$, they are given by $\epsilon_i = 1 - 1/(1 + 10^{-Q/10})$. Assuming independence within and between base reads, the combined error probability of a mismatch when the underlying bases do match is $\epsilon_c = \epsilon_1 + \epsilon_2 - (n/(n-1)) * \epsilon_1 * \epsilon_2$, where $n$ is the number of letters in the underlying alphabet. Using $\epsilon_c$, the substitution score is given by when two bases match is given by $b * \log_2(\gamma_{x,y} * (1 - \epsilon_c) * n + (1 - \gamma_{x,y}) * \epsilon_c * (n/(n-1)))$, where $b$ is the bit-scaling for the scoring and $\gamma_{x,y}$ is the probability that characters $x$ and $y$ represents the same underlying information (e.g. using IUPAC, $\gamma_{A,A} = 1$ and $\gamma_{A,N} = 1/4$. In the arguments listed above `fuzzyMatch` represents $\gamma_{x,y}$ and `errorProbability` represents $\epsilon_i$.

## Author(s)

H. Pages and P. Aboyoun

## References

K. Malde, The effect of sequence quality on sequence alignment, Bioinformatics, Feb 23, 2008.

## See Also

`pairwiseAlignment`, PairwiseAlignedXStringSet-class, DNAString-class, AAString-class, PhredQuality-class, SolexaQuality-class

## Examples

```
s1 <-
  DNAString("ACTTCACCAGCTCCCTGGCGGTAAGTTGATCAAAGGAAACGCAAAGTTTTCAAG")
s2 <-
  DNAString("GTTTCACTACTTCCTTTCGGGTAAGTAAATATATAAATATATAAAAATATAATTTTCATC")

## Fit a global pairwise alignment using edit distance scoring
pairwiseAlignment(s1, s2,
                  substitutionMatrix = nucleotideSubstitutionMatrix(0, -1, TRUE),
                  gapOpening = 0, gapExtension = -1)

## Examine quality-based match and mismatch bit scores for DNA/RNA
## strings in pairwiseAlignment.
## By default patternQuality and subjectQuality are PhredQuality(22L).
qualityMatrices <- qualitySubstitutionMatrices()
qualityMatrices["22", "22", "1"]
qualityMatrices["22", "22", "0"]

pairwiseAlignment(s1, s2)

## Get the substitution scores when the error probability is 0.1
subscores <- errorSubstitutionMatrices(errorProbability = 0.1)
submat <- matrix(subscores[,,"0"], 4, 4)
diag(submat) <- subscores[,,"1"]
dimnames(submat) <- list(DNA_ALPHABET[1:4], DNA_ALPHABET[1:4])
submat
pairwiseAlignment(s1, s2, substitutionMatrix = submat)

## Align two amino acid sequences with the BLOSUM62 matrix
aa1 <- AAString("HXBLVYMGCHFDCXVBEHIKQZ")
aa2 <- AAString("QRNYMYCFQCISGNEYKQN")
pairwiseAlignment(aa1, aa2, substitutionMatrix = "BLOSUM62", gapOpening = -3, gapExtens

## See how the gap penalty influences the alignment
pairwiseAlignment(aa1, aa2, substitutionMatrix = "BLOSUM62", gapOpening = -6, gapExtens

## See how the substitution matrix influences the alignment
pairwiseAlignment(aa1, aa2, substitutionMatrix = "BLOSUM50", gapOpening = -3, gapExtens

if (interactive()) {
  ## Compare our BLOSUM62 with BLOSUM62 from ftp://ftp.ncbi.nih.gov/blast/matrices/
  data(BLOSUM62)
  BLOSUM62["Q", "Z"]
```

```
    file <- "ftp://ftp.ncbi.nih.gov/blast/matrices/BLOSUM62"
    b62 <- as.matrix(read.table(file, check.names=FALSE))
    b62["Q", "Z"]
  }
```

## toComplex — *Turning a DNA sequence into a vector of complex numbers*

### Description

The `toComplex` utility function turns a DNAString object into a complex vector.

### Usage

```
toComplex(x, baseValues)
```

### Arguments

| | |
|---|---|
| `x` | A DNAString object. |
| `baseValues` | A named complex vector containing the values associated to each base e.g. `c(A=1+0i, G=0+1i, T=-1+0i, C=0-1i)` |

### Value

A complex vector of the same length as `x`.

### Author(s)

H. Pages

### See Also

DNAString

### Examples

```
seq <- DNAString("accacctgaccattgtcct")
baseValues1 <- c(A=1+0i, G=0+1i, T=-1+0i, C=0-1i)
toComplex(seq, baseValues1)

## GC content:
baseValues2 <- c(A=0, C=1, G=1, T=0)
sum(as.integer(toComplex(seq, baseValues2)))
## Note that there are better ways to do this (see ?alphabetFrequency)
```

## `translate` *DNA/RNA transcription and translation*

### Description

Functions for transcription and/or translation of DNA or RNA sequences, and related utilities.

### Usage

```
## Transcription:
transcribe(x)
cDNA(x)

## Translation:
codons(x)
translate(x)

## Related utilities:
dna2rna(x)
rna2dna(x)
```

### Arguments

| | |
|---|---|
| `x` | A DNAString object for `transcribe` and `dna2rna`.<br>An RNAString object for `cDNA` and `rna2dna`.<br>A DNAString, RNAString, MaskedDNAString or MaskedRNAString object for `codons`.<br>A DNAString, RNAString, DNAStringSet, RNAStringSet, MaskedDNAString or MaskedRNAString object for `translate`. |

### Details

`transcribe` reproduces the biological process of DNA transcription that occurs in the cell. It takes the naive approach to treat the whole sequence `x` as if it was a single exon. See `extractTranscripts` for a more powerful version that allows the user to extract a set of transcripts specified by the starts and ends of their exons as well as the strand from which the transcript is coming.

`cDNA` reproduces the process of synthesizing complementary DNA from a mature mRNA template.

`translate` reproduces the biological process of RNA translation that occurs in the cell. The input of the function can be either RNA or coding DNA. The Standard Genetic Code (see `?GENETIC_CODE`) is used to translate codons into amino acids. `codons` is a utility for extracting the codons involved in this translation without translating them.

`dna2rna` and `rna2dna` are low-level utilities for converting sequences from DNA to RNA and vice-versa. All what this converstion does is to replace each occurrence of T by a U and vice-versa.

### Value

An RNAString object for `transcribe` and `dna2rna`.

A DNAString object for `cDNA` and `rna2dna`.

Note that if the sequence passed to `transcribe` or `cDNA` is considered to be oriented 5'-3', then the returned sequence is oriented 3'-5'.

An XStringViews object with 1 view per codon for `codons`. When `x` is a MaskedDNAString or MaskedRNAString object, its masked parts are interpreted as introns and filled with the + letter in the returned object. Therefore codons that span across masked regions are represented by views that have a width > 3 and contain the + letter. Note that each view is guaranteed to contain exactly 3 base letters.

An AAString object for `translate`.

## See Also

`reverseComplement`, `GENETIC_CODE`, DNAString-class, RNAString-class, AAString-class, XStringSet-class, XStringViews-class, MaskedXString-class

## Examples

```
file <- system.file("extdata", "someORF.fa", package="Biostrings")
x <- read.DNAStringSet(file)
x

## The first and last 1000 nucleotides are not part of the ORFs:
x <- DNAStringSet(x, start=1001, end=-1001)

## Before calling translate() on an ORF, we need to mask the introns
## if any. We can get this information fron the SGD database
## (http://www.yeastgenome.org/).
## According to SGD, the 1st ORF (YAL001C) has an intron at 71..160
## (see http://db.yeastgenome.org/cgi-bin/locus.pl?locus=YAL001C)
y1 <- x[[1]]
mask1 <- Mask(length(y1), start=71, end=160)
masks(y1) <- mask1
y1
translate(y1)

## Codons
codons(y1)
which(width(codons(y1)) != 3)
codons(y1)[20:28]
```

---

`trimLRPatterns` *Trim Flanking Patterns from Sequences*

---

## Description

The `trimLRPatterns` function trims left and/or right flanking patterns from sequences.

## Usage

```
trimLRPatterns(Lpattern = "", Rpattern = "", subject,
               max.Lmismatch = 0, max.Rmismatch = 0,
               with.Lindels = FALSE, with.Rindels = FALSE,
               Lfixed = TRUE, Rfixed = TRUE, ranges = FALSE)
```

## Arguments

| | |
|---|---|
| `Lpattern` | The left pattern. |
| `Rpattern` | The right pattern. |
| `subject` | An XString object, XStringSet object, or character vector containing the target sequence(s). |
| `max.Lmismatch` | Either an integer vector of length `nLp = nchar(Lpattern)` representing an absolute number of mismatches (or edit distance if `with.Lindels` is `TRUE`) or a single numeric value in the interval `[0, 1)` representing a mismatch rate when aligning terminal substrings (suffixes) of `Lpattern` with the beginning (prefix) of `subject` following the conventions set by `neditStartingAt`, `isMatchingStartingAt`, etc.<br><br>When `max.Lmismatch` is `0L` or a numeric value in the interval `[0, 1)`, it is taken as a "rate" and is converted to `as.integer(1:nLp * max.Lmismatch)`, analogous to agrep (which, however, employs `ceiling`).<br><br>Otherwise, `max.Lmismatch` is treated as an integer vector where negative numbers are used to prevent trimming at the `i`-th location. When an input integer vector is shorter than `nLp`, it is augmented with enough `-1`s at the beginning to bring its length up to `nLp`. Elements of `max.Lmismatch` beyond the first `nLp` are ignored.<br><br>Once the integer vector is constructed using the rules given above, when `with.Lindels` is `FALSE`, `max.Lmismatch[i]` is the number of acceptable mismatches (errors) between the suffix `substring(Lpattern, nLp - i + 1, nLp)` of `Lpattern` and the first `i` letters of `subject`. When `with.Lindels` is `TRUE`, `max.Lmismatch[i]` represents the allowed "edit distance" between that suffix of `Lpattern` and `subject`, starting at position `1` of `subject` (as in `matchPattern` and `isMatchingStartingAt`).<br><br>For a given element `s` of the `subject`, the initial segment (prefix) `substring(s, 1, j)` of `s` is trimmed if `j` is the largest `i` for which there is an acceptable match, if any. |
| `max.Rmismatch` | Same as `max.Lmismatch` but with `Rpattern`, along with `with.Rindels` (below), and its initial segments (prefixes) `substring(Rpattern, 1, i)`.<br><br>For a given element `s` of the subject, with `nS = nchar(s)`, the terminal segment (suffix) `substring(s, nS - j + 1, nS)` of `s` is trimmed if `j` is the largest `i` for which there is an acceptable match, if any. |
| `with.Lindels` | If `TRUE`, indels are allowed in the alignments of the suffixes of `Lpattern` with the subject, at its beginning. See the `with.indels` arguments of the `matchPattern` and `neditStartingAt` functions for detailed information. |
| `with.Rindels` | Same as `with.Lindels` but for alignments of the prefixes of `Rpattern` with the subject, at its end. See the `with.indels` arguments of the `matchPattern` and `neditEndingAt` functions for detailed information. |
| `Lfixed, Rfixed` | Whether IUPAC extended letters in the left or right pattern should be interpreted as ambiguities (see `¿lowlevel-matching`` for the details). |
| `ranges` | If `TRUE`, then return the ranges to use to trim `subject`. If `FALSE`, then returned the trimmed `subject`. |

## Value

A new XString object, XStringSet object, or character vector with the "longest" flanking matches removed, as described above.

## Author(s)

P. Aboyoun and H. Jaffee

## See Also

`matchPattern`, `matchLRPatterns`, lowlevel-matching, XString-class, XStringSet-class

## Examples

```
Lpattern <- "TTCTGCTTG"
Rpattern <- "GATCGGAAG"
subject <- DNAString("TTCTGCTTGACGTGATCGGA")
subjectSet <- DNAStringSet(c("TGCTTGACGGCAGATCGG", "TTCTGCTTGGATCGGAAG"))

## Only allow for perfect matches on the flanks
trimLRPatterns(Lpattern = Lpattern, subject = subject)
trimLRPatterns(Rpattern = Rpattern, subject = subject)
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subjectSet)

## Allow for perfect matches on the flanking overlaps
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subjectSet,
               max.Lmismatch = 0, max.Rmismatch = 0)

## Allow for mismatches on the flanks
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subject,
               max.Lmismatch = 0.2, max.Rmismatch = 0.2)
maxMismatches <- as.integer(0.2 * 1:9)
maxMismatches
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subjectSet,
               max.Lmismatch = maxMismatches, max.Rmismatch = maxMismatches)

## Produce ranges that can be an input into other functions
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subjectSet,
               max.Lmismatch = 0, max.Rmismatch = 0, ranges = TRUE)
trimLRPatterns(Lpattern = Lpattern, Rpattern = Rpattern, subject = subject,
               max.Lmismatch = 0.2, max.Rmismatch = 0.2, ranges = TRUE)
```

---

# `xscat` *Concatenate sequences contained in XString, XStringSet and/or XStringViews objects*

---

## Description

This function mimics the semantic of `paste(..., sep="")` but accepts XString, XStringSet or XStringViews arguments and returns an XString or XStringSet object.

## Usage

```
xscat(...)
```

## Arguments

| | |
|---|---|
| `...` | One or more character vectors (with no NAs), XString, XStringSet or XStringViews objects. |

## Value

An XString object if all the arguments are either XString objects or character strings. An XStringSet object otherwise.

## Author(s)

H. Pages

## See Also

XString-class, XStringSet-class, XStringViews-class, `paste`

## Examples

```
## Return a BString object:
xscat(BString("abc"), BString("EF"))
xscat(BString("abc"), "EF")
xscat("abc", "EF")

## Return a BStringSet object:
xscat(BStringSet("abc"), "EF")

## Return a DNAStringSet object:
xscat(c("t", "a"), DNAString("N"))

## Arguments are recycled to the length of the longest argument:
xscat("x", LETTERS, c("3", "44", "555"))

## Concatenating big XStringSet objects:
library(drosophila2probe)
probes <- DNAStringSet(drosophila2probe)
mm <- complement(narrow(probes, start=13, end=13))
left <- narrow(probes, end=12)
right <- narrow(probes, start=14)
xscat(left, mm, right)

## Collapsing an XStringSet (or XStringViews) object with a small
## number of elements:
probes1000 <- as.list(probes[1:1000])
y1 <- do.call(xscat, probes1000)
y2 <- do.call(c, probes1000)  # slightly faster than the above
y1 == y2  # TRUE
## Note that this method won't be efficient when the number of
## elements to collapse is big (> 10000) so we need to provide a
## collapse() (or xscollapse()) function in Biostrings that will
## be efficient at doing this. Please complain on the Bioconductor
## mailing list (http://bioconductor.org/docs/mailList.html) if you
## need this.
```

## yeastSEQCHR1 *An annotation data file for CHR1 in the yeastSEQ package*

### Description

This is a single character string containing DNA sequence of yeast chromosome number 1. The data were obtained from the Saccharomyces Genome Database (ftp://genome-ftp.stanford.edu/pub/yeast/data_download/sequence/genomic_sequence/chromosomes/fasta/).

### Details

Annotation based on data provided by Yeast Genome project.

Source data built:Yeast Genome data are built at various time intervals. Sources used were downloaded Fri Nov 21 14:00:47 2003 Package built: Fri Nov 21 14:00:47 2003

### References

http://www.yeastgenome.org/DownloadContents.shtml

### Examples

```
data(yeastSEQCHR1)
nchar(yeastSEQCHR1)
```

# Index