String Search

COMP90049 Knowledge Technologies

Justin Zobel and Rao Kotagiri, CIS

Semester 1

THE UNIVERSITY OF MELBOURNE

String search with variation

Consider search for lines which mention of `Barton P. Miller`:

Allan Bricker, Tad Lebeck, Barton P. Miller
Mitali Bhattacharyya, David Cohrs, Barton Miller
David L. Cohrs, Barton P. Miller, Lisa A. Call
Barton P. Miller, Jong-Deok Choi
Jong-Deok Choi, Barton P. Miller, Robert Netzer
J.D. Choi, B.P. Miller
Barton P. Miller, Lars Fredriksen, Bryan So
Mitali Bhattacharyya, David Cohrs, Barton Miller
Young Moo Kang, Robert B. Miller, Roger Alan Pick
M. L. Powell, B. P. Miller
Joseph B. Miller, William F. O'Hearn
B.P. Miller, S. Tetelbaum, K. Webb
D. Draheim, B. Miller, S. Snyder

What is the exact string to be searched for?

A similar task: how to search a text file for email addresses from the `.com` domain?

Other examples of variant search

A progression of cases:

- Search for a word with known variant spelling (e.g., color, colour).
- Find the lines of text that contain two given words in a given order, with arbitrary strings in-between.
- Find the lines of text where the same string occurs twice; (a) adjacent or (b) with arbitrary strings in-between.

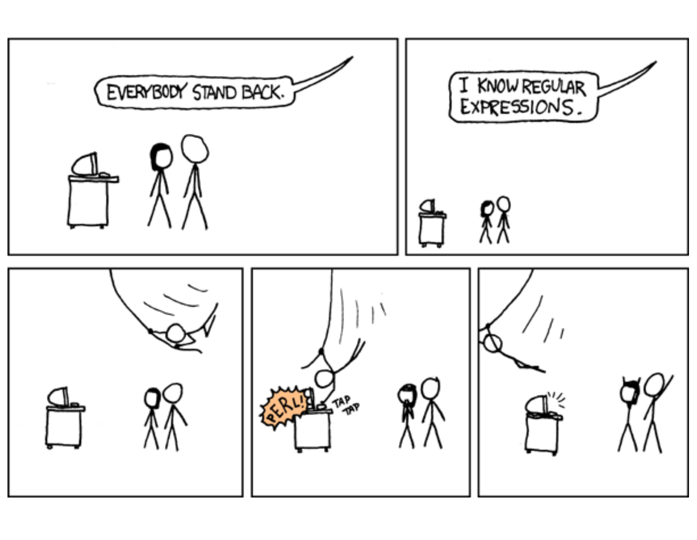
All of these are examples of search with *patterns*.

Search for patterns is a simple knowledge technology (arguably). It provides a mechanism for access into irregular data, but does lack the property of having an indeterminate outcome.

Pattern matching is a basic element of several programming languages (Python, Perl) and powerful Unix/Linux tools (shells, awk, sed, find, vi, egrep).

And it provides a context or contrast for more approximate mechanisms.

Regular expressions



Regular expressions

Regular expressions (regex, regexp) are patterns that match character strings.

They can be thought of as describing a set of strings.

- **Search:** Find the strings in a file that contain a substring that matches a given pattern (grep family).
  ```
  > egrep 'rudd' *.txt
  > egrep 'col(o|ou)r' *.txt
  ```
- **Find and replace:** Substitute some new string for the matching substring (sed, vi).
  ```
  s/rudd/gillard/g
  s/[dD]og/Canis lupis familiaris/g
  ```
- **Validate or test:** Check if new string is correct (awk, Python, Perl).
  ```
  $input ~ /gillard/
  $input ~ /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/
  ```
  Here the operator ~ in the expression `$input ~ /string/,` is a match operator that checks whether `$input` contains the pattern `/string/`.

Regular expressions

The four main concepts of regex mirror the four types of structure in imperative programming languages.

Sequence:	`i = 2; j = 3;`	Matching:	`/cat/`
Assignment:	`i = 2;`	Memoization:	`(`*pattern*`)`

Selection:	`if A:`
	`    do thing`
	`else:`
	`    do other thing`
		Alternation:	`/cat|dog/`

Repetition:	`while True:` `    i += 1`	Repetition:	`/(cat)*/`

Regular expressions

As the examples above show, regular expressions are a mix of literal characters and command or control characters. For example,

- `a` means "match the character `a`"
- | means *or*

`{ } [ ] ( ) ^ $ . | * + ? $ \` are known as *metacharacters* and need to be escaped by a backslash (\) to be used in a literal match; for example,
\$ means "match the character $", and
\\ means "match the character \".

Beware, some tools have different metacharacters. `?` in shells means the same as `.` in standard regex.

And in some cases \ turns a character into a metacharacter.

Here, we sometimes use / as a pattern delimiter. In some tools, it too is a metacharacter.

Matching

The foundation of regex is literal matching:

`/knowledge/`

- Each character matches itself.
- Matches are case sensitive.
- Whitespace is significant:
  `/over priced/` won't match "`overpriced`"
- Substrings are uninterpreted; they are not assumed to be whole words or have any specific semantics.
  `/lane/` will match "`planet`"

Another special case is newline. Many tools that incorporate regex are line-oriented, and either cannot match across a line break or do so in idiosyncratic ways.

Matching

The wildcard `.` is the most basic metacharacter

- Matches any single character (except a newline); good for crossword puzzles:

```
> egrep '.n.wl.d..' ../local/words-345.txt
  acknowledge
  acknowledged
  ⋮
```

The anchors `^` and `$` match the start and end of a line or string, respectively.

- `> egrep '^.n.wl.d..$' ../local/words-345.txt`
  `knowledge`

(Note that data for the subject is kept in the directory `/home/subjects/comp90049/local/` on the CSS servers.)

Alternation

The | metacharacter expresses alternation or disjunction

- `/a|b|c/` matches "`a`", "`b`", or "`c`".
- `/cat|dog/` matches "`cat`" or "`dog`".
- `/\$(US|AU|CD)/` matches "`$US`", "`$AU`", or "`$CD`".

A note on precedence: the | character has low precedence, and the parentheses in the last example are necessary.

Check – what is the difference between:

- `> egrep 'ed|ing$' /usr/share/dict/words`
- `> egrep '(ed|ing)$' /usr/share/dict/words`

Repetition

The precise number of characters to match may be unknown; instead, we specify a repetition construction.

Some repetitions involve an arbitrary number:

- `*`: zero or more of the preceding element
- `?`: zero or one of the preceding element
- `+`: one or more of the preceding element

These are *greedy* – they match as many characters as they can. So `.*` will always match a complete string and `a.*b` will pick up the *last* “`b`” in the string.

Sometimes we care, but only approximately, about number.

- `{n}`: exactly *n* of the preceding element
- `{m,n}`: between *m* and *n* (inclusive) of the preceding element
- `{n,}`: *n* or more of the preceding element
- `{,m}`: up to *m* of the preceding element

For example, `labell?ing` matches “`labeling`”, “`labelling`”.

Character classes

Sometimes, rather than one particular character or any character, we want to match any of a set of characters.

Some possible character classes:

- `/[Kk]nowledge/`
- `/[aeiou]/` –note that this is equivalent to `/a|e|i|o|u/` or `/(a|e|i|o|u)/`
- `/^\$[0-9]+/`
- `/^[A-Z][a-z]*/`
- `/ [A-Za-z]+ /`

Observe that ranges can be used to denote the character classes.

Observe also that within `[,]`, metacharacters may be used in their literal meaning. For example, in some languages, the class `[\$]` matches "\" or "$".

Negative classes

A second use of the ^ metacharacter is to negate character classes.
`/[^A-Za-z]/` matches any non-alpha character.

In some languages, ^ and - are the only metacharacters within ranges.
(But see the discussion of named classes on the next slide.)

What do these match?

- `/[^0-9]/`
- `/[^"]/`
- `/<[^>]>/`

Named classes

Some character classes are used so frequently that they have names:

- `[0-9]` = `[[:digit:]]` = `\d`
- `[a-zA-Z0-9_]` = `[[:word:]]` = `\w`
- `[\ \t\r\n\f]` = `[[:space:]]` = `\s`

As do their negations:

- `[^0-9]` = `\D`
- `[^a-zA-Z0-9_]` = `\W`
- `[^\ \t\r\n\f]` = `\S`

Beware again: Which named character classes are available and how they are represented depends on the software you use.

Back-references or memoization

Placing a pattern in parentheses leads to the match being stored as a variable.

The first stored pattern has the name `\1`, the *n*th is `\n`. Sadly, there is no way of operating on stored patterns, but they can be accessed for subsequent matching.

Example: What does `/([a-zA-Z]+) +\1/` match?

They are particularly powerful in string substitution.

Example: `s/([A-Z])[a-z]+ ([A-Z][a-z]+)/\1. \2/`

Putting it all together

Now we can parse the regex from earlier on:

```
/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/
```

- `^[A-Z0-9._%+-]+`: match one or more of these characters
- `@`: followed by an "`@`"
- `[A-Z0-9.-]+`: followed by one or more of these characters
- `\.`: followed by a dot
- `[A-Z]{2,4}$`: followed by 2–4 upper case letters, and then end of line

- What do you think this pattern is for?
- How might this pattern be improved?

Programming with patterns

There are several pattern-based programming languages, in particular `Python` and `Perl`. There are also good command-line tools, in particular `sed` and `awk`. (Perl is also used in this way.)

A quick look at `awk(Aho, Weinberger and Kernighan)` ...

- Line-oriented; each block of code describes a series of operations to be applied to a line of input. Every line is processed in turn.
- Code is C-like (i.e., Java-like, C++-like).
- Lines of input are parsed into fields, and assigned to variables `$1`, `$2`, `$3`, ...
- A line of input is only processed if it matches a pattern.
- Fields may be tested to see if they match a pattern.

Programming with patterns

```
Baughman Edward D. <Edward.Baughman@ENRON.com>
Baughman Edward <Edward.Baughman@ENRON.com>
Becker Lorraine <Lorraine.Becker@ENRON.com>
"Beck, Sally" <Sally.Beck@ENRON.com>,
Beck Sally <Sally.Beck@ENRON.com>
bejules@hotmail.com
Ben <Ben.Brasseaux@ENRON.com>
```

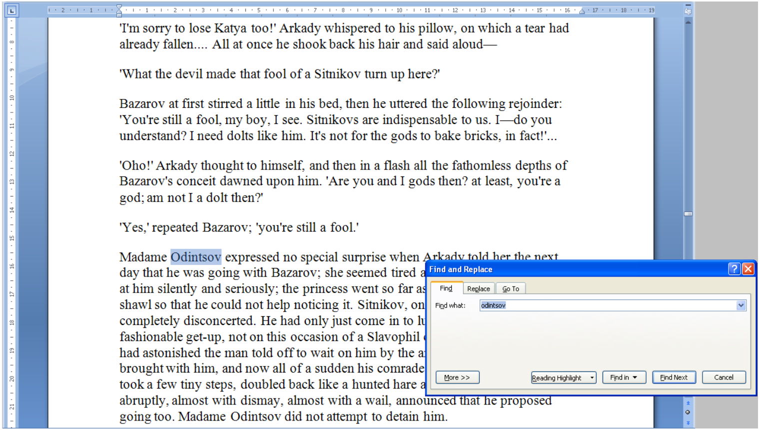
This is a complete `awk` program for processing the input above.

```
/<[^ ]*@ENRON[^ ]*>/{
    for( i=1 ; i<=NF ; i++ )
        if( $i ~ /^[A-Za-z]*$/ ) print $i;
}
```

`NF` is a special variable containing the number of fields in the current line. Other variables (e.g., `i`) are created automatically when they are referenced.

Exact string search

The task: find a *query* string *q* in a *target* string *t*.



Exact string search

A universal, elementary component of many software tools. Uses:

- Find words in text, functions in programs
- Extract lines from files
- Find files in file systems

(But note: not the same as the kind of query provided by web search tools, as the string can be any sequence of characters. In web search, the query must consist of complete words.)

In the most basic form, characters must match exactly (correct case).

Example: find the `ham` in

```
Cordell Hammett, David Luckham, Robert Balzer,
Thomas Cheatham, Charles Rich
```

What's happening algorithmically?

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

`bbanbantbanterbalanca`

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
ba
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
 bant
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
  b
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
   b
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
    bante
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
     b
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
      b
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
         b
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
        banter
```

Naïve matching

Consider the query `banter` and string `bbanbantbanterbalanca`.

```
bbanbantbanterbalanca
        banter
```

Success at position 9! But the method involves up to $|q|$ comparisons for each of $|t|$ characters.

Notation: $|s|$ is the number of characters in string s, and q and t are query and target respectively.

Something better: Boyer-Moore matching

Idea: Suppose we start the comparisons at the end of the query?

```
bbanbantbanterbalanca
banter
```

Something better: Boyer-Moore matching

Idea: Suppose we start the comparisons at the end of the query?

```
bbanbantbanterbalanca
banter
```

Analyse the query before we start searching:
Observe that, for example, if the mismatch is ‘a’ then we must shift at least 4 characters.

```
bbanbantbanterbalanca
    banter
```

Something better: Boyer-Moore matching

Idea: Suppose we start the comparisons at the end of the query?

```
bbanbantbanterbalanca
banter
```

Analyse the query before we start searching:
Observe that, for example, if the mismatch is ‘a’ then we must shift at least 4 characters.

```
bbanbantbanterbalanca
    banter
```

It is ‘a’ again; repeat.

```
bbanbantbanterbalanca
        banter
```

If the mismatch char did not occur in `banter`, we could jump 6.

Text compression

- Compression can be seen as a mechanism for efficient representation of repetition. For example:

... Don Quixote raised his eyes to heaven, and fixing his thoughts, apparently, upon his lady Dulcinea, exclaimed ... called Don Quixote of La Mancha, knight-errant and adventurer, and captive to the peerless and beautiful lady Dulcinea del Toboso ... without discussing Don Quixote's demand or asking who Dulcinea might be ... present yourselves before the lady Dulcinea del Toboso ... prove that the lady Dulcinea del Toboso has been trifling ...

... J raised his eyes to heaven, H fixing his thoughts, apparently, upon his F C, exclaimed ... called K, knight-errant H adventurer, H captive to I peerless H beautiful L ... without discussing J's demand or asking who C might be ... present yourselves before I L ... prove that I L has been trifling ...

Dictionary:

A←Don B←Quixote C←Dulcinea D←del E←Toboso
F←lady G←Mancha H←and I←the
J←A B K←I of La G L←F C D E

Text compression

When a common string is recognized and replaced by a code, the outcome is a codebook and (hopefully!) a more compact text.

A specific form of this “dictionary” approach is to look backwards for the most recent occurrence of the same string. Variations of this approach are the basis of the *zip* family of compression algorithms.

The methods were first described by Lempel and Ziv (and Lempel, Ziv and Welch), and are usually known as the LZ family.

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

`ehrt◇heherttherehere`

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇heherttherehere
ehrt◇(3,1)eherttherehere
```

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇heherttherehere
ehrt◇(3,1)(5,1)herttherehere
```

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇heherttherehere
ehrt◇(3,1)(5,1)(2,2)rttherehere
```

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇heherttherehere
ehrt◇(3,1)(5,1)(2,2)(6,1)ttherehere
```

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇heherttherehere
ehrt◇(3,1)(5,1)(2,2)(6,1)(6,1)therehere
```

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇heherttherehere
ehrt◇(3,1)(5,1)(2,2)(6,1)(6,1)(1,1)herehere
```

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `hehertttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇hehertttherehere
ehrt◇(3,1)(5,1)(2,2)(6,1)(6,1)(1,1)(5,3)ehere
```

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇heherttherehere
ehrt◇(3,1)(5,1)(2,2)(6,1)(6,1)(1,1)(5,3)(2,1)here
```

LZ-like example

Consider an alphabet of four characters, {`e h r t`}, and the string `heherttherehere`.

A simple take on LZ compression:

- Pretend that the alphabet is written to the left of the string.
- Encode each substring in the string by a pointer to the left to an occurrence of the same string.
- Each pointer is a pair consisting of *d*, *l*, distance to the left and number of characters to copy.

```
ehrt◇heherttherehere
ehrt◇(3,1)(5,1)(2,2)(6,1)(6,1)(1,1)(5,3)(2,1)here
```

The pointers and counts can be very compact – maybe 6 to 16 bits each. In effect the pointers be seen as creating a dictionary embedded in the string.

If the matched substrings become long, pointer+count size can be much less than the original string length. What happens with the string `aaaaaaaaaaaaaaaaaaa`...?

Aside: information theory

An underlying principle here is that a *message* has a definable *information content*.

In the context of knowing the kinds of things that can be included in a dictionary, *information theory* tells us the minimum possible size of the message.

Practical methods used in compression can approach these minima. These are elegant techniques that show how a principled analysis can yield significant practical outcomes.

Information theory can also be used to describe physical events, e.g., quantum events and the behaviour of black holes. It provides a profound link between the physical universe and the compuverse.

LZ and string search

A key element of LZ implementation is being able to find matching strings efficiently.

There are many ways of doing this! But they all provide mechanisms for exact string matching, where the set of strings is dynamic (varying, or strictly growing).

Much research in compression consists of discovering new ways to recognize interesting repeated patterns.

These patterns may exist only in a specific context (that word again). For example, English text is one such context. Another is, say, XML data.

LZ and string search

But which string to search for? Is it better to search for the longest match, or the nearest one?

Remember that the larger the number to be encoded, the more bits are required.

```
an angry angler tangled a tangy anglican angle.
```

LZ and string search

But which string to search for? Is it better to search for the longest match, or the nearest one?

Remember that the larger the number to be encoded, the more bits are required.

```
an angry angler tangled a tangy anglican angle.
an angry angler tangled a tangy anglican (9,4)e.
```

LZ and string search

But which string to search for? Is it better to search for the longest match, or the nearest one?

Remember that the larger the number to be encoded, the more bits are required.

```
an angry angler tangled a tangy anglican angle.
an angry angler tangled a tangy anglican (14,3)le.
```

LZ and string search

But which string to search for? Is it better to search for the longest match, or the nearest one?

Remember that the larger the number to be encoded, the more bits are required.

```
an angry angler tangled a tangy anglican angle.
an angry angler tangled a tangy anglican (24,5).
```

LZ and string search

But which string to search for? Is it better to search for the longest match, or the nearest one?

Remember that the larger the number to be encoded, the more bits are required.

```
an angry angler tangled a tangy anglican angle.
an angry angler tangled a tangy anglican(33,6).
```

LZ and string search

But which string to search for? Is it better to search for the longest match, or the nearest one?

Remember that the larger the number to be encoded, the more bits are required.

```
an angry angler tangled a tangy anglican angle.
an angry angler tangled a tangy anglican(33,6).
```

It can be formally shown that such problems cannot be optimally solved through local or greedy methods and are combinatorially expensive.

That is, discovery of the best solution involves trying every combination of possible substitutions. Some local pruning is possible, but thorough search for a best solution would make compression impossibly slow.

LZ and string search

LZ methods avoid the combinatorics altogether by using simple heuristics that are experimentally found to produce "good enough" solutions on typical data.

The main heuristic is to do no search at all (!!) and instead build a dictionary of string substitutions made so far.

- The initial dictionary consists of all single letters.
- Only dictionary entries can be matched. The longest match is always taken.
- When a substitution such as `ang` is made in `tangled`, the entry `angl` is added to the dictionary.

Now what happens with the string `aaaaaaaaaaaaaaaaaaa`...?

Another heuristic is to allow a form of forward matching, giving run-length encoding. For example,

```
dogcatcatcatcatcatcatcatdog
dogcat(3,18)dog
```

Summary

- What are some applications of string search? Both obvious and non-obvious.
- There is a link between repetitiveness, compressibility, and information theory.
- In what applications does string search involve uncertainty?
- What are regular expressions and what are they used for?
- What are the main concepts used in regular expressions?
- What kinds of search tasks can and cannot be addressed with regular expressions?

- Consolidate your understanding of the regular expression metacharacters; some useful references:
`docs.python.org/dev/howto/regex.html`
`perldoc perlretut` on any CSSE server
`java.sun.com/docs/books/tutorial/essential/regex/`

- Some readings:
`en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm`
`en.wikipedia.org/wiki/Lempel-Ziv-Welch`