# Road map

- Introduction
- Data Model and HTML encoding
- Wrapper induction
- **Automatic Wrapper Generation: Two Problems**
- String Matching and Tree Matching
- Multiple Alignments
- Building DOM Trees
- Extraction Given a List Page: Flat Data Records
- Extraction Given a List Page: Nested Data Records
- Extraction Given Multiple Pages
- Summary

# Automatic wrapper generation

- Wrapper induction (supervised) has two main shortcomings:
  - It is unsuitable for a large number of sites due to the manual labeling effort.
  - Wrapper maintenance is very costly. The Web is a dynamic environment. Sites change constantly. Since rules learnt by wrapper induction systems mainly use formatting tags, if a site changes its formatting templates, existing extraction rules for the site become invalid.

# Unsupervised learning is possible

- Due to these problems, automatic (or unsupervised) extraction has been studied.
- Automatic extraction is possible because data records (tuple instances) in a Web site are usually encoded using a very small number of fixed templates.
- It is possible to find these templates by mining repeated patterns.

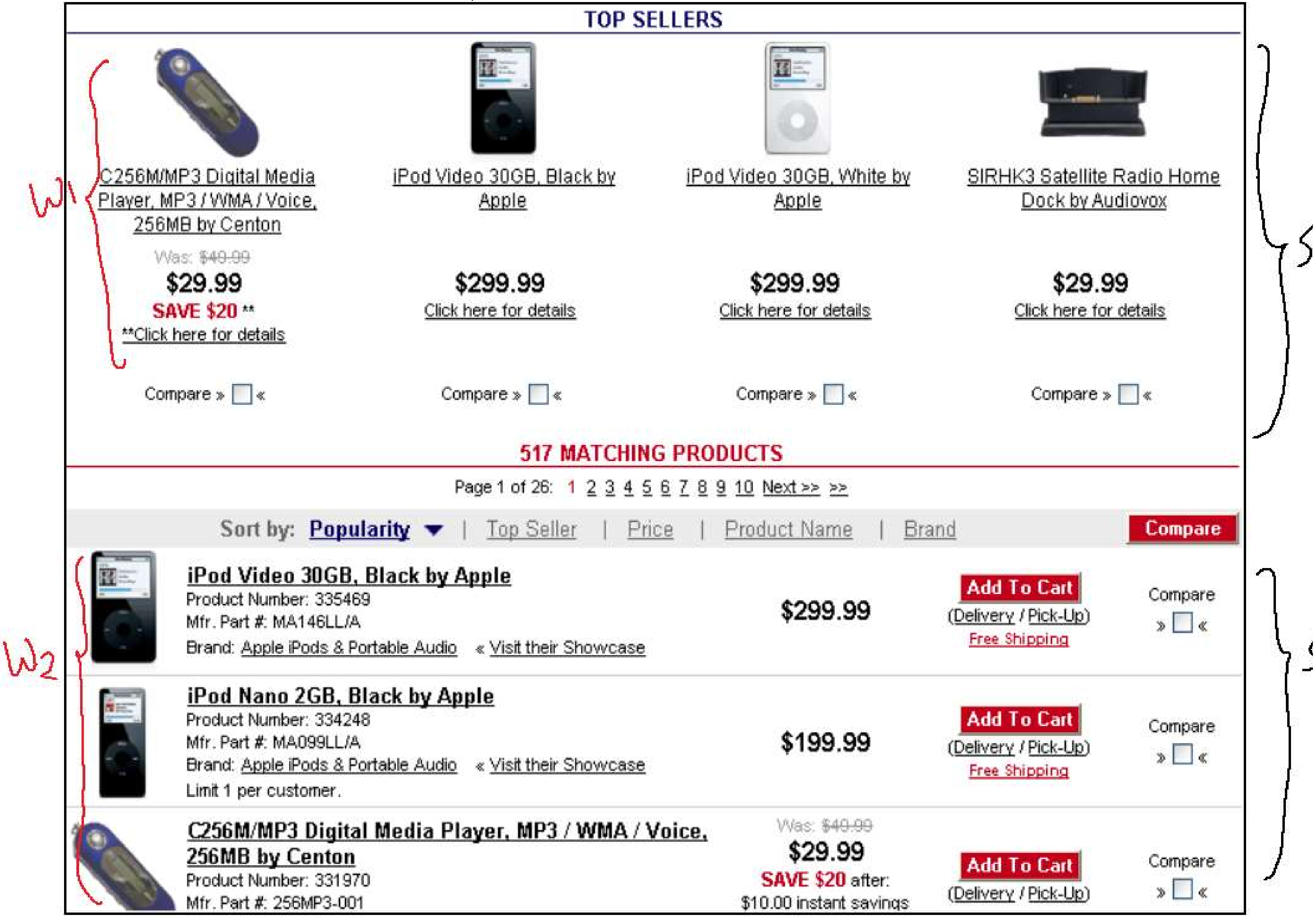
# Two data extraction problems

- In Sections 8.1.2 and 8.2.3, we described an abstract model of structured data on the Web (i.e., nested relations), and a HTML mark-up encoding of the data model respectively.
- The general problem of data extraction is to recover the hidden schema from the HTML mark-up encoded data.
- We study two extraction problems, which are really quite similar.

# Problem 1: Extraction given a single list page

- **Input**: A single HTML string ***S***, which contain *k* non-overlapping substrings $\boldsymbol{s_1, s_2, \ldots, s_k}$ with each $\boldsymbol{s_i}$ encoding an instance of a set type. That is, each $s_i$ contains a collection $\boldsymbol{W_i}$ of $\boldsymbol{m_i}$ ($\geq 2$) non-overlapping sub-substrings encoding *mi* instances of a tuple type.
- **Output**: ***k*** tuple types $\boldsymbol{\sigma_1, \sigma_2, \ldots, \sigma_k}$, and ***k*** collections $\boldsymbol{C_1, C_2, \ldots, C_k}$, of instances of the tuple types such that for each collection $\boldsymbol{C_i}$ there is a HTML encoding function $enc_i$ such that $enc_i$: $C_i \rightarrow W_i$ is a bijection.

S: Full Web Page



# Extraction results


$W_1$

(a). An example page segment

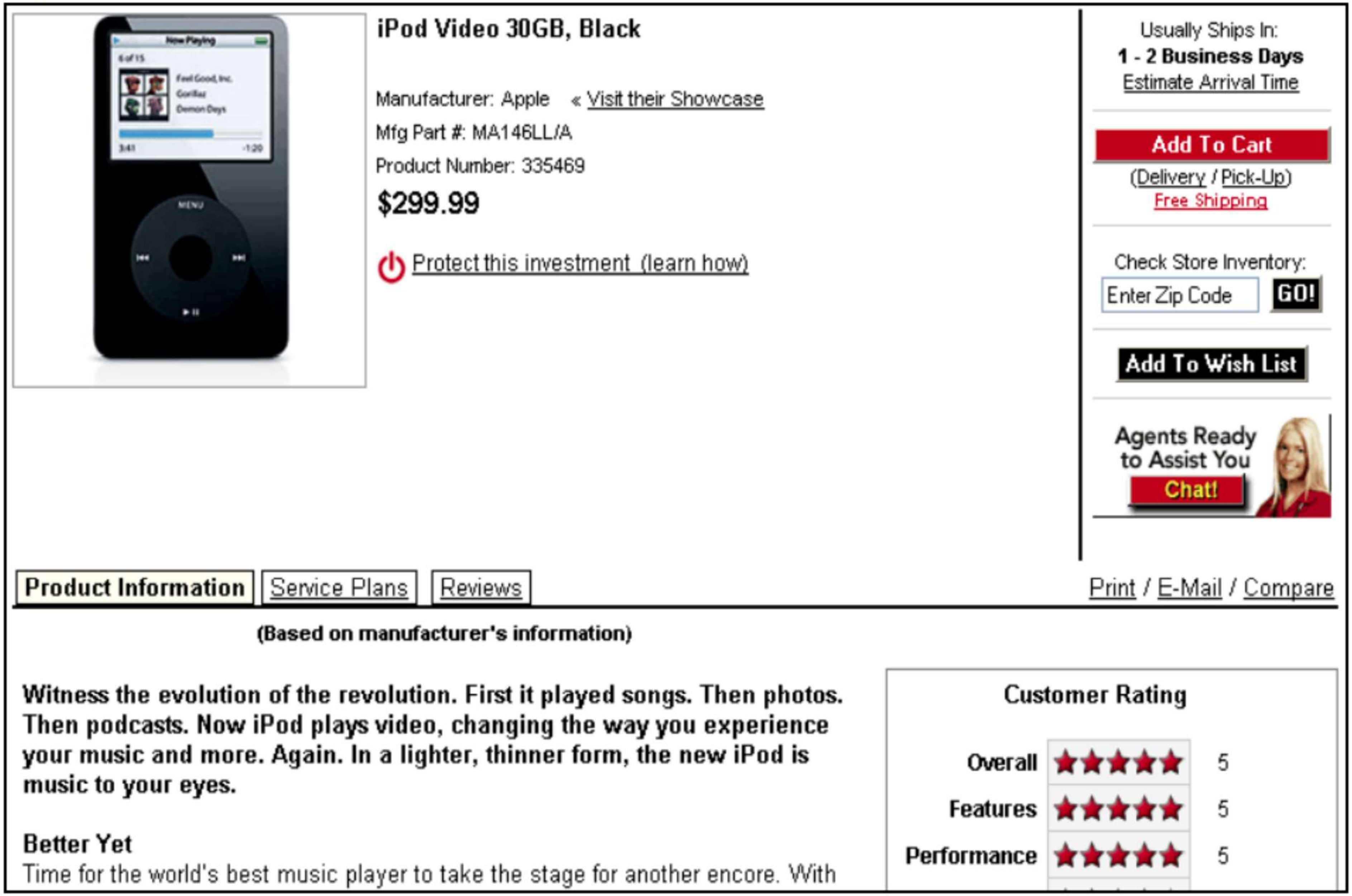
| | | | | | |
|---|---|---|---|---|---|
| image 1 | Cabinet Organizers by Copco | 9-in. | Round Turntable: White | ***** | $4.95 |
| image 1 | Cabinet Organizers by Copco | 12-in. | Round Turntable: White | ***** | $7.95 |
| image 2 | Cabinet Organizers | 14.75x9 | Cabinet Organizer (Non-skid): White | ***** | $7.95 |
| image 3 | Cabinet Organizers | 22x6 | Cookware Lid Rack | **** | $19.95 |

$C_1$

(b). Extraction results

# Problem 2: Data extraction given multiple pages

- **Input**: A collection *W* of *k* HTML strings, which encode *k* instances of the same type.
- **Output**: A type $\sigma$, and a collection *C* of instances of type $\sigma$, such that there is a HTML encoding *enc* such that *enc*: $C \rightarrow W$ is a bijection.


iPod Video 30GB, Black
Manufacturer: Apple « Visit their Showcase
Mfg Part #: MA146LL/A
Product Number: 335469
$299.99
Protect this investment (learn how)
Usually Ships In:
1 - 2 Business Days
Estimate Arrival Time
Add To Cart
(Delivery / Pick-Up)
Free Shipping
Check Store Inventory:
Enter Zip Code
GO!
Add To Wish List
Agents Ready to Assist You
Chat!
Product Information
Service Plans
Reviews
Print / E-Mail / Compare
(Based on manufacturer's information)
Witness the evolution of the revolution. First it played songs. Then photos. Then podcasts. Now iPod plays video, changing the way you experience your music and more. Again. In a lighter, thinner form, the new iPod is music to your eyes.
Better Yet
Time for the world's best music player to take the stage for another encore. With
Customer Rating
Overall 5
Features 5
Performance 5

# Templates as regular expressions

- A regular expression can be naturally used to model the HTML encoded version of a nested type.
- Given an alphabet of symbols $\Sigma$ and a special token "*#text*" that is not in $\Sigma$,
  - a *regular expression* over $\Sigma$ is a string over $\Sigma \cup$ {*#text*, *, ?, |, (, )} defined as follows:

# Regular Expressions

| Symbol(s) | Meaning | Example | Example matches |
|---|---|---|---|
| * | Matches the preceding character, subexpression, or bracketed character, 0 or more times. | a*b* | aaaaaaaa, aaabbbbb, bbbbbb |
| + | Matches the preceding character, subexpression, or bracketed character, 1 or more times. | a+b+ | aaaaaaaab, aaabbbbb, abbbbbb |
| [] | Matches any character within the brackets (i.e., "Pick any one of these things"). | [A-Z]* | APPLE, CAPITALS, QWERTY |
| () | A grouped subexpression (these are evaluated first, in the "order of operations" of regular expressions). | (a*b)* | aaabaab, abaaab, ababaaaaab |
| {m, n} | Matches the preceding character, subexpression, or bracketed character between *m* and *n* times (inclusive). | a{2,3}b{2,3} | aabbb, aaabbb, aabb |
| [^] | Matches any single character that is *not* in the brackets. | [^A-Z]* | apple, lowercase, qwerty |

<table>
<tr><th>Symbol(s)</th><th>Meaning</th><th>Example</th><th>Example matches</th></tr>
<tr><td></td><td></td><td></td><td>qwerty</td></tr>
<tr><td>|</td><td>Matches any character, string of characters, or subexpression, separated by the I (note that this is a vertical bar, or pipe, not a capital i).</td><td>b(a|i|e)d</td><td>bad, bid, bed</td></tr>
<tr><td>.</td><td>Matches any single character (including symbols, numbers, a space, etc.).</td><td>b.d</td><td>bad, bzd, b$d, b d</td></tr>
<tr><td>^</td><td>Indicates that a character or subexpression occurs at the beginning of a string.</td><td>^a</td><td>apple, asdf, a</td></tr>
<tr><td>\</td><td>An escape character (this allows you to use special characters as their literal meanings).</td><td>\. \| \\</td><td>. | \</td></tr>
<tr><td>$</td><td>Often used at the end of a regular expression, it means “match this up to the end of the string.” Without it, every regular expression has a de facto “.*” at the end of it, accepting strings where only the first part of the string matches. This can be thought of as analogous to the ^ symbol.</td><td>[A-Z]*[a-z]*$</td><td>ABCabc, zzzyx, Bob</td></tr>
<tr><td>?!</td><td>“Does not contain.” This odd pairing of symbols, immediately preceding a character (or regular expression), indicates that that character should not be found in that specific place in the larger string.</td><td>^((?![A-Z]).)*$</td><td>no-caps-here, $ymb0ls a4e f!ne</td></tr>
</table>

# Regular expressions

- The empty string $\varepsilon$ and all elements of $\Sigma \cup \{\#text\}$ are regular expressions.
- If $A$ and $B$ are regular expressions, then $AB$, $(A|B)$ and $(A)^?$ are regular expressions, where $(A|B)$ stands for $A$ or $B$ and $(A)^?$ stands for $(A|\varepsilon)$.
- If $A$ is a regular expression, $(A)^*$ is a regular expression, where $(A)^*$ stands for $\varepsilon$ or $A$ or $AA$ or ...

We also use $(A)+$ as a shortcut for $A(A)^*$, which can be used to model the set type of a list of tuples. $(A)^?$ indicates that $A$ is optional. $(A|B)$ represents a disjunction.

# Regular expressions and extraction

- Regular expressions are often employed to represent templates (or encoding functions).
- However, templates can also be represented as string or tree patterns as we will see later.
- Extraction:
  - Given a regular expression, a nondeterministic finite-state automaton can be constructed and employed to match its occurrences in string sequences representing Web pages.
  - In the process, data items can be extracted, which are text strings represented by *#text*.

# Road map

- Introduction
- Data Model and HTML encoding
- Wrapper induction
- Automatic Wrapper Generation: Two Problems
- **String Matching and Tree Matching**
- Multiple Alignments
- Building DOM Trees
- Extraction Given a List Page: Flat Data Records
- Extraction Given a List Page: Nested Data Records
- Extraction Given Multiple Pages
- Summary

# Some useful algorithms

- The key is to finding the encoding template from a collection of encoded instances of the same type.
- A natural way to do this is to detect repeated patterns from HTML encoding strings.
- String edit distance and tree edit distance are obvious techniques for the task. We describe these techniques.

# String edit distance

- String edit distance: the most widely used string comparison technique.
- The **edit distance** of two strings, *s*1 and *s*2, is defined as the minimum number of *point mutations* required to change *s*1 into *s*2, where a point mutation is one of:
  - (1) change a letter,
  - (2) insert a letter, and
  - (3) delete a letter.

# String edit distance (definition)

Assume we are given two strings $s_1$ and $s_2$. The following recurrence relations define the edit distance, $d(s_1, s_2)$, of two strings $s_1$ and $s_2$:

$$d(\varepsilon, \varepsilon) = 0 \qquad // \ \varepsilon \text{ represents an empty string}$$

$$d(s, \varepsilon) = d(\varepsilon, s) = |s| \qquad // \ |s| \text{ is the length of string } s$$

$$d(s_1+\mathrm{ch}_1, s_2+ch_2) = \min(d(s_1, s_2) + r(ch_1, ch_2), d(s_1+ch_1, s_2) + 1, d(s_1, s_2+ch_2) + 1)$$

where $ch_1$ and $ch_2$ are the last characters of $s_1$ and $s_2$ respectively, and $r(ch_1, ch_2) = 0$ if $ch_1 = ch_2$; $r(ch_1, ch_2) = 1$, otherwise.

# Dynamic programming

We can use a two-dimensional matrix, $m[0..|s_1|,0..|s_2|]$ to hold the edit distances. The low right corner cell $m(|s_1|+1, |s_2|+1)$ will furnish the required value of the edit distance $d(s_1, s_2)$.

$$m[0, 0] = 0$$

$$m[i, 0] = i, \ \ i = 1, 2..., |s_1|$$

$$m[0, j] = j, \ \ j = 1, 2 ..., |s_2|$$

$$m[i, j] = \min(m[i\text{-}1, j\text{-}1] + r(s_1[i], s_2[j]), m[i\text{-}1, j] + 1, m[i, j\text{-}1] + 1 ),$$

where $i = 1, 2..., |s_1|, j = 1, 2..., |s_2|$, and $r(s_1[i], s_2[j]) = 0$ if $s_1[\mathrm{i}] = s_2[j]$; $r(s_1[i], s_2[j]) = 1$, otherwise.

# An example

**Example 1**: We want to compute the edit distance and find the alignment of the following two strings:

$s_1$: X G Y X Y X Y X
$s_2$: X Y X Y X Y T X

- The edit distance matrix and back trace path

| | $s_1$ | X | G | Y | X | Y | X | Y | X |
|---|---|---|---|---|---|---|---|---|---|
| $s_2$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| X | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Y | 2 | 1 | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| X | 3 | 2 | 2 | 2 | 1 | 2 | 3 | 4 | 5 |
| Y | 4 | 3 | 3 | 2 | 2 | 1 | 2 | 3 | 4 |
| X | 5 | 4 | 4 | 3 | 2 | 2 | 1 | 2 | 3 |
| Y | 6 | 5 | 5 | 4 | 3 | 2 | 2 | 1 | 2 |
| T | 7 | 6 | 6 | 5 | 4 | 3 | 3 | 2 | 2 |
| X | 8 | 7 | 7 | 6 | 5 | 4 | 3 | 3 | 2 |

- alignment

$s_1$: X G Y X Y X Y – X
$s_2$: X – Y X Y X Y T X

# Example

- String1: ABRTGQAS
- String2: ATBTGYQS

- Find the distance.

# Example 2

- String1: ABRGQSE
- String2: YBRGEQUS

- Find the distance.