Overview

In this topic we will look at pattern-matching algorithms for strings. Particularly, we will look at the Rabin-Karp algorithm, the Knuth-Morris-Pratt algorithm, and the Boyer-Moore algorithm.

We will also consider a dynamic programming solution to the Longest Common Substring problem.

Finally we will examine some file compression algorithms, including Huffman coding, and the Ziv Lempel algorithms.

Pattern Matching

We consider the following problem. Suppose $T$ is a string of length $n$ over a finite alphabet $\Sigma$, and that $P$ is a string of length $m$ over $\Sigma$.

The pattern-matching problem is to find occurrences of $P$ within $T$. Analysis of the problem varies according to whether we are searching for all occurrences of $P$ or just the first occurrence of $P$.

For example, suppose that we have $\Sigma = \{a, b, c\}$ and

$$T = \text{abaaabaccaabbaccaabacaababaac}$$

$$P = \text{aab}$$

Our aim is to find all the substrings of the text that are equal to aab.
The naive method

The naive pattern matcher simply considers every possible shift $s$ in turn, using a simple loop to check if the shift is valid.

When $s = 0$ we have

```
abaaabccaabbaccaababacaababaac
aab
```

which fails at the second character of the pattern.

When $s = 1$ we have

```
abaaabccaabbaccaababacaababaac
aab
```

which fails at the first character of the pattern.

Eventually this will succeed when $s = 3$.

Analysis of the naive method

In the worst case, we might have to examine each of the $m$ characters of the pattern at every candidate shift.

The number of possible shifts is $n - m + 1$ so the worst case takes

$$m(n - m + 1)$$

comparisons.

The naive string matcher is inefficient because when it checks the shift $s$ it makes no use of any information that might have been found earlier (when checking previous shifts).

For example if we have

```
0000000100001000000100000000000000
00000001
```

then it is clear that no shift $s \leq 9$ can possibly work.

Rabin-Karp algorithm

The naive algorithm basically consists of two nested loops — the outermost loop runs through all the $n - m + 1$ possible shifts, and for each such shift the innermost loop runs through the $m$ characters seeing if they match.

Rabin and Karp propose a modified algorithm that tries to replace the innermost loop with a single comparison as often as possible.

Consider the following example, with alphabet being decimal digits.

```
122938491281760821308176283101
176
```

Suppose now that we have computer words that can store decimal numbers of size less than 1000 in one word (and hence compare such numbers in one operation).

Then we can view the entire pattern as a single decimal number and the substrings of the text of length $m$ as single numbers.

Rabin-Karp continued

Thus to try the shift $s = 0$, instead of comparing

```
1 - 7 - 6
```

against

```
1 - 2 - 2
```

character by character, we simply do one operation comparing 176 against 122.

It takes time $O(m)$ to compute the value 176 from the string of characters in the pattern $P$.

However it is possible to compute all the $n - m + 1$ decimal values from the text just in time $O(n)$, because it takes a constant number of operations to get the “next” value from the previous.

To go from 122 to 229 only requires dropping the 1, multiplying by 10 and adding the 9.
Rabin-Karp formalized

Being a bit more formal, let \( P[1..m] \) be an array holding the pattern and \( T[1..n] \) be an array holding the text.

We define the values

\[
p = P[m] + 10P[m-1] + \cdots + 10^{m-1}P[1]
\]

\[
t_s = T[s+m] + 10T[s+m-1] + \cdots + 10^{m-1}T[s+1]
\]

Then clearly the pattern matches the text with shift \( s \) if and only if \( t_s = p \).

The value \( t_{s+1} \) can be calculated from \( t_s \) easily by the operation

\[
t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s + m + 1]
\]

If the alphabet is not decimal, but in fact has size \( d \), then we can simply regard the values as \( d \)-ary integers and proceed as before.

But what if the pattern is long?

This algorithm works well, but under the unreasonable restriction that \( m \) is sufficiently small that the values \( p \) and \( \{t_s \mid 0 \leq s \leq n - m\} \) all fit into a single word.

To make this algorithm practical Rabin and Karp suggested using one-word values related to \( p \) and \( t_s \) and comparing these instead. They suggested using the values

\[
p' = p \mod q
\]

and

\[
t'_s = t_s \mod q
\]

where \( q \) is some large prime number but still sufficiently small that \( dq \) fits into one word.

Again it is easy to see that \( t'_{s+1} \) can be computed from \( t'_s \) in constant time.

The whole algorithm

If \( t'_s \neq p' \) then the shift \( s \) is definitely not valid, and can thus be rejected with only one comparison. If \( t'_s = p' \) then either \( t_s = p \) and the shift \( s \) is valid, or \( t_s \neq p \) and we have a spurious hit.

The entire algorithm is thus:

Compute \( p' \) and \( t'_0 \)

for \( s \leftarrow 0 \) to \( n - m \) do

if \( p' = t'_s \) then

if \( T[s+1..s+m] = P[1..m] \) then

output “shift \( s \) is valid”

end if

end if

Compute \( t'_{s+1} \) from \( t'_s \)
end for

The worst case time complexity is the same as for the naive algorithm, but in practice where there are few matches, the algorithm runs quickly.

Example

Suppose we have the following text and pattern

\[
5 4 1 4 2 1 3 5 6 2 1 4 1 4
4 1 4
\]

Suppose we use the modulus \( q = 13 \), then \( p' = 414 \mod 13 = 11 \).

What are the values \( t'_0, t'_1, \) etc associated with the text?

\[
5 4 \underline{1} \underline{4} \underline{2} 1 3 5 6 2 1 4 1 4
\]

\[
t'_0 = 8
\]

\[
5 4 \underline{3} \underline{4} 2 1 3 5 6 2 1 4 1 4
\]

\[
t'_1 = 11
\]

This is a genuine hit, so \( s = 1 \) is a valid shift.

\[
5 4 1 4 \underline{2} 1 3 5 6 2 1 4 1 4
\]

\[
t'_2 = 12
\]

We get one spurious hit in this search:

\[
5 4 1 4 2 1 3 5 6 2 \underline{1} 4 \underline{4} \underline{1} 4
\]

\[
t'_{10} = 11
\]
Finite automata

Recall that a finite automaton $M$ is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

- $Q$ is a finite set of states
- $q_0 \in Q$ is the start state
- $A \subseteq Q$ is a distinguished set of accepting states
- $\Sigma$ is a finite input alphabet
- $\delta : Q \times \Sigma \to Q$ is a function called the transition function

Initially the finite automaton is in the start state $q_0$. It reads characters from the input string $x$ one at a time, and changes states according to the transition function. Whenever the current state is in $A$, the set of accepting states, we say that $M$ has accepted the string read so far.

A finite automaton

Consider the following 4-state automaton:

- $Q = \{q_0, q_1, q_2, q_3\}$
- $A = \{q_3\}$
- $\Sigma = \{0, 1\}$
- $\delta$ is given by the following table

<table>
<thead>
<tr>
<th>$q$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_0$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_0$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_0$</td>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_0$</td>
<td>$q_0$</td>
</tr>
</tbody>
</table>

A string matching automaton

We shall devise a string matching automaton such that $M$ accepts any string that ends with the pattern $P$. Then we can run the text $T$ through the automaton, recording every time the machine enters an accepting state, thereby determining every occurrence of $P$ within $T$.

To see how we should devise the string matching automaton, let us consider the naive algorithm at some stage of its operation, when trying to find the pattern $\text{ababaa}$.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

Suppose we are maintaining a counter indicating how many pattern characters have matched so far — this shift $s$ fails at the 6th character. Although the naive algorithm would suggest trying the shift $s + 1$ we should really try the shift $s + 3$ next.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

Skipping invalid shifts

The reason that we can immediately eliminate the shift $s + 1$ is that we have already examined the following characters (while trying the shift $s$)

| a | b | b | a | b | a | b | a | a |

and it is immediate that the pattern does not start like this, and hence this shift is invalid.

To determine the smallest shift that is consistent with the characters examined so far we need to know the answer to the question:

“What is the longest suffix of this string that is also a prefix of the pattern $P$?”

In this instance we see that the last 3 characters of this string match the first 3 of the pattern, so the next feasible shift is $s + 6 - 3 = s + 3$.
The states

For a pattern $P$ of length $m$ we devise a string matching automaton as follows:

The states will be

$$Q = \{0, 1, \ldots, m\}$$

where the state $i$ corresponds to $P_i$, the leading substring of $P$ of length $i$.

The start state $q_0 = 0$ and the only accepting state is $m$.

This is only a partially specified automaton, but it is clear that it will accept the pattern $P$.

We will specify the remainder of the automaton so that it is in state $i$ if the last $i$ characters read match the first $i$ characters of the pattern.

The transition function

Now suppose, for example, that the automaton is given the string

$$a \ b \ b \ a \ b \ b \ 
\ldots$$

The first five characters match the pattern, so the automaton moves from state 0, to 1, to 2, to 3, to 4 and then 5. After receiving the sixth character $b$ which does not match the pattern, what state should the automaton enter?

As we observed earlier, the longest suffix of this string that is a prefix of the pattern $abbabaa$ has length 3, so we should move to state 3, indicating that only the last 3 characters read match the beginning of the pattern.

The entire automaton

We can express this more formally:

If the machine is in state $q$ and receives a character $c$, then the next state should be $q'$ where $q'$ is the largest number such that $P_{q'}$ is a suffix of $P_qc$.

Applying this rule we get the following finite state automaton to match the string $abbabaa$.

By convention here all the horizontal edges are pointing to the right, while all the curved line segments are pointing to the left.

Using the automaton

The automaton has the following transition function:

<table>
<thead>
<tr>
<th>$q$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Use it on the following string

$$ab\bar{a}b\bar{a}b\bar{a}ab\bar{a}\bar{a}a\bar{a}b\bar{a}b\bar{a}ab\bar{a}b\bar{a}b\bar{a}b\bar{a}b\bar{a}a$$

Compressing this information:

<table>
<thead>
<tr>
<th>Character</th>
<th>$a$</th>
<th>$b$</th>
<th>$a$</th>
<th>$b$</th>
<th>$b$</th>
<th>$a$</th>
<th>$b$</th>
<th>$b$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old state</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>New state</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Compressing this information:

$$ab\bar{a}b\bar{a}b\bar{a}b\bar{a}ab\bar{a}b\bar{a}a\bar{a}b\bar{a}b\bar{a}ab\bar{a}b\bar{a}b\bar{a}a$$

121230123456723456721112123456723456341
Analysis and implementation

Given a pattern $P$ we must first compute the transition function. Once this is computed the time taken to find all occurrences of the pattern in a text of length $n$ is just $O(n)$ — each character is examined precisely once, and no “backing-up” in the text is required. This makes it particularly suitable when the text must be read in from disk or tape and cannot be totally stored in an array.

The time taken to compute the transition function depends on the size of the alphabet, but can be reduced to $O(m|\Sigma|)$, by a clever implementation.

Therefore the total time taken by the program is $O(n + m|\Sigma|)$

Recommended reading: CLRS, Chapter 32, pages 906–922

Knuth-Morris-Pratt

The Knuth-Morris-Pratt algorithm is a variation on the string matching automaton that works in a very similar fashion, but eliminates the need to compute the entire transition function.

In the string matching automaton, for any state the transition function gives $|\Sigma|$ possible destinations—one for each of the $|\Sigma|$ possible characters that may be read next.

The KMP algorithm replaces this by just two possible destinations — depending only on whether the next character matches the pattern or does not match the pattern.

As we already know that the action for a matching character is to move from state $q$ to $q + 1$, we only need to store the state changes required for a non-matching character. This takes just one array of length $m$, and we shall see that it can be computed in time $O(m)$.

Regular expressions

The prefix function

Let us return to our example where we are matching the pattern $ababaa$.

Suppose as before that we are matching this against some text and that we detect a mismatch on the sixth character.

```
|   | a | b | b | a | b | x | y | z
|---|---|---|---|---|---|---|---|---
```

In the string-matching automaton we used information about what the actual value of $x$ was, and moved to the appropriate state.

In KMP we do exactly the same thing except that we do not use the information about the value of $x$ — except that it does not match the pattern. So in this case we simply consider how far along the pattern we could be after reading $abab$ — in this case if we are not at position 5 the next best option is that we are at position 2.
The KMP algorithm

The prefix function $\pi$ then depends entirely on the pattern and is defined as follows: $\pi(q)$ is the largest $k < q$ such that $P_k$ is a suffix of $P_q$.

The KMP algorithm then proceeds simply:

$q \leftarrow 0$
for $i$ from 1 to $n$ do
  while $q > 0$ and $T[i] \neq P[q + 1]$
    $q \leftarrow \pi(q)$
  end while
  if $P[q + 1] = T[i]$ then
    $q \leftarrow q + 1$
  end if
if $q = m$ then
  output "shift of $i - m$ is valid"
  $q \leftarrow \pi(q)$
end if
end for

This algorithm has nested loops. Why is it linear rather than quadratic?

Heuristics

Although the KMP algorithm is asymptotically linear, and hence best possible, there are certain heuristics which in some commonly occurring cases allow us to do better.

These heuristics are particularly effective when the alphabet is quite large and the pattern quite long, because they enable us to avoid even looking at many text characters.

The two heuristics are called the bad character heuristic and the good suffix heuristic.

The algorithm that incorporates these two independent heuristics is called the Boyer-Moore algorithm.

The algorithm without the heuristics

The algorithm before the heuristics are applied is simply a version of the naive algorithm, in which each possible shift $s = 0, 1, \ldots$ is tried in turn.

However when testing a given shift, the characters in the pattern and text are compared from right to left. If all the characters match then we have found a valid shift.

If a mismatch is found, then the shift $s$ is not valid, and we try the next possible shift by setting

$s \leftarrow s + 1$

and starting the testing loop again.

The two heuristics both operate by providing a number other than 1 by which the current shift can be incremented without missing any matches.

Bad characters

Consider the following situation:

once we noticed that imbalance

The two last characters ce match the text but the i in the text is a bad character.

Now as soon as we detect the bad character i we know immediately that the next shift must be at least 6 places or the i will simply not match.

Notice that advancing the shift by 6 places means that 6 text characters are not examined at all.
The bad character heuristic

The bad character heuristic involves precomputing a function

$$\lambda : \Sigma \rightarrow \{0, 1, \ldots, m\}$$

such that for a character $c$, $\lambda(c)$ is the right-most position in $P$ where $c$ occurs (and 0 if $c$ does not occur in $P$).

Then if a mismatch is detected when scanning position $j$ of the pattern (remember we are going from right-to-left so $j$ goes from $m$ to $1$), the bad character heuristic proposes advancing the shift by the equation:

$$s \leftarrow s + (j - \lambda(T[s + j]))$$

Notice that the bad-character heuristic might occasionally propose altering the shift to the left, so it cannot be used alone.

Good suffixes

Consider the following situation:

```
the late edition of edited
```

The characters of the text that do match with the pattern are called the good suffix. In this case the good suffix is `ed`. Any shift of the pattern cannot be valid unless it matches at least the good suffix that we have already found. In this case we must move the pattern at least 4 spaces in order that the `ed` at the beginning of the pattern matches the good suffix.

The good-suffix heuristic

The good-suffix heuristic involves precomputing a function

$$\gamma : \{1, \ldots, m\} \rightarrow \{1, \ldots, m\}$$

where $\gamma(j)$ is the smallest positive shift of $P$ so that it matches with all the characters in $P[j + 1..m]$ that still overlaps.

We notice that this condition can always be vacuously satisfied by taking $\gamma(j)$ to be $m$, and hence $\gamma(j) > 0$.

Therefore if a mismatch is detected at character $j$ in the pattern, the good-suffix heuristic proposes advancing the shift by

$$s \leftarrow s + \gamma(j)$$

The Boyer-Moore algorithm

The Boyer-Moore algorithm simply involves taking the larger of the two advances in the shift proposed by the two heuristics.

Therefore, if a mismatch is detected at character $j$ of the pattern when examining shift $s$, we advance the shift according to:

$$s \leftarrow s + \max(\gamma(j), j - \lambda(T[s + j]))$$

The time taken to precompute the two functions $\gamma$ and $\lambda$ can be shown to be $O(m)$ and $O(|\Sigma| + m)$ respectively.

Like the naive algorithm the worst case is when the pattern matches every time, and in this case it will take just as much time as the naive algorithm. However this is rarely the case and in practice the Boyer-Moore algorithm performs well.
Example

Consider the pattern:

\[ \text{one}_\text{shone}_\text{the}_\text{one}_\text{phone} \]

What is the last occurrence function \( \lambda \)?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
</tr>
<tr>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
</tr>
<tr>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
</tr>
<tr>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c ( \lambda(c) )</th>
<th>c ( \lambda(c) )</th>
<th>c ( \lambda(c) )</th>
<th>c ( \lambda(c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ( \lambda(a) )</td>
<td>0 ( \lambda(0) )</td>
<td>h ( \lambda(h) )</td>
<td>20 ( \lambda(20) )</td>
</tr>
<tr>
<td>b ( \lambda(b) )</td>
<td>0 ( \lambda(0) )</td>
<td>i ( \lambda(i) )</td>
<td>0 ( \lambda(0) )</td>
</tr>
<tr>
<td>c ( \lambda(c) )</td>
<td>0 ( \lambda(0) )</td>
<td>j ( \lambda(j) )</td>
<td>0 ( \lambda(0) )</td>
</tr>
<tr>
<td>d ( \lambda(d) )</td>
<td>0 ( \lambda(0) )</td>
<td>k ( \lambda(k) )</td>
<td>0 ( \lambda(0) )</td>
</tr>
<tr>
<td>e ( \lambda(e) )</td>
<td>23 ( \lambda(23) )</td>
<td>l ( \lambda(l) )</td>
<td>0 ( \lambda(0) )</td>
</tr>
<tr>
<td>f ( \lambda(f) )</td>
<td>0 ( \lambda(0) )</td>
<td>m ( \lambda(m) )</td>
<td>0 ( \lambda(0) )</td>
</tr>
<tr>
<td>g ( \lambda(g) )</td>
<td>0 ( \lambda(0) )</td>
<td>n ( \lambda(n) )</td>
<td>22 ( \lambda(22) )</td>
</tr>
</tbody>
</table>

Example continued

What is \( \gamma(22) \)? This is the smallest shift of \( P \) that will match the 1 character \( P[23] \), and this is 6.

\[ \text{one}_\text{shone}_\text{the}_\text{one}_\text{phone} \]

One shift that matches \( P\[22..23\] \) is 6.

\[ \text{one}_\text{shone}_\text{the}_\text{one}_\text{phone} \]

So \( \gamma(21) = 6 \).

The smallest shift that matches \( P\[21..23\] \) is also 6.

\[ \text{one}_\text{shone}_\text{the}_\text{one}_\text{phone} \]

So \( \gamma(20) = 6 \).

However the smallest shift that matches \( P\[20..23\] \) is 14.

\[ \text{one}_\text{shone}_\text{the}_\text{one}_\text{phone} \]

So \( \gamma(19) = 14 \).

What about \( \gamma(18) \)? What is the smallest shift that can match the characters \( \text{phone} \)? A shift of 20 will match all those that are still left.

\[ \text{one}_\text{shone}_\text{the}_\text{one}_\text{phone} \]

This then shows us that \( \gamma(j) = 20 \) for all \( j \leq 18 \), so

\[
\gamma(j) = \begin{cases} 
6 & 20 \leq j \leq 22 \\
14 & j = 19 \\
20 & 1 \leq j \leq 18 
\end{cases}
\]

Longest Common Subsequence

Consider the following problem

**LONGEST COMMON SUBSEQUENCE**

Instance: Two sequences \( X \) and \( Y \)

Question: What is a longest common subsequence of \( X \) and \( Y \)

Example

If \( X = \langle A, B, C, B, D, A, B \rangle \)

and \( Y = \langle B, D, C, A, B, A \rangle \)

then a longest common subsequence is either \( \langle B, C, B, A \rangle \) or \( \langle B, D, A, B \rangle \)
A recursive relationship

As is usual for dynamic programming problems we start by finding an appropriate recursion, whereby the problem can be solved by solving smaller subproblems.

Suppose that

\[ X = \langle x_1, x_2, \ldots, x_m \rangle \]
\[ Y = \langle y_1, y_2, \ldots, y_n \rangle \]

and that they have a longest common subsequence

\[ Z = \langle z_1, z_2, \ldots, z_k \rangle \]

If \( x_m = y_n \) then \( z_k = x_m = y_n \) and \( Z_{k-1} \) is a LCS of \( X_{m-1} \) and \( Y_{n-1} \).

Otherwise \( Z \) is either a LCS of \( X_{m-1} \) and \( Y \) or a LCS of \( X \) and \( Y_{n-1} \).

(This depends on whether \( z_k \neq x_m \) or \( z_k \neq y_n \) respectively — at least one of these two possibilities must arise.)

A recursive solution

This can easily be turned into a recursive algorithm as follows.

Given the two sequences \( X \) and \( Y \) we find the LCS \( Z \) as follows:

If \( x_m = y_n \) then find the LCS \( Z' \) of \( X_{m-1} \) and \( Y_{n-1} \) and set \( Z = Z'x_m \).

If \( x_m \neq y_n \) then find the LCS \( Z_1 \) of \( X_{m-1} \) and \( Y \), and the LCS \( Z_2 \) of \( X \) and \( Y_{n-1} \), and set \( Z \) to be the longer of these two.

It is easy to see that this algorithm requires the computation of the LCS of \( X_i \) and \( Y_j \) for all values of \( i \) and \( j \). We will let \( l(i, j) \) denote the length of the longest common subsequence of \( X_i \) and \( Y_j \).

Then we have the following relationship on the lengths

\[
l(i, j) = \begin{cases} 
0 & \text{if } ij = 0 \\
l(i - 1, j - 1) + 1 & \text{if } x_i = y_j \\
\max(l(i - 1, j), l(i, j - 1)) & \text{if } x_i \neq y_j
\end{cases}
\]

Memoization

The simplest way to turn a top-down recursive algorithm into a sort of dynamic programming routine is memoization. The idea behind this is that the return values of the function calls are simply stored in an array as they are computed.

The function is changed so that its first step is to look up the table and see whether \( l(i, j) \) is already known. If so, then it just returns the value immediately, otherwise it computes the value in the normal way.

Alternatively, we can simply accept that we must at some stage compute all the \( O(n^2) \) values \( l(i, j) \) and try to schedule these computations as efficiently as possible, using a dynamic programming table.

The dynamic programming table

We have the choice of memoizing the above algorithm or constructing a bottom-up dynamic programming table.

In this case our table will be an \((m + 1) \times (n + 1)\) table where the \((i, j)\) entry is the length of the LCS of \( X_i \) and \( Y_j \).

Therefore we already know the border entries of this table, and we want to know the value of \( l(m, n) \) being the length of the LCS of the original two sequences.

In addition to this however we will retain some additional information in the table - namely each entry will contain either a left-pointing arrow \( \leftarrow \), a upward-pointing arrow \( \uparrow \) or a diagonal arrow \( \downarrow \).

These arrows will tell us which of the subcases was responsible for the entry getting that value.
Our example

For our worked example we will use the sequences

\[ X = \langle 0, 1, 1, 0, 1, 0, 0, 1 \rangle \]

and

\[ Y = \langle 1, 1, 0, 1, 1, 0 \rangle \]

Then our initial empty table is:

\[
\begin{array}{ccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & x_0 & & & & & & \\
1 & x_0 & & & & & & \\
2 & y_2 & & & & & & \\
3 & y_1 & & & & & & \\
4 & y_0 & & & & & & \\
5 & y_0 & & & & & & \\
6 & y_0 & & & & & & \\
7 & y_0 & & & & & & \\
8 & y_0 & & & & & & \\
\end{array}
\]

The first table

First we fill in the border of the table with the zeros.

\[
\begin{array}{ccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & x_0 & & & & & & \\
1 & x_0 & & & & & & \\
2 & y_2 & & & & & & \\
3 & y_1 & & & & & & \\
4 & y_0 & & & & & & \\
5 & y_0 & & & & & & \\
6 & y_0 & & & & & & \\
7 & y_0 & & & & & & \\
8 & y_0 & & & & & & \\
\end{array}
\]

Now each entry \((i, j)\) depends on \(x_i, y_j\) and the values to the left \((i, j - 1)\), above \((i - 1, j)\), and above-left \((i - 1, j - 1)\).

In particular, we proceed as follows:

If \(x_i = y_j\) then put the symbol \(\downarrow\) in the square, together with the value \(l(i - 1, j - 1) + 1\).

Otherwise put the greater of the values \(l(i - 1, j)\) and \(l(i, j - 1)\) into the square with the appropriate arrow.

The first row

It is easy to compute the first row, starting in the \((1, 1)\) position:

\[
\begin{array}{ccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & x_0 & & & & & & \\
1 & x_0 & & & & & & \\
2 & y_2 & & & & & & \\
3 & y_1 & & & & & & \\
4 & y_0 & & & & & & \\
5 & y_0 & & & & & & \\
6 & y_0 & & & & & & \\
7 & y_0 & & & & & & \\
8 & y_0 & & & & & & \\
\end{array}
\]

Computation proceeds as described above.

The final array

After filling it in row by row we eventually reach the final array:

\[
\begin{array}{ccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & x_0 & & & & & & \\
1 & x_0 & & & & & & \\
2 & y_2 & & & & & & \\
3 & y_1 & & & & & & \\
4 & y_0 & & & & & & \\
5 & y_0 & & & & & & \\
6 & y_0 & & & & & & \\
7 & y_0 & & & & & & \\
8 & y_0 & & & & & & \\
\end{array}
\]

This then tells us that the LCS of \(X = X_{8}\) and \(Y = Y_{6}\) has length 5 — because the entry \(l(8, 6) = 5\).

This time we have kept enough information, via the arrows, for us to compute what the LCS of \(X\) and \(Y\) is.
Finding the LCS

The LCS can be found (in reverse) by tracing the path of the arrows from \((l(m, n))\). Each diagonal arrow encountered gives us another element of the LCS.

As \(l(8, 6)\) points to \(l(7, 6)\) so we know that the LCS is the LCS of \(X_7\) and \(Y_6\).

Now \(l(7, 6)\) has a diagonal arrow, pointing to \(l(6, 5)\) so in this case we have found the last entry of the LCS — namely it is \(x_7 = y_6 = 0\).

Then \(l(6, 5)\) points (upwards) to \(l(5, 5)\), which points diagonally to \(l(4, 4)\) and hence 1 is the second-last entry of the LCS.

Proceeding in this way, we find that the LCS is 11010

Notice that if at the very final stage of the algorithm (where we had a free choice) we had chosen to make \(l(8, 6)\) point to \(l(8, 5)\) we would have found a different LCS 11011

Analysis

The analysis for longest common subsequence is particularly easy.

After initialization we simply fill in \(mn\) entries in the table — with each entry costing only a constant number of comparisons. Therefore the cost to produce the table is \(\Theta(mn)\)

Following the trail back to actually find the LCS takes time at most \(O(m + n)\) and therefore the total time taken is \(\Theta(mn)\).

Data Compression Algorithms

Data compression algorithms exploit patterns in data files to compress the files. Every compression algorithm should have a corresponding decompression algorithm that can recover (most of) the original data.

Data compression algorithms are used by programs such as WinZip, pkzip and zip. They are also used in the definition of many data formats such as pdf, jpeg, mpeg and .doc.

Data compression algorithms can either be lossless (e.g. for archiving purposes) or lossy (e.g. for media files).

We will consider some lossless algorithms below.
Huffman coding

A nice application of a greedy algorithm is found in an approach to data compression called Huffman coding.

Suppose that we have a large amount of text that we wish to store on a computer disk in an efficient way. The simplest way to do this is simply to assign a binary code to each character, and then store the binary codes consecutively in the computer memory.

The ASCII system for example, uses a fixed 8-bit code to represent each character. Storing \( n \) characters as ASCII text requires 8\( n \) bits of memory.

Simplification

Let \( C \) be the set of characters we are working with. To simplify things, let us suppose that we are storing only the 10 numeric characters 0, 1, ..., 9. That is, set \( C = \{0, 1, \ldots, 9\} \).

A fixed length code to store these 10 characters would require at least 4 bits per character. For example we might use a code like this:

<table>
<thead>
<tr>
<th>Char</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
</tbody>
</table>

However in any non-random piece of text, some characters occur far more frequently than others, and hence it is possible to save space by using a variable length code where the more frequently occurring characters are given shorter codes.

Non-random data

Consider the following data, which is taken from a Postscript file.

<table>
<thead>
<tr>
<th>Char</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1294</td>
</tr>
<tr>
<td>9</td>
<td>1525</td>
</tr>
<tr>
<td>6</td>
<td>2260</td>
</tr>
<tr>
<td>4</td>
<td>2561</td>
</tr>
<tr>
<td>2</td>
<td>4442</td>
</tr>
<tr>
<td>3</td>
<td>5960</td>
</tr>
<tr>
<td>7</td>
<td>6878</td>
</tr>
<tr>
<td>8</td>
<td>8865</td>
</tr>
<tr>
<td>1</td>
<td>11610</td>
</tr>
<tr>
<td>0</td>
<td>70784</td>
</tr>
</tbody>
</table>

Notice that there are many more occurrences of 0 and 1 than the other characters.

A good code

What would happen if we used the following code to store the data rather than the fixed length code?

<table>
<thead>
<tr>
<th>Char</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
</tr>
<tr>
<td>2</td>
<td>01111</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>00101</td>
</tr>
<tr>
<td>5</td>
<td>01100</td>
</tr>
<tr>
<td>6</td>
<td>00100</td>
</tr>
<tr>
<td>7</td>
<td>01110</td>
</tr>
<tr>
<td>8</td>
<td>000</td>
</tr>
<tr>
<td>9</td>
<td>011101</td>
</tr>
</tbody>
</table>

To store the string 0748901 we would get

00000111010100100010010000000001
using the fixed length code and

10111000101000011110110110101101010
using the variable length code.
Prefix codes

In order to be able to decode the variable length code properly it is necessary that it be a prefix code — that is, a code in which no codeword is a prefix of any other codeword.

Decoding such a code is done using a binary tree.

Cost of a tree

Now assign to each leaf of the tree a value, \( f(c) \), which is the frequency of occurrence of the character \( c \) represented by the leaf.

Let \( d_T(c) \) be the depth of character \( c \)'s leaf in the tree \( T \).

Then the number of bits required to encode a file is

\[
B(T) = \sum_{c \in C} f(c) d_T(c)
\]

which we define as the cost of the tree \( T \).

Optimal trees

A tree representing an optimal code for a file is always a full binary tree — namely, one where every node is either a leaf or has precisely two children.

Therefore if we are dealing with an alphabet of \( s \) symbols we can be sure that our tree has precisely \( s \) leaves and \( s - 1 \) internal nodes, each with two children.

Huffman invented a greedy algorithm to construct such an optimal tree.

The resulting code is called a Huffman code for that file.
Huffman’s algorithm

The algorithm starts by creating a forest of single nodes, each representing one character, and each with an associated value, being the frequency of occurrence of that character. These values are placed into a priority queue (implemented as a linear array).

\[
\begin{align*}
5: & 1294 \\
9: & 1525 \\
6: & 2260 \\
4: & 2561 \\
2: & 4442 \\
3: & 5960 \\
7: & 6878 \\
8: & 8865 \\
1: & 11610 \\
0: & 70784
\end{align*}
\]

Then repeat the following procedure \( s - 1 \) times:

Remove from the priority queue the two nodes \( L \) and \( R \) with the lowest values, and create an internal node of the binary tree whose left child is \( L \) and right child \( R \).

Compute the value of the new node as the sum of the values of \( L \) and \( R \) and insert this into the priority queue.

The first few steps

Given the data above, the first two entries off the priority queue are 5 and 9 so we create a new node

\[
\begin{align*}
2819 & \quad 5:1294 \\
9:1525 & \quad 2:4442 \\
4:2561 & \quad 6:2260 \\
2819 & \quad 2:4442
\end{align*}
\]

The priority queue is now one element shorter, as shown below:

\[
\begin{align*}
6: & 2260 \\
4: & 2561 \\
2819 & \quad 2:4442 \\
\cdot \cdot \cdot \\
5:1294 & \quad 9:1525
\end{align*}
\]

The next two are 6 and 4 yielding

\[
\begin{align*}
2819 & \quad 2:4442 \\
4821 & \quad 6:2260 \\
2819 & \quad 4821 \\
\cdot \cdot \cdot \\
5:1294 & \quad 9:1525
\end{align*}
\]

Why does it work?

In order to show that Huffman’s algorithm works, we must show that there can be no prefix codes that are better than the one produced by Huffman’s algorithm.

The proof is divided into two steps:

First it is necessary to demonstrate that the first step (merging the two lowest frequency characters) cannot cause the tree to be non-optimal. This is done by showing that any optimal tree can be reorganised so that these two characters have the same parent node. (see CLRS, Lemma 16.2, page 388)

Secondly we note that after making an optimal first choice, the problem can be reduced to finding a Huffman code for a smaller alphabet. (see CLRS, Lemma 16.3, page 391)

See CLRS (page 388) for the pseudo-code corresponding to this algorithm.
Adaptive Huffman Coding

Huffman coding requires that we have accurate estimates of the probabilities of each character occurring.

In general, we can make estimates of the frequencies of characters occurring in English text, but these estimates are not useful when we consider other data formats.

Adaptive Huffman coding calculates character frequencies on the fly and uses these dynamic frequencies to encode characters. This technique can be applied to binary files as well as text files.

Algorithms: Adaptive Huffman Coding

The Adaptive Huffman Coding algorithms seek to create a Huffman tree on the fly. A Huffman Coding allows us to encode frequently occurring characters in a lesser number of bits than rarely occurring characters. Adaptive Huffman Coding determines the Huffman Tree only from the frequencies of the characters already read.

Recall that prefix codes are defined using a binary tree. It can be shown that a prefix code is optimal if and only if the binary tree has the sibling property.

A binary tree recording the frequency of characters has the sibling property iff

1. every node except the root has a sibling.
2. each right-hand sibling (including non-leaf nodes) has at least as high a frequency as its left-hand sibling

(The frequency of non-leaf nodes ins the sum of the frequency of it's children).

As characters are read it is possible to efficiently update the frequencies, and modify the binary tree so that the sibling property is preserved. It is also possible to do this in a deterministic way so that a similar process can decompress the code.

See http://www.cs.duke.edu/~jsv/Papers/Vit87.jacmACMv for more details.

As opposed to the LZ algorithms that follow, Huffman methods only encode one character at a time. However, best performance often comes from combining compression algorithms (for example, gzip combines LZ77 and Adaptive Huffman Coding).
Ziv-Lempel compression algorithms

The Ziv-Lempel compression algorithms are a family of compression algorithms that can be applied to arbitrary file types.

The Ziv-Lempel algorithms represent recurring strings with abbreviated codes. There are two main types:

- **LZ77** variants use a buffer to look for recurring strings in a small section of the file.

- **LZW** variants dynamically create a dictionary of recurring strings, and assigns a simple code to each such string.

### Algorithms: LZ77

The **LZ77** algorithms use a sliding window. The sliding window is a buffer consisting of the last \( m \) letters encoded \( (a_0...a_{m-1}) \) and the next \( n \) letters to be encoded \( (b_0...b_{n-1}) \).

Initially we let \( a_0 = a_1 = ... = a_{n-1} = w_0 \) and output \( \langle 0, 0, a \rangle \) where \( w_0 \) is the first letter of the word to be compressed.

The algorithm looks for the longest prefix of \( b_0...b_{n-1} \) appearing in \( a_0...a_{m-1} \). If the longest prefix found is \( b_0...b_{k-1} = a_i...a_{i+k-1} \), then the entire prefix is encoded as the tuple \( \langle i, k, b_k \rangle \) where \( i \) is the offset, \( k \) is the length and \( b_k \) is the next character.

### LZ77 Example

Suppose that \( m = n = 4 \) and we would like to compress the word \( w = aabacbaa \)

<table>
<thead>
<tr>
<th>Word</th>
<th>Window</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>aabacbaa</td>
<td>aaaa aaba</td>
<td>( \langle 0, 2, b \rangle )</td>
</tr>
<tr>
<td>aabacbaa</td>
<td>aaab a abac</td>
<td>( \langle 2, 3, c \rangle )</td>
</tr>
<tr>
<td>baa</td>
<td>abac baa</td>
<td>( \langle 1, 2, a \rangle )</td>
</tr>
</tbody>
</table>

This outputs \( \langle 0, 0, a \rangle \langle 0, 2, b \rangle \langle 2, 3, c \rangle \langle 1, 2, a \rangle \)

### LZ77 Example cont.

To decompress the code we can reconstruct the sliding window at each step of the algorithm. Eg, given

<table>
<thead>
<tr>
<th>Input</th>
<th>Window</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle 0, 0, a \rangle )</td>
<td></td>
<td>( \langle 0, 0, a \rangle )</td>
</tr>
<tr>
<td>( \langle 0, 2, b \rangle )</td>
<td>aaaa aab?</td>
<td>aab</td>
</tr>
<tr>
<td>( \langle 2, 3, c \rangle )</td>
<td>aaab a abac</td>
<td>abac</td>
</tr>
<tr>
<td>( \langle 1, 2, a \rangle )</td>
<td>abac baa?</td>
<td>baa</td>
</tr>
</tbody>
</table>

Note the trick with the third triple \( \langle 2, 3, c \rangle \) that allows the look-back buffer to overflow into the look ahead buffer. See [http://en.wikipedia.org/wiki/LZ77_and_LZ78](http://en.wikipedia.org/wiki/LZ77_and_LZ78) for more information.
Algorithms: LZW

The LZW algorithms use a dynamic dictionary. The dictionary maps words to codes and is initially defined for every byte (0-255). The compression algorithm is as follows:

```
w = null
while(k = next byte)
  if wk in the dictionary
    w = wk
  else
    add wk to dictionary
    output code for w
    w = k
output code for w
```

Algorithms: LZW

The decompression algorithm is as follows:

```
k = next byte
output k
w = k
while(k = next byte)
  if there’s no dictionary entry for k
    entry = w + first letter of w
  else
    entry = dictionary entry for k
  output entry
  add w + first letter of entry to dictionary
  w = entry
```

LZW Example

Consider the word \(w = \text{aababa}\), and a dictionary \(D\) where \(D[0] = a\), \(D[1] = b\) and \(D[2] = c\). The compression algorithm proceeds as follows:

<table>
<thead>
<tr>
<th>Read</th>
<th>Do</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(w = a)</td>
<td>–</td>
</tr>
<tr>
<td>(a)</td>
<td>(w = a, D[3] = aa)</td>
<td>0</td>
</tr>
<tr>
<td>(b)</td>
<td>(w = b, D[4] = ab)</td>
<td>0</td>
</tr>
<tr>
<td>(a)</td>
<td>(w = a, D[5] = ba)</td>
<td>1</td>
</tr>
<tr>
<td>(b)</td>
<td>(w = ab)</td>
<td>–</td>
</tr>
<tr>
<td>(a)</td>
<td>(w = a, D[6] = aba)</td>
<td>4</td>
</tr>
<tr>
<td>(c)</td>
<td>(w = c, D[7] = ac)</td>
<td>0</td>
</tr>
<tr>
<td>(b)</td>
<td>(w = b, D[8] = cb)</td>
<td>2</td>
</tr>
<tr>
<td>(a)</td>
<td>(w = ba)</td>
<td>–</td>
</tr>
<tr>
<td>(a)</td>
<td>(w = a, D[9] = baa)</td>
<td>5</td>
</tr>
</tbody>
</table>

LZW Example cont.

To decompress the code \(00140250\) we initialize the dictionary as before. Then

<table>
<thead>
<tr>
<th>Read</th>
<th>Do</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(w = a)</td>
<td>(a)</td>
</tr>
<tr>
<td>0</td>
<td>(w = a, D[3] = aa)</td>
<td>(a)</td>
</tr>
<tr>
<td>1</td>
<td>(w = b, D[4] = ab)</td>
<td>(b)</td>
</tr>
<tr>
<td>4</td>
<td>(w = ab, D[5] = ba)</td>
<td>(ab)</td>
</tr>
<tr>
<td>0</td>
<td>(w = a, D[6] = aba)</td>
<td>(a)</td>
</tr>
<tr>
<td>2</td>
<td>(w = c, D[7] = ac)</td>
<td>(c)</td>
</tr>
<tr>
<td>5</td>
<td>(w = ba, D[8] = cb)</td>
<td>(ba)</td>
</tr>
<tr>
<td>0</td>
<td>(w = a, D[9] = baa)</td>
<td>(a)</td>
</tr>
</tbody>
</table>

See

http://en.wikipedia.org/wiki/LZ77_and_LZ78, also.
1. String matching is the problem of finding all matches for a given pattern, in a given sample of text.

2. The Rabin-Karp algorithm uses prime numbers to find matches in linear time in the expected case.

3. A String matching automata works in linear time, but requires a significant amount of precomputing.

4. The Knuth-Morris-Pratt uses the same principal as a string matching automata, but reduces the amount of precomputation required.

5. The Boyer-Moore algorithm uses the bad character and good suffix heuristics to give the best performance in the expected case.

6. The longest common subsequence problem is can be solved using dynamic programming.

7. Dynamic programming can improve the efficiency of divide and conquer algorithms by storing the results of sub-computations so they can be reused later.

8. Data Compression algorithms use pattern matching to find efficient ways to compress file.

9. Huffman coding uses a greedy approach to recode the alphabet with a more efficient binary code.

10. Adaptive Huffman coding uses the same approach, but with the overhead of precomputing the code.

11. LZ77 uses pattern matching to express segments of the file in terms of recently occurring segments.

12. LZW uses a hash function to store commonly occurring strings so it can refer to them by their key.