ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Text Processing

Goodrich et al. chapter 11

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Introduction

- The String class in C++
- Pattern matching algorithms
  - Brute-force
  - Boyer-Moore algorithm
- Text indexing
  - Tries
  - Compressed Tries
  - Suffix Tries
- Text Similarity Testing
  - Longest common sub-sequence
- Text compression
  - The Huffman coding tree

The string class in C++

- The STL `string` class is actually a specialised form of the template class `basic_string<T>`
  - `string` is just a short way of saying `basic_string<char>`
- You can access an individual character in the string as if it were an array:
  - `mystring[5]` will return the 6th character in the string `mystring[0]` returns the 1st character.
  - But recall the array bounds are not checked
  - if you want array bounds checked use `mystring.at(5)`
- The `string` class has many methods
  - which you can find in the MSDEV help under `basic_string` members

The find() method of the string class

- The **`find()`** method checks to see if a given string is a substring.
  - That is it checks if the *pattern* defined by the substring occurs in the string
- **`find()`** returns the index of the beginning of the first occurrence of a substring (pattern) in a string
- if it does not find the substring it will return `n` (the length of the string).

```
string s = "done it. done by me", q = "one", r = "one
  by", t = "do me";
cout << s.find(q);  // will print 1
cout << s.find(r);  // will print 10
cout << s.find(t);  // will print 19
```

Implementing the methods of the `string` class

- Most of the methods of the string class are directly implemented by representing the string internally as a null-terminated array of characters.
  - For example, the `length()` method of the string class equates to the `strlen(str)` function of C++ where `str` is a null-terminated array of characters (i.e. a c-string)
  - takes a null-terminated array of characters (c-string) and iterates through the array until it finds the NULL character to determine the length of the c-string.
- The `find()` method is more complex
  - It is also a very common task which needs to be done as efficiently as possible
    - For example, searching a text for occurrences of a given phrase
    - Or searching for a certain DNA sub-sequence in a larger sequence
- How might `find()` be implemented internally?
  - We will look here at two different pattern matching algorithms.
    - Brute Force pattern matching
    - The Boyer-Moore algorithm
  - To show how performance can be significantly improved by applying an algorithm for pattern matching

Pattern matching algorithms

- Usage would include tasks like recursively searching files for virus patterns, searching databases for keys or data, text and word processing and any other task that requires handling large amounts of data at very high speed.

Brute Force pattern matching: the algorithm

- We assume we are looking for a substring (pattern) of length p in a string of length s

For each possible starting position of the pattern in the string

start at the first character in the pattern

while the next character in the pattern matches the next character in the string

move on to the next character in the pattern and string

end while

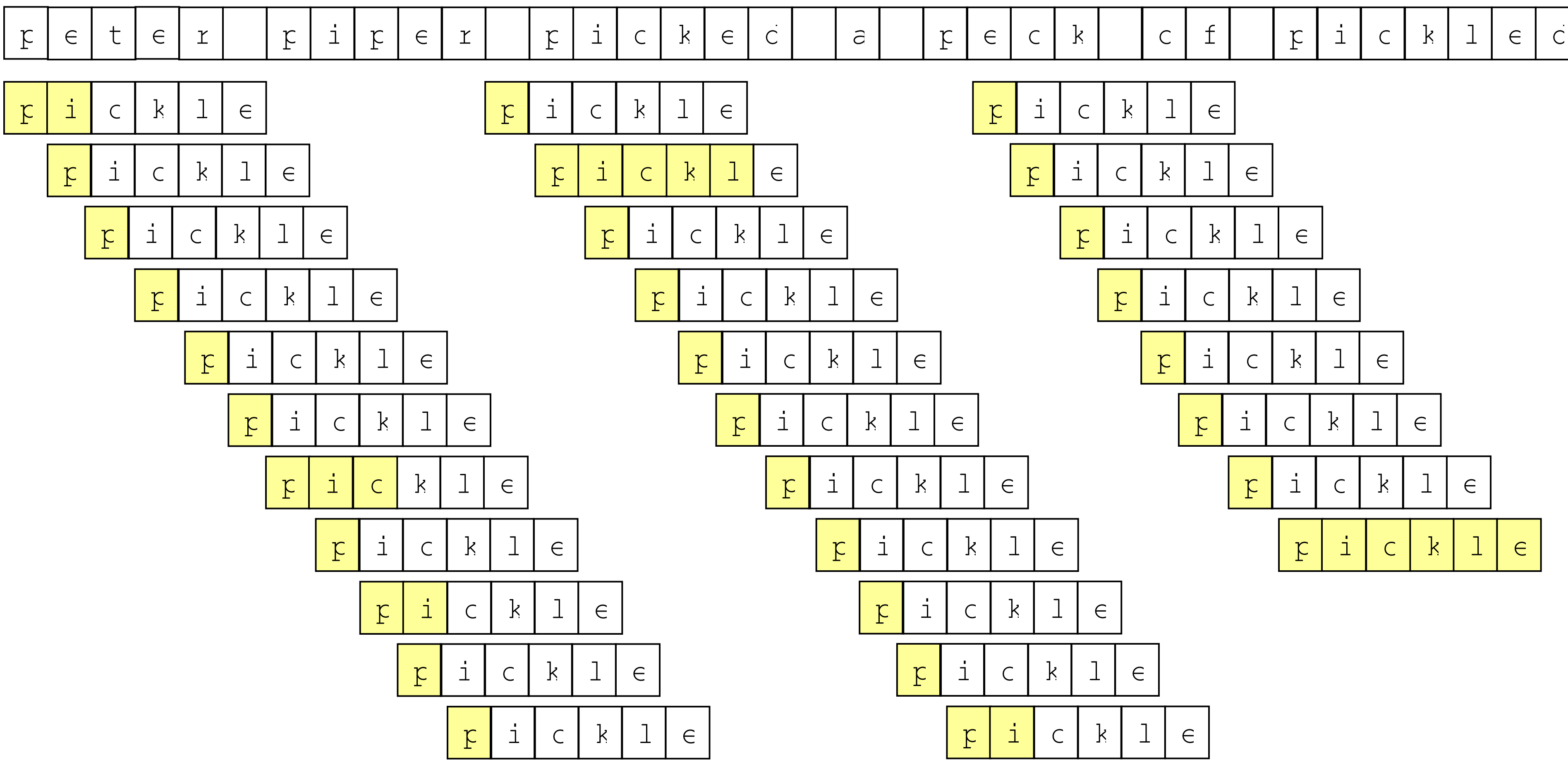
if the whole pattern has been matched

return the starting position of the pattern

End For

Brute Force example

- Text string "peter piper picked a peck of pickled pepper"
- Pattern string "pickle"
- The comparisons are shown coloured yellow
- To find the pattern in the string there are 44 comparisons



ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Brute Force: more formal pseudo code

```
Pattern of length p, string to be searched of length s
For each possible starting position of the pattern in the string
For (i = 0 to s – p) do
     start at the first character in the pattern
    set j = 0
     while the next character in the pattern matches the next
        character in the string
     while j < p and text[i+j] == pattern[j]
          move on to the next character in the pattern and string
           increment j
     End while
     if the whole pattern has been matched
     if j == p then
     Return the starting position of the pattern
           return i
End for
Or return value to indicate no match found
return s
```

Efficiency of the brute force algorithm

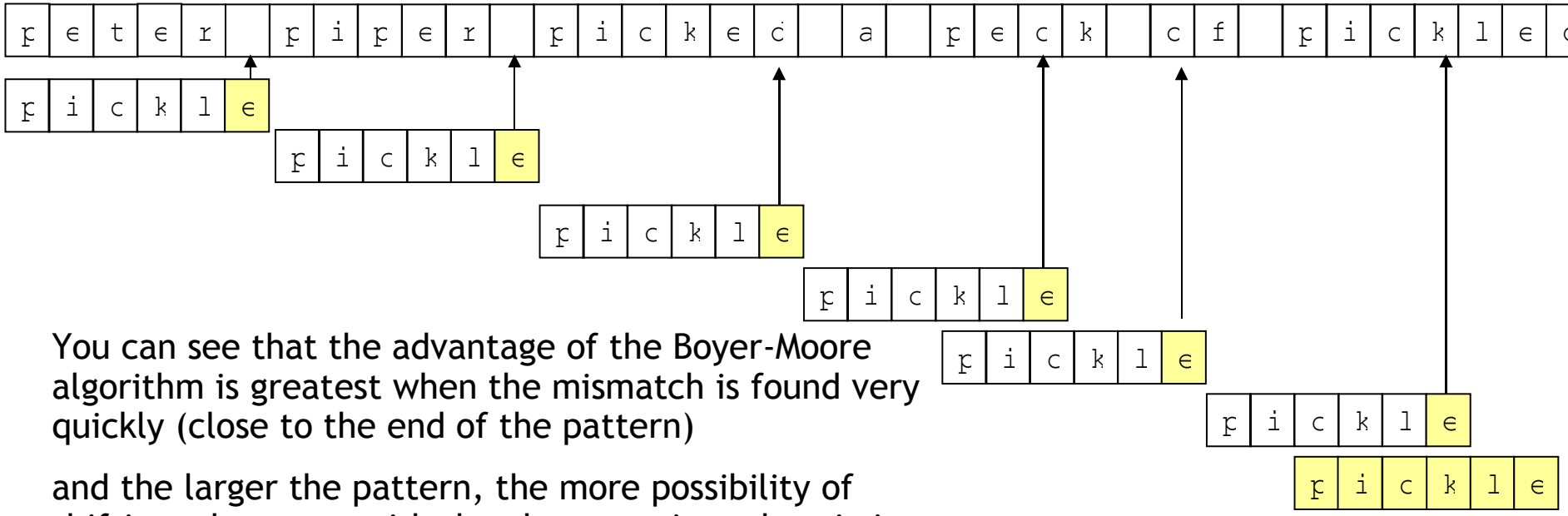
- Looking for string of length `m` in a text of length `n`
- For each candidate index on the text, we have to compare potentially m characters
  - Outer loop executed up to n – m + 1 times
    - thats the for loop in our pseudo code
  - Inner loop makes up to m comparisons
    - That's the while loop in our pseudo code
- Total number of comparisons is **O((n-m+1)m)** (worst case)
  - Can be simplified as **O(mn)** since this is the most significant term
  - And if the length of the pattern m is a constant factor of the whole text n …
  - … such as m = n/2, where the pattern is half the length of the text
  - then the running time is **$O(n^2)$**

Boyer Moore algorithm

- This algorithm depends on having *pre-processed the pattern string*.
- It improves performance, by eliminating large portions of the text at a time.
- It does this by comparing the pattern with the string *from the end of the pattern backwards.*
  - Goodrich calls this the *looking-glass* heuristic
- When a mismatch occurs, a whole portion of the string can be jumped over
  - the *character-jump* heuristic (Goodrich again)
  - the pattern is shifted along the string until a character in the pattern that does match at this position is found.
  - if the character does not exist in the pattern, then the pattern is shifted right past this position
- The amount to move the pattern along can be found quickly if the pattern-string has been indexed so that there is a last-occurrence map for each character.
  - the map stores the index of the last occurrence of this character in the pattern.
  - This is the pre-processing step

Boyer-Moore example

- Using the same example as we used for the brute force pattern match (which took 44 comparisons)
- The number of comparisons has been reduced to 12 !



You can see that the advantage of the Boyer-Moore algorithm is greatest when the mismatch is found very quickly (close to the end of the pattern)

and the larger the pattern, the more possibility of shifting a long way with the character jump heuristic

Actually the worst case analysis is still **O(mn)** for this simplified Boyer-Moore presented, but the worst case would be unlikely to occur for English text

There is an alternative using an alternative character-jump heuristic, which can be shown to be **O(m + n)** even in the worst case.

Text Indexing

- Text indexing is a way of significantly speeding up pattern matching by pre-processing the *text*
  - it can then be searched quickly for many different patterns
  - A good approach if a series of searches (using different pattern strings) is to be performed on a fixed text
  - Recall the Boyer-Moore algorithm pre-processed the *pattern* - good if the same pattern is to be matched against many texts
- The initial cost of pre-processing is compensated for by a speed-up in every query.
  - Useful, for example, for a web-site dedicated to searching the text of Shakespeare's Hamlet, or a search engine offering web pages on the Hamlet topic
  - Or searching for different DNA sequences in a database of genomes
  - What is a genome?? http://www.genomenewsnetwork.org/resources/whats_a_genome/Chp1_1_1.shtml#genome1
- To support such fast pattern matching a tree-based data structure called a T*rie* is used
  - The main application for Tries is information retrieval (trie comes from the word retrieval)
- As well as pattern matching, Tries are also used to support *prefix matching*
  - Prefix matching involves looking for all strings that *start with* a certain pattern

The standard trie

- The Trie stores all the strings that can be found in the text, such that no string is a substring of another.
  - The standard Trie represents the strings with paths from the root to the leaf nodes of the tree.
  - In a standard Trie each node stores 1 character
- Let S be a set of strings from alphabet Σ
  - Each node of the Trie, except the root is labelled with a character from Σ
  - There is exactly one path through the tree to each leaf node
  - Each path through the Trie from root to a leaf node represents a string from the set S

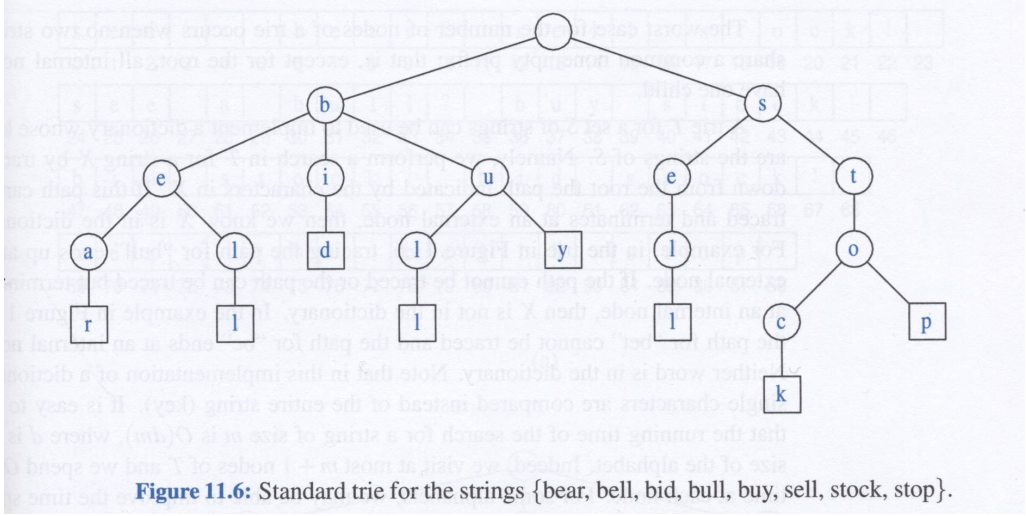


Figure 11.6: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

Diagram: Goodrich et al page 551

Properties of the standard Trie

- Every non-leaf node in the Trie has at most d children, where d is the total number of characters in the 'alphabet' from which the strings are made
  - Because every child stores a different character
- If the number of strings is s, then there are s leaf nodes
  - Because every string ends at a leaf node
- The height of the Trie is the length of the longest string stored
  - Each string follows a path from root to a leaf of the trie
- The number of nodes in the Trie is O(n) where n is the sum of the lengths of all the strings
  - The worst case occurs when no 2 strings share a common non-empty prefix

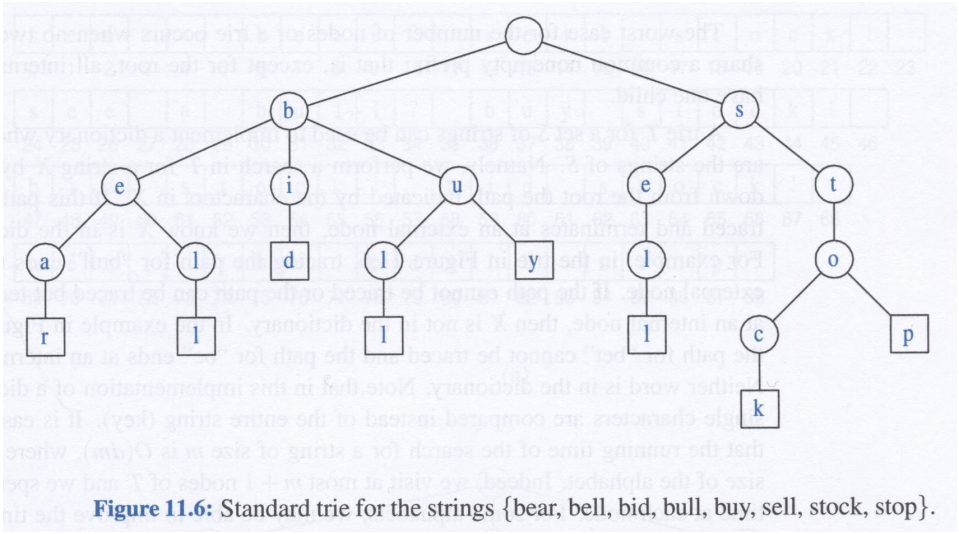


Figure 11.6: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

- How a trie is used to *index* the words in a text
  - the common words such as articles and prepositions (e.g. "the", "a" , "on" etc) are excluded
  - The leaf nodes are augmented by providing the index/indices into the text where the string which ends here can be found

performance of the Trie for word matching

- The trie described can be used for word-matching or prefix-matching
  - Word matching differs from standard matching
  - The pattern cannot match an arbitrary substring of the text, but only one of its words (or a pre-fix of one of its words)
- If the word is of length **`w`**, and there are **`d`** letters in the alphabet from which the text is taken, then the algorithm to determine if the word is in the text is **`O(dw)`**
  - For each of the w characters in the word, we may have to compare d children of the current node to find the next character of the word
  - So once the text is indexed, the number of words in the text is not a factor in the time it takes to determine if any word is in the text
- Of course the no of words in the text will affect the time it takes to index the text (prepare the Trie)
  - But this is only done once

Non-standard Tries – the Compressed Trie

- Similar to a standard trie, but each non-leaf node has at least 2 children.
- This is achieved by replacing each redundant chain of nodes by a compressed node
  - A redundant chain is one where each node has only one child
- The nodes in a compressed trie are labelled with strings rather than individual characters
  - The advantage is that the number of nodes in a compressed trie is proportional to the number of strings, and not to their total length.
  - And the path through the tree to find any particular word is likely to be a lot shorter.

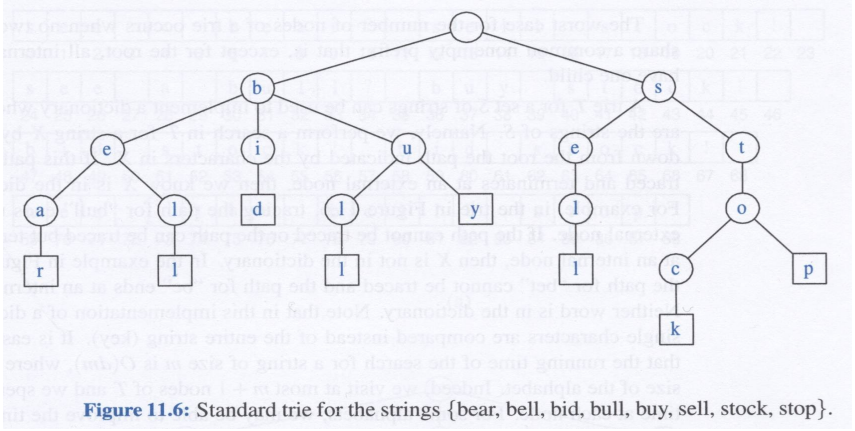


Figure 11.6: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

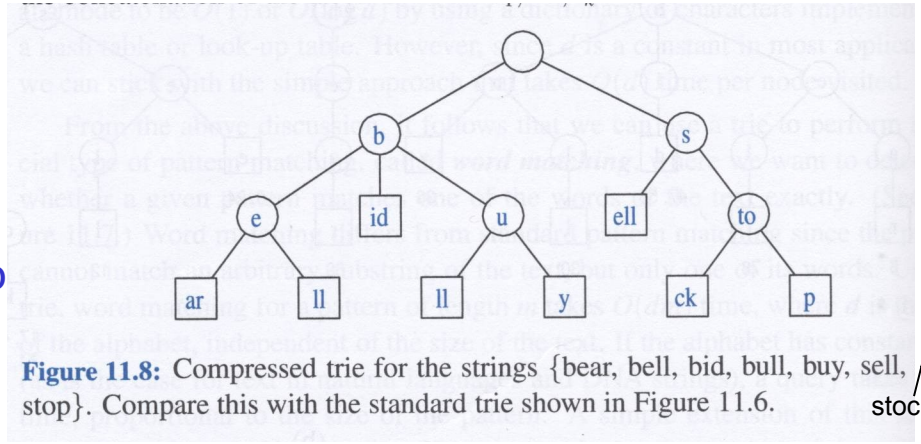


Figure 11.8: Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stop}. Compare this with the standard trie shown in Figure 11.6.

Non-standard Tries – the Suffix Trie

- A suffix trie is used to determine if a particular pattern occurs in the text T
  - A suffix trie is a trie where all the strings in the collection of 'words' stored in the tree are actually the possible suffixes of the same text string T.
  - There are n possible suffixes for a text of length n
  - When used with compression techniques, the space required for a suffix trie for a text of length n is `O(n)`

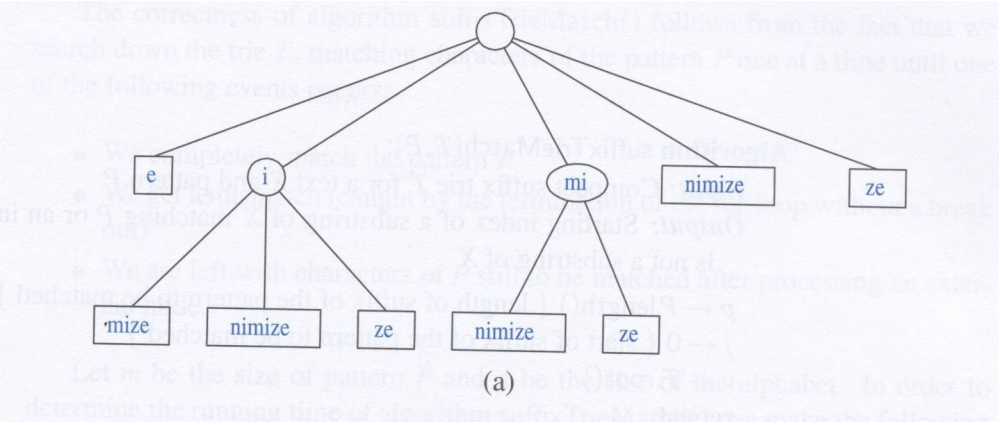


Diagram: Goodrich et al page 559

Example, the suffixes of the word minimize

'e', 'ze', 'ize', 'mize', 'imize',

'nimize', 'inimize', 'minimize'

The suffix tree for minimize is shown opposite

We can see that any substring of 'minimize' can be determined by tracing through this tree

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Text Similarity Testing

- A common requirement is to test *similarity* between 2 strings:
  - to perform 'fuzzy matching' on 2 text strings
  - to identify plagiarism!
  - to find the probability that 2 dna sequences are related
- Simple way to define the similarity between strings is based on large common substrings
  - A brute force mechanism for determining the longest subsequence that 2 strings have in common would take exponential time!
  - There are dynamic programming techniques which can find the longest common sub-sequence of strings of length m and n in `O(mn)` time
- We will not study these techniques now!!

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Huffman Coding

- This is the final text-processing topic we have studied
- See the separate set of overheads on huffman coding