UNIT-6

TEXT SEARCH ALGORITHM

Searching in Compressed Text (Overview)

- What is Text Compression
- Definition The Shannon Bound Huffman Codes The Kolmogorov Measure Searching in Non-adaptive Codes KMP in Huffman Codes
- Searching in Adaptive Codes
- The Lempel-Ziv Codes Pattern Matching in Z-Compressed Files Adapting Compression for Searching

THE TECHNIQUES ARE

Three classical text retrieval techniques have been defined for organizing items in a textual database, for rapidly identifying the relevant items

The techniques are

- Full text scanning(streaming)
- Word inversion
- Multi attribute retrieval
- Text streaming architecture

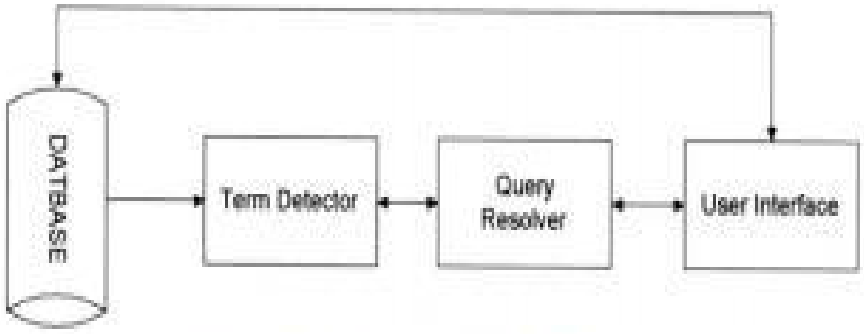


Figure 9.1 Text Streaming Architecture

Software Text Search Algorithms

- In software streaming techniques, the item to be searched is read into memory and then the algorithm is applied.
- There are four major algorithms associated with software text search:
- Brute force approach
- Knuth-Morris-Pratt

- Boyer-Moore
- Shift-OR algorithm
- Rabin -karp

Brute Force :

This approach is the simplest string matching algorithm. The idea is to try and match the search string against the input text. If as soon as a mis-match is detected in the comparison process , shift the input text one position and start the comparison process over.

KNUTH-MORRIS PRATT

Even in the worst case it does not depend upon the length of the text pattern being searched for. The basic concept behind the algorithm is that whenever a mismatch is detected , the previous matched characters define the number of characters that can be skipped in the input stream prior to process again . Starting the comparison

Position :1 2 3 4 5 6 7 8

Input stream a b d a d e f g

Search pattern a b d f

Worked example of the search algorithm[edit]

To illustrate the algorithm's details, consider a (relatively artificial) run of the algorithm, where `W` = "ABCDABD" and `S` = "ABC ABCDAB ABCDABCDABDE". At any given time, the algorithm is in a state determined by two integers:

- `m`, denoting the position within `S` where the prospective match for `W` begins,
- `i`, denoting the index of the currently considered character in `W`.

In each step the algorithm compares `S[m+i]` with `W[i]` and advances `i` if they are equal. This is depicted, at the start of the run, like

```
          1       2
m: 012345678901234567890 12
S: ABCABCDAB ABCDABCDABDE
W: ABCDABD
i: 0123456
```

The algorithm compares successive characters of `W` to "parallel" characters of `S`, moving from one to the next by incrementing `i` if they match. However, in the fourth step `S[3] = ' '` does not match `W[3] = 'D'`. Rather than beginning to search again at `S[1]`, we note that no `'A'` occurs between positions 1 and 2 in `W`; hence, having checked all those characters previously (and knowing they matched the corresponding characters in `S`), there

is no chance of finding the beginning of a match. Therefore, the algorithm sets `m = 3` and `i = 0`.

```
          1       2
m: 012345678901234567890 12
S: ABCABCDAB ABCDABCDABDE
W:    ABCDABD
i:    0123456
```

This match fails at the initial character, so the algorithm sets `m = 4` and `i = 0`

```
          1       2
m: 012345678901234567890 12
S: ABC ABCDABABCDABCDABDE
W:     ABCDABD
i:     0123456
```

Here `i` increments through a nearly complete match `"ABCDAB"` until `i = 6` giving a mismatch at `W[6]` and `S[10]`. However, just prior to the end of the current partial match, there was that substring `"AB"` that could be the beginning of a new match, so the algorithm must take this into consideration. As these characters match the two characters prior to the current position, those characters need not be checked again; the algorithm sets `m = 8` (the start of the initial prefix) and `i = 2` (signaling the first two characters match) and continues matching. Thus the algorithm not only omits previously matched characters of `S` (the `"BCD"`), but also previously matched characters of `W` (the prefix `"AB"`).

```
          1       2
m: 012345678901234567890 12
S: ABC ABCDABABCDABCDABDE
W:       ABCDABD
i:       0123456
```

This search fails immediately, however, as `W` does not contain another `"A"`, so as in the first trial, the algorithm returns to the beginning of `W` and begins searching at the mismatched character position of `S`: `m = 10`, reset `i = 0`.

```
          1       2
m: 012345678901234567890 12
S: ABC ABCDABABCDABCDABDE
W:         ABCDABD
i:         0123456
```

The match at `m=10` fails immediately, so the algorithm next tries `m = 11` and `i = 0`.

```
          1       2
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:          ABCDABD
i:          0123456
```

Once again, the algorithm matches `"ABCDAB"`, but the next character, `'C'`, does not match the final character `'D'` of the word `W`. Reasoning as before, the algorithm sets `m = 15`, to start at the two-character string `"AB"` leading up to the current position, set `i = 2`, and continue matching from the current position.

```
          1       2
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:             ABCDABD
i:             0123456
```

This time the match is complete, and the first character of the match is `S[15]`

Description of pseudocode for the search algorithm[edit]

The above example contains all the elements of the algorithm. For the moment, we assume the existence of a "partial match" table `T`, described below, which indicates where we need to look for the start of a new match in the event that the current one ends in a mismatch. The entries of `T` are constructed so that if we have a match starting at `S[m]` that fails when comparing `S[m + i]` to `W[i]`, then the next possible match will start at index `m + i - T[i]` in `S` (that is, `T[i]` is the amount of "backtracking" we need to do after a mismatch). This has two implications: first, `T[0] = -1`, which indicates that if `W[0]` is a mismatch, we cannot backtrack and must simply check the next character; and second, although the next possible match will *begin* at index `m + i - T[i]`, as in the example above, we need not actually check any of the `T[i]` characters after that, so that we continue searching from `W[T[i]]`. The following is a sample pseudocode implementation of the KMP search algorithm.

```
algorithm kmp_search:
input:
    an array of characters, S (the text to be searched)
    an array of characters, W (the word sought)
output:
    an integer (the zero-based position in S at which W is found)

define variables:
    an integer, m ← 0 (the beginning of the current match in S)
    an integer, i ← 0 (the position of the current character in W)
```

```
        an array of integers, T (the table, computed elsewhere)

while m + i < length(S) do
if W[i] = S[m + i] then
if i = length(W) - 1 then
return m
let i ← i + 1
else
if T[i] > -1 then
let m ← m + i - T[i], i ← T[i]
else
let i ← 0, m ← m + 1

    (if we reach here, we have searched all of S unsuccessfully)
return the length of S
```

Efficiency of the search algorithm[edit]

Assuming the prior existence of the table T, the search portion of the Knuth–Morris–Pratt algorithm has complexity $O(n)$, where n is the length of S and the O is big-O notation. Except for the fixed overhead incurred in entering and exiting the function, all the computations are performed in the **while** loop. To bound the number of iterations of this loop; observe that T is constructed so that if a match which had begun at S[m] fails while comparing S[m + i] to W[i], then the next possible match must begin at S[m + (i - T[i])]. In particular, the next possible match must occur at a higher index than m, so that T[i] < i.

This fact implies that the loop can execute at most $2n$ times, since at each iteration it executes one of the two branches in the loop. The first branch invariably increases i and does not change m, so that the index m + i of the currently scrutinized character of S is increased. The second branch adds i - T[i] to m, and as we have seen, this is always a positive number. Thus the location m of the beginning of the current potential match is increased. At the same time, the second branch leaves m + i unchanged, for m gets i - T[i] added to it, and immediately after T[i] gets assigned as the new value of i, hence new_m + new_i = old_m + old_i - T[old_i] + T[old_i] = old_m + old_i. Now, the loop ends if m + i = n; therefore, each branch of the loop can be reached at most n times, since they respectively increase either m + i or m, and m ≤ m + i: if m = n, then certainly m + i ≥ n, so that since it increases by unit increments at most, we must have had m + i = n at some point in the past, and therefore either way we would be done.

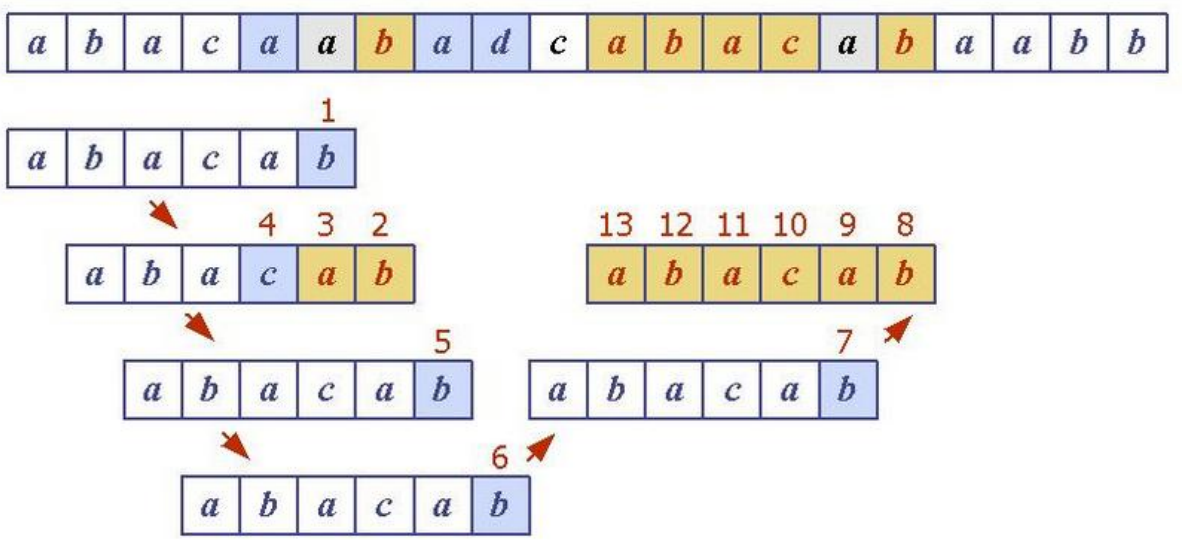
Thus the loop executes at most $2n$ times, showing that the time complexity of the search algorithm is $O(n)$.

Here is another way to think about the runtime: Let us say we begin to match W and S at position i and p. If W exists as a substring of S at p, then W[0..m] = S[p..p+m]. Upon success, that is, the word and the text matched at the positions (W[i] = S[p+i]), we

increase i by 1. Upon failure, that is, the word and the text does not match at the positions ($W[i] \neq S[p+i]$), the text pointer is kept still, while the word pointer is rolled back a certain amount ($i = T[i]$, where T is the jump table), and we attempt to match $W[T[i]]$ with $S[p+i]$. The maximum number of roll-back of i is bounded by i, that is to say, for any failure, we can only roll back as much as we have progressed up to the failure. Then it is clear the runtime is $2n$.

BOYER MOORE ALGORITHM

- string algorithm is significantly enhanced as the comparison Process started at the end of the search pattern processing right to left versus the start of the search pattern.
- The advantage is that large jumps are mismatched character in the input stream the search pattern which occurs frequently.



Examples:

1) Input:

txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

Output:

Pattern found at index 10

2) Input:

txt[] = "AABAACAADAABAAABAA"

```
pat[] = "AABA"

Output:

Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

Naive Algorithm
KMP Algorithm
Rabin Karp Algorithm
Finite Automata based Algorithm

In this post, we will discuss Boyer Moore pattern searching algorithm. Like KMP and Finite Automata algorithms, Boyer Moore algorithm also preprocesses the pattern.
Boyer Moore is a combination of following two approaches.
1) Bad Character Heuristic
2) Good Suffix Heuristic
Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It preporcesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.
In this post, we will discuss bad character heuristic, and discuss Good Suffix heuristic in the next post.

The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of pattern is called the Bad Character. Whenever a character doesn't match, we slide the pattern in such a way that aligns the bad character with the last occurrence of it in pattern. We preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes O(n/m) time in the best case.

```
/* Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm */

# include <limits.h>
```

```
# include <string.h>
# include <stdio.h>

# define NO_OF_CHARS 256

// A utility function to get maximum of two integers
intmax (inta, intb) { return(a > b)? a: b; }

// The preprocessing function for Boyer Moore's bad character heuristic
voidbadCharHeuristic( char*str, intsize, intbadchar[NO_OF_CHARS])
{
   inti;

   // Initialize all occurrences as -1
   for(i = 0; i < NO_OF_CHARS; i++)
       badchar[i] = -1;

   // Fill the actual value of last occurrence of a character
   for(i = 0; i < size; i++)
       badchar[(int) str[i]] = i;
}

/* A pattern searching function that uses Bad Character Heuristic of
  Boyer Moore Algorithm */
voidsearch( char*txt,  char*pat)
{
   intm = strlen(pat);
   intn = strlen(txt);

   intbadchar[NO_OF_CHARS];
```

```
/* Fill the bad character array by calling the preprocessing
   function badCharHeuristic() for given pattern */
badCharHeuristic(pat, m, badchar);

ints = 0;  // s is shift of the pattern with respect to text
while(s <= (n - m))
{
   intj = m-1;

   /* Keep reducing index j of pattern while characters of
      pattern and text are matching at this shift s */
   while(j >= 0 && pat[j] == txt[s+j])
      j--;

   /* If the pattern is present at current shift, then index j
      will become -1 after the above loop */
   if(j < 0)
   {
      printf("\n pattern occurs at shift = %d", s);

      /* Shift the pattern so that the next character in text
         aligns with the last occurrence of it in pattern.
         The condition s+m < n is necessary for the case when
         pattern occurs at the end of text */
      s += (s+m < n)? m-badchar[txt[s+m]] : 1;

   }

   else
      /* Shift the pattern so that the bad character in text
         aligns with the last occurrence of it in pattern. The
```

```
            max function is used to make sure that we get a positive
            shift. We may get a negative shift if the last occurrence
            of bad character in pattern is on the right side of the
            current character. */
          s += max(1, j - badchar[txt[s+j]]);
    }
}

/* Driver program to test above funtion */
intmain()
{
    chartxt[] = "ABAAABCD";
    charpat[] = "ABC";
    search(txt, pat);
    return0;
}
```

Run on IDE

Output:

```
pattern occurs at shift = 4
```

HARDWARE TEXT SEARCH ALGORITHMS

- Specialized hardware machine to perform the searches and pass the results to the main computer which supported the user interface and retrieval of hits.
- Since the searcher is hardware based, scalability is achieved By increasing the number of hardware search devices.
- The only limit on speed is the time it takes to flow the text off of secondary storage By having one search machine per disk, the maximum time it takes to search a database of any size will be the time to search one disk

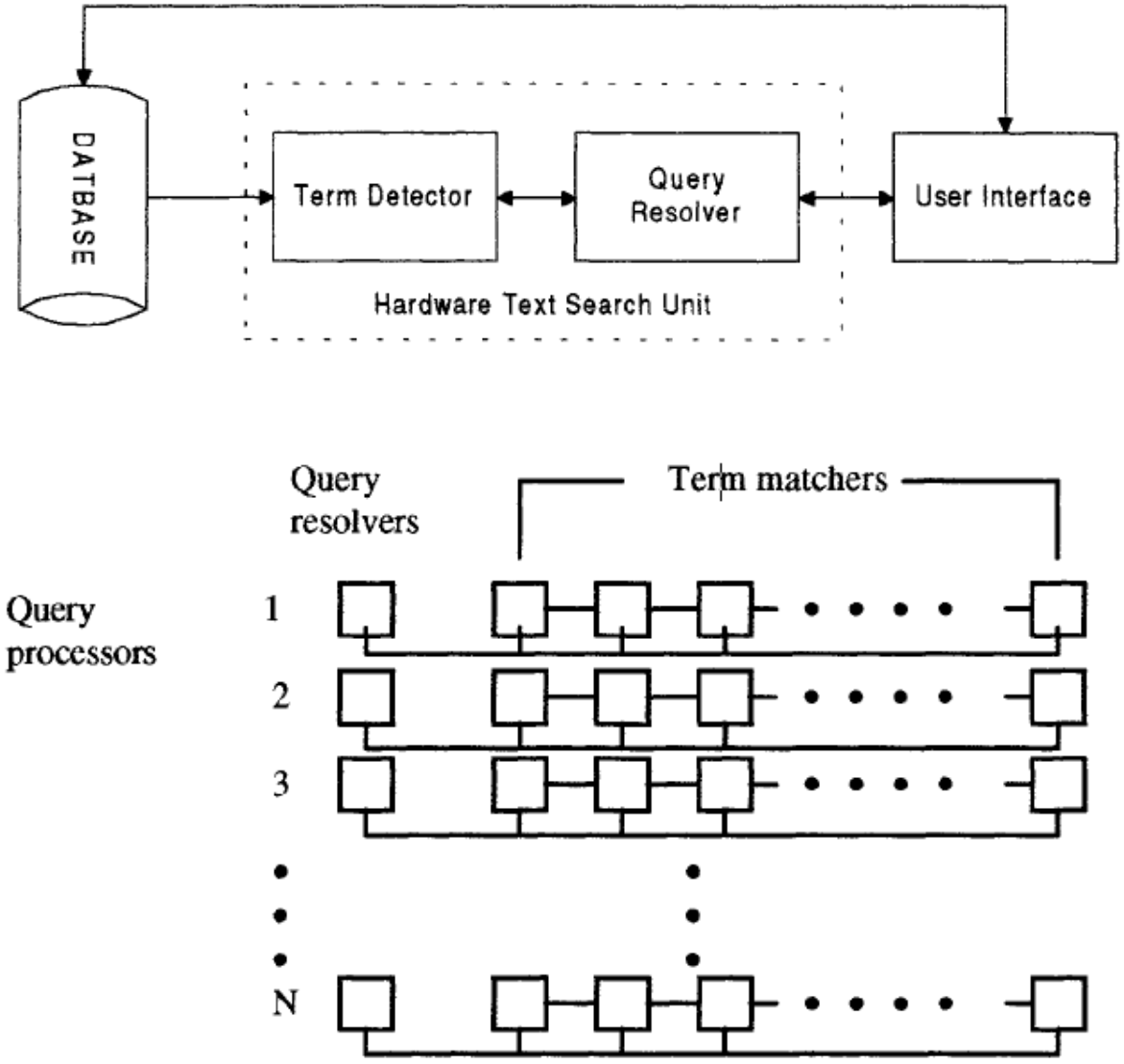


Figure 9.10 GESCAN Text Array Processor

The Fast Data Finder (FDF) is the most recent specialized hardware text search unit still in use in many organizations. It was developed to search text and has been used to search English and foreign languages. The early Fast Data Finders consisted of an array of programmable text processing cells connected in series forming a pipeline hardware search processor (Mettler-93). The cells are implemented using a VSLI chip. In the TREC tests each chip contained 24
processor cells with a typical system containing 3600 cells (the FDF-3 has a rack mount configuration with 10,800 cells). Each cell is a comparator for a single character, limiting the total number of characters in a query to the number of cells. The cells are interconnected with an 8-bit data path and approximately 20-bit control path. The text to be searched passes through each cell in a pipeline fashion until the complete database has been searched. As data are analyzed at each cell, the 20 control lines states are modified depending upon their current state and the results from the comparator. An example of a Fast Data Finder system is shown in Figure 9.11.

A cell is composed of both a register cell (Rs) and a comparator (Cs). The input from the Document database is controlled and buffered by the microprocessor/memory and feed

through the comparators. The search characters are stored in the registers. The connection between the registers reflects the control lines that are also passing state information.

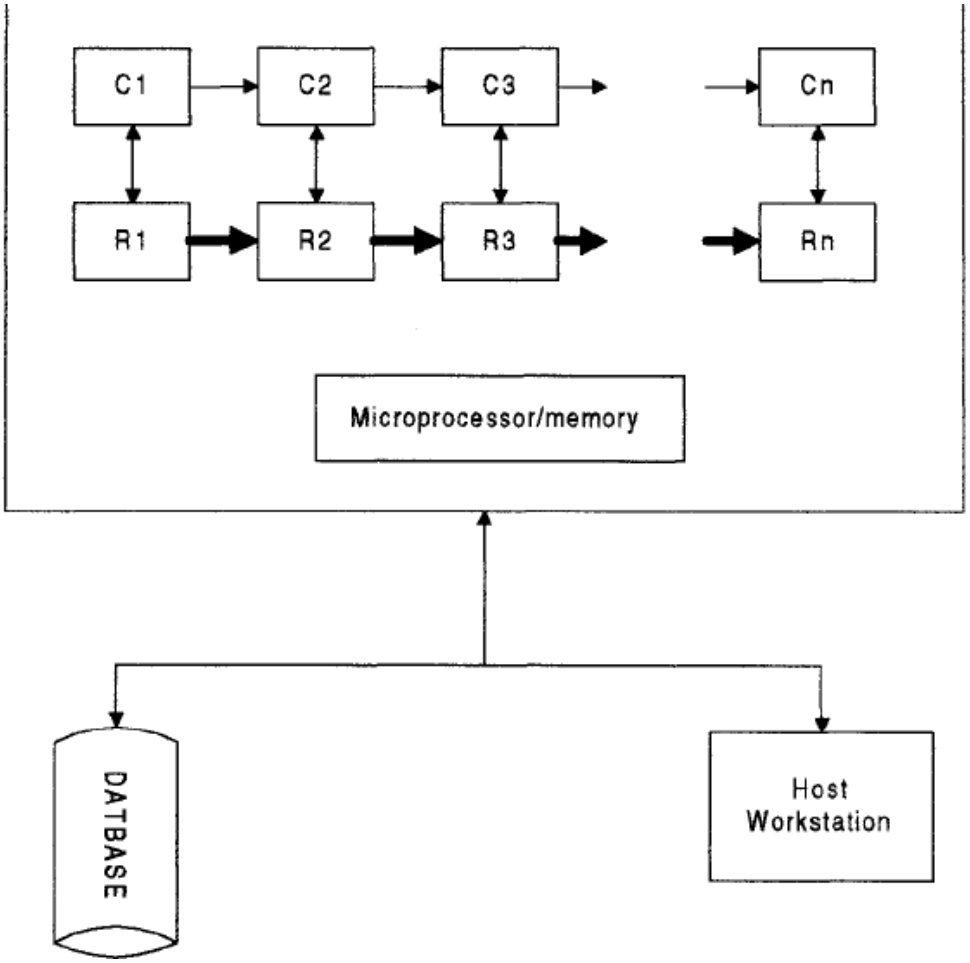


Fig 9.1Fast data finder architecture

INFORMATION SYSTEM EVALUATION

- In recent years the evaluation of IRS and techniques for indexing, sorting, searching and retrieving information have become increasingly important.
- This growth in interest is due to two major reasons:
  1.The growing number of retrieval systems being used
  2.Additional focus on evaluation methods themselves

- There are many reasons to evaluate the effectiveness of an IRS
  1.To aid in the selection of a system to procure
  2.To monitor and evaluate system effectiveness
  3.To evaluate query generation process for improvements
  4.To determine the effects of changes made to an existing information system
- From a human judgment standpoint, relevancy can be considered:
  1.Subjective: depends upon a specific user's judgment

2.Situational : relates to a user's requirements
3.Cognitive : depends on human perception and behaviour
4.Temporal : changes over time
5.Measurable : observable at points in time

- Ingwersen categorizes the information view into four types of "aboutness":
  1.AuthorAboutness:determined by the author's language as matched by the system in natural language retrieval
  2. Indexer Aboutness : determined by the indexer's transformation of the author's natural language into a controlled vocabulary
  3.Request Aboutness : determined by the user's or intermediary's processing of a search statement into a query
  4.UserAboutness : determined by the indexer's attempt to represent the document according to presupposition about what the user will want to know

- Measures used in system evaluation
  To define the measures that can be used in evaluating IRS, it is useful to define the major functions associated with identifying relevant items in an information system.