4/30/2020

Algorithm

COMP20290 Git Repository

This Document Contains Brief Description About The Algorithm Module Practical (1-9)

Ronit Dahiya
UNIVERSITY COLLEGE DUBLIN

PRACTICAL 1

In this practical, we use the Russian Peasant Algorithm to take two input from user and then multiply it to get the result. The algorithm is explained in the code.

The excel file contains the computation time for number of inputs given. It also contains the graph for the same.

PRACTICAL 2

In this practical we learnt about the basic growth functions. We learned to calculate the growth order of several algorithm. The assignment work is submitted in the word file in Practical2_AlgorithmCompare folder.

PRACTICAL 3

In this practical we implemented Fibonacci sequence and tower of Hanoi. Both computation of Fibonacci using iterative and recursive method calculated and compared.

PRACTICAL 4

In this practical we implemented different sorting techniques like InsertionSort, SelectionSort & BogoSort. We have also compared the computation time for the all these sorting techniques.

GeneratorArray is the file that creates random number of random sizes. More details are in the code. This file is used to provide different data inputs for these sorting algorithms to test their computation time.

Excel sheet containing graph uploaded with the sorting algorithm implementation.

PRACTICAL 5

In this practical we implemented different and efficient sorting technique like mergeSort. This is more efficient than previous technique. The computation file excel is uploaded with graph

GeneratorArray is the file that creates random number of random sizes. More details are in the code. This file is used to provide different data inputs for merge sorting algorithms to test their computation time.

PRACTICAL 6

In this practical we implemented another sorting technique quickSort. Computation file excel is uploaded with graph.

GeneratorArray is the file that creates random number of random sizes. More details are in the code. This file is used to provide different data inputs for merge sorting algorithms to test their computation time.

PRACTICAL 7

In this we implemented a unique algorithm, knuth-morris-path algorithm which is used for string search. Its also used for pattern searching. It enable us to find a sub string inside a big database.

Bruteforce, knurh-morris-pratt algorithm implemented.

We observe that the KMP differs from the brute-force algorithm by keeping track of information gained form previous comparison. Failure function shows how much last comparison can be re-used if it fails.

PRACTICAL 8

Tries are implemented in this mode. They are extremely useful for finding a string/word in a dictionary of words. In the code we are able to search and also insert an element.

PRACTICAL 9

Runlength encoding performed in this practical.

- Number of bits in the binary file '4run.bin'

```
C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>java  BinaryDump 40 < 4runs.bin
0000000000000001111111000000011111111111
40 bits
```

  - Command: **java BinaryDump 40 <4runs.bin**
  - Output: **0000000000000001111111000000011111111111 40 bits**

- File compressed using RunLength encoding. (Runlength & BinaryDump combined)

```
C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>java RunLength -<4runs.bin | java BinaryDump
0000111100000111
0000011100001011
32 bits
```

  - Command: **java Runlength - <4runs.bin | java BinaryDump**
  - Output: **0000111100000111**
    0000011100001011
    32 bits

- Compression Ratio: compressed bits/original bits

CR = 32/40 = 0.8

- Now, we create another new binary file and put the compressed file into it.

```
C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>java BinaryDump 40 <4runsrle.bin
00001111000001110000011100001011
32 bits
```

  - Command**: java BinaryDump 40 <4runsrle.bin**
  - Output**: 00001111000001110000011100001011 32 bits**

ASCII

- Number of bits in the text file 'abra.text'

```
C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>java BinaryDump 8 <abra.txt
01000001
01000010
01010010
01000001
01000011
01000001
01000100
01000001
01000010
01010010
01000001
00100001
96 bits
```

  - Command**: java BinaryDump 8 <abra.txt**
  - Output: **96 bits**

- File compressed using RunLength encoding. (Runlength & BinaryDump combined)

```
C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>java RunLength -< abra.txt | java BinaryDump 8
00000001    00000010    00000001
00000001    00000001    00000100
00000101    00000010    00000001
00000001    00000001    00000010
00000001    00000101    00000001
00000100    00000001    00000001
00000001    00000001    00000010
00000010    00000001    00000001
00000001    00000010    00000001
00000001    00000001    00000101
00000001    00000101    00000001
00000101    00000001    00000010
00000001    00000001    00000001
00000001    00000001    00000100
00000100    00000001    00000001
00000010    00000011    416 bits
00000001    00000001
00000001    00000101
00000001    00000001
00000011    00000001
00000001    00000001
00000011    00000001
00000001    00000101
```

  - Command: **java Runlength - <abra.txt | java BinaryDump**
  - Output: **416 bit**

- Compression Ratio: compressed bits/original bits

CR = 96/416 = .230

- My txt file original vs compressed bit.

COMPRESSION RATIO: 2072/9232 =0.224

REASON:

RunLength Encoding is a lossless algorithm that performs good compression only in certain file types. What it does is simply replaces the data values in a file with a count number and a single value.

Its worth noting that it is only effective when there are more than 4 characters sequence repeating as three character would result in coding two repeating characters which ultimately increase the size of file, thus defeating its purpose.

Since in the above the character are large, so runLength encoding reduces the size of the file, thus performing good compression.

BITMAP COMPRESSION

SMALL BITMAP FILE (X48)

- Number of bits in bitmap file q32x48.bin

```
00000000
00000000
00000000
00000000
00000000
1536 bits

C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>
```

- Number of bits in bitmap file q32x48rle.bin

```
00010110
00001100
00010011
00001110
01000001
1144 bits

C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>
```

COMPRESSION RATIO: 1536/1144 = 1.34

BIG BITMAP FILE (X96)

- Number of bits in bitmap file x96 : **6144 bits**

```
00000000
00000000
00000000
6144 bits

C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>java RunLength -<q64x96.b
in> q64x96rle.bin
```

- Compressed file bit: **2296bit**

```
00100000
11111111
00000000
01000001
2296 bits

C:\Users\Ronit\Desktop\Git_Repo\Practical9\out\production\Practical9>
```

COMPRESSION RATIO: 6144/2296 = 2.67

RunLength supports file format like BMP. Although RLE doesn't provide much compression but its easy to implement and less execution time.