17.11.2020

Assembler Programming Languages - project report

Project topic:

Pattern searching with Rabin-Karp algorithm

Paweł Szafraniec

Informatics, sem. 5

Year 2020/2021

Instructor: dr. inż. Piotr Czekalski

Lab group: Thursday 10^{30}-12^{00}

Table of contents

1. Algorithm description

Rabin–Karp algorithm is a string-searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern, and then checks for a match at the remaining positions. The hash values for all substrings in the pattern must be calculated carefully to omit spurious hits (fig. 1) - situations, where hash values for the pattern and the substring of the text are the same, but the digits does not match. Pseudocode for the algorithm is presented on fig.2.

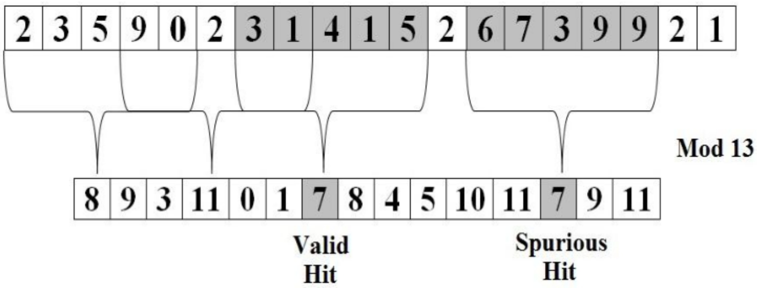


Fig.1

```
RABIN-KARP-MATCHER(T, P, d, q)
1   n = T.length
2   m = P.length
3   h = d^(m-1) mod q
4   p = 0
5   t_0 = 0
6   for i = 1 to m                      // preprocessing
7       p = (dp + P[i]) mod q
8       t_0 = (dt_0 + T[i]) mod q
9   for s = 0 to n - m                  // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n - m
14          t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

Fig. 2

In pseudocode steps 6-9 hash values for the pattern and for the first window of the text are calculated. Each next window hash value of the text is calculated based on its previous window’s hash value (fig. 3). Step 10 compares the values of hashes, and if they are equal algorithm compares letters of the pattern hash with letters of the text window hash.

t: A B C D E   p: B C

t_1

t_0 (for AB) = 11

t_1 (for BC) = (d * (t0 - text [0]*h) + text [0 + text.L]) mod q =
(5 * (11- (97 * 5)) + 99) mod 13 = -2271 mod 13 = 4

$p_H = 4$

Fig. 3

2. Implementation description

2.1 Purpose of the algorithm implementation

Algorithm was implemented to compare speed execution of Assembler and C# codes. However, main purpose was to get familiar with writing in Assembler programming language and use its tools to maximize efficiency of the algorithm.

2.2 Description of input parameters

Program is designed as a console application. User is asked to enter text, pattern to be searched and modulo value. The last input parameter should be a prime number, high enough to limit number of spurious hits in comparing hash values.

Besides those mentioned above, 4 more parameters are passed to the assembler function: text length, pattern length, number of distinct elements in the text and array containing places where pattern occurs in the text.

3. Analysis of selected piece of assembler code

Piece of assembler code presented below (fig. 4) presents main loop for the algorithm (pseudocode steps 9-12). Basically it performs pattern and text window hashes comparison and depending on the result of this operation compares characters of the pattern and text window or moves to step calculating hash value of the next text window.

```
97  MainLoop: ; looping through text
98      ; for (i=0; i<= (text.length - pattern.length); i++)
99      cmpeqpd xmm8,xmm9 ; compare hashes for pattern and text window SSE2
100     xor edx,edx
101     jz CharCheck            ; jump CharCheck if r11=r12
102     jnz Continue            ; else
103         CharCheck: ; check if all digits are matching
104         ; for (j=0; j<pattern.length; j++)
105         movd rax,xmm0       ; load address of the text
106         add rax,rcx         ; address + i
107         add rax,rdx         ; address + j
108         mov r11b,[rax]      ; load text[i+j]
109         movd rax,xmm1       ; load address of the pattern
110         add rax,rdx         ; address + j
111         mov r12b,[rax]      ; load pattern[j]
112         cmp r11b,r12b       ; compare r11 and r12
113         jnz Continue        ; jump if text[i+j] != pattern[j]
114
115         inc edx             ; edx++
116         cmp edx,r10d        ; check the loop
117         jnz CharCheck       ; loop if i<pattern.length
118         jz Found            ; jump if j=pattern.length -> loop finished, all digits match
119     Found:
120         movd r11,xmm10      ; load address of results array
121         mov [r11],ecx       ; add position
122         add r11,4           ; move to the next position in the array
123         movd xmm10,r11      ; save the address
124     Continue:
125         cmp ecx,r9d         ; compare i to text.length-pattern.length
126         jb NextWindow       ; jump if below
127         jae Round           ; jump if above or equal
128     NextWindow:
```

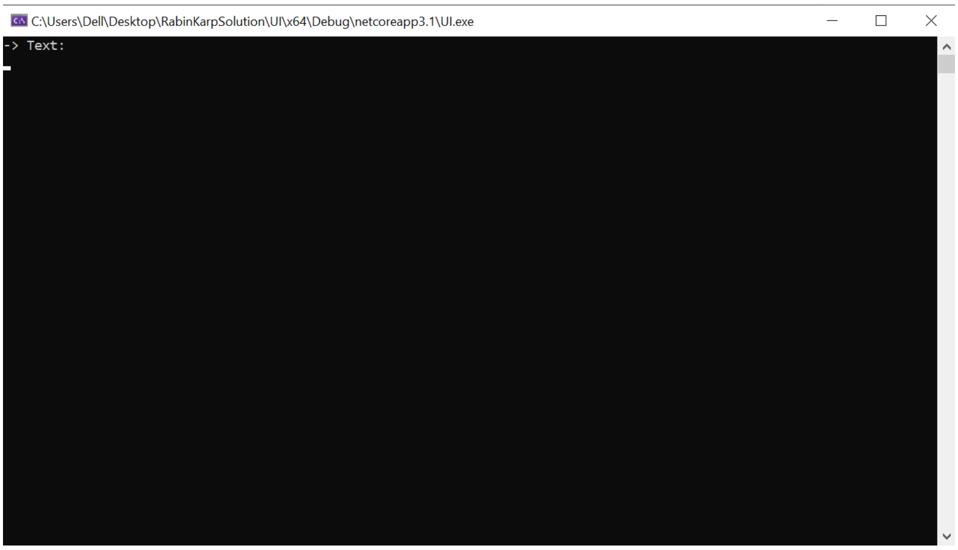
Fig 4.

cmpeqpd instruction is used to compare values of hashes of pattern and window text. If those values are equal (line 101) algorithm jumps to CharCheck loop (lines 105-118) otherwise it jumps to line 125. In CharCheck loop **movd** instruction loads address of first element of the text. **add** instructions in steps 106 and 107 are used to determine current position of the first element of the text window and character in that window respectively. **mov** instruction in step number 109 loads value of text array to compare it with value of pattern array loaded in step number 111 which is calculated with similar manner as it is for the pattern array character (steps 109,110). **jnz** instruction in step 113 checks whether value of character of the pattern is the same as corresponding value of character of the text. If result is different than 0 means that pattern and text window are not the same. In that situation algorithm jumps outside the CharCheck loop. If **jnz** instruction in step 113 returns value equal 0, value of *edx* register is incremented - it allows to move to the next character of the text window. **cmp** instruction in step 116 checks whether algorithm reached the end of pattern array. If yes (step 118) - all characters matched – algorithm jumps to Found section. In this section, **movd** instruction loads current address of result array and **mov** instruction in the next step stores value of *ecx* register (index on which matched pattern starts) in that address. Step 122 shifts address of the result array to space available for next, potentially found match. However, if **cmp** instruction in step 117 returns value different value than 0, algorithm jumps to step 105 again to perform next iteration of characters comparison. Loop finishes either when last element of pattern array is reached or when corresponding characters does not match. In this case, algorithm moves to step 125. If **cmp** instruction gives value less than 0, algorithm moves to NextWindow section calculating hash value of next text window. Otherwise it moves forward to execute next iteration of MainLoop loop.

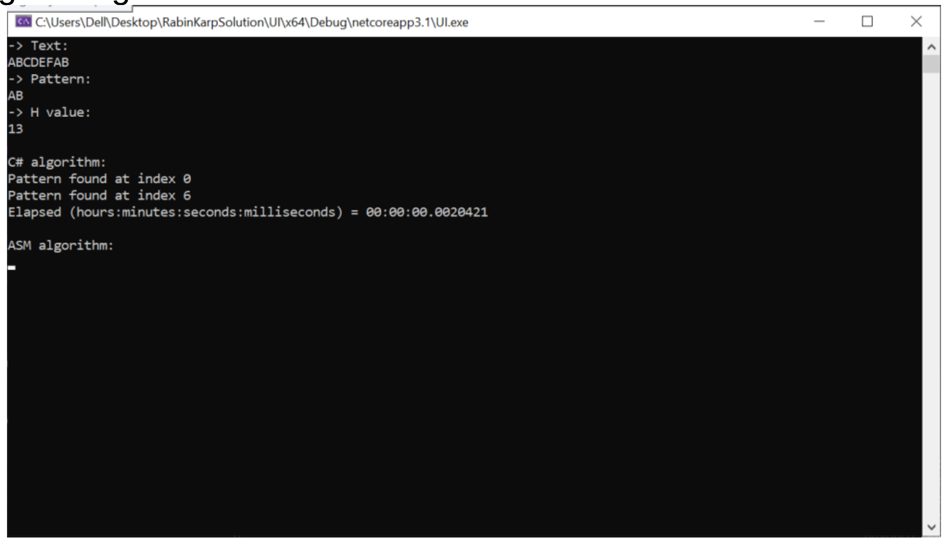
4. Runtime of the algorithm

User interface:

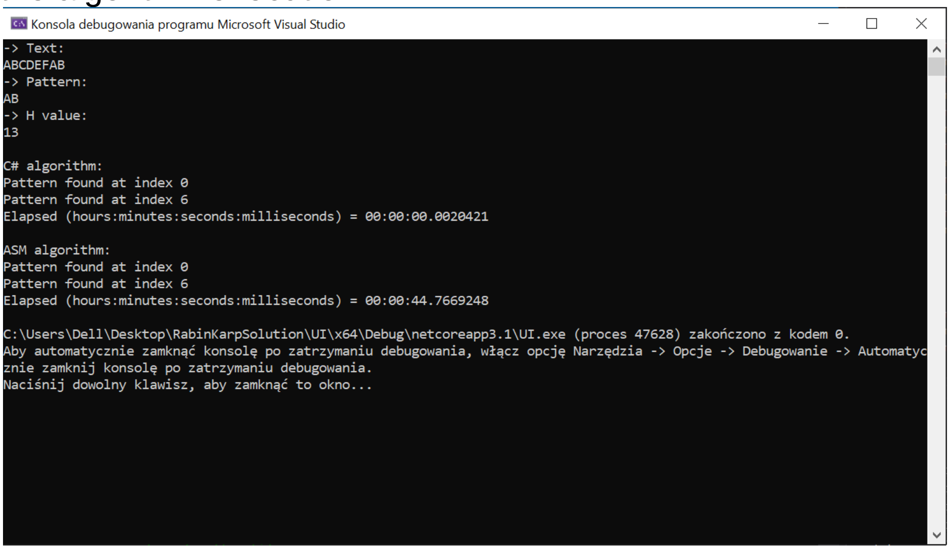
a. Before the algorithm execution:



b. During the algorithm execution:



c. After the algorithm execution:



5. Results analysis

	Text length	**Pattern length**	**Modulo value**	**C# algorithm speed *(seconds:ms)***	**ASM algorithm speed *(seconds:ms)***
1.	61	2	13	0.0287435	0.825984
2.	52	6	13	0.006830	0.015916
3.	513	25	13	0.5865617	0.6091401
4.	819	3	13	0.0207394	0.0446363
5.	40	40	13	0.0008818	0.0149902
6.	40	40	47	0.0010103	0.0167516
7.	808	5	47	0.0013084	0.0180633
8.	237	2	47	0.0015727	0.0294101
9.	2737	4	47	0.3672151	**0.3610959**
10.	4094	4	47	0.5564094	**0.4645512**
11.	4094	4	13	0.517115	**0.4780579**
12.	4094	2	13	0.5357226	**0.4737911**
13.	4094	15	13	0.1754161	0.2568507
14.	4094	4094	13	0.0036794	0.282582

Fig.5

	Text complexity	**Pattern complexity**	**Modulo value**	**C# algorithm speed *(seconds:ms)***	**ASM algorithm speed *(seconds:ms)***
1.	low	low	13	0.0162240	0.739181
2.	high	high	13	0.0011909	0.207720
3.	high	low	13	0.0007283	0.193743
4.	low	Low	47	0.0722760	0.1390187
5.	High	high	47	0.0009653	0.0151222
6.	High	Low	47	0.0012401	0.0219922

Fig. 6

6. Testing & debugging

Algorithm was tested with different datasets. For each of them it returned correct results. Algorithm was also debugged. Instructions work as intended and so – algorithm also works as intended. Program does not contain bugs or not managed errors.

7. Conclusions

Writing algorithm in Assembler was great opportunity to learn this language. In fact, that was first time when I wrote such long code in this language. I learned many useful features, such as: calling convention, different types of loops and their usage, data management, different types of instructions. Unfortunately I was not able to use many SIMD instructions due to algorithm specification. The algorithm works properly and every instruction I've used does exactly what it should. Writing that algorithm was quite a challenge for me, but the final effect was worth the effort.

I was hoping that the Assembler algorithm would work faster than the C# one, however in most of cases C# algorithm is way faster (fig.5). It turns out, that Assembler algorithm is faster only in cases when text is very long and the pattern to search relatively short. However it is not clear how long text and how short pattern should be to be sure that the assembler algorithm will be faster. Data complexity affects the speed of the algorithm (fig. 6), but for each case the impact is different. Interesting thing here is, that for low text and pattern complexity algorithm is more than 3 times slower than for high text complexity and low pattern complexity. It may be caused by the fact that in first situation pattern was found in more places than in the second situation. It can lead to a conclusion, that process of characters matching is very time consuming. Nevertheless, for each situation C# is faster than the Assembler. Modulo value selection influence on speed of execution of the algorithm- rows 5 and 6 on the fig.6 presents analysis for the same text and pattern but different modulo value. Similarly for rows 10 and 11, however in that case greater modulo value gives better result. Conclusion is that modulo value must be selected carefully according to quantity of distinct characters in the text to obtain the best possible time of execution. Unfortunately, but generally C# algorithm is faster than the Assembler algorithm, however for searching short patterns in a long text it may be worth to use Assembler implementation of the algorithm.