성균관대학교

Assignment 1 Report

자료구조개론_SWE2015_42 (염익준 교수님)

2019313611 김지훈

2022-10-6

목차

1. 문제 설명 및 사전 이론

- char *strstr(char *s, char *pat) (slide 1 in lecture 4)
- nfind() in slide 3 in lecture 4
- Fast pattern matching in slide 5 in lecture 4

Implement three pattern matching programs which return the number of matched occasions instead of returning the first index of the matched patterns using the three algorithms.

Note that you should modify them to find all the matched patterns.

① char *strstr function

형식

```
#include <string.h>
char *strstr(const char *string1, const char *string2);
```

함수 설명

strstr() 함수는 string1에서 string2의 첫 번째 표시 시작 위치에 대한 포인터를 return합니다. String2가 string1에 나타나지 않으면 strstr() 함수는 NULL을 return합니다.[1]

strstr() 함수의 시간 복잡도는 O(mn)입니다. (m = string1의 길이, n = string2의 길이)

C언어에서는 CPU cache 이용을 극대화할 수 있도록 <string.h> header file에 선언되어 있는 strstr function을 assembly 코드로 작성하였습니다. assemblier 최적화를 하여 O(mn)의 시간복잡도를 가지고 있더라도 빠른 속도로 함수가 실행됩니다. strstr.asm의 경로는 아래와 같습니다.

경로 : C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\crt\src\intel\strstr.asm [2]

② nfind() function

시간 복잡도

nfind() 함수의 시간 복잡도는 O(mn)입니다. (m = string1의 길이, n = string2의 길이)

---

[1] "strstr() – 서브스트링 찾기", IBM, https://www.ibm.com/docs/ko/i/7.3?topic=functions-strstr-locate-substring

[2] https://www.sysnet.pe.kr/2/0/1843

③ Fast pattern matching function (Use failure function)

시간 복잡도

pmatch() 함수의 시간 복잡도는 O(m+n)입니다. (m = string1의 len, n = string2의 len)

2. 사용 코드 (.c file로도 첨부하였습니다.)

함수 및 구문들의 설명들은 전부 주석으로 추가하였습니다.

특이한 점은 string과 pat을 만든 후 file에 저장한 후, 그 file에서 다시 불러와서 사용한다는 점인데, 이렇게 만든 이유는 두 가지입니다.

첫 번째는 string 크기 검사 때문입니다. 문자열들이 제가 원하는 크기로 만들어졌는지 확인하기 위하여 file을 이용하였습니다.

두 번째는 random string 뿐만 아니라 특정 경우의 string들도 실행해볼 것인데, 이 경우 코드에 1MB 이상의 문자열을 적기는 매우 힘들기 때문에 file을 이용하였습니다.

각 실행들을 10번씩 반복해본 후 그 때의 실행 시간들을 평균 내어 마지막에 출력하였습니다. 이는 실행 횟수가 적을 경우 오차율이 매우 커지기 때문입니다.

Assignment 1 code_2019313611_김지훈.c

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>

// 100 MB = 100,000 KB = 100,000,000 byte
// char 는 1 byte 변수이므로 "MAX_LENGTH"를 100,000,001 로 setting 합니다. (최대 사용 용량 50MB)
#define MAX_LENGTH 100000001

/* 지역변수로 선언 시 스택에 생성되어 "MAX_LENGTH"일 때 Runtime Error 가 생길 수 있지만,
 * 전역변수로 선언 시 힙 영역에 생성되어 제한으로부터 자유롭기에 전역변수로 선언하였습니다. */
char *string;
char *pat;
```

```c
char *t; // "string"내에 등장하는 "pat"의 가장 앞을 가르키는 pointer

// 1. char *strstr(char *s, char *pat) (slide 1 in lecture 4)
/* The strstr function is built-in in the string.h header file */
int strstrfind (char *string, char *pat) {
    int counter = 0; // variable for count "pat" in "string"

    // strstr 함수는 "string"에서 "pat"의 첫 번째로 등장하는 자리를 char
pointer 로 retrun 합니다.
    t = strstr(string, pat);
    // "string"에 "pat"이 없을 때까지 반복합니다.
    while(t != NULL) {
        /* Case of find "pat" in "string" */
        counter++;
        // "pat"을 찾은 pointer 의 다음 pointer 부터 다시 "pat"을 찾습니다.
        t = strstr(t+1, pat);
    } // end while

    return counter;
} // end strstrfind function

// 2. nfind() in slide 3 in lecture 4
int nfind (char *string, char *pat) {
    int i, j; // variable for using the "for" statement
    int start = 0; // first index of "string" to compare
    int lasts = strlen(string)-1; // last index of "string"
    int lastp = strlen(pat)-1; // last index of "pat"
    int endmatch = lastp; // 비교할 "string"의 마지막 index

    int counter = 0; // variable for count "pat" in "string"

    // i 는 "string"의 idx, j 는 "pat"의 idx
    for (i=0 ; endmatch<=lasts ; endmatch++,start++) {
        /* if endmatch > lasts, exit the "for" statement.
         * else "endmatch"++ and "start"++ */
        if (string[endmatch] == pat[lastp]){
            /* "string"의 endmatch 번째 문자와 "pat"의 lastp 번째 문자가 같은 경우
             * string[start]와 pat[0]부터 pat[lastp-1]까지 문자를 모두
비교합니다 */
            for (j=0,i=start ; j<lastp&&string[i]==pat[j] ; i++,j++)
                ; // end for

            if (j == lastp) {
                /* string[start]와 pat[0]부터 string[endmatch]와 pat[lastp]까지
모두 같은 경우 */
                counter++;
            } // end if
        } // end if
    } // end for

    return counter;
} // end nfind function

// 3. Fast pattern matching in slide 5 in lecture 4
/* failure function value array */
```

```c
int failure[MAX_LENGTH];

/* fail function is that sets the failure function value to the failure
array. */
void fail (char *pat) {
    int i, j; // variable for using the "for" statement
    int n = strlen(pat); // variable for length of "pat"
    failure[0] = -1; // Let set failure[0] = -1

    for (j = 1; j < n; j++){
        /* Set the failure array from 1 to n-1. */
        i = failure[j-1];
        while (pat[j] != pat[i+1] && i>=0)
            /* pat[j]와 pat[i+1]가 같을 때 exit 합니다. i = -1 가 될 때도
exit 합니다.*/
            i = failure[i]; // end while
        if (pat[j] == pat[i+1])
            /* 위의 while 문을 pat[j]와 pat[i+1]가 같아서 빠져나온 case
             * (pat[j]와 pat[0]이 같아서 빠져나온 case 도 포함) */
            failure[j] = i+1; // end if
        else
            /* 위의 while 문을 i = -1 가 되어서 빠져나왔는데, pat[j]와 pat[0]가
다른 문자인 경우 */
            failure[j] = -1; // end else
    } // end for
} // end fail function

/* pmatch function is that counts the number of "pat" in the "string" using
the failure array. */
int pmatch (char* string, char* pat) {
    int i=0; int j=0; // variable for using the "while" statement
    int counter = 0; // variable for count "pat" in "string"
    int lens=strlen(string); int lenp=strlen(pat);

    // i 는 "string"의 idx, j 는 "pat"의 idx
    while (i < lens) {
        if (string[i] == pat[j]) {
            i++;
            j++;
        } // end if
        else if (j == 0)
            i++; // end else if
        else
            j = failure[j-1]+1; // end else

        // j == lenp 라면 pattern 을 string 에서 찾았다는 것입니다.
        if(j == lenp) {
            // 따라서 counter 에 1 을 더해준 다음
            counter++;
            // j 에 failure[j-1]+1 을 넣어주며 탐색을 계속합니다.
            j = failure[j-1]+1;
        } // end if
    } // end while
    return counter;
} // end pmatch function
```

```c
/* 'a'부터 'z'까지의 random character 을 return 해주는 함수 */
char getRandomCharacter() {
    return 'a' + (rand()%26);
} // end getRandomCharacter function

/* 특정한 크기의 random string 을 만들고 "string.txt"에 입력하는 함수 */
void makeStringFile(int sizeidx) {
    int len; // variable for using the "for" statement
    int MB = (int)1e6;
    // "string"의 size 가 차례대로 1MB, 5MB, 10MB, 20MB, 30MB, 40MB, 50MB 로
만들어집니다.
    // ex) sizeidx = 3 -> Set string array size to 20MB
    int stringSize[7] = {1*MB, 5*MB, 10*MB, 20*MB, 30*MB, 40*MB, 50*MB};

    // 위에서 정해진 string size 크기만큼의 random string 을 만듭니다.
    for(len=0 ; len < stringSize[sizeidx] ; len++) {
        string[len] = getRandomCharacter();
    } // end for

    // "string.txt" file 을 만든 후, 그 파일에 위에서 만든 string 을 입력합니다.
    FILE *writeStringFile = fopen("./string.txt","w");
    fprintf(writeStringFile, string);
    fclose(writeStringFile); // Close the file pointer.
} // end makeStringFile

/* 특정한 크기의 random string 을 만들고 "pat.txt"에 입력하는 함수 */
void makePatFile(int sizeidx) {
    int len; // variable for using the "for" statement
    // string 의 size 가 차례대로 10byte, 50byte, 100byte, 200byte, 300byte,
400byte, 500byte 로 만들어집니다.
    // ex) sizeidx = 2 -> Set string array size to 100byte
    int patSize[7] = {10,50,100,200,300,400,500};

    // 위에서 정해진 "pat" size 크기만큼의 random pat 을 만듭니다.
    for(len=0 ; len < patSize[sizeidx] ; len++) {
        pat[len] = getRandomCharacter();
    } // end for

    // "pat.txt" file 을 만든 후, 그 파일에 위에서 만든 pat 을 입력합니다.
    FILE *writePatFile = fopen("./pat.txt","w");
    fprintf(writePatFile, pat);
    fclose(writePatFile); // Close the file pointer.
} // end makePatFile function

/* "string.txt"파일을 읽고 string 문자열에 넣어주는 함수 */
void readString() {
    // Open the "string.txt" file in the same folder in r(read) mode.
    FILE *readStringFile = fopen("./string.txt","r");
    /* 파일에 있는 문자열을 MAX_LENGTH 만큼 읽어서 string(char pointer array)에
넣어줍니다. */
    if(readStringFile == NULL) { // Exception handling
        /* Case of "string.txt" file does not exist */
        printf("Could not find file \"string.txt\".\n");
        exit(1); // End the program
```

```
    } // end if
    fread(string, 1, MAX_LENGTH, readStringFile);
    fclose(readStringFile); // close file pointer
} // end readString function

/* "pat.txt"파일을 읽고 pat 문자열에 넣어주는 함수 */
void readPat() {
    // Open the "pat.txt" file in the same folder in r(read) mode.
    FILE *patFile = fopen("./pat.txt","r");
    /* 파일에 있는 문자열을 MAX_LENGTH 만큼 읽어서 pat(char pointer array)에
넣어줍니다. */
    if(patFile == NULL) { // Exception handling
        /* Case of "pat.txt" file does not exist */
        printf("Could not find file \"pat.txt\".\n");
        exit(1); // End the program
    } // end if
    fread(pat, 1, MAX_LENGTH, patFile);
    fclose(patFile); // close file pointer
} // end readPat function

int main(void) {
    // MAX_LENGTH * 1byte 크기의 동적메모리 할당
    string = (char *)malloc(sizeof(char) * MAX_LENGTH);
    pat = (char *)malloc(sizeof(char) * MAX_LENGTH);
    t = (char *)malloc(sizeof(char) * MAX_LENGTH);

    srand(time(NULL));

    // Display about each case
    int stringcase = 2; int patcase = 2; // user가 원하는 string and pat 의
길이를 넣어준다.

    char stringsize[15], patsize[15];
    switch (stringcase) {
        case 0:
            strcpy(stringsize, "1MB (1e6 개)");
            break;
        case 1:
            strcpy(stringsize, "5MB (5e6 개)");
            break;
        case 2:
            strcpy(stringsize, "10MB (1e7 개)");
            break;
        case 3:
            strcpy(stringsize, "20MB (2e7 개)");
            break;
        case 4:
            strcpy(stringsize, "30MB (3e7 개)");
            break;
        case 5:
            strcpy(stringsize, "40MB (4e7 개)");
            break;
        case 6:
            strcpy(stringsize, "50MB (5e7 개)");
            break;
```

```
        default:
            break;
    } // end switch
    switch (patcase) {
        case 0:
            strcpy(patsize, "10byte");
            break;
        case 1:
            strcpy(patsize, "50byte");
            break;
        case 2:
            strcpy(patsize, "100byte");
            break;
        case 3:
            strcpy(patsize, "200byte");
            break;
        case 4:
            strcpy(patsize, "300byte");
            break;
        case 5:
            strcpy(patsize, "400byte");
            break;
        case 6:
            strcpy(patsize, "500byte");
            break;
        default:
            break;
    } // end switch

    int number; // variable for using the "for" statement
    int count = 0; // variable for string 에 있는 pat 의 개수
    double averageCount = 0; // 10 회 측정 중 string 에 있는 pat 의 개수를 평균
내기위해 선언된 변수

    // 시간 측정에 사용되는 variable
    time_t start, end; // variable checked when code start and end
    double strstrTime = 0, nfindTime = 0, pmatchTime = 0;

    /* 총 10 회 측정 후 측정시간을 평균낸다. */
    for(number = 0; number < 10 ; number++) {
        makeStringFile(stringcase); // random string 을 생성 후, "string.txt"에
입력합니다.
        makePatFile(patcase); // random pat 을 생성 후, "pat.txt"에 입력합니다.

        readString(); // "string.txt"에 있는 문자열을 string 에 넣어줍니다.
        readPat(); // "pat.txt"에 있는 문자열을 pat 에 넣어줍니다.

        // 1. strstr code
        start = clock();
        count = strstrfind(string, pat); // Find the number of "pat" in
"string" using the strstr function.
        end = clock();
        strstrTime += (double)(end - start);
        //printf("pattern count by strstr  : %d\n", count);
        //printf("strstr  소요시간: %.0lfms\n", (double)(end - start));
```

```
        // 2. nfind code
        start = clock();
        count = nfind(string, pat); // Find the number of "pat" in "string"
using the nfind function.
        end = clock();
        nfindTime += (double)(end - start);
        //printf("pattern count by nfind   : %d\n", count);
        //printf("nfind   소요시간: %.0lfms\n", (double)(end - start));

        // 3. pmatch code
        start = clock();
        fail(pat); // make failure array
        count = pmatch(string, pat); // Find the number of "pat" in "string"
using the pmatch function.
        end = clock();
        pmatchTime += (double)(end - start);
        //printf("pattern count by pmatch : %d\n", count);
        //printf("match 소요시간: %.0lfms\n", (double)(end - start));

        averageCount += count;
        //puts(""); // spacing
    } // end for

    // Display the result
    printf("Case %d-%d) string size: %s / patsize : %s.\n", stringcase+1,
patcase+1, stringsize, patsize);
    printf("Pattern count in string: %.1lf\n", averageCount/10);
    printf("strstr 평균 소요시간: %.1lfms\n", strstrTime/10);
    printf("nfind  평균 소요시간: %.1lfms\n", nfindTime/10);
    printf("pmatch 평균 소요시간: %.1lfms\n", pmatchTime/10);
    puts("");

    // Frees the allocated memory.
    free(string);
    free(pat);
    free(t);

} // end main function
```

3. 코드 실행 결과

string은 총 3가지 pattern을 주어 다양한 결과들이 나오도록 실행해보았습니다.

첫 번째는 random string입니다. 위 코드에 있는 `getRandomCharacter()` 함수를 이용하여 랜덤 char들을 받아 string을 만듭니다.

두 번째는 specific string입니다. 50byte 길이의 문장을 계속 반복하여 string을 만듭니다.

세 번째는 one alphabet string입니다. 하나의 alphabet을 반복하여 string을 만듭니다.

① Random string

여기서 pattern count가 모두 0개가 나오는 것을 볼 수 있는데, 이유는 다음과 같습니다. 10자리 이상의 alphabet string은 26^{10}이상의 경우의 수가 있는데, 이는 약 1.41×10^{14}으로 50MB 이하의 크기의 랜덤 string을 만들었기 때문에 pat이 등장하지 않았음을 유추할 수 있습니다.

아래 실행에서는 pat size를 50byte(50개의 char)로 고정한 후, 각 string size별 소요시간 결과입니다. 이 때 string과 pat은 모든 실행에서 새로운 string을 사용하였습니다.

```
Case 1) string size: 1MB (1e6개) / patsize : 50byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 1.7ms
nfind   평균 소요시간: 4.9ms
pmatch  평균 소요시간: 4.8ms

Case 2) string size: 5MB (5e6개) / patsize : 50byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 7.1ms
nfind   평균 소요시간: 24.5ms
pmatch  평균 소요시간: 21.0ms

Case 3) string size: 10MB (1e7개) / patsize : 50byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 13.5ms
nfind   평균 소요시간: 46.1ms
pmatch  평균 소요시간: 44.0ms

Case 4) string size: 20MB (2e7개) / patsize : 50byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 25.9ms
nfind   평균 소요시간: 80.9ms
pmatch  평균 소요시간: 73.6ms

Case 5) string size: 30MB (3e7개) / patsize : 50byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 35.6ms
nfind   평균 소요시간: 114.9ms
pmatch  평균 소요시간: 109.1ms

Case 6) string size: 40MB (4e7개) / patsize : 50byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 41.4ms
nfind   평균 소요시간: 135.1ms
pmatch  평균 소요시간: 139.9ms

Case 7) string size: 50MB (5e7개) / patsize : 50byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 58.4ms
nfind   평균 소요시간: 177.6ms
pmatch  평균 소요시간: 152.8ms
```

다음은 string size를 10MB(1e7개의 char)로 고정한 후, 각 pat size별 소요시간 결과입니다. 이 때 string과 pat은 모든 실행에서 새로운 string을 사용하였습니다.

```
Case 1) string size: 10MB (1e7개) / patsize : 10byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 9.9ms
nfind   평균 소요시간: 38.8ms
pmatch  평균 소요시간: 37.2ms

Case 2) string size: 10MB (1e7개) / patsize : 50byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 11.0ms
nfind   평균 소요시간: 40.8ms
pmatch  평균 소요시간: 34.9ms

Case 3) string size: 10MB (1e7개) / patsize : 100byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 9.4ms
nfind   평균 소요시간: 35.5ms
pmatch  평균 소요시간: 31.0ms

Case 4) string size: 10MB (1e7개) / patsize : 200byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 10.1ms
nfind   평균 소요시간: 37.2ms
pmatch  평균 소요시간: 36.8ms

Case 5) string size: 10MB (1e7개) / patsize : 300byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 11.6ms
nfind   평균 소요시간: 37.3ms
pmatch  평균 소요시간: 32.7ms

Case 6) string size: 10MB (1e7개) / patsize : 400byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 9.7ms
nfind   평균 소요시간: 36.4ms
pmatch  평균 소요시간: 35.1ms

Case 7) string size: 10MB (1e7개) / patsize : 500byte.
Pattern count in string: 0.0
strstr  평균 소요시간: 11.1ms
nfind   평균 소요시간: 36.6ms
pmatch  평균 소요시간: 33.0ms
```

② Specific string

이번 실행에서는 pat을 "Hello World! I love C program. Hello World! 50len."로 고정한 후, 각 사이즈에 맞게 pat을 반복하여 string을 만든 후 실행한 결과입니다.

```
Case 1) string size: 1MB (1e6개) / patsize : 50byte.
Pattern count in string: 20000.0
strstr  평균 소요시간: 1.8ms
nfind   평균 소요시간: 6.8ms
pmatch  평균 소요시간: 4.0ms

Case 2) string size: 5MB (5e6개) / patsize : 50byte.
Pattern count in string: 100000.0
strstr  평균 소요시간: 8.4ms
nfind   평균 소요시간: 35.0ms
pmatch  평균 소요시간: 19.8ms

Case 3) string size: 10MB (1e7개) / patsize : 50byte.
Pattern count in string: 200000.0
strstr  평균 소요시간: 15.6ms
nfind   평균 소요시간: 63.7ms
pmatch  평균 소요시간: 35.0ms

Case 4) string size: 20MB (2e7개) / patsize : 50byte.
Pattern count in string: 400000.0
strstr  평균 소요시간: 23.0ms
nfind   평균 소요시간: 118.9ms
pmatch  평균 소요시간: 72.6ms

Case 5) string size: 30MB (3e7개) / patsize : 50byte.
Pattern count in string: 600000.0
strstr  평균 소요시간: 36.0ms
nfind   평균 소요시간: 158.0ms
pmatch  평균 소요시간: 98.6ms

Case 6) string size: 40MB (4e7개) / patsize : 50byte.
Pattern count in string: 800000.0
strstr  평균 소요시간: 44.4ms
nfind   평균 소요시간: 205.5ms
pmatch  평균 소요시간: 126.1ms

Case 7) string size: 50MB (5e7개) / patsize : 50byte.
Pattern count in string: 1000000.0
strstr  평균 소요시간: 54.7ms
nfind   평균 소요시간: 258.9ms
pmatch  평균 소요시간: 161.8ms
```

다음은 string을 "Hello World! I love C program. Hello World! 50len." × 200,000인 10MB 문자열로 고정하였습니다. 이 후, pat도 위의 문자열을 반복하여 각 사이즈에 맞게 만든 후 실행한 결과입니다. (10byte string은 만들기가 애매하여 실행을 하지 않았습니다.)

```
Case 1) string size: 10MB (1e7개) / patsize : 50byte.
Pattern count in string: 200000.0
strstr  평균 소요시간: 14.8ms
nfind   평균 소요시간: 59.8ms
pmatch  평균 소요시간: 35.4ms

Case 2) string size: 10MB (1e7개) / patsize : 100byte.
Pattern count in string: 199999.0
strstr  평균 소요시간: 18.6ms
nfind   평균 소요시간: 85.5ms
pmatch  평균 소요시간: 32.7ms

Case 3) string size: 10MB (1e7개) / patsize : 200byte.
Pattern count in string: 199997.0
strstr  평균 소요시간: 29.7ms
nfind   평균 소요시간: 131.2ms
pmatch  평균 소요시간: 31.2ms

Case 4) string size: 10MB (1e7개) / patsize : 300byte.
Pattern count in string: 199995.0
strstr  평균 소요시간: 38.0ms
nfind   평균 소요시간: 178.6ms
pmatch  평균 소요시간: 29.5ms

Case 5) string size: 10MB (1e7개) / patsize : 400byte.
Pattern count in string: 199993.0
strstr  평균 소요시간: 46.0ms
nfind   평균 소요시간: 214.0ms
pmatch  평균 소요시간: 34.2ms

Case 6) string size: 10MB (1e7개) / patsize : 500byte.
Pattern count in string: 199991.0
strstr  평균 소요시간: 54.6ms
nfind   평균 소요시간: 258.3ms
pmatch  평균 소요시간: 31.6ms
```

③ One alphabet string

이번 실행에서는 pat과 string을 "a"하나로 만든 문자열을 사용하였습니다. 아래는 pat = "a" × 50로 고정하였고, string을 각 string size에 맞게 "a"를 반복하여 만든 후 실행한 결과입니다.

```
Case 1) string size: 1MB (1e6개) / patsize : 50byte.
Pattern count in string: 999951.0
strstr  평균 소요시간: 35.6ms
nfind   평균 소요시간: 146.9ms
pmatch  평균 소요시간: 6.5ms

Case 2) string size: 5MB (5e6개) / patsize : 50byte.
Pattern count in string: 4999951.0
strstr  평균 소요시간: 141.2ms
nfind   평균 소요시간: 646.7ms
pmatch  평균 소요시간: 36.4ms

Case 3) string size: 10MB (1e7개) / patsize : 50byte.
Pattern count in string: 9999951.0
strstr  평균 소요시간: 262.2ms
nfind   평균 소요시간: 1229.3ms
pmatch  평균 소요시간: 73.1ms

Case 4) string size: 20MB (2e7개) / patsize : 50byte.
Pattern count in string: 19999951.0
strstr  평균 소요시간: 500.3ms
nfind   평균 소요시간: 2439.4ms
pmatch  평균 소요시간: 127.6ms

Case 5) string size: 30MB (3e7개) / patsize : 50byte.
Pattern count in string: 29999951.0
strstr  평균 소요시간: 715.9ms
nfind   평균 소요시간: 3738.4ms
pmatch  평균 소요시간: 187.9ms

Case 6) string size: 40MB (4e7개) / patsize : 50byte.
Pattern count in string: 39999951.0
strstr  평균 소요시간: 970.6ms
nfind   평균 소요시간: 4840.3ms
pmatch  평균 소요시간: 228.2ms

Case 7) string size: 50MB (5e7개) / patsize : 50byte.
Pattern count in string: 49999951.0
strstr  평균 소요시간: 1200.0ms
nfind   평균 소요시간: 6170.0ms
pmatch  평균 소요시간: 283.8ms
```

아래는 string을 "a" × 1e7로 고정한 후, pat을 각 pat size에 맞게 "a"를 반복하여 만든 후 실행한 결과입니다.

```
Case 1) string size: 10MB (1e7개) / patsize : 10byte.
Pattern count in string: 9999991.0
strstr  평균 소요시간: 90.2ms
nfind   평균 소요시간: 223.4ms
pmatch  평균 소요시간: 59.1ms

Case 2) string size: 10MB (1e7개) / patsize : 50byte.
Pattern count in string: 9999951.0
strstr  평균 소요시간: 235.2ms
nfind   평균 소요시간: 1133.6ms
pmatch  평균 소요시간: 56.3ms

Case 3) string size: 10MB (1e7개) / patsize : 100byte.
Pattern count in string: 9999901.0
strstr  평균 소요시간: 418.8ms
nfind   평균 소요시간: 2220.3ms
pmatch  평균 소요시간: 60.4ms

Case 4) string size: 10MB (1e7개) / patsize : 200byte.
Pattern count in string: 9999801.0
strstr  평균 소요시간: 898.1ms
nfind   평균 소요시간: 4272.5ms
pmatch  평균 소요시간: 59.4ms

Case 5) string size: 10MB (1e7개) / patsize : 300byte.
Pattern count in string: 9999701.0
strstr  평균 소요시간: 1291.0ms
nfind   평균 소요시간: 6438.9ms
pmatch  평균 소요시간: 61.7ms

Case 6) string size: 10MB (1e7개) / patsize : 400byte.
Pattern count in string: 9999601.0
strstr  평균 소요시간: 1696.1ms
nfind   평균 소요시간: 8484.2ms
pmatch  평균 소요시간: 58.7ms

Case 7) string size: 10MB (1e7개) / patsize : 500byte.
Pattern count in string: 9999501.0
strstr  평균 소요시간: 2133.4ms
nfind   평균 소요시간: 10710.0ms
pmatch  평균 소요시간: 58.1ms
```

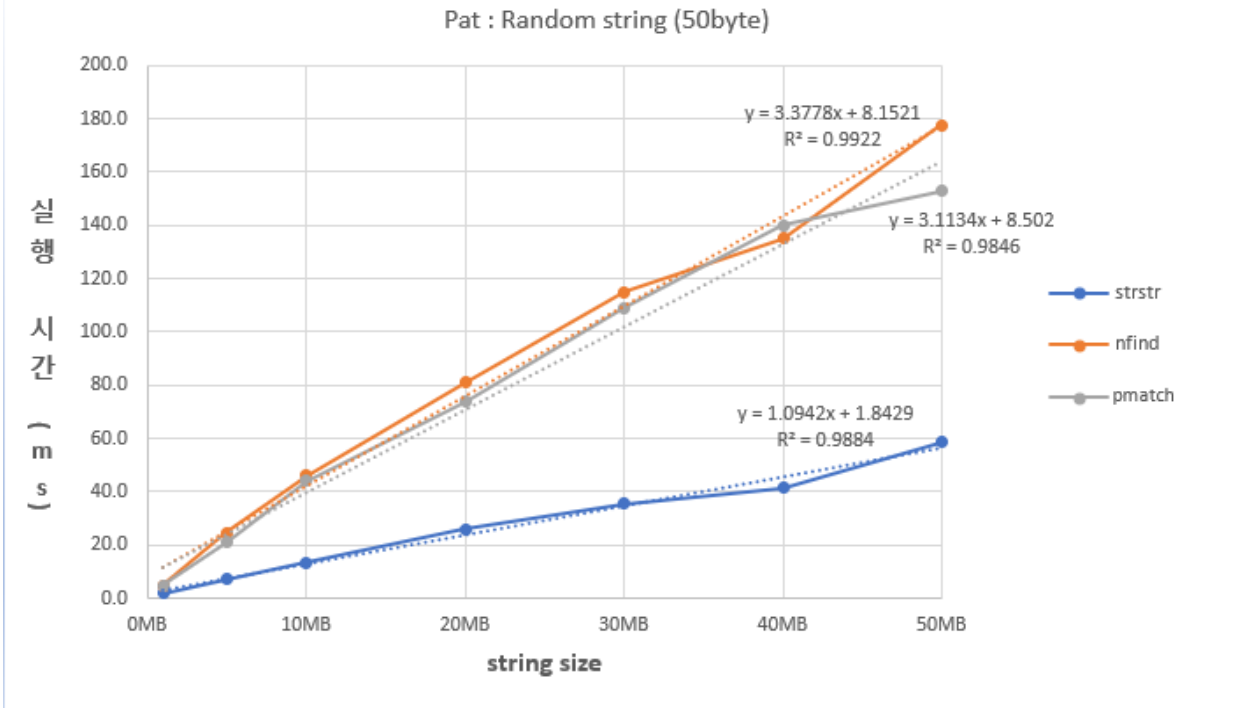
4. 결과 처리

아래는 위의 결과들을 표와 그래프로 정리한 것입니다.

① Random string

먼저 pat을 고정한 결과입니다.

	1MB	5MB	10MB	20MB	30MB	40MB	50MB
strstr	1.7	7.1	13.5	25.9	35.6	41.4	58.4
nfind	4.9	24.5	46.1	80.9	114.9	135.1	177.6
pmatch	4.8	21.0	44.0	73.6	109.1	139.9	152.8



위 그래프에서 3개의 함수들이 모두 1차함수 형태를 띄는 것을 볼 수 있습니다.

strstr function과 nfind function의 시간복잡도는 O(mn), pmatch function의 시간복잡도가 O(m+n는 입니다. 이 때, n은 50으로 고정되어 있고, m이 n에 비해 매우 크기 때문에 세 함수 모두 O(m)의 시간 복잡도를 가지게 된다고 볼 수 있습니다.

여기서 random으로 문자를 생성할 경우, pat count가 모두 0이 나오는 것을 위 결과를 통해 볼 수 있었습니다. nfind function내에 아래와 같은 구문이 있습니다.

```
for (j=0,i=start ; j<lastp&&string[i]==pat[j] ; i++,j++)
```

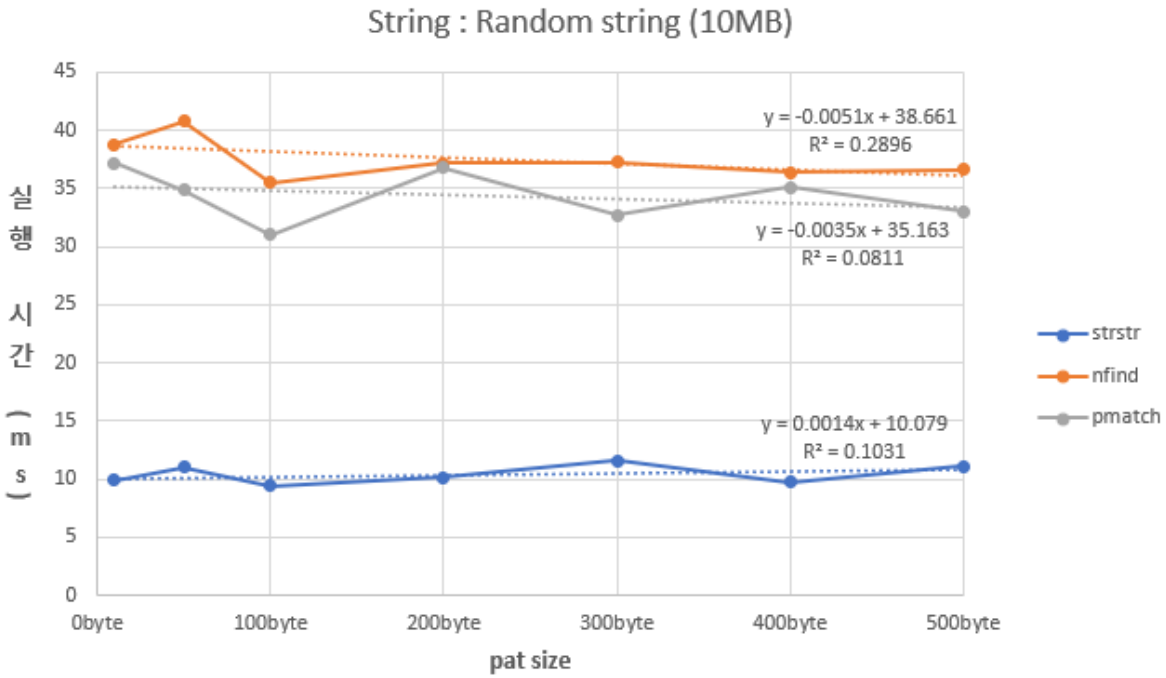
이 구문에서 `string[i]`와 `pat[j]`가 다른 경우가 많아서 for문을 빠르게 빠져나오게 됩니다. 따라서 시간복잡도인 O(mn)에서 n이 1과 근사한 값을 가지게 되고, nfind function과 pmatch function 매우 유사한 속도를 보이게 되는 것을 알 수 있습니다. 그래프에서 실제로 큰 차이가

없는 것을 확인할 수 있습니다.

strstr function도 위의 이유로 pmatch function과 유사한 속도를 보일 것 같지만, 사전 이론에서 언급된 내용처럼 strstr function은 assembly 언어로 최적화 되어있기에 위 경우에서 가장 빠른 속도를 띄는 것을 볼 수 있습니다.

그 다음은 string을 고정한 결과입니다.

	10byte	50byte	100byte	200byte	300byte	400byte	500byte
strstr	9.9	11.0	9.4	10.1	11.6	9.7	11.1
nfind	38.8	40.8	35.5	37.2	37.3	36.4	36.6
pmatch	37.2	34.9	31.0	36.8	32.7	35.1	33.0



여기서는 세 함수가 모두 상수함수를 띄는 것을 것 수 있습니다.

strstr function과 nfind function의 시간복잡도는 O(mn), pmatch function의 시간복잡도는 O(m+n) 입니다. (m은 1e7의 값으로 고정) 이 때, nfind function내에 아래와 같은 구문이 있습니다.

```
for (j=0,i=start ; j<lastp&&string[i]==pat[j] ; i++,j++)
```

이 때 `string[i]`와 `pat[j]`이 같은 경우가 거의 없어서 for문을 계속 실행시키지 않게 되어 n이 0이 가까운 값을 가지게 됩니다. 따라서 nfind function과 이와 똑같은 시간복잡도를 가진 strstr function이 상수함수를 띄게 됩니다.

그리고 m이 n에 비해 매우 크기 때문에 pmatch function은 n의 크기에 상관없이 항상 비슷한

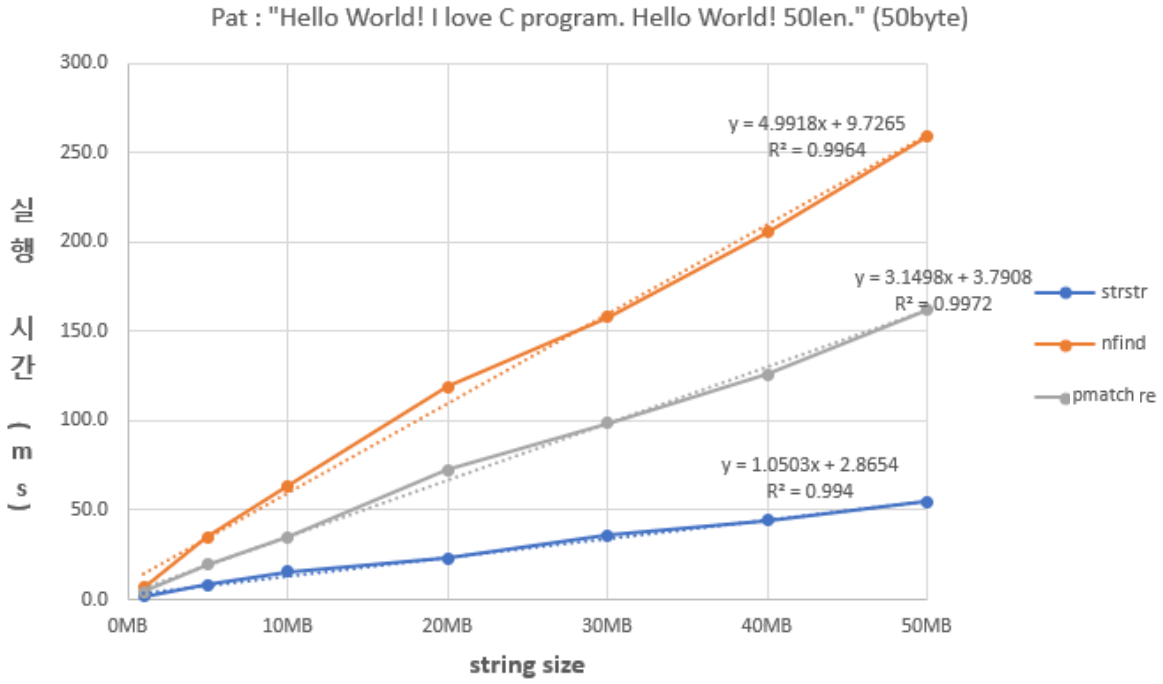
시간을 가져서 상수함수를 띄는 것을 알 수 있습니다.

위 경우에서도 strstr function이 pmatch function보다 빠른 것을 볼 수 있는데, 이는 위의 이유와 동일하게 strstr function이 assembly 언어로 최적화 되어있기 때문입니다.

② Specific string

먼저 pat을 고정한 결과입니다.

	1MB	5MB	10MB	20MB	30MB	40MB	50MB
strstr	1.8	8.4	15.6	23.0	36.0	44.4	54.7
nfind	6.8	35.0	63.7	118.9	158.0	205.5	258.9
pmatch	4.0	19.8	35.0	72.6	98.6	126.1	161.8



위 그래프에서 3개의 함수들이 모두 1차함수 형태를 띄는 것을 볼 수 있습니다. 이는 random string일 때의 이유와 동일합니다. 전체적으로 함수 실행 시간이 늘어난 것을 확인할 수 있는데, 이는 string 문자열을 pat의 반복을 이용하여 만들었기에 탐색할 문자들의 수가 많아졌기 때문입니다.

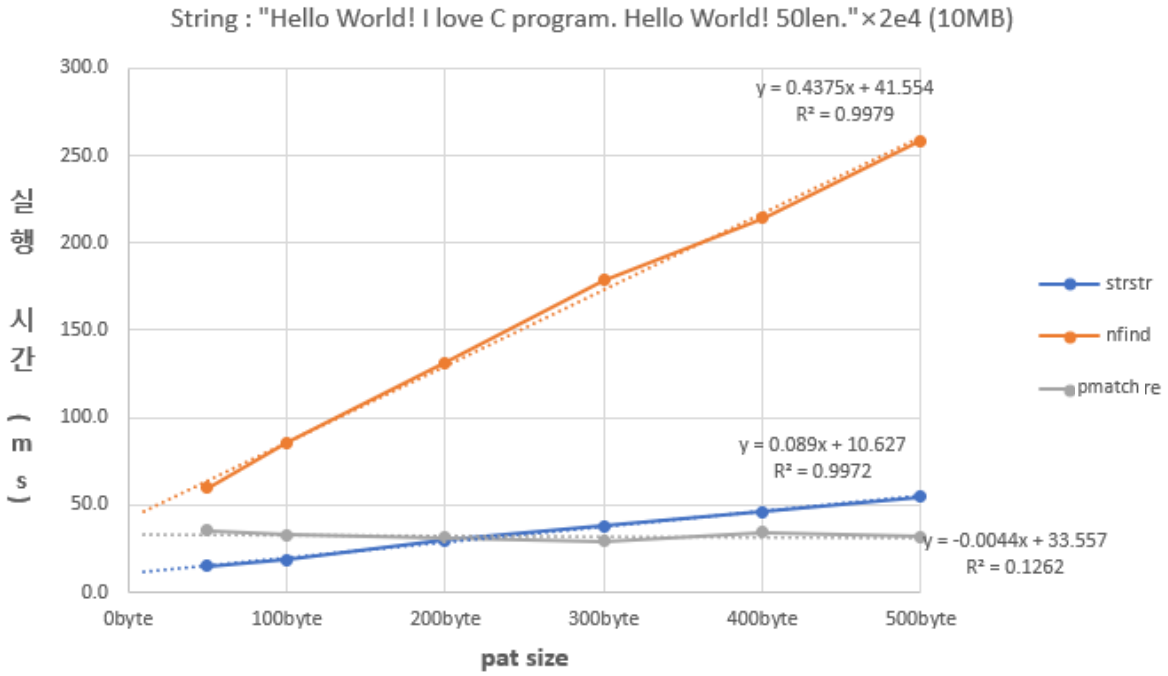
random string일 때와 다르게 nfind function과 pmatch function의 속도가 차이나는 것을 확인할 수 있는데, random string일 때는 문자열을 만들 때 규칙성이 거의 없었고, string안에 pat이 있지 않았습니다. 하지만 specific string은 pat의 규칙성이 있고, string에 pat이 있기에 nfind의 시간복잡도인 O(mn)에서 n이 1보다 큰 값을 가지게 됩니다. 따라서, pmatch function의 시간복잡

도가 O(m+n)과 비교했을 때 상대적으로 느린 속도로 함수가 실행되는 것을 분석할 수 있습니다.

다만 이 경우에도 마찬가지로 strstr function보다는 pmatch function이 더 느린 것을 확인할 수 있습니다.

다음은 string을 고정한 결과입니다.

	50byte	100byte	200byte	300byte	400byte	500byte
strstr	14.8	18.6	29.7	38.0	46.0	54.6
nfind	59.8	85.5	131.2	178.6	214.0	258.3
pmatch	35.4	32.7	31.2	29.5	34.2	31.6



여기서는 strstr function과 nfind function이 일차함수, pmatch function이 상수함수를 띄는 것을 것 수 있습니다.

strstr function과 nfind function의 시간복잡도는 O(mn), pmatch function의 시간복잡도는 O(m+n)입니다. (m은 1e7의 값으로 고정) 이 때, nfind function내에 아래와 같은 구문이 있습니다.

```
for (j=0,i=start ; j<lastp&&string[i]==pat[j] ; i++,j++)
```

specific string의 경우는 `string[i]`와 `pat[j]`이 같은 경우가 길이에 비례하여 생기게 됩니다. 따라서 n에 비례하는 O(n)의 시간복잡도를 가지게 되고, nfind function과 이와 똑같은 시간복잡도를 가진 strstr function이 일차함수를 띄게 됩니다.

그리고 m이 n에 비해 매우 크기 때문에 pmatch function은 n의 크기에 상관없이 항상 비슷한 시간을 가져서 상수함수를 띄는 것을 알 수 있습니다.

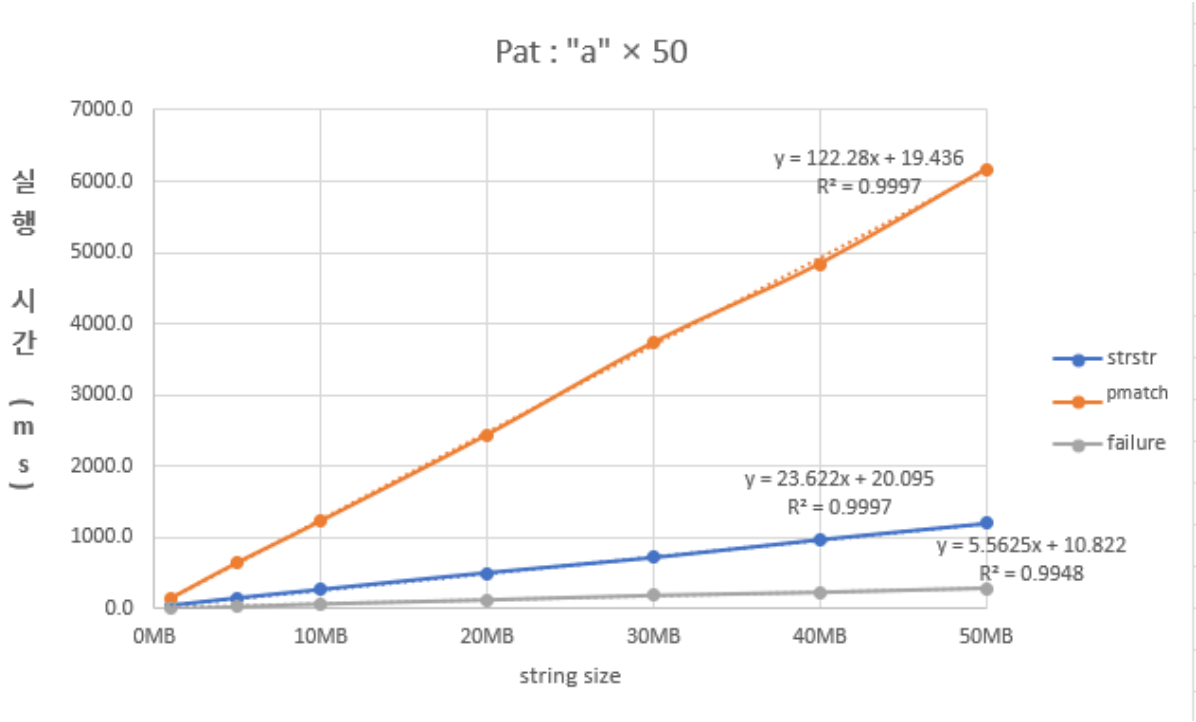
위 경우에서도 strstr function이 pmatch function보다 빠른 것을 볼 수 있는데, 이는 위의 이유와 동일하게 strstr function이 assembly 언어로 최적화 되어있기 때문입니다.

(이는 random string의 경우와 동일합니다.)

③ One alphabet string

먼저 pat을 고정한 경우입니다.

	1MB	5MB	10MB	20MB	30MB	40MB	50MB
strstr	35.6	141.1	262.2	500.3	715.9	970.6	1200.0
nfind	146.9	646.7	1229.3	2439.4	3738.4	4840.3	6170.0
pmatch	6.5	36.4	73.1	127.6	187.9	228.2	283.8

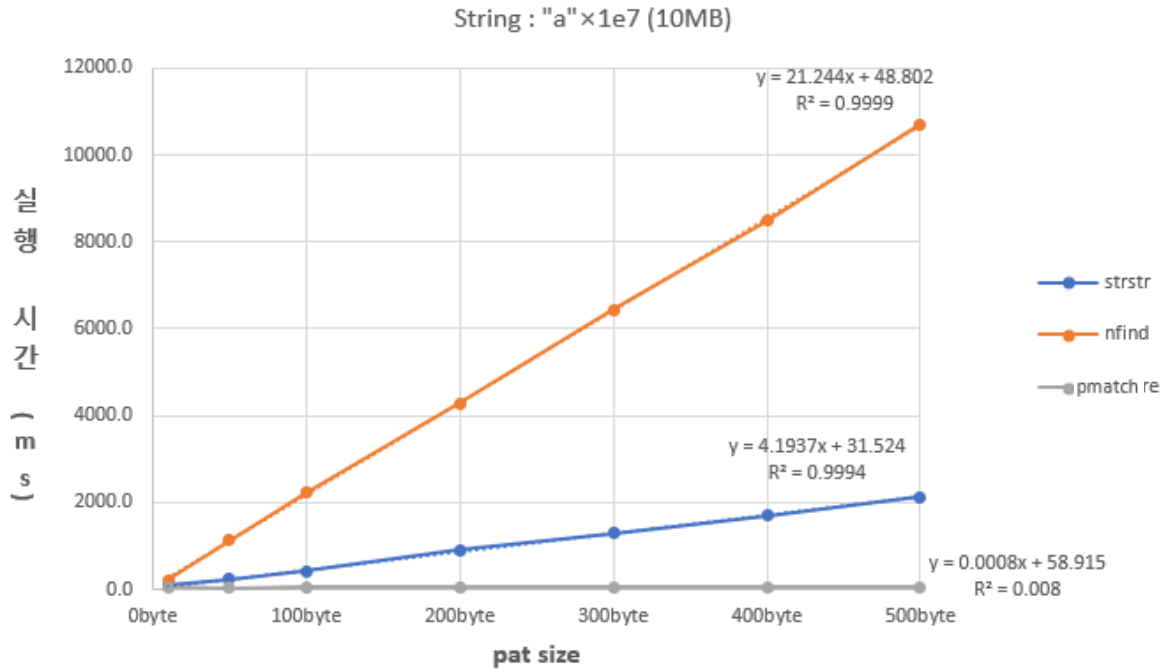


위 그래프에서 3개의 함수들이 모두 1차함수 형태를 띄는 것을 볼 수 있습니다. 이는 random string일 때의 이유와 동일합니다. 전체적으로 함수 실행 시간이 많이 늘어난 것을 확인할 수 있는데, 이는 string 문자열과 pat 문자열을 "a"라는 하나의 문자를 반복하여 만들었기에 탐색할 문자들의 수가 훨씬 많아졌기 때문입니다.

specific string일 때와 또 다르게 nfind function과 pmatch function의 속도가 다를 뿐만 아니라 strstr function보다 빠른 것을 확인할 수 있는데, strstr function일 때와 nfind function일 때 거의 모든 경우를 탐색하기 때문입니다. (worst case) 따라서 시간복잡도인 O(mn)에서 n이 매우 큰 값을 가지게 되므로 pmatch function의 시간복잡도가 O(m+n)과 비교했을 때 크게 느린 속도를 보이게 됩니다.

다음은 string을 고정한 경우입니다.

	10byte	50byte	100byte	200byte	300byte	400byte	500byte
strstr	90.2	235.2	418.8	898.1	1291.0	1696.1	2133.4
nfind	223.4	1133.6	2220.3	4272.5	6438.9	8484.2	10710.0
pmatch	59.1	56.3	60.4	59.4	61.7	58.7	58.1



여기서도 동일하게 strstr function과 nfind function이 일차함수, pmatch function이 상수함수를 띄는 것을 것 수 있습니다.

strstr function과 nfind function의 시간복잡도는 O(mn), pmatch function의 시간복잡도는 O(m+n)입니다. (m은 1e7의 값으로 고정) 이 때, nfind function내에 아래와 같은 구문이 있습니다.

```
for (j=0,i=start ; j<lastp&&string[i]==pat[j] ; i++,j++)
```

specific string의 경우는 `string[i]`와 `pat[j]`이 같은 경우가 길이에 비례하여 생기게 됩니다. 따라서 n에 비례하는 O(n)의 시간복잡도를 가지게 되고, nfind function과 이와 똑같은 시간복잡도를 가진 strstr function이 일차함수를 띄게 됩니다.

(specific string의 경우와 동일합니다.)

그리고 m이 n에 비해 매우 크기 때문에 pmatch function은 n의 크기에 상관없이 항상 비슷한 시간을 가져서 상수함수를 띄는 것을 알 수 있습니다. 특히 이 경우에는 strstr function과 nfind function 보다 훨씬 빠른 속도를 띄는 것을 볼 수 있습니다.