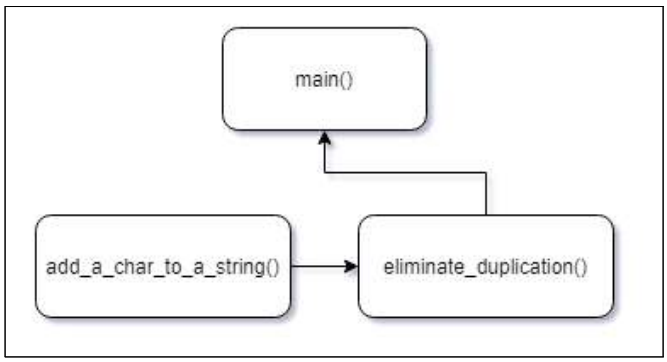
20121277 김주호 HW2 보고서

1번.

- 함수 구성 다이어그램



- 함수 설명

① void add_a_char_to_a_string()

string 문자열과, char 하나를 인자로 받아서, 인자로 들어온 string 문자열 맨 뒤에 char 하나를 뒤에 붙여준다.

input : char * str, char c

output : 없음

② void eliminate_duplication()

target이 되는 str을 입력 받고, 중복되는 문자를 제거한 뒤에 이를 new_string에 새기는 함수이다. 작동하는 방식에 대한 간단한 description을 함수명 아래 주석으로 설명하였다.

input : char * str, char * new_string

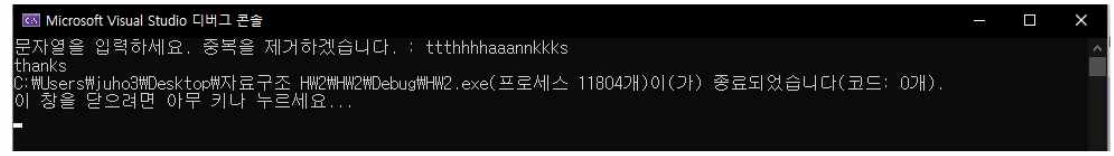
output : 없음

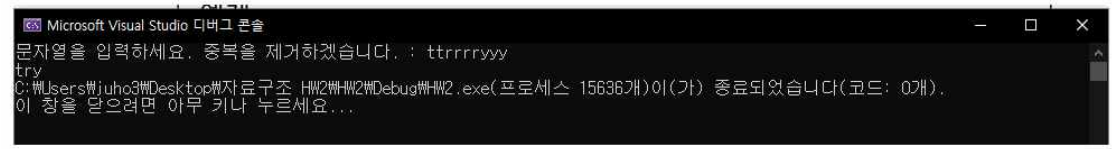
- 코드

```
#include <stdio.h >
#include <stdlib.h >
#include <string.h >
#define max_str_length 100

void add_a_char_to_a_string(char * str, char c) {
        char * p = str;
        while (*p !='₩0') { // 문자열 끝 탐색
                p++;
        }
        *p = c;
```

```
        *(p +1) ='₩0';
}

void eliminate_duplication(char * str, char * new_string) {
        /* string을 앞부터 순회하면서 char를 tmp에 담고,
        'tmp에 담겨있는 char'와 다른 내용의 char를 만난다면,
        char를 꺼내어 new_string에 대입한다.*/
        char tmp;
        for (int i =0; str[i] !=0; i ++) {
                if (i ==0) {
                        tmp = str[i];
                        add_a_char_to_a_string(new_string, str[i]);
                }
                else if (str[i -1] != str[i]) {
                        tmp = str[i];
                        add_a_char_to_a_string(new_string, str[i]);
                }
        }
}

int main() {
        char str[max_str_length];
        char new_string[max_str_length] = { 0, };
        int len_of_str;
        printf("문자열을 입력하세요. 중복을 제거하겠습니다. : ");
        if (scanf("%s", str) ==-1) {
                return 0;
        }
        str[max_str_length -1] =0;
        len_of_str = strlen(str);
        if (len_of_str <2 || len_of_str >30) {
                printf("잘못된 입력입니다.₩n");
                exit(1);
        }
        eliminate_duplication(str, new_string);
        printf("%s", new_string);
        return 0;
}
```

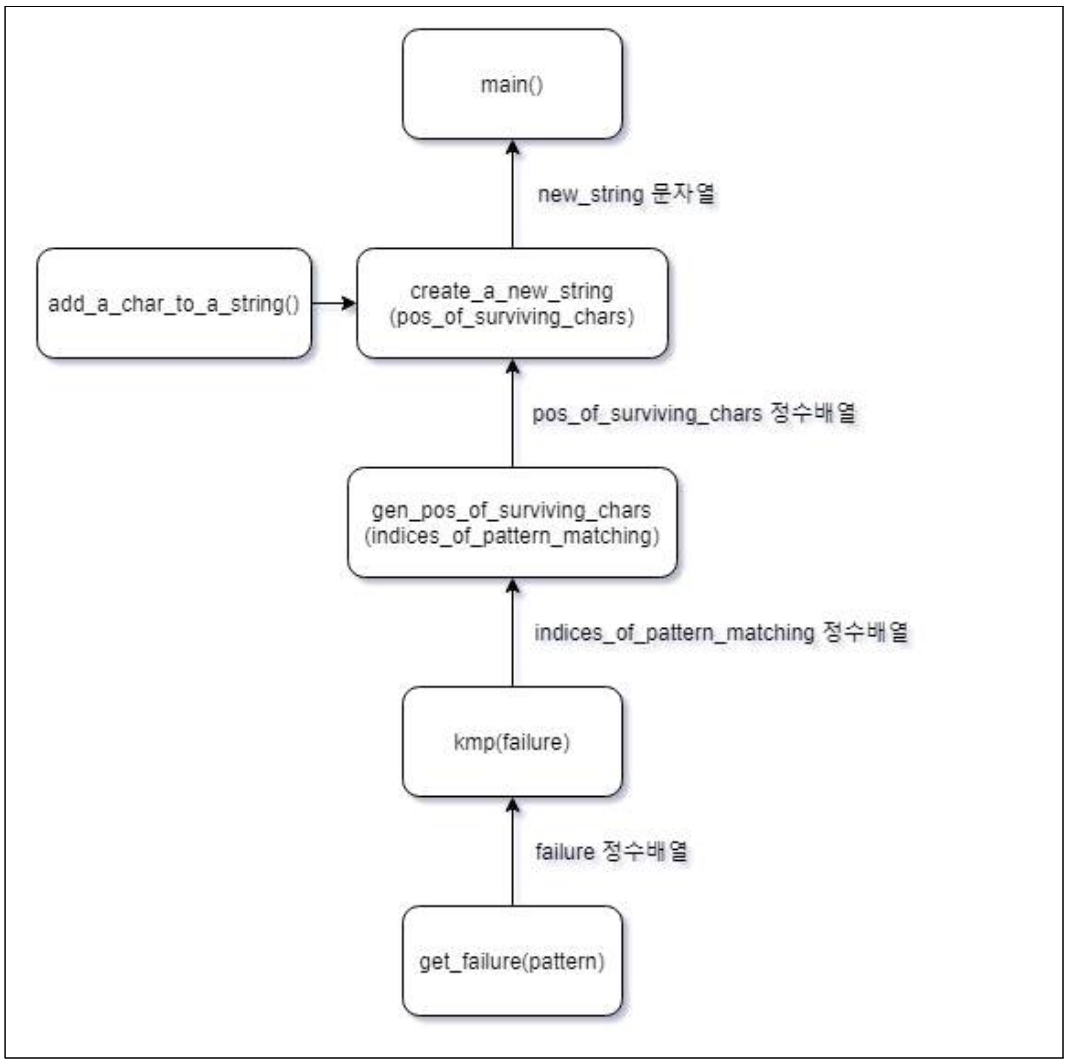
- 실행 결과





2번.

- 함수 구성 다이어그램



- **함수 설명**

① void get_failure()

pattern 문자열을 순회하면서, failure 정수 배열을 얻어내는 함수이다. 사용 편의를 위하여, failure 배열의 원소를 모두 1씩 증가시켰다.

input : int * failure, char * pattern

return : 없음

② void kmp()

get_failure()에서 만든 failure 배열을 인자로 받아서, 이를 이용하여, indices_of_pattern_matching_size 숫자 배열을 생성하는 데에 목적이 있는 함수이다. 예를 들어, bbbb 라는 문자열에, bbb 패턴을 매칭한다면, {0,1}을 생성하는 함수이다.

input : int * failure, int * indices_of_pattern_matching,
int * indices_of_pattern_matching_size, char * str, char * pattern

return : 없음

③ void gen_pos_of_surviving_chars()

kmp()에서 만든 배열 indices_of_pattern_matching_size을 인자로 받아서, 이를 활용하여, pos_of_surviving_chars 숫자 배열을 생성하는 데에 목적이 있는 함수이다. 예를 들어, bbbabbbc 라는 문자열에 bbb 패턴을 매칭한다면, {0,0,0,1,0,0,0,1}을 생성한다.

input : int * indices_of_pattern_matching, int * indices_of_pattern_matching_size, int * pos_of_surviving_chars, char * str, char * pattern

output : 없음

④ void add_a_char_to_a_string()

1번 문제를 해결할 때, 사용한 함수와 같다. string 문자열과, char 하나를 인자로 받아서, 인자로 들어온 string 문자열 맨 뒤에 char 하나를 뒤에 붙여준다.

⑤ void create_a_new_string()

gen_pos_of_surviving_chars() 함수를 이용하여 구한, pos_of_surviving_chars를 인자로 받아와서, new_string을 생성하는 함수이다. 삭제 뒤에 살아남은 char들의 위치가 배열 pos_of_surviving_chars에 마킹이 되어있어, 마킹이 된 위치의 char를 new_string에 대입한다.

input : char * new_str, char * str, int * pos_of_surviving_chars

output : 없음

* ② 함수와 ③ 함수를 만들어 new_string을 생성한 이유.
②의 과정을 거치지 않고 곧바로 ③ 함수를 생성할 수도 있었고, 더 나아가 ②, ③, ⑤를 하나의 과정으로 엮어 new_string을 생성할 수 있었으나 이를 분리하였다. ②, ③의 과정을 넣음으로써, 새로운 배열을 생성하게 하여 메모리가 커지는 희생이 있었지만, 이를 분리한 이유는 '확장성' 때문이다. ②의 함수가 뒤의 3번 문제를 해결하는 데에 기능하고, ③함수 역시 matcing되는 문자를 삭제하는 문제가 아닌 "다른 문자로 치환하는 문제"같은 것을 해결할 때 유용하게 이용될 것이다.

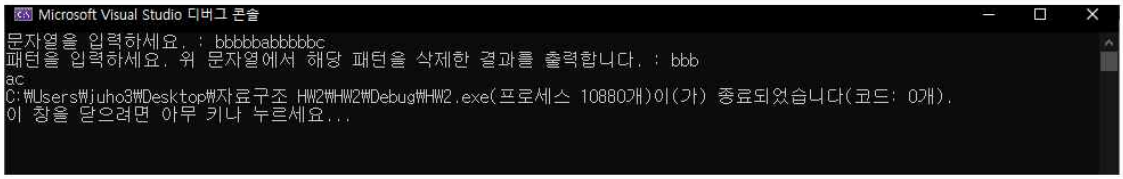
- 코드

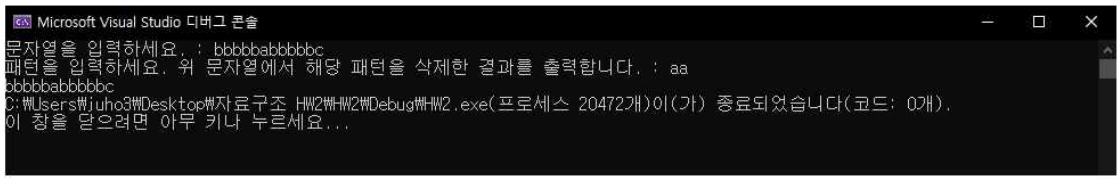
```
#include <stdio.h >
#include <stdlib.h >
#include <string.h >
#define max_string_size 30
#define max_pattern_size 30

void get_failure(int * failure, char * pattern) {
        /* 입력받은 pattern에 대한 failure 배열을 만드는 함수 */
        int i, n = strlen(pattern);
        failure[0] =-1;
        for (int j =1; j < n; j ++) {
                i = failure[j -1];
                while ((pattern[j] != pattern[i +1]) && (i >=0))
                        i = failure[i];
                if (pattern[j] == pattern[i +1])
                        failure[j] = i +1;
                else  failure[j] =-1;
        }

        for (int i =0; i < n; i ++) { // 사용의 편의를 위하여 원소를 모두 1씩 증가시
킴
                failure[i] ++;
        }
}

void kmp(int * failure, int * indices_of_pattern_matching, int *
indices_of_pattern_matching_size, char * str, char * pattern) {
        /* indices_of_pattern_matching 배열을 만드는 것이 목적인 함수이다.
        string에서 pattern에 매칭이되는 시작 index들을 원소로 저장한다.
        ex) 입력 : bbbbabbbc / bbb 에 대해서 {0, 1, 5} 라는 배열을 만듦.*/
        int len_of_string = strlen(str), len_of_pattern = strlen(pattern);
        int begin, matched;

        begin =0;
        matched =0;
        while (begin <= len_of_string - len_of_pattern) {
                if (matched < len_of_pattern && str[begin + matched] ==
pattern[matched]) {
                        matched +=1;
                        if (matched == len_of_pattern) {
```

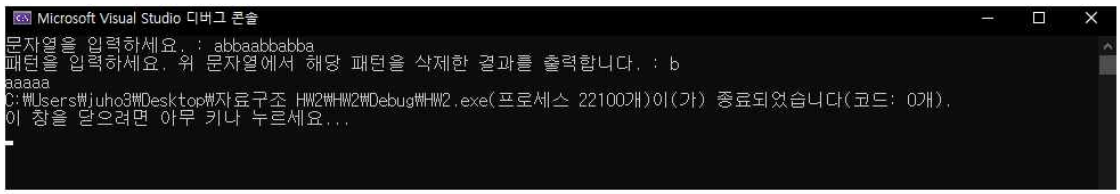
```
indices_of_pattern_matching[(*indices_of_pattern_matching_size)++] =begin;
                        }
                }
                else {
                        if (matched ==0) {
                                begin +=1;
                        }
                        else {
                                begin += matched - failure[matched -1];
                                matched = failure[matched -1];
                        }
                }
        }
}

void    gen_pos_of_surviving_chars(int    *    indices_of_pattern_matching,    int    *
indices_of_pattern_matching_size, int * pos_of_surviving_chars, char * str, char *
pattern) {
        /* pos_of_surviving_chars 배열을 생성하는 것이 목적인 함수이다.
        pattern에 matching이 되어 지워질 char의 index 위치를 0으로 마킹하고,
        살아남은 char(surviving char를 의미. 지워지지 않고 살아남은 char)의 index
위치를 1로 마킹한다.
        ex) 입력 : bbbbabbbc / bbb 에 대해 {0,0,0,0,1,0,0,0,1}이라는 배열을 만
듦.*/
        int matching_ind;
        int len_of_string = strlen(str);
        int len_of_pattern = strlen(pattern);
        for (int i =0; i < len_of_string; i ++) { // 기본적으로 모두 1로 마킹해둔다.
                pos_of_surviving_chars[i] =1;
        }
        for (int i =0;i <*indices_of_pattern_matching_size; i ++) { // 지워질 char의
index 위치를 0으로 마킹한다.
                matching_ind = indices_of_pattern_matching[i];
                for (int j = matching_ind;j <matching_ind + len_of_pattern; j ++) {
                        pos_of_surviving_chars[j] =0;
                }
        }
}

void add_a_char_to_a_string(char * str, char c) {
        char * p = str;
        while (*p !='₩0') { // 문자열 끝 탐색
                p++;
        }
        *p = c;
        *(p +1) ='₩0';
}

void create_a_new_string(char * new_str, char * str, int * pos_of_surviving_chars) {
        /* string에서 pattern에 매칭되는 부분을 삭제한, new_string을 생성한다. */
        int size_of_string = strlen(str);
        for (int i =0; i < size_of_string; i ++) {
                if    (pos_of_surviving_chars[i]    ==1)    {    //    살아남은    char만을
```

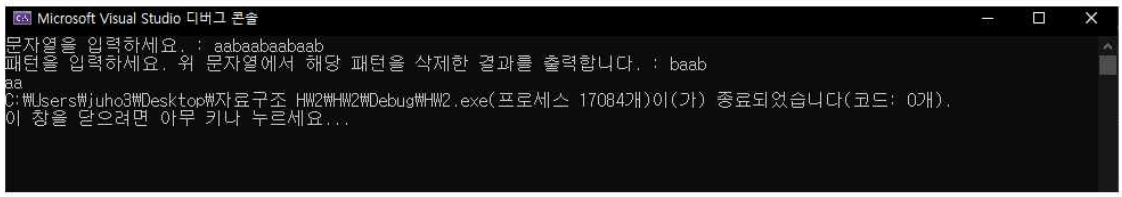
```
new_string에 대입한다.
                        add_a_char_to_a_string(new_str, str[i]);
                }
        }
}

int main() {
        char str[max_string_size];
        char pattern[max_pattern_size];
        char new_str[max_string_size] = {0,};
        int failure[max_pattern_size];
        int indices_of_pattern_matching[max_string_size];
        int indices_of_pattern_matching_size =0;
        int pos_of_surviving_chars[max_pattern_size];

        int len_of_str;
        int len_of_pat;
        printf("문자열을 입력하세요. : ");
        if (scanf("%s", str) ==-1) {
                return 0;
        }
        printf("패턴을 입력하세요. 위 문자열에서 해당 패턴을 삭제한 결과를 출력합니다. : ");
        if (scanf("%s", pattern) ==-1) {
                return 0;
        }
        str[max_string_size -1] =0;
        pattern[max_pattern_size -1] =0; // str과 pattern이 널문자로 끝나지 않을 수도 있다는 경고 없애기
        len_of_str = strlen(str);
        len_of_pat = strlen(pattern);
        if (len_of_str >=30 || len_of_str <1 ||
                len_of_pat >=10 || len_of_pat <1
                || len_of_str < len_of_pat) { // 예외 처리
                printf("잘못된 입력입니다.");
                exit(1);
        }
        get_failure(failure, pattern); // failure 배열 생성
        kmp(failure, indices_of_pattern_matching, &indices_of_pattern_matching_size, str, pattern); // indices_of_pattern_matching 배열 생성
        gen_pos_of_surviving_chars(indices_of_pattern_matching, &indices_of_pattern_matching_size, pos_of_surviving_chars, str, pattern);
        //  pos_of_surviving_chars 배열 생성.
        create_a_new_string(new_str, str, pos_of_surviving_chars); // new_string 생성
        printf("%s", new_str);

        return 0;
}
```

- 실행 결과



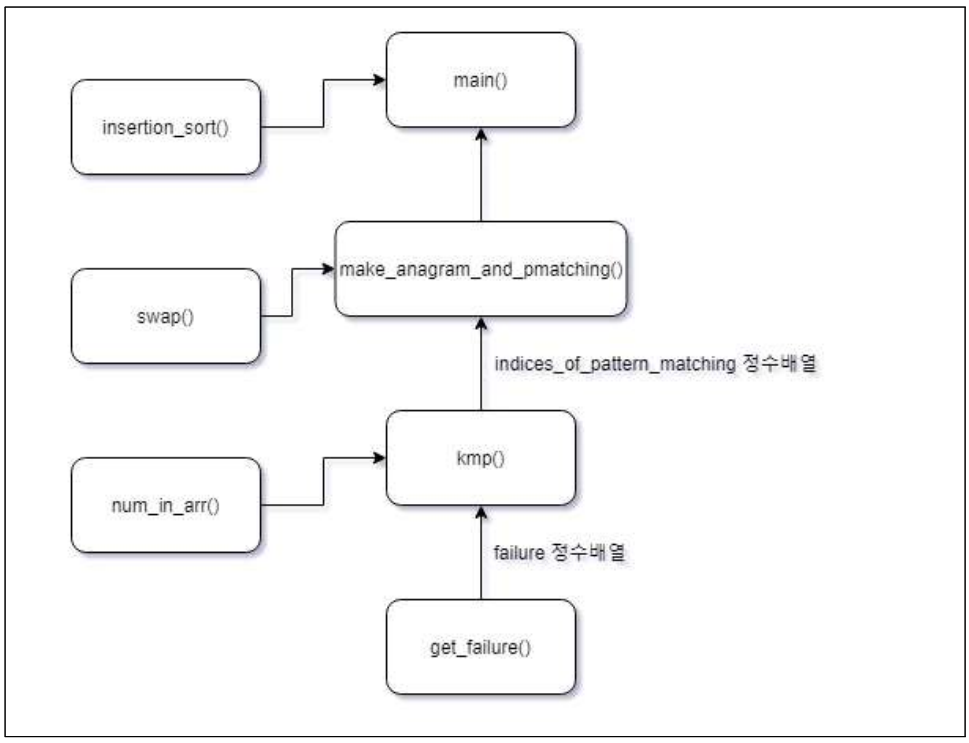






3번

- 함수 구성 다이어그램



- **함수 설명**

① void swap()

문자열 string의 i번째 char와 j번째 char를 바꾸는 함수이다.

input : int i, int j, char * str

output : 없음

② void insertion_sort()

pmatch를 통해 얻은 결과의 배열을 정렬하기 위한 함수이다. 삽입 정렬로 구현하였다.

input : int arr[], int size

output : 없음

③ int num_in_arr()

세 번째 인자로 입력되는 int num이 해당 배열에 원소로 들어 있는지를 검사하는 함수이다.

input : int * arr, int size, int num

output : num이 해당 배열에 들어 있다면 1, 들어 있지 않다면 0

④ void get_failure()

pattern 문자열을 순회하면서, failure 정수 배열을 얻어내는 함수이다. 문제2번을 해결하는데 사용되었던 함수와 동일하다.

input : int * failure, char * pattern

output : 없음

⑤ void kmp()

get_failure()에서 만든 failure 배열을 인자로 받아서, 이를 이용하여, indices_of_pattern_matching_size 숫자 배열을 생성하는 데에 목적이 있는 함수이다. 문제2번에서 사용되었던 kmp()와의 차이점은 3번에서는 kmp()를 '여러 차례' 실행하면서, 그 결과를 indices_of_pattern_matching 배열에 누적하기 때문에, ③함수를 통해 해당 결과물이 이미 배열에 기록되어 있는지를 검사하는 조건문이 추가되었다.

input : int * failure, int * indices_of_pattern_matching,
int * indices_of_pattern_matching_size, char * str, char * pattern

return : 없음

⑥ void make_anagram_and_pmatching();

백트래킹 기법으로 주어진 문자열에 대한 모든 anagram들을 생성하고, 하나의

anagram이 생성될 때마다, 해당 anagram으로 패턴 매치를 진행한다. 그리고 그 결과물을 indices_of_pattern_matching 배열에 기록하게 된다.
input : int totalN, int N, char * pattern, char * str, int * failure, int * indices_of_pattern_matching, int * indices_of_pattern_matching_size
output : 없음

* 백트래킹 기법
해를 얻을 때까지 모든 가능성을 시도하는 기법이다. 모든 가능성은 하나의 트리처럼 구성할 수 있으며, 가지 중에 해결책이 있다. 트리를 검사하기 위해 깊이 우선 탐색을 사용한다. 탐색 중에 오답을 만나면 이전 분기점으로 돌아간다. 시도해보지 않은 다른 해결 방법이 있으면 시도한다. 해결 방법이 더 없으면 더 이전의 분기점으로 돌아간다. 모든 트리의 노드를 검사해도 답을 못 찾을 경우, 이 문제의 해결책은 없는 것이다.
백트래킹은 보통 재귀 함수로 구현된다. 재귀로 파생된 해결 방법은 하나 이상의 변수가 필요한데, 이것은 현재 시점에서 적용할 수 있는 변수값들을 알고 있다. 백트래킹은 깊이 우선 탐색과 대략 같으나 기억 공간은 덜 차지한다. 현재의 상태를 보관하고 바꾸는 동안만 차지한다.
탐색 속도를 높이기 위해, 재귀 호출을 하기 전에 시도할 값을 정하고 조건(전진 탐색의 경우)을 벗어난 값을 지우는 알고리즘을 적용할 수 있다. 아니면 그 값을 제외한 다른 값들을 탐색할 수도 있다.

- **코드**

```
#include <stdio.h >
#include <stdlib.h >
#include <string.h >
#define max_string_size 30
#define max_pattern_size 10

// 필요한 작은 단위의 함수들 구현
void swap(int i, int j, char * str);
void insertion_sort(int arr[], int size);
int num_in_arr(int * arr, int size, int num);
// 메인 알고리즘
void get_failure(int * failure, char * pattern);
void kmp(int * failure, int * indices_of_pattern_matching, int * indices_of_pattern_matching_size, char * str, char * pattern);
void make_anagram_and_pmatching(int totalN, int N, char * pattern, char * str, int * failure,
    int * indices_of_pattern_matching, int * indices_of_pattern_matching_size);
```

```
void swap(int i, int j, char * str) {
    int temp;
    if (i == j) {
        return;
    }
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
    return;
}

void add_a_char_to_a_string(char * str, char c) {
    char * p = str;
    while (*p !='₩0') {// 문자열 끝 탐색
        p++;
    }
    *p = c;
    *(p +1) ='₩0';
}

void get_failure(int * failure, char * pattern) {
    /* 입력받은 pattern에 대한 failure 배열을 만드는 함수 */
    int i, n = strlen(pattern);
    failure[0] =-1;
    for (int j =1; j < n; j ++) {
        i = failure[j -1];
        while ((pattern[j] != pattern[i +1]) && (i >=0))
            i = failure[i];
        if (pattern[j] == pattern[i +1])
            failure[j] = i +1;
        else  failure[j] =-1;
    }
    for (int i =0; i < n; i ++) {  // 사용의 편의를 위하여 원소를 모두 1씩 증
가시킴
        failure[i] +=1;
    }
}

int num_in_arr(int * arr, int size, int num) {
    int check =1;
    for (int i =0; i <size; i ++) {
        if (arr[i] == num) {
            return check;
        }
    }
    return 0;
}
```

```
void insertion_sort(int arr[], int size) {
    /* indices_of_pattern_matching 배열의 원소들을 정렬하기 위한 함수 */
    int i, j, key;
    for (i =1; i <size; i ++) {
        key = arr[i];
        // 현재 정렬된 배열은 i-1까지이고, i-1번째부터 역순으로 조사.
        for (j = i -1; j >=0 && arr[j] > key; j --) {
            arr[j +1] = arr[j];
        }
        arr[j +1] = key;
    }
}

void kmp(int * failure, int * indices_of_pattern_matching, int *
indices_of_pattern_matching_size, char * str, char * pattern) {
    /* indices_of_pattern_matching 배열을 만드는 것이 목적인 함수이다.
    string에서 pattern에 매칭이되는 시작 index들을 원소로 저장한다.
    ex) 입력 : bbbbabbbc / bbb 에 대해서 {0, 1, 5} 라는 배열을 만듦.*/
    int len_of_string, len_of_pattern;
    int begin, matched;
    int check =1;
    len_of_string = strlen(str);
    len_of_pattern = strlen(pattern);
    begin =0;
    matched =0;
    while (begin <= len_of_string - len_of_pattern) {
        if (matched < len_of_pattern && str[begin + matched] ==
pattern[matched]) {
            matched +=1;
            // begin 인덱스가 이미 indices_of_pattern_matching 배열의 원소
로 들어있지는 않은지 점검
            if (matched == len_of_pattern
& & ! n u m _ i n _ a r r ( i n d i c e s _ o f _ p a t t e r n _ m a t c h i n g ,
*indices_of_pattern_matching_size, begin)) {

indices_of_pattern_matching[(*indices_of_pattern_matching_size)++] =begin;
            }
        }
        else {
            if (matched ==0) {
                begin +=1;
            }
            else {
                begin += matched - failure[matched -1];
                matched = failure[matched -1];
            }
        }
    }
}
```

```
void make_anagram_and_pmatching(int totalN, int N, char * pattern, char * str, int * failure,
    int * indices_of_pattern_matching, int * indices_of_pattern_matching_size) {
    /* 입력된 pattern의 anagram 하나가 생성될 때마다, 이 anagram pattern으로 pmatch를 진행하는 함수이다.
    pmatch의 결과는 indices_of_pattern_matching 정수배열에 누적되면서 기록된다.*/
    int i;
    if (N ==1) {
        for (i =0; i < totalN; i ++) {
            //add_a_char_to_a_string(anagrams[count], str[i]);
            get_failure(failure, pattern);
            kmp(failure, indices_of_pattern_matching, indices_of_pattern_matching_size, str, pattern);
        }
        return;
    }
    for (i =0; i < N; i ++) {
        /* 백트래킹 기법 */
        // swap을 하고,
        swap(i, N -1, pattern);
        // 재귀로 들어가며
        make_anagram_and_pmatching(totalN, N -1, pattern, str, failure,
                                   indices_of_pattern_matching, indices_of_pattern_matching_size);
        // 재귀가 끝난 뒤에, 재귀로 들어가기 전 상태로 pattern을 복구.
        swap(i, N -1, pattern);
    }
    return;
}

int main() {
    char str[max_string_size];
    char pattern[max_pattern_size];
    int failure[max_pattern_size];
    int indices_of_pattern_matching[max_string_size];
    int indices_of_pattern_matching_size =0;
    int len_of_str, len_of_pat; // 입력 받은 pattern의 길이를 저장할 변수
    printf("문자열을 입력해 주세요. : ");
    if (scanf("%s", str) ==-1) {
        return 0;
    }
    printf("패턴을 입력해 주세요. anagrams까지 포함하여 해당 패턴에 매칭되는 위치를 알려줍니다. : ");
    if (scanf("%s", pattern) ==-1) {
        return 0;
    }
```

```
    str[max_string_size -1] =0;
    pattern[max_pattern_size -1] =0; // str과 pattern이 널문자로 끝나지 않을 수도 있다는 경고 없애기

    len_of_str = strlen(str);
    len_of_pat = strlen(pattern);
    if (len_of_str >=30 || len_of_str <1 ||
        len_of_pat >=10 || len_of_pat <1
        || len_of_str < len_of_pat) { // 예외 처리
        printf("잘못된 입력입니다.");
        exit(1);
    }
    make_anagram_and_pmatching(len_of_pat, len_of_pat, pattern, str, failure,
        indices_of_pattern_matching, &indices_of_pattern_matching_size);
    // indices_of_pattern_matching 배열의 원소들을 정렬한 뒤 출력
    insertion_sort(indices_of_pattern_matching, indices_of_pattern_matching_size); // 삽입 정렬
    printf("[ ");
    for (int i =0; i < indices_of_pattern_matching_size; i ++) {
        printf("%d ", indices_of_pattern_matching[i]);
    }
    printf("]");
    return 0;
}
```

- 실행 결과