# 자료구조(Data Structure)

# Programming Assignment 2

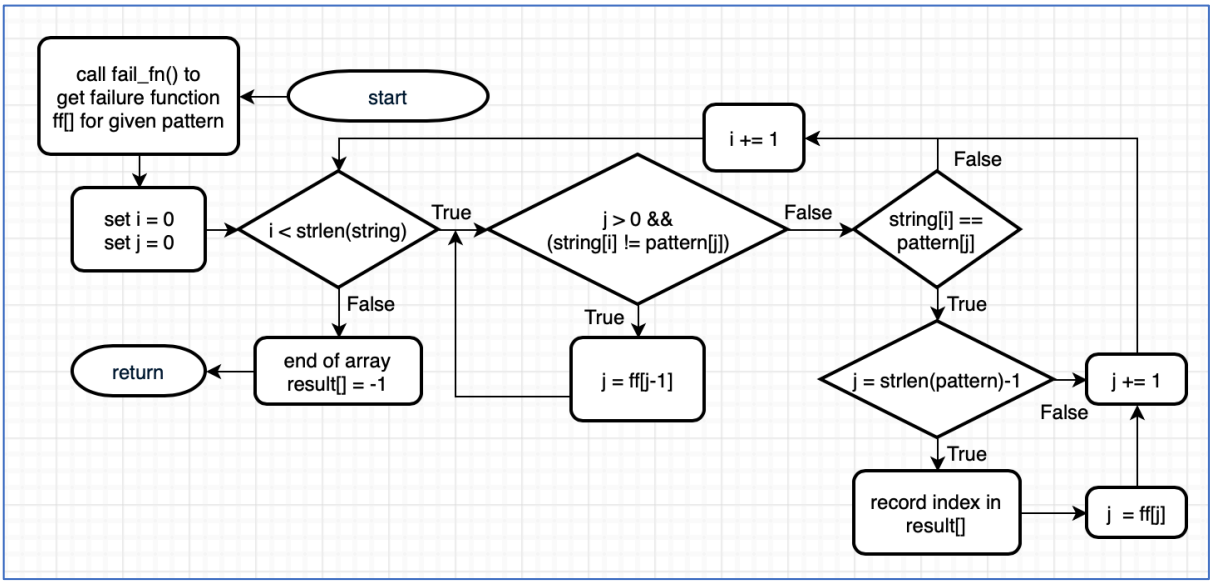
20161603
신민준

# 목차

# 1. 문제1

문제 1은 Knuth–Morris–Pratt 알고리즘을 사용해 주어진 string 속에서 모든 일치하는 pattern을 찾아 pattern들이 시작하는 index들을 구하는 문제입니다. 전체적인 코드 아이디어는 교재에서 제공하는 것과 비슷하나, 교재에서 출력을 하는 부분을 제거하고, index를 배열에 저장하는 코드를 넣었습니다. pmatch_all 함수가 끝난 후 해당 배열을 리턴해 메인 함수에서 리턴받은 배열을 출력하도록 했습니다.

## 1.1 프로그램 구조




**Flowchart 1**

## 1.2 세부 설명

교재에서 보인 것과는 다르게, 저는 failure function을 만들 때, ff[] 배열의 원소가 검색할 패턴의 인덱스를 정확히 반영하도록 구현했습니다. 이렇게 구현하는 것이 off-by-one 에러들을 구현하는 과정에 있어서 최소한으로 줄일 수 있을것이라 생각합니다.

구현한 failure function은 다음과 같습니다.

```
/**
 * Failure function for Knuth-Morris-Pratt algorithm.
 *    failure function is also created using KMP algorithm.
 *    failure function's element value is the exact index to look for next.
 * @param  pattern Pattern string to find failure function for
 * @return         Failure function for given pattern
 */
int* fail_fn(char* pattern) {
   int plen = (int)strlen(pattern);
   int* ff = (int*)malloc(sizeof(int)*plen);
   int j = 0;

   ff[0] = 0;
   for(int i=1 ; i<plen ; i++) { // i: index for pattern 1
      while(j > 0 && pattern[i] != pattern[j])
         // j: index for pattern 2, which is used for comparing
         j = ff[j-1];
```

```
      if(pattern[i] == pattern[j])
         ff[i] = ++j;
      else
         ff[i] = 0;
   }
   return ff; // this pointer needs to be free'd after
}
```

**Code 1**

이 함수 fail_fn()은 Failure function을 보이는 배열 ff를 리턴합니다. 함수 fail_fn()은 pmatch_all() 함수의 초기화 과정에서 호출되어 Failure function을 int형 포인터 변수 ff에 저장시킵니다.

```
   int* ff = fail_fn(pattern);
   int* result = (int*)malloc(sizeof(int)*(slen+1));
```

**Code 2**

또한, 교재와는 다른 조건인 "모든 인덱스 출력"을 만족시키기 위해 result라는 포인터 변수를 만들고, 메모리를 할당했습니다. 이 때, 최대로 가능한 결과의 개수는 입력받은 스트링의 길이와 같기 때문에, 스트링 길이+1 만큼의 메모리를 할당합니다.

```
for(int i=0 ; i<(int)strlen(string) ; i++) { // i: index for string
      while(j>0 && pattern[j] != string[i]) // j: index for pattern
         // change pattern's index if characters did not match
         j = ff[j-1];
      if(pattern[j] == string[i]) {
         if(j == plen-1) {
            // reached end of pattern -> found pattern
            result[r_cnt++] = i+1-plen; // record index in result
            j = ff[j]; // go for another match
         } else {
            j++;
         }
      }
   }
   result[r_cnt] = -1; // end of array is signified by -1
```

**Code 3**

이후, KMP 알고리즘을 따라가며 패턴을 찾고, 패턴을 찾은 경우 result[] 배열에 해당 패턴의 인덱스를 저장하고 다시 패턴을 찾습니다. 이후, result[] 배열의 마지막을 -1 값으로 정해, 배열이 끝났음을 나타냅니다. 이후, main 함수로 result 배열을 리턴하고, main 함수에서 해당 결과를 출력합니다. 이 프로그램의 총 시간 복잡도는 string 길이가 n, pattern의 길이가 m일 때, $O(n+m)$입니다.

```
result = pmatch_all(str, pat);

   while(result[i] != -1) // printing results
      printf("%d\n", result[i++]);
```

**Code 4**

### 1.3 참고 사항

- Failure function인 fail_fn()을 구현할 때, 만약 일반적인 nested loop을 사용해 구현했다면, 패턴의 길이를 m이라 가정할 때, $O(m^3)$의 시간 복잡도를 가지게 됩니다. 하지만 이 또한 [Code 1]에서와 같이 KMP 알고리즘을 활용해 구현했기 때문에, fail_fn() 함수의 시간 복잡도는 $O(m)$입니다.
- [Code 2]에서 result 배열의 할당을 할 때, worst-case scenario를 가정해 입력받은 스트링의 길이만큼을 먼저 할당해주었기에, 메모리의 낭비가 생깁니다. 이를 고치려면, result 배열에 결과값이 하나씩 추가될 때 마다, realloc 함수를 사용해 배열의 길이를 조금씩 늘이는 방향으로 구현하면 될 것입니다.

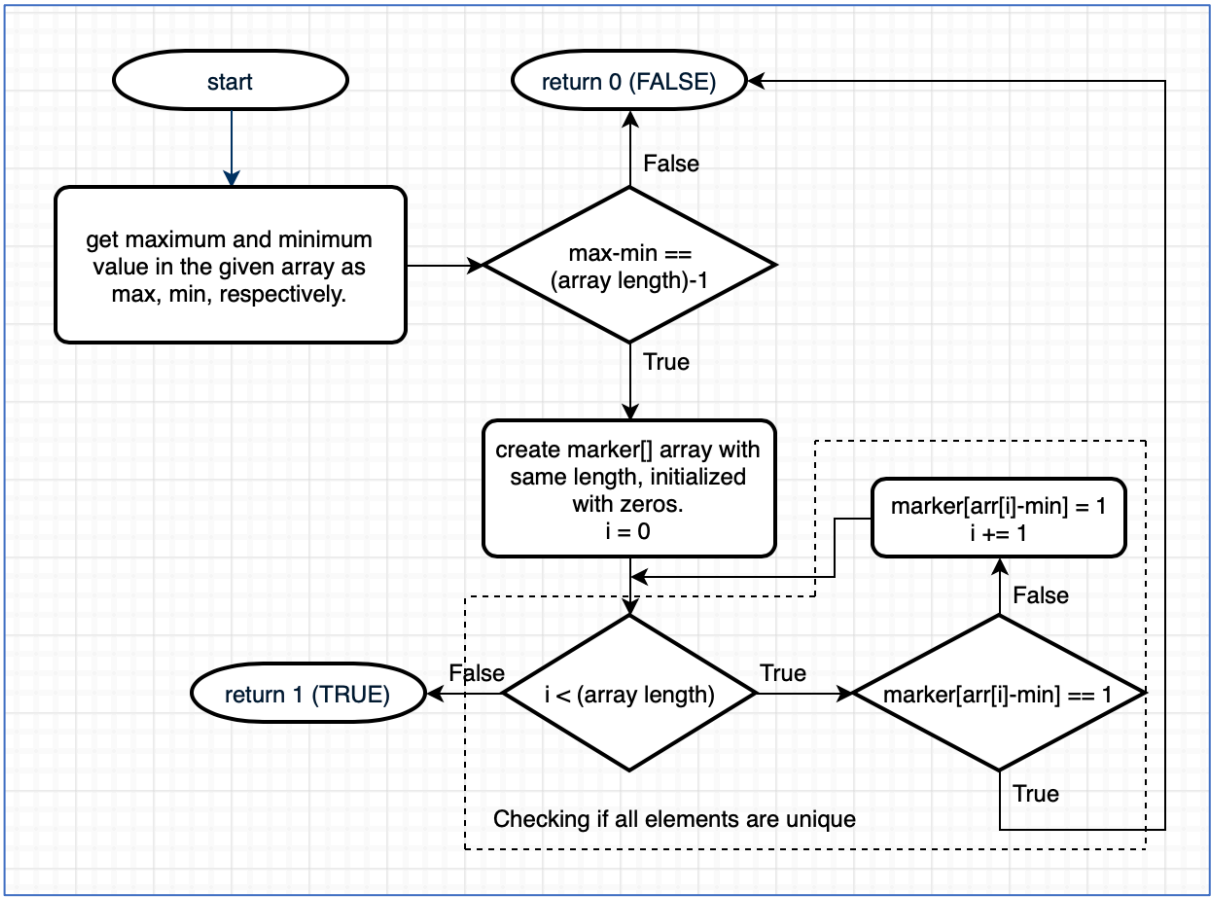
## 2. 문제2

문제 2는 parameter로 받은 array가 연속되어있는 숫자로 이루어져 있는지 확인하는 문제입니다. 1 이상의 정수를 원소로 하는 배열은 다음과 같은 조건을 만족한다면 모든 원소가 연속적입니다.

- 배열의 최댓값과 최솟값을 더한 값이 (배열의 길이 - 1)와 같음
- 배열의 각 원소가 모두 unique함(중복되는 값이 없음)

따라서, 이 문제는 입력받은 배열이 위 두 조건을 만족하는지 확인하는 프로그램을 만들면 해결됩니다.

### 2.1 프로그램 구조




**Flowchart 2**

### 2.2 세부 설명

먼저, 배열 안의 원소 중 최댓값과 최솟값을 찾아야 합니다. 이는 다음과 같이 구현했습니다.

```
#define INT_MAX 2147483637
#define INT_MIN -2147483648
...(중략)
int max=INT_MIN, min=INT_MAX;
...(중략)
// gets maximum & minimum numbers from the array
   for(int i=0 ; i<len ; i++) {
      if(arr[i] > max)
         max=arr[i];
```

```
    if(arr[i] < min)
      min=arr[i];
  }
```

**Code 5**

그 다음, 구한 min값과 max값의 관계로부터 max-min+1이 배열의 길이와 같은지 확인합니다. 만일 같지 않다면, false를 나타내는 0을 리턴합니다.

max-min+1이 배열의 길이와 같다면, 각 원소들의 값이 서로 중복되지 않는지 확인해야 합니다. 이는 다음과 같이 구현했습니다.

```
if(max-min == len-1) { // condition 1: max-min == array length-1
   marker = calloc(len, sizeof(int)); // allocates space initialized with 0
   // condition 2: check if all elements are unique
   for(int i=0 ; i<len ; i++) {
      if(marker[arr[i]-min] == 1) {
         // duplicate elements found
         free(marker);
         return 0;
      }
      marker[arr[i]-min] = 1;
   }
   free(marker);
   return 1;
}
return 0;
```

**Code 6**

Code 6에서 볼 수 있듯이, 0으로 초기화된 배열 marker를 만들고, arr 배열의 각 원소들에 대해 marker[arr[i]-min] 의 값을 확인합니다. 만약 확인한 값이 0이 아닌 1이라면, 중복된 원소가 존재한다는 의미이므로, FALSE값인 0을 리턴하고, 모든 원소에 대해 확인한 값이 0이라면, 각 원소들의 값이 unique하다는 의미이기에 두번째 조건도 만족시키므로 TRUE를 나타내는 값 1을 리턴합니다. 이 프로그램의 loop는 최대 주어진 배열의 길이만큼 반복되기에, 시간복잡도는 배열의 길이를 n으로 가정했을 때, $O(n)$입니다.

### 2.3 참고 사항

- 이 문제를 풀기 위해 사용한 알고리즘은 주어진 배열의 원소가 음수일 경우 정상적으로 작동하지 않습니다. 다행히도 주어진 조건이 1에서 100까지의 값이었기 때문에 이같은 알고리즘을 사용할 수 있었습니다.

## 3. 문제 코드

### 3.1 문제1 코드

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/**
 * Failure function for Knuth-Morris-Pratt algorithm.
 *    failure function is also created using KMP algorithm.
 *    failure function's element value is the exact index to look for next.
 * @param  pattern Pattern string to find failure function for
 * @return         Failure function for given pattern
 */
int* fail_fn(char* pattern) {
   int plen = (int)strlen(pattern);
   int* ff = (int*)malloc(sizeof(int)*plen);
   int j = 0;

   ff[0] = 0;
   for(int i=1 ; i<plen ; i++) { // i: index for pattern 1
      while(j > 0 && pattern[i] != pattern[j])
         // j: index for pattern 2, which is used for comparing
         j = ff[j-1];
      if(pattern[i] == pattern[j])
         ff[i] = ++j;
      else
         ff[i] = 0;
   }
   return ff; // this pointer needs to be free'd after
}
/**
 * Find all index of pattern in string using Knuth-Morris-Pratt algorithm
 * @param  string  Pointer of string to find pattern in
 * @param  pattern Pointer of pattern string to find
 * @return         Pointer to array holding all indexes of found patterns.
 *                 The end of array is noted with value -1
 */
int* pmatch_all(char* string, char* pattern) {
   int j=0, r_cnt=0;
   int plen = strlen(pattern);
   int slen = strlen(string);
   int* ff = fail_fn(pattern);
   int* result = (int*)malloc(sizeof(int)*(slen+1));

   for(int i=0 ; i<(int)strlen(string) ; i++) { // i: index for string
      while(j>0 && pattern[j] != string[i]) // j: index for pattern
         // change pattern's index if characters did not match
         j = ff[j-1];
      if(pattern[j] == string[i]) {
         if(j == plen-1) {
            // reached end of pattern -> found pattern
```

```
            result[r_cnt++] = i+1-plen; // record index in result
            j = ff[j]; // go for another match
         } else {
            j++;
         }
      }
   }
   result[r_cnt] = -1; // end of array is signified by -1

   free(ff); // freeing failure function, as it was malloc'ed in fail_fn()
   return result;
}
int main(int argc, const char* argv[]) {
   int *result;
   int i=0;
   char str[31];
   char pat[31];

   scanf(" %[^\n]s", str);
   scanf(" %[^\n]s", pat);

   result = pmatch_all(str, pat);

   while(result[i] != -1) // printing results
      printf("%d\n", result[i++]);

   free(result); // freeing result, as it was allocated in pmatch_all()
   return 0;
}
```

### 3.2 문제2 코드

```
#define INT_MAX 2147483637
#define INT_MIN -2147483648

#include <stdio.h>
#include <stdlib.h>
/**
 * Finds if array has consecutive natural numbers.
 * If an array is consecutive, it satisfies following two conditions.
 *    1. Array's maximum number-minimum number == array length-1
 *    2. Every element in the array is unique
 * @param  arr Pointer to array holding natural numbers.
 * @param  len Length of the array
 * @return     1 if given array is consecutive
 *            0 if given array is not consecutive
 */
int is_consecutive(int* arr, int len) {
   int max=INT_MIN, min=INT_MAX;
   int* marker;
```

```
   if(len <= 0)
      return 0;

   // gets maximum & minimum numbers from the array
   for(int i=0 ; i<len ; i++) {
      if(arr[i] > max)
         max=arr[i];
      if(arr[i] < min)
         min=arr[i];
   }

   if(max-min == len-1) { // condition 1: max-min == array length-1
      marker = calloc(len, sizeof(int)); // allocates space initialized with 0
      // condition 2: check if all elements are unique
      for(int i=0 ; i<len ; i++) {
         if(marker[arr[i]-min] == 1) {
            // duplicate elements found
            free(marker);
            return 0;
         }
         marker[arr[i]-min] = 1;
      }
      free(marker);
      return 1;
   }
   return 0;
}
int main(int argc, const char* argv[]) {
   int* array;
   int array_length;

   scanf(" %d", &array_length);

   // allocate space depending on the array_length given
   array = (int*)malloc(sizeof(int)*array_length);
   for(int i=0 ; i<array_length ; i++) {
      scanf(" %d", &array[i]);
   }

   // print 1 if consecutive, 0 otherwise
   printf("%d\n", is_consecutive(array, array_length));
   free(array);

   return 0;
}
```