자료구조(Data Structure)

Programming Assignment 1

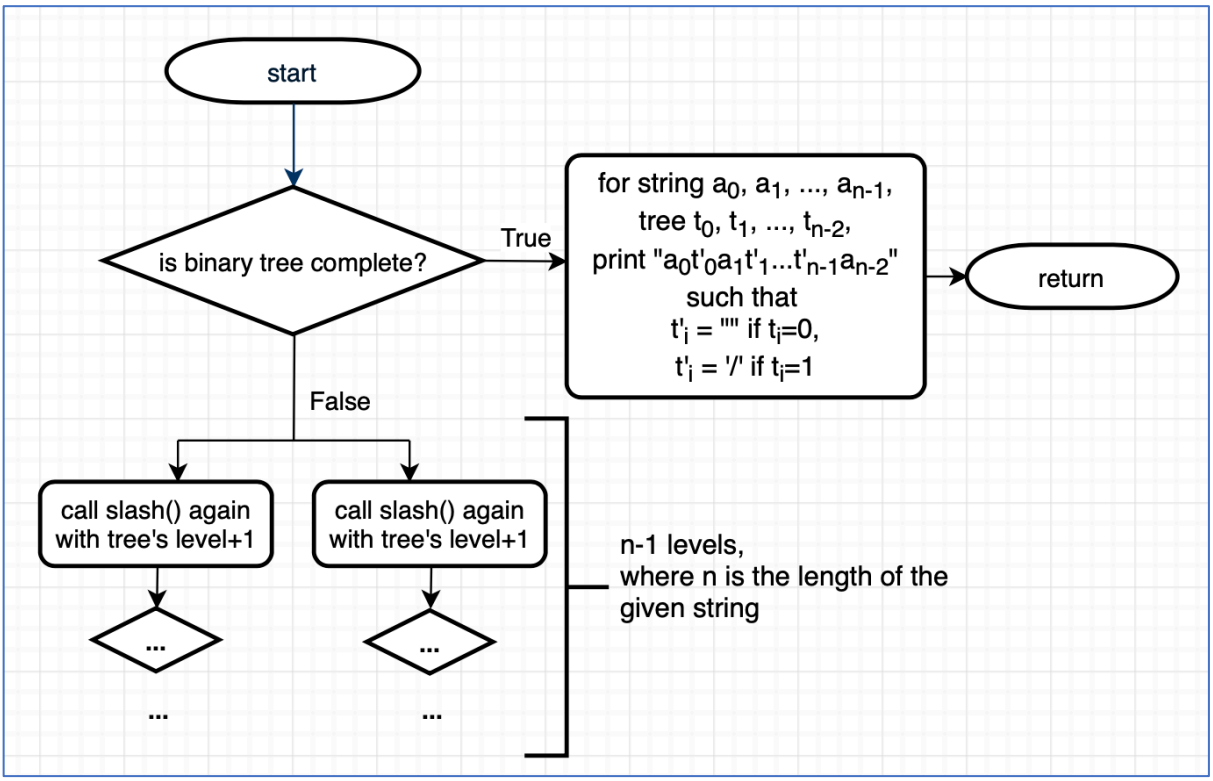
20161603
신민준

목차

1. 문제1

문제1은 입력받은 수열 사이에 '/' 문자를 넣거나 넣지 않음으로써 생기는 모든 문자열을 출력하는 문제입니다. 이를 해결하기 위해 만든 프로그램의 flowchart는 다음과 같습니다.

1.1 프로그램 구조



Flowchart 1

1.2 세부 설명

이 문제는 결론적으로, 빈 문자와 '/'로 만들 수 있는 모든 배열을 만들고, 입력 받은 수열 사이사이를 이 새로 만든 배열(앞으로 트리라고 명시)로 각각 채우는 것과 같습니다. 따라서, recursion을 사용해 다음과 같이 0과 1을 사용한 트리를 만들었습니다.

```
// appends 1 or 0 to binary_tree and calls on itself, with index increased.
binary_tree[tree_index] = 1;
slash(original_string, ostr_len, binary_tree, tree_index + 1);
binary_tree[tree_index] = 0;
slash(original_string, ostr_len, binary_tree, tree_index + 1);
```

Code 1

[Code 1] 에서 보이다시피, 현재 binary_tree의 현재 인덱스 위치에 1 또는 0을 넣고 인덱스를 1 증가시킨 후 다음 recursion을 call하는 방법으로 트리를 만들었습니다.

End condition은 트리의 길이가 입력 받은 수열의 길이보다 1 작을 때 입니다.

```
// end condition
if(ostr_len-1 == tree_index) {
```

```
    // prints original_string with translated binary_tree characters between.
    for(i=0 ; i<ostr_len ; i++) {
       printf("%c", original_string[i]);
       if(i < ostr_len-1 && binary_tree[i])
          printf("/");
    }
    printf("\n");
    return;
}
```

Code 2

입력 받은 수열의 길이보다 1 작은 길이에 도달하면 그 때 두 배열을 번갈아 가며 출력하고, 함수를 종료하도록 프로그래밍했습니다.
이 방법을 사용했을 시, 시간복잡도는 $O(n \cdot 2^n)$입니다.

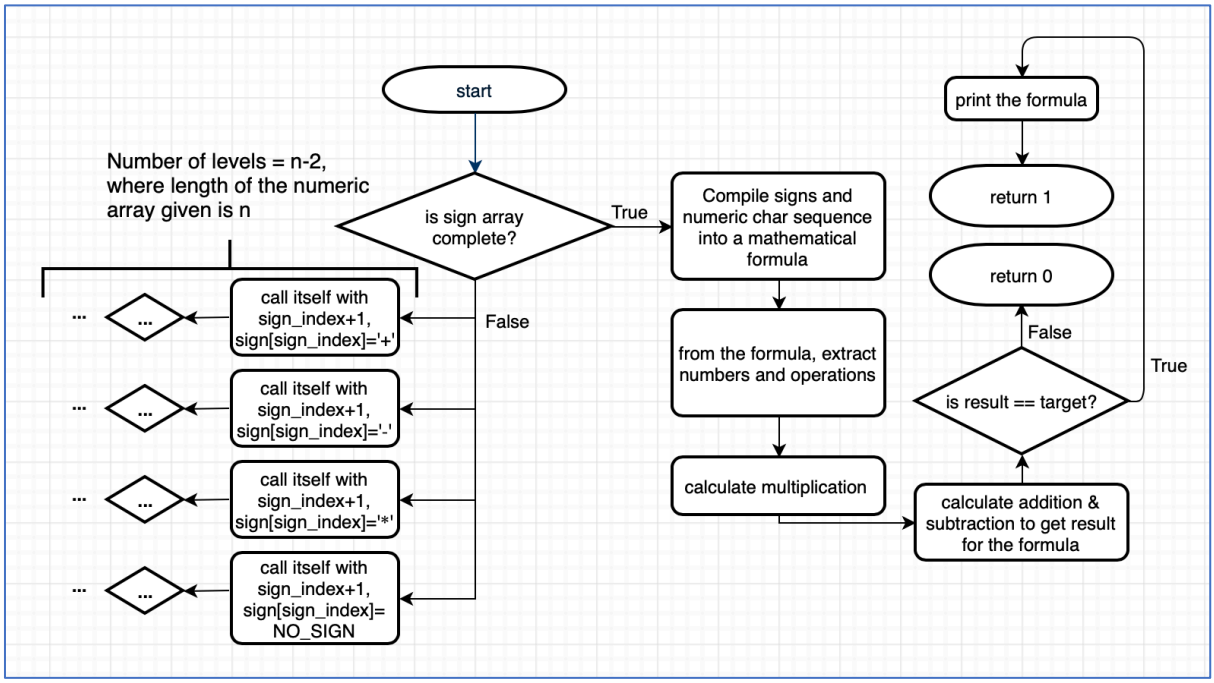
1.3 참고 사항

만약 트리를 0과 1이 아니라, 빈 문자와 '/'로 바로 만들었더라면, 따로 0과 1을 각 문자로 변환하는데 드는 시간을 단축시킬 수 있었을 것입니다. 또한, 지금 이 프로그램의 시간복잡도가 $O(n \cdot 2^n)$인데, 프린트 하는 부분을 recursion의 end condition에 두지 않고, 각 level을 내려갈 때 마다 해당하는 문자를 프린트했더라면 시간복잡도를 $O(2^n)$까지 줄일 수 있었을 것입니다.

2. 문제2

목표하는 수를 얻는 수학 식을 얻기 위해선, 가능한 모든 수학식을 따져보는 것이 가장 간단합니다. 이는 숫자의 배열 사이에 부호를 넣는 모든 가짓수를 따짐으로써 해결할 수 있습니다. 따라서 본질적으로 보았을 때, 문제2는 문제1과 동일한 방법으로 풀 수 있습니다. 문제1 프로그램과 이 프로그램을 비교했을 때에 바뀐 점은, 0과 1의 배열을 만드는 대신, 연산기호 '+', '-', '*', 그리고 빈 문자를 뜻하는 NO_SIGN 문자, 이 네 문자로 배열을 만들었다는 점입니다. 다음 Flowchart는 이를 보여주고 있습니다.

2.1 프로그램 구조



Flowchart 2

2.2 세부 설명

이 recursion은 하나의 level을 추가할 때 마다, 각 sign에 해당하는 문자를 sign 배열에 추가합니다.

```
/**
    * Call next recursive.
    * NOTE: LOGICAL OR operation is not working. Hence using addition.
    *       probably due to how compiler optimises it.
    */
   sign[sign_index] = '+';
   success += print_valid(seq, seq_len, target, sign, sign_index+1);

   sign[sign_index] = '-';
   success += print_valid(seq, seq_len, target, sign, sign_index+1);

   sign[sign_index] = '*';
   success += print_valid(seq, seq_len, target, sign, sign_index+1);
```

```
sign[sign_index] = NO_SIGN;
success += print_valid(seq, seq_len, target, sign, sign_index+1);

return success;
```

Code 3

이 방식으로 level을 올라가다 보면, 문제1에서처럼 +, -, *, NO_SIGN의 네 문자로 만들 수 있는 모든 배열을 만들 수 있습니다. 이 sign 배열이 완성되면, end condition을 만나게 됩니다. End condition은 (sign배열의 길이) == (입력 받은 숫자 배열의 길이 – 1) 입니다.
End condition을 만나게 되면, 먼저 수 배열 사이사이에 sign 배열을 집어넣어 수학적 식을 문자열로 만들게 됩니다.

```
// create arithmic expression by combining numeric sequence with signs.
for(int i=0 ; i<seq_len-1 ; i++) {
   // if sign is NO_SIGN, exclude that sign from appending to expression.
   if(sign[i] != NO_SIGN) expression[exp_idx++] = sign[i];
   expression[exp_idx++] = seq[i+1];
}
expression[exp_idx] = '\0';
```

Code 4

만약 집어넣어야 하는 부호가 NO_SIGN 부호라면, [Code 4]에서처럼 부호를 붙이는 것을 생략하고 다음 숫자를 식에 붙여넣습니다.
수학적 식이 모두 완성되어 expression 문자열에 저장된 후, expression 문자열에서 숫자와 부호를 따로 추출해내어 각각 numbers, ops 배열에 저장합니다. 이 과정은 [Code 5]에서 볼 수 있습니다.

```
// extract integers and operations from the expression
// this is done by extracting integer, and since extract_int removes
// read numeric characters, the character at index 0 must be the sign.
numbers[0] = extract_int(expression);
for(i=0 ; expression[0] != '\0' ; i++) {
   // ops stores mathematical operations
   ops[i] = expression[0];
   remove_strhead(expression, 1);
   // numbers stores numbers inside the expression
   numbers[i+1] = extract_int(expression);
}
```

Code 5

[Code 5]에서 사용된 remove_strhead() 함수는 지정된 수 만큼의 element들을 string의 첫번째 문자부터 삭제하는 함수이고, extract_int()는 특정 문자열에서 숫자를 추출하고, 해당하는 부분을 문자열에서 지우는 함수입니다. 이 두 함수는 다음과 같습니다.

```
/**
 * Removes certain number of characters from string head.
 * @param  str   String to remove chars from.
 * @param  count Number of chars to remove.
 * @return       Length of remaining characters.
 */
int remove_strhead(char str[], int count) {
   int idx, new_idx = 0;
   int str_len = strlen(str);
   for(idx=count ; idx < str_len ; idx++) {
      str[new_idx++] = str[idx];
   }
   str[new_idx] = '\0';
   return new_idx;
}
```

Code 6

```
/**
 * Extract number from character array. Reads numbers until it meets its first
 * non-numeric character. Removed read number from character array after.
 * @param  expression Character array that needs to be parsed.
 * @return            Parsed number from array.
 */
int extract_int(char expression[]) {
   int extractedInt = 0;
   int index;
   // breaks loop when expression[index] is non-numeric character.
   for(index=0 ; '0' <= expression[index] && expression[index] <= '9' ; index++) {
      extractedInt = extractedInt*10 + expression[index]-'0';
   }
   remove_strhead(expression, index);

   return extractedInt;
}
```

Code 7

이 과정을 모두 거친 후, 프로그램에는 NO_SIGN 기호가 전부 처리된 이후의 sign 배열과 numbers 배열을 가지고 있습니다.

이제 남은 연산은 곱셈, 덧셈, 그리고 뺄셈인데, 여기서 곱셈은 나머지 두 연산보다 우선순위가 높으므로 먼저 계산되어야 합니다. 이를 위해 sign 배열에서 곱셈 기호를 찾고, 해당 기호에 해당하는 두 숫자를 계산해 바로 numbers 배열에 넣고, 계산이 끝난 곱셈 기호와 숫자는 배열에서 지우는 방식을 택했습니다.

이후, 뺄셈은 빼는 수를 음수로 바꿔 계산한다면 결국 덧셈과 똑같이 계산할 수 있기에, 두 연산을 함께 수행해 수학 식의 최종 결과값을 얻었습니다.

```
// calculating multiplication first, since * is preceded before + or -
for(i=0 ; i<ops_len ; i++) {
   if(ops[i] == '*') { //calculate only if the operation is *
      // save the result in one array and overwrites the other by
      // shifting all elements after it.
      numbers[i+1] *= numbers[i];
      for(int idx=i ; idx<numbers_len-1 ; idx++)
         numbers[idx] = numbers[idx+1];
      numbers_len--; // since two numbers were calculated to make one.
      // same for the operations, since a operator was used
      for(int idx=i ; idx<ops_len-1 ; idx++)
         ops[idx] = ops[idx+1];
      ops_len--;
      i--; // to compensate for reduced overall array length
   }
}

// since remaining operations are either + or -, it'll be easier to
// first change numbers to negative counterparts if its matching operation
// is -, and then add up all numbers.
int result = numbers[0];
for(i=0 ; i<ops_len ; i++) {
   // if matching operation for the number is -, set number to negative
   // and then add up.
   result += (ops[i]=='+') ? numbers[i+1] : -numbers[i+1];
}
```

Code 8

연산 결과값인 result가 target과 같다면, 해당 식을 출력한 후, 1을 리턴하도록 했고, 만약 같지 않다면, 0을 리턴하도록 했습니다.

```
// if this is the correct formula for the target, print the formula.
if(result == target) {
   // excluding such cases that has '03' for its numbers.
   for(int i=1 ; i<(int)strlen(exp_bak) ; i++) {
      if(exp_bak[i-1]=='0' && exp_bak[i] <= '9' && '0' <= exp_bak[i])
         return 0;
   }

   // printing the formula
   printf("%c", seq[0]);
   for(int i=0 ; i<seq_len-1 ; i++) {
      if(sign[i] != NO_SIGN) // excluding sign if it is NO_SIGN
         printf("%c", sign[i]);
      printf("%c", seq[i+1]);
   }
   printf("\n");
   return 1;
```

```
    } else {
      return 0;
    }
```

Code 9

이 프로그램의 시간 복잡도는 $O(n^2 \cdot 4^n)$입니다.

2.3 참고 사항

- [Code 3]에서 보인 것처럼 리턴값을 다루는 이유는 LOGICAL OR문이 의도한 대로 작동하지 않기 때문입니다. || 기호를 사용하면, 양 쪽의 명령 중 처음 실행한 명령이 True 값을 가질 때, 다른 명령은 아예 실행도 되지 않아 정상적인 결과를 기대하기 어려웠습니다. 따라서 True를 뜻하는 리턴값과 False를 뜻하는 리턴값을 더해서 최종 리턴값을 구했습니다. 만일 input의 크기가 무한히 커진다면, 이 방법은 오버플로우를 일으킵니다.
- [Code 7]의 extract_int 함수의 일부 기능은 이미 string.h 헤더파일에 atoi 함수로 구현되어있음을 뒤늦게 알았습니다. 만일 다음에 유사한 기능이 필요하게 된다면, 해당 함수를 사용해 시간을 절약할 수 있을 것입니다.

3. 문제 코드

3.1 문제1 코드

```c
#include <stdio.h>
#include <string.h>
/**
 * Prints slashed text recursively
 * @param original_string Character array from the input
 * @param ostr_len       Length of original_string
 * @param binary_tree     Binary tree of 0 and 1 in the making.
 *                       Needs to be empty on initial call.
 * @param tree_index      Current index of the tree, where it's empty.
 */
void slash(char original_string[], int ostr_len, int binary_tree[], int tree_index) {
   int i;
   // end condition
   if(ostr_len-1 == tree_index) {
      // prints original_string with translated binary_tree characters between.
      for(i=0 ; i<ostr_len ; i++) {
         printf("%c", original_string[i]);
         if(i < ostr_len-1 && binary_tree[i])
            printf("/");
      }
      printf("\n");
      return;
   }

   // appends 1 or 0 to binary_tree and calls on itself, with index increased.
   binary_tree[tree_index] = 1;
   slash(original_string, ostr_len, binary_tree, tree_index + 1);
   binary_tree[tree_index] = 0;
   slash(original_string, ostr_len, binary_tree, tree_index + 1);
}
int main(int argc, const char* argv[]) {
   char o_string[11];
   int temp_tree[9];
   scanf("%[^\n]10s", o_string);
   slash(o_string, strlen(o_string), temp_tree, 0);
   return 0;
}
```

3.2 문제2 코드

```c
#define NO_SIGN '#'

#include <stdio.h>
#include <string.h>

/**
```

```
 * Removes certain number of characters from string head.
 * @param  str   String to remove chars from.
 * @param  count Number of chars to remove.
 * @return       Length of remaining characters.
 */
int remove_strhead(char str[], int count) {
   int idx, new_idx = 0;
   int str_len = strlen(str);
   for(idx=count ; idx < str_len ; idx++) {
      str[new_idx++] = str[idx];
   }
   str[new_idx] = '\0';
   return new_idx;
}
/**
 * Extract number from character array. Reads numbers until it meets its first
 * non-numeric character. Removed read number from character array after.
 * @param  expression Character array that needs to be parsed.
 * @return            Parsed number from array.
 */
int extract_int(char expression[]) {
   int extractedInt = 0;
   int index;
   // breaks loop when expression[index] is non-numeric character.
   for(index=0 ; '0' <= expression[index] && expression[index] <= '9' ; index++) {
      extractedInt = extractedInt*10 + expression[index]-'0';
   }
   remove_strhead(expression, index);

   return extractedInt;
}
/**
 * Recursive function for creating and checking mathematical formula.
 * @param  seq       Sequence of numeric characters from input.
 * @param  seq_len    Length of seq
 * @param  target     Target value that we strive to achieve.
 * @param  sign       Array of characters, consists only of '+','-','*',NO_SIGN
 * @param  sign_index Current index of sign array.
 * @return           0 if all recursions failed to find matching formula for
 *                  target.
 *                  Any positive integers if otherwise.
 */
int print_valid(char seq[], int seq_len, int target, char sign[], int sign_index) {
   int success = 0;

   // end condition for recursion
   if(seq_len-1 == sign_index) {
      int numbers[10], numbers_bak[10];
      int numbers_len, ops_len, olen_bak;
      char ops[10], ops_bak[10];
      char expression[20];
```

```
    char exp_bak[20];
    expression[0] = seq[0];
    int exp_idx = 1;

    // create arithmic expression by combining numeric sequence with signs.
    for(int i=0 ; i<seq_len-1 ; i++) {
       // if sign is NO_SIGN, exclude that sign from appending to expression.
       if(sign[i] != NO_SIGN) expression[exp_idx++] = sign[i];
       expression[exp_idx++] = seq[i+1];
    }
    expression[exp_idx] = '\0';

    // exp_bak - backup of expression for future printing
    strcpy(exp_bak, expression);

    int i;
    // extract integers and operations from the expression
    // this is done by extracting integer, and since extract_int removes
    // read numeric characters, the character at index 0 must be the sign.
    numbers[0] = extract_int(expression);
    for(i=0 ; expression[0] != '\0' ; i++) {
       // ops stores mathematical operations
       ops[i] = expression[0];
       remove_strhead(expression, 1);
       // numbers stores numbers inside the expression
       numbers[i+1] = extract_int(expression);
    }

    numbers_len = i+1;
    ops_len = i;
    // backup ops and numbers, for future printing
    for(int i=0 ; i<numbers_len ; i++) {
       numbers_bak[i] = numbers[i];
       ops_bak[i] = ops[i];
    }
    olen_bak = ops_len;

    // calculating multiplication first, since * is preceded before + or -
    for(i=0 ; i<ops_len ; i++) {
       if(ops[i] == '*') { //calculate only if the operation is *
          // save the result in one array and overwrites the other by
          // shifting all elements after it.
          numbers[i+1] *= numbers[i];
          for(int idx=i ; idx<numbers_len-1 ; idx++)
             numbers[idx] = numbers[idx+1];
          numbers_len--; // since two numbers were calculated to make one.
          // same for the operations, since a operator was used
          for(int idx=i ; idx<ops_len-1 ; idx++)
             ops[idx] = ops[idx+1];
          ops_len--;
          i--; // to compensate for reduced overall array length
```

```
        }
    }

    // since remaining operations are either + or -, it'll be easier to
    // first change numbers to negative counterparts if its matching operation
    // is -, and then add up all numbers.
    int result = numbers[0];
    for(i=0 ; i<ops_len ; i++) {
        // if matching operation for the number is -, set number to negative
        // and then add up.
        result += (ops[i]=='+') ? numbers[i+1] : -numbers[i+1];
    }

    // if this is the correct formula for the target, print the formula.
    if(result == target) {
        // excluding such cases that has '03' for its numbers.
        for(int i=1 ; i<(int)strlen(exp_bak) ; i++) {
            if(exp_bak[i-1]=='0' && exp_bak[i] <= '9' && '0' <= exp_bak[i])
                return 0;
        }

        // printing the formula
        printf("%c", seq[0]);
        for(int i=0 ; i<seq_len-1 ; i++) {
            if(sign[i] != NO_SIGN) // excluding sign if it is NO_SIGN
                printf("%c", sign[i]);
            printf("%c", seq[i+1]);
        }
        printf("\n");
        return 1;
    } else {
        return 0;
    }
}

/**
 * Call next recursive.
 * NOTE: LOGICAL OR operation is not working. Hence using addition.
 *       probably due to how compiler optimises it.
 */
sign[sign_index] = '+';
success += print_valid(seq, seq_len, target, sign, sign_index+1);

sign[sign_index] = '-';
success += print_valid(seq, seq_len, target, sign, sign_index+1);

sign[sign_index] = '*';
success += print_valid(seq, seq_len, target, sign, sign_index+1);

sign[sign_index] = NO_SIGN;
success += print_valid(seq, seq_len, target, sign, sign_index+1);
```

```
    return success;
}
int main(int argc, const char* argv[]) {
    char numbers_str[11], arithmic[9];
    int target;
    scanf("%10s", numbers_str);
    scanf("%d", &target);
    // if no valid formula for target is found, print "None"
    if(!print_valid(numbers_str, strlen(numbers_str), target, arithmic, 0)) {
        printf("None\n");
    }
    return 0;
}
```