令和７年度

千葉大学先進科学プログラム入学者選考課題

課題論述（数理情報学）

解答例

1

出題意図
データ圧縮を題材としてプログラムを読み取る力を見ている。また、効率の良いアルゴリズムへの理解も見ている。

解答
問1
(1)　もう1記号加えたら割合の和の2分の一を超えるところで分割する
(2)　②
(3)
ア　s
イ　i
ウ　i+1
エ　t
(4)
e: 0.25, 2, 00
a: 0.20, 2, 01
t: 0.18, 2, 10
h: 0.15, 3, 110
d: 0.12, 4, 1110
f: 0.10, 4, 1111

(5)　11 回

問2
(1)　(3, 3, c)
(2)　一致する文字列の先頭が参照部の先頭に最も近いものが選ばれる
(3)　0で埋めている
(4)　位置(ラベル)を-1、長さを0としている

問3
(1)　参照部の開始位置を変数 offset で管理し、配列 buffer の末尾の次が先頭につながるようにしている。配列の論理的な番号に offset の値を加えて配列のサイズで剰余を取った値を配列の実際の番号とする。

(2)

```
int findmatch(int p)
{
            int i;
            for(i=0;i<ENCSIZE;i++)
              if(buffer[(p+i+offset)%(REFSIZE+ENCSIZE)] !=
                     buffer[(REFSIZE+i+offset)%(REFSIZE+ENCSIZE)])
                     break;
            return(i);
}
```

2

（出題意図）
ガウスの消去法と呼ばれる連立1次方程式を数値的に解く解法プログラムに関する問題である。高校生であれば連立1次方程式を解くことは難しくなく、かつ多くの解法手段を学んでいる。この中で、解法手段を一般化し、プログラムとして表現する力を確認する。
本問ではプログラムを C 言語で与え、それを読み、理解できるかを問う。また、再帰表現、多重の for 文の理解力を問う。また、プログラムの前提を適切に把握し、例外処理を行う力を確認する。
(1)
問題:
12.00 4.00 -2.00 0.00
6.00 -2.00 -3.00 -4.00
-6.00 2.00 13.00 -16.00

12.00 4.00 -2.00 0.00
0.00 -4.00 -2.00 -4.00
0.00 4.00 12.00 -16.00

12.00 4.00 -2.00 0.00
0.00 -4.00 -2.00 -4.00
0.00 0.00 10.00 -20.00

12.00 4.00 -2.00 0.00
0.00 -4.00 -2.00 -4.00
0.00 0.00 10.00 -20.00

12.00 4.00 -2.00 0.00
0.00 -4.00 -2.00 -4.00
0.00 0.00 1.00 -2.00

12.00 4.00 0.00 -4.00
0.00 -4.00 0.00 -8.00
0.00 0.00 1.00 -2.00

12.00 4.00 0.00 -4.00
0.00 1.00 0.00 2.00
0.00 0.00 1.00 -2.00

12.00 0.00 0.00 -12.00
0.00 1.00 0.00 2.00
0.00 0.00 1.00 -2.00

1.00 0.00 0.00 -1.00
0.00 1.00 0.00 2.00
0.00 0.00 1.00 -2.00

1.00 0.00 0.00 -1.00
0.00 1.00 0.00 2.00
0.00 0.00 1.00 -2.00

解:
x1 = -1.00
x2 = 2.00
x3 = -2.00

(2) 解答例(赤字の部分が追加部分である。)

```
void back(double matrix[N][N+1], int step) {
    double factor;
    int i, j;
    void printmatrix();
    for (; step>=0; step--){
        factor = matrix[step][step];
        for (j = step; j < N + 1; j++) {
            matrix[step][j] /= factor;
        }
        printmatrix(matrix);
        for (i = 0; i < step; i++) {
            factor = matrix[i][step];
            for (j = step; j < N + 1; j++) {
                matrix[i][j] -= factor * matrix[step][j];
            }
        }
        printmatrix(matrix);
    }
}
```

(3)

forward 関数において matrix[step][step]=0 のとき、変数 factor が無限大となり解くことができない。matrix[step][step]が極めて小さな数字のときも同様である。

そこで、方程式の順番を入れ替えることで、この問題を解決する。ここでは、matrix[i][step]の絶対値が最大な方程式と順番を入れ替えることでこの問題を解決している。ただし i は step より大きい。

(4)

(3)の考察によれば、上記コメントアウト部において、matrix[i][step]の絶対値の最大値、つまり fabs(matrix[row][step])=0 となる場合は解が求まらない、つまり解なしとなる。

コードの修正例:

(36)行目と(37)行目の間に

```
if(fabs(matrix[row][step])==0){
    printf("解なし¥n");
    exit(0);
}
```

を追加する。

3

（出題意図）

リスト（文字列）検索の Knuth–Morris–Pratt （KMP）法を題材として、具体的な説明からアルゴリズムを考え、コードとして表現できる能力を問う。初出の用語に対しても説明と具体例を基に適切な理解を行い、抽象化してコードとして表現できる能力を確認する。

また、コードの計算量について、適切に考えることができることを問う。

（1） 特定のリストに対する比較回数の解析

19 回

sequence の各インデックスで不一致までの比較回数は、5, 1, 4, 1, 2, 1 となり、インデックス 6 から 5 回比較して pattern と一致し、また、そこで インデックスが sequence の長さになり、終了する。このため、比較回数は、5+1+4+1+2+1+5 = 19 となる。

（2） 一般的なリストに対する比較回数の最小・最大の解析

- 最小値

一致するものがなければ sequence のインデックス 0 から$n-1$までのn個のインデックスで、それぞれ 1 文字の比較がされ、次のインデックスに進むので is_equal が

n回 呼び出される。

- 最大値

sequence が単一の要素から構成され、pattern は最後の要素を除いて sequence と同じ要素で構成されている場合には（たとえば、sequence=[1, 1, 1, 1, 1], pattern=[1, 1, 2]）、sequence の各インデックスから pattern の長さに相当する m回の比較が行われて失敗する。ただし、sequence のインデックス $n-(m-1)$ では、$m-1$回の比較が終わると sequence の範囲を超えることになるので終了する。このため、$\{n-(m-1)\}m+(m-1) = nm - m^2 + 2m - 1$なので、

$(nm - m^2 + 2m - 1)$回 呼び出される。

pattern の長さ$m = 1$の場合は、sequence の長さに相当するn回の比較が行われる。上記の式はこの場合も成り立つ。

(3) 基本的なリスト操作プログラミング

基本的なリスト操作を理解していることを確認する。

are_lists_equal

```
def are_lists_equal(list1: list, list2: list) -> bool:
    if len(list1) != len(list2):
        return False
    for elem1, elem2 in zip(list1, list2):
        if not is_equal(elem1, elem2):
            return False
    return True
```

zip を用いないで range などを用いても良い。また、以下のような関数型プログラミングを用いても良い。

```
def are_lists_equal(list1: list, list2: list) -> bool:
    if len(list1) != len(list2):
        return False
    return all(map(is_equal, list1, list2))
```

(4) リスト操作(build_next のための準備)

説明文を理解し、LPS の長さの計算のためのリスト操作を理解できていることを確認する。

lps_length

```
def lps_length(list: list) -> int:
    length = len(list)-1
    while length > 0:
        if are_lists_equal(list[0:length],
                           list[len(list)-length:len(list)]):
            break
        length -= 1
    return length
```

(ア)　0:length

(イ)　len(list)-length:len(list)

(ア)の開始インデックスや(イ)の終了インデックスを省略してももちろん良い。

(5) find_with_next の準備

説明文の意味を理解していることを問う。

build_next

```
def build_next(pattern : list) -> list:
    plen = len(pattern)
    next = [-1] * plen
    for i in range(1, plen):
        next[i] = lps_length(pattern[0:i])
    return next
```

(6) KMP 法の考え方に基づいたパターンマッチ関数

説明文の意味を理解し、KMP 法の考え方を理解出来ていることを問う。

find_with_next

```
def find_with_next(sequence: list, pattern: list,
                           next: list) -> list:
    indices = []
    sindex = 0 # sequence のインデックス
    pindex = 0 # pattern のインデックス
    slen = len(sequence)  # sequence の長さ
    plen = len(pattern)   # pattern の長さ
    while sindex < slen and pindex < plen:
        while (pindex >= 0 and
               not is_equal(sequence[sindex], pattern[pindex])):
            pindex = next[pindex]
        sindex = sindex + 1
        pindex = pindex + 1
        if pindex >= plen:
            indices.append(sindex-plen) # 一致したインデックスを保存
            pindex = 0
    return indices
```

(ウ)　next[pindex]
(エ)　sindex + 1
(オ)　pindex + 1
(カ)　sindex – plen
(キ)　0

（7）特定のリストに対する KMP 法の考え方に基づいた関数における呼び出し回数の解析

14 回

sequence[0]から sequence[4]まで 5 回比較し、pattern と不一致となる。
next[4]→2 から sequence[4]と pattern[2]を比較し、sequence[5]と pattern[3]までの比較 2 回で不一致となる。
next[3]→1 であるから sequence[5]と pattern[1]を比較し、比較 1 回で不一致となる。
next[1]→0 であるから、sequence[5]と pattern[0]を比較し、比較 1 回で不一致となる。
next[0]→-1 であるから、sequence のインデックスを1つ進め、sequence[6]と pattern[0]を比較し、最後まで 5 回比較して一致となり、終了する。
上記の比較回数を合計すると、5+2+1+1+5 ＝ 14 となる。