2018 年度 情報領域演習第三
— 第 4 回 —

注意事項

- プログラムの形式が指定されている場合には，それに従うこと．従わなくても「成功」と表示されることがあるが，得点にはならない．
- 締切りは 1 週間後に設定しているが，演習時間終了時点での提出状況についても評価の対象となるので，時間内にできるだけ多くの問題に挑戦するとよい．

はじめに

前回同様，CED において以下のコマンド (青字の部分) を実行することで出席を表明できる．ただし，N は所属するクラス名 1 ，2 ，3 で置き換え，N の前には空白を入れないこと．

```
[p1610999@blue00 ~]> /ced-home/staff/18jr3/04/checkerN
提出開始: 10 月 xx 日 14 時 40 分
提出締切: 10 月 yy 日 14 時 39 分
ユーザ: p1610999, 出席状況: 2018-10-05-14-58
 問題:    結果 | 提出日時           | ハッシュ値                       |
    1:  未提出 |                    |                                  |
    2:  未提出 |                    |                                  |
                        ⋮
```

出席を表明するには，授業の開始から 30 分以内に実行する必要がある．実行しても出席状況が「欠席」である場合は教員に伝えること．なお，上の出力は例であるので，実際の締切りは各自コマンドで確認せよ．

提出も同じコマンドを用いて以下のように実行する．ただし，N はクラス名 (1 から 3)，X は 1 から 8 のいずれかの問題番号であり，prog.c は提出する C プログラムのファイル名であり，X の前後には忘れずに空白を入れること．

```
[p1610999@blue00 ~]> /ced-home/staff/18jr3/04/checkerN X prog.c
```

ここで，ファイル名として指定するのは，問題番号 X の問題を解く C プログラムのソースファイルであり，コンパイル済みの実行ファイルではない．ファイル名は特に指定しないが,「英数字からなる文字列.c」などとすることが望ましい．このコマンドを実行することにより，指定されたファイルがコンパイルされ，実行テストプログラムが自動的に起動される．コンパイルに失敗した場合にはエラーとなり，提出されたことにはならないので注意すること．コンパイルが成功した場合には複数回の実行テストが行われ，実行テストにも成功すると「成功」と出力される．実行テストに失敗すると「失敗」と出力されるとともに反例となる入力ファイルのパスが出力されるので，失敗した理由を見つけるための参考にするとよい．ただし，入力のない問題の場合は失敗しても反例は出力されない．

正しく提出できたか確認するためには以下のように出席の表明と同じコマンドを用いて確認できる．

```
[p1610999@blue00 ~]> /ced-home/staff/18jr3/04/checkerN
提出開始: 10 月 xx 日 14 時 40 分
提出締切: 10 月 yy 日 14 時 39 分
ユーザ: p1610999, 出席状況: 2018-10-05-14-47
 問題:    結果 | 提出日時           | ハッシュ値                       |
    1:  成功 | 2018-10-05-15-33   | 9b762c81932f3980cf03a768e044e65b |
                        ⋮
```

ハッシュ値は，提出した C プログラムのソースファイルの MD5 値である．CED では md5sum コマンドを用いて MD5 値を見ることにより，提出したファイルと手元のファイルが同じものであるかを確認することができる．

```
[p1610999@blue00 ~]> md5sum prog.c
9b762c81932f3980cf03a768e044e65b
```

なお，一度「成功」となった問題に対し，実行テストに失敗するプログラムを送信してしまうと，結果が「失敗」になってしまうので注意すること．

1 構造体とポインタ

前回の演習で，黄金比 φ によって $a+b\varphi$ $(a,b$ は整数$)$ と表される数を次のような構造体で表し，和・積・冪乗を計算する関数を実装した:

```
1 struct golden {
2   long long int a;
3   long long int b;
4 };
```

構造体を関数に渡すときは，メンバの値が渡されるだけである点に注意しよう．これを理解するために，まず，以下のようなプログラムを考えよう (構造体 golden はこの前に上のように宣言されているものとする).

```
1 void zero(struct golden x) { x.a = 0; x.b = 0; } /* メンバ a もメンバ b も 0 にする関数 (?) */
2 int main() {
3   struct golden x = {3, 2};
4   printf("a=%lld,␣b=%lld\n", x.a, x.b);
5   zero(x);
6   printf("a=%lld,␣b=%lld\n", x.a, x.b);
7 }
```

これをコンパイルして実行すると，4 行めでも 6 行めでも a=3, b=2 と出力されてしまい，関数 zero でのメンバ変更が反映されない．これは，zero(x) という関数呼び出しにおいて，変数 x の中身しか渡されていないためである．つまり，メンバ a が 3，メンバ b が 2 である構造体が渡され，関数 zero はそのメンバをどちらも 0 にしただけで，x 自身の値は変更していない．これは，次のプログラムを考えるとよくわかるかもしれない:

```
1 void f(int x) { x = 0; }
2 int main() {
3   int x = 3;
4   printf("x=%d\n", x);
5   f(x);
6   printf("x=%d\n", x);
7 }
```

このプログラムでも，4 行めも 6 行めも x=3 が出力され，f(x); によって変数 x の値は変化していない．これは関数 f に渡されるのは 3 という変数 x の中身だけだからである．

構造体の中身を変更する関数を作るにはポインタ (アドレスを表す変数) を用いて，

```
1 void zero(struct golden *p) { (*p).a = 0; (*p).b = 0; } /* メンバ a もメンバ b も 0 にする */
2 int main() {
3   struct golden x = {3, 2};
4   printf("a=%lld,␣b=%lld\n", x.a, x.b);
5   zero(&x);
6   printf("a=%lld,␣b=%lld\n", x.a, x.b);
7 }
```

とすればよい．ここでは，関数 zero を構造体のアドレス p を受け取るように変更していて，そのアドレス指している構造体 *p のメンバを 0 にする関数となっている．関数を呼び出すときは 5 行めのように構造体 x のアドレス &x を渡す．このように構造体はポインタとして利用されることが多く，(*p).a のような記述

を頻繁に行う必要があるため，C 言語ではそれを p->a と略記できるようになっている．ポインタ p が指す構造体のメンバ a と解釈すれば自然な表記と思えるかもしれない．なお，この略記を用いる際には p は必ず構造体のポインタ型であることを忘れてはならない．これを用いると上の関数 zero は，

```
1 void zero(struct golden *p) { p->a = 0; p->b = 0; } /* メンバ a もメンバ b も 0 にする */
```

と定義することができる．

なお，関数に構造体を渡すよりも構造体のアドレスを渡すほうが望ましい理由は他にもある．通常，C 言語の関数呼び出しは，まず引数のためのメモリを確保して，そこに実際の引数を格納し，関数の定義どおりに実行される．このため，関数に構造体を渡す場合，関数が呼び出されるたびに引数を表す構造体のためのメモリ (構造体のすべてのメンバの値を格納する領域) が確保されるが，関数に構造体のアドレスを渡す場合，引数のために確保されるメモリはアドレスの分だけで済む．一般にアドレスに必要なメモリは構造体に必要なメモリよりも小さいため，関数に構造体のアドレスを渡すほうが望ましい．

ポインタ宣言に関する注意　これまでポインタ変数については何度も触れてきたはずだが，改めて復習しておこう (知っている人にとっては少ししつこく感じるかもしれないが，今後セグメンテーション違反 (セグメントエラー) に苦しまないために念のために注意しておく)．ポインタ変数はアドレスを格納する変数のことで，格納される値の型が int 型ならポインタ変数は int* 型，struct golden 型ならポインタ変数は struct golden* 型と宣言する．変数 p を int 型のポインタ変数として宣言したければ，int* p; や int *p; のように宣言するが，後者をお勧めしておこう．前者の宣言の仕方で覚えてしまうと，変数 p と q をどちらも int 型のポインタ変数として宣言したいときに，int* p, q; と書いてしまうかもしれないが，これは int *p, q; という意味になり，p はポインタ変数になるが，q はただの int 型の変数になってしまう．後者の宣言の仕方で覚えていれば，int *p, *q; とすればよいことがわかる．

以下の宣言で何が起きるかが理解できていれば，セグメンテーション違反が起きてもすぐに原因がわかるようになるかもしれない．

```
1 int x;      /* 変数 x の値を格納する場所が用意される．値は未定                    */
2 int y = 42; /* 変数 y の値を格納する場所が用意される．値は 42                     */
3 int *p;     /* int 型のアドレスを格納する変数 p の場所が用意される．値(アドレス)は未定 */
```

変数 x が int 型の値であると宣言したからといって，1 行めの直後に printf("%d\n",x); などとすると何が出力されるかわからないことは想像できるだろう．これは x にはまだ何も代入していないためで，たまたまその場所に値が入っているかもしれないが，それがプログラマの意図した値であることは保証されていない．これと同様の理由で，3 行めの直後に *p = 42 などとしてはいけない．これは p の場所にあるものが未定であるためで，たまたまその場所に「適切なアドレス」が入っているかもしれないが，メモリ空間の中の使えるアドレスは限定されているので，運良く使えるアドレスであるとは限らない．「適切なアドレス」が必要であれば，(3 行めの後で) 以下のように，malloc 関数によってアドレスを用意してから int 型の値を代入する．

```
1 p = (int*) malloc(sizeof(int)); /* int 型に必要なメモリを用意して，そのアドレスを p に代入     */
2 *p = 42;                        /* 用意したアドレスが指す場所に int 型の値 42 を代入          */
3 printf("%d\n",*p);              /* 42が出力される.前の行がなければここでセグメンテーション違反 */
4 free(p);                        /* 使わなくなったメモリは解放                                 */
```

malloc 関数の引数は確保したいメモリのバイト数であるが，型によって必要なバイト数は異なる．C 言語には型に必要なバイト数を調べる sizeof 演算子が用意されているので，int 型なら sizeof(int) を引数として渡せばよい．また，malloc 関数が返すのは void* 型のアドレスなので int* 型へ型変換 (キャスト) する必要があることにも注意しよう．用意したアドレスが不要になったら，free 関数を使ってメモリを解放する．解放した後は p が指すアドレスはもはや「適切なアドレス」ではなくなるので，*p に値を代入してはいけない．

問題 1

まずは構造体を使わないポインタの問題から．

1 つの整数値を「状態」とする機械を考える．この機械は，整数が与えられるたびにその整数が状態の整数値に加算され，その結果が新たな状態となる．たとえば，状態が -5 のときに，整数 8 が読み込まれると状態が 3 になる．ただし，初期状態は 0 とし，整数の絶対値が 100 以上になる場合には 0 に戻るものとする．このとき，標準入力として与えられる整数に従って，更新された状態を標準出力に出力するプログラムを作成せよ．ただし，プログラムは指定された形式で作成するものとし，標準入力で与えられる整数は 1 行に 1 つずつ与えられ，各整数は `int` 型で表現できる範囲のもので，計算中に `int` 型の範囲を出ないと仮定してよい．また，標準出力には，1 行に 1 つずつ状態の整数値をを出力するものとする．

入力例

```
12
-52
32
101
-192
```

出力例

```
12
-40
-8
93
-99
```

入力例

```
34
66
-5
18
-72
-61
-15
```

出力例

```
34
0
-5
13
-59
0
-15
```

プログラムは以下の形式で記述するものとする．

```c
#include<stdio.h>
char buf[128];

/* p が指すアドレスにある整数に i を加算 */
void update(int *p, int i) {
  /* ここに適切な定義を書く */
  return;
}
/* main 関数の定義内は変更してはいけない */
int main() {
  int x = 0, i;
  while(fgets(buf, sizeof(buf), stdin) != NULL) {
    sscanf(buf, "%d", &i);
    update(&x, i);
    printf("%d\n", x);
  }
  return 0;
}
```

問題 2

次は構造体のポインタを使った問題．

2 つの整数値 (x, y) (xy-平面上の座標) を「状態」とする機械を考える．この機械は，整数の組 (a, b) (ベクトル) が与えられるたびに，このベクトルによって状態の座標にある点を平行移動した点の座標が新たな状態となる．たとえば，状態が $(3, -5)$ のときに，$(-4, 3)$ が読み込まれると状態が $(-1, -2)$ となる．ただし，初期状態は原点 $(0, 0)$ とし，原点からの距離が 100 以上になる場合には原点に戻るものとする．このとき，標準入力として与えられる整数の組に従って，更新された状態を標準出力に出力するプログラムを作成せよ．ただし，プログラムは指定された形式で作成するものとし，標準入力で与えられる整数の組は 1 行につき 1 組ずつ，空白でつないだ 2 つの整数として与えられ，各整数の絶対値は 2^{15} 以下で，計算中に

もこの範囲を出ないと仮定してよい．また，標準出力には，1 行に 1 組ずつ整数値の組を空白でつないだ 2 つの整数の形で出力するものとする．

入力例

```
12 -27
-19 75
20 20
10 -26
20 17
```

出力例

```
12 -27
-7 48
13 68
23 42
43 59
```

入力例

```
12 34
-56 78
90 -12
-34 45
40 -5
-78 9
2 57
```

出力例

```
12 34
0 0
90 -12
56 33
0 0
-78 9
0 0
```

プログラムは以下の形式で記述するものとする．

```c
#include<stdio.h>
char buf[128];

struct zahyo { int x; int y; };

/* p が指すアドレスにある構造体の座標を (i,j) だけ平行移動 */
void update(struct zahyo *p, int i, int j) {
  /* ここに適切な定義を書く */
  return;
}
/* main 関数の定義内は変更してはいけない */
int main() {
  struct zahyo xy = {0, 0};
  int i, j;
  while(fgets(buf, sizeof(buf), stdin) != NULL) {
    sscanf(buf, "%d␣%d", &i, &j);
    update(&xy, i, j);
    printf("%d␣%d\n", xy.x, xy.y);
  }
  return 0;
}
```

2 スタック

アルゴリズムを学習する上で最も基本的なデータ構造の 1 つが**スタック**である．スタックは，複数の要素を格納したり取り出したりできるデータ構造で，後に格納した要素から先に取り出せるという性質をもつ (Last-In-First-Out, LIFO)．慣例上，スタックに要素を格納することを**プッシュ** (push) といい，スタックから要素を取り出すことを**ポップ** (pop) という．

C 言語によるプログラミングでは，スタックを実現する方法として，連結リストを使う方法も考えられるが，今回は配列を使ってスタックを実現する．ただ，C 言語の配列は長さが固定されていて，スタックのように要素を増やしたり減らしたりすることはできないため，要素の個数の情報を含むような，以下のような構造体を使ってスタックを実現する:

```c
#define MAXSTACK 128            /* スタックの要素数の最大値 */
```

```
2 typedef char elementtype;
3 struct stack {
4   int top;                          /* 最後に入れた要素の位置（添字），スタックが空なら -1      */
5   elementtype contents[MAXSTACK]; /* 要素を含む配列．0番めからtop 番めまでがスタックの要素    */
6 };
```

ここでは，要素の型を elementtype としていて，2 行めで char 型の別名と定めている．「それならわざわざ elementtype なんて使わなくても char と書けばいいのではないか」と思うかもしれないが，このように別名を使うのには理由がある．それは，スタックの要素として他の型を使いたいとき，たとえば次の問題のように int 型を使いたいときは，2 行目を変更するだけで，スタックを表す構造体やそれを操作する関数をそのまま再利用できるからである．

問題 3

スタックを操作する以下の関数を実装し，操作列を解釈するプログラムを作成せよ:

- `void initstack(struct stack *p);`
  構造体 stack のアドレスを受け取り，そのスタックを空にする関数.
- `int stackempty(struct stack *p);`
  構造体 stack のアドレスを受け取り，そのスタックが空なら 1，空でなければ 0 を返す関数.
- `elementtype pop(struct stack *p);`
  構造体 stack のアドレスを受け取り，そのスタックから要素をポップし，その要素を返す関数.
- `void push(struct stack *p, elementtype e);`
  構造体 stack のアドレスを受け取り，そのスタックに e をプッシュする関数.

要素の型は int 型とし，標準入力として与えられるスタック操作の列は，1 行につき 1 つの操作が与えられ，各操作は整数値か文字 'p' のいずれかで表現されており，整数値の時はその数のプッシュ，文字 'p' ならポップの操作を表す．これに対し，作成すべきプログラムは，操作ごとに，その時点でスタックに含まれる各要素を [と] で囲み，先に入れた方から順に 列べたものを標準出力に出力する．作成するプログラムの形式は指定しないが，構造体 stack の定義と上の 4 つの操作関数の定義が必要である (引数や返り値の型を変更してはならない)．ただし，スタックが空になった状態でポップ操作が行われたときは Underflow と出力して終了し，スタックが満杯になった状態でプッシュ操作が行われたときは Overflow と出力して終了すること (いずれも exit(1); などで終了するとよい [1])．なお，MAXSTACK の値は 128 とする

入力例

```
12
3
45
p
67
p
p
```

出力例

```
[12]
[12][3]
[12][3][45]
[12][3]
[12][3][67]
[12][3]
[12]
```

[1] stdlib.h のインクルードが必要.

入力例

```
182
-8585
p
-1031
p
p
p
1101
-42
```

出力例

```
[182]
[182][-8585]
[182]
[182][-1031]
[182]

Underflow
 ↑ここでプログラムは終了
```

入力例

```
1
2
3
⋮
127
128
129
130
131
```

出力例

```
[1]
[1][2]
[1][2][3]
    ⋮
[1][2][3]···[127]
[1][2][3]···[127][128]
Overflow
 ↑ここでプログラムは終了
```

ヒント　入力の読み込みは以下のプログラムを参考にするとよい.

```
1  int main(){
2    struct stack s;
3    int i;
4    /* スタック s の初期化 */
5    while(fgets(buf,sizeof(buf),stdin) != NULL) {
6      if(buf[0]=='p') {
7        /* スタック s からのポップ操作 */
8      } else {
9        sscanf(buf,"%d", &i);
10       /* スタック s への i のプッシュ操作 */
11     }
12     /* スタック s の中身を指定された形式で出力 */
13   }
14 }
```

問題 4

標準入力の文字列に対し，全ての括弧 ((と), { と }, [と], < と >) が正しく入れ子になっているかを判定するプログラムを作成せよ.「正しく入れ子になっている」とは，開いていない括弧を閉じたり，開いた括弧を閉じなかったり，閉じていない括弧があるのに別の種類の括弧を閉じてしまったりすることがない，ということである．たとえば，文字列

```
{<University>(of)([Electro]-[Communications])}
Un(ive)rs[ity o]f E<lec(tr(o-C))omm[uni]{}>cations
Uni((versi)(((())))())ty of [E[l[e[c]]<t>]{<>}]ro-Communications
```

は，括弧も正しい入れ子になっているが，文字列

```
University(of)Electro->Communications   ←開いていない括弧を閉じている (>)
University[of]Electro<-Communications   ←開いた括弧 (<) を閉じていない
University(of<Electro)->Communications  ←閉じていない括弧 (<) があるのに
                                          別の種類の括弧を閉じている ())
```

は，いずれもそれぞれの右に示した理由により正しくない．標準入力には改行で終わる文字列が与えられ，作成すべきプログラムは，その文字列に対して全ての括弧が正しい入れ子になっていれば `Good` を，正しくなければ `Bad` を標準出力に出力する．なお，入力される文字列は 512 文字以下であると仮定してよい．

入力例

```
{<Univ>(of)([Elec]-[Comm])}
```

出力例

```
Good
```

入力例

```
Un(iv)[ o]f E<lec((-C))omm[]{}>
```

出力例

```
Good
```

入力例

```
Uni((v)((()))()) of [E[l[e[c]]<>]{<>}]-Comm
```

出力例

```
Good
```

入力例

```
Univ(of)Elec->Comm
```

出力例

```
Bad
```

入力例

```
Univ[of]Elec<-Comm
```

出力例

```
Bad
```

入力例

```
Univ(of<Elec)->Comm
```

出力例

```
Bad
```

問題 5

`char` 型の要素を持つ 2 次元配列 `canvas[N][N]` があるとする (`N` は 1 以上の奇数)．`canvas` の中央の位置 (`canvas[N/2][N/2]`) からはじめ，その上下左右にある空白文字を全て文字 `.` (ピリオド) に置き換えるプログラムを作成せよ．2 番目の入力例 (ハート型) のように，ハート型の外側の空白文字は置き換えの対象とならないことに注意せよ．また，作成するプログラムにおいては，`N` はマクロで定義され，11 であるものとし，中央 `canvas[N/2][N/2]` には空白があると仮定してよい．`canvas` にセットする文字列の読み込みと結果の表示する `main` 関数をヒントに示しているので参考にするとよい．

入力例

```
+---------+
|         |
|         |
+----     |
|         |
|         |
|         |
|     ----+
|         |
|         |
+---------+
```

出力例

```
+---------+
|.........|
|.........|
+----.....|
|.........|
|.........|
|.........|
|.....----+
|.........|
|.........|
+---------+
```

入力例

```
  __   __
 /  \ /  \
|    v    |
|         |
|         |
 \       /
  \     /
   \   /
    \ /
     V
```

出力例

```
  __   __
 /..\ /..\
|....v....|
|.........|
|.........|
 \......./
  \...../
   \.../
    \./
     V
```

入力例

```
  _______
 /       \
|    ____/
|   /
|  |
|  |
|   \____
|        \
 \_______/
```

出力例

```
...........
..._______..
./       \.
|    ____/.
|   /......
|  |.......
|  |.......
|   \____..
|        \.
.\_______/.
...........
```

ヒント

- 入力を配列に読み取るには次のプログラムを参考にするとよい．このプログラムは入力となる N ×N の部屋のレイアウトを読み取り，それを char 型の 2 次元配列 canvas に格納するプログラムで，canvas[i][j] に座標 (i,j) にある文字が格納される．このプログラムは最後に canvas の内容を標準出力に出力するため，このまま実行すると，入力した N ×N の内容をそのまま出力するプログラムとなっている．本問題は，このプログラム内の /* ここを埋めればよい */ のところを適切に記述すればよい．入力ファイルは演習の Web ページに載せているので適宜ダウンロードせよ．

```
1  #include<stdio.h>
2  #define N 11
3  char buf[N+2];
4  char canvas[N][N];
5
6  struct zahyo { int x, y; };
7
8  int main() {
9    int i, j;
10   i = 0;
11   /* 入力を 1行ずつ読み込んで heya へ格納 */
12   while( fgets(buf,sizeof(buf),stdin) != NULL ) {
13     for(j=0;j<N;++j)
14       canvas[i][j] = buf[j];
15     ++i;
16   }
17
18   /* ここを埋めればよい */
19
20   /* canvas の出力 */
```

```
21   for(i=0;i<N;++i) {
22     for(j=0;j<N;++j)
23       printf("%c",canvas[i][j]);
24     printf("\n");
25   }
26   return 0;
27 }
```

- 座標を要素とするスタックを用いて以下のようなアルゴリズムを考えればよい.

  1. 空のスタックを用意して，中心の座標 (N/2, N/2) をプッシュする.
  2. スタックが空になるまで以下を繰り返す.
     (i) スタックのトップから座標 (p_x, p_y) をポップ
     (ii) (p_x, p_y) の上下左右の全ての座標 (x, y) について，以下を行う.
     「もし，canvas[x][y] が空白ならば，スタックに (x, y) をプッシュして，canvas[x][y] に '.' を入れる」

3 キュー

スタックと並んで最も基本的であるデータ構造が**キュー**である．キューもスタックと同様，複数の要素を格納したり取り出したりできるデータ構造であるが，先に格納した要素から先に取り出せるという性質をもつ (First-In-First-Out, FIFO)．C 言語では，以下のような構造体を使ってキューを実現できる:

```
1 #define MAXQUEUE 128              /* キューの要素を格納する配列の大きさ                          */
2 typedef char elementtype;
3 struct queue {
4   int front;                      /* 次に要素を取り出す位置（添字），rear と同じならキューは空 */
5   int rear;                       /* 次に要素を入れるべき位置（添字）                           */
6   elementtype contents[MAXQUEUE]; /* 要素を含む配列．front 番めから rear-1番めがスタックの要素 */
7 };
```

スタックと違い，最後に入れた要素と次に取り出す要素が異なるので，配列を用いたキューの実装では，それぞれの添字を記憶しておく必要がある．メンバ front が次に取り出す位置，つまり，キューに格納されている要素のうち，最初に入れた要素の添字を表し，メンバ rear が次に要素を入れる位置，つまり，キューに格納されている要素のうち，最後に入れた要素の添字に 1 加えた数である．キューに何も要素がないとき (空のとき) は，front と rear は等しくなる．

問題 6

キューを操作する以下の関数を実装し，操作列を解釈するプログラムを作成せよ:

- `void initqueue(struct queue *p);`
  構造体 queue のアドレスを受け取り，そのキューを空にする関数.
- `int queueempty(struct queue *p);`
  構造体 queue のアドレスを受け取り，そのキューが空なら 0，空でなければ 1 を返す関数.
- `elementtype getq(struct queue *p);`
  構造体 queue のアドレスを受け取り，そのキューから要素を取り出し，その要素を返す関数.
- `void putq(struct queue *p, elementtype e);`
  構造体 queue のアドレスを受け取り，そのキューに e を入れる関数.

本問題で実装するキューでは，キューに要素を入れるたびに rear が増え，キューから要素取り出すたびに front が増えるので，front や rear はどちらも MAXQUEUE まで増え続ける．このため，キューに格納されている要素の数が MAXQUEUE に達していなくても，それ以上格納できなくなる場合がある．これを回避するプログラムは 問題 7 で扱う．

要素の型は int 型とし，標準入力として与えられるキュー操作の列は，1 行につき 1 つの操作が与えられ，各操作は整数値か文字 'g' のいずれかで表現されており，整数値の時はその数をキューに入れ，文字 'g' ならキューから取り出す操作を表す．これに対し，作成すべきプログラムは，操作ごとに，その時点でキューに含まれる各要素を [と] で囲み，先に入れた方から順に 列べたものを標準出力に出力する．作成するプログラムの形式は指定しないが，構造体 queue の定義と上の 4 つの操作関数の定義が必要である (引数や返り値の型を変更してはならない)．ただし，キューが空になった状態で要素を取り出す操作が行われたときは Underflow と出力して終了し，front が MAXQUEUE になった状態で要素を入れる操作が行われたときは Sorry と出力して終了すること (Overflow ではないことに注意)．なお，MAXQUEUE の値は 128 とする

入力例

```
12
3
45
g
67
g
g
```

出力例

```
[12]
[12][3]
[12][3][45]
[3][45]
[3][45][67]
[45][67]
[67]
```

入力例

```
182
-8585
g
-1031
g
g
g
1101
-42
```

出力例

```
[182]
[182][-8585]
[-8585]
[-8585][-1031]
[-1031]

Underflow
 ↑ここでプログラムは終了
```

入力例

```
1
2
3
⋮
127
128
129
130
131
```

出力例

```
[1]
[1][2]
[1][2][3]
⋮
[1][2][3]···[127]
[1][2][3]···[127][128]
Sorry
 ↑ここでプログラムは終了
```

入力例

```
0
g
0
g
0
g
⋮（この間に 0，g が交互に 124 回ずつ）
0
g
0
g
0
```

出力例

```
[0]

[0]

[0]

⋮（この間に [0] と空行が交互に 124 回ずつ）
[0]

Sorry
 ↑ここでプログラムは終了
```

ヒント　入力の読み込みは問題 3 を参考にするとよい．

問題 7

前の問題におけるキューの実装では，front と rear がともに増加し続けるため，仮にキューが空であったとしても，それ以上要素が格納できない場合がありうる．これを回避するために，front (または rear) が MAXQUEUE に達したら 0 に戻る，というように実装すると，MAXQUEUE の分だけ要素をキューに格納できる．問題 6 と同様の設定で，この改善を行ったプログラムを作成せよ．ただし，実際にこの実装を行うとわかるかもしれないが，キューが満杯の状態においては front (次に取り出す位置) と rear (次に格納する位置) が一致するため，キューが空の状態と区別することができない．そこで，本問題で実装するキューは最大で MAXQUEUE より 1 つ少ない数だけ要素を格納できるものとし，その状態から要素を入れる操作が行われたときに Overflow と出力して終了する．なお，この問題では Sorry と出力されることはないことに注意せよ．入出力の例は 問題 6 とほぼ同じであるが，異なるものは以下の通り．

入力例

```
1
2
3
⋮
127
128
129
130
131
```

出力例

```
[1]
[1][2]
[1][2][3]
    ⋮
[1][2][3]···[127]
Overflow
 ↑ここでプログラムは終了
```

入力例

```
0
g
0
g
0
g
⋮（この間に 0，g が交互に 124 回ずつ）
0
g
0
g
0
```

出力例

```
[0]

[0]

[0]

⋮（この間に [0] と空行が交互に 124 回ずつ）
[0]

[0]

[0]
```

問題 8

下の入力例に示すような，左上の座標を (0,0)，右下の座標を (9,9) とする部屋 (下向きが x 軸，右向きが y 軸) があり，* は通ることのできない壁であるとし，S はスタート地点，G はゴール地点とする．このような部屋において，上下左右にしか移動できないコマが指定したスタート S からゴール G までの最短距離を計算するプログラムを作成せよ．ここで，S から G までの最短距離とは，S から G に移動するまでに通るコマ (S 自身は除く) の数の最小値である．標準入力は，10×10 の部屋のレイアウトで，周囲は全て壁であり，S と G は必ず 1 つずつ存在し，S から G までたどり着ける道があるものと仮定してよい．作成すべきプログラムは S から G までの最短距離を標準出力に出力する．

入力例

```
**********
*S       *
*        *
*        *
*        *
*        *
*        *
*        *
*       G*
**********
```

出力例

```
14
```

入力例

```
**********
*       S*
*    *****
*        *
*        *
*        *
*        *
*****    *
*G       *
**********
```

出力例

```
16
```

入力例

```
**********
*        *
*S  *    *
*****    *
*        *
*    *****
*    *  G*
*    *   *
*        *
**********
```

出力例

```
19
```

ヒント

- 入力を配列に読み取るには次のプログラムを参考にするとよい．このプログラムは入力となる N ×N の部屋のレイアウトを読み取り，それを char 型の 2 次元配列 heya に格納するプログラムで，heya[i][j] に座標 (i,j) にある文字が格納され，スタート地点とゴール地点の座標がそれぞれ start と goal に，問題 2 で利用した構造体 zahyo として記録されている．このプログラムは何も出力しないが，最後に heya の内容や start , goal に入っている座標を出力するように変更すると，動作確認できるだろう．部屋の入力ファイルは演習の Web ページに載せているので適宜ダウンロードするとよい．

```c
#include<stdio.h>
#define N 10
char buf[N+2];
char heya[N][N];

struct zahyo { int x, y; };

int main() {
  int i, j;
  struct zahyo start, goal;

  /* 入力を 1行ずつ読み込んで heya へ格納 */
  i = 0;
  while( fgets(buf,sizeof(buf),stdin) != NULL ) {
    for(j=0;j<N;++j) {
      if (buf[j]=='S') { /* 文字が S のとき start に記録 */
        start.x = i; start.y = j;
      } else if (buf[j]=='G') { /* 文字が G のとき start に記録 */
        goal.x = i; goal.y = j;
      }
      heya[i][j] = buf[j];
    }
    ++i;
  }
  return 0;
}
```

- 座標を要素とするキューを用いて以下のようなアルゴリズムを考えればよい.

  1. スタート地点からの距離を格納するための 2 次元配列 `kyori[N][N]` を用意し，全ての値を `-1` で初期化しておく (まだ訪問してないなら `-1` ，訪問していたら最短距離が格納される).
  2. 空のキューを用意して，スタート地点の座標 (s_x, s_y) をキューに入れる.
  3. `kyori[`s_x`][`s_y`]` に `0` を入れる.
  4. キューが空になるまで以下を繰り返す.
     (i) キューの先頭の座標 (c_x, c_y) を取り出す.
     (ii) (c_x, c_y) の上下左右の全ての座標 (x, y) について，以下を行う.
     「もし `kyori[`x`][`y`]` の値が `-1` (つまり未訪問) かつ `heya[`x`][`y`]` が壁でないならば，キューに (x, y) を入れて，`kyori[`x`][`y`]` を `kyori[`c_x`][`c_y`]+1` とする」
  5. ゴール地点における `kyori` の値にゴールまでの最短距離が格納されている.

  このアルゴリズムのアイデアは「まだ訪問していない場所に訪問するたびに，その位置をキューに追加し，その位置の最短距離の情報を更新する」というものである．スタート地点から近いものから順にキューに格納されることに注意すれば，このアルゴリズムが正しく最短距離を計算していることがわかる.

まとめ

問題 4 において，スタックでなくキューを使ってしまうと，直前に閉じた括弧がわからないため，正しく括弧の入れ子が判定できない．逆に，問題 8 において，キューでなくスタックを使ってしまうと，最短でない経路で先に訪問することがあるため，正しく計算されない．なお，問題 5 はスタックで解くアルゴリズムを紹介しているが，この問題についてはキューでも解くことができる.