2018 年度 情報領域演習第三
— 第 7 回 —

注意事項

- 今回から整列アルゴリズムについて実装を通じて学習する．問題文だけでなく，その前にある説明をよく読み，必要であれば参考図書等で自習するとよい．わからないことがあれば TA や教員に質問しよう．
- 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．また，遅刻や途中退室は記録に残るので注意すること．
- 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．また，採点の際は `checker` とは異なる入出力例を用いて動作確認するので，「失敗」する反例に対する小手先の対策だけでは得点にならないことがあるので注意せよ．
- `checker` コマンドの実行時間には上限 (十分長いと考えられる値) が設けられている．時間内に終了しない場合にはチェック対象のプログラムの実行が中断され「失敗」と判定される．このようなときには明らかな誤り (無限ループなど) あるいは冗長なプログラミングによる非効率が原因と考えられるのでプログラムを見直すこと．

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/18jr3/07/checkerN
提出開始: 11 月 xx 日 14 時 40 分 0 秒
提出締切: 11 月 yy 日 14 時 39 分 0 秒
ユーザ: p1610999, 出席状況: 2018-11-xx-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 7 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/18jr3/07/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」と表示される．実行テストに失敗すると「失敗」と表示されるとともに反例となる入力と出力が表示されるので，失敗した理由を見つけるための参考にするとよい．なお，入出力が大きい場合やファイルとして入力したい場合には共に表示されるパスにあるファイルを使うと便利である．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/18jr3/07/checkerN
提出開始: 11 月 xx 日 14 時 40 分 0 秒
提出締切: 11 月 yy 日 14 時 39 分 0 秒
ユーザ: p1610999, 出席状況: 2018-11-xx-14-47
 問題:    結果 | 提出日時           | ハッシュ値                        |
    1:    成功 | 2018-11-xx-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 アルゴリズム

問題を解くための形式的な手順のことをアルゴリズムという．たとえば,「与えられた正の整数に対し，その約数を列挙せよ」という問題であれば,「その数を 1 から順に割っていって割り切るものだけを出力する」という手順はアルゴリズムと言える．同じ問題であってもアルゴリズムは 1 つとは限らないことに注意しよう．たとえば,「与えられた 2 つの正の整数に対しそれらの最大公約数を求めよ」という問題であれば，互いに引き算 (または割り算) をしていく互除法というアルゴリズムもあるし，それぞれの約数を求めてから共通する最大のものを求めるというアルゴリズムもある.「与えられた配列を小さい順に並べ替えよ」という整列の問題に対してであれば，選択ソートや挿入ソート，クイックソート，マージソートなど多くのアルゴリズムをプログラミング通論で学習したことであろう．

同じ問題を解くだけなのにこんなにアルゴリズムを学習する必要があるのか，と疑問に思った人もいるかもしれない．同じ問題を解くアルゴリズムであっても，それぞれに特徴があり，最適なアルゴリズムは 1 つとは限らない．実行時間，使用するメモリ，プログラムの簡潔さ，など様々な規準によって適切にアルゴリズムを選択する必要がある．

今回からの演習は，与えられたアルゴリズムを適切にプログラムに書き下せるようになるためのものである．なぜそのアルゴリズムによって問題が解けるかも考えながらプログラムを書いてみるとよくわかるかもしれない．

1.1 最大値探索

まず，整列問題より簡単な「最大値を探す問題」に挑戦しよう．あとで見るように，この問題は整列問題を解く上でも重要な足がかりとなる．

問題 1

配列で与えられた複数の数値のうち最大となるものの添字を求める関数 `max_index` を定義せよ．引数と返り値の型は以下の通りである:

```
int max_index(int a[], int n);
```

この関数は,「`int` 型の値を要素にもつ配列 `a` に対し，先頭から `n` 個 (つまり `0` 番めから `n-1` 番めまで) のうちの最大値の添字を返す関数」である．最大値ではなく最大値の添字を返すということに注意せよ．たとえば，`{4, 6, 2, 8, 7}`という配列であれば，最大値は `8` なのでその添字である `3` を返す (先頭が `0` 番めであることに注意)．最大値となる要素が複数あった場合には，そのうちで最も末尾に近いものの添字を返すものとする．作成すべきプログラムの形式は後に示す `main` 関数を含むものとし，関数 `max_index` の定義を追加することでプログラムを完成させよ．標準入力は 1 行以上 128 行以下でどの行にも 1 つの `int` 型の整

数値を表す文字列だけが含まれているものとする．

入力例

```
4
6
2
8
7
```

出力例

```
3
```

入力例

```
-182
123
456
-7890
```

出力例

```
2
```

入力例

```
2
7
0
2
0
1
7
```

出力例

```
6
```

作成すべきプログラムは以下の形式とする．※印を含むコメント以外は書き換えてはならない．

```c
#include<stdio.h>

int max_index(int a[], int n) {
  /* ※ここを適切なプログラムで埋める */
}

int main() {
  char buf[128];
  int arr[128], i = 0;
  while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128)
    sscanf(buf,"%d",&arr[i++]);
  printf("%d\n", max_index(arr, i));
  return 0;
}
```

問題 2

次のような xy 平面上の格子点の座標を表す構造体 `point` を考えよう．

```c
struct point { int x, y; };
```

メンバ `x` が x 座標を表し，メンバ `y` が y 座標を表す．問題 1 と同様に，配列で与えられた複数の格子点の座標のうち最大となるものを見つける関数を定義したいが，「座標が最大」というのはどう判断したらいいだろうか．たとえば，2 つの座標 $(2,3)$ と $(1,4)$ は，x 座標を比較すれば前者の方が大きいが，y 座標を比較すれば後者の方が大きい．そこで，座標を比較するための規準として次の 3 つの規準を考える:

規準 **X** x 座標が大きい方を「大きい座標」とする．x 座標が等しい場合には，y 座標が大きい方を「大きい座標」とする．

規準 **Y** y 座標が大きい方を「大きい座標」とする．y 座標が等しい場合には，x 座標が大きい方を「大きい座標」とする．

規準 **D** 原点からの距離が遠い方を「大きい座標」とする．原点からの距離が等しい場合には，x 座標が大きい方を「大きい座標」とする．x 座標も等しい場合には，y 座標が大きい方を「大きい座標」とする．

この問題では，まず，それぞれの規準で座標を比較するための関数 `compare_by` を定義しよう．関数 `compare_by` の引数と返り値の型は以下の通りである [1]:

```
int compare_by(struct point p1, struct point p2, char c);
```

この関数は，「`struct point` 型で与えられる 2 つの座標 `p1` と `p2` に対し，文字 `c` で指定された規準で比較した結果を `1`，`0`，`-1` として返す関数」である．文字 `c` は `'X'`，`'Y'`，`'D'` のいずれかであり，それぞれ，規準 **X**，規準 **Y**，規準 **D** に対応する．また，返り値は，`1` であれば `p1` が `p2` より大きい，`0` であれば `p1` と `p2` が等しい，`-1` であれば `p2` が `p1` より大きい，と解釈する．

この問題では，まず関数 `compare_by` を定義し，それを利用して，配列で与えられた複数の格子点の座標のうち，指定された規準で最大となるものの添字を求める関数 `max_index_by` を定義することが目標である．関数 `max_index_by` の引数と返り値の型は以下の通りである:

```
int max_index_by(struct point a[], int n, char c);
```

この関数は，「`struct point` 型の座標を要素にもつ配列 `a` に対し，先頭から `n` 個 (つまり `0` 番めから `n-1` 番めまで) のうち，文字 `c` で指定された規準で最大の要素の添字を返す関数」である．問題 1 と同じく，最大値ではなく最大値の添字を返すということに注意せよ．最大となる座標が複数あった場合には，そのうちで最も末尾に近いものの添字を返すものとする．作成すべきプログラムの形式は後に示す `main` 関数を含むものとし，関数 `compare_by` と関数 `max_index_by` の定義を追加することでプログラムを完成させよ．標準入力は 2 行以上 129 行以下で，最初の行は規準を表す `'X'`，`'Y'`，`'D'` のいずれか 1 文字，2 行め以降の各行には 2 つの `int` 型の整数値を空白で挟んだ文字列が含まれているものとする．

入力例

```
X
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
2
```

入力例

```
Y
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
1
```

入力例

```
D
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
4
```

作成すべきプログラムは以下の形式とする．※印を含むコメント以外は書き換えてはならない．

[1] 以前述べたように関数の引数として構造体を渡すときはポインタを渡す方が望ましいが，ここでは簡単のために構造体を渡している．

```c
#include<stdio.h>

struct point { int x, y; };

int compare_by(struct point p1, struct point p2, char c) {
  /* ※ここを適切なプログラムで埋める */
}

int max_index_by(struct point a[], int n, char c) {
  /* ※compare_by を利用して適切に埋める */
}

int main() {
  char c, buf[128];
  struct point p, arr[128];
  int i = 0;
  scanf("%c␣",&c);
  while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
    sscanf(buf,"%d␣%d",&p.x,&p.y);
    arr[i] = p;
    ++i;
  }
  printf("%d\n", max_index_by(arr, i, c));
  return 0;
}
```

ヒント　以前の課題にも「原点からの距離」を比較する問題があったが，今回についても $\sqrt{x^2+y^2}$ のような具体的な距離を計算する必要はないことに注意せよ．互いにどちらが遠いかの比較ができればよいので x^2+y^2 を計算すればよく，平方根をとって距離を求める必要はない．

1.2　単純な整列アルゴリズム

配列を特定の順序でならべかえる整列問題に挑戦しよう．問題 2 で扱った構造体 `point` を要素とする配列を指定された規準で昇順に整列する問題である．今回は単純な整列アルゴリズムを 3 つ実装してもらう．すでにアルゴリズム論第一で計算量という概念を学習したはずだが，これらの整列アルゴリズムは入力の大きさ (配列の長さ) n に対して，実行時間が $O(n^2)$ だけかかることが知られている．つまり，計算時間は n の 2 乗に比例する関数で近似できることになる．今回の課題では，このことを実感するために，作成するプログラムでは比較回数も数えるものとする．

問題 3

最も理解しやすく単純な整列アルゴリズムが選択ソート (*selection sort*) である．この問題の主な目的は，構造体 `point` を要素とする配列 `a` に対して選択ソートを行う関数 `selection_sort` を定義することである．引数と返り値の型は以下の通りである:

```c
void selection_sort(struct point a[], int n, char c);
```

この関数は「配列 `a` のうち，先頭 `n` 個 (0 番めから `n-1` 番めまで) について，文字 `c` で指定された規準で昇順に整列する関数」である．この関数は，以下のアルゴリズムの通りに，`max_index_by` を繰り返し用いれば簡単に実現できる [2]．

`k` を `n` から 2 まで 1 ずつ減らしながら以下を繰り返す：

[2] この方法では最大値を末尾から詰めていく方式を取っているが，最小値を先頭から詰めていく方式による選択ソートもある．

1. `max_index_by` により配列 `a` の先頭 `k` 個の最大値の添字を計算し，それを `i` とする．
2. `i` 番めと `k-1` 番めを交換 (この時点で `k-1` 番め以降の要素は整列されている)

作成すべきプログラムは，標準入力から問題 2 と同様の入力を受け取り，規準を表す文字に基づいて整列を行い，比較回数と整列した結果を標準出力に出力するプログラムである．ここで,「比較回数」とは要素同士の比較を行った回数のことで，関数 `compare_by` を呼び出した回数である．作成すべきプログラムの形式は後に示す形式とする．

入力例

```
X
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
10  ←これが比較回数
-4 2
-3 2
0 -5
3 -4
3 1
```

入力例

```
Y
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
10
0 -5
3 -4
3 1
-4 2
-3 2
```

入力例

```
D
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
10
3 1
-3 2
-4 2
0 -5
3 -4
```

作成すべきプログラムは以下の形式とする．※印を含むコメント以外は書き換えてはならない．

```c
#include<stdio.h>
int count = 0; /* ←比較回数を数えるグローバル変数 */

struct point { int x, y; };

int compare_by(struct point p1, struct point p2, char c) {
  ++count;     /* ←ここで比較回数を 1回分カウント */
  /* ※ ここは問題 2 と同じ  */
}

int max_index_by(struct point a[], int n, char c) {
  /* ※ ここも問題 2 と同じ  */
}

void selection_sort(struct point a[], int n, char c) {
  /* ※ ここはアルゴリズムに従って適切に埋める */
}

int main() {
  char c, buf[128];
  struct point p, arr[128];
```

```
22   int i = 0, n;
23   scanf("%c␣",&c);
24   while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
25     sscanf(buf,"%d␣%d",&p.x,&p.y);
26     arr[i] = p;
27     ++i;
28   }
29   n = i;
30   selection_sort(arr, n, c);
31   printf("%d\n", count);
32   for(i=0;i<n;++i)
33     printf("%d␣%d\n", arr[i].x, arr[i].y);
34   return 0;
35 }
```

いろいろな入力で試してみるとわかる (試さなくてもわかるかもしれない) が，入力の長さが同じならば必ず比較回数は等しくなる．具体的には入力となる配列の要素が n 個のときは，比較回数は $\frac{n(n-1)}{2}$ 回になる．これは，最初に n 個の最大値を求めるために $n-1$ 回の比較を行い，次に $n-1$ 個の最大値を求めるために $n-2$ 回の比較を行い，...，と繰り返すためで，総比較回数は

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\frac{n(n-1)}{2}$$

と計算できる．すなわち，計算量は $O(n^2)$ である．

次の整列アルゴリズムの挿入ソート (*insertion sort*) も，選択ソートと同じ計算量 $O(n^2)$ を持つ整列アルゴリズムであるが，入力によっては比較回数が節約できることを確認してみよう．

問題 4

まず，挿入ソートに必要な補助関数 `insert_by` を定義しよう．引数と返り値の型は以下の通りである:

```
void insert_by(struct point a[], int n, char c);
```

この関数は,「配列 `a` の先頭 `n-1` 個 (つまり `0` 番めから `n-2` 番めまで) が文字 `c` の規準で昇順に整列されているとき，`a[n-1]` を適切な位置に移動して，配列 `a` の先頭 `n` 個が昇順に整列された状態にする関数」である．

まず，問題をわかりやすくするために，各要素が構造体 `point` ではなく整数である場合を考えよう．たとえば，`{2, 3, 7, 11, 5}` という配列の先頭 5 個 (つまり全部) に対して，この関数を適用すると，`{2, 3, 5, 7, 11}`という配列になる．つまり,「末尾以外が整列されている状態のときに，末尾を適切な位置に移動して，末尾も含めて整列されている状態にする」という関数である．末尾の数が大きければ先頭に近い方は移動しなくてもよいので，末尾から順に比較して大きければ後ろに移動させるという方法が効率がよい．

この問題では，引き続き構造体 `point` を要素とする配列を対象とするため，問題 2 で定義した `compare_by` を利用して大小比較を行う．作成すべきプログラムは，標準入力から問題 2 と同様の入力を受け取り，規準を表す文字に基づいて末尾の要素を挿入するプログラムである．末尾 (最後の行) を除く全ての要素は，入力する段階でその規準で整列されているものと仮定してよい．作成すべきプログラムの形式は後に示す．

入力例

```
X
-4 2
0 -5
3 -4
3 1   ← この行までは整列されている
-3 2 ← これが挿入すべき要素
```

出力例

```
-4 2
-3 2
0 -5
3 -4
3 1
```

入力例

```
Y
3 -4
3 1
-4 2
-3 2 ← この行までは整列されている
0 -5 ← これが挿入すべき要素
```

出力例

```
0 -5
3 -4
3 1
-4 2
-3 2
```

入力例

```
D
3 1
-3 2
-4 2
0 -5 ← この行までは整列されている
3 -4 ← これが挿入すべき要素
```

出力例

```
3 1
-3 2
-4 2
0 -5
3 -4
```

作成すべきプログラムは以下の形式とする．※印を含むコメント以外は書き換えてはならない．

```c
#include<stdio.h>

struct point { int x, y; };

int compare_by(struct point p1, struct point p2, char c) {
  /* ※ ここは問題2と同じ */
}

void insert_by(struct point a[], int n, char c) {
  /* ※ ここを適切なプログラムで埋める */
}

int main() {
  char c, buf[128];
  struct point p, arr[128];
  int i = 0, n;
  scanf("%c␣",&c);
  while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
    sscanf(buf,"%d␣%d",&p.x,&p.y);
    arr[i] = p;
    ++i;
  }
  n = i;
  insert_by(arr, n, c);
  for(i=0;i<n;++i)
    printf("%d␣%d\n", arr[i].x, arr[i].y);
  return 0;
}
```

問題 5

単純でありながら比較回数を節約できる整列アルゴリズムが挿入ソート (*insertion sort*) である．この問題の主な目的は，構造体 `point` を要素とする配列 `a` に対して挿入ソートを行う関数 `insertion_sort` を定義することである．引数と返り値の型は以下の通りである:

```c
void insertion_sort(struct point a[], int n, char c);
```

この関数は「配列 a のうち，先頭 n 個 (0 番めから n-1 番めまで) について，文字 c で指定された規準で昇順に整列する関数」である．この関数は，以下のアルゴリズムの通りに，insert_by を繰り返し用いれば簡単に実現できる．

> k を 2 から n まで 1 ずつ増やしながら以下を繰り返す：
>
> > (配列 a の先頭 k-1 個が整列された状態になっているので)
> > insert_by により配列 a の先頭 k 個を整列された状態にする．

最初は何も整列されていないが，先頭 1 つの要素だけ見れば (長さ 1 の) 整列された状態である．その末尾に 1 つ要素を追加して長さを 2 として関数 insert_by を使えば，先頭 2 つの要素が整列された状態になる．その末尾に 1 つ要素を追加して長さを 3 として関数 insert_by を使えば，先頭 3 つの要素が整列された状態になる，...，と長さ n が整列されるまで繰り返すわけである．

作成すべきプログラムは，標準入力から問題 2 と同様の入力を受け取り，規準を表す文字に基づいて整列を行い，比較回数と整列した結果を標準出力に出力するプログラムである．ここで,「比較回数」とは要素同士の比較を行った回数のことで，関数 compare_by を呼び出した回数である．作成すべきプログラムの形式は後に示す形式とする．

入力例

```
X
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
7     ←これが比較回数
-4 2
-3 2
0 -5
3 -4
3 1
```

入力例

```
Y
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
9
0 -5
3 -4
3 1
-4 2
-3 2
```

入力例

```
D
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
6
3 1
-3 2
-4 2
0 -5
3 -4
```

作成すべきプログラムは以下の形式とする．※印を含むコメント以外は書き換えてはならない．

```c
#include<stdio.h>

int count = 0;

struct point { int x, y; };

int compare_by(struct point p1, struct point p2, char c) {
  ++count;
  /* ※ ここは問題2と同じ */
}

```

```c
void insert_by(struct point a[], int n, char c) {
  /* ※ ここは問題 4 と同じ */
}

void insertion_sort(struct point a[], int n, char c) {
  /* ※ ここを適切なプログラムで埋める */
}

int main() {
  char c, buf[128];
  struct point p, arr[128];
  int i = 0, n;
  scanf("%c␣",&c);
  while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
    sscanf(buf,"%d␣%d",&p.x,&p.y);
    arr[i] = p;
    ++i;
  }
  n = i;
  insertion_sort(arr, n, c);
  printf("%d\n", count);
  for(i=0;i<n;++i)
    printf("%d␣%d\n", arr[i].x, arr[i].y);
  return 0;
}
```

上の入出力例からもわかるように，挿入ソートにおける比較回数は入力の長さが同じでも入力の順序によって異なる．具体的には，挿入ソートにおける比較回数は

(長さ 2 に対する `insert_by`における比較回数)
+ (長さ 3 に対する `insert_by`における比較回数)
+ ...
+ (長さ n に対する `insert_by`における比較回数)

と計算でき，一般的に「長さ i に対する `insert_by` における比較回数」は最大 $i-1$ ではあるものの $i-1$ より小さいことが多いので，総比較回数は

$$(2-1)+(3-1)+\cdots+(n-1)=\frac{n(n-1)}{2}$$

よりも小さくなることが多い．実際，入力がすでに整列されている場合には，各回の `insert_by` において 1 回しか比較されないため，総比較回数は $n-1$ 回となる．一方，入力が整列したい順序とは逆順に列んでいる場合には，毎回先頭まで比較することになるため，比較回数が $\frac{n(n-1)}{2}$ となり最大となる．

問題 6

選択ソート，挿入ソートと並んで単純な整列アルゴリズムがバブルソートである．バブルソートでは，隣同士の交換だけを繰り返して，水の中の泡がぶくぶく上がっていくように，小さな値は前に，大きな値は後ろに徐々に移動することで昇順に整列される．この問題の主な目的は，構造体 `point` を要素とする配列 `a` に対してバブルソートを行う関数 `bubble_sort` を定義することである．引数と返り値の型は以下の通りである:

```c
void bubble_sort(struct point a[], int n, char c);
```

この関数は「配列 `a` のうち，先頭 `n` 個 (`0` 番めから `n-1` 番めまで) について，文字 `c` で指定された規準で昇順に整列する関数」である．この関数は，以下のアルゴリズムの通りに実装すること．

k を n-2 から 0 まで 1 ずつ減らしながら以下を繰り返す：

l を 0 から k まで 1 ずつ増やしながら以下を繰り返す：

a[l] と a[l+1] を比較して，a[l] が a[l+1] より大きければ交換する

2 重の繰り返しが少しわかりにくく感じる場合は，次のように考えればよい．

(1) a[0] と a[1] を比較して必要なら交換，
a[1] と a[2] を比較して必要なら交換，
⋮
a[n-2] と a[n-1] を比較して必要なら交換，を行う (この結果，最大値が a[n-1] に来る)．

(2) a[0] と a[1] を比較して必要なら交換，
a[1] と a[2] を比較して必要なら交換，
⋮
a[n-3] と a[n-2] を比較して必要なら交換，を行う (この結果，2 番めに大きい値が a[n-2] に来る)．

(3) a[0] と a[1] を比較して必要なら交換，
a[1] と a[2] を比較して必要なら交換，
⋮
a[n-4] と a[n-3] を比較して必要なら交換，を行う (この結果，3 番めに大きい値が a[n-3] に来る)．

⋮

($n-2$) a[0] と a[1] を比較して必要なら交換，
a[1] と a[2] を比較して必要なら交換，　を行う (この結果，$n-2$ 番めに大きい値が a[2] に来る)．

($n-1$) a[0] と a[1] を比較して必要なら交換，　を行う (この結果，$n-1$ 番めに大きい値が a[1] に来る)．

作成すべきプログラムは，標準入力から問題 2 と同様の入力を受け取り，規準を表す文字に基づいて整列を行い，比較回数と整列した結果を標準出力に出力するプログラムである．ここで，「比較回数」とは要素同士の比較を行った回数のことで，関数 compare_by を呼び出した回数である．

入力例

```
X
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
10  ←これが比較回数
-4 2
-3 2
0 -5
3 -4
3 1
```

入力例

```
Y
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
10
0 -5
3 -4
3 1
-4 2
-3 2
```

入力例

```
D
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
10
3 1
-3 2
-4 2
0 -5
3 -4
```

入出力の例を見ればわかるように，選択ソートと同じくバブルソートにおける比較回数は入力の長さによって決まる．具体的には，繰り返しの回数を数えればよいので，総比較回数は

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\frac{n(n-1)}{2}$$

となる．

1.3 整列ソートの安定性

同じ順位の要素が複数ある場合，整列の結果はどのようにあるべきだろう．たとえば，あるレポートを提出順に採点したところ，

高橋 93 点，　鈴木 74 点，　田中 85 点，　佐藤 85 点

だったとしよう．これを降順に並べ替えることを考える．問題 3 の選択ソート (「最小値を見つけて最後と交換」の繰り返し) を用いて降順に並べると，

高橋 93 点，　佐藤 85 点，　田中 85 点，　鈴木 74 点

となるが，問題 5 の挿入ソート (「前を整列してから適切な位置を挿入」を繰り返し) を用いて降順に並べると，

高橋 93 点，　田中 85 点，　佐藤 85 点，　鈴木 74 点

となる．どちらが正しい整列結果と言えるだろうか．結論から言えばどちらも正しい整列結果である．ただ，この例に関していえば，もともと提出順に並んでいたわけで，田中くんと佐藤くんが同じ得点だったとしても，先に提出した田中くんの方が評価されるべきかもしれない．このような場合，同じ点数であれば元の順序を保っていることが望ましい．

同じ順位となるものについて元の順序を必ず保つことができる整列アルゴリズムを，**安定ソート**という．選択ソートは，上で見た通り，交換するときに同じ順位の要素の順序が変わることがあるため，安定ソートではない．挿入ソートは，挿入するときに同じ順位であれば後ろに挿入するようにすればよいので，安定ソートになる．バブルソートは，隣同士を比較したときに同じ順位であれば交換しなければよいので，安定ソートになる．

ただし，安定ソートではないアルゴリズムであっても安定ソートとして利用する方法がある．入力となる配列の各要素に元の添字の情報を追加し，同じ順位であれば添字も見て大小を比較するという方法である．たとえば，上の例なら，

高橋 93 点 (1)，　鈴木 74 点 (2)，　田中 85 点 (3)，　佐藤 85 点 (4)

のように提出した順序 (括弧内の数字) も追加しておいて，として，大小を比較する際に「まず得点を比較し，得点が同じときは括弧内を見て大小を決定する」とすれば，安定ソートではないアルゴリズムを使ったとしても，整列結果は

高橋 93 点 (1)，　田中 85 点 (3)，　佐藤 85 点 (4)，　鈴木 74 点 (2)

となり，先に提出した田中くんの方が同じ得点の佐藤くんより必ず先になる．ただし，元の配列の整列結果としては括弧内の数字は余計なので，それを取り除いた

高橋 93 点，　田中 85 点，　佐藤 85 点，　鈴木 74 点

が最終的な整列結果となる．

問題 7

安定かどうかを区別させるために構造体 `point` に対する大小比較の規準として以下のものを考える:

規準 **x** x 座標が大きい方を「大きい座標」とする.

規準 **y** y 座標が大きい方を「大きい座標」とする.

規準 **d** 原点からの距離が遠い方を「大きい座標」とする.

先述の規準と区別するために規準を表す文字が小文字になっているが, 大きな違いは等しい場合の比較方法がない点である. つまり, 等しい場合は順位が等しいものとみなし, (安定性を気にせずに) 整列する場合はどちらを先にしてもよい, ということである.

この問題では, 問題 3 と同様に選択ソートのアルゴリズムの実装を行うが, 安定性を持つように変更した関数 `stable_selection_sort` を定義することが目標である. この関数による整列では, 指定された規準によって昇順に並べる整列を行い, 同じ順位のものは元の順位が保たれる. 関数 `stable_selection_sort` の引数と返り値の型は以下の通りである:

```
void stable_selection_sort(struct point a[], int n, char c);
```

この関数は,「`struct point` 型の座標を要素にもつ配列 `a` に対し, 先頭から `n` 個 (つまり `0` 番めから `n-1` 番めまで) のうち, 文字 `c` で指定された規準で安定な整列を行う関数」である. `c` は `'x'`, `'y'`, `'d'` のいずれかで, 標準入力から与えられる. この関数の定義の中で使用する, 元の添字情報を含む配列の要素は以下の構造体で表すものとする:

```
struct point_loc { int loc; struct point p; };
```

メンバ `loc` が元の添字を表し, メンバ `p` で大小を比較して等しい場合には `loc` によって比較する. 関数 `stable_selection_sort` では,

1. 構造体 `point_loc` を要素とする配列 `arr` を用意する.
2. 元の配列 `a` の各要素に適切な添字情報をつけたものを `arr` に格納する.
3. `arr` を選択ソートアルゴリズムにより整列する (大小比較で等しい場合は添字情報を使用).
4. 整列結果を元の配列 `a` に戻す.

という処理を行う必要がある. 標準入力から与えられるのは 2 行以上 129 行以下で, 最初の行は規準を表す `'x'`, `'y'`, `'d'` のいずれか 1 文字 (この問題では小文字であることに注意), 2 行め以降の各行には 2 つの `int` 型の整数値を空白で挟んだ文字列が含まれているものとする. 標準出力には整列結果が出力される (この問題では比較回数は不要).

作成すべきプログラムの形式は問わないが, 関数 `stable_selection_sort` を上述の指示に従って適切に実装する必要がある. 入力の扱い方についてはこれまでの問題で示したプログラムを参考にするよい.

入力例

```
x
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
-4 2
-3 2
0 -5
3 1
3 -4
```

入力例

```
y
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
0 -5
3 -4
3 1
-3 2
-4 2
```

入力例

```
d
0 -5
-3 2
3 1
-4 2
3 -4
```

出力例

```
3 1
-3 2
-4 2
0 -5
3 -4
```

まとめ

3 つの単純な整列アルゴリズムを実装するとともに，それらの計算量や安定性について学習した．3 つのアルゴリズムの特徴をまとめると以下のようになる．

ソートの種類	比較回数	時間計算量	安定性
選択ソート	常に $\frac{n(n-1)}{2}$	$O(n^2)$	安定でない
挿入ソート	$n-1$ 以上 $\frac{n(n-1)}{2}$ 以下	$O(n^2)$	安定
バブルソート	常に $\frac{n(n-1)}{2}$	$O(n^2)$	安定

この表だけ見ると，挿入ソートが最も優れているようにも見えるが，コードの簡潔さからいえばバブルソートが最も簡潔であり，理解しやすさからすれば選択ソートが (個人差はあるが) 最も理解しやすい．今回はいずれのアルゴリズムの時間計算量も $O(n^2)$ であったが，次回はより高速なアルゴリズムを実装する．