2018 年度 情報領域演習第三
— 第 8 回 —

注意事項

- 今回も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．

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．

```
[p1710999@blue00 ~]> /ced-home/staff/18jr3/08/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: p1710999, 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 6 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．

```
[p1710999@blue00 ~]> /ced-home/staff/18jr3/08/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．

```
[p1710999@blue00 ~]> /ced-home/staff/18jr3/08/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: p1710999, 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．

```
[p1710999@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 効率のよい整列アルゴリズム

今回扱う 2 つの整列アルゴリズムは，どれも前回実装したアルゴリズムより効率のよいものである．前回に引き続き，構造体 `point` を要素とする配列の整列を行うものとし，大小比較の規準として，規準 **X**，規準 **Y**，規準 **D** を用い，昇順に並べかえる整列を考えるが，規準を表す文字は，`compare_by` やソート関数に毎回引数として渡すのではなく，グローバル変数 `kijun` に保持しておき，この値によって比較方法を変更するような関数 `compare` を用いるものとする．関数 `compare` の引数と返り値の型は以下の通りである．

```
int compare(struct point p1, struct point p2);
```

1.1 クイックソート

まずは，クイックソートと呼ばれるアルゴリズムを実装してみよう．クイックソートは「大きな問題を小さな問題に分けて解く」という分割統治法の 1 つである．ピボット (pivot) と呼ばれる値と比較して小さなものと大きなものに分割し，それをまた小さいもの同士，大きなもの同士で同じことを繰り返すことで整列を完成させる．まず，分割するための関数 `partition` を実装し，続いて整列する関数 `quicksort` を実装する．

問題 1

クイックソートアルゴリズムの根幹をなす関数 `partition` を実装せよ．引数と返り値の型は以下の通りである:

```
int partition(struct point a[], int m, int n);
```

この関数は,「構造体 `point` を要素にもつ配列 `a` の `m` 番めから `n` 番めの要素について，`m` 番めの要素より小さいものを前に，大きなものを後ろに移す関数」である．返り値は `m` 番めの要素の移動先の添字とする．`m` 番めの要素を基準に小さなものと大きなものに分割 (partition) していることから，関数名を `partition` としている．分割するだけなら小さいもの同士や大きなもの同士の順序は問わないが，クイックソートの考案者であるホーア (C. A. R. Hoare) による効率のよい以下の分割方法 [1] によって実装せよ (以下の記述では添字が小さい方を左，大きい方を右として読むこと).

(1) `m` 番め (左端) の値を `pivot` に入れ，2 つの変数 `l`，`r` を用意し，それぞれ `m+1` (`pivot` を除いて左端の添字) と `n` (右端の添字) とする．

(2) 以下を繰り返す (`l` は左から右へ，`r` は右から左へ動いていく)：

  (i) `l` が `n` より左にあり `l` 番めの要素が `pivot` より小さい限り，`l` を右に移し続ける．

  (ii) `r` が `m` より右にあり `r` 番めの要素が `pivot` より大きい限り，`r` を左に移し続ける．

[1] 厳密に言えば，ここで示す方法はホーアによる分割方法とは少し異なる．ホーアの方法では，`l` や `r` が `m` から `n` の間から出ているかをチェックする必要がなかったが，ここでは同じピボットが何度も比較されないように工夫した方法を扱っている．興味のある人は，シラバスで紹介している「アルゴリズムイントロダクション (第 3 版)」に紹介されているホーアによる本来の分割方法を見てみるとよい．

(iii) `l` が `r` より左にあれば `l` 番めと `r` 番めを交換し，そうでなければ繰り返しを終了する．

(iv) `l` を 1 つ右に，`r` を 1 つ左に移す．

(3) `r` 番めの値を `m` 番めに入れ，`pivot` (`m` 番めの値) を `r` 番めに入れる．

(4) `r` を返す．

自然数を要素に持つ配列を例にして，この分割がどのように実現されるか見てみよう．以下のような配列に対して 2 番めから 9 番めまでの要素を対象に分割する場合を考える (`m` を 2，`n` を 9 として関数が呼ばれた場合である)．

	2	3	4	5	6	7	8	9	
···	42	19	67	34	17	12	49	58	···

この場合，(1) の手続きが終了した時点では `pivot` は 42，`l` は 3，`r` は 9 であり，(2) の繰り返しが始まる．1 回めは，(i) により，$19 < 42$，$67 > 42$ から `l` が 4 となり，(ii) により，$58 > 42$，$49 > 42$，$12 > 42$ から `r` が 7 となり，(iii) により，`l` 番めの要素と `r` 番めの要素を交換するため，

	2	3	4	5	6	7	8	9	
···	42	19	12	34	17	67	49	58	···

となり，(iv) により `l` が 5，`r` が 6 となる．2 回めは，(i) により，$34 < 42$，$17 < 42$，$67 > 42$ から `l` が 7 となり，(ii) により，$17 < 42$ から `r` が 6 のままとなるため，(iii) により，繰り返しが終了する．(3) の手続きによって，

	2	3	4	5	6	7	8	9	
···	17	19	12	34	42	67	49	58	···

となり，(4) の手続きにより，6 が返され分割が終了する．この分割により，`pivot` とした 42 より小さいものが左に大きなものが右になったことがわかる．

この問題では，まず関数 `compare` を定義し [2]，それを利用して，関数 `partition` を定義することが目的である．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` と関数 `partition` の定義を追加することでプログラムを完成させよ．標準入力から与えられる入力は 3 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．また標準出力には，1 行めに関数 `partition` の返り値を出力し，2 行め以降に分割後の配列を入力と同じ形式で出力する．

入力例

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

出力例

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

入力例

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

出力例

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

[2] 前回作成した `compare_by` の引数 `c` を取り除き，関数定義内の `c` を `kijun` に変更するだけでよいはずである．

入力例

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

出力例

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

作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えること．

```
#include<stdio.h>

char kijun; /* 大小比較の規準を表すグローバル変数 */

struct point { int x, y; };

int compare(struct point p1, struct point p2) {
  /* ※ここは適切なプログラムで埋める */
}

int partition(struct point a[], int m, int n) {
  /* ※ここも適切なプログラムで埋める */
}

int main() {
  char buf[128];
  struct point p, arr[128];
  int i = 0, n;
  scanf("%c␣",&kijun);  /* 大小比較の規準を表す文字を読み込む */
  while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
    sscanf(buf,"%d␣%d",&p.x,&p.y);
    arr[i] = p;
    ++i;
  }
  n = i;  /* n には読み込んだ要素数が格納される */
  printf("%d\n", partition(arr, 0, n-1));  /* partition の返り値を出力 */
  for(i=0;i<n;++i)
    printf("%d␣%d\n", arr[i].x, arr[i].y); /* partition 後の配列を出力 */
  return 0;
}
```

メモ　ピボットと等しい要素であっても交換が起こることに注意せよ．ピボットと等しい要素を交換の対象としない場合でも正しく分割できるが，等しい要素が多い場合には分割の結果に偏り出てしまう．この後のクイックソートでは分割が偏りがないことが重要になるため，等しい要素でも交換するようなアルゴリズムを用いている．

問題 2

構造体 `point` を要素とする配列 `a` に対してクイックソートを行う関数 `quicksort` を定義しよう．引数と返り値の型は以下の通りである:

```
void quicksort(struct point a[], int m, int n);
```

この関数は「配列 a の m 番めから n 番めまでを，クイックソートアルゴリズムにより，指定された規準で昇順に整列する関数」である．クイックソートによる整列は以下のように関数 partition を繰り返し用いることで実現できる：

- m が n より小さければ以下の手続きを行う．
  1. 関数 partition を用い，配列 a の m 番めから n 番めまでを分割し，その返り値 (ピボットの移動先) を p とする．
  2. 配列 a の m 番めから p-1 番めまでをクイックソートにより整列する．
  3. 配列 a の p+1 番めから n 番めまでをクイックソートにより整列する．
- m が n 以上なら何もしない (配列のうち長さが 1 以下の部分の整列であるため)．

下線部が示すように，クイックソートによる整列の内部でクイックソートによる整列を用いていることに注意せよ．再帰関数の考え方を用いれば簡単に定義できる．

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 quicksort の定義を追加することでプログラムを完成させよ．標準入力から与えられる入力は 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．また標準出力には，1 行めに compare の呼ばれた回数を出力し，2 行め以降に整列後の配列を入力と同じ形式で出力する．

入力例

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

出力例

```
15
-4 2
-3 2
-2 -6
0 -5
3 -4
3 1
4 -3
```

入力例

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

出力例

```
19
-2 -6
0 -5
3 -4
4 -3
3 1
-4 2
-3 2
```

入力例

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

出力例

```
15
3 1
-3 2
-4 2
0 -5
3 -4
4 -3
-2 -6
```

作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えること．

```
1 #include<stdio.h>
2
```

```
3  int count = 0;
4  char kijun;
5
6  struct point { int x, y; };
7
8  int compare(struct point p1, struct point p2) {
9    ++count;
10   /* ※ここは問題 1 と同じ */
11 }
12
13 int partition(struct point a[], int m, int n) {
14   /* ※ここも問題 1 と同じ */
15 }
16
17 void quicksort(struct point a[], int m, int n) {
18   /* ※ここを適切なプログラムで埋める */
19 }
20
21 int main() {
22   char buf[128];
23   struct point p, arr[128];
24   int i = 0, n;
25   scanf("%c␣",&kijun);
26   while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
27     sscanf(buf,"%d␣%d",&p.x,&p.y);
28     arr[i] = p;
29     ++i;
30   }
31   n = i;
32   quicksort(arr, 0, n-1);
33   printf("%d\n", count);
34   for(i=0;i<n;++i)
35     printf("%d␣%d\n", arr[i].x, arr[i].y);
36   return 0;
37 }
```

メモ　クイックソートアルゴリズムは，分割統治のアイデアを利用することで平均的には $O(n \log n)$ という時間計算量で実行される．このことは比較回数を数えることでも確認できる．この点では，前回実装した 3 つのアルゴリズム (選択ソート，挿入ソート，バブルソート) よりも優れていることがわかる．ただ，偏りのある分割が繰り返されるような場合には $O(n^2)$ の実行時間が必要になるという欠点や，分割の際に起こる要素同士の交換により安定性が失われてしまうという欠点がある．

1.2　マージソート

マージソートは，クイックソートと同様に分割統治法を利用した整列アルゴリズムである．クイックソートでは，入力となる配列を大小比較によって分割することを繰り返すことで整列を行なっていたが，マージソートでは，入力となる配列を半分に分割し，それぞれを整列してから，整列されたもの同士を併合 (merge) することで整列を行う．マージソートは，入力に関係なく半分に分割して整列を行うため，クイックソートのように入力によって分割に偏りが生じることがなく，どんな入力に対しても $O(n \log n)$ の効率的な整列が可能となっている．今回は，まず，整列されたもの同士を併合する関数 `merge` を実装し，続いて整列する関数 `merge_sort` を実装する．

問題 3

マージソートアルゴリズムの根幹をなす関数 merge を実装せよ．引数と返り値の型は以下の通りである:

```
void merge(struct point a[], int m, int n, int h);
```

この関数は，「構造体 point を要素にもつ配列 a について，前半 (m 番めから h 番めまで) と後半 (h+1 番めから n 番めまで) がそれぞれ昇順に整列されているときに，それらを組み合わせて全体 (m 番めの要素から n 番めまで) が昇順に整列されている状態にする関数」である．

ここでは，この関数の自然数を要素に持つ配列を例にして説明しよう．以下のような配列の一部分に対して，関数 merge が実行された場合を考える．

	2	3	4	5	6	7	8	
···	17	35	42	52	15	19	48	···

このとき，m を 2，n を 8，h を 5 として関数 merge が呼ばれたとしよう．前半 (2 番めから 5 番めまで) と後半 (6 番めから 8 番めまで) がそれぞれ昇順に整列されていることに注意せよ．関数 merge の実行後は以下のような配列になる．

	2	3	4	5	6	7	8	
···	15	17	19	35	42	48	52	···

これを実現するために様々な方法が考えられるが，同じ配列の上で交換しながら整列するのは難しいため，別の配列を利用するのが一般的である．上の自然数の配列の例なら，

(1) 2 番め (17) と 6 番め (15) を比較して小さい方 (15) を別の配列の 2 番めに置く．

(2) 2 番め (17) と 7 番め (19) を比較して小さい方 (17) を別の配列の 3 番めに置く．

(3) 3 番め (35) と 7 番め (19) を比較して小さい方 (19) を別の配列の 4 番めに置く．

(4) 3 番め (35) と 8 番め (48) を比較して小さい方 (35) を別の配列の 5 番めに置く．

(5) 4 番め (42) と 8 番め (48) を比較して小さい方 (42) を別の配列の 6 番めに置く．

(6) 5 番め (52) と 8 番め (48) を比較して小さい方 (48) を別の配列の 7 番めに置く．

(7) 後半に要素がなくなったので，前半の残りの要素をそのまま別の配列に移す．

もちろん，最後は前半の要素が先になくなる場合もある．また，安定な整列が可能になるように，順位が同じ要素同士は前半の要素を先にする点に注意して実装する必要がある．

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 merge の定義を追加することでプログラムを完成させよ．標準入力から与えられる入力は 3 行以上 129 行以下で，最初の行には規準を表す 'X'，'Y'，'D' のいずれか 1 文字と非負整数値 h を空白で挟んだ文字列が含まれ，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．入力は $h+3$ 行以上存在し，前半 (2 行めから $h+2$ 行めまで) と後半 ($h+3$ 行めから最後の行まで) は指定された規準で整列されているものと仮定してよい．また標準出力には，1 行めに compare の呼ばれた回数を出力し，2 行め以降に前半と後半を併合した後の配列を入力と同じ形式で出力する．

入力例

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

出力例

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

入力例

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

出力例

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

入力例

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

出力例

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

作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えること．

```c
#include<stdio.h>

int count = 0;
char kijun;

struct point { int x, y; };

int compare(struct point p1, struct point p2) {
  ++count;
  /* ※ここは問題1と同じ */
}

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

int main() {
  char buf[128];
  struct point p, arr[128];
  int i = 0, h, n;
  scanf("%c␣%d␣",&kijun, &h); /* 規準をkijunに，前半終了の添字をhに格納する */
  while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
    sscanf(buf,"%d␣%d",&p.x,&p.y);
    arr[i] = p;
    ++i;
  }
  n = i;
  merge(arr, 0, n-1, h);
  printf("%d\n", count);
  for(i=0;i<n;++i)
    printf("%d␣%d\n", arr[i].x, arr[i].y);
  return 0;
}
```

ヒント　併合した結果が正しくても比較回数が合わない原因として，安定な整列が行われていない可能性が考えられる．前半と後半に同じ要素があるときにどのように処理されているか注意しよう．

問題 4

構造体 `point` を要素とする配列 `a` に対してマージソートを行う関数 `merge_sort` を定義しよう [3]．引数と返り値の型は以下の通りである:

```
void merge_sort(struct point a[], int m, int n);
```

この関数は「配列 `a` の `m` 番めから `n` 番めまでを，マージソートアルゴリズムにより，指定された規準で昇順に整列する関数」である．マージソートによる整列は以下のように関数 `merge` を繰り返し用いることで実現できる：

- `m` が `n` より小さければ以下の手続きを行う．
  1. `(m+n)/2` (`m` と `n` の平均の小数点以下を切り捨てた値) を前半終了の添字 `h` とする．
  2. 配列 `a` の `m` 番めから `h` 番めまでをマージソートにより整列する．
  3. 配列 `a` の `h+1` 番めから `n` 番めまでをマージソートにより整列する．
  4. 関数 `merge` により，配列 `a` の `m` 番めから `h` 番めまでと `h+1` 番めから `n` 番めまでを併合する．
- `m` が `n` 以上なら何もしない (配列のうち長さが 1 以下の部分の整列であるため)．

下線部が示すように，マージソートによる整列の内部でマージソートによる整列を用いていることに注意せよ．クイックソートと同じく再帰関数の考え方を用いれば簡単に定義できる．

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 `merge_sort` の定義を追加することでプログラムを完成させよ．標準入力から与えられる入力は 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．また標準出力には，1 行めに `compare` の呼ばれた回数を出力し，2 行め以降に整列後の配列を入力と同じ形式で出力する．

入力例

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

出力例

```
14
-4 2
-3 2
-2 -6
0 -5
3 -4
3 1
4 -3
```

入力例

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

出力例

```
12
-2 -6
0 -5
3 -4
4 -3
3 1
-4 2
-3 2
```

[3] 関数名が `mergesort` でないことに注意せよ．

入力例

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

出力例

```
13
3 1
-3 2
-4 2
0 -5
3 -4
4 -3
-2 -6
```

作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えること．

```
#include<stdio.h>

int count = 0;
char kijun;

struct point { int x, y; };

int compare(struct point p1, struct point p2) {
  ++count;
  /* ※ここは問題1と同じ */
}

void merge(struct point a[], int m, int n, int h) {
  /* ※ここは問題3と同じ */
}

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

int main() {
  char buf[128];
  struct point p, arr[128];
  int i = 0, n;
  scanf("%c␣",&kijun);
  while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
    sscanf(buf,"%d␣%d",&p.x,&p.y);
    arr[i] = p;
    ++i;
  }
  n = i;
  merge_sort(arr, 0, n-1);
  printf("%d\n", count);
  for(i=0;i<n;++i)
    printf("%d␣%d\n", arr[i].x, arr[i].y);
  return 0;
}
```

2 応用問題

2.1 乱択クイックソート

クイックソートの欠点として，分割の際に偏りがある場合には分割統治の恩恵が得られない，というものがあった．たとえば，以下のように，初めから昇順に並んでいる配列に対してクイックソートアルゴリズムを適用する場合を考えよう．

0	1	2	3	4	5	6
12	23	34	45	56	67	78

この場合，最初に 0 番めをピボットして分割すると，前半の要素はなく，残りの全ての要素が後半の要素となる．その後半の分割においても，1 番めをピボットして分割するため，残りの全ての要素が後半の要素となる．これを繰り返した結果，全ての要素について他の要素と大小比較することとなり，比較回数は選択ソートやバブルソートと同じく $\frac{n(n-1)}{2}$ 回になってしまう．

これを解決する方法として，ピボットを先頭の要素に固定せずに，毎回ランダムに選ぶという乱択クイックソートが提案されている．ランダムな選択を利用した乱択アルゴリズムの 1 つであるが，比較回数の期待値を計算すると $O(n \log n)$ であることが知られている [4]．

問題 5

問題 2 のプログラムのうち，関数 `partition` のプログラムを適切に変更して，乱択クイックソートアルゴリズムを実装せよ．C 言語において乱数を取得するには `rand` 関数を用いればよい．`rand` 関数を利用するには，ヘッダファイルとして `stdlib.h` をインクルードする必要がある．`rand` 関数の返り値は任意の `int` 型の数値を取りうるため，`m` から `n` までの整数値の中からランダムな値を取得するには，

```
m + rand()%(n-m+1)
```

などとすればよい (ただし，`m` が `n` 以下の場合に限る)．

作成すべきプログラムは問題 2 に準ずるが，比較回数は問わないため `count` に関する部分は全て削除すること．

入力例

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

出力例

```
-4 2    ← 比較回数は出力しない
-3 2
-2 -6
0 -5
3 -4
3 1
4 -3
```

入力例

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

出力例

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

[4] シラバスでも紹介されている教科書「アルゴリズムイントロダクション (第 3 版)」の第 7 章に鮮やかな証明があるので，意欲があれば読んでみるとよい．

入力例

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

出力例

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

メモ　実行テストでは比較回数は出力させる必要はないが，手元で比較回数を出力させてみて，通常のクイックソートよりどの程度効率がよくなっているか確認してみることをお勧めする．

2.2　連結リストのマージソート

配列ではなく連結リストを整列する場合，マージソートを利用すると効率的に整列を行うことができる．連結リストのマージソートは，分割と併合の両方のステップにおいて配列のマージソートとは異なる．以下の説明では，連結リストの節点として以下の構造体を用いることとし，頭 (ダミー) を先頭にしたリストを用いた実装を想定している：

```
typedef struct point elementtype;
struct node {
  elementtype element;
  struct node *next;
}
typedef struct node* list;
```

まず，連結リストの分割は以下のようにして行う．配列では，長さを基準に前半と後半に分けることによって 2 つの配列に分けているように見せていたが，連結リストでは，長さを取得することなく 2 つの連結リストに分けることができる．具体的には，元のリスト l の先頭を指すポインタ変数 p_1，p_2 を用意し，p_2 が 2 つ next をたどる度に p_1 が 1 つ next をたどり，p_2 が 2 つ next をたどれなくなった時点[5] での p_1 の指す節点の next が後半のリストの頭の next となるようにし，p_1 指す節点の next を NULL とすることで，元のリストの頭が前半のリストの頭となる．

また，併合についても配列とはやや異なる．配列では併合するために別の配列を用意する必要があったが，連結リストでは 2 つのリストを先頭から順に見ていき，小さい方が前になるように順に挿入してリストを完成させればよい．なお，安定な整列を実現するため，互いに等しい要素であれば，前半のリストの要素が前になるように先に挿入する．

問題 6

マージソートアルゴリズムを利用して，連結リストに対する整列を行う以下の関数を定義せよ：

- `void split(list l1, list l2);`
  l1 に与えられた連結リストを上述の要領で分割し，分割した結果を l1 と l2 の指す節点を頭とする連結リストに格納する (引数で与えられる l2 は空リストであるものと仮定する)．l1 の要素数は l2 の要素数より多いか等しくなる．

- `void merge(list l1, list l2);`
  昇順に整列された連結リスト l1，l2 に対し，それらを上述の要領で併合した連結リストを l1 に格納する．

- `void merge_sort(list l);`
  連結リストを指定された規準で昇順に整列する．

[5] p_2 が NULL か，p_2 の指す節点の next が NULL になったとき．

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，先に示した 3 つの関数の定義を追加することでプログラムを完成させよ．標準入力から与えられる入力は 2 行以上 10000 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．また標準出力には，1 行めに `compare` の呼ばれた回数を出力し，2 行め以降に整列後の配列を入力と同じ形式で出力する．

入力例

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

出力例

```
14
-4 2
-3 2
-2 -6
0 -5
3 -4
3 1
4 -3
```

入力例

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

出力例

```
12
-2 -6
0 -5
3 -4
4 -3
3 1
-4 2
-3 2
```

入力例

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

出力例

```
13
3 1
-3 2
-4 2
0 -5
3 -4
4 -3
-2 -6
```

作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えること．

```c
#include<stdio.h>
#include<stdlib.h>

int count = 0;
char kijun;

struct point { int x, y; };
typedef struct point elementtype;

struct node { elementtype element; struct node *next; };
typedef struct node* list;

int compare(struct point p1, struct point p2) {
  ++count;
  /* ※ここは問題 1 と同じ */
}

```

```
18 void split(list l1, list l2) {
19   /* ※ここは適切なプログラムで埋める */
20 }
21
22 void merge(list l1, list l2) {
23   /* ※ここも適切なプログラムで埋める */
24 }
25
26 void merge_sort(list l) {
27   /* ※ここも適切なプログラムで埋める */
28 }
29
30 int main() {
31   char buf[128];
32   struct point p;
33   list l, last;
34   scanf("%c␣",&kijun);
35
36   last = l = (list)malloc(sizeof(struct node));
37   while(fgets(buf,sizeof(buf),stdin)!=NULL) {
38     sscanf(buf,"%d␣%d",&p.x,&p.y);
39     last = last->next = (list)malloc(sizeof(struct node));
40     last->element = p;
41   }
42   last->next = NULL;
43
44   merge_sort(l);
45
46   printf("%d\n", count);
47   while((l=l->next) != NULL) {
48     p = l->element;
49     printf("%d␣%d\n", p.x, p.y);
50   }
51   return 0;
52 }
```