2018 年度 情報領域演習第三
— 第 9 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．

```
[p1610999@blue00 ~]> /ced-home/staff/18jr3/09/checkerN
提出開始: 12 月 xx 日 14 時 40 分 0 秒
提出締切: 12 月 yy 日 14 時 39 分 0 秒
ユーザ: p1610999, 出席状況: 2018-12-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 5 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/18jr3/09/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/18jr3/09/checkerN
提出開始: 12 月 xx 日 14 時 40 分 0 秒
提出締切: 12 月 yy 日 14 時 39 分 0 秒
ユーザ: p1610999, 出席状況: 2018-12-xx-14-47
 問題:     結果 | 提出日時            | ハッシュ値                        |
```

```
  1:    成功 | 2018-12-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 効率のよい整列アルゴリズム (続き)

前回の演習では，効率のよい整列アルゴリズムのうち，クイックソートとマージソートのプログラムを作成したが，どちらも分割統治の考え方により $O(n \log n)$ の実行時間を実現していた．今回の演習で扱うヒープソートは，分割統治とは全く異なる方法で $O(n \log n)$ の実行時間を実現する整列アルゴリズムである．

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．また，前回と同様に，規準を表す文字はグローバル変数 `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);
```

この関数定義は前回のものを再利用するだけでよい．

1.1 ヒープソート

ヒープソートアルゴリズムを理解するためには，まず，配列におけるヒープと呼ばれる概念を理解する必要がある．配列の各要素が以下の条件を満たしているとき，その配列は**ヒープ**であるという．

- $2i+1$ 番めの要素は i 番めの要素以下 (等しいか小さい) である．
- $2i+2$ 番めの要素は i 番めの要素以下 (等しいか小さい) である．

つまり，ヒープにおいては，1 番めや 2 番めは 0 番め以下，3 番め 4 番めは 1 番め以下，5 番めや 6 番めは 2 番め以下，... といったことが必ず成り立っている．たとえば，自然数を要素とする配列を考えるとき，

0	1	2	3	4	5	6	7	8
57	35	42	35	18	40	22	21	29

はヒープであるが，この 4 番めと 5 番めを交換しただけの

0	1	2	3	4	5	6	7	8
57	35	42	35	40	18	22	21	29

はヒープではない (4 番めが 1 番めより大きい)．一般的にヒープは昇順でも降順でもないため，整列するために役に立つかはすぐにわからないかもしれないが，後で見るように，上に示したヒープの条件が重要な役割を果たす．便宜上，i 番めの要素を「$2i+1$ 番め (または $2i+2$ 番め) の要素の**親**」，$2i+1$ 番め (または $2i+2$ 番め) の要素を「i 番めの要素の**子**」と呼ぶ．したがって，ヒープの条件を端的に言い換えると子の値は必ず親の値以下 ということである．なお，ヒープは昇順でも降順でもないものの，

> 0 番めから k 番めまでがヒープの条件を満たすとき，0 番めの要素がその中で最大となる

という性質が成り立つことも覚えておこう．

問題 1

ヒープソートの仕組みを理解しているか確認するために，まず，ヒープであるかどうか判定する関数 is_heap を作成しよう．

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

この関数は「構造体 point を要素にもつ配列 a のうち，先頭 n 個 (0 番めから n-1 番めまで) がヒープになっていれば 1 を返し，ヒープでなければ 0 を返す関数」である．ヒープの条件判定に用いる大小比較は前回定義した compare 関数を用いるものとする．関数 is_heap はヒープアルゴリズムを実装する上で必要な関数ではないが，この問題はヒープの条件を理解しているかを問う問題である．

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 is_heap 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．作成すべきプログラムは，入力の各行を 0 番めから配列に格納した際に，最後の要素までの配列がヒープになっていれば Yes. を，ヒープになっていなければ No. を標準出力に出力する．

入力例

```
X
4 3
-1 3
2 1
-2 -1
-1 2
1 2
```

出力例

```
Yes.
```

入力例

```
Y
4 3
-1 3
2 1
-2 -1
-1 2
1 2
```

出力例

```
No.
```

入力例

```
D
4 3
-1 3
2 1
-2 -1
-1 2
1 2
```

出力例

```
Yes.
```

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  char kijun;
4  struct point { int x, y; };
5
6  int compare(struct point p1, struct point p2) {
7    /* ※ここは前回と同じ */
8  }
9
10 int is_heap(struct point a[], int n) {
```

```
11    /* ※ここを適切なプログラムで埋める */
12  }
13
14  int main() {
15    char buf[128];
16    struct point p, arr[128];
17    int i = 0, n;
18    scanf("%c␣",&kijun);
19    while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
20      sscanf(buf,"%d␣%d",&p.x,&p.y);
21      arr[i] = p;
22      ++i;
23    }
24    n = i;
25    if (is_heap(arr, n)) printf("Yes.\n"); /* 関数 is_heap が 1 を返せば Yes.を出力 */
26    else printf("No.\n");                  /* 0 を返せば No.を出力 */
27    return 0;
28  }
```

ヒント　ヒープの条件を確認する際に入力された要素以外の配列の領域を参照しないように注意しよう．誤って参照してしまうと，セグメンテーション違反 (セグメントエラー) の原因となり，環境によって正しく動作しないプログラムになってしまう．

問題 2

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

```
void pushdown(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 について，m+1 番めから n 番めまでがヒープの条件を満たしているときに，m 番めの要素を適切な位置に挿入して m 番めから n 番めまでがヒープの条件を満たしているようにする関数」である．m 番めを m+1 から後ろに押し込んでいるように見えることから pushdown という関数名となっている．

具体的にどのように押し込むかを，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 1 番めから 7 番めに対して，関数 pushdown が呼ばれた場合を考える (引数 m が 1，n が 7 で関数が呼ばれた場合である).

	1	2	3	4	5	6	7	
···	20	42	35	18	40	22	29	···

このとき，2 番めから 7 番めはヒープの条件を満たしていることに注意せよ．1 番めも含めてもなおヒープの条件を満たしていることを確認するためには，1 番めと (その子である) 3 番め，1 番めと (もう 1 つの子である) 4 番めの関係を調べる必要があるが，3 番めと 4 番めのうち大きな方と比較すればよい．大きな方が 1 番め以下であればヒープの条件を満たしているので目的は達成されたことになる．そうでなければ大きな方と 1 番めを交換すれば [1]，1 番めと 3 番め，1 番めと 4 番めの関係は共にヒープの条件を満たすことになる：

	1	2	3	4	5	6	7	
···	35	42	20	18	40	22	29	···

1 番めと大きな方である 3 番めが交換されたが，この結果，3 番めと (その子である) 7 番めの関係が崩れる可能性が出る (この例では，2 番めから 7 番めをヒープにすること考えているため，8 番めは対象外なので見なくてよい)．実際，3 番めと 7 番めの関係はヒープの条件を満たしていないので交換する：

	1	2	3	4	5	6	7	
···	35	42	29	18	40	22	20	···

[1] 小さな方と 1 番めを交換しても，ヒープの条件は満たされないままである．

ヒープを満たしていない場合はどちらかと交換してヒープの条件を満たすように変更する．これで 2 番めから 7 番めがヒープの条件を満たすようになった．

一般的には以下のようなアルゴリズムで pushdown が実現できる．

- 2*m+2 が n 以下であれば，2*m+1 番めの要素と 2*m+2 番めの要素を比較して大きな方の添字 (互いに等しければ 2*m+2 の方[2]) を i とする．i 番めの要素と m 番めの要素を比較し，i 番めの要素の方が大きければこれらを交換した上で，i番めから n番めに対して関数 pushdownを呼ぶ．大きくなければそのまま終了する．
- 2*m+1 が n であれば，2*m+1 番めの要素と m 番めの要素を比較して，2*m+1 番めの要素の方が大きければこれらを交換して終了する．大きくなければそのまま終了する．
- 2*m+1 が n より大きければ何もせずに終了する (既にヒープの条件が満たされている)．

下線部が示すように，関数 pushdown による操作の内部で関数 pushdown を用いていることに注意せよ．これは再帰関数の考え方を用いれば定義できる．

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 pushdown 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．2 行めから順に配列の 0 番めから格納したとき，1 番めの要素から最後の要素まではヒープの条件を満たしていると仮定してよい．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 行め以降に配列の 0 番めから最後の要素までに対して関数 pushdown を呼んだ後の配列 (ヒープ) を入力と同じ形式で出力する．

入力例

```
X
1 1
1 -2
-1 2
0 -2
0 -2
-2 1
```

出力例

```
2
1 1
1 -2
-1 2
0 -2
0 -2
-2 1
```

入力例

```
Y
1 1
1 -2
-1 2
0 -2
0 -2
-2 1
```

出力例

```
3
-1 2
1 -2
1 1
0 -2
0 -2
-2 1
```

入力例

```
D
1 1
1 -2
-1 2
0 -2
0 -2
-2 1
```

出力例

```
4
1 -2
0 -2
-1 2
0 -2
1 1
-2 1
```

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．

[2]等しいときは 2*m+1 の方を選んでヒープの条件を満たすように変更できるが，2*m+2 を選んだ方がヒープの条件を満たすまでの交換回数が少なくなる確率が高い．

```c
#include<stdio.h>

int count = 0;
char kijun;
struct point { int x, y; };

int compare(struct point p1, struct point p2) {
  ++count;
  /* ※ここは前回と同じ */
}

void pushdown(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;
  pushdown(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;
}
```

ヒント 関数 pushdown を定義する際に関数 is_heap を使うことはないが，うまく定義できないときに，プログラムのところどころで関数 is_heap で動作を確認するとよい．

問題 3

構造体 point を要素とする配列 a に対してヒープソートを行う関数 heap_sort を定義しよう．引数と返り値の型は以下の通りである:

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

この関数は「配列 a の先頭 n 個の要素 (0 番めから n-1 番めまで) を，ヒープ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．ヒープソートによる整列は，以下のように,「ヒープの作成」と「ヒープからの取り出し」の 2 つの段階に分けられるが，どちらの段階においても関数 pushdown が重要な役割を果たしている．

(1) ヒープの作成
k を n/2-1 から 0 まで 1 つずつ減らしながら以下を繰り返す．

関数 pushdown を使って，k 番めの要素を k+1 番めから n-1 番めのヒープに押し込む
(この結果，k 番めから n-1 番めまでがヒープの条件を満たす).

(2) ヒープからの取り出し
k を n-1 から 1 まで 1 つずつ減らしながら以下を繰り返す．

(i) 0 番めと k 番めを交換する
(この結果，0 番めから k 番めまでのうちの最大値が k 番めに入る)

(ii) 関数 pushdown を使って，0 番めの要素を 1 番めから k-1 番めのヒープに押し込む
(この結果，0 番めから k-1 番めまでがヒープの条件を満たす).

ヒープの作成において，n/2-1 から開始しているのは，n/2 番めから n-1 番めまでの要素には子がおらず，n/2 番めから n-1 番めまでだけ見れば必ずヒープの条件が満たされるためである．関数 pushdown を繰り返し使って，ヒープを 1 つずつ左に伸ばしていくことにより，最終的に 0 番めから n-1 番めまでがヒープの条件を満たすことになる．

ヒープからの取り出しでは，ヒープでは 0 番めが必ず最大になるという性質を利用して後ろから整列を完成させている．最初は 0 番めから n-1 番めまでがヒープの条件を満たすので，0 番めと n-1 番めを交換することにより，n-1 番めに最大値が入る．この交換により，0 番めから n-2 番めまでのうち，0 番めだけがヒープの条件を満たさなくなったので，関数 pushdown を使ってヒープの条件を満たすようにする．この結果，0 番めから n-2 番めまでの最大値が 0 番めに入るので，それを n-2 番めと交換する，... ということが繰り返され，最終的に，0 番めから n-1 番めまでが昇順に整列された状態になる．

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 heap_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

```
20
-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

```
21
-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

```
20
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  #include<stdlib.h>
3
4  int count = 0;
5  char kijun;
6  struct point { int x, y; };
7
8  int compare(struct point p1, struct point p2) {
9    ++count;
10   /* ※ここは前回と同じ */
11 }
12
13 void pushdown(struct point a[], int m, int n) {
14   /* ※ここは問題2と同じ */
15 }
16
17 void heap_sort(struct point a[], 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   heap_sort(arr, n);
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　ヒープソートにおける比較回数は，[1 回の関数 pushdown の計算に必要な比較回数]×[関数 pushdown が呼ばれる回数] によって計算することができる．長さ n の配列であれば，[1 回の関数 pushdown の計算に必要な比較回数] が $2\log_2 n$ 以下になることは次のようにしてわかる．pushdown は 2 回比較するごとに子の要素をたどるが，子の要素をたどる度に添字が 2 倍 (正確には 2 倍プラス 1 以上) となるため，$2k$ 回比較されれば添字は少なくとも $2^k - 1$ 以上になる．添字は n より小さいので $2^k - 1 < n$，すなわち $k \leq \log_2 n$ であり，比較回数は $2\log_2 n$ 以下になる．また，[関数 pushdown が呼ばれる回数] は，ヒープの作成に $\frac{n}{2}$ 回以下，ヒープからの取り出しに n 回かかるため，合計 $\frac{3}{2}n$ 回以下となる．よって，ヒープソートにおける比較回数は $\frac{3}{2}n \times 2\log_2 n = 3n\log n$ 以下となる．これがヒープソートの時間計算量が $O(n\log n)$ となる理由である．

2　比較に頼らない整列アルゴリズム

大小比較に基づく整列アルゴリズムでは少なくとも $O(n\log n)$ 回の比較が必要になることが知られている．この理由を理解するために，1 から 5 までの数字が 1 つずつ入った長さ 5 の配列を整列する場合を考え

よう．要素を比較して整列するアルゴリズムでは，どんな整列アルゴリズムであっても最大 7 回の比較が必要であることが以下のように説明できる．

仮に，どんな入力であっても 6 回以下の比較で整列を終えられる「スーパー整列アルゴリズム」があったとしよう．このアルゴリズムでは，最大 6 回しか比較しないので，比較のための `if` 文による分岐を 6 回しか通らない．つまり，真真真真真真のようにすべて真の分岐を通ったか，真真真真真偽のように最後だけ偽の分岐を通ったか，真真真真偽真のように分岐を通ったか，真真真真偽偽のように分岐を通ったか，... のいずれかになるため，$2^6 = 64$ 通りの可能性しかない．ところが，入力となる長さ 5 の配列として考えられるのは $5 \times 4 \times 3 \times 2 \times 1 = 120$ 通りなので，異なる配列なのに同じ分岐を通って整列が終了するということが起こってしまう．7 回比較すれば $2^6 = 128$ 通りの可能性があるので，整列は可能かもしれない [3]．

配列の要素が n 個の場合であれば，$n!$ 通りの入力の可能性があるので，$2^m \geq n!$ となるような m だけ比較を行う必要がある．つまり，少なくとも $\log_2 n!$ 以上の回数は比較が必要であり，この数はおよそ $n \log_2 n - 1.44n$ であることが知られている [4]．したがって，$O(n \log n)$ 回の比較が必要であることが言えて，時間計算量も最低でも $O(n \log n)$ となる．これが「比較に頼る整列アルゴリズムの理論的限界」である．

今回実装するアルゴリズムは比較に頼らないことで，この $O(n \log n)$ の壁を越えることができるようなアルゴリズムである．なお，安定な整列アルゴリズムになっていることを重視するために，以下のような前々回導入した小文字の**規準 x** を用いるものとする：

規準 x x 座標が大きい方を「大きい座標」とする．

今回の問題の目的は，この規準に基づいて昇順に整列するプログラムを作成することである．ただし，以下のアルゴリズムは，比較に頼らないアルゴリズムであるため，関数 `compare`を用意する必要はないことに注意せよ．

2.1 バケツソート

要素の種類が限られている場合，比較を使わずに整列することが可能である．たとえば，0 から 3 まで数値が入った以下のような配列を整列することを考えよう．

0	1	2	3	4	5	6	7	8
3	2	0	3	0	3	3	0	2

これに対し，要素は 0 から 3 までのいずれかであることが事前に分かっているのであれば，0 を入れるバケツ，1 を入れるバケツ，2 を入れるバケツ，3 を入れるバケツを用意する．先頭から順に 3 なら 3 のバケツへ，2 なら 2 のバケツへ，といった具合に最後の要素までバケツに入れれば，0 のバケツから順に要素を取り出すことで昇順に要素が並ぶことになる．

0			1	2		3			
0	0	0		2	2	3	3	3	3

つまり，要素の種類が限られていればその分だけバケツを用意すればよく，どんなに長い配列でも一度も比較せずに整列が終了する．

ただ，それぞれのバケツにいくつ要素が入るかは予測できないので，C 言語でこれを実現するには工夫が必要である．また，整列の安定性 (等しい順位であれば元々前にあった方が前になること) も考慮したい．そこで，2 次元座標の配列に対して **規準 x** によって昇順に整列する場合，どのようにすればよいかを具体例を使ってみてみよう．たとえば，入力として与えられた配列 `a` が以下のような内容であったとする：

0	1	2	3	4	5	6	7	8
$(3,1)$	$(2,0)$	$(0,2)$	$(3,2)$	$(0,3)$	$(3,0)$	$(3,1)$	$(0,1)$	$(2,2)$

x 座標が 0 から 3 までの整数であるため，以下のようにバケツソートが実現できる．

(1) バケツ内の個数をカウントするための配列 `c` を用意し，各要素を 0 としておく．
(いまの場合は `a` の要素の x 座標が 0 から 3 までのいずれかであるので長さ 4 の配列を用意すればよい)

0	1	2	3
0	0	0	0

[3] 実際可能であるがこのアルゴリズムを書き下すのはとても大変である．
[4] スターリングの公式による．

(2) 配列 a の先頭の要素から順に読み，x 座標が 3 であれば c の 3 番めに 1 を足し，2 であれば c の 2 番めに 1 を足し，... といった具合で最後の要素まで行う．
(この結果，各バケツに入る要素の数がわかる)

0	1	2	3
3	0	2	4

(3) 配列 c の 1 番めの要素から順に読み，「i 番めと $i-1$ 番めの和を i 番めに上書きする」を繰り返す．
(この結果，元の c の 0 番めから i 番めの合計が c の i 番めに入る)

0	1	2	3
3	3	5	9

(4) a と同じ大きさで同じ内容を含む配列 b を用意する． これは (2) と同時に行ってもよい．

0	1	2	3	4	5	6	7	8
$(3,1)$	$(2,0)$	$(0,2)$	$(3,2)$	$(0,3)$	$(3,0)$	$(3,1)$	$(0,1)$	$(2,2)$

(5) 配列 b の後ろの要素から順に以下を繰り返す．

(i) その要素 p が (i,j) であれば，c の i 番めを 1 減らす．

(ii) 要素 p を a の c[i] 番めに入れる．

(これにより，b の要素 p に応じて配列 c と配列 a は次のように更新されていく)

p	c				a								
	0	1	2	3	0	1	2	3	4	5	6	7	8
$(2,2)$	3	3	4	9					$(2,2)$				
$(0,1)$	2	3	4	9			$(0,1)$		$(2,2)$				
$(3,1)$	2	3	4	8			$(0,1)$		$(2,2)$				$(3,1)$
$(3,0)$	2	3	4	7			$(0,1)$		$(2,2)$			$(3,0)$	$(3,1)$
$(0,3)$	1	3	4	7		$(0,3)$	$(0,1)$		$(2,2)$			$(3,0)$	$(3,1)$
$(3,2)$	1	3	4	6		$(0,3)$	$(0,1)$		$(2,2)$		$(3,2)$	$(3,0)$	$(3,1)$
$(0,2)$	0	3	4	6	$(0,2)$	$(0,3)$	$(0,1)$		$(2,2)$		$(3,2)$	$(3,0)$	$(3,1)$
$(2,0)$	0	3	3	6	$(0,2)$	$(0,3)$	$(0,1)$	$(2,0)$	$(2,2)$		$(3,2)$	$(3,0)$	$(3,1)$
$(3,1)$	0	3	3	5	$(0,2)$	$(0,3)$	$(0,1)$	$(2,0)$	$(2,2)$	$(3,1)$	$(3,2)$	$(3,0)$	$(3,1)$

このようにして整列が完成する．同じ順位 (x 座標が同じ) である場合は元の順序が保たれるという安定な整列が実現されていることもわかるであろう．

問題 4

構造体 point を要素とする配列 a に対してバケツソートを行う関数 bucket_sort を定義しよう．引数と返り値の型は以下の通りである:

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

この関数は「配列 a の先頭 n 番めの要素について，安定なバケツソートアルゴリズムにより，**規準 x** で昇順に整列する関数」である．各要素の x 座標は 0 から 99 までの整数値であるとする．

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 bucket_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 1 行以上 128 行以下で，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，整列後の配列を入力と同じ形式で出力する．

入力例

```
1 2
0 3
3 0
2 1
```

出力例

```
0 3
1 2
2 1
3 0
```

入力例

```
3 1
2 0
0 2
3 2
0 3
3 0
3 1
0 1
2 2
```

出力例

```
0 2
0 3
0 1
2 0
2 2
3 1
3 2
3 0
3 1
```

入力例

```
0 2456
0 -999
0 1231
```

出力例

```
0 2456
0 -999
0 1231
```

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>

struct point { int x, y; };

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

2.2 基数ソート

比較に頼ったソートの時間計算量には $O(n \log n)$ という限界があったが，バケツソートは入力の配列の長さに比例する $O(n)$ の実行時間で整列を終えることができる．ただし，実際の実行時間は，出現回数を数える配列の長さにも依存することに注意しよう．たとえば，上の構造体 `point` を要素とする配列を整列する例において，元の配列 `a` やそれと同じ長さをもつ配列 `b` だけでなく，出現回数を数える配列 `c` に対しても各要素に対する操作が必要になる．したがって，x 座標が 0 から 999,999 までの整数値であれば，どんなに短い配列の整列でも長さ 1,000,000 の配列が必要になる．

この問題を解決するのが基数ソートと呼ばれるアルゴリズムである．桁ごとにバケツソートを行うソートで，k 桁なら k 回バケツソートを行うため，時間計算量は $O(kn)$ となる．たとえば，基数ソートアルゴリズムによって昇順に整列する具体例として，以下の 3 桁の自然数を要素とする配列を入力とした場合を考えよう．

0	1	2	3	4	5	6	7
524	136	314	531	620	331	85	540

これに対し，まず 1 の位をキー (大小の規準) としてバケツソートを行う．

0	1	2	3	4	5	6	7
620	540	531	331	524	314	85	136

1 の位は 0 から 9 までなので，バケツとなる配列の長さは 10 だけあればよい．バケツソートは安定ソートであるため，524 と 314 のように 1 の位が同じもの同士では元の順序が保たれることに注意しよう．この時点では 1 の位のみが整列されている．次に，10 の位をキーとしてバケツソートを行うと，

0	1	2	3	4	5	6	7
314	620	524	531	331	136	540	85

となる．安定ソートであるため，10 の位が同じであるもの同士は以前の順位が保たれ 1 の位が昇順に並ぶ．つまり，この時点では下 2 桁だけみれば整列されている状態である．最後に，100 の位をキーとしてバケツソートを行う (85 の 100 の位は 0 とする) と，

0	1	2	3	4	5	6	7
85	136	314	331	524	531	540	620

となり，3 桁とも整列された状態になる．4 桁であれば 4 回，5 桁であれば 5 回のバケツソートが必要になる．

上の説明では，整数値を 10 進法で見て「1 の位でバケツソート，10 の位でバケツソート，100 の位でバケツソート，...」と進めていたが，計算機上で基数ソートを行う場合には通常 16 進法や 32 進法などの 2 の冪乗を用いる．これは，一般の計算機では，1 の位，10 の位，100 の位を計算するために 10 で割り算するよりも，2 のベキ乗で割り算する方が遥かに高速であるためである．たとえば，2345 は 2 進表現で 100100101001 であり，10 進表現で下 1 桁に当たる 5 (2 進表現で 101) を計算するためには全ての桁を見て計算する必要があるが，16 ($= 2^4$) 進表現で下 1 桁は 2 進表現での下 4 桁 (1001) をそのまま取り出せばよい．

C 言語において，与えられた整数値 n に対して 2^r 進法で下から d 桁め (最下桁を 0 桁めとする) を取り出すには，

```
(n >> (d * r)) & ((1 << r) - 1)
```

とする．まず，`(1 << r)` は 1 を r ビット上位にシフトして下位を 0 で埋めた数で，2 進表現では $\texttt{1}\overbrace{\texttt{00...0}}^{r}$ となり，`&` の後ろの `((1 << r) - 1)` は $\overbrace{\texttt{11...1}}^{r}$ を表すことがわかる．`&` はビット積演算子で両方のビットが 1 である桁のみを 1 にする演算子なので，x `&` `((1 << r) - 1)` とすると x のうち下位 r ビットを知ることができる．`(n >> (d * r))` は，n を dr ビット下位にシフトした数なので，上の式は，r ビットをまとめて 1 桁と見たときの下から d 桁めを表すことになる．たとえば，2345 を 2^3 進法で表したとき (つまり，3 ビットを 1 桁と見たとき) の下から 2 桁めは，

$$\begin{array}{ccccccccc} & & & & 3 & & 2 & & 1 & & 0 \\ 2345 & = & & & 100 & & 100 & & 101 & & 001 \end{array}$$

より 100 であるが，これは `(2345 >> (2 * 3)) & ((1 << 3) - 1)` により計算できる．

問題 5

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

```
void radix_sort(struct point a[], int n, int r, int dmax);
```

この関数は「配列 a の先頭 n 番めの要素について，r ビットを 1 桁として下位 dmax 桁に対して行う基数ソートアルゴリズムにより，**規準 x** で昇順に整列する関数」である．各要素の x 座標は 0 から $2^{r \times dmax} - 1$ までの整数値であるとする．

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 radix_sort の定義を追加することでプログラムを完成させよ．基数ソートは桁ごとにバケツソートを繰り返すアルゴリズムであるが，問題 4 の関数 bucket_sort はそのまま使えないことに注意せよ．標準入力から与えられる入力は 2 行以上 129 行以下で，最初の行の 2 つの整数値 r，m は，r ビットを 1 桁と見て基数ソートを行い，m 桁までの整数値を扱うことを表し，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．各行の最初の整数 (x 座標) は 0 以上 2^{rm} 未満であると仮定してよい．また標準出力には，各桁のバケツソートが終わるたびにその時点での配列の内容を入力と同じ形式で出力する．ただし，各バケツソートの出力の最後には -- と改行も出力せよ (入出力例を参考にすること)．この出力処理は radix_sort の関数定義の中で記述する必要がある．

入力例

```
1 2 ← 1 ビットを 1 桁として 2 桁の数値と見て整列
1 2
0 3
3 0
2 1
```

出力例

```
0 3
2 1
1 2
3 0
--
0 3
1 2
2 1
3 0
--
```

入力例

```
3 4 ← 3 ビットを 1 桁として 4 桁の数値と見て整列
2345 2
3000 0
3041 1
2345 -1
```

出力例

```
3000 0
2345 2
3041 1
2345 -1
--
3041 1
2345 2
2345 -1
3000 0
--
2345 2
2345 -1
3000 0
3041 1
--
2345 2
2345 -1
3000 0
3041 1
--
```

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>

struct point { int x, y; };

void radix_sort(struct point a[], int n, int r, int dmax) {
  /* ※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, r, dmax;
  scanf("%d␣%d␣",&r, &dmax);

  while(fgets(buf,sizeof(buf),stdin)!=NULL && i<128) {
    sscanf(buf,"%d␣%d",&p.x,&p.y);
    arr[i] = p;
    ++i;
  }
  n = i;
  radix_sort(arr, n, r, dmax);
  return 0;
}
```