プログラミング演習 第 11 回

問 A [11A]: 選択ソート

複数の行を読み込んで、それを選択ソートで辞書順に整列し、辞書で前に現われるものほど先に出力するプログラムを作れ。入力される行は 1 行 80 文字以下で 10000 行以下とする。

図 1 にプログラムの main 関数および必要な型定義を示す。図 1 のプログラム断片は問 B でも利用する。このプログラム断片を用い、「/* ここに sort 関数を定義 */」の部分に選択ソートの関数を定義することでプログラムを完成させること。さらに、選択ソートを行う sort 関数の一部は図 2 に示してある。sort 関数は、図 2 の「/* ここに追加 */」の部分にプログラム断片を追加することで完成させること。

ヒントと注意

- 二つの文字列が辞書順でどちらが先に現れるかを調べるために strcmp 関数を使ってよい。
- プログラミング通論では、配列の 1 番目から n 番目までの要素を整列したが、この課題では 0 番目の要素から n-1 番目の要素を整列することに注意せよ。
- これまで入力の終わりには特別な目印があったが、この問題のプログラムは入力の最後まで読みとる。標準入力が端末の場合、プログラムへの入力を終えるには Ctrl-D(コントロールキーを押しながら D を押す) を入力すればよい。

入出力例

入力

```
abcde
                <-- 空行が与えられている
abc
.
aabb
12345
```

出力

```
                <-- 空行が印字されている
.
12345
aabb
abc
abcde
```

問 B [11B]: クイックソート

問 A と同じことをクイックソートを使って実現するプログラムを作れ。問 A と同様に、図 1 を雛形にせよ。

クイックソートを行う sort 関数と sort 関数で使われる関数の一部は図 3 に示してある。図 3 で未完成の partition 関数における、「/* ここに追加 */」の部分にプログラム断片を追加することでプログラムを完成させよ。

プログラミング通論の資料では swap 関数を使わずに書いているが、この問題では要素の交換に swap 関数を使うこと。

ヒントと注意

プログラミング通論では配列の最後の要素の次に番兵を置くことになっているが、この問題では番兵を使わないプログラムを書くこと。番兵を用いず他の対策もしなければ、partition 関数で a の l 番目の要素が a の $l+1$ 番目以降のどの要素より大きかった時にセグメントエラーになる可能性がある。例えば、入力の先頭に最大の要素が与えられた場合に、このような状況になる。どこで失敗するかと、どのような対策をすればよいか考えよ。

```
#include <stdio.h>
#include <string.h>

#define LIMIT 10000   /* 最大行数 */
#define MAX_LEN 80    /* 行の長さの上限 */

typedef struct {
  char* line;
} recordtype;

recordtype text[LIMIT];

void swap(recordtype *x, recordtype *y) {
  recordtype temp;
  temp = *x; *x = *y; *y = temp;
}

/* ここに sort 関数を定義 */

int main() {
  int i, len;
  int nlines;  /* 読み込んだ行数 */
  char buf[MAX_LEN + 2];

  /* 読み込み */
  for (nlines = 0; nlines < LIMIT; nlines++) {
    if (fgets(buf, sizeof(buf), stdin) == NULL)
      break;                                        /* 入力の終わりで抜ける */
    len = strlen(buf);
    if (buf[len-1] == '\n') buf[len-1] = '\0';  /* 行末の改行文字を消す */
    text[nlines].line = strdup(buf);            /* なぜ strdup が必要か? */
  }

  /* ソート */
  sort(text, nlines);

  /* 表示 */
  for (i = 0; i < nlines; i++)
    printf("%s\n", text[i].line);

  return 0;
}
```

図 1: 問 A と問 B の main 関数

問 C [11C]: アカウント表 その 3

7 桁の整数の学籍番号と 8 桁の文字列のアカウントを空白で区切って並べた組が 1 行に 1 つずつ与えられる。この入力を読み込んで、学籍番号の昇順に出力するプログラムを作れ。入力は 10000 件以下とする。

ただし、この問題では最初に配列に読み込んでから整列してはいけない。挿入ソートの要領で 1 行読み込むたびに配列のしかるべき位置に挿入することで整列された配列を作れ。

図 4 のプログラムで insert 関数を定義してプログラムを完成させること。insert 関数は次のプロトタイプ宣言を持つ。

```
void insert(struct student* a, int n, struct student v);
```

引数は順に、挿入する先の配列、配列が保持している要素数、挿入したい要素である。

insert 関数は、0 番目から $n-1$ 番目までに要素が整列されて入っている配列を受けとって、0 番目から n 番目までに要素が整列されて入っている状態にする。

入出力例

入力

```
2013005 xx011005
2013001 aa011001
2013002 bb011002
2013004 nn011004
2013003 mm011003
```

出力

```
2013001 aa011001
2013002 bb011002
2013003 mm011003
2013004 nn011004
2013005 xx011005
```

```
/* 配列 a の先頭から n 個の要素を選択ソートで整列する */
void sort(recordtype a[], int n) {
  int i, j, minindex;

  for (i = 0; i < n - 1; i++) {
    /* ここに追加 */
    /* 次の printf 文は動作確認用であるので提出する前に消すこと */
    printf("a[%d](%s)  <-> a[%d](%s)\n",
           i, a[i].line, minindex, a[minindex].line);
    swap(&a[minindex], &a[i]);
  }
}
```

図 2: 選択ソート関数

```
/* ある k (l < k <= r) に対して
 *   a[x] <= a[k] (l <= x < k)
 *   a[x] >= a[k] (k < x <= r)
 * となるように a の l から r 番目を並びかえて、 k を返す */
int partition(recordtype a[], int l, int r) {
  int i, k;
  recordtype v;

  v = a[l]; i = l; k = r + 1;
  /* ここに追加 */
  swap(&a[l], &a[k]);

  return k;
}

/* 配列 a の l から r 番目をソートする */
void quicksort(recordtype a[], int l, int r) {
  int i;
  if (l < r) {
    i = partition(a, l, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
}

/* 配列 a の先頭 n 個の要素 (a[n-1] まで) をソートする */
void sort(recordtype a[], int n) {
  quicksort(a, 0, n - 1);
}
```

図 3: クイックソート関数

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

#define LIMIT 10000

struct student {
  int id;
  char account[9];
};

void set(struct student* s, int id, char *account) {
  s->id = id;
  strcpy(s->account, account);
}

int main(void) {
#define BUFSIZE 80
  struct student *students, v;
  char buf[BUFSIZE], *account;
  int nstudents;   /* 読み込んだ学生の数 */
  int i, len, id;

  /* LIMIT 件分の配列を確保 */
  students = (struct student*) malloc(sizeof(struct student) * LIMIT);

  /* 読み込み */
  for (nstudents = 0; nstudents < LIMIT; nstudents++) {
    if (fgets(buf, sizeof(buf), stdin) == NULL)
      break;                                    /* 入力の終わりで抜ける */
    len = strlen(buf);
    if (buf[len-1] == '\n') buf[len-1] = '\0';  /* 行末の改行文字を消す */
    buf[7] = '\0';
    id = atoi(buf);
    account = buf + 8;
    while (*account == ' ') account++;
    set(&v, id, account);              /* v の id と account を設定 */
    insert(students, nstudents, v);   /* students 配列の適切な位置に v を挿入 */
  }

  /* 表示 */
  for (i = 0; i < nstudents; i++)
    printf("%7d %s\n", students[i].id, students[i].account);

  return 0;
}
```

図 4: アカウント表を整列するプログラムの断片

問 4 [11D]: アカウント表の整列プログラムの改善

student 構造体はアカウントの文字列を保持しているため、要素の挿入時に要するコピーに時間がかかる。問 C のプログラムを修正し、まずカーソルを作って、カーソルをソートしてから student 構造体をコピーするようにせよ。

あるいは、student 構造体へのポインタを配列に納めることにより、要素の挿入時に要するコピーの時間を短縮せよ。

なお、カーソルを用いる場合には、問 C のように整列しながら読み込むことはできない。全ての入力を読み込んでからカーソルを作って整列するか、読み込みながらカーソルだけ整列すればよい。

student 構造体へのポインタを配列に納める場合には、問 C と同様に、1 行を読み込むたびに配列のしかるべき位置に挿入すること。

いずれの方式で実装した場合でも、問 C との比較で、

- 要素比較に必要なメモリ参照の手間、
- 要素挿入に必要なメモリ参照の手間 (読み出しと書き込みに分けて考えること)

を考察すること。