複数の文字を含む区間の検索

石畑　清（明治大学理工学部）
ishihata@cs.meiji.ac.jp

連載第2回は，複数の文字を含む最短区間を見つける問題を取り上げる．1999年12月の京都大会の問題Hである．

この問題は，計算量が$O(n)$のアルゴリズムを見つけることを要求している．ここで，nは文字列の長さである．$O(n^2)$のアルゴリズムでは計算時間が所定の制限（3分）を超えるように，テストデータとして大きなテキストファイルが用意されていた．

教科書に載っていないアルゴリズムを自分で考案するというと，何だか恐ろしげに聞こえるかもしれないが，実際のプログラミングの場面でも，アルゴリズムを自分で作らなければならないケースは結構多い．アルゴリズムの教科書に載っているのは，典型的な問題に対するアルゴリズムだけで，少しでも状況設定が違っていれば，自分で工夫せざるを得ないのである．教科書に書いてある技法を基礎的な知識として使った上で，手順やデータ構造に工夫をこらして，計算速度を上げる努力が必要とされる．

今回取り上げた問題は，このようなアルゴリズムの工夫の例題として良問だと思う．アルゴリズムを勉強したことのある学生なら，この後で紹介する速いアルゴリズムを何とか見つけられるレベルである．逆に簡単すぎるということもなかろう．問題としてのハードルの高さが高からず低からずになっている．

ここで注意を1つ．この連載では，今回と同様，アルゴリズムの工夫をテーマとする問題を多く取り上げることになるだろう．プログラミングについて面白い議論を展開しようとすると，どうしてもこの種の問題を選ぶことになる．しかし，このことから，ACMプログラミングコンテストがアルゴリズムの問題ばかり出題しているとは誤解しないでほしい．実際は逆で，アルゴリズムの工夫を要求する問題は，出題される問題のごく一部でしかない．多くの問題は，問題が設定している仕様を正確に読み取って，それを忠実に実現することを要求する種類のものである．例外的な入力データに対しても正しく動作することが1つのポイントになる．アルゴリズムが苦手であっても，コンテストで勝ち抜くことは十分に可能である．こわがらずに参加してほしい．

■問題：複数の文字を含む区間の検索

長さnの文字列が与えられている．文字列に含まれているのは，ASCIIコード21以上7E以下（それぞれ16進）の表示可能文字だけである．この文字列をテキストと呼ぶ．これとは別に，k個の文字（値の範囲はテキストと同様）が与えられている．k個の文字それぞれをキーと呼ぶ．nとkには，$1 \leqq n \leqq 1000000$，$1 \leqq k \leqq 50$という制約がある．

テキストの中で，k種類のキーすべてを含む区間のうち，最短のものを求めよ．区間の中ではどんな順番でキーが現れてもよいし，同じキーが2回以上現れてもよい．

たとえば，テキストがprogram-promenade，キーがp，m，aだったとする（$n=17$，$k=3$）．テキストの中でキーすべてを含む区間には，program，am-p，m-promena，promenaなどがある．その中で，最も短い（字数が少ない）am-pが求める区間である．キーすべてを含む区間には，上に挙げたもののほかにテキスト全体などいくらでもあるが，それは字数の点で損なので，初めから考えなくてよい．

コンテストでは，最短の区間が複数個ある場合は，同点の区間が何個あるかを答え，さらに最初に現れた区間の文字列をそのまま印刷することにしていた．ここでは，話を簡単にするために，最短の区間の長さだけを答えればよいことにしよう．問題の本質は変わらない．

また，テキストとキーの入力も結構やっかいだが，

ここでは省略する．以下では，必要な情報が次の変数に入力済みだと仮定する．

```
int  n;
char text[1000000];
int  k;
char key[50];
```

n と k の意味は問題どおりである．配列 text の先頭から順に，つまり text[0]~text[n−1] にテキストが入っている．キーは，key[0]~key[k−1] である．key[0]~key[k−1] の中に同じ文字が 2 回以上現れることはない．

なお，以下のプログラムでは 2 つの定数 False と True を使っている．C の #define 指令を使って，False を 0，True を 1 と定義してある．

■単純なプログラム— $O(n^2)$

最も簡単なのは，テキスト内のそれぞれの位置を出発点として，区間をどこまで伸ばせばキーすべてを含むようになるか調べる方法であろう．たとえば，次のようなプログラムになる．

```
int search_segments(void)
{
  int is_key[128], rem;
  int min, len, p, q, c, j;

  for (c = 0; c < 128; c++)
    is_key[c] = False;
  min = n+1;
  for (p = 0; p < n; p++) {
    for (j = 0; j < k; j++) {
      c = key[j];
      is_key[c] = True;
    }
    rem = k;
    for (q = p; q < n; q++) {
      c = text[q];
      if (is_key[c]) {
        is_key[c] = False;
        rem--;
        if (rem == 0) {
          len = q-p+1;
          if (len < min)
            min = len;
          break;
        }
      }
    }
  }
  return (min);
}
```

区間の左端 p を 0 から n−1 まで動かしていく．n−1 までとする代わりに n−k までとしてもよいが，同じことである．それぞれの p の位置で，右端 q を動かしながら，k 種類のキーが出現したか否かを調べていく．変数 rem は，まだ出現していないキーの種類の数である．k 種類のキーがすべて出現したら，q に関する繰返しを止めて，今見つかった区間が最短かどうかを調べる．

プログラミングの上で注意すべきなのは，文字の値を配列の添字として使っている点である．文字の値に基づいて動作を決める部分を高速に実現しようとすれば欠かせないテクニックであろう．この後に示すプログラムにも，同様の配列の使い方がある．C の場合，文字を添字として使う時は，char 型の変数から取り出した値がマイナスにならないか注意しなければならない．この問題の場合は，text にも key にも 21 ～ 7E（16 進）の値しか入れないので，マイナスになることはなく，特段の配慮は不要である．

このアルゴリズムの計算量は明らかに $O(n^2)$ である．最初に述べたように，$O(n^2)$ では遅すぎる．

ここに示したプログラムは，単純明快を旨としたもので，何の工夫もなかった．もっと速くするための細かな工夫はあれこれ可能である．たとえば，text[p] がキーでなければ，何もしなくてよい．p を左端とする区間が最短であるはずはないからである．また，q の繰返しの範囲を p~n−1 としているが，これは p~p+min−1 にできる（テキストの右端に注意）．今までに見つかっている最短区間より長い区間を見つける必要はないからである．これらの工夫によって，$O(n^2)$ ではあっても，かなり速くすることができる．コンテストで合格になる可能性すらある．しかし，これから後に示すアルゴリズムに負けることは間違いない．つまらない努力はやめることにしよう．

■各キーの最新の位置を記憶する方法 — $O(kn)$

もう少し賢い方法に思い至るのはそれほど難しくないだろう．今度は，添字 q でテキストの上を走査しながら，k 種類のキーそれぞれが一番最近に現れた位置を記憶することにする．具体的には，配列 pos を用意して，pos[key[j]] に j 番のキーの最近の位置を記録する．k 種類のキーに対する pos[key[j]] の最小値（$0 \leq j < k$）を p とすると，どのキーも p の点ま

たはそれより右の位置に存在するはずなので，p を左端，q を右端とする区間が作れることになる．

```
int search_segments(void)
{
  int is_key[128], pos[128];
  int min, len, p, q, c, d, j;

  for (c = 0; c < 128; c++)
    is_key[c] = False;
  for (j = 0; j < k; j++) {
    c = key[j];
    is_key[c] = True;
    pos[c] = -1;
  }
  min = n+1;
  for (q = 0; q < n; q++) {
    c = text[q];
    if (is_key[c]) {
      pos[c] = q;
      p = q;
      for (j = 0; j < k; j++) {
        d = key[j];
        if (pos[d] < p)
          p = pos[d];
      }
      if (p >= 0) {
        len = q-p+1;
        if (len < min)
          min = len;
      }
    }
  }
  return (min);
}
```

key[j] がまだ現れていない場合は，pos[key[j]] に −1 を入れることにしている．最小値 p の値が負であれば，まだ現れていないキーがあることになるので，区間を発見した場合の処理は行わない．

このプログラムでは，内側の j に関する繰返し（最小値を求める部分）に $O(k)$ の計算量がかかる．それを囲む q に関する繰返しが $O(n)$ 回だから，全体の計算量は $O(kn)$ である．$O(kn)$ も $O(n^2)$ もパラメータ 2 つの掛け算なので，ともに二乗オーダのアルゴリズムということになるが，明らかに $O(kn)$ の方が $O(n^2)$ よりだいぶ速い．n が 100 万近くになるのに対して，k はせいぜい数十にしかならないからである．この程度のプログラムが書ければ，コンテストでは合格である．

しかし，コンテストから離れて純粋にアルゴリズムの問題として考えた場合，これで満足してはいけない．二乗オーダのアルゴリズムでは不十分である．この問題の場合，$O(n)$ まで速くすることができないか考えてみるべきだろう．実際，以下に示すとおり $O(n)$ のアルゴリズムは実現可能である．

■各キーの最新の位置を双方向リストで管理する— $O(n)$

前項のプログラムと同じ考え方，つまりそれぞれのキーの最近の出現位置を記録する方法のままで，もっと速くできないか考えてみよう．

この方法で必要とされているのは，複数個のデータの値が次々と書き換えられていく状況で，常に最小値を高速に取り出せるようにすることである．これは，優先順位付き待ち行列（priority queue）にほかならない．優先順位付き待ち行列を実現するデータ構造の 1 つとしてヒープが知られている．ヒープを使えば，データの値の書き換えも，最小値の取り出しも $O(\log k)$ の計算量で行える．この問題に対してヒープを使えば，計算量 $O(n \log k)$ のプログラムが得られる．しかし，これはプログラムを極端に複雑にするわりに大したスピードアップになるとは思えない．

実は，この問題の場合は，データの書き換えが特殊な形をしている．データが書き換えられるのは，キーが新たに出現した場合で，現在の位置 q をそのキーの最新の位置として記憶する．これは，他のどのキーの位置よりも大きい．つまり，書き換えられた結果のデータは必ず最大値になる．このことを利用すると，1 回あたり $O(1)$ の計算量で操作のできる優先順位付き待ち行列の実現法が可能であることに気づく．

k 種類のキーそれぞれの出現位置を記録するセルを用意し，これらを双方向リストでつなぐことにする．リスト内の順番は，出現位置の順とする．こうすると，出現位置の最小値は，リストの先頭の要素から取るだけでよいので，すぐに見つかる．キーが新たに出現した場合は，そのキーに対するセルをいったん双方向リストからはずして，リストの末尾に再挿入する．そのセルの値は必ず最大値なので，リストの末尾が正しい位置である．これによって，出現位置の順を保つことができる．双方向のポインタを持ったリストにしたのは，今までの位置からはずす操作を $O(1)$ の計算量で実現するためである．

```
struct cell {
  struct cell *previous, *next;
  int pos;
};
typedef struct cell *list;

struct cell dummy;
struct cell key_cell[128];

void move_to_tail(list x)
{
  list b, f;

  if (x->pos >= 0) {
    b = x->previous;
    f = x->next;
    b->next = f;
    f->previous = b;
  }
  b = dummy.previous;
  b->next = x;
  dummy.previous = x;
  x->previous = b;
  x->next = &dummy;
}

int search_segments(void)
{
  int is_key[128];
  int min, len, p, q, c, j;

  dummy.next = &dummy;
  dummy.previous = &dummy;
  for (c = 0; c < 128; c++)
    is_key[c] = False;
  for (j = 0; j < k; j++) {
    c = key[j];
    is_key[c] = True;
    key_cell[c].pos = -1;
    move_to_tail(&key_cell[c]);
  }
  min = n+1;
  for (q = 0; q < n; q++) {
    c = text[q];
    if (is_key[c]) {
      key_cell[c].pos = q;
      move_to_tail(&key_cell[c]);
      p = dummy.next->pos;
      if (p >= 0) {
        len = q-p+1;
        if (len < min)
          min = len;
      }
    }
  }
  return (min);
}
```

このプログラムを実行した時の双方向リストの様子を**図-1** に示す．使用したデータは，問題の説明の項に挙げた例である．

セル dummy は，双方向リストの先頭の要素の前，最後の要素の次に置かれるダミーである．両端を 1 つのセルでおさえるので，リストは環状になる．最小値は，必ず dummy の次の要素にある．新たなキーが出現したときは，関数 move_to_tail を使って dummy の直前にセルを移動すればよい．move_to_tail の if 文の中が今までの位置からセルをはずす操作，その後が dummy の直前にセルを挿入する操作である．

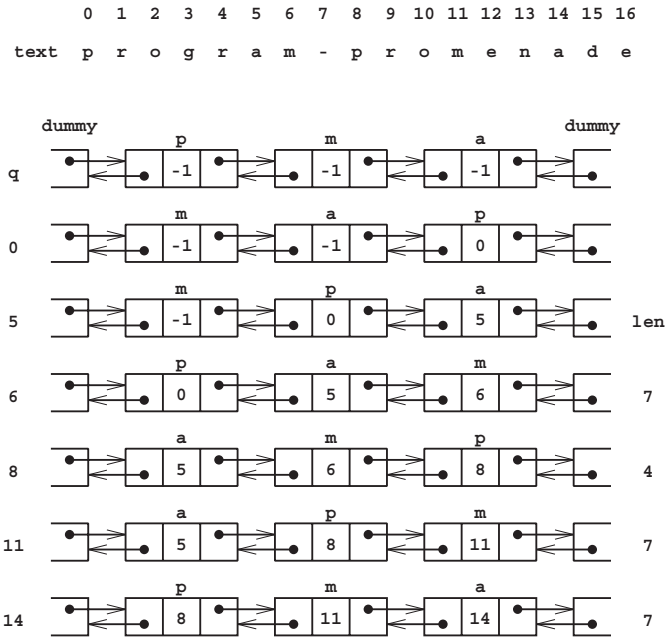

図-1

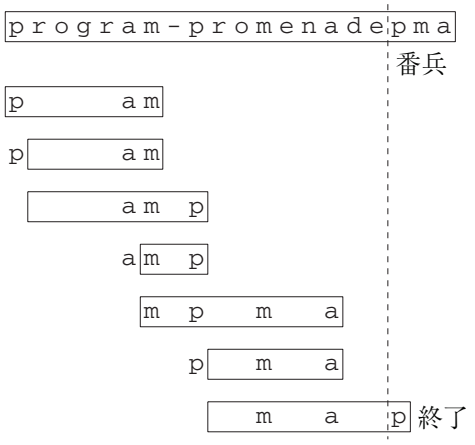

図-2

最小値を求める操作も，リスト内でセルを動かす操作も，$O(1)$ で実現できる．1 文字あたりの計算量が $O(1)$ なので，全体の計算量は $O(n)$ である．

■尺取虫戦略によるプログラム— $O(n)$

$O(n)$ のアルゴリズムが 1 つ見つかった．これより計算量を下げられるはずはないから，オーダだけでいえば，これが最善である．しかし，定数倍のスピードアップはまだ可能かもしれない．同じく $O(n)$ の計算量で，今までの方法とは違う考え方に基づくアルゴリズムを紹介しよう．

今度の方法の概略を疑似プログラムに書くと次のようになる．ここでは，変数 p が区間の左端，q が区間の右端を指している．キーごとの出現数カウンタを用意して，区間 [p .. q] の中のそれぞれの出現回数を数える．

```
p = 0;
q = -1;
for (;;) {
    すべてのキーの出現回数が 1 以上になるまで,
        q を右に動かす.
    q がテキストの右端を過ぎたら終了.
    どれかのキーの出現回数が 0 になるまで,
        p を右に動かす.
    この時点で，局所的に最短の区間が [p−1 .. q] で
    あることが分かる.
}
```

初めに，区間の右端をどんどん右に動かして，区間 [p .. q] がすべてのキーを含むようになったところで止める．次に，区間の左端を右に動かしていく．区間の幅を縮める方向の動きだから，それぞれのキーの出現回数は徐々に減っていく．k 種類のキーのうち，どれか 1 つの出現回数が 0 になったところで，こちらの繰返しを止める．この時点で，区間 [p .. q] は条件を満たさないが，[p−1 .. q] はすべてのキーを含んでいたはずなので，条件を満たす．しかも，区間 [p−1 .. q] は，左右どちらの端も，これ以上縮める方向に動かすと条件を満たさなくなる．つまり，その位置で局所的に最短の区間である．区間を全体として右に動かしながら同じ操作を繰り返せば，局所的に最短の区間を全部調べることになるので，全体の最短区間が見つかるはずである．

この方法の動作例を**図-2** に示す．この図は，q に関する繰返しが終わった時点，p に関する繰返しが終わった時点それぞれにおける区間の位置を順に示したものである．上から下に時間順に並べてある．

```
int search_segments(void)
{
  int is_key[128], count[128], rem;
  int min, len, p, q, c, j;

  for (c = 0; c < 128; c++)
    is_key[c] = False;
  for (j = 0; j < k; j++) {
    c = key[j];
    is_key[c] = True;
    count[c] = 0;
    text[n+j] = c;      /* 番兵 */
  }
  rem = k;
  min = n+1;
```

```
  p = 0;
  q = -1;
  for (;;) {
    for (;;) {
      q++;
      c = text[q];
      if (is_key[c]) {
        count[c]++;
        if (count[c] == 1) {
          rem--;
          if (rem == 0)
            break;
        }
      }
    }
    if (q >= n)
      break;
    for (;;) {
      c = text[p];
      p++;
      if (is_key[c]) {
        count[c]--;
        if (count[c] == 0) {
          rem++;
          break;
        }
      }
    }
    len = q-(p-1)+1;
    if (len < min)
      min = len;
  }
  return (min);
}
```

このプログラムでは，最初に配列 text の n 番以降，つまりテキストの直後にキーの k 種類の文字をコピーしている．これは一種の番兵である．テキストの終わりを越えたら，必ずすべてのキーを含む区間が見つかる．これによって，テキストの終わりに到達したかどうかのチェックを q を動かす繰返しの中から外に追い出すことが可能になっている．このため，このプログラムでは，配列 text の長さを 1000050 としなければならない．

このプログラムの計算量は $O(n)$ である．二重のループの格好になっていて，外側のループも内側のループも最大 n 回繰り返すことがあるが，全体で n^2 になることは起こらない．変数 p も q も増える一方なので，内側の 2 つのループを回る回数がそれぞれ通算で n を超えないからである．

このプログラムにおける区間の動きは面白い．右端 q を少し動かして先頭を伸ばし，次に左端 p を動かして尻尾を縮める．これを繰り返しながら全体として右に移動していく．この様子は細長い虫が地面を這っていく様子に似ている．そこで筆者は，このやり方を尺取虫方式と名付けた．

尺取虫方式が適用できるのは，この問題だけではない．ほかにも尺取虫方式が有効な問題は数多く存在する．1 次元配列の上で最長区間や最短区間を求める問題の場合は，尺取虫方式をまず考えてみるとよいと思う．アルゴリズムの発想法として分割統治法などが知られているが，筆者の経験では分割統治法よりも尺取虫方式の方が有効になる例が多いくらいである．

なお，尺取虫方式は，ある関係を保ったまま，2 つの添字を並行して進めていくという点でマージの動作と似ている．マージのプログラムは，普通次のように書く．

```
while ( .....) {
  if ( ..... ) {
    .....; i++;
  }
  else {
    .....; j++;
  }
}
```

これは，次のように書き換えることができる．

```
while ( ..... ) {
  while ( ..... ) {
    .....; i++;
  }
  while ( ..... ) {
    .....; j++;
  }
}
```

こうしてみると，尺取虫とマージは同じ原則に基づくプログラムの書き方であることが納得できると思う．

尺取虫のプログラムと 1 つ前の双方向リスト方式は，ともに $O(n)$ なので，どちらが速いか気になるところである．筆者は，詳しい検討をしていないので，この問に対する答を持ち合わせていない．尺取虫方式は，テキスト内の文字を 2 回ずつ調べるが，変数に対する操作は単純なもので済む．これに対して，双方向リストの場合は，リストセル内のポインタの操作が複雑で，メモリのアクセス時間がネックになりそうである．精密な実験をしてみないと，どちらが速いか結論を出すわけにはいかない．

（平成 14 年 3 月 19 日受付）