平成 21 年度

東京大学大学院情報理工学系研究科

コンピュータ科学専攻

入学試験問題

専門科目 I

解答

is2009

2012 年 8 月 7 日

問題 1

(1)

$$P_0' = (1-t)P_0 + tP_1$$
$$P_1' = (1-t)P_1 + tP_2$$

より

$$P = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2$$

(2)

二次のベジエ曲線が放物線であることはよく知られている。確かめたいときは $P_0 = (1,1), P_1 = (0,0), P_2 = (-1,1)$ にして確かめてみるとよい。

(3)

$$P = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

(4)(!)

図のように点を配置した……という文言より、

$$P_0 = (r, 0)$$
$$P_1 = (r, s)$$
$$P_2 = (s, r)$$
$$P_3 = (0, r)$$

とできる。その上で

$$\begin{aligned} P(1/2) &= (r\cos(\frac{\pi}{4}), r\sin(\frac{\pi}{4})) \\ &= (\frac{r}{\sqrt{2}}, \frac{r}{\sqrt{2}}) \end{aligned} \tag{1}$$

これを用いて

$$P(\frac{1}{2}) = (\frac{4r+3s}{8}, \frac{4r+3s}{8}) \tag{2}$$

式 **??**、式 **??**より、

$$s = \frac{4(\sqrt{2}-1)r}{3}$$

という解き方でも問題的にはいける気がするんですが、どうでしょう？数学的なとき方とかあったら教えてください。

(5)

これは説明しづらいので各自 CG の教科書の 66 ページを参照してください。

問題 2

(1)

必要な空間：$O(mn)$　（長さ n のビットベクトルが m 本）
$member$ の手間：$O(1)$　（1 箇所の値を見ればいいだけ）
$find$ の手間：$O(m)$　（m 個のビットベクトルを全部見る必要がある）
$merge$ の手間：$O(n)$　（長さ n のビットベクトルを新たにに計算）

$member$ が効率的に行えるので，$member$ の処理を多く行いたいようなときに使うべきである．

(2)

必要な空間：$O(n)$　（常に n 個のノード）
$member$ の手間：$O(\frac{n}{m})$ [*1]　（1 つのリストを走査する必要がある）
$find$ の手間：$O(n)$　（全てのリストを走査する必要がある，ノードは全部で n 個）
$merge$ の手間：$O(n)$　（連結のため，リストの末尾を求める必要がある．末尾を保存していれば $O(1)$）

必要な空間が小さいので，空間を小さくしたいようなときに使うべきである．

(3)

ハッシュ表の大きさを h とし，n に対し衝突が無視できる程度に大きいとする．

必要な空間：$O(h)$　（ハッシュ表の大きさ分）
$member$ の手間：$O(1)$　（ハッシュ表を引けばよい）
$find$ の手間：$O(1)$　（ハッシュ表を引けばよい）
$merge$ の手間：$O(n)$　（含まれる全ての要素の値を更新する必要がある）

$member$ と $find$ が効率的に行えるので，それらの処理を多く行いたいときに使うべきである．

(4)

Union-Find 木を用いればよい．これは，各集合を木で表した物であり，各ノードは親ノードへのポインタを持つ．集合を木の根のノードで区別する．

$merge$ の処理は，2 つの木のうち片方の木の根の親を，もう片方の木の根に設定すれば良い．また，$find$ の処理は，ノードを根へ辿ればよい．

ここで，$merge$ の際に，2 つの木のうちノード数の少ない方の木を多い方の木の下につけるようにする．また，$find$ の処理の際に，根を求めたら，辿った全てのノードに関して，親ノードへのポインタを根へ張り替える．以上の工夫を行うと，$find$ 処理の計算量は平均して $O(\log n)$ になる．

[*1] m で割るべきかは「平均」の意味が曖昧なので微妙

解説

Union-Find 木は五十嵐先生の教科書では Merge-Find 木とかいう名前で乗っています．僕のスライド (http://www.slideshare.net/iwiwi/ss-3578491) でも解説しているので良かったらどうぞ．

上の工夫を両方やると，計算量は実際にはアッカーマン関数の逆関数とかになってもっと速いんですが，ランダウの記号的に，それを $O(\log n)$ と言っても問題ないです．

問題 3

(1)

$L(M)$ は決定性有限オートマトンに受理されるから, 次のように定義される決定性有限オートマトン $M = (Q_1, \Sigma_1, \delta_1, q_0^1, F_1)$ を考えればよい.

- $Q_1 = Q$ 状態は M の状態と同じ物を用いる
- $q_0^1 = q_0$ 初期状態も M の初期状態と同じ物を用いる
- $\delta_1 = \delta$ 状態遷移関数も M の状態遷移関数と同じ物を用いる
- $F_1 = Q - F$ 受理状態は M の状態のうち受理状態でないものを用いる

この決定性有限オートマトンは確かに $\Sigma^* - L(M)$ を受理し，そうでないものは受理しない. よって，決定性有限オートマトンで受理可能であるため，命題は証明された.

(2)

次のようなアルゴリズムを用いることで，$L(M) = \emptyset$ となっているかどうかを判定することができる.

```
begin
  OLD := null;
  NEW := q0;
  while OLD != NEW do
    OLD := NEW;
    NEW := NEW or {q|q は NEW に含まれる状態から直接遷移できる状態}
  end
  if NEW contains F then
    return NO;
  else
    return YES;
  end
end
```

Q に含まれる状態は有限個であり，while 文をの中を 1 度実行するたびに NEW に含まれる状態は増える (増えない場合は while 文を抜ける) ため，このアルゴリズムは必ず停止する．また，このアルゴリズムは必ず正しい答えを返す．よってこのアルゴリズムは完全正当性を満たす（註:完全正当性はそのアルゴリズムが必ず停止し，常に正しい答えが帰ってくることを要求する)

(3)

存在する．アルゴリズムの概要を以下に示す．

まず，次のように定義される決定性有限オートマトン $M = (Q, \Sigma, \delta, q_0, F)$ を作る．

- $Q = Q_1 \times Q_2$
- $q = (q_0^1, q_0^2)$
- $\delta((q^1, q^2), a) = (\delta_1(q^1, a), \delta_2(q^2, a))(q^1 \in Q_1, q^2 \in Q_2, a \in \Sigma)$
- $F = F_1 \times \bar{F_2}$

(註:M_1 と M_2 の両方を同時に動かすイメージ)

この決定性有限オートマトンは，$L(M_1) - L(M_2)$ なる言語を受理する．ここで，(2) で与えたアルゴリズムを用いることで，$L(M_1) - L(M_2) = \emptyset$ であるかどうかを判定することができる．ここで，$(L(M_1) - L(M_2) = \emptyset) \Longleftrightarrow L(M_1) \subseteq L(M_2)$ であるから，以上により判定することができる．

問題 4

※ソースは動作確認していないのでおかしいところがあるかもしれません

(1)

問題点

1. OP_OPEN でファイルをオープンした後に OP_READ でファイルの内容を読み出す使い方を想定して作成されたものと考えられるが、変数 fd が static 指定されていないため OP_OPEN で開いたファイルのファイルディスクリプタを後に続く OP_READ の操作まで保存しておくことができない。
2. switch 文内の case OP_OPEN:で pkt->hd.sz の値を確認しないまま rcv 関数を呼んでいるため、pkt->data のサイズ (512 バイト) を超えた値を指定してデータを送りつけられた場合にメモリを上書きされてしまうといういわゆるバッファオーバーフローの脆弱性がある。

セキュリティホールのメカニズム

バッファオーバーフロー攻撃では受け取り用バッファのサイズを越えてデータを送りつけることでプログラムを誤動作させる。特に悪意のあるコードによってメモリ領域を上書きした場合はそこへプログラムカウンタが移動したときに動作を乗っ取ることが出来てしまう。C 言語などではスタック内でローカル変数に割り当てられる領域の高位に関数の戻り先アドレスを保存しているため、そのアドレスを書き換え悪意のあるコードへジャンプさせることが出来る。

(メモ)

「変数 fd に初期値を代入していないため、OP_OPEN の前に OP_READ を送信してしまった場合に無関係なファイルの内容を読み出してしまう可能性がある。」というのも問題だと思いますが「致命的な問題が一つ」とあるので上記のとおりにしておきました。

ソースコード

```
1  void op(int sock, packet *pkt) {
2    static int fd = -1; /* fix(1) */
3    int n = -1;
4    switch (pkt->hd.cmd_or_stat) {
5      case OP_OPEN:
6        if (pkt->hd.sz >= sizeof (pkt->data)) { /* fix (1) */
7          pkt->hd.cmd_or_stat = FAIL;           /* fix (1) */
8          pkt->hd.sz = 0;                       /* fix (1) */
9          break;                                /* fix (1) */
10       }                                       /* fix (1) */
11       rcv(sock, pkt->hd.sz, pkt->data);
12       pkt->data[pkt->hd.sz] = 0;
13       fd = open(pkt->data, O_RDWR);
14       if (fd > 0) pkt->hd.cmd_or_stat = SUCCESS;
15       else pkt->hd.cmd_or_stat = FAIL;
16       pkt->hd.sz = 0;
17       break;
18     case OP_READ:
19       if (fd > 0) n = read(fd, pkt->data, 512);
20       if (n < 0) {
21         pkt->hd.cmd_or_stat = FAIL;
22         pkt->hd.sz = 0;
23       } else {
```

```
24          pkt->hd.cmd_or_stat = SUCCESS;
25          pkt->hd.sz = n;
26        }
27        break;
28      case OP_CLOSE:
29        /* the following code is omitted. */
30    }
31  }
```

(2)

```
1  void client(void) {
2    const filename = "/home/foo/data.txt";
3    packet pkt;
4    int sock = reqConnection();
5    pkt.hd.cmd_or_stat = OP_OPEN;
6    pkt.hd.sz = strlen(filename);
7    strcpy(pkt.data, filename);
8    snd(sock, sizeof (header) + pkt.hd.sz, &pkt);
9    rcv(sock, sizeof (header), &pkt);
10   if (pkt.hd.cmd_or_stat == SUCCESS) {
11     while (1) {
12       pkt.hd.cmd_or_stat = OP_READ;
13       snd(sock, sizeof (header), &pkt);
14       rcv(sock, sizeof (header), &pkt);
15       if (pkt.hd.cmd_or_stat == SUCCESS && pkt.hd.sz > 0) {
16         rcv(sock, pkt.hd.sz, pkt.data);
17         pkt.data[pkt.hd.sz] = 0;
18         printf("%s", pkt.data);
19       } else {
20         break;
21       }
22     }
23     pkt.hd.cmd_or_stat = OP_CLOSE;
24     pkt.hd.sz = 0;
25     snd(sock, sizeof (header), &pkt);
26     rcv(sock, sizeof (header), &pkt);    // (*)
27   }
28   closeConnection(sock);
29 }
```

(*) の行は必要かどうか不明。問題本文のソースコードでは OP_CLOSE の内容が省略されているが、とりあえず OP_CLOSE の送信は書いておいたほうが無難だろう。

read システムコールの戻り値 0 はファイル終端を意味するのでループを抜ける。FAIL の場合も同様にそこで表示を終了することにしている。

(3)

read システム関数の呼び出し回数を減らすことで応答性能を向上させることができるため、一つのパケットに格納できるサイズより大きいバッファを用意して 1 回の read 呼び出しで読み出すデータの量を増やし、そのバッファからデータを取り出すようにすればよい。

(この問題もソースコードは必要なんだろうか？)

```
1  void op(int sock, packet *pkt) {
2    static int fd = -1; /* fix (1) */
3    static char buf[4096]; /* fix (3) */
4    static int nread = -1; /* fix (3) */
5    static int head = 0;   /* fix (3) */
6    int n = -1;
7    switch (pkt->hd.cmd_or_stat) {
8      case OP_OPEN:
9        if (pkt->hd.sz >= sizeof (pkt->data)) { /* fix (1) */
10         pkt->hd.cmd_or_stat = FAIL;           /* fix (1) */
11         pkt->hd.sz = 0;                       /* fix (1) */
12         break;                                /* fix (1) */
```

```
13        }                                          /* fix (1) */
14        rcv(sock, pkt->hd.sz, pkt->data);
15        pkt->data[pkt->hd.sz] = 0;
16        fd = open(pkt->data, O_RDWR);
17        if (fd > 0) pkt->hd.cmd_or_stat = SUCCESS;
18        else pkt->hd.cmd_or_stat = FAIL;
19        pkt->hd.sz = 0;
20        break;
21      case OP_READ:
22        if (nread <= head) {                                  /* fix (3) */
23          if (fd > 0) nread = read(fd, buf, sizeof (buf)); /* fix (3) */
24          head = 0;                                         /* fix (3) */
25        }                                                     /* fix (3) */
26        n = nread - head > 512 ? 512 : nread - head;        /* fix (3) */
27        if (n > 0) {                                          /* fix (3) */
28          memcpy(pkt->data, buf + head, n);                 /* fix (3) */
29          head += n;                                        /* fix (3) */
30        }                                                     /* fix (3) */
31        /* if (fd > 0) n = read(fd, pkt->data, 512); */     /* fix (3) */
32        if (n < 0) {
33          pkt->hd.cmd_or_stat = FAIL;
34          pkt->hd.sz = 0;
35        } else {
36          pkt->hd.cmd_or_stat = SUCCESS;
37          pkt->hd.sz = n;
38        }
39        break;
40      case OP_CLOSE:
41        /* the following code is omitted. */
42    }
43  }
```

専門科目 II

問題 2-2

(1)

9

(2)

以下のような $h(l)$ を考える．

$$h(l) = \sum_{i=1}^{l} v_i$$

これを $l = 1, 2, ..n$ について求めておく．順番に求めれば $O(n)$ で計算できる．また，簡単のため $h(0) = 0$ としておく．

$h(l)$ を用いると，

$$g(i,j) = \frac{1}{j-i+1}\left(h\left(j\right) - h\left(i-1\right)\right)$$

とすればよく，任意の $g(i,j)$ を $O(1)$ で計算できる．

(3)

以下のように計算すればよい．

$$S_k(X) = \min_{k-1 \le i < n} \left\{ \frac{1}{k}\left(\left(k-1\right) \times S_{k-1}\left(X[1..i]\right) + g\left(i+1, n\right)\right) \right\}$$

(4)

動的計画法によって計算すればよい．

$$dp1(k,i) = \begin{cases} g(1,i) & k = 1 \\ \min_{k-1 \le j < i} \left\{ \frac{1}{k}\left(\left(k-1\right) \times dp1(k-1,j) + g\left(j+1, i\right)\right) \right\} & k > 1 \end{cases}$$

とすると，(3) と同様に，$dp1(k,i)$ は $S_k(X[1..i])$ となり，$dp1(k,n)$ が $S_k(X)$ となる．

また，$k > 1$ に対し，

$$dp2(k,i) = \operatorname*{arg\,min}_{k-1 \le j < i} \left\{ \frac{1}{k}\left(\left(k-1\right) \times dp1(k-1,j) + g\left(j+1, i\right)\right) \right\}$$

とする．$dp2(k,i)$ は，$dp1(k,i)$ を達成する分割（つまり，$D_{min}(X[1..i], k)$ ）の一番最後の分割位置を意味する．

これらの必要な値は，k や i が小さい順に計算してゆけばよく，$O(kn^2)$ で計算できる．

次に，

$$f(k,i) = \begin{cases} X[1..i] \text{ のみを含む列} & k = 1 \\ f(k-1, dp2(k,i)) \text{ の末尾に } X[dp2(k,i)+1..i] \text{ を加えた列} & k > 1 \end{cases}$$

とすると，$D_{min}(X,k) = f(k,n)$ となる．この計算は，$O(k+n)$ で行える．

以上で，$D_{min}(X,k)$ を $O(kn^2 + (k+n)) = O(kn^2)$ で求めることができる．

解説

文章にすると非常に分かりにくいですが，要は値を記憶しながら (3) のようなことを繰り返せば全部計算できる，ということ．

動的計画法の問題を考える際には，状態が何であるかを考えることが重要です．例えば，この問題では，場所 i で分割され，それより左で k 回分割されるのならば，i 以降がどう分割されようが，i 以前の最善な分割のされ方は変わらない．よって，$dp1$ や $dp2$ の引数は k, i のみで良く，従って状態は $O(nk)$ 個で良いわけです．

問題 3

(1)

```
1  #define SIZE 256
2
3  typedef int data_t;
4
5  struct stack {
6    int top;
7    data_t data[SIZE];
8  };
9
10 struct stack *create (void)
11 {
12   struct stack *stk;
13   stk = (struct stack *)malloc(sizeof(struct stack));
14   stk->top = 0;
15   return stk;
16 }
17
18 void push (struct stack *stk, data_t d)
19 {
20   if (stk->top >= SIZE) {
21     fprintf(stderr, "stack␣overflow\n");
22     exit(EXIT_FAILURE);
23   } else {
24     stk->data[stk->top] = d;
25     stk->top++;
26   }
27 }
28
29 data_t pop (struct stack *stk)
30 {
31   if (stk->top < 1) {
32     fprintf(stderr, "stack␣underflow\n");
33     exit(EXIT_FAILURE);
34   } else {
35     data_t d;
36     stk->top--;
37     return stk->data[stk->top];
38   }
39 }
```

(2)

(a)

モジュールの定義の変更に対する耐久性がないのが問題である。

モジュールの外からの直接的なモジュール内のデータの変更は、モジュールの実装に依存したものとなる。そのためそのような命令は、モジュールの定義が変更されたときに意図していなかった動作をする可能性があり、不便である上、危険である。モジュールの定義の変更に合わせてそのような命令を変更するにあたっても、命令ごとに一々変更する必要があり、面倒な上、正しくない変更をしてしまう可能性や変更漏れの起きる可能性が高い。

(b)

データ隠蔽と呼ばれる機能を提供すればよい。

すなわち、モジュールの構成要素ごとにモジュール外からのアクセス可能性を定められるようにすればよい。

データ隠蔽により、モジュールが提供する機能を利用するのに必要な手続きや変数以外にはモジュール外からアクセスできないようになり、変更に対する耐久性 (とモジュールの抽象化可能性) が向上する。

問題 4

(1)

X_3	X_2	X_1	X_0	L_1	L_0	Z
0	0	0	0	*	*	1
0	0	0	1	1	1	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

(2)

$L_1 = (\text{not } X_3) \text{ and } (X_2 \text{ or } ((\text{not } X_1) \text{ and } X_0))$
$L_0 = (\text{not}(X_3 \text{ or } X_2))$
$Z = \text{not } (X_3 \text{ or } X_2 \text{ or } X_1 \text{ or } X_0)$

(3)

(4) をみてね。

(4)

N が 2^m で表せないとき X の LSB 側を 0 で拡張し入力長が 2^m であるものとして考える。N は高々 2 倍にしかならないので遅延時間が対数オーダーならば議論に影響を与えない。

(i)

$N <= 4$ のとき (1) の回路を利用すればよい。

(ii)

$N = 2^k$ のとき遅延時間 O(logN) の回路 C_k が構成できたとする。この回路を利用して $N = 2^{k+1}$ の時の遅延時間 O(logN) の回路の構成法を示せばよい。

構成する回路の出力を L_k - L_0 , X とする。上位 2^k ビットを C_k に入力。出力を $L0_{k-1}$ - $L0_0$, $X0$ とする。下位 2^k ビットを C_k に入力。出力を $L1_{k-1}$ - $L1_0$, $X1$ とする。

出力を以下のように構成する。

・$L_k = X0$

・$0 <= i <= k-1$ に対して $L_i = (L0_i \text{ and } (\text{not } X0))$ or $(L1_i \text{ and } X0)$

・$X = X0 \text{ and } X1$

N が 2 倍になっているのに対して遅延時間は定数増加しているのでこの回路の遅延時間は O(logN) である。

(i)(ii) より任意の N に対して遅延時間 O(logN) の回路の構成法を示すことが出来た。