令和 2 年度

千葉大学先進科学プログラム入学者選考課題

課題論述　　情報　課題 I, II, III

解答例

[I]

問 1 53
0

問 2 52
104

問 3 このプログラムは開平法を実現したものであり，次のとおりとなる。

$n = e^2 + a$

問 4
(11) 行目の 100 を 64 に変更
(16) 行目の 100 を 64 に変更
(19) 行目の 10 を 8 に変更
(26) 行目の 10 を 8 に変更

[II]

問 1　地図 B において, 地域数が 12 であることから各地域を格納するための要素数は 12, 1 つの地域が隣接する地域が最大 5 であることから隣接する地域を格納するための要素数は終端（-1）を含め 6 あればよい。

地域 0 は, 地域 1 および地域 2 と接していることから {1，2，-1} と表記でき, 地域 1 は, 地域 0, 地域 2 および地域 6 に接していることから {0，2，6，-1} と表記できる。他の地域についても同様にすると, 隣接関係 areaB は下のコードのように記述できる。

リスト 1: 隣接関係 areaB

```
int areaB[12][6] = {
  {1, 2, -1},
  {0, 2, 6, -1},
  {0, 1, 3, 5, 6, -1},
  {2, 4, 5, -1},
  {3, 5, -1},
  {2, 3, 4, 6, 7, -1},
  {1, 2, 5, 7, 9, -1},
  {5, 6, 8, 9, -1},
  {7, 9, 10, 11, -1},
  {6, 7, 8, 10, -1},
  {8, 9, 11, -1},
  {8, 10, -1}
};
```

問 2　与えられる引数を a，b としたとき, 地域 a について, 隣接関係 areaB を参照して areaB[a][0]～areaB[a][5] に地域 b が含まれるかを調べればよい。終端 (-1) を見つけた場合には地域 a と地域 b は接していない。

リスト 2: 関数 isbeside

```
int isbeside(int a, int b) {
  int i;
  for(i=0; areaB[a][i] != -1; i++) {
    if(areaB[a][i] == b)
      return 1;
  }

  return 0;
}
```

問 3

リスト 3: 色分けを行うコード

```
#include<stdio.h>

int main(void) {
  int color[12];
  color[0] = 0;
  int cur, col, fixed, flag;

  for(cur=1; cur<12; cur++) {
    for(col = 0; ; col++) {
      for(fixed=0; fixed<cur; fixed++) {
        flag = 0;
        if(isbeside(cur, fixed)==1 && col==color[fixed]) {
          flag = 1;
          break;
        }
      }
      if(flag==0) {
        color[cur] = col;
        break;
      }
    }
  }

  /* 以下3行は確認用の表示部分。回答には必要ない。 */
  for(cur=0; cur<12; cur++)
    printf("%d, ",color[cur]);
  printf("\n");

  return 0;
}
```

上のコードを実行すると, color[12] = {0, 1, 2, 0, 1, 3, 0, 1, 0, 2, 1, 2} が得られる。これに従うと, 地図 B を図 3 に示すように塗分けられる。

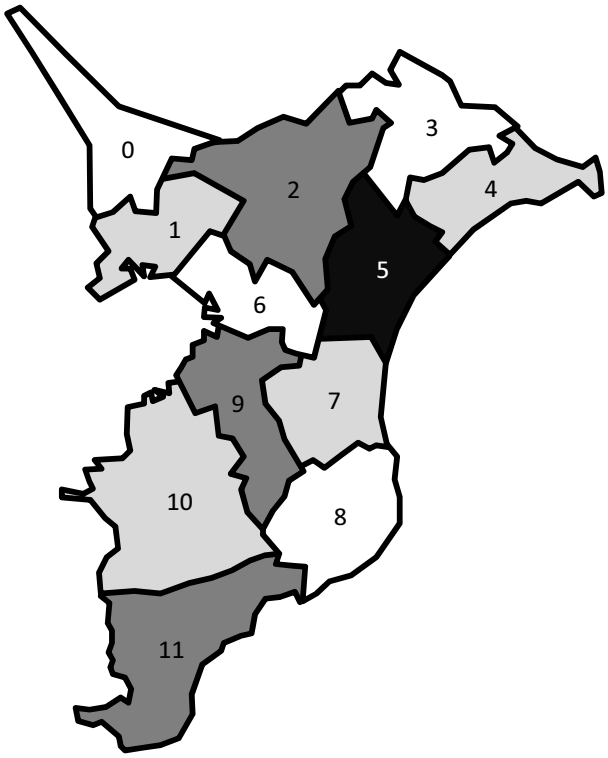

図 3: 地図 B の塗分け例

[III]

問 1

(a)
```
for( k=0; k<n; k++){
  C[i][j] = C[i][j] + A[i][k] * B[k][j];
}
```

(b) 式 (3) より、ひとつの $c_{i,j}$ の計算には a、b 同士の乗算が n 回必要である。また全成分数は n^2 であることから、全 $c_{i,j}$ を計算するのに必要な乗算の回数は

$$n \times n^2 = n^3$$

である。

問 2

(a) $b_{i,j}$ 全体を常にキャッシュメモリ上に格納できることから、はじめに全体を読み込めばよい。したがって読み込む回数は $b_{i,j}$ の総数と等しく n^2 。

(b) 問 1(b) と 2(a) の結果から読み込みにかかる時間は n^2T_m、演算にかかる時間は n^3T_c である。したがって合計時間は $T = n^3T_c + n^2T_m$

(c) $n^2T_m > n^3T_c$ の場合、すなわち $n < T_m/T_c$ の場合、読み込みが支配的である。逆に $n > T_m/T_c$ の場合、演算が支配的である。

問 3

(a) 例えば $c_{1,1}$ を計算するのに必要な b は $b_{,1}$ の列である。サンプルコードのループでは次に計算するのは $c_{1,2}$ であり、必要な b は $b_{,2}$ の列である。$c_{1,1}$ と $c_{1,2}$ の計算に必要な b の重複がないため、各 c の計算ごとに n 個の b を読み直さねばならない。

ここで i ループと j ループを入れ替えると、$c_{1,1}$ の次に計算するのは $c_{2,1}$ であり、必要な b は $b_{,1}$ の列であり、$c_{1,1}$ で必要な b と同じであるため b を再利用でき、再度読み直す必要はない。これは列が同じ限り他の行にも当てはまり、$c_{,1}$ 全体に対して必要な b の読み込み回数は n である。したがって全体で必要な読み込み回数はこれに行数をかけた n^2 となる。

(b) 大きいサイズの行列はキャッシュメモリ上に乗り切らない。そこでキャッシュメモリに乗る程度のサイズの行列に分解することを考える。つまり大きな行列と行列の積を、小さい行列と行列の積と和に変更する。具体的には以下のようにする。

A、B、C 全体を $n_\mathrm{b} \times n_\mathrm{b}$ の小行列に分解する。小行列全体がキャッシュメモリに乗るようにサイズ n_b を調整する。小行列同士の行列積を行い、結果を足し合わせて全体の行列積を計算できる。

```
int i, j, k, ib, jb, kb;
for( i=0; i<n; i+=nb){
  for( j=0; j<n; j+=nb){
    for( k=0; k<n; k+=nb){
      for( ib=i; ib<(i+nb); ib++){
        for( jb=j; jb<(j+nb); jb++){
          for( kb=k; kb<(k+nb); kb++){
            C[ib][jb] += A[ib][kb] * B[kb][jb];
          }(以下省略)
```