C_Program/print_all.txt

```
------------------------------------------------
FILE: ALDS1_1_A.c
------------------------------------------------
/*
1 insertionSort(A, N) // N個の要素を含む0-オリジンの配列A
2   for i が 1 から N-1 まで
3     v = A[i]
4     j = i - 1
5     while j >= 0 かつ A[j] > v
6       A[j+1] = A[j]
7       j--
8     A[j+1] = v

*/

/*
テスト用の解説メモ

入出力の流れ
- 入力: N の後に N 個の整数
- 出力: ソートの各ステップごとに配列全体を1行出力

出力タイミング
1) 初期配列を1行出力
2) i=1..N-1 の各挿入操作後に1行出力
   → 合計 N 行が出力される

手計算確認例
入力
6
5 2 4 6 1 3

出力
5 2 4 6 1 3
2 5 4 6 1 3
2 4 5 6 1 3
2 4 5 6 1 3
1 2 4 5 6 3
1 2 3 4 5 6

ポイント
- 挿入対象 v を左へずらし、v より大きい要素を右に1つずつ移動
- 既に整列済みの部分配列 [0..i-1] を保つのが目的
*/
#include <stdio.h>

void insertionSort(int A[], int N);
void printArray(int A[], int N);

int main(){
    int N;
    scanf("%d", &N);

    int A[N];

    for(int i = 0; i < N; i++){
        scanf("%d", &A[i]);
    }
```



```
    insertionSort(A, N);
}

void insertionSort(int A[], int N){
    printArray(A, N);
    for(int i = 1; i < N; i++){
        int v = A[i];
        int j = i - 1;
        while(j >= 0 && A[j] > v){
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = v;
        printArray(A, N);
    }
}

void printArray(int A[], int N){
    for(int i = 0; i < N; i++){
        if (i > 0) printf(" ");
        printf("%d", A[i]);
    }
    printf("\n");
}


------------------------------------------------
FILE: ALDS1_1_B.c
------------------------------------------------
/*
最大公約数
２つの自然数 x, y を入力とし、それらの最大公約数を求めるプログラムを作成してください。

２つの整数 x と y について、x ÷ d と y ÷ d の余りがともに 0 となる d のうち最大のものを、
x と y の最大公約数（Greatest Common Divisor）と言います。
例えば、35 と14 の最大公約数 gcd (35, 14) は 7 となります。
これは、35 の約数{1, 5, 7, 35}、14 の約数 {1, 2, 7, 14} の公約数 {1, 7} の最大値となります。

入力
x と y が１つの空白区切りで１行に与えられます。

出力
最大公約数を１行に出力してください。

制約
1 ≤ x, y ≤ 109
ヒント
整数 x, y について、x ≥ y ならば x と y の最大公約数は y と x % y の最大公約数に等しい。ここで x % y は x を y で割った余りである。

入力例 1
147 105
入力例 1 に対する出力
21
*/
#include <stdio.h>

int gcd(int x, int y);

int main() {
    int x, y;
    scanf("%d %d", &x, &y);
```



```
    int result = gcd(x, y);
    printf("%d\n", result);

    return 0;
}

int gcd(int x, int y){
    int tmp;
    if (x < y) {
        tmp = x;
        x = y;
        y = tmp;
    }

    if (y == 0) {
        return x;
    }
    return gcd(y, x % y);
}



------------------------------------------------
FILE: ALDS1_1_C.c
------------------------------------------------
/*
素数
約数が 1 とその数自身だけであるような自然数を素数と言います。
例えば、最初の8 つの素数は2, 3, 5, 7, 11, 13, 17, 19 となります。1 は素数ではありません。

n 個の整数を読み込み、それらに含まれる素数の数を出力するプログラムを作成してください。

入力
最初の行に n が与えられます。続く n 行に n 個の整数が与えられます。

出力
入力に含まれる素数の数を１行に出力してください。

制約
1 ≤ n ≤ 10,000
2 ≤ 与えられる整数 ≤ 108
入力例 1
6
2
3
4
5
6
7
入力例 1 に対する出力
4
*/
#include <stdio.h>

int isPrime(int x);

int main(){
    int n;
    scanf("%d", &n);

    int x[n];
    for(int i = 0; i < n; i++){
```



```
        scanf("%d", &x[i]);
    }

    int count = 0;
    for(int i = 0; i < n; i++){
        if(isPrime(x[i])){
            count++;
        }
    }
    printf("%d\n", count);
    return 0;
}

int isPrime(int x){
    if(x < 2) return 0;
    for(int i = 2; i * i <= x; i++){
        if(x % i == 0){
            return 0;
        }
    }
    return 1;
}


------------------------------------------------
FILE: ALDS1_1_D.c
------------------------------------------------
/*
最大の利益
FX取引では、異なる国の通貨を交換することで為替差の利益を得ることができます。
例えば、１ドル100円の時に 1000ドル買い、価格変動により 1ドル 108円になった時に売ると、
(108円 100円)
 1000ドル
 8000円の利益を得ることができます。

ある通貨について、時刻
 における価格
 (
)が入力として与えられるので、価格の差
 (ただし、
 とする）の最大値を求めてください。

入力
最初の行に整数
 が与えられます。続く
 行に整数
 (
) が順番に与えられます。

出力
最大値を１行に出力してください。

制約
入力例 1
6
5
3
1
3
4
3
入力例 1 に対する出力
```

```
3
入力例 2
3
4
3
2
入力例 2 に対する出力
-1
*/

#include<stdio.h>

int difference(int x[],int n);

int main(){
    int n;
    scanf("%d",&n);

    int x[n];

    for(int i = 0; i < n; i++){
        scanf("%d",&x[i]);
    }

    int differences = difference(x,n);
    printf("%d\n",differences);

    return 0;
}

int difference(int x[],int n){
    int minv = x[0];
    int maxv = x[1] - x[0];

    for(int i = 1; i < n; i++){
        if (maxv < x[i] - minv){
            maxv = x [i] - minv;
        }

        if (minv > x[i]){
            minv = x[i];
        }
    }
    return maxv;
}


-----------------------------------------------
FILE: ALDS1_10_A.c
-----------------------------------------------
/*
フィボナッチ数列
フィボナッチ数列の第
 項の値を出力するプログラムを作成してください。ここではフィボナッチ数列を以下の再帰的な式で定義します：



入力
１つの整数
 が与えられます。
```



```
出力
フィボナッチ数列の第
 項の値を出力してください。

制約
入力例 1
3
出力例 1
3
*/

#include <stdio.h>

long long fib[50];

long long fibonacci(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;

    if (fib[n] != 0) {
        return fib[n];
    }

    fib[n] = fibonacci(n - 1) + fibonacci(n - 2);
    return fib[n];
}

int main(void) {
    int n;
    scanf("%d", &n);

    printf("%lld\n", fibonacci(n));

    return 0;
}


-----------------------------------------------
FILE: ALDS1_10_B.c
-----------------------------------------------
/*
連鎖行列積
 個の行列の連鎖
 が与えられたとき、スカラー乗算の回数が最小になるように積
 の計算順序を決定する問題を連鎖行列積問題(Matrix-Chain Multiplication problem)と言います。

 個の行列について、行列
 の次元が与えられたとき、積
 の計算に必要なスカラー乗算の最小の回数を求めるプログラムを作成してください。

入力
入力の最初の行に、行列の数
 が与えられます。続く
 行で行列
 の次元が空白区切りの２つの整数
、
 で与えられます。
 は行列の行の数、
 は行列の列の数を表します。

出力
最小の回数を１行に出力してください。
```



```
制約
入力例 1
6
30 35
35 15
15 5
5 10
10 20
20 25
出力例 1
15125
*/

#include <stdio.h>
#include <limits.h>

#define MAX_N 100

int p[MAX_N + 1];
int m[MAX_N + 1][MAX_N + 1];

int matrixChain(int n) {
    for (int i = 1; i <= n; i++) {
        m[i][i] = 0;
    }

    for (int l = 2; l <= n; l++) {
        for (int i = 1; i <= n - l + 1; i++) {
            int j = i + l - 1;
            m[i][j] = INT_MAX;
            for (int k = i; k < j; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                }
            }
        }
    }

    return m[1][n];
}

int main(void) {
    int n;
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        int r, c;
        scanf("%d %d", &r, &c);
        if (i == 0) {
            p[0] = r;
        }
        p[i + 1] = c;
    }

    printf("%d\n", matrixChain(n));

    return 0;
}
```



```
-----------------------------------------------
FILE: ALDS1_10_C.c
-----------------------------------------------
/*
最長共通部分列
最長共通部分列問題 (Longest Common Subsequence problem: LCS)は、２つの与えられた列
 と
 の最長共通部分列を求める問題です。

ある列
 が
 と
 両方の部分列であるとき、
 を
 と
 の共通部分列と言います。例えば、
,
 とすると、列
 は
 と
 の共通部分列です。一方、列
 は
 と
 の最長共通部分列ではありません。なぜなら、その長さは 3 であり、長さ 4 の共通部分列
 が存在するからです。長さが 5 以上の共通部分列が存在しないので、列
 は
 と
 の最長共通部分列の１つです。

与えられた２つの文字列
、
に対して、最長共通部分列
 の長さを出力するプログラムを作成してください。与えられる文字列は英文字のみで構成されています。

入力
複数のデータセットが与えられます。最初の行にデータセットの数
 が与えられます。続く
 行にデータセットが与えられます。各データセットでは２つの文字列
,
 がそれぞれ１行に与えられます。

出力
各データセットについて
,
 の最長共通部分列
 の長さを１行に出力してください。

制約
の長さ
 または
 の長さが 100 を超えるデータセットが含まれる場合、
は 20 以下である。
入力例 1
3
abcbdab
bdcaba
abc
abc
abc
bc
出力例 1
4
```

```
3
2
参考文献
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
*/

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

#define MAX_LEN 1001

int dp[MAX_LEN][MAX_LEN];
char X[MAX_LEN];
char Y[MAX_LEN];

int max(int a, int b) {
    return (a > b) ? a : b;
}

int lcs(char *X, char *Y) {
    int m = strlen(X);
    int n = strlen(Y);

    for (int i = 0; i <= m; i++) {
        dp[i][0] = 0;
    }
    for (int j = 0; j <= n; j++) {
        dp[0][j] = 0;
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[m][n];
}

int main(void) {
    int q;
    scanf("%d", &q);

    for (int i = 0; i < q; i++) {
        scanf("%s", X);
        scanf("%s", Y);
        printf("%d\n", lcs(X, Y));
    }

    return 0;
}


----------------------------------------------
FILE: ALDS1_10_D.c
----------------------------------------------
/*
最適二分探索木
```


最適二分探索木とは、
個のキーと
個のダミーキーから、1回の探索にかかるコストの期待値が最小となるように構成された二分探索木のことです。

ソート済みの
個の異なるキーの列
から、二分探索木を構成することを考えます。 各キー
に対して探索が起きる確率は
です。 また、探索は
に含まれない値に対しても起こりうるので、
に含まれない値を表す
個のダミーキー
を用意します。 具体的には、ダミーキー
は以下のように定義されます。

なら、
は
よりも小さいすべての値を表す。
なら、
は
よりも大きいすべての値を表す。
なら、
は
よりも大きく、
よりも小さいすべての値を表す。
各ダミーキー
で探索が終わる確率は
であることもわかっています。 ここで、
と
に関して、以下の等式が成立します。
これらの情報より、ある二分探索木
における1回の探索コストの期待値は以下の式で得られます。ここで、
は
における節点
の深さです。
この値を最小化するように構成された二分探索木を、最適二分探索木と呼びます。

各キー
は内部節点、各ダミーキー
は葉になります。 例えば、入力例1の場合の最適二分探索木は以下のようになります。

課題
個の各キーに対して探索が起きる確率
と、
個の各ダミーキーで探索が終わる確率
が与えられるので、このときの最適二分探索木の1回の探索コストの期待値を出力するプログラムを作成してください。

入力

```
...

...
```

1行目にキーの個数を表す整数
が与えられます。
2行目に各キーに対して探索が起きる確率
が小数点以下4桁の実数で与えられます。
3行目に各ダミーキーで探索が終わる確率

が小数点以下4桁の実数で与えられます。

出力
最適二分探索木の探索コストの期待値を1行に出力してください。絶対誤差が
以下のとき正答と判定されます。

制約
入力例 1
```
5
0.1500 0.1000 0.0500 0.1000 0.2000
0.0500 0.1000 0.0500 0.0500 0.0500 0.1000
```
出力例 1
```
2.75000000
```
入力例 2
```
7
0.0400 0.0600 0.0800 0.0200 0.1000 0.1200 0.1400
0.0600 0.0600 0.0600 0.0600 0.0500 0.0500 0.0500 0.0500
```
出力例 2
```
3.12000000
*/

#include <stdio.h>
#include <float.h>

#define MAX_N 101

double p[MAX_N];
double q[MAX_N];
double e[MAX_N][MAX_N];
double w[MAX_N][MAX_N];

double min(double a, double b) {
    return (a < b) ? a : b;
}

double optimalBST(int n) {
    for (int i = 1; i <= n + 1; i++) {
        e[i][i - 1] = q[i - 1];
        w[i][i - 1] = q[i - 1];
    }

    for (int l = 1; l <= n; l++) {
        for (int i = 1; i <= n - l + 1; i++) {
            int j = i + l - 1;
            e[i][j] = DBL_MAX;
            w[i][j] = w[i][j - 1] + p[j] + q[j];

            for (int r = i; r <= j; r++) {
                double t = e[i][r - 1] + e[r + 1][j] + w[i][j];
                if (t < e[i][j]) {
                    e[i][j] = t;
                }
            }
        }
    }

    return e[1][n];
}

int main(void) {
    int n;
    scanf("%d", &n);
```



```
    for (int i = 1; i <= n; i++) {
        if (scanf("%lf", &p[i]) != 1) return 1;
    }

    for (int i = 0; i <= n; i++) {
        if (scanf("%lf", &q[i]) != 1) return 1;
    }

    double result = optimalBST(n);
    printf("%.8lf\n", result);

    return 0;
}


----------------------------------------------
FILE: ALDS1_11_A.c
----------------------------------------------
/*
グラフ
グラフ
の表現方法には隣接リスト(adjacency list) による表現と隣接行列(adjacency matrices)による表現があります。

隣接リスト表現では、
の各頂点に対して 1 個、合計
個のリスト
でグラフを表します。頂点
に対して、隣接リスト
は
に属する辺
におけるすべての頂点
を含んでいます。つまり、
は
において
と隣接するすべての頂点からなります。

一方、隣接行列表現では、頂点
から頂点
へ辺がある場合
が 1、ない場合 0 であるような
の行列
でグラフを表します。

隣接リスト表現の形式で与えられた有向グラフ
の隣接行列を出力するプログラムを作成してください。
は
個の頂点を含み、それぞれ
から
までの番号がふられているものとします。

入力
最初の行に
の頂点数
が与えられます。続く
行で各頂点
の隣接リスト
が以下の形式で与えられます：
```

```
  ...

  は頂点の番号、
  は
  の出次数、
   は
  に隣接する頂点の番号を示します。

  出力
  出力例に従い、
   の隣接行列を出力してください。
   の間に 1 つの空白を入れてください。

  制約
  入力例
  4
  1 2 2 4
  2 1 4
  3 0
  4 1 3
  出力例
  0 1 0 1
  0 0 0 1
  0 0 0 0
  0 0 1 0
  */

  /*
  グラフの表現方法の解説

  【隣接リスト（Adjacency List）】
  - 各頂点ごとに、隣接する頂点のリストを保持
  - メモリ効率が良い（辺の数に比例）
  - 全体の辺を調べるときに効率的

  例: 有向グラフで 1→2, 1→4 の場合
    隣接リスト[1] = [2, 4]

  【隣接行列（Adjacency Matrix）】
  - n×n の行列 A で、A[i][j] = 1 なら i→j の辺がある
  - 2つの頂点が辺でつながっているか高速に判定可能
  - メモリ効率は悪い（O(n²)）

  【問題の要点】
  隣接リスト形式の入力を隣接行列に変換

  入力形式:
  u k v1 v2 ... vk
    → u: 頂点番号
    → k: 出次数（隣接する頂点の個数）
    → v1, v2, ..., vk: 隣接頂点のリスト

  【具体例】
  入力:
  4
  1 2 2 4     ← 頂点1から頂点2,4へ辺がある
  2 1 4       ← 頂点2から頂点4へ辺がある
  3 0         ← 頂点3から出ていく辺なし
  4 1 3       ← 頂点4から頂点3へ辺がある

  隣接行列:
      1  2  3  4
```



```
  1 [ 0  1  0  1 ]  ← 1→2, 1→4
  2 [ 0  0  0  1 ]  ← 2→4
  3 [ 0  0  0  0 ]  ← 出ていく辺なし
  4 [ 0  0  1  0 ]  ← 4→3

  【実装ステップ】
  1. n×n の隣接行列を 0 で初期化
  2. n 行分の隣接リスト情報を読み込み
  3. 各隣接頂点について A[u][v] = 1 を設定
  4. 行列を出力

  【計算量】
  入力: O(V + E) – V個の頂点、E個の辺
  行列出力: O(V²)
  総合: O(V + E + V²)
  */

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

  #define MAX_V 101

  int main(void) {
      int n;
      scanf("%d", &n);

      // 隣接行列を 0 で初期化
      int Adj[MAX_V][MAX_V];
      memset(Adj, 0, sizeof(Adj));

      // 隣接リスト形式の入力を読み込む
      for (int i = 0; i < n; i++) {
          int u, k;
          scanf("%d %d", &u, &k);

          // 頂点 u に隣接する k 個の頂点を読み込む
          for (int j = 0; j < k; j++) {
              int v;
              scanf("%d", &v);
              // 辺 u→v が存在することを記録
              Adj[u][v] = 1;
          }
      }

      // 隣接行列を出力
      for (int i = 1; i <= n; i++) {
          for (int j = 1; j <= n; j++) {
              if (j > 1) printf(" ");
              printf("%d", Adj[i][j]);
          }
          printf("\n");
      }

      return 0;
  }


  ----------------------------------------
  FILE: ALDS1_11_B.c
  ----------------------------------------
  /*
  深さ優先探索
```


深さ優先探索（Depth First Search: DFS）は、可能な限り隣接する頂点を訪問する、という戦略に基づくグラフの探索アルゴリズムです。未探索の接続辺が残されている頂点の中で最後に発見した頂点
の接続辺を再帰的に探索します。
の辺をすべて探索し終えると、
を発見したときに通ってきた辺を後戻りして探索を続行します。

探索は元の始点から到達可能なすべての頂点を発見するまで続き、未発見の頂点が残っていれば、その中の番号が一番小さい 1 つを新たな始点として探索を続けます。

深さ優先探索では、各頂点に以下のタイムスタンプを押します：

タイムスタンプ
:
を最初に発見した発見時刻を記録します。
タイムスタンプ
:
の隣接リストを調べ終えた完了時刻を記録します。
以下の仕様に基づき、与えられた有向グラフ
に対する深さ優先探索の動作を示すプログラムを作成してください：

は隣接リスト表現の形式で与えられます。各頂点には
から
までの番号がふられています。
各隣接リストの頂点は番号が小さい順に並べられています。
プログラムは各頂点の発見時刻と完了時刻を報告します。
深さ優先探索の過程において、訪問する頂点の候補が複数ある場合は頂点番号が小さいものから選択します。
最初に訪問する頂点の開始時刻を 1 とします。
入力
最初の行に
の頂点数
が与えられます。続く
行で各頂点
の隣接リストが以下の形式で与えられます：

```
  ...

  は頂点の番号、
  は
  の出次数、
    は
  に隣接する頂点の番号を示します。

  出力
  各頂点について
  、
  、
  を空白区切りで 1 行に出力してください。
   は頂点の番号、
   はその頂点の発見時刻、
   はその頂点の完了時刻です。頂点の番号順で出力してください。

  制約
  入力例 1
  4
  1 1 2
  2 1 4
  3 0
  4 1 3
  出力例 1
  1 1 8
  2 2 7
```



```
  3 4 5
  4 3 6

  入力例 2
  6
  1 2 2 3
  2 2 3 4
  3 1 5
  4 1 6
  5 1 6
  6 0
  出力例 2
  1 1 12
  2 2 11
  3 3 8
  4 9 10
  5 4 7
  6 5 6

  入力例2に対応するグラフ(発見時刻/終了時刻)


  参考文献
  Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
  */

  /*
  深さ優先探索（DFS）の解説

  【DFSとは】
  グラフの全ての頂点を訪問するアルゴリズム
  「できるだけ深く」進みながら探索

  【タイムスタンプの意味】
  d[u]: 頂点uを最初に発見した時刻
  f[u]: 頂点uの全ての隣接頂点を調べ終えた時刻

  【DFSの流れ】

  例: 頂点1 → 2 → 4 → 3 → （戻る）
     1–1:発見, 2–2:発見, 4–3:発見, 3–4:発見, 3–5:完了, 4–6:完了, 2–7:完了, 1–8:完了

  タイムスタンプの付与：
  - 頂点を訪問した時に d[u] = ++timer
  - 頂点の全隣接頂点を探索し終えたとき f[u] = ++timer

  【具体例1】
  グラフ:
  1 → 2 → 4 → 3 (→はなし)

  DFS実行:
  timer=0

  訪問(1): d[1]=1, timer=1
    訪問(2): d[2]=2, timer=2
      訪問(4): d[4]=3, timer=3
        訪問(3): d[3]=4, timer=4
          (3から出ていく辺なし)
        完了(3): f[3]=5, timer=5
      完了(4): f[4]=6, timer=6
    完了(2): f[2]=7, timer=7
  完了(1): f[1]=8, timer=8
```

```
結果:
1 1 8
2 2 7
3 4 5
4 3 6

【具体例2】
グラフが複数のコンポーネントに分かれている場合：
- 1つ目のコンポーネント: 1–2–3–5–6, 4–6
- timer は続きの値から開始

訪問順序: 1(d=1) → 2(d=2) → 3(d=3) → 5(d=4) → 6(d=5)
完了順序: 6(f=6) → 5(f=7) → 3(f=8) → 2(f=11)
次のコンポーネント: 4(d=9) → 6はすでに訪問済み
完了: 4(f=10)
完了: 1(f=12)

【計算量】
時間: O(V + E) – 全頂点と全辺を訪問
空間: O(V) – 再帰スタックと配列

【実装のポイント】
1. グラフを隣接行列で表現
2. visited配列で訪問済みかどうかを判定
3. d, f配列でタイムスタンプを記録
4. 未訪問の頂点から順番に DFS を開始
*/

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

#define MAX_V 101

int timer = 0;
bool visited[MAX_V];
int d[MAX_V];  // 発見時刻
int f[MAX_V];  // 完了時刻
int Adj[MAX_V][MAX_V];

// 深さ優先探索
void dfs(int u, int n) {
    visited[u] = true;
    d[u] = ++timer;  // 発見時刻を記録

    // u から隣接する全ての頂点を探索
    for (int v = 1; v <= n; v++) {
        if (Adj[u][v] == 1 && !visited[v]) {
            dfs(v, n);
        }
    }

    f[u] = ++timer;  // 完了時刻を記録
}

int main(void) {
    int n;
    scanf("%d", &n);

    // 隣接行列と訪問済み配列を初期化
    memset(Adj, 0, sizeof(Adj));
```



```
    memset(visited, false, sizeof(visited));

    // 隣接リスト形式の入力を読み込む
    for (int i = 0; i < n; i++) {
        int u, k;
        scanf("%d %d", &u, &k);

        for (int j = 0; j < k; j++) {
            int v;
            scanf("%d", &v);
            Adj[u][v] = 1;
        }
    }

    // 各頂点から DFS を開始（未訪問の場合のみ）
    for (int i = 1; i <= n; i++) {
        if (!visited[i]) {
            dfs(i, n);
        }
    }

    // 結果を出力
    for (int i = 1; i <= n; i++) {
        printf("%d %d %d\n", i, d[i], f[i]);
    }

    return 0;
}


------------------------------------------------
FILE: ALDS1_11_C.c
------------------------------------------------
/*
幅優先探索の解説

【BFSとは】
グラフの全頂点を訪問し、始点から各頂点への最短距離を求めるアルゴリズム
「できるだけ浅く」層ごとに探索（DFSとの対比）

【最短距離とは】
パス上の辺の本数の最小値

【キューを使う理由】
BFSではキュー（FIFO）を使用：
1. 始点をキューに入れる
2. キューから頂点uを取り出す
3. uから隣接する未訪問の頂点をキューに追加
4. 距離をインクリメント
5. キューが空になるまで繰り返す

DFSではスタック（LIFO）を使うのに対し、
BFSではキュー（FIFO）を使うことで層ごとの処理が実現できる

【具体例】
グラフ: 1–2, 1–4, 2–4, 4–3

開始頂点: 1

■初期化
dist[1]=0（始点の距離）
dist[2]=dist[3]=dist[4]=−1（未訪問）
```



```
キュー: [1]

■ステップ1: u=1 を取り出す
  隣接: 2, 4
  訪問(2): dist[2]=0+1=1, キュー: [2, 4]
  訪問(4): dist[4]=0+1=1, キュー: [2, 4]

■ステップ2: u=2 を取り出す
  隣接: 4
  4は既に訪問済み, キュー: [4]

■ステップ3: u=4 を取り出す
  隣接: 3
  訪問(3): dist[3]=1+1=2, キュー: [3]

■ステップ4: u=3 を取り出す
  隣接: なし, キュー: []

結果:
1: 距離 0
2: 距離 1
3: 距離 2
4: 距離 1

【計算量】
時間: O(V + E) – 各頂点と各辺を1回訪問
空間: O(V) – キューと距離配列

【実装のポイント】
1. キューで層ごとの処理を実現
2. 訪問済みかどうかを距離配列で判定（−1=未訪問）
3. 隣接する全未訪問頂点を同じ距離でキューに追加
4. 到達不可能な頂点は−1で出力
*/

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

#define MAX_V 101
#define QUEUE_SIZE 10001

int dist[MAX_V];      // 距離配列
int Adj[MAX_V][MAX_V]; // 隣接行列
int queue[QUEUE_SIZE]; // キュー
int queue_front = 0;
int queue_rear = 0;

// キューに要素を追加
void enqueue(int x) {
    queue[queue_rear++] = x;
}

// キューから要素を取り出す
int dequeue(void) {
    return queue[queue_front++];
}

// キューが空かどうかを判定
int is_empty(void) {
    return queue_front >= queue_rear;
}
```



```
// 幅優先探索
void bfs(int start, int n) {
    // 距離の初期化（−1: 未訪問）
    for (int i = 1; i <= n; i++) {
        dist[i] = -1;
    }

    dist[start] = 0;  // 始点の距離は0
    enqueue(start);

    while (!is_empty()) {
        int u = dequeue();

        // u から隣接する全ての頂点を探索
        for (int v = 1; v <= n; v++) {
            if (Adj[u][v] == 1 && dist[v] == -1) {
                dist[v] = dist[u] + 1;
                enqueue(v);
            }
        }
    }
}

int main(void) {
    int n;
    scanf("%d", &n);

    // 隣接行列を初期化
    memset(Adj, 0, sizeof(Adj));

    // 隣接リスト形式の入力を読み込む
    for (int i = 0; i < n; i++) {
        int u, k;
        scanf("%d %d", &u, &k);

        for (int j = 0; j < k; j++) {
            int v;
            scanf("%d", &v);
            Adj[u][v] = 1;
        }
    }

    // 頂点1から幅優先探索を開始
    bfs(1, n);

    // 結果を出力
    for (int i = 1; i <= n; i++) {
        printf("%d %d\n", i, dist[i]);
    }

    return 0;
}


------------------------------------------------
FILE: ALDS1_11_D.c
------------------------------------------------
/*
連結成分分解
SNS の友達関係を入力し、双方向の友達リンクを経由してある人からある人へたどりつけるかどうかを判定するプログラムを作成してください。

入力
```

```
1行目にSNS のユーザ数を表す整数
と友達関係の数
が空白区切りで与えられます。SNS の各ユーザには 0 から
までのID が割り当てられています。

続く
行に 1 つの友達関係が各行に与えられます。 1 つの友達関係は空白で区切られた 2 つの整数
`
`で与えられ、
と
が友達であることを示します。

続く 1 行に、質問の数
が与えられます。続く
行に質問が与えられます。 各質問は空白で区切られた 2 つの整数
`
`で与えられ、「
から
へたどり着けますか？」という質問を意味します。

出力
各質問に対して
から
にたどり着ける場合は yes と、たどり着けない場合は no と 1 行に出力してください。

制約
入力例
10 9
0 1
0 2
3 4
5 7
5 6
6 7
6 8
7 8
8 9
3
0 1
5 9
1 3
出力例
yes
yes
no
*/

/*
連結成分分解の解説

【問題の本質】
SNSの友達関係をグラフで表現
→ 2人が同じ連結成分に属しているかを高速に判定

【Union-Find（素集合データ構造）とは】
複数のグループ（連結成分）を管理するデータ構造

操作:
1. find(x): xが属するグループの代表元を返す
2. union(x, y): xとyを同じグループに統合

【Union-Findの原理】
```



```
親ポインタ配列 parent[] を使用：
- parent[i] = i の親ノード番号
- parent[x] = x の場合、xが根（代表元）

例: 0–1, 0–2, 3–4 の友達関係

初期状態:
parent = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

union(0, 1):
  root(0)=0, root(1)=1
  parent[1] = 0
  parent = [0, 0, 2, 3, 4, 5, 6, 7, 8, 9]

union(0, 2):
  root(0)=0, root(2)=2
  parent[2] = 0
  parent = [0, 0, 0, 3, 4, 5, 6, 7, 8, 9]

union(3, 4):
  root(3)=3, root(4)=4
  parent[4] = 3
  parent = [0, 0, 0, 3, 3, 5, 6, 7, 8, 9]

質問 find(0) == find(1):
  root(0) = 0, root(1) = 0
  → 同じグループ → yes

質問 find(1) == find(3):
  root(1) = 0, root(3) = 3
  → 異なるグループ → no

【最適化: 経路圧縮（Path Compression）】
find(x) を呼び出すとき、パス上の全ノードを直接根に繋ぎ直す
→ 次回以降の find が高速化

【計算量（経路圧縮なし）】
find: O(n) – 最悪の場合、チェーン全体を走査
union: O(n) – find を2回呼び出し

【計算量（経路圧縮あり）】
find: ほぼ O(1) – 平均的に高速
union: ほぼ O(1) – 平均的に高速

【具体例: 10ユーザー、9つの友達関係】

入力:
0–1, 0–2, 3–4, 5–7, 5–6, 6–7, 6–8, 7–8, 8–9

処理:
union(0,1) → {0,1}
union(0,2) → {0,1,2}
union(3,4) → {3,4}
union(5,7) → {5,7}
union(5,6) → {5,6,7}
union(6,7) → {5,6,7}（既に同じグループ）
union(6,8) → {5,6,7,8}
union(7,8) → {5,6,7,8}（既に同じグループ）
union(8,9) → {5,6,7,8,9}

連結成分: {0,1,2}, {3,4}, {5,6,7,8,9}
```



```
質問:
0と1: find(0)==find(1) → yes
5と9: find(5)==find(9) → yes
1と3: find(1)==find(3) → no
*/

#include <stdio.h>

#define MAX_N 100001

int parent[MAX_N];

// 代表元（根）を見つける関数
// 経路圧縮を使用して最適化
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]);  // 経路圧縮
    }
    return parent[x];
}

// 2つの要素を同じグループに統合
void unite(int x, int y) {
    int root_x = find(x);
    int root_y = find(y);

    if (root_x != root_y) {
        parent[root_x] = root_y;  // xの根をyの根に繋ぎ替え
    }
}

int main(void) {
    int n, m;
    scanf("%d %d", &n, &m);

    // 初期化: 各要素が独立したグループ
    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }

    // 友達関係を処理（グループ化）
    for (int i = 0; i < m; i++) {
        int x, y;
        scanf("%d %d", &x, &y);
        unite(x, y);
    }

    // 質問に答える
    int q;
    scanf("%d", &q);

    for (int i = 0; i < q; i++) {
        int x, y;
        scanf("%d %d", &x, &y);

        if (find(x) == find(y)) {
            printf("yes\n");
        } else {
            printf("no\n");
        }
    }
```



```
    return 0;
}


------------------------------------------------
FILE: ALDS1_12_A.c
------------------------------------------------
/*
最小全域木（Minimum Spanning Tree）– プリム法

最小全域木とは、重み付き無向グラフの全頂点を含む木のうち、
辺の重みの総和が最小となるものです。

プリム法のアルゴリズム:
1. 始点を選び、その頂点を訪問済みとする
2. 訪問済み頂点から未訪問頂点への辺のうち、最小コストの辺を選ぶ
3. その辺で繋がる未訪問頂点を訪問済みにし、辺のコストを総和に加える
4. 全頂点が訪問済みになるまで2–3を繰り返す

時間計算量: O(V^2)（隣接行列使用時）
*/

#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define MAX_N 100

int Weight[MAX_N+1][MAX_N+1];  // 隣接行列（重み）
int minCost[MAX_N+1];          // 各頂点への最小コスト
bool visited[MAX_N+1];         // 訪問済みフラグ

// プリム法による最小全域木のコスト計算
int prim(int n) {
    int totalCost = 0;

    // 初期化：全頂点の最小コストを無限大に設定
    for (int i = 1; i <= n; i++) {
        minCost[i] = INT_MAX;
        visited[i] = false;
    }

    // 始点（頂点1）のコストを0に設定
    minCost[1] = 0;

    // n個の頂点を順番に木に追加
    for (int i = 0; i < n; i++) {
        int u = -1;
        int minWeight = INT_MAX;

        // 未訪問の頂点の中で最小コストの頂点を選択
        for (int v = 1; v <= n; v++) {
            if (!visited[v] && minCost[v] < minWeight) {
                minWeight = minCost[v];
                u = v;
            }
        }

        // 選択した頂点を訪問済みにし、コストを加算
        visited[u] = true;
        totalCost += minWeight;
```

```
        // 選択した頂点から到達可能な未訪問頂点の最小コストを更新
        for (int v = 1; v <= n; v++) {
            if (!visited[v] && Weight[u][v] != -1 && Weight[u][v] < minCost[v]) {
                minCost[v] = Weight[u][v];
            }
        }
    }

    return totalCost;
}

int main() {
    int n;
    scanf("%d", &n);

    // 隣接行列の入力（-1は辺が存在しないことを表す）
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            scanf("%d", &Weight[i][j]);
        }
    }

    // 最小全域木のコストを計算して出力
    int result = prim(n);
    printf("%d\n", result);

    return 0;
}



----------------------------------------
FILE: ALDS1_12_B.c
----------------------------------------
/*
単一始点最短経路 I – ダイクストラ法

ダイクストラ法は、重み付き有向グラフにおいて、
始点から各頂点への最短経路を求めるアルゴリズムです。

アルゴリズムの流れ:
1. 始点の距離を0、他の頂点の距離を無限大(未確定)に初期化
2. 未訪問の頂点の中で、距離が最小の頂点uを選択
3. uを訪問済みにする
4. uから到達可能な各頂点vについて、距離を更新（緩和操作）
   dist[v] = min(dist[v], dist[u] + weight(u,v))
5. 全頂点が訪問済みになるまで2-4を繰り返す

時間計算量: O(V^2)（優先度付きキューなしの実装）
空間計算量: O(V^2)（隣接行列使用）

注意: このアルゴリズムは辺の重みが非負の場合のみ正しく動作します
*/

#include <stdio.h>
#include <stdbool.h>

#define MAX_N 100
#define INF -1  // 到達不可能を表す

int adj[MAX_N][MAX_N];  // 隣接行列（0は辺なし）
```



```
int dist[MAX_N];          // 始点からの最短距離
bool visited[MAX_N];      // 訪問済みフラグ

// ダイクストラ法による最短経路の計算
void dijkstra(int n) {
    // 初期化
    for (int i = 0; i < n; i++) {
        dist[i] = INF;        // 初期状態では到達不可能
        visited[i] = false;  // 未訪問
    }

    // 始点（頂点0）の距離を0に設定
    dist[0] = 0;

    // n回のループで全頂点を処理
    for (int i = 0; i < n; i++) {
        int u = -1;
        int minDist = INF;

        // 未訪問かつ距離が確定している頂点の中で最小距離の頂点を選択
        for (int v = 0; v < n; v++) {
            if (!visited[v] && dist[v] != INF) {
                if (u == -1 || dist[v] < minDist) {
                    minDist = dist[v];
                    u = v;
                }
            }
        }

        // 到達可能な未訪問頂点がなければ終了
        if (u == -1) break;

        // 選択した頂点を訪問済みにする
        visited[u] = true;

        // uから到達可能な頂点の距離を更新（緩和操作）
        for (int v = 0; v < n; v++) {
            if (adj[u][v] != 0) {  // 辺が存在する場合
                int newDist = dist[u] + adj[u][v];
                // より短い経路が見つかった場合、距離を更新
                if (dist[v] == INF || newDist < dist[v]) {
                    dist[v] = newDist;
                }
            }
        }
    }

    // 結果の出力
    for (int i = 0; i < n; i++) {
        printf("%d %d\n", i, dist[i]);
    }
}

int main() {
    int n;
    scanf("%d", &n);

    // 隣接行列の初期化（0で初期化）
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            adj[i][j] = 0;
        }
```



```
    }

    // グラフの入力（隣接リスト形式）
    for (int i = 0; i < n; i++) {
        int u, k;
        scanf("%d %d", &u, &k);  // 頂点番号uと隣接頂点の数k

        // k個の隣接頂点とその辺の重みを入力
        for (int j = 0; j < k; j++) {
            int v, c;
            scanf("%d %d", &v, &c);  // 隣接頂点vと辺の重みc
            adj[u][v] = c;
        }
    }

    // ダイクストラ法の実行
    dijkstra(n);

    return 0;
}



----------------------------------------
FILE: ALDS1_12_C.c
----------------------------------------
/*
単一始点最短経路 II
与えられた重み付き有向グラフ
 について、単一始点最短経路のコストを求めるプログラムを作成してください。
 の頂点
 を始点とし、
 から各頂点
 について、最短経路上の辺の重みの総和
 を出力してください。

入力
最初の行に
 の頂点数
 が与えられます。続く
 行で各頂点
 の隣接リストが以下の形式で与えられます：




 ...

 の各頂点には
 から
 の番号がふられています。
 は頂点の番号であり、
 は
 の出次数を示します。
 は
 に隣接する頂点の番号であり、
 は
 と
 をつなぐ有向辺の重みを示します。
```



```
出力
各頂点の番号
 と距離
 を１つの空白区切りで順番に出力してください。

制約
0 から各ノードへは必ず経路が存在する
入力例 1
5
0 3 2 3 3 1 1 2
1 2 0 2 3 4
2 3 0 3 3 1 4 1
3 4 2 1 0 1 1 4 4 3
4 2 2 1 3 3
出力例 1
0 0
1 2
2 2
3 1
4 3

参考文献
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
*/

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

#define QMAX 35000
typedef struct { int t, s; } QUE;
QUE que[QMAX];
int qsize;

#define PARENT(i) ((i)>>1)
#define LEFT(i)   ((i)<<1)
#define RIGHT(i)  (((i)<<1)+1)

void min_heapify(int i) {
    int l, r, min;
    QUE qt;

    l = LEFT(i), r = RIGHT(i);
    if (l < qsize && que[l].t < que[i].t) min = l; else min = i;
    if (r < qsize && que[r].t < que[min].t) min = r;
    if (min != i) {
        qt = que[i], que[i] = que[min], que[min] = qt;
        min_heapify(min);
    }
}

void deq() {
    que[0] = que[--qsize];
    min_heapify(0);
}

void enq(int s, int t) {
    int i, min;
    QUE qt;

    i = qsize++;
```

```
    que[i].s = s, que[i].t = t;
    while (i > 0 && que[min = PARENT(i)].t > que[i].t) {
        qt = que[i], que[i] = que[min], que[min] = qt;
        i = min;
    }
}

#define INF   0x30303030
#define MAX_V 10002
int hi[MAX_V], *to[MAX_V], *dd[MAX_V];
int dist[MAX_V];

void dijkstra(int n, int start) {
    int i, s, e, t, nt;

    memset(dist, INF, n << 2);
    qsize = 0;
    dist[start] = 0, enq(start, 0);
    while (qsize) {
        s = que[0].s, t = que[0].t, deq();
        if (dist[s] < t) continue;
        for (i = 0; i < hi[s]; i++) {
            e = to[s][i], nt = t + dd[s][i];
            if (dist[e] > nt) dist[e] = nt, enq(e, nt);
        }
    }
}

#if 1
#define gc() getchar_unlocked()
#define pc(c) putchar_unlocked(c)
#else
#define gc() getchar()
#define pc(c) putchar(c)
#endif

int in() {
    int n = 0, c = gc();
    do n = 10*n + (c & 0xf), c = gc(); while (c >= '0');
    return n;
}

void out(int n) {
    int i;
    char ob[20];

    if (!n) { pc('0'); return; }
    i = 0; while (n) ob[i++] = n%10 + '0', n/=10;
    while (i--) pc(ob[i]);
}

int main() {
    int n, i, j, k, u;

    n = in();
    for (i = 0; i < n; i++) {
        u = in(), hi[u] = k = in();
        to[u] = malloc(k<<2);
        dd[u] = malloc(k<<2);
        for (j = 0; j < k; j++) {
            to[u][j] = in(), dd[u][j] = in();
        }
```



```
    }
    dijkstra(n, 0);
    for (i = 0; i < n; i++) {
        out(i), pc(' '), out(dist[i]), pc('\n');
    }
    return 0;
}


----------------------------------------
FILE: ALDS1_13_A.c
----------------------------------------
/*
8クイーン問題
8 クイーン問題とは、
 のマスから成るチェス盤に、どのクイーンも他のクイーンを襲撃できないように、8 つのクイーンを配置する問題です。チェスでは、クイーンは次のように 8 方向のマスにいるコマを襲撃することができます。



すでにクイーンが配置されている
 個のマスが指定されるので、それらを合わせて 8 つのクイーンを配置したチェス盤を出力するプログラムを作成してください。

入力
1 行目に整数
 が与えられます。続く
 行にクイーンが配置されているマスが 2 つの整数

,
 で与えられます。

,
 はそれぞれ 0 から始まるチェス盤の行と列の番号を表します。

出力
出力は
 のチェス盤を表す文字列で、クイーンが配置されたマスを 'Q'、配置されていないマスを '.' で表します。

制約
入力に対する解はただ 1 つ存在する。
入力例 1
2
2 2
5 3
出力例 1
......Q.
Q.......
..Q.....
.......Q
.....Q..
...Q....
.Q......
....Q...
*/

#include <stdio.h>

char q[8];
char a[8], b[8], u[16], v[16];

void found()
{
    int r, c;
```



```
    for (r = 0; r < 8; r++) {
        for (c = 0; c < 8; c++) {
            if (q[r] == c) putchar('Q');
            else           putchar('.');
        }
        putchar('\n');
    }
}

int check(int r)
{
    int c;

    if (r == 8) { found(); return 1; }
    if (b[r]) { if (check(r+1)) return 1; }
    for (c = 0; c < 8; c++) {
        if (b[r] || a[c] || u[r+c] || v[r-c+7]) continue;
        a[c] = u[r+c] = v[r-c+7] = 1;
        q[r] = c;
        if (check(r+1)) return 1;
        a[c] = u[r+c] = v[r-c+7] = 0;
    }
    return 0;
}

int main()
{
    int k, r, c;

    scanf("%d", &k);
    while (k--) {
        scanf("%d%d", &r, &c);
        q[r] = c;
        a[c] = b[r] = u[r+c] = v[r-c+7] = 1;
    }
    check(0);
    return 0;
}


----------------------------------------
FILE: ALDS1_13_B.c
----------------------------------------
/*
8 パズル
8 パズルは 1 つの空白を含む
 のマス上に 8 枚のパネルが配置され、空白を使ってパネルを上下左右にスライドさせ、絵柄を揃えるパズルです。

この問題では、次のように空白を 0、各パネルを 1 から 8 の番号でパズルを表します。

1 3 0
4 2 5
7 8 6
1 回の操作で空白の方向に 1 つのパネルを移動することができ、ゴールは次のようなパネルの配置とします。

1 2 3
4 5 6
7 8 0
8 パズルの初期状態が与えられるので、ゴールまでの最短手数を求めるプログラムを作成してください。

入力
入力はパネルの数字あるいは空白を表す
```



```
 個の整数です。空白で区切られた 3 つの整数が 3 行で与えられます。

出力
最短手数を 1 行に出力してください。

制約
与えられるパズルは必ず解くことができる。
入力例
1 3 0
4 2 5
7 8 6
出力例
4
*/

#include <stdio.h>

typedef unsigned char uchar;

#define HASHSIZ 543217
typedef struct { unsigned s; char x, v; } HASH;
HASH hash[HASHSIZ+3], *hashend = hash+HASHSIZ;

int lookup(unsigned s, int x)
{
    long long t = ((long long)x << 32) + s;
    HASH *hp = hash + (int)(t % HASHSIZ);
    while (hp->s) {
        if (hp->s == s && hp->x == x) return hp->v;
        if (++hp == hashend) hp = hash;
    }
    return 0;
}

void insert(unsigned s, int x, char v)
{
    long long t = ((long long)x << 32) + s;
    HASH *hp = hash + (int)(t % HASHSIZ);
    while (hp->s) {
        if (hp->s == s && hp->x == x) { hp->v = v; break; }
        if (++hp == hashend) hp = hash;
    }
    hp->s = s, hp->x = x, hp->v = v;
}

#define QMAX 50000
typedef struct { unsigned s; char x, v; } T;
T s[QMAX+3]; int top, end;
int hi[9] = {2,3,2,3,4,3,2,3,2};
int to[9][4] = {{1,3},{0,2,4},{1,5},{0,4,6},{1,3,5,7},{2,4,8},{3,7},{4,6,8},{5,7}};

void toArray(char *nine, unsigned s, int x)
{
    int i;

    for (i = 8; i >= 0; i--) {
        if (i == x) nine[i] = 0;
        else nine[i] = s & 0xf, s >>= 4;
    }
}

unsigned swap(char *nine, int zero, int nx)
```

```
{
    int i;
    unsigned res;

    res = 0;
    for (i = 0; i < 9; i++) {
        if      (i == zero) res <<= 4, res |= nine[nx];
        else if (i != nx)   res <<= 4, res |= nine[i];
    }
    return res;
}

#define MAGIC 20

int main()
{
    int i, x, nx, v, nv, t;
    unsigned a, na;
    unsigned gs; int gx;
    char nine[9];

    gs = 0; for (i = 0; i < 9; i++) {
        scanf("%d", &t);
        if (t == 0) gx = i;
        else gs <<= 4, gs |= t;
    }
    s[0].s = 0x12345678, s[0].x = 8;
    s[1].x = -1;
    top = 0, end = 2;
    insert(s[0].s, s[0].x, 0);
    v = 0;
    while (top != end) {
        a = s[top].s, x = s[top].x;
        if (++top == QMAX) top = 0;
        if (x < 0) {
            if (v == MAGIC) break;
            v++;
            s[end].x = -1; if (++end == QMAX) end = 0;
            continue;
        }
        if (x == gx && a == gs) goto done;
        if (lookup(a, x) < v) continue;

        toArray(nine, a, x);
        for (i = 0; i < hi[x]; i++) {
            nx = to[x][i];
            na = swap(nine, x, nx), nv = v+1;
            t = lookup(na, nx);
            if (!t || nv < t) {
                s[end].s = na, s[end].x = nx;
                if (++end == QMAX) end = 0;
                insert(na, nx, nv);
            }
        }
    }

    s[0].s = gs, s[0].x = gx, s[1].x = -1;
    top = 0, end = 2;
    v = 0;
    while (top != end) {
        a = s[top].s, x = s[top].x;
        if (++top == QMAX) top = 0;
```



```
        if (x < 0) {
            v++;
            s[end].x = -1; if (++end == QMAX) end = 0;
            continue;
        }
        if ((t = lookup(a, x)) > 0) { v += t; break; }

        toArray(nine, a, x);
        for (i = 0; i < hi[x]; i++) {
            nx = to[x][i];
            na = swap(nine, x, nx), nv = v+1;
            t = lookup(na, nx);
            if (!t || nv < t) {
                s[end].s = na, s[end].x = nx;
                if (++end == QMAX) end = 0;
            }
        }
    }
done:
    printf("%d\n", v);
    return 0;
}


-----------------------------------------------
FILE: ALDS1_13_C.c
-----------------------------------------------
/*
1 5 パズル
15 パズルは 1 つの空白を含む
 のマス上に 15 枚のパネルが配置され、空白を使ってパネルを上下左右にスライドさせ、絵柄を揃えるパズルです。

この問題では、次のように空白を 0、各パネルを 1 から 15 の番号でパズルを表します。

1 2 3 4
6 7 8 0
5 10 11 12
9 13 14 15
1 回の操作で空白の方向に 1 つのパネルを移動することができ、ゴールは次のようなパネルの配置とします。

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
15 パズルの初期状態が与えられるので、ゴールまでの最短手数を求めるプログラムを作成してください。

入力
入力はパネルの数字あるいは空白を表す
 個の整数です。空白で区切られた 4 つの整数が 4 行で与えられます。

出力
最短手数を 1 行に出力してください。

制約
与えられるパズルは 45 手以内で解くことができる。
入力例
1 2 3 4
6 7 8 0
5 10 11 12
9 13 14 15
出力例
8
```



```
*/

#include <stdio.h>

typedef unsigned long long ull;

#define QMAX 100000
typedef struct { ull s; char x, v; char t; } QUE;
QUE que[QMAX]; int qsize;

#define PARENT(i) ((i)>>1)
#define LEFT(i)   ((i)<<1)
#define RIGHT(i)  (((i)<<1)+1)

void min_heapify(int i)
{
    int l, r;
    int min;

    l = LEFT(i), r = RIGHT(i);
    if (l < qsize && que[l].t < que[i].t) min = l; else min = i;
    if (r < qsize && que[r].t < que[min].t) min = r;
    if (min != i) {
        QUE t = que[i]; que[i] = que[min]; que[min] = t;
        min_heapify(min);
    }
}

void deq()
{
  que[0] = que[--qsize];
  min_heapify(0);
}

void enq(ull s, int x, int v, int t)
{
    int i, min;
    QUE qt;

    i = qsize++;
    que[i].s = s, que[i].x = x, que[i].v = v, que[i].t = t;
    while (i > 0 && que[min = PARENT(i)].t > que[i].t) {
        qt = que[i]; que[i] = que[min]; que[min] = qt;
        i = min;
    }
}

#define HASHSIZ 200003
typedef struct { ull s; char v; } HASH;
HASH hash [HASHSIZ+3], *hashend  = hash +HASHSIZ;

int lookup(ull s)
{
    HASH *hp = hash + (int)(s % HASHSIZ);
    while (hp->s) {
        if (hp->s == s) return hp->v;
        if (++hp == hashend) hp = hash;
    }
    return -1;
}
```



```
void insert(ull s, char v)
{
    HASH *hp = hash + (int)(s % HASHSIZ);
    while (hp->s) {
        if (hp->s == s) { hp->v = v; break; }
        if (++hp == hashend) hp = hash;
    }
    hp->s = s, hp->v = v;
}

#define GOAL 0x123456789ABCDEF0LL
#define ABS(a) ((a)>=0?(a):-(a))

int hi[16] = {2,3,3,2,3,4,4,3,3,4,4,3,2,3,3,2};
int to[16][4] = {
{1,4},{0,2,5},{1,3,6},{2,7},
{0,5,8},{1,4,6,9},{2,5,7,10},{3,6,11},
{4,9,12},{5,8,10,13},{6,9,11,14},{7,10,15},
{8,13},{9,12,14},{10,13,15},{11,14}};
double magic = 1;

void toArray(char *p16, ull s)
{
    int i;

    for (i = 15; i >= 0; i--) {
        p16[i] = s & 0xf, s >>= 4;
    }
}

ull swap(char *p16, int zero, int nx)
{
    int i;
    ull res;

    res = 0;
    for (i = 0; i < 16; i++) {
        res <<= 4;
        if      (i == zero) res |= p16[nx];
        else if (i == nx)   res |= p16[zero];
        else                res |= p16[i];
    }
    return res;
}

int eval(ull s)
{
    int i, r1, c1, r2, c2;
    int e, t;

    e = 0;
    for (i = 15; i >= 0; i--) {
        t = s & 0xf, s >>= 4;
        if (t) {
            t--;
            r1 = i >> 2, c1 = i & 3;
            r2 = t >> 2, c2 = t & 3;
            e += ABS(r1-r2) + ABS(c1-c2);
        }
    }
    return (int)(e*magic);
```

```
}

int main()
{
    int i, x, nx, v, nv, t;
    ull s, ns;
    char p16[16];

    s = 0; for (i = 0; i < 16; i++) {
        scanf("%d", &t);
        p16[i] = t, s <<= 4, s |= t;
        if (t == 0) x = i;
    }
    qsize = 0;
    t = eval(s);
    if      (t <= 22) magic = 1.12;
    else if (t <= 25) magic = 1.14;
    else              magic = 1.28;

    enq(s, x, 0, (int)(t*magic));

    insert(s, 0);
    while (qsize) {
        s = que[0].s, x = que[0].x, v = que[0].v, deq();
        if (s == GOAL) break;
        if (lookup(s) < v) continue;

        toArray(p16, s);
        for (i = 0; i < hi[x]; i++) {
            nx = to[x][i];
            ns = swap(p16, x, nx), nv = v+1;
            t = lookup(ns);
            if (t < 1 || nv < t) {
                enq(ns, nx, nv, eval(ns)+v);
                insert(ns,  nv);
            }
        }
    }
    printf("%d\n", v);
    return 0;
}


-----------------------------------------
FILE: ALDS1_2_A.c
-----------------------------------------
/*
バブルソート
バブルソートはその名前が表すように、泡（Bubble）が水面に上がっていくように配列の要素が動いていきます。バブルソートは次のようなアルゴリズムで数列を昇順に並び変えます。

1 bubbleSort(A, N) // N 個の要素を含む 0-オリジンの配列 A
2   flag = 1 // 逆の隣接要素が存在する
3   while flag
4     flag = 0
5     for j が N-1 から 1 まで
6       if A[j] < A[j-1]
7         A[j] と A[j-1] を交換
8         flag = 1
数列 A を読み込み、バブルソートで昇順に並び変え出力するプログラムを作成してください。また、バブルソートで行われた要素の交換回数も報告してください。

入力
入力の最初の行に、数列の長さを表す整数 N が与えられます。2 行目に、N 個の整数が空白区切りで与えられます。
```



```
出力
出力は 2 行からなります。1 行目に整列された数列を 1 行に出力してください。数列の連続する要素は 1 つの空白で区切って出力してください。2 行目に交換回数を出力してください。

制約
1 ≤ N ≤ 100
0 ≤ A の要素 ≤ 100
入力例 1
5
5 3 2 4 1
出力例 1
1 2 3 4 5
8

入力例 2
6
5 2 4 6 1 3
出力例 2
1 2 3 4 5 6
9
*/

#include<stdio.h>

int bubbleSort(int x[],int n){
    int count = 0, flag = 1;

    while(flag){
        flag = 0;
        for(int i = n - 1; i > 0; i--){
            if(x[i] < x[i - 1]){
                int temp = x[i];
                x[i] = x[i - 1];
                x[i - 1] = temp;
                count++;
                flag = 1;
            }
        }
    }
    return count;
}

int main(){
    int n;
    scanf("%d",&n);

    int x[n];
    for(int i = 0; i < n; i++){
        scanf("%d",&x[i]);
    }

    int count = bubbleSort(x,n);

    for(int i = 0; i < n; i++){
        if(i) printf(" ");
        printf("%d",x[i]);
    }
    printf("\n%d\n",count);

    return 0;
}
```



```

-----------------------------------------
FILE: ALDS1_2_B.c
-----------------------------------------
/*
選択ソート
選択ソートは、各計算ステップで 1 つの最小値を「選択」していく、直観的なアルゴリズムです。

1 selectionSort(A, N) // N個の要素を含む0-オリジンの配列A
2   for i が 0 から N-1 まで
3     minj = i
4     for j が i から N-1 まで
5       if A[j] < A[minj]
6         minj = j
7     A[i] と A[minj] を交換
数列Aを読み込み、選択ソートのアルゴリズムで昇順に並び替え出力するプログラムを作成してください。上の疑似コードに従いアルゴリズムを実装してください。

疑似コード 7 行目で、i と minj が異なり実際に交換が行われた回数も出力してください。

入力
入力の最初の行に、数列の長さを表す整数 N が与えられます。2 行目に、N 個の整数が空白区切りで与えられます。

出力
出力は 2 行からなります。1 行目に整列された数列を 1 行に出力してください。数列の連続する要素は 1 つの空白で区切って出力してください。2 行目に交換回数を出力してください。

制約
1 ≤ N ≤ 100
0 ≤ A の要素 ≤ 100
入力例 1
6
5 6 4 2 1 3
出力例 1
1 2 3 4 5 6
4

入力例 2
6
5 2 4 6 1 3
出力例 2
1 2 3 4 5 6
3
*/

#include<stdio.h>

int selectionSort(int x[],int n){
    int count = 0;
    int minj,temp;

    for(int i = 0;i < n;i++){
        minj = i;
        for(int j = i; j < n; j++){
            if(x[j] < x[minj]){
                minj = j;
            }
        }
        if(i != minj){
        temp = x[minj];
        x[minj] = x[i];
        x[i] = temp;
        count++;
        }
```



```
    }
    return count;
}

int main(){
    int n;
    scanf("%d",&n);

    int x[n];
    for(int i = 0; i < n; i++){
        scanf("%d",&x[i]);
    }

    int count = selectionSort(x,n);

    for(int i = 0; i < n; i++){
        if(i) printf(" ");
        printf("%d",x[i]);
    }
    printf("\n%d\n",count);

    return 0;
}


-----------------------------------------
FILE: ALDS1_2_C.c
-----------------------------------------
/*
安定なソート
トランプのカードを整列しましょう。ここでは、4 つの絵柄(S, H, C, D)と 9 つの数字(1, 2, ..., 9)から構成される計 36 枚のカードを用います。例えば、ハートの 8 は"H8"、ダイヤの 1 は"D1"と表します。

バブルソート及び選択ソートのアルゴリズムを用いて、与えられた N 枚のカードをそれらの数字を基準に昇順に整列するプログラムを作成してください。アルゴリズムはそれぞれ以下に示す疑似コードに従うものとします。数列の要素は 0 オリジンで記述され

1 BubbleSort(C, N)
2   for i = 0 to N-1
3     for j = N-1 downto i+1
4       if C[j].value < C[j-1].value
5         C[j] と C[j-1] を交換
6
7 SelectionSort(C, N)
8   for i = 0 to N-1
9     minj = i
10    for j = i to N-1
11      if C[j].value < C[minj].value
12        minj = j
13    C[i] と C[minj] を交換
また、各アルゴリズムについて、与えられた入力に対して安定な出力を行っているか報告してください。ここでは、同じ数字を持つカードが複数ある場合それらが入力に出現する順序で出力されることを、「安定な出力」と呼ぶことにします。（※常に安定な出力

入力
1 行目にカードの枚数 N が与えられます。 2 行目に N 枚のカードが与えられます。各カードは絵柄と数字のペアを表す 2 文字であり、隣合うカードは 1 つの空白で区切られています。

出力
1 行目に、バブルソートによって整列されたカードを順番に出力してください。隣合うカードは 1 つの空白で区切ってください。
2 行目に、この出力が安定か否か（Stable またはNot stable）を出力してください。
3 行目に、選択ソートによって整列されたカードを順番に出力してください。隣合うカードは 1 つの空白で区切ってください。
4 行目に、この出力が安定か否か（Stable またはNot stable）を出力してください。

制約
1 ≤ N ≤ 36
入力例 1
5
```

```
H4 C9 S4 D2 C3
出力例 1
D2 C3 H4 S4 C9
Stable
D2 C3 S4 H4 C9
Not stable
入力例 2
2
S1 H1
出力例 2
S1 H1
Stable
S1 H1
Stable
*/

#include <stdio.h>

typedef struct {
    char suit;
    int value;
    int index;
} Card;

void printCards(Card A[], int N) {
    for (int i = 0; i < N; i++) {
        if (i > 0) printf(" ");
        printf("%c%d", A[i].suit, A[i].value);
    }
    printf("\n");
}

int isStable(Card A[], int N) {
    for (int i = 1; i < N; i++) {
        if (A[i - 1].value == A[i].value && A[i - 1].index > A[i].index) {
            return 0;
        }
    }
    return 1;
}

void bubbleSort(Card A[], int N) {
    for (int i = 0; i < N; i++) {
        for (int j = N - 1; j >= i + 1; j--) {
            if (A[j].value < A[j - 1].value) {
                Card tmp = A[j];
                A[j] = A[j - 1];
                A[j - 1] = tmp;
            }
        }
    }
}

void selectionSort(Card A[], int N) {
    for (int i = 0; i < N; i++) {
        int minj = i;
        for (int j = i; j < N; j++) {
            if (A[j].value < A[minj].value) {
                minj = j;
            }
        }
        if (i != minj) {
```



```
            Card tmp = A[i];
            A[i] = A[minj];
            A[minj] = tmp;
        }
    }
}

int main(void) {
    int N;
    scanf("%d", &N);

    Card original[36];
    for (int i = 0; i < N; i++) {
        char s;
        int v;
        scanf(" %c%d", &s, &v);
        original[i].suit = s;
        original[i].value = v;
        original[i].index = i;
    }

    Card bubble[36];
    Card select[36];
    for (int i = 0; i < N; i++) {
        bubble[i] = original[i];
        select[i] = original[i];
    }

    bubbleSort(bubble, N);
    printCards(bubble, N);
    printf(isStable(bubble, N) ? "Stable\n" : "Not stable\n");

    selectionSort(select, N);
    printCards(select, N);
    printf(isStable(select, N) ? "Stable\n" : "Not stable\n");

    return 0;
}


----------------------------------------------
FILE: ALDS1_2_D.c
----------------------------------------------
/*
Shell Sort
次のプログラムは、挿入ソート(ALDS1_1_A)を応用して
 個の整数を含む数列
 を昇順に整列するプログラムです。

1  insertionSort(A, n, g)
2      for i = g to n-1
3          v = A[i]
4          j = i - g
5          while j >= 0 && A[j] > v
6              A[j+g] = A[j]
7              j = j - g
8              cnt++
9          A[j+g] = v
10
11 shellSort(A, n)
12     cnt = 0
13     m = ?
```



```
14     G[] = {?, ?,..., ?}
15     for i = 0 to m-1
16         insertionSort(A, n, G[i])
shellSort(A, n) は、一定の間隔
 だけ離れた要素のみを対象とした挿入ソートである insertionSort(A, n, g) を、最初は大きい値から
 を狭めながら繰り返します。これをシェルソートと言います。

上の疑似コードの ? を埋めてこのプログラムを完成させてください。
 と数列
 が与えられるので、疑似コード中の
、
 個の整数
、入力
を昇順にした列を出力するプログラムを作成してください。ただし、出力は以下の条件を満 たす必要があります。

cnt の値は
 を超えてはならない
入力
1 行目に整数
 が与えられます。続く
 行目に
 個の整数
 が与えられます。

出力
1 行目に整数
、2 行目に
 個の整数
 を空白区切りで出力してください。
3 行目に、
 を用いた場合のプログラムが終了した直後の cnt の値を出力してください。
続く
 行に整列した
 を出力してください。

この問題では、1 つの入力に対して複数の解答があります。条件を満たす出力は全て正解となります。

制約
入力例 1
5
5
1
4
3
2
出力例 1
2
4 1
3
1
2
3
4
5
入力例 2
3
3
2
1
出力例 2
1
1
```



```
3
1
2
3
*/

/*
シェルソートの解説

【アルゴリズムの仕組み】
1. 大きな間隔gで要素を分割し、各グループを挿入ソート
2. 間隔を徐々に狭めて(g=4 → g=1)繰り返す
3. 最後にg=1で通常の挿入ソートを実行

【間隔列の生成】
h = 3*h + 1 で生成: 1, 4, 13, 40, 121, ...
→ n以下の値を逆順に使用

【なぜ効率的か】
- 大きな間隔で粗く整列 → データが「ほぼ整列」状態に
- 小さな間隔での挿入ソートが高速化（移動距離が短い）
- 計算量: O(n^1.5) 程度

【例】配列 [5, 1, 4, 3, 2], n=5
間隔列G = [4, 1]
1) g=4: [2, 1, 4, 3, 5]（間隔4の要素を比較）
2) g=1: [1, 2, 3, 4, 5]（通常の挿入ソート）
*/

#include <stdio.h>

long long cnt;    // 要素の移動回数をカウント
int A[1000000];   // ソート対象の配列
int n;            // 配列のサイズ
int G[50];        // 間隔列（最大50個）
int m;            // 間隔列の個数

// 間隔gで挿入ソート（シェルソートのサブルーチン）
// g=1なら通常の挿入ソート、g>1なら飛び飛びの挿入ソート
void insertionSort(int A[], int n, int g) {
    for (int i = g; i < n; i++) {
        int v = A[i];          // 挿入する値
        int j = i - g;         // g個前の位置から比較開始

        // vより大きい要素を右にg個ずつずらす
        while (j >= 0 && A[j] > v) {
            A[j + g] = A[j];   // g個右にずらす
            j = j - g;         // g個前へ
            cnt++;             // 移動回数をカウント
        }
        A[j + g] = v;          // 適切な位置にvを挿入
    }
}

// シェルソート本体
void shellSort(int A[], int n) {
    cnt = 0;
    m = 0;

    // 間隔列Gを生成: h = 3*h + 1 (1, 4, 13, 40, ...)
    // n以下の値を配列Gに格納
    for (int h = 1; h <= n; h = 3 * h + 1) {
```

```
        G[m++] = h;
    }

    // 大きい間隔から小さい間隔へ降順で挿入ソート
    // 例: G = [1, 4, 13] なら 13 → 4 → 1 の順
    for (int i = m - 1; i >= 0; i--) {
        insertionSort(A, n, G[i]);
    }
}

int main(void) {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &A[i]);
    }

    shellSort(A, n);

    // 出力1: 間隔列の個数m
    printf("%d\n", m);

    // 出力2: 間隔列G（降順）を空白区切りで1行に
    for (int i = m - 1; i >= 0; i--) {
        if (i < m - 1) printf(" ");
        printf("%d", G[i]);
    }
    printf("\n");

    // 出力3: 要素の移動回数cnt
    printf("%lld\n", cnt);

    // 出力4: ソート済み配列（各要素を1行ずつ）
    for (int i = 0; i < n; i++) {
        printf("%d\n", A[i]);
    }

    return 0;
}



----------------------------------------------
FILE: ALDS1_3_A.c
----------------------------------------------
/*
スタック
逆ポーランド記法は、演算子をオペランドの後に記述する数式やプログラムを記述する記法です。例えば、一般的な中間記法で記述された数式 (1+2)*(5+4) は、逆ポーランド記法では 1 2 + 5 4 + * と記述されます。逆ポーランド記法では、中間記法でな

逆ポーランド記法で与えられた数式の計算結果を出力してください。

入力
1 つの数式が 1 行に与えられます。連続するシンボル（オペランドあるいは演算子）は 1 つの空白で区切られて与えられます。

出力
計算結果を 1 行に出力してください。

制約
2 ≤ 式に含まれるオペランドの数 ≤ 100
1 ≤ 式に含まれる演算子の数 ≤ 99
演算子は +, -, * のみを含み、1 つのオペランドは106 以下の正の整数
−1 × 10^9 ≤ 計算途中の値 ≤ 10^9
入力例 1
```



```
1 2 +
出力例 1
3
入力例 2
1 2 + 3 4 - *
出力例 2
-3
*/

/*
逆ポーランド記法の解説

【逆ポーランド記法とは】
演算子をオペランドの後に記述する記法
例: (1+2)*(5+4) → 1 2 + 5 4 + *

【スタックを使った計算方法】
1. 数値が来たらスタックに積む(push)
2. 演算子が来たら：
   - スタックから2つ取り出す(pop)
   - 演算を実行
   - 結果をスタックに積む(push)
3. 最後にスタックに残った値が答え

【例】1 2 + 3 4 - *
処理:        スタック
1     →     [1]
2     →     [1, 2]
+     →     [3]          (1+2=3)
3     →     [3, 3]
4     →     [3, 3, 4]
-     →     [3, -1]      (3-4=-1)
*     →     [-3]         (3*(-1)=-3)

答え: -3
*/

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

#define MAX 100

// スタックの実装
int stack[MAX];
int top = 0;

// スタックに値を積む
void push(int x) {
    stack[top++] = x;
}

// スタックから値を取り出す
int pop() {
    return stack[--top];
}

int main(void) {
    char s[100];

    // 入力を空白区切りで読み込んで処理
    while (scanf("%s", s) != EOF) {
```



```
        // 数値かどうかを判定（演算子でないもの）
        if (s[0] == '+' || s[0] == '-' || s[0] == '*') {
            // 演算子の場合
            int b = pop();  // 右オペランド
            int a = pop();  // 左オペランド

            // 演算を実行してスタックに積む
            if (s[0] == '+') {
                push(a + b);
            } else if (s[0] == '-') {
                push(a - b);
            } else if (s[0] == '*') {
                push(a * b);
            }
        } else {
            // 数値の場合、スタックに積む
            push(atoi(s));
        }
    }

    // スタックに残った値が答え
    printf("%d\n", pop());

    return 0;
}



----------------------------------------------
FILE: ALDS1_3_B.c
----------------------------------------------
/*
キュー
名前 namei と必要な処理時間 timei を持つ n 個のプロセスが順番に一列に並んでいます。ラウンドロビンスケジューリングと呼ばれる処理方法では、CPU がプロセスを順番に処理します。各プロセスは最大 q ms（これをクオンタムと呼びます）だけ処理

例えば、クオンタムを 100 msとし、次のようなプロセスキューを考えます。

A(150) - B(80) - C(200) - D(200)
まずプロセス A が 100 ms だけ処理され残りの必要時間 50 ms を保持しキューの末尾に移動します。

B(80) - C(200) - D(200) - A(50)
次にプロセス B が 80 ms だけ処理され、時刻 180 ms で終了し、キューから削除されます。

C(200) - D(200) - A(50)
次にプロセス C が 100 ms だけ処理され、残りの必要時間 100 ms を保持し列の末尾に移動します。

D(200) - A(50) - C(100)
このように、全てのプロセスが終了するまで処理を繰り返します。

ラウンドロビンスケジューリングをシミュレートするプログラムを作成してください。

入力
入力の形式は以下の通りです。

n q
name1 time1
name2 time2
...
namen timen
最初の行に、プロセス数を表す整数 n とクオンタムを表す整数 q が 1 つの空白区切りで与えられます。

続く n 行で、各プロセスの情報が順番に与えられます。文字列 namei と整数 timei は 1 つの空白で区切られています。
```



```
出力
プロセスが完了した順に、各プロセスの名前と終了時刻を空白で区切って 1 行に出力してください。

制約
1 ≤ n ≤ 100,000
1 ≤ q ≤ 1,000
1 ≤ timei ≤ 50,000
1 ≤ 文字列 namei の長さ ≤ 10
1 ≤ timei の合計 ≤ 1,000,000
入力例 1
5 100
p1 150
p2 80
p3 200
p4 350
p5 20
出力例 1
p2 180
p5 400
p1 450
p3 550
p4 800
*/

/*
ラウンドロビンスケジューリングの解説

【キューとは】
先入れ先出し(FIFO: First In First Out)のデータ構造
最初に入れたものが最初に出てくる（行列と同じ）

【ラウンドロビンの処理手順】
1. キューの先頭からプロセスを取り出す
2. クオンタムq ms分だけ処理する
3. まだ時間が残っていれば、キューの末尾に戻す
4. 完了していれば、終了時刻を出力
5. キューが空になるまで繰り返す

【例】q=100の場合
初期: A(150) - B(80) - C(200) - D(200)
時刻0:   A→100ms処理 → B(80) - C(200) - D(200) - A(50)
時刻100: B→80ms処理→完了(180ms) → C(200) - D(200) - A(50)
時刻180: C→100ms処理 → D(200) - A(50) - C(100)
時刻280: D→100ms処理 → A(50) - C(100) - D(100)
時刻380: A→50ms処理→完了(430ms) → C(100) - D(100)
時刻430: C→100ms処理→完了(530ms) → D(100)
時刻530: D→100ms処理→完了(630ms) → 終了
*/

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

#define MAX 100000

// プロセス構造体
typedef struct {
    char name[11];
    int time;
} Process;

// キューの実装（配列による循環キュー）
```

```
Process queue[MAX];
int head = 0;  // キューの先頭
int tail = 0;  // キューの末尾
int size = 0;  // キューのサイズ

// キューにプロセスを追加 (enqueue)
void enqueue(Process p) {
    queue[tail] = p;
    tail = (tail + 1) % MAX;
    size++;
}

// キューからプロセスを取り出す (dequeue)
Process dequeue() {
    Process p = queue[head];
    head = (head + 1) % MAX;
    size--;
    return p;
}

int main(void) {
    int n, q;
    scanf("%d %d", &n, &q);

    // 初期プロセスをキューに登録
    for (int i = 0; i < n; i++) {
        Process p;
        scanf("%s %d", p.name, &p.time);
        enqueue(p);
    }

    int current_time = 0;

    // キューが空になるまで処理
    while (size > 0) {
        Process p = dequeue();

        if (p.time <= q) {
            // クオンタム以内に終了する場合
            current_time += p.time;
            printf("%s %d\n", p.name, current_time);
        } else {
            // クオンタムで処理しきれない場合
            current_time += q;
            p.time -= q;
            enqueue(p);  // キューの末尾に戻す
        }
    }

    return 0;
}



----------------------------------------------
FILE: ALDS1_3_C.c
----------------------------------------------
/*
双方向連結リスト
以下の命令を受けつける双方向連結リストを実装してください。

insert x: 連結リストの先頭にキー x を持つ要素を継ぎ足す。
```



```
delete x: キー x を持つ最初の要素を連結リストから削除する。そのような要素が存在しない場合は何もしない。
deleteFirst: リストの先頭の要素を削除する。
deleteLast: リストの末尾の要素を削除する。

入力
入力は以下の形式で与えられます。

n
command1
command2
...
commandn
最初の行に命令数 n が与えられます。続く n 行に命令が与えられる。命令は上記4つの命令のいずれかです。キーは整数とします。

出力
全ての命令が終了した後の、連結リスト内のキーを順番に出力してください。連続するキーは 1 つの空白文字で区切って出力してください。

制約
命令数は 2,000,000 を超えない。
delete 命令の回数は 20 を超えない。
0 ≤ キーの値 ≤ 10^9。
命令の過程でリストの要素数は 10^6を超えない。
delete, deleteFirst, または deleteLast 命令が与えられるとき、リストには 1 つ以上の要素が存在する。

入力例 1
7
insert 5
insert 2
insert 3
insert 1
delete 3
insert 6
delete 5
出力例 1
6 1 2

入力例 2
9
insert 5
insert 2
insert 3
insert 1
delete 3
insert 6
delete 5
deleteFirst
deleteLast
出力例 2
1
*/

/*
双方向連結リストの解説

【双方向連結リストとは】
各要素が「次の要素」と「前の要素」の両方へのポインタを持つデータ構造
先頭からも末尾からもアクセスでき、挿入・削除がO(1)で可能

【構造体の定義】
typedef struct Node {
    int key;              // データ
    struct Node *next;    // 次の要素へのポインタ
```



```
    struct Node *prev;    // 前の要素へのポインタ
} Node;

【基本操作】
1. insert(x): 先頭に新しい要素を追加
   - 新しいノードを作成
   - head の前に挿入

2. delete(x): 指定された値を持つ最初の要素を削除
   - リストを先頭から探索
   - 見つかったら前後のポインタを繋ぎ直す

3. deleteFirst(): 先頭の要素を削除
   - head を次の要素に移動

4. deleteLast(): 末尾の要素を削除
   - tail を前の要素に移動

【メモリ管理】
- insert時: malloc()で新しいノードを確保
- delete時: free()でメモリを解放（メモリリーク防止）
*/

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

// 双方向連結リストのノード構造体
typedef struct Node {
    int key;
    struct Node *next;
    struct Node *prev;
} Node;

// ダミーノード（番兵）
Node *nil;

// 新しいノードを作成
Node* createNode(int key) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->key = key;
    node->next = nil;
    node->prev = nil;
    return node;
}

// リストの初期化（ダミーノードを使う方式）
void init() {
    nil = (Node *)malloc(sizeof(Node));
    nil->next = nil;
    nil->prev = nil;
}

// 先頭に挿入
void insert(int key) {
    Node *x = createNode(key);
    x->next = nil->next;
    nil->next->prev = x;
    nil->next = x;
    x->prev = nil;
}
```



```
// 指定したキーを持つ最初の要素を削除
void deleteKey(int key) {
    Node *cur = nil->next;
    while (cur != nil) {
        if (cur->key == key) {
            cur->prev->next = cur->next;
            cur->next->prev = cur->prev;
            free(cur);
            return;  // 最初の要素だけ削除
        }
        cur = cur->next;
    }
}

// 先頭の要素を削除
void deleteFirst() {
    if (nil->next == nil) return;
    Node *x = nil->next;
    nil->next = x->next;
    x->next->prev = nil;
    free(x);
}

// 末尾の要素を削除
void deleteLast() {
    if (nil->prev == nil) return;
    Node *x = nil->prev;
    x->prev->next = nil;
    nil->prev = x->prev;
    free(x);
}

// リストの内容を出力
void printList() {
    Node *cur = nil->next;
    int first = 1;
    while (cur != nil) {
        if (!first) printf(" ");
        printf("%d", cur->key);
        first = 0;
        cur = cur->next;
    }
    printf("\n");
}

int main(void) {
    int n, key;
    char command[20];

    scanf("%d", &n);
    init();

    for (int i = 0; i < n; i++) {
        scanf("%s", command);

        if (strcmp(command, "insert") == 0) {
            scanf("%d", &key);
            insert(key);
        } else if (strcmp(command, "delete") == 0) {
            scanf("%d", &key);
            deleteKey(key);
        } else if (strcmp(command, "deleteFirst") == 0) {
```

```
            deleteFirst();
        } else if (strcmp(command, "deleteLast") == 0) {
            deleteLast();
        }
    }

    printList();

    return 0;
}



---------------------------------------------
FILE: ALDS1_3_D.c
---------------------------------------------
/*
Areas on the Cross-Section Diagram
地域の治水対策として、洪水の被害状況をシミュレーションで仮想してみよう。

図のように
 の区画からなる格子上に表された地域の模式断面図が与えられるので、地域にできる各水たまりの面積を報告してください。



与えられた地域に対して限りなく雨が降り、地域から溢れ出た水は左右の海に流れ出ると仮定します。 例えば、図の断面図では、左から面積が 4、2、1、19、9 の水たまりができます。

入力
模式断面図における斜面を '/' と '\'、平地を '_' で表した文字列が 1 行に与えられます。例えば、図の模式断面図は文字列 \\///\_/\/\\\\/_/\\///__\\\_\\/_\/_/\ で与えられます。

出力
次の形式で水たまりの面積を出力してください。



 ...

 1 行目に地域にできる水たまりの総面積を表す整数
 を出力してください。

 2 行目に水たまりの数
 、各水たまりの面積
 を断面図の左から順番に空白区切りで出力してください。

制約
文字列の長さ
ただし、得点の 50 点分は以下の条件を満たす。

水たまりの数は 1 つ以下であり（
）、かつ文字列の長さは 100 以下である。
入力例 1
\\/
出力例 1
4
1 4

入力例 2
\\//\_/\/\\\\/_/\\///__\\\_\\/_\/_/\
出力例 2
35
5 4 2 1 19 9
```



```
 */

/*
断面図の水たまり面積計算の解説

【アルゴリズムの仕組み】
スタックを2つ使って、'\' と '/' の対応関係から水たまりを検出

【基本原理】
1. '\'（下り坂）の位置をスタックに記録
2. '/'（上り坂）が来たら、対応する '\' とペアを作る
   → この間の距離が水たまりの面積
3. 同じ谷の中にある複数の水たまりを統合（マージ）

【具体例】\\//
位置:  0 1 2 3
文字:  \ \ / /

処理:
位置0: '\' → s1にpush(0)
位置1: '\' → s1にpush(1)
位置2: '/' → s1からpop(1), 面積 = 2-1 = 1
             s2にpush(1, 1) ※(開始位置, 面積)
位置3: '/' → s1からpop(0), 面積 = 3-0 = 3
             s2に(1,1)が残っていて1>0なので統合
             統合後の面積 = 3 + 1 = 4
             s2にpush(0, 4)

結果: 総面積=4, 水たまり1個(面積4)

【マージの必要性】
\\//\// の場合:
- 最初の \\// で1つの水たまり
- 次の \// でもう1つの水たまり
- しかし実際は大きな1つの水たまりなので統合が必要

【2つのスタック】
s1_stack: '\' の出現位置を記録
s2_stack: (水たまりの左端, 面積) のペアを記録
*/

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

#define MAX_LEN 20001

// スタック1: '\' の位置を記録
int s1_stack[MAX_LEN];
int s1_top = 0;

// スタック2: (位置, 面積) のペアを記録
typedef struct {
    int pos;
    int area;
} Pair;

Pair s2_stack[MAX_LEN];
int s2_top = 0;

// スタック1の操作
void s1_push(int x) {
    s1_stack[s1_top++] = x;
```



```
}

int s1_pop() {
    return s1_stack[--s1_top];
}

int s1_empty() {
    return s1_top == 0;
}

// スタック2の操作
void s2_push(int pos, int area) {
    s2_stack[s2_top].pos = pos;
    s2_stack[s2_top].area = area;
    s2_top++;
}

Pair s2_pop() {
    return s2_stack[--s2_top];
}

int s2_empty() {
    return s2_top == 0;
}

Pair s2_top_pair() {
    return s2_stack[s2_top - 1];
}

int main(void) {
    char s[MAX_LEN];

    if (scanf("%s", s) != 1) {
        return 0;
    }

    int total_area = 0;
    int len = strlen(s);

    for (int j = 0; j < len; j++) {
        if (s[j] == '\\') {
            // 下り坂の位置をスタックに記録
            s1_push(j);
        } else if (s[j] == '/' && !s1_empty()) {
            // 上り坂が来たら、対応する下り坂を取り出す
            int i = s1_pop();

            // 現在の断面で形成される面積を計算
            int current_area = j - i;
            total_area += current_area;

            // マージ処理: 範囲 (i, j) 内に含まれる水たまりを統合
            while (!s2_empty() && s2_top_pair().pos > i) {
                Pair p = s2_pop();
                current_area += p.area;
            }

            // 統合された水たまりを記録
            s2_push(i, current_area);
        }
    }
```



```
    // 出力1: 総面積
    printf("%d\n", total_area);

    // 出力2: 水たまりの数と各面積
    // スタックから取り出すと逆順になるので、一度配列に移す
    int results[MAX_LEN];
    int count = 0;

    while (!s2_empty()) {
        Pair p = s2_pop();
        results[count++] = p.area;
    }

    // 逆順になっているので反転
    printf("%d", count);
    for (int i = count - 1; i >= 0; i--) {
        printf(" %d", results[i]);
    }
    printf("\n");

    return 0;
}



---------------------------------------------
FILE: ALDS1_4_A.c
---------------------------------------------
/*
線形探索
 個の整数を含む数列
 と、
 個の異なる整数を含む数列
 を読み込み、
 に含まれる整数の中で
 に含まれるものの個数
 を出力するプログラムを作成してください。

入力
1 行目に
、2 行目に
 を表す
 個の整数、3 行目に
、4 行目に
 を表す
 個の整数が与えられます。

出力
 を 1 行に出力してください。

制約
の要素
の要素
 の要素は互いに異なる
入力例 1
5
1 2 3 4 5
3
3 4 1
出力例 1
3
入力例 2
```

```
  3
  3 1 2
  1
  5
  出力例 2
  0
  入力例 3
  5
  1 1 2 2 3
  2
  1 2
  出力例 3
  2
  */

  /*
  線形探索の解説

  【線形探索とは】
  配列の先頭から順番に目的の要素を探す最も基本的な探索アルゴリズム
  時間計算量: O(n)

  【番兵（sentinel）の活用】
  通常の線形探索では、ループ内で「配列の範囲チェック」と「要素の一致チェック」の
  2つの条件判定が必要：
    while (i < n && A[i] != key)

  番兵を使うと：
  1. 配列の末尾（n番目）に探索対象のkeyを配置
  2. これにより必ず途中でkeyが見つかるため、範囲チェックが不要に
  3. ループ内の条件判定が1つで済む: while (A[i] != key)
  4. 最後に「末尾で見つかったか（=実際には存在しない）」を判定

  【具体例】
  配列S = [1, 2, 3, 4, 5], key = 3 を探す場合

  ■ 番兵なし:
    i=0: i<5 && S[0]!=3 → 真 → 継続
    i=1: i<5 && S[1]!=3 → 真 → 継続
    i=2: i<5 && S[2]!=3 → 偽 → 発見！

  ■ 番兵あり:
    配列を [1, 2, 3, 4, 5, 3] に拡張（末尾に3を追加）
    i=0: S[0]!=3 → 真 → 継続
    i=1: S[1]!=3 → 真 → 継続
    i=2: S[2]!=3 → 偽 → 発見！
    i!=n (2!=5) なので本当に存在する

  【このアルゴリズムの利点】
  - 条件判定が1回で済むため高速化
  - コードがシンプルになる
  */

  #include <stdio.h>

  // 線形探索（番兵を使用）
  // 見つかったら 1, 見つからなければ 0 を返す
  int linearSearch(int n, int A[], int key) {
      int i = 0;
      A[n] = key; // 配列の末尾に番兵をセット

      // 範囲チェック（i < n）が不要になる
```



```
      while (A[i] != key) {
          i++;
      }

      // 見つかった位置が末尾（番兵）でなければ真の発見
      return i != n;
  }

  int main(void) {
      int n, q, key;
      int count = 0;

      // 数列 S の読み込み
      scanf("%d", &n);
      int S[n + 1]; // 番兵のために 1 つ多めに確保
      for (int i = 0; i < n; i++) {
          scanf("%d", &S[i]);
      }

      // 数列 T の読み込みと同時に探索を行う
      scanf("%d", &q);
      for (int i = 0; i < q; i++) {
          scanf("%d", &key);
          // S の中に key が存在するかチェック
          if (linearSearch(n, S, key)) {
              count++;
          }
      }

      // 合計数を出力
      printf("%d\n", count);

      return 0;
  }



  ----------------------------------------------
  FILE: ALDS1_4_B.c
  ----------------------------------------------
  /*
  二分探索
   個の整数を含む数列
  と、
  個の異なる整数を含む数列
  を読み込み、
  に含まれる整数の中で
  に含まれるものの個数
  を出力するプログラムを作成してください。

  入力
  1行目に
  、2行目に
  を表す
   個の整数、3行目に
  、4行目に
  を表す
   個の整数が与えられます。

  出力
   を1行に出力してください。
```



```
  制約
   の要素は昇順に整列されている
  の要素
  の要素
   の要素は互いに異なる
  入力例 1
  5
  1 2 3 4 5
  3
  3 4 1
  出力例 1
  3
  入力例 2
  3
  1 2 3
  1
  5
  出力例 2
  0
  入力例 3
  5
  1 1 2 2 3
  2
  1 2
  出力例 3
  2
  */

  /*
  二分探索の解説

  【二分探索とは】
  整列済み配列に対して、目的の要素を高速に探すアルゴリズム
  時間計算量: O(log n) ← 線形探索のO(n)より圧倒的に高速

  【アルゴリズムの仕組み】
  1. 探索範囲の中央の要素と目的の値を比較
  2. 一致すれば発見
  3. 目的の値が中央より小さければ左半分を探索
  4. 目的の値が中央より大きければ右半分を探索
  5. 探索範囲がなくなるまで繰り返す

  【具体例】配列 [1, 2, 3, 4, 5] で key=4 を探す
  初期: left=0, right=4
    1回目: mid=2, S[2]=3 < 4 → 右側を探索 → left=3
    2回目: mid=3, S[3]=4 == 4 → 発見！

  【具体例】配列 [1, 2, 3, 4, 5] で key=6 を探す
  初期: left=0, right=4
    1回目: mid=2, S[2]=3 < 6 → 右側を探索 → left=3
    2回目: mid=3, S[3]=4 < 6 → 右側を探索 → left=4
    3回目: mid=4, S[4]=5 < 6 → 右側を探索 → left=5
    left > right なのでループ終了 → 見つからない

  【なぜ高速か】
  毎回探索範囲が半分になるため、n個の要素でも約log₂(n)回で終わる
  例: n=1000 → 約10回, n=1000000 → 約20回

  【前提条件】
  配列が昇順（または降順）に整列されている必要がある
  */
```



```
  #include <stdio.h>

  // 二分探索: keyが配列Sの中に存在すれば1、なければ0を返す
  int binarySearch(int S[], int key, int n) {
      int left = 0;
      int right = n - 1;
      int mid;

      // 左端が右端を追い越すまで続ける
      while (left <= right) {
          // 真ん中のインデックスを計算
          mid = (left + right) / 2;

          if (S[mid] == key) {
              return 1; // 見つかった
          }
          // keyの方が大きい → 右側（大きい方）を探す
          else if (S[mid] < key) {
              left = mid + 1; // 左端をmidの右隣へ
          }
          // keyの方が小さい → 左側（小さい方）を探す
          else {
              right = mid - 1; // 右端をmidの左隣へ
          }
      }

      return 0; // 最後まで見つからなかった
  }

  int main(void) {
      int n;
      scanf("%d", &n);

      int S[n];
      for (int i = 0; i < n; i++) {
          scanf("%d", &S[i]);
      }

      int q;
      scanf("%d", &q);

      int count = 0;
      for (int i = 0; i < q; i++) {
          int key;
          scanf("%d", &key);
          // Sの中にkeyが存在するかチェック
          count += binarySearch(S, key, n);
      }

      printf("%d\n", count);

      return 0;
  }



  ----------------------------------------------
  FILE: ALDS1_4_C.c
  ----------------------------------------------
  /*
  辞書
  以下の命令を実行する簡易的な「辞書」を実装してください。
```

```
insert
: 辞書に
 を追加する。
find
: 辞書に
 が含まれる場合 'yes'と、含まれない場合 'no'と出力する。
入力
最初の行に命令の数
 が与えられます。続く
 行に
 件の命令が順番に与えられます。命令の形式は上記のとおりである。

出力
各 find 命令について、yes または no を 1 行に出力してください。

制約
与えられる文字列は、'A', 'C', 'G', 'T' の 4 種類の文字から構成される。
文字列の長さ
入力例 1
6
insert AAA
insert AAC
find AAA
find CCC
insert CCC
find CCC
出力例 1
yes
no
yes
入力例 2
13
insert AAA
insert AAC
insert AGA
insert AGG
insert TTT
find AAA
find CCC
find CCC
insert CCC
find CCC
insert T
find TTT
find T
出力例 2
yes
no
no
yes
yes
yes
*/

/*
辞書（ハッシュテーブル）の解説

【問題の特性】
- 文字列は'A', 'C', 'G', 'T'の4文字のみで構成
- 文字列長は最大12文字
- この制約を活かして効率的なハッシュ関数を設計可能
```



```
【ハッシュテーブルとは】
キーと値のペアを格納するデータ構造
ハッシュ関数でキーを数値（インデックス）に変換し、配列に格納
平均的にO(1)で挿入・検索が可能

【ハッシュ関数の設計】
各文字を以下のように数値化:
  'A' → 1, 'C' → 2, 'G' → 3, 'T' → 4
文字列全体を5進数とみなして数値化:
  例: "AC" → 1*5 + 2 = 7
  例: "ACG" → 1*5² + 2*5 + 3 = 38

【衝突（collision）の解決】
チェイン法: 同じハッシュ値を持つ要素を連結リストで管理
  例: hash(key1) == hash(key2) の場合、両方を同じリストに追加

【具体例】
insert AAA: hash("AAA") = 1*25 + 1*5 + 1 = 31
            H[31]のリストに"AAA"を追加
find AAA:   hash("AAA") = 31
            H[31]のリストを探索 → "AAA"が見つかる → "yes"
find CCC:   hash("CCC") = 2*25 + 2*5 + 2 = 62
            H[62]のリストを探索 → 空 → "no"

【ハッシュテーブルのサイズ】
文字列長12、4文字 → 最大4^12 = 16777216通り
実際にはそこまで必要ないので、適切なサイズ(1046527)を選択
素数を選ぶことで衝突を減らせる
*/

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

#define M 1046527  // ハッシュテーブルのサイズ（素数）
#define L 14       // 文字列の最大長+2

// 連結リストのノード構造体
typedef struct Node {
    char key[L];
    struct Node *next;
} Node;

// ハッシュテーブル（連結リストの配列）
Node *H[M];

// 文字を数値に変換
int getChar(char ch) {
    if (ch == 'A') return 1;
    if (ch == 'C') return 2;
    if (ch == 'G') return 3;
    if (ch == 'T') return 4;
    return 0;
}

// 文字列をハッシュ値に変換
long long getKey(char str[]) {
    long long sum = 0, p = 1;
    for (int i = 0; i < strlen(str); i++) {
        sum += p * getChar(str[i]);
        p *= 5;
```



```
    }
    return sum;
}

// ハッシュ値からインデックスを計算
int h(long long key) {
    return key % M;
}

// 辞書に文字列を挿入
void insert(char str[]) {
    long long key = getKey(str);
    int index = h(key);

    // 既に存在するかチェック
    Node *cur = H[index];
    while (cur != NULL) {
        if (strcmp(cur->key, str) == 0) {
            return; // 既に存在する場合は何もしない
        }
        cur = cur->next;
    }

    // 新しいノードを作成して先頭に追加
    Node *newNode = (Node *)malloc(sizeof(Node));
    strcpy(newNode->key, str);
    newNode->next = H[index];
    H[index] = newNode;
}

// 辞書に文字列が存在するか検索
int find(char str[]) {
    long long key = getKey(str);
    int index = h(key);

    Node *cur = H[index];
    while (cur != NULL) {
        if (strcmp(cur->key, str) == 0) {
            return 1; // 見つかった
        }
        cur = cur->next;
    }
    return 0; // 見つからない
}

int main(void) {
    int n;
    scanf("%d", &n);

    // ハッシュテーブルの初期化
    for (int i = 0; i < M; i++) {
        H[i] = NULL;
    }

    for (int i = 0; i < n; i++) {
        char command[10], str[L];
        scanf("%s %s", command, str);

        if (strcmp(command, "insert") == 0) {
            insert(str);
        } else if (strcmp(command, "find") == 0) {
            if (find(str)) {
```



```
                printf("yes\n");
            } else {
                printf("no\n");
            }
        }
    }

    return 0;
}



----------------------------------------------
FILE: ALDS1_4_D.c
----------------------------------------------
/*
割り当て
重さがそれぞれ
 の
 個の荷物が、ベルトコンベアから順番に流れてきます。これらの荷物を
 台のトラックに積みます。各トラックには連続する 0 個以上の荷物を積むことができますが、それらの重さの和がトラックの最大積載量
 を超えてはなりません。最大積載量
 はすべてのトラックで共通です。

、
、
 が与えられるので、すべての荷物を積むために必要な最大積載量
 の最小値を求めるプログラムを作成してください。

入力
最初の行に荷物の数
 とトラックの数
 が空白区切りで与えられます。続く
 行に
 個の整数
 がそれぞれ 1 行に与えられます。

出力
 の最小値を 1 行に出力してください。

制約
入力例 1
5 3
8
1
7
3
9
出力例 1
10
1 台目のトラックに 2 つの荷物
, 2 台目のトラックに 2 つの荷物
、3 台目のトラックに 1 つの荷物
 を積んで、最大積載量の最小値が 10 となります。


入力例 2
4 2
1
2
2
6
```

```
出力例 2
6
１台目のトラックに３つの荷物
, ２台目のトラックに１つの荷物
 を積んで、最大積載量の最小値が 6 となります。
*/

/*
割り当て問題（二分探索の応用）の解説

【問題の本質】
最大積載量Pの最小値を求める最適化問題
→ 「答えに対する二分探索」で解決

【アルゴリズムの仕組み】
1. 答えの範囲を決定
   - 最小値: max(荷物の重さ) ← 最大の荷物は必ず積める必要がある
   - 最大値: sum(全荷物の重さ) ← 全部を1台に積む場合

2. 二分探索で答えを探す
   left = max(w[i]), right = sum(w[i])
   while (left < right) {
       mid = (left + right) / 2
       if (check(mid)) → midで積める
           right = mid  // より小さい値を探す
       else
           left = mid + 1  // より大きい値が必要
   }

3. check関数: 容量Pで全荷物をk台以内に積めるか判定
   - 荷物を順番に現在のトラックに積む
   - 積めなくなったら次のトラックへ
   - 使用台数がk以下ならtrue

【具体例】入力例1: n=5, k=3, w=[8,1,7,3,9]
初期: left=9（最大荷物）, right=28（合計）

1回目: mid=18
  check(18): [8,1,7] [3,9] → 2台で積める → right=18

2回目: left=9, right=18, mid=13
  check(13): [8,1] [7,3] [9] → 3台で積める → right=13

3回目: left=9, right=13, mid=11
  check(11): [8,1] [7,3] [9] → 3台で積める → right=11

4回目: left=9, right=11, mid=10
  check(10): [8,1] [7,3] [9] → 3台で積める → right=10

5回目: left=9, right=10, mid=9
  check(9): [8] [1,7] [3] [9] → 4台必要 → left=10

終了: left=10 → 答え10

【なぜこれが正しいか】
- check(P)がtrueなら、P以上の値でもtrue（単調性）
- 二分探索で最小のtrueとなるPを見つける
- 計算量: O(n log(sum)) ← check関数がO(n)、二分探索がO(log(sum))
*/

#include <stdio.h>
```



```
int n, k;
int w[100000];

// 最大積載量がPのとき、k台以内で全荷物を積めるか判定
int check(int P) {
    int i = 0;
    int trucks = 1;  // 使用するトラック台数
    int current_load = 0;  // 現在のトラックの積載量

    while (i < n) {
        // 荷物がトラックに積める場合
        if (current_load + w[i] <= P) {
            current_load += w[i];
            i++;
        } else {
            // 積めない場合、次のトラックへ
            trucks++;
            current_load = 0;

            // トラック台数が超過した場合は不可能
            if (trucks > k) {
                return 0;
            }
        }
    }

    return 1;  // k台以内で積めた
}

int main(void) {
    scanf("%d %d", &n, &k);

    int left = 0;   // 最小値: 最大の荷物の重さ
    int right = 0;  // 最大値: 全荷物の合計

    for (int i = 0; i < n; i++) {
        scanf("%d", &w[i]);
        if (w[i] > left) {
            left = w[i];
        }
        right += w[i];
    }

    // 二分探索で最小の最大積載量を求める
    while (left < right) {
        int mid = (left + right) / 2;

        if (check(mid)) {
            // midで積める → より小さい値を探す
            right = mid;
        } else {
            // midでは積めない → より大きい値が必要
            left = mid + 1;
        }
    }

    printf("%d\n", left);

    return 0;
}
```



```
----------------------------------------------
FILE: ALDS1_5_A.c
----------------------------------------------
/*
総当たり
長さ
 の数列
 と整数
 に対して、
 の要素の中のいくつかの要素を足しあわせて
 が作れるかどうかを判定するプログラムを作成してください。
 の各要素は１度だけ使うことができます。

数列
 が与えられたうえで、質問として
 個の
 が与えれるので、それぞれについて "yes" または "no" と出力してください。

入力
１行目に
、２行目に
 を表す
 個の整数、３行目に
、４行目に
 個の整数
が与えられます。

出力
各質問について
 の要素を足しあわせて
 を作ることができれば yes と、できなければ no と出力してください。

制約
の要素
入力例 1
5
1 5 7 10 21
4
2 4 17 8
出力例 1
no
no
yes
yes
*/

/*
部分集合の和問題（動的計画法）の解説

【問題の本質】
配列Aの要素を0個以上使用して、各質問の数値を作れるかどうか判定
→ 「どの値が作れるか」を事前に計算する動的計画法

【アルゴリズムの仕組み】
1. possibleSums配列を用意（作れる値を記録）
2. possibleSums = {0}（何も足さない）からスタート
3. 配列Aの各要素について、現在作れる値に足した値も追加
4. 各質問について、possibleSumsに存在するか確認

【具体例】A = [1, 5, 7], 質問 = [1, 6, 8]
初期: possibleSums = {0}
```



```
i=0, A[0]=1:
  既存の値 {0} に1を足す → {1}
  possibleSums = {0, 1}

i=1, A[1]=5:
  既存の値 {0, 1} に5を足す → {5, 6}
  possibleSums = {0, 1, 5, 6}

i=2, A[2]=7:
  既存の値 {0, 1, 5, 6} に7を足す → {7, 8, 12, 13}
  possibleSums = {0, 1, 5, 6, 7, 8, 12, 13}

質問: 1 → possibleSumsに1がある → yes
質問: 6 → possibleSumsに6がある → yes
質問: 8 → possibleSumsに8がある → yes

【時間計算量】
- ループ回数: 要素数×作れる値の種類数
- 最悪の場合: O(n × n × max_sum)
  ※作れる値の最大数は2^n（2nビット未満）

【空間計算量】
- O(作れる値の総数) ≤ O(2^n)
  ※最大で2^n個の異なる値を作れる

【例】n=5, A=[1,5,7,10,21]
質問: 2 → 1+1は不可（1は1度だけ）→ no
質問: 4 → 1+5=6, 5+10=15... 作れない → no
質問: 17 → 7+10=17 → yes
質問: 8 → 1+7=8 → yes
*/

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

#define MAX_SUM 100001

int main(void) {
    int n;
    scanf("%d", &n);

    int A[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &A[i]);
    }

    // possibleSums[i] = iが作れるかどうか（1で作れる、0で作れない）
    int possibleSums[MAX_SUM];
    memset(possibleSums, 0, sizeof(possibleSums));
    possibleSums[0] = 1;  // 0は何も足さずに作れる

    // 配列Aの各要素について
    for (int i = 0; i < n; i++) {
        // 大きい値から小さい値へ更新（要素を2回以上使わないため）
        // 逆順でループして、既に計算された値を使わない
        for (int j = MAX_SUM - 1; j >= A[i]; j--) {
            // possibleSums[j-A[i]]が1なら、j も作れる
            if (possibleSums[j - A[i]]) {
                possibleSums[j] = 1;
            }
        }
```

```
    }

    // 質問処理
    int q;
    scanf("%d", &q);

    for (int i = 0; i < q; i++) {
        int query;
        scanf("%d", &query);

        if (query < MAX_SUM && possibleSums[query]) {
            printf("yes\n");
        } else {
            printf("no\n");
        }
    }

    return 0;
}



----------------------------------------
FILE: ALDS1_5_B.c
----------------------------------------
/*
マージソート
マージソート（Merge Sort）は分割統治法に基づく高速なアルゴリズムで、次のように実装することができます。

merge(A, left, mid, right)
  n1 = mid - left;
  n2 = right - mid;
  L[0...n1], R[0...n2] を生成
  for i = 0 to n1-1
    L[i] = A[left + i]
  for i = 0 to n2-1
    R[i] = A[mid + i]
  L[n1] = INFTY
  R[n2] = INFTY
  i = 0
  j = 0
  for k = left to right-1
    if L[i] <= R[j]
      A[k] = L[i]
      i = i + 1
    else
      A[k] = R[j]
      j = j + 1

mergeSort(A, left, right)
  if left+1 < right
    mid = (left + right)/2;
    mergeSort(A, left, mid)
    mergeSort(A, mid, right)
    merge(A, left, mid, right)
 個の整数を含む数列
 を上の疑似コードに従ったマージソートで昇順に整列するプログラムを作成してください。また、mergeにおける比較回数の総数を報告してください。

入力
1行目に
、2行目に
 を表す
```



```
 個の整数が与えられます。

出力
1行目に整列済みの数列
 を出力してください。数列の隣り合う要素は1つの空白で区切ってください。2行目に比較回数を出力してください。

制約
の要素
入力例 1
10
8 5 9 2 6 3 7 1 10 4
出力例 1
1 2 3 4 5 6 7 8 9 10
34
*/

/*
マージソートの解説

【マージソートとは】
分割統治法に基づく高速ソートアルゴリズム
時間計算量: O(n log n) - 最悪の場合でも保証される
メモリ効率: O(n) - 補助配列が必要

【アルゴリズムの流れ】
1. 配列を左右に分割（分割フェーズ）
2. 各半分を再帰的にソート
3. ソート済みの両配列をマージ（併合フェーズ）

【具体例】[8, 5, 9, 2, 6, 3, 7, 1]

■分割フェーズ:
[8, 5, 9, 2, 6, 3, 7, 1]
  ↓分割
[8, 5, 9, 2] [6, 3, 7, 1]
  ↓分割
[8, 5] [9, 2] [6, 3] [7, 1]
  ↓分割
[8] [5] [9] [2] [6] [3] [7] [1]

■マージフェーズ（ボトムアップ）:
[8] [5] → [5, 8]
[9] [2] → [2, 9]
[6] [3] → [3, 6]
[7] [1] → [1, 7]
  ↓マージ
[5, 8] [2, 9] → [2, 5, 8, 9]
[3, 6] [1, 7] → [1, 3, 6, 7]
  ↓マージ
[2, 5, 8, 9] [1, 3, 6, 7] → [1, 2, 3, 5, 6, 7, 8, 9]

【マージ処理のポイント】
1. 左右の配列に番兵（INFTY）を追加
2. 両配列の先頭を比較
3. より小さい方を結果配列に追加
4. すべての要素が処理されるまで繰り返す

【なぜO(n log n)か】
- 分割の深さ: log n
- 各レベルでのマージ処理: 全体でO(n)
- 合計: O(n × log n)
```



```
【比較回数の計算例】
n=10の場合、マージソートの比較回数は約n log n ≈ 33〜34回
（実際には若干異なる可能性がある）
*/

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

#define INFTY 2000000000
#define MAX_N 500000

int A[MAX_N];
int L[MAX_N / 2 + 1];
int R[MAX_N / 2 + 1];
long long compare_count = 0;

// leftからmid-1とmidからright-1の2つのソート済み配列をマージ
void merge(int left, int mid, int right) {
    int n1 = mid - left;    // 左側のサイズ
    int n2 = right - mid;   // 右側のサイズ

    // 左側をL配列にコピー
    for (int i = 0; i < n1; i++) {
        L[i] = A[left + i];
    }

    // 右側をR配列にコピー
    for (int i = 0; i < n2; i++) {
        R[i] = A[mid + i];
    }

    // 番兵を追加（これにより範囲チェックが不要になる）
    L[n1] = INFTY;
    R[n2] = INFTY;

    int i = 0;  // L配列のインデックス
    int j = 0;  // R配列のインデックス

    // 2つのソート済み配列をマージ
    for (int k = left; k < right; k++) {
        compare_count++;  // 比較1回を記録

        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

// マージソート本体
void mergeSort(int left, int right) {
    // 要素が2個以上ある場合のみ分割
    if (left + 1 < right) {
        int mid = (left + right) / 2;

        // 左側をソート
        mergeSort(left, mid);

        // 右側をソート
```



```
        mergeSort(mid, right);

        // ソート済みの左右をマージ
        merge(left, mid, right);
    }
}

int main(void) {
    int n;
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        scanf("%d", &A[i]);
    }

    // マージソート実行
    mergeSort(0, n);

    // ソート結果の出力
    for (int i = 0; i < n; i++) {
        if (i > 0) printf(" ");
        printf("%d", A[i]);
    }
    printf("\n");

    // 比較回数の出力
    printf("%lld\n", compare_count);

    return 0;
}



----------------------------------------
FILE: ALDS1_5_C.c
----------------------------------------
/*
コッホ曲線
整数
 を入力し、深さ
 の再帰呼び出しによって作成されるコッホ曲線の頂点の座標を出力するプログラムを作成してください。

コッホ曲線はフラクタルの一種として知られています。フラクタルとは再帰的な構造を持つ図形のことで、以下のように再帰的な関数の呼び出しを用いて描画することができます。

与えられた線分
 を 3 等分する。
線分を 3等分する 2 点
 を頂点とする正三角形
 を作成する。
線分
、線分
、線分
、線分
に対して再帰的に同じ操作を繰り返す。

 を端点とします。
n
入力
1 つの整数
 が与えられます。

出力
```

```
コッホ曲線の各頂点の座標
 を出力してください。1 行に 1 点の座標を出力してください。端点の 1 つ
 から開始し、一方の端点
 で終えるひとと続きの線分の列となる順番に出力してください。出力は 0.0001 以下の誤差を含んでいてもよいものとします。

制約
入力例 1
1
出力例 1
0.00000000 0.00000000
33.33333333 0.00000000
50.00000000 28.86751346
66.66666667 0.00000000
100.00000000 0.00000000
入力例 2
2
出力例 2
0.00000000 0.00000000
11.11111111 0.00000000
16.66666667 9.62250449
22.22222222 0.00000000
33.33333333 0.00000000
38.88888889 9.62250449
33.33333333 19.24500897
44.44444444 19.24500897
50.00000000 28.86751346
55.55555556 19.24500897
66.66666667 19.24500897
61.11111111 9.62250449
66.66666667 0.00000000
77.77777778 0.00000000
83.33333333 9.62250449
88.88888889 0.00000000
100.00000000 0.00000000
*/

/*
コッホ曲線（フラクタル図形）の解説

【フラクタルとは】
自己相似な図形を再帰的に生成する構造
コッホ曲線は「線分を3等分して、中央部分に正三角形の頂部を追加」する操作を繰り返す

【基本操作】
与えられた線分 P1→P2 に対して:
1. 線分を3等分する点を P1' と P2' とする
   P1' = P1 + (P2 - P1) / 3
   P2' = P1 + 2 * (P2 - P1) / 3

2. P1' と P2' の上に正三角形を作る
   → 頂点 P3 を計算（60度回転）

3. 4つの新しい線分を生成:
   P1 → P1' → P3 → P2' → P2

4. 再帰的に各線分に同じ操作を繰り返す

【座標計算】
P1' = P1 + (P2 - P1) / 3
P2' = P1 + 2 * (P2 - P1) / 3

P3は P1' と P2' 上の正三角形の頂点:
```



```
dx = P2'.x - P1'.x
dy = P2'.y - P1'.y
P3.x = P1'.x + (dx - sqrt(3)*dy) / 2
P3.y = P1'.y + (dx*sqrt(3) + dy) / 2

【具体例】 n=1, 初期線分 (0,0)→(100,0)
深さ1の操作:
P1' = (33.33, 0)
P2' = (66.67, 0)
P3 = (50, 28.87)
出力: (0,0) → (33.33,0) → (50,28.87) → (66.67,0) → (100,0)

【再帰の終了条件】
深さが0になったら、線分 P1→P2 をそのまま出力

【出力形式】
全ての頂点を繋ぐ順番に1行ずつ出力
*/

#include <stdio.h>
#include <math.h>

// 2次元の点を表す構造体
typedef struct {
    double x;
    double y;
} Point;

// コッホ曲線を再帰的に描画
void koch(Point p1, Point p2, int depth) {
    if (depth == 0) {
        // 深さ0のとき、線分の終点を出力
        printf("%.8f %.8f\n", p2.x, p2.y);
        return;
    }

    // 線分を3等分する点を計算
    Point p1_new, p2_new, p3;

    // 線分の1/3の点
    p1_new.x = p1.x + (p2.x - p1.x) / 3.0;
    p1_new.y = p1.y + (p2.y - p1.y) / 3.0;

    // 線分の2/3の点
    p2_new.x = p1.x + 2.0 * (p2.x - p1.x) / 3.0;
    p2_new.y = p1.y + 2.0 * (p2.y - p1.y) / 3.0;

    // p1_new と p2_new 上に正三角形を作る
    // 中央の頂点 p3 を計算（60度回転）
    double dx = p2_new.x - p1_new.x;
    double dy = p2_new.y - p1_new.y;

    p3.x = p1_new.x + (dx - sqrt(3.0) * dy) / 2.0;
    p3.y = p1_new.y + (dx * sqrt(3.0) + dy) / 2.0;

    // 再帰的に4つの線分を処理
    // P1 → P1' → P3 → P2' → P2
    koch(p1, p1_new, depth - 1);
    koch(p1_new, p3, depth - 1);
    koch(p3, p2_new, depth - 1);
    koch(p2_new, p2, depth - 1);
}
```



```

int main(void) {
    int n;
    scanf("%d", &n);

    // 初期線分の開始点を出力
    Point start = {0.0, 0.0};
    Point end = {100.0, 0.0};

    printf("%.8f %.8f\n", start.x, start.y);

    // コッホ曲線を描画（深さn）
    koch(start, end, n);

    return 0;
}



------------------------------------------
FILE: ALDS1_5_D.c
------------------------------------------
/*
反転数

数列
 について、
 かつ
 である組
 の個数を反転数と言います。反転数は次のバブルソートの交換回数と等しくなります。

bubbleSort(A)
  cnt = 0 // 反転数
  for i = 0 to A.length-1
    for j = A.length-1 downto i+1
      if A[j] < A[j-1]
    swap(A[j], A[j-1])
    cnt++

  return cnt
数列
 が与えられるので、
 の反転数を求めてください。上の疑似コードのアルゴリズムをそのまま実装するとTime Limit Exceeded になることに注意してください。

入力
1 行目に数列
 の長さ
 が与えられます。2 行目に
 が空白区切りで与えられます。

出力
反転数を 1 行に出力してください。

制約
 はすべて異なる値である
入力例 1
5
3 5 2 1 4
出力例 1
6
入力例 2
3
```



```
3 1 2
出力例 2
2
*/

/*
反転数をマージソートで効率的に計算する解説

【反転数とは】
i < j かつ A[i] > A[j] である組(i, j)の個数
→ ソートに必要な交換回数と等しい

【なぜバブルソートではTLEか】
バブルソートのアルゴリズム:
  for i = 0 to n-1
    for j = n-1 downto i+1
      if A[j] < A[j-1]
        swap

時間計算量: O(n²)
n=100000の場合、約10^10回の操作で間に合わない

【マージソートで効率化】
マージソートはO(n log n)で、その過程で反転数を計算できる

■分割フェーズで反転数をカウント:
マージ時、左配列の要素が右配列の要素より大きい場合、
左配列の残り全ての要素は反転を形成

■具体例: [3, 5, 2, 1, 4]
分割: [3, 5] [2, 1, 4]

[3, 5] → 反転数 1 (3>5でない, 5>3で反転1)
[2, 1, 4] → [2, 1] [4]
  [2, 1] → 反転数 1 (2>1)
  [4] → 反転数 0
  マージ [2, 1] と [4]: 反転数 0
  結果 [1, 2, 4]

マージ [3, 5] と [1, 2, 4]:
3 vs 1: 1 < 3 → 反転数 +2 (3,5)
2 vs 3: 2 < 3 → 反転数 +1 (3)
...

【マージでの反転数計算】
左配列のインデックスi、右配列のインデックスjで比較:
if (L[i] > R[j]) → 反転数 += (n1 - i)
  理由: i以降の左配列すべての要素がR[j]より大きい

【具体例】
[3, 5] と [2]
3 > 2: 反転数 += (2-0) = 2
*/

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

#define MAX_N 500000
#define INFTY 2000000001

int A[MAX_N];
int L[MAX_N / 2 + 1];
```

```
    int R[MAX_N / 2 + 1];
    long long inv_count = 0;

    // マージしながら反転数をカウント
    void merge(int left, int mid, int right) {
        int n1 = mid - left;      // 左側のサイズ
        int n2 = right - mid;     // 右側のサイズ

        // 左側をL配列にコピー
        for (int i = 0; i < n1; i++) {
            L[i] = A[left + i];
        }

        // 右側をR配列にコピー
        for (int i = 0; i < n2; i++) {
            R[i] = A[mid + i];
        }

        // 番兵を追加
        L[n1] = INFTY;
        R[n2] = INFTY;

        int i = 0;  // L配列のインデックス
        int j = 0;  // R配列のインデックス

        // マージ
        for (int k = left; k < right; k++) {
            if (L[i] <= R[j]) {
                A[k] = L[i];
                i++;
            } else {
                A[k] = R[j];
                j++;

                // 反転数を追加
                // L[i]以降の要素がすべてR[j]より大きいので、
                // n1-i個の反転が存在する
                inv_count += (long long)(n1 - i);
            }
        }
    }

    // マージソート（反転数を記録）
    void mergeSort(int left, int right) {
        if (left + 1 < right) {
            int mid = (left + right) / 2;

            // 左側をソート
            mergeSort(left, mid);

            // 右側をソート
            mergeSort(mid, right);

            // ソート済みの左右をマージ（反転数をカウント）
            merge(left, mid, right);
        }
    }

    int main(void) {
        int n;
        scanf("%d", &n);
```



```
        for (int i = 0; i < n; i++) {
            scanf("%d", &A[i]);
        }

        // マージソート実行（反転数をカウント）
        mergeSort(0, n);

        // 反転数を出力
        printf("%lld\n", inv_count);

        return 0;
    }
```

FILE: ALDS1_6_A.c

```
/*
計数ソート
計数ソートは各要素が 0 以上
 以下である要素数
 の数列に対して線形時間(
)で動く安定なソーティングアルゴリズムです。

入力数列
 の各要素
 について、
 以下の要素の数をカウンタ配列
 に記録し、その値を基に出力配列
 における
 の位置を求めます。同じ数の要素が複数ある場合を考慮して、要素
 を出力（
 に入れる）した後にカウンタ
 は修正する必要があります。詳しくは以下の疑似コードを参考にしてください。

1 CountingSort(A, B, k)
2     for i = 0 to k
3         C[i] = 0
4
5     // C[i] に i の出現数を記録する
6     for j = 1 to n
7         C[A[j]]++
8
9     // C[i] に i 以下の数の出現数を記録する
10    for i = 1 to k
11        C[i] = C[i] + C[i-1]
12
13    for j = n downto 1
14        B[C[A[j]]] = A[j]
15        C[A[j]]--
数列
 を読み込み、計数ソートのアルゴリズムで昇順に並び替え出力するプログラムを作成してください。上記疑似コードに従ってアルゴリズムを実装してください。

入力
入力の最初の行に、数列
 の長さを表す整数
 が与えられます。2 行目に、
 個の整数が空白区切りで与えられます。

出力
整列された数列を1行に出力してください。数列の連続する要素は 1 つの空白で区切って出力してください。
```



```
制約
入力例 1
7
2 5 1 3 2 3 0
出力例 1
0 1 2 2 3 3 5
*/

/*
計数ソートの解説

【計数ソートとは】
整数のみに対して使える特殊なソートアルゴリズム
時間計算量: O(n + k) - 要素数nと最大値kに依存
安定ソート: 同じ値の要素の相対順序が保たれる

【なぜ高速か】
比較ソート（クイックソート、マージソートなど）の限界はO(n log n)
しかし計数ソートは「比較」を行わず、各値の出現位置を直接計算

【アルゴリズムの流れ】

1. C配列で出現回数をカウント
   例: A = [2, 5, 1, 3, 2, 3, 0]
   C = [1, 1, 2, 2, 0, 1] (C[i]はiの出現回数)

2. C配列を累積和に変換
   各値より小さい値がいくつあるか記録
   C = [1, 2, 4, 6, 6, 7] (C[i]はi以下の値の個数)

3. 後ろから前へ走査して出力位置を確定
   - A[n] = A[7] = 0 → C[0]=1 → B[1]=0, C[0]--
   - A[6] = A[6] = 3 → C[3]=6 → B[6]=3, C[3]--
   - ... (後ろから処理することで安定性を実現)

【具体例】A = [2, 5, 1, 3, 2, 3, 0]

■ステップ1: 出現回数をカウント
C配列: [1, 1, 2, 2, 0, 1]
  0は1回、1は1回、2は2回、3は2回、4は0回、5は1回

■ステップ2: 累積和に変換
C配列: [1, 2, 4, 6, 6, 7]
  0以下: 1個、1以下: 2個、2以下: 4個、3以下: 6個、...

■ステップ3: 後ろから処理（安定性のため）
j=7: A[7]=0, B[C[0]]=B[1]=0, C[0]--
j=6: A[6]=3, B[C[3]]=B[6]=3, C[3]--
j=5: A[5]=3, B[C[3]]=B[5]=3, C[3]--
j=4: A[4]=2, B[C[2]]=B[4]=2, C[2]--
j=3: A[3]=2, B[C[2]]=B[3]=2, C[2]--
j=2: A[2]=1, B[C[1]]=B[2]=1, C[1]--
j=1: A[1]=5, B[C[5]]=B[7]=5, C[5]--

結果: B = [_, 0, 1, 2, 2, 3, 3, 5]

【なぜ後ろから処理するのか】
同じ値が複数ある場合、後ろから処理することで
元の配列での相対順序を保つ（安定性）

【制約】
```



```
- 値が0以上kの整数のみ対応
- 値が非常に大きいと、C配列が巨大になり不効率
*/

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

#define MAX_N 100001
#define MAX_K 100001

int main(void) {
    int n;
    scanf("%d", &n);

    int A[n + 1];
    int k = 0;  // 最大値を記録

    for (int i = 1; i <= n; i++) {
        scanf("%d", &A[i]);
        if (A[i] > k) {
            k = A[i];
        }
    }

    // C配列: 出現回数と累積和を記録
    int C[k + 1];
    memset(C, 0, sizeof(C));

    // ステップ1: 各値の出現回数をカウント
    for (int j = 1; j <= n; j++) {
        C[A[j]]++;
    }

    // ステップ2: 累積和に変換（i以下の値の個数）
    for (int i = 1; i <= k; i++) {
        C[i] = C[i] + C[i - 1];
    }

    // B配列: 出力結果を格納
    int B[n + 1];

    // ステップ3: 後ろから処理（安定性を保つ）
    for (int j = n; j >= 1; j--) {
        B[C[A[j]]] = A[j];
        C[A[j]]--;
    }

    // 出力
    for (int i = 1; i <= n; i++) {
        if (i > 1) printf(" ");
        printf("%d", B[i]);
    }
    printf("\n");

    return 0;
}
```

FILE: ALDS1_6_B.c

```
/*
Partition
partition ( A, p, r )は、配列 A[ p..r ] を A[ p..q − 1] の各要素が A[q] 以下で、A[ q +1.. r ] の各要素が A[ q ] より大きい A[ p..q − 1] と A[q + 1..r ] に分割し、インデックス q を戻り値として返します。

数列
 を読み込み、次の疑似コードに基づいた partition を行うプログラムを作成してください。

1 partition(A, p, r)
2   x = A[r]
3   i = p−1
4   for j = p to r−1
5     if A[j] <= x
6       i = i+1
7       A[i] と A[j] を交換
8   A[i+1] と A[r] を交換
9   return i+1
ここで、
 は配列
 の最後の要素を指す添え字で、
 を基準として配列を分割することに注意してください。

入力
入力の最初の行に、数列
 の長さを表す整数
 が与えられます。2 行目に、
 個の整数
 (
) が空白区切りで与えられます。

出力
分割された数列を1行に出力してください。数列の連続する要素は 1 つの空白で区切って出力してください。また、partition の基準となる要素を 【  】で示してください。

制約
入力例 1
12
13 19 9 5 12 8 7 4 21 2 6 11
出力例 1
9 5 8 7 4 2 6 [11] 21 13 19 12
*/

/*
Partition（分割）の解説

【Partitionとは】
クイックソートの基本となるアルゴリズム
基準値（ピボット）よりも小さい要素と大きい要素に分割

【基本的な考え方】
- 最後の要素を基準値xとして選択
- 2つのポインタ（iとj）を使用して分割
  - i: 基準値以下の要素がある領域の末端
  - j: スキャン中の位置

【Partitionのアルゴリズム】

■最初の状態
配列: [13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11]
基準値x = 11（最後の要素）
i = -1（初期値はp-1）

■ステップごとの処理
j=0: A[0]=13 > 11 → スキップ
```



```
j=1: A[1]=19 > 11 → スキップ
j=2: A[2]=9 <= 11 → i=0, swap(A[0], A[2]): [9, 19, 13, 5, 12, 8, 7, 4, 21, 2, 6, 11]
j=3: A[3]=5 <= 11 → i=1, swap(A[1], A[3]): [9, 5, 13, 19, 12, 8, 7, 4, 21, 2, 6, 11]
j=4: A[4]=12 > 11 → スキップ
j=5: A[5]=8 <= 11 → i=2, swap(A[2], A[5]): [9, 5, 8, 19, 12, 13, 7, 4, 21, 2, 6, 11]
j=6: A[6]=7 <= 11 → i=3, swap(A[3], A[6]): [9, 5, 8, 7, 12, 13, 19, 4, 21, 2, 6, 11]
j=7: A[7]=4 <= 11 → i=4, swap(A[4], A[7]): [9, 5, 8, 7, 4, 13, 19, 12, 21, 2, 6, 11]
j=8: A[8]=21 > 11 → スキップ
j=9: A[9]=2 <= 11 → i=5, swap(A[5], A[9]): [9, 5, 8, 7, 4, 2, 19, 12, 21, 13, 6, 11]
j=10: A[10]=6 <= 11 → i=6, swap(A[6], A[10]): [9, 5, 8, 7, 4, 2, 6, 12, 21, 13, 19, 11]

■最後にピボット配置
swap(A[i+1], A[r]): swap(A[7], A[11])
結果: [9, 5, 8, 7, 4, 2, 6, 11, 21, 13, 19, 12]
      基準値11は位置7に確定

【重要なポイント】
1. iはA[j] <= xの要素が格納される領域を管理
2. i+1がピボットの最終位置
3. 左側：全てxより小さい
4. 右側：全てxより大きい

【計算量】
時間計算量: O(n) − 配列を1回走査
空間計算量: O(1) − インプレース処理
*/

#include <stdio.h>

// partition関数
// 配列Aをp~rの範囲でピボット（A[r]）を基準に分割
// 戻り値：分割後のピボット位置
int partition(int A[], int p, int r) {
    int x = A[r];  // ピボット：最後の要素
    int i = p − 1; // 基準値以下の要素がある領域の末端

    // j を p から r−1 まで走査
    for (int j = p; j < r; j++) {
        // A[j] が基準値以下なら、i+1の位置に移動
        if (A[j] <= x) {
            i = i + 1;
            // A[i] と A[j] を交換
            int temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }

    // ピボットを最終位置に配置
    int temp = A[i + 1];
    A[i + 1] = A[r];
    A[r] = temp;

    return i + 1;  // 分割位置を返す
}

int main(void) {
    int n;
    scanf("%d", &n);

    int A[n];
    for (int i = 0; i < n; i++) {
```



```
        scanf("%d", &A[i]);
    }

    // partition を実行（インデックス0〜n−1で分割）
    int q = partition(A, 0, n − 1);

    // 分割された結果を出力
    for (int i = 0; i < n; i++) {
        if (i > 0) printf(" ");

        if (i == q) {
            // ピボット位置は[]で囲む
            printf("[%d]", A[i]);
        } else {
            printf("%d", A[i]);
        }
    }
    printf("\n");

    return 0;
}


---------------------------------------------
FILE: ALDS1_6_C.c
---------------------------------------------
/*
クイックソート
 枚のカードの列を整列します。1 枚のカードは絵柄(S, H, C, またはD)と数字のペアで構成されています。これらを以下の疑似コードに基づくクイックソートで数字に関して昇順に整列するプログラムを作成してください。partition は ALDS1_6_B の疑

1 quicksort(A, p, r)
2   if p < r
3     q = partition(A, p, r)
4     quickSort(A, p, q−1)
5     quickSort(A, q+1, r)
ここで、
 はカードが格納された配列であり、partition における比較演算はカードに書かれた「数」を基準に行われるものとします。

また、与えられた入力に対して安定な出力を行っているかを報告してください。ここでは、同じ数字を持つカードが複数ある場合、それらが入力で与えられた順序であらわれる出力を「安定な出力」とします。

入力
1行目にカードの枚数
 が与えられます。

2行目以降で
 枚のカードが与えられます。各カードは絵柄を表す 1 つの文字と数（整数）のペアで 1 行に与えられます。絵柄と数は 1 つの空白で区切られています。

出力
1行目に、この出力が安定か否か（StableまたはNot stable）を出力してください。

2行目以降で、入力と同様の形式で整列されたカードを順番に出力してください（
 を出力する必要はありません）。

制約
カードに書かれている数
入力に絵柄と数の組が同じカードは 2 枚以上含まれない
入力例 1
6
D 3
H 2
D 1
S 3
```



```
D 2
C 1
出力例 1
Not stable
D 1
C 1
D 2
H 2
D 3
S 3
入力例 2
2
S 1
H 1
出力例 2
Stable
S 1
H 1
*/

/*
クイックソートの解説

【クイックソートとは】
分割統治型のソートアルゴリズム
平均時間計算量: O(n log n)
最悪時間計算量: O(n²) − ピボット選択が悪い場合

【安定性について】
クイックソートは「不安定なソート」
理由: partition で要素が大きく移動するため、
      同じ値の要素の相対順序が保証されない

【安定性の判定ロジック】
ソート後に隣接する要素をチェック：
- 同じ値のカードペアで、元のid が逆順 → "Not stable"
- 元の入力順がずっと保たれている → "Stable"

例: (H,2)がid=1, (D,2)がid=3 の場合
  入力順: id1 (H,2) → id3 (D,2)
  ソート後: id3 (D,2) → id1 (H,2)  ← 逆転 → Not stable

【クイックソートの流れ】
1. 最後の要素をピボットとして選択
2. partition で分割
3. 左右を再帰的にソート

【具体例: D3, H2, D1, S3, D2, C1】
各カードに元のid を付与：
D3(id=0), H2(id=1), D1(id=2), S3(id=3), D2(id=4), C1(id=5)

ソート結果：
D1(id=2), C1(id=5), D2(id=4), H2(id=1), D3(id=0), S3(id=3)

ソート後、同じ値の隣接ペアをチェック：
- D1とC1: 値が異なる
- C1とD2: 値が異なる
- D2とH2: 値が異なる
- H2とD3: 値が異なる
- D3とS3: 値が異なる

同じ値のペア：なし
```

ただし全体をチェック：
– H2(id=1) と D2(id=4): 値が同じで id順序が入力順と異なる可能性
→ 実装では隣接ペアのみチェック

【実装のポイント】
1. DATA 構造体に c（絵柄）, a（値）, id（元の入力順）を保持
2. partition でカードの値（a フィールド）を基準に比較
3. quick_sort で再帰的に分割・ソート
4. ソート後、隣接する要素で同じ値かつ id が逆順をチェック
*/

```
#include <stdio.h>

// カード構造体
typedef struct {
    char c;     // 絵柄: S, H, C, D
    int a;      // 数字: 1–13
    int id;     // 元の入力順序
} DATA;

// partition関数（ALDS1_6_B に基づく）
// ピボット: a[hi] の値を基準に分割
int partition(DATA *a, int lo, int hi) {
    int i, j;
    int key;
    DATA t;

    key = a[hi].a;  // ピボット値
    i = lo - 1;

    // lo から hi-1 まで走査
    for (j = lo; j < hi; j++) {
        // a[j] の値がピボット以下なら、i+1 の位置に移動
        if (a[j].a <= key) {
            i++;
            t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }

    // ピボットを最終位置に配置
    t = a[++i];
    a[i] = a[hi];
    a[hi] = t;

    return i;  // 分割位置を返す
}

// クイックソート関数
void quick_sort(DATA *a, int lo, int hi) {
    int mi;

    if (lo < hi) {
        mi = partition(a, lo, hi);
        quick_sort(a, lo, mi - 1);  // 左側を再帰的にソート
        quick_sort(a, mi + 1, hi);  // 右側を再帰的にソート
    }
}

DATA a[100002];
```



```
int main(void) {
    int N, i, f;

    scanf("%d", &N);

    // 入力を読み込み
    for (i = 0; i < N; i++) {
        scanf(" %c %d", &a[i].c, &a[i].a);
        a[i].id = i;  // 元の入力順を記録
    }

    // クイックソート実行
    quick_sort(a, 0, N - 1);

    // 安定性判定
    // 同じ値のカード が元のid順 と異なったら不安定
    f = 0;
    for (i = 1; i < N; i++) {
        // 隣接する要素で同じ値かつ元のid が逆順 → 不安定
        if (a[i - 1].a == a[i].a && a[i - 1].id > a[i].id) {
            f = 1;
            break;
        }
    }

    // 出力
    if (f) {
        printf("Not stable\n");
    } else {
        printf("Stable\n");
    }

    for (i = 0; i < N; i++) {
        printf("%c %d\n", a[i].c, a[i].a);
    }

    return 0;
}
```

FILE: ALDS1_6_D.c

```
/*
最小コストソートの解説

【問題の理解】
荷物を交換するのにコストがかかる
交換コスト = 2つの荷物の重さの合計

例: 1 5 3 4 2 を昇順にソート

■重さと値の対応
元の位置: [1:0, 5:1, 3:2, 4:3, 2:4]
（値:位置）

■目標位置
1→0, 2→4, 3→2, 4→3, 5→1
（値→目標位置）

■実装のアプローチ
```


選択ソートを使い、各位置で最小値を探す
交換時、コスト =（現在の値）+（最小値）を加算

【具体例】

位置0: [1, 5, 3, 4, 2]
最小値は1（既に位置0）→ コスト0

位置1: [1, 5, 3, 4, 2]
最小値は2（位置4）
交換: 5 + 2 = 7 → コスト += 7
結果: [1, 2, 3, 4, 5]

位置2: [1, 2, 3, 4, 5]
最小値は3（既に位置2）→ コスト0

位置3: [1, 2, 3, 4, 5]
最小値は4（既に位置3）→ コスト0

位置4: [1, 2, 3, 4, 5]
最小値は5（既に位置4）→ コスト0

合計コスト: 7 ✓

【別の例】
4 3 2 1

位置0: 最小値1（位置3）
交換: 4 + 1 = 5 → [1, 3, 2, 4]

位置1: 最小値2（位置2）
交換: 3 + 2 = 5 → [1, 2, 3, 4]

位置2: 最小値3（位置2）
スキップ

位置3: 最小値4（位置3）
スキップ

合計コスト: 5 + 5 = 10 ✓

【計算量】
時間: O(n²) – 選択ソートの形式
空間: O(1) – インプレース処理

【アルゴリズム】
for i = 0 to n-1:
  j = i で最小値を探す
  if (A[j] != A[i]):
    cost += A[i] + A[j]
    swap(A[i], A[j])
*/

```
#include <stdio.h>

int main(void) {
    int n;
    scanf("%d", &n);

    int A[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &A[i]);
```



```
    }

    long long cost = 0;  // 総コスト

    // 選択ソートで最小交換コストを計算
    for (int i = 0; i < n; i++) {
        // i 以降で最小値を探す
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (A[j] < A[minIdx]) {
                minIdx = j;
            }
        }

        // 最小値が現在位置と異なったら交換
        if (minIdx != i) {
            // 交換コスト = 2つの値の合計
            cost += A[i] + A[minIdx];

            // スワップ
            int temp = A[i];
            A[i] = A[minIdx];
            A[minIdx] = temp;
        }
    }

    printf("%lld\n", cost);

    return 0;
}
```

FILE: ALDS1_7_A.c

```
/*
根付き木
与えられた根付き木 $T$ の各節点 $u$ について、以下の情報を出力するプログラムを作成してください。

$u$ の節点番号
$u$ の親の節点番号
$u$ の深さ
$u$ の節点の種類（根、内部節点または葉）
$u$ の子のリスト
ここでは、与えられる木は $n$ 個の節点を持ち、それぞれ $0$ から $n - 1$ の番号が割り当てられているものとします。

入力
入力の最初の行に、節点の個数 $n$ が与えられます。続く $n$ 行目に、各節点の情報が次の形式で 1 行に与えられます。

$id$ $k$ $c_1$ $c_2$ ... $c_k$

$id$ は節点の番号、$k$ は次数を表します。$c_1$ $c_2$ ...$c_k$ は 1 番目の子の節点番号、... $k$ 番目の子の節点番号を示します。

出力
次の形式で節点の情報を出力してください。節点の情報はその番号が小さい順に出力してください。

node $id$: parent = $p$, depth = $d$, $type$, [$c_1$...$c_k$]

$p$ は親の番号を示します。ただし、親を持たない場合は -1 とします。$d$ は節点の深さを示します。

$type$は根、内部節点、葉をそれぞれあらわす root、internal node、leaf の文字列のいずれかです。ただし、根が葉や内部節点の条件に該当する場合は root とします。
```

c_1...c_k は子のリストです。順序木とみなし入力された順に出力してください。カンマ空白区切りに注意してください。出力例にて出力形式を確認してください。

制約
$1 \leq n \leq 100,000$
節点の深さは 20 を超えない。
任意の 2 つの節点間には必ず経路が存在する。
入力例 1
```
13
0 3 1 4 10
1 2 2 3
2 0
3 0
4 3 5 6 7
5 0
6 0
7 2 8 9
8 0
9 0
10 2 11 12
11 0
12 0
```
出力例 1
```
node 0: parent = -1, depth = 0, root, [1, 4, 10]
node 1: parent = 0, depth = 1, internal node, [2, 3]
node 2: parent = 1, depth = 2, leaf, []
node 3: parent = 1, depth = 2, leaf, []
node 4: parent = 0, depth = 1, internal node, [5, 6, 7]
node 5: parent = 4, depth = 2, leaf, []
node 6: parent = 4, depth = 2, leaf, []
node 7: parent = 4, depth = 2, internal node, [8, 9]
node 8: parent = 7, depth = 3, leaf, []
node 9: parent = 7, depth = 3, leaf, []
node 10: parent = 0, depth = 1, internal node, [11, 12]
node 11: parent = 10, depth = 2, leaf, []
node 12: parent = 10, depth = 2, leaf, []
```
入力例 2
```
4
1 3 3 2 0
0 0
3 0
2 0
```
出力例 2
```
node 0: parent = 1, depth = 1, leaf, []
node 1: parent = -1, depth = 0, root, [3, 2, 0]
node 2: parent = 1, depth = 1, leaf, []
node 3: parent = 1, depth = 1, leaf, []
```
参考文献
```
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
*/

#include <stdio.h>
#include <stdlib.h>

#define MAX_N 100000

typedef struct Node {
    int parent;
    int depth;
    int *children;  // 動的配列へのポインタ
    int num_children;
} Node;
```



```
void setDepth(Node nodes[], int id, int depth) {
    nodes[id].depth = depth;
    for (int i = 0; i < nodes[id].num_children; i++) {
        setDepth(nodes, nodes[id].children[i], depth + 1);
    }
}

int main(){
    int n;
    scanf("%d", &n);

    Node nodes[n];

    // 配列の初期化
    for (int i = 0; i < n; i++) {
        nodes[i].parent = -1;
        nodes[i].depth = 0;
        nodes[i].num_children = 0;
        nodes[i].children = NULL;
    }

    int id, k;
    for(int i = 0;i < n;i++){
        scanf("%d %d", &id, &k);
        nodes[id].num_children = k;
        if (k > 0) {
            nodes[id].children = (int*)malloc(sizeof(int) * k);
        }
        for(int j = 0;j < k;j++){
            int child;
            scanf("%d", &child);
            nodes[id].children[j] = child;
            nodes[child].parent = id;
        }
    }

    // ルートノードを見つけて深さを設定
    int root = -1;
    for (int i = 0; i < n; i++) {
        if (nodes[i].parent == -1) {
            root = i;
            break;
        }
    }
    setDepth(nodes, root, 0);

    for(int i = 0;i < n;i++){
        printf("node %d: parent = %d, depth = %d, ", i, nodes[i].parent, nodes[i].depth);
        if(nodes[i].parent == -1)
            printf("root, [");
        else if(nodes[i].num_children == 0)
            printf("leaf, [");
        else
            printf("internal node, [");

        for(int j = 0;j < nodes[i].num_children;j++){
            if(j > 0)
                printf(", ");
            printf("%d", nodes[i].children[j]);
        }
        printf("]\n");
    }
```



```
    // メモリ解放
    for (int i = 0; i < n; i++) {
        if (nodes[i].children != NULL) {
            free(nodes[i].children);
        }
    }

    return 0;
}


---------------------------------------------
FILE: ALDS1_7_B.c
---------------------------------------------
/*
二分木
与えられた二分木
 の各節点
 について、以下の情報を出力するプログラムを作成してください。

 の節点番号
 の親
 の兄弟
 の子の数
 の深さ
 の高さ
節点の種類（根、内部節点または葉）
ここでは、与えられる二分木は
 個の節点を持ち、それぞれ
 から
 の番号が割り当てられているものとします。

入力
入力の最初の行に、節点の個数
 が与えられます。続く
 行目に、各節点の情報が以下の形式で 1 行に与えられます。




 は節点の番号、
 は左の子の番号、
 は右の子の番号を表します。子を持たない場合は
 (
)は -1 で与えられます。

出力
次の形式で節点の情報を出力してください。

node
: parent =
, sibling =
, degree =
, depth =
, height =
,


 は親の番号を表します。親を持たない場合は -1 とします。
 は兄弟の番号を表します。兄弟を持たない場合は -1 とします。
```



```
`
`
 はそれぞれ節点の子の数、深さ、高さを表します。

 は根、内部節点、葉をそれぞれ表す root、internal node、leaf の文字列のいずれかです。ただし、根が葉や内部節点の条件に該当する場合は root とします。

出力例にて、空白区切り等の出力形式を確認してください。

制約
入力例 1
9
0 1 4
1 2 3
2 -1 -1
3 -1 -1
4 5 8
5 6 7
6 -1 -1
7 -1 -1
8 -1 -1
出力例 1
node 0: parent = -1, sibling = -1, degree = 2, depth = 0, height = 3, root
node 1: parent = 0, sibling = 4, degree = 2, depth = 1, height = 1, internal node
node 2: parent = 1, sibling = 3, degree = 0, depth = 2, height = 0, leaf
node 3: parent = 1, sibling = 2, degree = 0, depth = 2, height = 0, leaf
node 4: parent = 0, sibling = 1, degree = 2, depth = 1, height = 2, internal node
node 5: parent = 4, sibling = 8, degree = 2, depth = 2, height = 1, internal node
node 6: parent = 5, sibling = 7, degree = 0, depth = 3, height = 0, leaf
node 7: parent = 5, sibling = 6, degree = 0, depth = 3, height = 0, leaf
node 8: parent = 4, sibling = 5, degree = 0, depth = 2, height = 0, leaf

参考文献
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
*/

#include <stdio.h>

typedef struct Node {
    int parent;
    int depth;
    int height;
    int left;
    int right;
} Node;

void setDepth(Node nodes[], int id, int depth) {
    if (id == -1) return;
    nodes[id].depth = depth;
    setDepth(nodes, nodes[id].left, depth + 1);
    setDepth(nodes, nodes[id].right, depth + 1);
}

int setHeight(Node nodes[], int id) {
    if (id == -1) return -1;
    int h1 = setHeight(nodes, nodes[id].left);
    int h2 = setHeight(nodes, nodes[id].right);
    int height = (h1 > h2 ? h1 : h2) + 1;
    nodes[id].height = height;
    return height;
}

int main(){
```

```
    int n;
    scanf("%d", &n);

    Node nodes[n];

    // 初期化
    for (int i = 0; i < n; i++) {
        nodes[i].parent = -1;
        nodes[i].depth = 0;
        nodes[i].height = 0;
        nodes[i].left = -1;
        nodes[i].right = -1;
    }

    // 入力読み込み
    for (int i = 0; i < n; i++) {
        int id, left, right;
        scanf("%d %d %d", &id, &left, &right);
        nodes[id].left = left;
        nodes[id].right = right;
        if (left != -1) {
            nodes[left].parent = id;
        }
        if (right != -1) {
            nodes[right].parent = id;
        }
    }

    // ルートを見つける
    int root = -1;
    for (int i = 0; i < n; i++) {
        if (nodes[i].parent == -1) {
            root = i;
            break;
        }
    }

    // 深さと高さを計算
    setDepth(nodes, root, 0);
    setHeight(nodes, root);

    // 出力
    for (int i = 0; i < n; i++) {
        // 兄弟を見つける
        int sibling = -1;
        if (nodes[i].parent != -1) {
            int p = nodes[i].parent;
            if (nodes[p].left == i) {
                sibling = nodes[p].right;
            } else {
                sibling = nodes[p].left;
            }
        }

        // 子の数 (degree)
        int degree = 0;
        if (nodes[i].left != -1) degree++;
        if (nodes[i].right != -1) degree++;

        // タイプ
        const char *type;
        if (nodes[i].parent == -1) {
```



```
            type = "root";
        } else if (degree == 0) {
            type = "leaf";
        } else {
            type = "internal node";
        }

        printf("node %d: parent = %d, sibling = %d, degree = %d, depth = %d, height = %d, %s\n",
               i, nodes[i].parent, sibling, degree, nodes[i].depth, nodes[i].height, type);
    }

    return 0;
}


----------------------------------------
FILE: ALDS1_7_C.c
----------------------------------------
/*
木の巡回
以下に示すアルゴリズムで、与えられた２分木のすべての節点を体系的に訪問するプログラムを作成してください。

根節点、左部分木、右部分木の順で節点の番号を出力する。これを木の先行順巡回 (preorder tree walk) と呼びます。
左部分木、根節点、右部分木の順で節点の番号を出力する。これを木の中間順巡回 (inorder tree walk) と呼びます。
左部分木、右部分木、根節点の順で節点の番号を出力する。これを木の後行順巡回 (postorder tree walk) と呼びます。
与えられる２分木は
 個の節点を持ち、それぞれ
 から
 の番号が割り当てられているものとします。

入力
入力の最初の行に、節点の個数
 が与えられます。続く
 行目に、各節点の情報が以下の形式で１行に与えられます。



 は節点の番号、
 は左の子の番号、
 は右の子の番号を表します。子を持たない場合は
 (
) は -1 で与えられます。

出力
１行目に"Preorder"と出力し、２行目に先行順巡回を行った節点番号を順番に出力してください。

３行目に"Inorder"と出力し、４行目に中間順巡回を行った節点番号を順番に出力してください。

５行目に"Postorder"と出力し、６行目に後行順巡回を行った節点番号を順番に出力してください。

節点番号の前に１つの空白文字を出力してください。

制約
入力例 1
9
0 1 4
1 2 3
2 -1 -1
3 -1 -1
4 5 8
5 6 7
```



```
6 -1 -1
7 -1 -1
8 -1 -1
出力例 1
Preorder
 0 1 2 3 4 5 6 7 8
Inorder
 2 1 3 0 6 5 7 4 8
Postorder
 2 3 1 6 7 5 8 4 0

参考文献
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
*/

#include <stdio.h>

typedef struct Node {
    int parent;
    int depth;
    int height;
    int left;
    int right;
} Node;

void setDepth(Node nodes[], int id, int depth) {
    if (id == -1) return;
    nodes[id].depth = depth;
    setDepth(nodes, nodes[id].left, depth + 1);
    setDepth(nodes, nodes[id].right, depth + 1);
}

int setHeight(Node nodes[], int id) {
    if (id == -1) return -1;
    int h1 = setHeight(nodes, nodes[id].left);
    int h2 = setHeight(nodes, nodes[id].right);
    int height = (h1 > h2 ? h1 : h2) + 1;
    nodes[id].height = height;
    return height;
}

int preorder(Node nodes[], int id) {
    if (id == -1) return 0;
    printf(" %d", id);
    preorder(nodes, nodes[id].left);
    preorder(nodes, nodes[id].right);
    return 0;
}

int inorder(Node nodes[], int id) {
    if (id == -1) return 0;
    inorder(nodes, nodes[id].left);
    printf(" %d", id);
    inorder(nodes, nodes[id].right);
    return 0;
}

int postorder(Node nodes[], int id) {
    if (id == -1) return 0;
    postorder(nodes, nodes[id].left);
    postorder(nodes, nodes[id].right);
    printf(" %d", id);
```



```
    return 0;
}

int main(){
    int n;
    scanf("%d", &n);

    Node nodes[n];

    // 初期化
    for (int i = 0; i < n; i++) {
        nodes[i].parent = -1;
        nodes[i].depth = 0;
        nodes[i].height = 0;
        nodes[i].left = -1;
        nodes[i].right = -1;
    }

    // 入力読み込み
    for (int i = 0; i < n; i++) {
        int id, left, right;
        scanf("%d %d %d", &id, &left, &right);
        nodes[id].left = left;
        nodes[id].right = right;
        if (left != -1) {
            nodes[left].parent = id;
        }
        if (right != -1) {
            nodes[right].parent = id;
        }
    }

    // ルートを見つける
    int root = -1;
    for (int i = 0; i < n; i++) {
        if (nodes[i].parent == -1) {
            root = i;
            break;
        }
    }

    // 深さと高さを計算
    setDepth(nodes, root, 0);
    setHeight(nodes, root);

    // 出力
    printf("Preorder\n");
    preorder(nodes, root);
    printf("\n");
    printf("Inorder\n");
    inorder(nodes, root);
    printf("\n");
    printf("Postorder\n");
    postorder(nodes, root);
    printf("\n");

    /*
    // 出力
    for (int i = 0; i < n; i++) {
        // 兄弟を見つける
        int sibling = -1;
        if (nodes[i].parent != -1) {
```

```
            int p = nodes[i].parent;
            if (nodes[p].left == i) {
                sibling = nodes[p].right;
            } else {
                sibling = nodes[p].left;
            }
        }

        // 子の数 (degree)
        int degree = 0;
        if (nodes[i].left != -1) degree++;
        if (nodes[i].right != -1) degree++;

        // タイプ
        const char *type;
        if (nodes[i].parent == -1) {
            type = "root";
        } else if (degree == 0) {
            type = "leaf";
        } else {
            type = "internal node";
        }

        printf("node %d: parent = %d, sibling = %d, degree = %d, depth = %d, height = %d, %s\n",
               i, nodes[i].parent, sibling, degree, nodes[i].depth, nodes[i].height, type);
    }
    */

    return 0;
}


----------------------------------------------
FILE: ALDS1_7_D.c
----------------------------------------------
/*
Reconstruction of a Tree
ある二分木に対して、それぞれ先行順巡回 (preorder tree walk) と中間順巡回 (inorder tree walk) を行って得られる節点の列が与えられるので、その二分木の後行順巡回 (postorder tree walk) で得られる節点の列を出力するプログラムを作

入力
１行目に二分木の節点の数
 が与えられます。
２行目に先行順巡回で得られる節点の番号の列が空白区切りで与えられます。
３行目に中間順巡回で得られる節点の番号の列が空白区切りで与えられます。

節点には
 から
 までの整数が割り当てられています。
 が根とは限らないことに注意してください。

出力
後行順巡回で得られる節点の番号の列を１行に出力してください。節点の番号の間に１つの空白を入れてください。

制約
入力例 1
5
1 2 3 4 5
3 2 4 1 5
出力例 1
3 4 2 5 1
入力例 2
4
```



```
1 2 3 4
1 2 3 4
出力例 2
4 3 2 1
*/

#include <stdio.h>

#define MAX_N 40

int n;
int preorder[MAX_N];
int inorder[MAX_N];
int postorder[MAX_N];
int post_index = 0;

// inorder配列の中でvalueの位置を探す
int search(int arr[], int start, int end, int value) {
    for (int i = start; i <= end; i++) {
        if (arr[i] == value) {
            return i;
        }
    }
    return -1;
}

// preorderとinorderから木を再構築し、postorderを生成
void reconstruct(int pre_start, int pre_end, int in_start, int in_end) {
    if (pre_start > pre_end || in_start > in_end) {
        return;
    }

    // preorderの最初の要素がルート
    int root = preorder[pre_start];

    // inorder配列でルートの位置を探す
    int root_index = search(inorder, in_start, in_end, root);

    // 左部分木のサイズ
    int left_size = root_index - in_start;

    // 左部分木を再帰的に処理
    reconstruct(pre_start + 1, pre_start + left_size, in_start, root_index - 1);

    // 右部分木を再帰的に処理
    reconstruct(pre_start + left_size + 1, pre_end, root_index + 1, in_end);

    // 後行順なので、左→右→ルートの順で処理
    postorder[post_index++] = root;
}

/*
// 【inorder + postorder から preorder を生成】
// 使う場合は pre_index を用意して、preorder 配列に書き込みます。
void reconstruct_in_post_to_pre(int in_start, int in_end, int post_start, int post_end) {
    if (in_start > in_end || post_start > post_end) {
        return;
    }

    // postorderの最後の要素がルート
    int root = postorder[post_end];
```



```
    // inorder配列でルートの位置を探す
    int root_index = search(inorder, in_start, in_end, root);

    // 左部分木のサイズ
    int left_size = root_index - in_start;

    // preorderはルート→左→右
    preorder[pre_index++] = root;

    // 左部分木を再帰的に処理
    reconstruct_in_post_to_pre(in_start, root_index - 1, post_start, post_start + left_size - 1);

    // 右部分木を再帰的に処理
    reconstruct_in_post_to_pre(root_index + 1, in_end, post_start + left_size, post_end - 1);
}
*/

/*
// 【preorder + postorder から inorder を生成】
// 注意: 一意に決まるのは「完全二分木」が前提の場合のみ
// 使う場合は in_index を用意して、inorder 配列に書き込みます。
void reconstruct_pre_post_to_in(int pre_start, int pre_end, int post_start, int post_end) {
    if (pre_start > pre_end || post_start > post_end) {
        return;
    }

    if (pre_start == pre_end) {
        inorder[in_index++] = preorder[pre_start];
        return;
    }

    int root = preorder[pre_start];
    int left_root = preorder[pre_start + 1];

    // postorderで左部分木の根の位置を探す
    int left_root_index = search(postorder, post_start, post_end, left_root);

    int left_size = left_root_index - post_start + 1;

    // 左部分木
    reconstruct_pre_post_to_in(pre_start + 1, pre_start + left_size, post_start, left_root_index);

    // ルート
    inorder[in_index++] = root;

    // 右部分木
    reconstruct_pre_post_to_in(pre_start + left_size + 1, pre_end, left_root_index + 1, post_end - 1);
}
*/

int main() {
    scanf("%d", &n);
    // preorder読み込み
    for (int i = 0; i < n; i++) {
        scanf("%d", &preorder[i]);
    }

    // inorder読み込み
    for (int i = 0; i < n; i++) {
        scanf("%d", &inorder[i]);
    }
```



```
    // 木を再構築してpostorderを生成
    post_index = 0;
    reconstruct(0, n - 1, 0, n - 1);

    // postorder出力
    for (int i = 0; i < n; i++) {
        if (i > 0) printf(" ");
        printf("%d", postorder[i]);
    }
    printf("\n");
    return 0;
}


----------------------------------------------
FILE: ALDS1_8_A.c
----------------------------------------------
/*
二分探索木I
探索木は、挿入、検索、削除などの操作が行えるデータ構造で、辞書あるいは優先度付きキューとして用いることができます。探索木の中でも最も基本的なものが二分探索木です。二分探索木は、各節点にキーを持ち、次に示す二分探索木条件(Binary search

 を２分探索木に属するある節点とする。
 を
 の左部分木に属する節点とすると、
 のキー
のキーである。また、
 を
 の右部分木に属する節点とすると、
 のキー
 のキーである。
次の図は二分探索木の例です。


例えば、キーが80の節点の左部分木に属する節点のキーは80以下であり、右部分木に属する節点のキーは80以上になっています。二分探索木に中間順巡回を行うと、昇順に並べられたキーの列を得ることができます。

二分探索木は、データの挿入や削除が行われても常にこのような条件が全ての節点で成り立つように実装しなければなりません。リストと同様に、節点をポインタで連結することで木を表し、各節点には値（キー）に加えその親、左の子、右の子へのポインタを持

二分探索木
 に新たに値
 を挿入するには以下の疑似コードに示す insert を実行します。insert は、キーが
、左の子が
、右の子が
 であるような節点
 を受け取り、
 の正しい位置に挿入します。

1 insert(T, z)
2     y = NIL // x の親
3     x = 'T の根'
4     while x ≠ NIL
5         y = x // 親を設定
6         if z.key < x.key
7             x = x.left // 左の子へ移動
8         else
9             x = x.right // 右の子へ移動
10    z.p = y
11
12    if y == NIL // T が空の場合
13        'T の根' = z
14    else if z.key < y.key
```

```
    15          y.left = z // z を y の左の子にする
    16      else
    17          y.right = z // z を y の右の子にする
    二分探索木
     に対し、以下の命令を実行するプログラムを作成してください。

    insert
    :
    にキー
     を挿入する。
    print: キーを木の中間順巡回(inorder tree walk)と先行順巡回(preorder tree walk)アルゴリズムで出力する。
    挿入のアルゴリズムは上記疑似コードに従ってください。

    入力
    入力の最初の行に、命令の数
     が与えられます。続く
     行目に、insert
     または print の形式で命令が 1 行に与えられます。

    出力
    print命令ごとに、中間順巡回アルゴリズム、先行順巡回アルゴリズムによって得られるキーの順列をそれぞれ 1 行に出力してください。各キーの前に 1 つの空白を出力してください。

    制約
    命令の数は
     を超えない。
    print 命令の数は
     を超えない。
    キー
    上記の疑似コードのアルゴリズムに従う場合、木の高さは 100 を超えない。
    二分探索木中のキーに重複は発生しない。
    入力例 1
    8
    insert 30
    insert 88
    insert 12
    insert 1
    insert 20
    insert 17
    insert 25
    print
    出力例 1
     1 12 17 20 25 30 88
     30 12 1 20 17 25 88
    参考文献
    Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
    */

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

    typedef struct Node {
        int key;
        struct Node *parent;
        struct Node *left;
        struct Node *right;
    } Node;

    Node *root = NULL;

    // 新しいノードを作成して初期化する
    Node *create_node(int key) {
```



```
        Node *node = (Node *)malloc(sizeof(Node));
        if (node == NULL) {
            exit(1);
        }
        node->key = key;
        node->parent = NULL;
        node->left = NULL;
        node->right = NULL;
        return node;
    }

    // 疑似コードの insert に従って BST に挿入する
    void insert_node(Node *z) {
        Node *y = NULL;
        Node *x = root;

        // 挿入位置を探索
        while (x != NULL) {
            y = x;
            if (z->key < x->key) {
                x = x->left;
            } else {
                x = x->right;
            }
        }

        // 親を設定し、親の左/右にぶら下げる
        z->parent = y;
        if (y == NULL) {
            root = z;
        } else if (z->key < y->key) {
            y->left = z;
        } else {
            y->right = z;
        }
    }

    // 中間順巡回（昇順に出力される）
    void inorder(Node *u) {
        if (u == NULL) return;
        inorder(u->left);
        printf(" %d", u->key);
        inorder(u->right);
    }

    // 先行順巡回（根→左→右）
    void preorder(Node *u) {
        if (u == NULL) return;
        printf(" %d", u->key);
        preorder(u->left);
        preorder(u->right);
    }

    int main(void) {
        int m;
        if (scanf("%d", &m) != 1) {
            return 0;
        }

        for (int i = 0; i < m; i++) {
            char cmd[10];
            scanf("%s", cmd);
```



```
            if (strcmp(cmd, "insert") == 0) {
                // insert x: x を BST に挿入
                int key;
                scanf("%d", &key);
                Node *z = create_node(key);
                insert_node(z);
            } else if (strcmp(cmd, "print") == 0) {
                // print: 中間順 → 先行順 の順に出力
                inorder(root);
                printf("\n");
                preorder(root);
                printf("\n");
            }
        }

        return 0;
    }



    ---------------------------------------------
    FILE: ALDS1_8_B.c
    ---------------------------------------------
    /*
    二分探索木II
    A: Binary Search Tree I に、find 命令を追加し、二分探索木
     に対し、以下の命令を実行するプログラムを作成してください。

    insert
    :
    にキー
     を挿入する。
    find
    :
    にキー
     が存在するか否かを報告する。
    print: キーを木の中間順巡回(inorder tree walk)と先行順巡回(preorder tree walk)アルゴリズムで出力する。
    入力
    入力の最初の行に、命令の数
     が与えられます。続く
     行目に、insert
    、find
     またはprintの形式で命令が 1 行に与えられます。

    出力
    find
     命令ごとに、
    に
     が含まれる場合 yes と、含まれない場合 no と 1 行に出力してください。

    さらに print 命令ごとに、中間順巡回アルゴリズム、先行順巡回アルゴリズムによって得られるキーの順列をそれぞれ 1 行に出力してください。各キーの前に 1 つの空白を出力してください。

    制約
    命令の数は
     を超えない。
    print 命令の数は
     を超えない。
    キー
    上記の疑似コードのアルゴリズムに従う場合、木の高さは
     を超えない。
    2 分探索木中のキーに重複は発生しない。
```



```
    入力例 1
    10
    insert 30
    insert 88
    insert 12
    insert 1
    insert 20
    find 12
    insert 17
    insert 25
    find 16
    print
    出力例 1
    yes
    no
     1 12 17 20 25 30 88
     30 12 1 20 17 25 88
    参考文献
    Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
    */

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

    typedef struct Node {
        int key;
        struct Node *parent;
        struct Node *left;
        struct Node *right;
    } Node;

    Node *root = NULL;

    Node *create_node(int key) {
        Node *node = (Node *)malloc(sizeof(Node));
        if (node == NULL) {
            exit(1);
        }
        node->key = key;
        node->parent = NULL;
        node->left = NULL;
        node->right = NULL;
        return node;
    }

    void insert_node(Node *z) {
        Node *y = NULL;
        Node *x = root;

        while (x != NULL) {
            y = x;
            if (z->key < x->key) {
                x = x->left;
            } else {
                x = x->right;
            }
        }

        z->parent = y;

        if (y == NULL) {
```

```
            root = z;
        } else if (z->key < y->key) {
            y->left = z;
        } else {
            y->right = z;
        }
    }

    Node *find_node(int key) {
        Node *x = root;

        while (x != NULL && key != x->key) {
            if (key < x->key) {
                x = x->left;
            } else {
                x = x->right;
            }
        }

        return x;
    }

    void inorder(Node *u) {
        if (u == NULL) return;
        inorder(u->left);
        printf(" %d", u->key);
        inorder(u->right);
    }

    void preorder(Node *u) {
        if (u == NULL) return;
        printf(" %d", u->key);
        preorder(u->left);
        preorder(u->right);
    }

    int main(void) {
        int m;
        if (scanf("%d", &m) != 1) {
            return 0;
        }

        for (int i = 0; i < m; i++) {
            char cmd[10];
            scanf("%s", cmd);

            if (strcmp(cmd, "insert") == 0) {
                int key;
                scanf("%d", &key);
                Node *z = create_node(key);
                insert_node(z);
            } else if (strcmp(cmd, "find") == 0) {
                int key;
                scanf("%d", &key);
                Node *result = find_node(key);
                if (result != NULL) {
                    printf("yes\n");
                } else {
                    printf("no\n");
                }
            } else if (strcmp(cmd, "print") == 0) {
                inorder(root);
```



```
                printf("\n");
                preorder(root);
                printf("\n");
            }
        }

        return 0;
    }
```

```
----------------------------------------
FILE: ALDS1_8_C.c
----------------------------------------
```

```
/*
２分探索木III
B: Binary Search Tree II に、delete 命令を追加し、二分探索木
 に対し、以下の命令を実行するプログラムを作成してください。

insert
:
 にキー
 を挿入する。
find
:
 にキー
 が存在するか否かを報告する。
delete
: キー
 を持つ節点を削除する。
print: キーを木の中間順巡回(inorder tree walk)と先行順巡回(preorder tree walk)アルゴリズムで出力する。
二分探索木
 から与えられたキー
 を持つ節点
 を削除する delete k は以下の３つの場合を検討したアルゴリズムに従い、二分探索木条件を保ちつつ親子のリンク（ポインタ）を更新します：

 が子を持たない場合、
 の親
 の子（つまり
) を削除する。
 がちょうど１つの子を持つ場合、
 の親の子を
 の子に変更、
 の子の親を
 の親に変更し、
 を木から削除する。
 が子を２つ持つ場合、
 の次節点
 のキーを
 のキーへコピーし、
 を削除する。
 の削除では 1. または 2. を適用する。ここで、
 の次節点とは、中間順巡回で
 の次に得られる節点である。
入力
入力の最初の行に、命令の数
 が与えられます。続く
 行目に、insert
 . find
 . delete
または print の形式で命令が１行に与えられます。
```



```
出力
find
 命令ごとに、
 に
 が含まれる場合 yes と、含まれない場合 no と１行に出力してください。

さらに print 命令ごとに、中間順巡回アルゴリズム、先行順巡回アルゴリズムによって得られるキーの順列をそれぞれ１行に出力してください。各キーの前に１つの空白を出力してください。

制約
命令の数は
を超えない。
print命令の数は
を超えない。
キー
上記の疑似コードのアルゴリズムに従う場合、木の高さは
 を超えない。
二分探索木中のキーに重複は発生しない。
入力例 1
18
insert 8
insert 2
insert 3
insert 7
insert 22
insert 1
find 1
find 2
find 3
find 4
find 5
find 6
find 7
find 8
print
delete 3
delete 7
print
出力例 1
yes
yes
yes
no
no
no
yes
yes
 1 2 3 7 8 22
 8 2 1 3 7 22
 1 2 8 22
 8 2 1 22
参考文献
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
*/

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

typedef struct Node {
    int key;
    struct Node *parent;
    struct Node *left;
```



```
    struct Node *right;
} Node;

Node *root = NULL;

Node *create_node(int key) {
    Node *node = (Node *)malloc(sizeof(Node));
    if (node == NULL) {
        exit(1);
    }
    node->key = key;
    node->parent = NULL;
    node->left = NULL;
    node->right = NULL;
    return node;
}

void insert_node(Node *z) {
    Node *y = NULL;
    Node *x = root;

    while (x != NULL) {
        y = x;
        if (z->key < x->key) {
            x = x->left;
        } else {
            x = x->right;
        }
    }

    z->parent = y;
    if (y == NULL) {
        root = z;
    } else if (z->key < y->key) {
        y->left = z;
    } else {
        y->right = z;
    }
}

Node *find_node(int key) {
    Node *x = root;
    while (x != NULL && key != x->key) {
        if (key < x->key) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    return x;
}

Node *find_minimum(Node *x) {
    while (x->left != NULL) {
        x = x->left;
    }
    return x;
}

Node *find_successor(Node *x) {
    if (x->right != NULL) {
        return find_minimum(x->right);
```

```
    }
    Node *y = x->parent;
    while (y != NULL && x == y->right) {
        x = y;
        y = y->parent;
    }
    return y;
}

void transplant(Node *u, Node *v) {
    if (u->parent == NULL) {
        root = v;
    } else if (u == u->parent->left) {
        u->parent->left = v;
    } else {
        u->parent->right = v;
    }
    if (v != NULL) {
        v->parent = u->parent;
    }
}

void delete_node(Node *z) {
    if (z->left == NULL) {
        transplant(z, z->right);
    } else if (z->right == NULL) {
        transplant(z, z->left);
    } else {
        Node *y = find_minimum(z->right);
        if (y->parent != z) {
            transplant(y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }
        transplant(z, y);
        y->left = z->left;
        y->left->parent = y;
    }
    free(z);
}

void inorder(Node *u) {
    if (u == NULL) return;
    inorder(u->left);
    printf(" %d", u->key);
    inorder(u->right);
}

void preorder(Node *u) {
    if (u == NULL) return;
    printf(" %d", u->key);
    preorder(u->left);
    preorder(u->right);
}

int main(void) {
    int m;
    if (scanf("%d", &m) != 1) {
        return 0;
    }

    for (int i = 0; i < m; i++) {
```



```
        char cmd[10];
        scanf("%s", cmd);

        if (strcmp(cmd, "insert") == 0) {
            int key;
            scanf("%d", &key);
            Node *z = create_node(key);
            insert_node(z);
        } else if (strcmp(cmd, "find") == 0) {
            int key;
            scanf("%d", &key);
            Node *result = find_node(key);
            if (result != NULL) {
                printf("yes\n");
            } else {
                printf("no\n");
            }
        } else if (strcmp(cmd, "delete") == 0) {
            int key;
            scanf("%d", &key);
            Node *z = find_node(key);
            if (z != NULL) {
                delete_node(z);
            }
        } else if (strcmp(cmd, "print") == 0) {
            inorder(root);
            printf("\n");
            preorder(root);
            printf("\n");
        }
    }

    return 0;
}
```

FILE: ALDS1_8_D.c

```
/*
Treap
二分探索木は、挿入されるデータ列の特徴によっては、偏った木になり、検索・挿入・削除操作の効率が悪くなります。例えば、整列された
個のデータが順番に挿入されれば、木はリストのような形になり、その高さは
になります。挿入されるデータが固定されていれば、要素をランダムにシャッフルすることにより平衡な木を構築することができるでしょう。しかし、データ構造としての二分木では要求に応じて異なる操作が繰り返し行われるため、適宜データを一つずつ処理(

二分木の各節点に、ランダムに選択された優先度を割り当て、以下の条件を満たすように節点を順序付けることによって、平衡な二分木を保つことができます。ここで、各節点のキー(key)と優先度(priority)はそれぞれ重複がないものとします。

二分探索木条件.
が
の左の子なら
 かつ
が
の右の子なら
ヒープ条件.
が
の子なら
このような木を、二分探索木とヒープの特徴からTreap ( tree + heap ) と呼びます。

例えば、次の図はTreapの例です。

挿入
```


Treapに新たにデータを挿入するには、キーに加えランダムに選択した優先度を割り当てた節点を、まずは通常の二分探索木と同様の方法で挿入します。例えば、上のTreapにkey = 6, priority = 90 である節点を挿入すると次のようになります。

このままの状態では、ヒープ条件を破ってしまうため、ヒープ条件を満たすまで回転を繰り返します。回転とは、次の図のように、二分探索木条件を満たしつつ、親子関係を逆転させる操作です。

回転は次のプログラムのようにポインタを繋ぎ変えます。

```
# 節点tを根とする部分木を右に回転する
def rotate_right(t):
    s = t.left
    t.left = s.right
    s.right = t
    return s # 新しい部分木の根となる
# 節点tを根とする部分木を左に回転する
def rotate_left(t):
    s = t.right
    t.right = s.left
    s.left = t
    return s # 新しい部分木の根となる
```

上のTreapに回転操作を行うと以下のように二分探索木が構築されます。

条件を満たすようにTreapに新しい要素を挿入するinsert操作は以下のようになります。

```
# キーがkeyで優先度がpriの節点をTreapに挿入する
# t: 探索で訪問中の節点
def insert(t, key, pri):                        # 再帰的に探索する
    if t is NIL:
        return Node(key, pri)                   # 葉に到達したら新しい節点を生成して返す
    if key == t.key:
        return t                                # 重複したkeyは無視する

    if key < t.key:                             # 左の子へ移動する
        t.left = insert(t.left, key, pri)       # 左の子へのポインタを更新する
        if t.pri < t.left.pri:                  # 左の子の方が優先度が高い場合は右に回転
            t = rotate_right(t)
    else:                                       # 右の子へ移動する
        t.right = insert(t.right, key, pri)     # 右の子へのポインタを更新する
        if t.pri < t.right.pri:                 # 右の子の方が優先度が高い場合は左に回転
            t = rotate_left(t)
    return t
```

削除
Treapの節点を削除する場合は、以下の手順で対象となる節点を回転によって葉まで移動した後、削除します。

```
def search_and_delete(t, key):
    if t is NIL:
        return NIL
    if key < t.key:                             # 削除対象を検索する
        t.left = search_and_delete(t.left, key)
    elif key > t.key:
        t.right = search_and_delete(t.right, key)
    else:                                       # 削除対象を発見
        return delete_target(t, key)
    return t

def delete_target(t, key):
    if t.left is NIL and t.right is NIL:        # 葉の場合
        return NIL
    elif t.left is NIL:                         # 右の子のみを持つ場合は左回転
        t = rotate_left(t)
    elif t.right is NIL:                        # 左の子のみを持つ場合は右回転
```



```
        t = rotate_right(t)
    else:                                       # 左の子と右の子を両方持つ場合
        if t.left.pri > t.right.pri:            # 優先度が高い方を持ち上げる
            t = rotate_right(t)
        else:
            t = rotate_left(t)
    return search_and_delete(t, key)
```

Treap
 に対して、以下の命令を、上記のアルゴリズムに基づいて実行するプログラムを作成してください。

```
insert (
,
):
```

 にキーが
、優先度が
の要素 を挿入する。

```
find (
):
```

 にキー
 が存在するか否かを報告する。

```
delete (
): キー
```

 を持つ節点を削除する。

```
print(): キーを木の中間順巡回(inorder tree walk)と先行順巡回(preorder tree walk)アルゴリズムで出力する。
```

入力
入力の最初の行に、命令の数
 が与えられます。続く
 行に、insert
、find
、delete
 または print の形式で命令が 1 行に与えられます。

出力
find
 命令ごとに、
に
 が含まれる場合 yes と、含まれない場合 no と 1 行に出力してください。

さらに print 命令ごとに、中間順巡回アルゴリズム、先行順巡回アルゴリズムによって得られるキーの順列をそれぞれ 1 行に出力してください。各キーの前に 1 つの空白を出力してください。

制約
命令の数は
を超えない。
上記のアルゴリズムに従う場合、木の高さは
 を超えない。
二分探索木中のキーに重複は発生しない。
二分探索木中の優先度に重複は発生しない。
print命令の数は
を超えない。
出力のサイズは
 Mバイトを超えない。
入力例 1

```
16
insert 35 99
insert 3 80
insert 1 53
insert 14 25
insert 80 76
insert 42 3
insert 86 47
insert 21 12
insert 7 10
```

```
insert 6 90
print
find 21
find 22
delete 35
delete 99
print
出力例 1
 1 3 6 7 14 21 35 42 80 86
 35 6 3 1 14 7 21 80 42 86
yes
no
 1 3 6 7 14 21 42 80 86
 6 3 1 80 14 7 21 42 86
参考文献
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT\ Press.
Randomized Search Trees, R. Seidel, C.R. Aragon.
*/

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

typedef struct Node {
    int key;
    int priority;
    struct Node *left;
    struct Node *right;
} Node;

Node *root = NULL;

Node *create_node(int key, int priority) {
    Node *node = (Node *)malloc(sizeof(Node));
    if (node == NULL) {
        exit(1);
    }
    node->key = key;
    node->priority = priority;
    node->left = NULL;
    node->right = NULL;
    return node;
}

Node *rotate_right(Node *t) {
    Node *s = t->left;
    t->left = s->right;
    s->right = t;
    return s;
}

Node *rotate_left(Node *t) {
    Node *s = t->right;
    t->right = s->left;
    s->left = t;
    return s;
}

Node *insert(Node *t, int key, int priority) {
    if (t == NULL) {
        return create_node(key, priority);
    }
```



```
    if (key == t->key) {
        return t;
    }

    if (key < t->key) {
        t->left = insert(t->left, key, priority);
        if (t->priority < t->left->priority) {
            t = rotate_right(t);
        }
    } else {
        t->right = insert(t->right, key, priority);
        if (t->priority < t->right->priority) {
            t = rotate_left(t);
        }
    }
    return t;
}

Node *find_node(Node *t, int key) {
    if (t == NULL) return NULL;
    if (key == t->key) return t;
    if (key < t->key) {
        return find_node(t->left, key);
    } else {
        return find_node(t->right, key);
    }
}

Node *delete_target(Node *t, int key);

Node *search_and_delete(Node *t, int key) {
    if (t == NULL) {
        return NULL;
    }
    if (key < t->key) {
        t->left = search_and_delete(t->left, key);
    } else if (key > t->key) {
        t->right = search_and_delete(t->right, key);
    } else {
        return delete_target(t, key);
    }
    return t;
}

Node *delete_target(Node *t, int key) {
    if (t->left == NULL && t->right == NULL) {
        free(t);
        return NULL;
    } else if (t->left == NULL) {
        t = rotate_left(t);
    } else if (t->right == NULL) {
        t = rotate_right(t);
    } else {
        if (t->left->priority > t->right->priority) {
            t = rotate_right(t);
        } else {
            t = rotate_left(t);
        }
    }
    return search_and_delete(t, key);
}
```



```
void inorder(Node *u) {
    if (u == NULL) return;
    inorder(u->left);
    printf(" %d", u->key);
    inorder(u->right);
}

void preorder(Node *u) {
    if (u == NULL) return;
    printf(" %d", u->key);
    preorder(u->left);
    preorder(u->right);
}

int main(void) {
    int m;
    if (scanf("%d", &m) != 1) {
        return 0;
    }

    for (int i = 0; i < m; i++) {
        char cmd[10];
        scanf("%s", cmd);

        if (strcmp(cmd, "insert") == 0) {
            int key, priority;
            scanf("%d %d", &key, &priority);
            root = insert(root, key, priority);
        } else if (strcmp(cmd, "find") == 0) {
            int key;
            scanf("%d", &key);
            Node *result = find_node(root, key);
            if (result != NULL) {
                printf("yes\n");
            } else {
                printf("no\n");
            }
        } else if (strcmp(cmd, "delete") == 0) {
            int key;
            scanf("%d", &key);
            root = search_and_delete(root, key);
        } else if (strcmp(cmd, "print") == 0) {
            inorder(root);
            printf("\n");
            preorder(root);
            printf("\n");
        }
    }

    return 0;
}


----------------------------------------------
FILE: ALDS1_9_A.c
----------------------------------------------
/*
完全二分木
すべての葉が同じ深さを持ち、すべての内部節点の次数が 2 であるような二分木を完全二分木と呼びます。また、二分木の最下位レベル以外のすべてのレベルは完全に埋まっており、最下位レベルは最後の節点まで左から順に埋まっているような木も（おおよそ

二分ヒープは、次の図のように、木の各節点に割り当てられたキーが 1 つの配列の各要素に対応した完全二分木で表されたデータ構造です。
```



```
二分ヒープを表す配列を
、二分ヒープのサイズ（要素数）を
 とすれば、
 に二分ヒープの要素が格納されます。木の根の添え字は
 であり、節点の添え字
 が与えられたとき、その親
、左の子
、右の子
 はそれぞれ
、
、
 で簡単に算出することができます。

完全二分木で表された二分ヒープを読み込み、以下の形式で二分ヒープの各節点の情報を出力するプログラムを作成してください。

node
: key =
, parent key =
, left key =
, right key =
,

ここで、
 は節点の番号（インデックス）、
 は節点の値、
 は親の値、
 は左の子の値、
 は右の子の値を示します。これらの情報をこの順番で出力してください。ただし、該当する節点が存在しない場合は、出力を行わないものとします。

入力
入力の最初の行に、二分ヒープのサイズ
 が与えられます。続いて、ヒープの節点の値（キー）を表す
 個の整数がそれらの節点の番号順に空白区切りで与えられます。

出力
上記形式で二分ヒープの節点の情報をインデックスが 1 から
 に向かって出力してください。各行の最後が空白となることに注意してください。

制約
節点のキー
入力例 1
5
7 8 1 2 3
出力例 1
node 1: key = 7, left key = 8, right key = 1,
node 2: key = 8, parent key = 7, left key = 2, right key = 3,
node 3: key = 1, parent key = 7,
node 4: key = 2, parent key = 8,
node 5: key = 3, parent key = 8,

参考文献
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
*/

#include <stdio.h>

int H[250000];
int H_size;

int parent(int i) {
```

```
    return i / 2;
}

int left(int i) {
    return 2 * i;
}

int right(int i) {
    return 2 * i + 1;
}

int main(void) {
    scanf("%d", &H_size);

    for (int i = 1; i <= H_size; i++) {
        scanf("%d", &H[i]);
    }

    for (int i = 1; i <= H_size; i++) {
        printf("node %d: key = %d, ", i, H[i]);

        if (parent(i) >= 1) {
            printf("parent key = %d, ", H[parent(i)]);
        }

        if (left(i) <= H_size) {
            printf("left key = %d, ", H[left(i)]);
        }

        if (right(i) <= H_size) {
            printf("right key = %d, ", H[right(i)]);
        }

        printf("\n");
    }

    return 0;
}


----------------------------------------------
FILE: ALDS1_9_B.c
----------------------------------------------
/*
最大ヒープ
「節点のキーがその親のキー以下である」という max-ヒープ条件を満たすヒープを、max-ヒープと呼びます。max-ヒープでは、最大の要素が根に格納され、ある節点を根とする部分木の節点のキーは、その部分木の根のキー以下となります。親子間のみに大小

例えば、下図はmax-ヒープの例です。



与えられた配列から以下の疑似コードに従ってmax-ヒープを構築するプログラムを作成してください。

 は、節点
 を根とする部分木が max-ヒープになるよう
 の値を max-ヒープの葉へ向かって下降させます。ここで
 をヒープサイズとします。

1  maxHeapify(A, i)
2      l = left(i)
3      r = right(i)
4      // 左の子、自分、右の子で値が最大のノードを選ぶ
```



```
5      if l ≤ H and A[l] > A[i]
6          largest = l
7      else
8          largest = i
9      if r ≤ H and A[r] > A[largest]
10         largest = r
11
12     if largest ≠ i  // i の子の方が値が大きい場合
13         A[i] と A[largest] を交換
14         maxHeapify(A, largest) // 再帰的に呼び出し
次の buildMaxHeap(A) はボトムアップに maxHeapify を適用することで配列
 を max-ヒープに変換します。

1 buildMaxHeap(A)
2    for i = H/2 downto 1
3        maxHeapify(A, i)
入力
入力の最初の行に、ヒープのサイズ
 が与えられます。続いて、ヒープの節点の値を表す
 個の整数が節点の番号が 1 から
 に向かって順番に空白区切りで与えられます。

出力
max-ヒープの節点の値を節点の番号が 1 から
 に向かって順番に 1 行に出力してください。各値の直前に 1 つの空白文字を出力してください。

制約
節点の値
入力例 1
10
4 1 3 2 16 9 10 14 8 7
出力例 1
 16 14 10 8 7 9 3 2 4 1

参考文献
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
*/

#include <stdio.h>

int H[500000];
int H_size;

int left(int i) {
    return 2 * i;
}

int right(int i) {
    return 2 * i + 1;
}

void maxHeapify(int A[], int i) {
    int l = left(i);
    int r = right(i);
    int largest;

    if (l <= H_size && A[l] > A[i]) {
        largest = l;
    } else {
        largest = i;
    }
```



```
    if (r <= H_size && A[r] > A[largest]) {
        largest = r;
    }

    if (largest != i) {
        int temp = A[i];
        A[i] = A[largest];
        A[largest] = temp;
        maxHeapify(A, largest);
    }
}

void buildMaxHeap(int A[]) {
    for (int i = H_size / 2; i >= 1; i--) {
        maxHeapify(A, i);
    }
}

int main(void) {
    scanf("%d", &H_size);

    for (int i = 1; i <= H_size; i++) {
        scanf("%d", &H[i]);
    }

    buildMaxHeap(H);

    for (int i = 1; i <= H_size; i++) {
        printf(" %d", H[i]);
    }
    printf("\n");

    return 0;
}


----------------------------------------------
FILE: ALDS1_9_C.c
----------------------------------------------
/*
優先順位キュー
優先度付きキュー (priority queue) は各要素がキーを持ったデータの集合
 を保持するデータ構造で、主に次の操作を行います :

: 集合
 に要素
 を挿入する
: 最大のキーを持つ
 の要素を
 から削除してその値を返す
優先度付きキュー
 に対して
、
を行うプログラムを作成してください。ここでは、キューの要素を整数とし、それ自身をキーとみなします。

入力
優先度付きキュー
 への複数の命令が与えられます。各命令は、insert
、extractまたはendの形式で命令が 1 行に与えられます。ここで
 は挿入する整数を表します。

end命令が入力の終わりを示します。
```



```
出力
extract命令ごとに、優先度付きキュー
から取り出される値を 1 行に出力してください。

制約
命令の数は
を超えない。
k
入力例 1
insert 8
insert 2
extract
insert 10
extract
insert 11
extract
extract
end
出力例 1
8
10
11
2
*/

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

int H[2000000];
int H_size = 0;

int parent(int i) {
    return i / 2;
}

int left(int i) {
    return 2 * i;
}

int right(int i) {
    return 2 * i + 1;
}

void maxHeapify(int i) {
    int l = left(i);
    int r = right(i);
    int largest;

    if (l <= H_size && H[l] > H[i]) {
        largest = l;
    } else {
        largest = i;
    }

    if (r <= H_size && H[r] > H[largest]) {
        largest = r;
    }

    if (largest != i) {
        int temp = H[i];
        H[i] = H[largest];
```

```
        H[largest] = temp;
        maxHeapify(largest);
    }
}

void insert(int key) {
    H_size++;
    int i = H_size;
    H[i] = key;

    while (i > 1 && H[parent(i)] < H[i]) {
        int temp = H[i];
        H[i] = H[parent(i)];
        H[parent(i)] = temp;
        i = parent(i);
    }
}

int extract() {
    if (H_size < 1) {
        return -1;
    }

    int max = H[1];
    H[1] = H[H_size];
    H_size--;
    maxHeapify(1);

    return max;
}

int main(void) {
    char cmd[10];

    while (1) {
        scanf("%s", cmd);

        if (strcmp(cmd, "insert") == 0) {
            int key;
            scanf("%d", &key);
            insert(key);
        } else if (strcmp(cmd, "extract") == 0) {
            printf("%d\n", extract());
        } else if (strcmp(cmd, "end") == 0) {
            break;
        }
    }

    return 0;
}


----------------------------------------------
FILE: ALDS1_9_D.c
----------------------------------------------
/*
ヒープソート
以下のように、ソートアルゴリズムはそれらの計算量や安定性など、様々な特徴を持ちます。

アルゴリズム　計算量
（最悪）　計算量
（平均）　安定性　　メモリ
```



```
効率 テクニック　　特徴
挿入ソート
ALDS1_1_A　O(
)　O(
)　〇　〇　挿入 整列されたデータに対して高速。
バブルソート
ALDS1_2_A　O(
)　O(
)　〇　〇　スワップ
選択ソート
ALDS1_2_B　O(
)　O(
)　×　〇　スワップ
シェルソート
ALDS1_2_D　O(
)　O(
)　×　〇　挿入
マージソート
ALDS1_5_A　O(
)　O(
)　〇　×　分割統治　安定で高速。
入力の配列に加えてメモリが必要。
計数ソート
ALDS1_6_A　O(
)　O(
)　〇　×　バケット　安定で高速。
要素の値に制限がある。
クイックソート
ALDS1_6_B　O(
)　O(
)　×　△　分割統治　データの隔たりに対する対策を実装すれば、高速でメモリ使用量も少ない。対策がなければ時間・メモリともに計算量が悪化する。
ヒープソート
ALDS1_9_D（この問題）　O(
)　O(
)　×　〇　ヒープ構造　　高速で、入力の配列以外にメモリが必要ない。
安定ではなく、ランダムアクセスが多く発生する。

ヒープソート（Heap Sort）はヒープのデータ構造に基づくソートで、入力配列内でソート処理を達成できる（メモリ効率のよい）、高速なソートアルゴリズムです。ヒープソートは、次のように実装することができます。

1  maxHeapify(A, i)
2      l = left(i)
3      r = right(i)
4      // select the node which has the maximum value
5      if l ≤ heapSize and A[l] > A[i]
6          largest = l
7      else
8          largest = i
9      if r ≤ heapSize and A[r] > A[largest]
10         largest = r
11
12     if largest ≠ i
13         swap A[i] and A[largest]
14         maxHeapify(A, largest)
15
16 heapSort(A):
17     // buildMaxHeap
18     for i = N/2 downto 1:
19         maxHeapify(A, i)
20     // sort
21     heapSize ← N
22     while heapSize ≥ 2:
23         swap(A[1], A[heapSize])
```



```
24         heapSize--
25         maxHeapify(A, 1)
一方、ヒープソートでは、離れた要素が頻繁にスワップされ、連続でない要素へのランダムアクセスが多く発生してしまいます。

要素の数列
が与えられます。最大ヒープを満たし、ヒープソートを行ったときに疑似コード25行目のmaxHeapifyにおけるスワップ回数の総数が最大となるような数列
の順列を１つ出力してください。

入力
1行目に、数列の長さを表す整数
 が与えられます。２行目に、
 個の整数が空白区切りで与えられます。

出力
条件を満たす数列を １ 行に出力してください。数列の連続する要素は１つの空白で区切って出力してください。

この問題では、１つの入力に対して複数の解答があります。条件を満たす出力は全て正解となります。

制約

 の要素
の要素は全て異なる
入出力例
入力例1
8
1 2 3 5 9 12 15 23
出力例1
23 9 15 2 5 3 12 1
*/

#include <stdio.h>

int A[500000];
int N;

int left(int i) {
    return 2 * i;
}

int right(int i) {
    return 2 * i + 1;
}

void maxHeapify(int A[], int i, int heapSize) {
    int l = left(i);
    int r = right(i);
    int largest;

    if (l <= heapSize && A[l] > A[i]) {
        largest = l;
    } else {
        largest = i;
    }

    if (r <= heapSize && A[r] > A[largest]) {
        largest = r;
    }

    if (largest != i) {
        int temp = A[i];
        A[i] = A[largest];
        A[largest] = temp;
```



```
        maxHeapify(A, largest, heapSize);
    }
}

void buildMaxHeap(int A[], int n) {
    for (int i = n / 2; i >= 1; i--) {
        maxHeapify(A, i, n);
    }
}

void heapSort(int A[], int n) {
    buildMaxHeap(A, n);
    int heapSize = n;
    while (heapSize >= 2) {
        int temp = A[1];
        A[1] = A[heapSize];
        A[heapSize] = temp;
        heapSize--;
        maxHeapify(A, 1, heapSize);
    }
}

int main(void) {
    scanf("%d", &N);

    for (int i = 1; i <= N; i++) {
        scanf("%d", &A[i]);
    }


    for (int i = 1; i <= N; i++) {
        for (int j = i + 1; j <= N; j++) {
            if (A[i] < A[j]) {
                int temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
        }
    }


    buildMaxHeap(A, N);

    for (int i = 1; i <= N; i++) {
        if (i > 1) printf(" ");
        printf("%d", A[i]);
    }
    printf("\n");

    return 0;
}


----------------------------------------------
FILE: P2.c
----------------------------------------------
```