#include<iostream>

#include<cstring>

#include<algorithm>

#include<cstdlib>

#include<vector>

#include<iterator>

#include<sstream>

#include<utility>

using namespace std;

template<class ElemType>

void print(const vector<ElemType>& v) {

for (typename vector<ElemType>::const\_iterator it = v.begin(); it != v.end(); ++it) {

cout << \*it << ' ';

}

}

//根据堆的定义，可知应当建立一个完全二叉树，完全二叉树可以用顺序存储结构来存储，一般来说存储的第一个元素

//的标号是从1开始，但本题中我采用的策略是从0开始，这使得父节点与其孩子的结点的标号关系发生了一点小变化

//即父结点的标号\*2+1=左孩子的标号、父结点的标号\*2+2=右孩子的标号。此外，在建堆开始时要找的最后一个非叶子

//结点的标号为：（总的数据个数-2）/2，我为什么要这么做呢？因为我之前的所有题都没有在容器的最开始预留一个

//空间，我想与之前保持一致

//堆调整（最大堆）

template<class ElemType>

void heapASCAdjust(vector<ElemType>& v, int hole, int size) {

if (hole < 0 || hole >= size) {

return;

}

else {

while (hole >= 0) {//注意：hole表示的是每次处理的结点的标号，且我们只处理非叶子结点，而且我们

//只比较非叶子结点与其孩子，如果一旦发生交换，那么就交换到底，那么如何判断交换到底，即已经

//使得处理的部分符合堆的定义，即

int L = 2 \* hole + 1;

int R = 2 \* hole + 2;

if (L > size - 1) {

--hole;

continue;

}

int tmp\_hole = hole;

while (L <= size - 1) {//L在合法的范围内

pair<ElemType, int>max\_e;

max\_e.first = v[L];

max\_e.second = L;

if (R <= size - 1) {//R在合法的范围内

if (v[L] < v[R]) {

max\_e.first = v[R];

max\_e.second = R;

}

}

if (v[tmp\_hole] < max\_e.first) {//注意：如果进行了一次交换，同时要对tmp\_hole进行变更

v[max\_e.second] = v[tmp\_hole];//也要对L、R进行相应的变更，要确保以标号为tmp\_hole

v[tmp\_hole] = max\_e.first;//为根结点的子树一定要符合堆的定义，即要穷尽向下比较

tmp\_hole = max\_e.second;

L = 2 \* tmp\_hole + 1;

R = 2 \* tmp\_hole + 2;

}

else{//表明以v[tmp\_hole]为根结点的子树暂时符合堆的定义

break;

}

}

--hole;//执行到这一步表明正在处理的这个结点有左右子树但是并没有发生任何交换

}

print(v);

cout << endl;

}

}

//堆调整（最小堆）

template<class ElemType>

void heapDASCAdjust(vector<ElemType>& v, int hole, int size) {

if (hole < 0 || hole >= size) {

return;

}

else {

while (hole >= 0) {//注意：hole表示的是每次处理的结点的标号，且我们只处理非叶子结点，而且我们

//只比较非叶子结点与其孩子，如果一旦发生交换，那么就交换到底，那么如何判断交换到底，即已经

//使得处理的部分符合堆的定义，即

int L = 2 \* hole + 1;

int R = 2 \* hole + 2;

if (L > size - 1) {

--hole;

continue;

}

int tmp\_hole = hole;

while (L <= size - 1) {//L在合法的范围内

pair<ElemType, int>min\_e;

min\_e.first = v[L];

min\_e.second = L;

if (R <= size - 1) {//R在合法的范围内

if (v[L] > v[R]) {

min\_e.first = v[R];

min\_e.second = R;

}

}

if (v[tmp\_hole] > min\_e.first) {

v[min\_e.second] = v[tmp\_hole];

v[tmp\_hole] = min\_e.first;

tmp\_hole = min\_e.second;

L = 2 \* tmp\_hole + 1;

R = 2 \* tmp\_hole + 2;

}

else {//表明以v[tmp\_hole]为根结点的子树暂时符合堆的定义

break;

}

}

--hole;//执行到这一步表明正在处理的这个结点有左右子树但是并没有发生任何交换

}

print(v);

cout << endl;

}

}

//建堆

template<class ElemType>

void heapSort(vector<ElemType>& v, const int& flag) { //flag：堆类型标记

if (v.size() < 2) {

return;

}

else {

if (flag != 1 && flag != 2) {

return;

}

int hole;//暂时将hole理解为要处理的最初的序号

if (flag == 1) {//要做升序排列，用到大根堆

for (int i = v.size(); i > 1; --i) {

hole = (i - 2) / 2;//暂时将hole理解为要处理的最初的序号

heapASCAdjust(v,hole, i);

ElemType tmp = v[0];

v[0] = v[i - 1];

v[i - 1] = tmp;

}

print(v);

}

else if (flag == 2) {//要做降序排列，用到小根堆

for (int i = v.size(); i > 1; --i) {

hole = (i - 2) / 2;//暂时将hole理解为要处理的最初的序号

heapDASCAdjust(v, hole, i);

ElemType tmp = v[0];

v[0] = v[i - 1];

v[i - 1] = tmp;

}

print(v);

}

}

}

int main(){

//数据类型标记（0：int，1：double，2：char，3：string）

int flag;

cin >> flag;

switch (flag) {

case 0: {

vector<int>v;

int tmp;

int flag2;

cin >> flag2;

cin.ignore();

while (cin>>tmp) {//最好采用这种输入方式判断 ，而不要采用cin.peek()!='\n'的方式判断输入是否结束，否则的话会比较耗时

v.push\_back(tmp);

if (cin.get() == '\n')break;

}

heapSort(v, flag2);

}

break;

case 1: {

vector<double>v;

double tmp;

int flag2;

cin >> flag2;

cin.ignore();

while (cin >> tmp) {

v.push\_back(tmp);

if (cin.get() == '\n')break;

}

heapSort(v, flag2);

}

break;

case 2: {

vector<char>v;

char tmp;

int flag2;

cin >> flag2;

cin.ignore();

while (cin >> tmp) {

v.push\_back(tmp);

if (cin.get() == '\n')break;

}

heapSort(v, flag2);

}

break;

case 3:{

vector<string>v;

string tmp;

int flag2;

cin >> flag2;

cin.ignore();

getline(cin, tmp);

string s = "";

for (typename string::const\_iterator it = tmp.begin(); it != tmp.end(); ++it) {

if (\*it != ' ') {

s += \*it;

if (it == tmp.end() - 1) {

v.push\_back(s);

break;

}

}

else if (it == tmp.end() - 1) {

if (\*it != ' ') {

s += \*it;

}

v.push\_back(s);

}

else {

v.push\_back(s);

s = "";

}

}

heapSort(v, flag2);

}

break;

default: cout << "err";

}

return 0;

}