|  |
| --- |
| Университет итмо, кафедра вт |
| Лабораторная работа №1 по дисциплине «Алгоритмы и структуры данных» |
| «Введение в алгоритмы» |
| Группа Р3202 |
| **Выполнила: Орлова Кристина Александровна** |
| **Преподаватель: Косяков Михаил Сергеевич** |

|  |
| --- |
| *14.03.18* |

**№2025**

В ходе составления алгоритма было выявлено, что схваток будет тем больше, чем ближе количества участников команд друг к другу.  
Поэтому, зная количество команд и участников, можно найти число, которым нужно заполнить все команды. Если какие-то участники останутся, то их нужно добавить по одному в каждую команду.

|  |
| --- |
|  |
| #include <iostream> |
|  | #include <ctime> |
|  |  |
|  | using namespace std; |
|  |  |
|  | int count(int const \*team, int k) { |
|  | int sum = 0; |
|  | int count = 0; |
|  |  |
|  | for (int i = 0; i < k - 1; i++) { |
|  | for (int j = i + 1; j < k; j++) { |
|  | sum += team[j]; |
|  | } |
|  | count += team[i] \* sum; |
|  |  |
|  | sum = 0; |
|  | } |
|  |  |
|  | return count; |
|  | } |
|  |  |
|  | void createTeams(int \*team, int n, int k) { |
|  | int manCount = n / k; |
|  | int rest = n % k; |
|  |  |
|  | for (int i = 0; i < k; i++) { |
|  | team[i] = manCount; |
|  |  |
|  | if (rest != 0) { |
|  | team[i]++; |
|  | rest--; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | int test (int n, int k) { |
|  | int team[k]; |
|  |  |
|  | createTeams(team, n, k); |
|  |  |
|  | return count(team, k); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Сначала все команды заполняются одинаковым числом игроков |
|  | \* Если остались еще игроки, не распределенные по командам, то их по очереди по одному добавляют в команду |
|  | \* @return |
|  | \*/ |
|  | int main() { |
|  | int number; |
|  | int n; |
|  | int k; |
|  | //unsigned int start\_time = clock(); // начальное время |
|  |  |
|  | cin >> number; |
|  |  |
|  | int variants[number]; |
|  |  |
|  | for (int i = 0; i < number; i++) { |
|  | cin >> n >> k; |
|  |  |
|  | variants[i] = test(n, k); |
|  | } |
|  |  |
|  | for (int i = 0; i < number; i++) { |
|  | cout << variants[i] << endl; |
|  | } |
|  |  |
|  | //unsigned int end\_time = clock(); // конечное время |
|  | //cout << "Time is " << (end\_time - start\_time) / 1000.0 << endl; |
|  | return 0; |
|  | } |
|  |  |

**№1005**

Решением данной задачи является перебор.  
Создается два массива размером числа комбинаций: массив чисел и массив индексов. Массив чисел сперва заполняется исходными числами. Далее он последовательно расширяется суммами этих чисел. Одновременно с этим заполняется массив индексов, каждый из которых указывает, с какими исходными числами нужно еще сложить данную сумму и записать полученные значения в массив. Если индекс 0, то с этой суммой больше не нужно работать. В процессе составления сумм выбирается и запоминается минимальная разность между суммарными весами кучек.

|  |
| --- |
| #include <iostream> |
|  | #include <cmath> |
|  |  |
|  | using namespace std; |
|  |  |
|  | int count(int const \*array, int n, int sum) { |
|  | int min = -1; |
|  | int k = 0; |
|  | int last = 0; |
|  | int old = 0; |
|  | int number = (int) pow(2, n) - 1; |
|  | int variants[number]; |
|  | int indexes[number]; |
|  |  |
|  | for (int i = 0; i < number; i++) { |
|  | if (i < n) { |
|  | variants[i] = array[i]; |
|  |  |
|  | if (i + 1 < n) { |
|  | indexes[i] = i + 1; |
|  | } else { |
|  | indexes[i] = 0; |
|  | } |
|  |  |
|  | k++; |
|  | } else { |
|  | variants[i] = 0; |
|  | indexes[i] = 0; |
|  | } |
|  | } |
|  |  |
|  | if (k == number) { |
|  | min = variants[0]; |
|  | } |
|  |  |
|  | while (k != number) { |
|  | old = k; |
|  |  |
|  | for (int i = last; i < old; i++) { |
|  | int index = indexes[i]; |
|  |  |
|  | if (index != 0) { |
|  | int digit = variants[i]; |
|  | int x = 0; |
|  |  |
|  | for (int j = index; j < n; j++) { |
|  | x = digit + variants[j]; |
|  | variants[k] = x; |
|  |  |
|  | if (j + 1 < n) { |
|  | indexes[k] = j + 1; |
|  | } else { |
|  | indexes[k] = 0; |
|  | } |
|  |  |
|  | k++; |
|  |  |
|  | if (min == -1 || min > abs(sum - 2 \* x)) { |
|  | min = abs(sum - 2 \* x); |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | last = old; |
|  | } |
|  |  |
|  | return min; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Перебор. |
|  | \* Сначала массив заполняется исходными числами. |
|  | \* Далее он последовательно расширяется суммами чисел с исходными числами, |
|  | \* на которые указывают индексы из соответствующего массива. |
|  | \* Если индекс 0, то с этим числом больше не нужно работать. |
|  | \* В процессе составления новых сумм выбирается минимальная разность между суммарными весами кучек. |
|  | \* @return |
|  | \*/ |
|  | int main() { |
|  | int number; |
|  | int sum = 0; |
|  | cin >> number; |
|  |  |
|  | int array[number]; |
|  |  |
|  | for (int i = 0; i < number; i++) { |
|  | cin >> array[i]; |
|  | sum += array[i]; |
|  | } |
|  |  |
|  | cout << count(array, number, sum) << endl; |
|  | return 0; |
|  | } |

**№1155**

Строится граф связей. Составляются смежные пары вершин, с которыми можно проделывать операции добавления и удаления дуонов. Составляются несмежные пары, к одной вершине которых от другой можно дойти за 2 вершины. К этим парам прописываются пути через смежные пары. Затем запускается цикл, в котором проверяется последовательно каждая пара на возможность удаления дуонов. Он работает до тех пор, пока не останется ни одного.

|  |
| --- |
| #include <iostream> |
|  |  |
|  | using namespace std; |
|  |  |
|  | string giveAnswer(int a, int b, int c, int d, int e, int f, int g, int h) { |
|  | string answer; |
|  |  |
|  | if (a + c + h + f == g + e + b + d) { |
|  | answer = ""; |
|  |  |
|  | while (a != 0 || b != 0 || c != 0 || d != 0 || e != 0 || f != 0 || g != 0 || h != 0) { |
|  | if (a != 0 && e != 0) { |
|  | answer += "AE-\n"; |
|  | a--; |
|  | e--; |
|  | continue; |
|  | } |
|  | if (a != 0 && d != 0) { |
|  | answer += "AD-\n"; |
|  | a--; |
|  | d--; |
|  | continue; |
|  | } |
|  | if (a != 0 && b != 0) { |
|  | answer += "AB-\n"; |
|  | a--; |
|  | b--; |
|  | continue; |
|  | } |
|  | if (e != 0 && h != 0) { |
|  | answer += "EH-\n"; |
|  | e--; |
|  | h--; |
|  | continue; |
|  | } |
|  | if (e != 0 && f != 0) { |
|  | answer += "EF-\n"; |
|  | e--; |
|  | f--; |
|  | continue; |
|  | } |
|  | if (d != 0 && h != 0) { |
|  | answer += "DH-\n"; |
|  | d--; |
|  | h--; |
|  | continue; |
|  | } |
|  | if (c != 0 && d != 0) { |
|  | answer += "CD-\n"; |
|  | c--; |
|  | d--; |
|  | continue; |
|  | } |
|  | if (b != 0 && f != 0) { |
|  | answer += "BF-\n"; |
|  | b--; |
|  | f--; |
|  | continue; |
|  | } |
|  | if (b != 0 && c != 0) { |
|  | answer += "BC-\n"; |
|  | c--; |
|  | b--; |
|  | continue; |
|  | } |
|  | if (f != 0 && g != 0) { |
|  | answer += "FG-\n"; |
|  | f--; |
|  | g--; |
|  | continue; |
|  | } |
|  | if (g != 0 && h != 0) { |
|  | answer += "GH-\n"; |
|  | g--; |
|  | h--; |
|  | continue; |
|  | } |
|  | if (c != 0 && g != 0) { |
|  | answer += "CG-\n"; |
|  | c--; |
|  | g--; |
|  | continue; |
|  | } |
|  | if (a != 0 && g != 0) { |
|  | answer += "BF+\nAB-\nFG-\n"; |
|  | a--; |
|  | g--; |
|  | continue; |
|  | } |
|  | if (e != 0 && c != 0) { |
|  | answer += "DH+\nCD-\nEH-\n"; |
|  | e--; |
|  | c--; |
|  | continue; |
|  | } |
|  | if (b != 0 && h != 0) { |
|  | answer += "EF+\nBF-\nEH-\n"; |
|  | h--; |
|  | b--; |
|  | continue; |
|  | } |
|  | if (d != 0 && f != 0) { |
|  | answer += "AC+\nEF-\nAD-\n"; |
|  | d--; |
|  | f--; |
|  | continue; |
|  | } |
|  | } |
|  | } else { |
|  | answer = "IMPOSSIBLE"; |
|  | } |
|  |  |
|  | return answer; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* Строится граф связей. |
|  | \* Составляются смежные пары. |
|  | \* Составляются пары, в которых с одного конца до другого можно добраться за две вершины. |
|  | \* К этим парам прописываются пути. |
|  | \* @return |
|  | \*/ |
|  | int main() { |
|  | int a, b, c, d, e, f, g, h; |
|  |  |
|  | cin >> a >> b >> c >> d >> e >> f >> g >> h; |
|  | cout << giveAnswer(a, b, c, d, e, f, g, h) << endl; |
|  | return 0; |
|  | } |

**№1296**

За один проход складываются все числа. Если в какой-то момент сумма станет меньше нуля, то она обнуляется и продолжает складываться с оставшимися числами. Пик суммы фиксируется.

|  |
| --- |
| #include <iostream> |
|  |  |
|  | using namespace std; |
|  |  |
|  | int count(int const \*array, int n) { |
|  | int max = 0; |
|  | int sum = 0; |
|  |  |
|  | for (int i = 0; i < n; i++) { |
|  | sum += array[i]; |
|  |  |
|  | if (sum < 0) { |
|  | sum = 0; |
|  | } |
|  |  |
|  | if (sum > max) { |
|  | max = sum; |
|  | } |
|  | } |
|  |  |
|  | return max; |
|  | } |
|  |  |
|  | int main() { |
|  | int number; |
|  | cin >> number; |
|  |  |
|  | int array[number]; |
|  |  |
|  | for (int i = 0; i < number; i++) { |
|  | cin >> array[i]; |
|  | } |
|  |  |
|  | cout << count(array, number) << endl; |
|  | return 0; |
|  | } |

**№1401**

Смысл алгоритма в том, что любую доску размером 2^n x 2^n нужно разбивать на 4 квадрата. Данные 4 квадрата соединяются цифрами, кроме того из них, который содержит 0 (или любую другую цифру, которая для данного квадрата является "0"). Затем нужно рассмотреть левый верхний квадрат и то же самое сделать для него. Так нужно делать для всех исходных квадратов, рекурсивно углубляясь до тех пор, пока размер квадрата не станет равен 4. Далее нужно поменять квадрат на правый верхний и т.д., постепенно поднимаясь к исходной доске и углубляясь вновь.

|  |
| --- |
| #include <iostream> |
|  | #include <cmath> |
|  | #include <iomanip> |
|  | #include <list> |
|  |  |
|  | using namespace std; |
|  |  |
|  | list <int> depth; |
|  |  |
|  | int findBelongingToSquare(int x, int y, int number) { |
|  | int half = number / 2 - 1; |
|  | x = x % number; |
|  | y = y % number; |
|  |  |
|  | if (x <= half && y <= half) { |
|  | return 1; |
|  | } |
|  |  |
|  | if (x <= half && y > half) { |
|  | return 2; |
|  | } |
|  |  |
|  | if (x > half && y <= half) { |
|  | return 3; |
|  | } |
|  |  |
|  | if (x > half && y > half) { |
|  | return 4; |
|  | } |
|  | } |
|  |  |
|  | int findDepth(int number) { |
|  | int result = 0; |
|  |  |
|  | while (number >= 8) { |
|  | result++; |
|  | number /= 2; |
|  | } |
|  |  |
|  | return result; |
|  | } |
|  |  |
|  | bool findBelongingToOneSquare(int x1, int y1, int x2, int y2, int number) { |
|  | int square1 = findBelongingToSquare(x1, y1, number); |
|  | int square2 = findBelongingToSquare(x2, y2, number); |
|  |  |
|  | return square1 == square2; |
|  | } |
|  |  |
|  | void changeCoordinate(int numberOfSquare, int squareSize, int \*i1, int \*j1, int i0, int j0) { |
|  | if (squareSize == 4) { |
|  | squareSize \*= 2; |
|  | } |
|  |  |
|  | switch (numberOfSquare) { |
|  | case 1: |
|  | \*i1 = 0 + i0; |
|  | \*j1 = 0 + j0; |
|  | break; |
|  | case 2: |
|  | \*i1 = 0 + i0; |
|  | \*j1 = squareSize / 2 + j0; |
|  | break; |
|  | case 3: |
|  | \*i1 = squareSize / 2 + i0; |
|  | \*j1 = 0 + j0; |
|  | break; |
|  | case 4: |
|  | \*i1 = squareSize / 2 + i0; |
|  | \*j1 = squareSize / 2 + j0; |
|  | break; |
|  | default: |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | void incLastInList() { |
|  | int value = depth.back(); |
|  |  |
|  | depth.pop\_back(); |
|  | value++; |
|  | depth.push\_back(value); |
|  | } |
|  |  |
|  | void insertList(int number) { |
|  | int c = findDepth(number); |
|  |  |
|  | while (depth.size() < c) { |
|  | depth.push\_back(1); |
|  | } |
|  | } |
|  |  |
|  | void setCoordinate(int \*i1, int \*j1, int number) { |
|  | int i0 = 0; |
|  | int j0 = 0; |
|  | int size = number; |
|  |  |
|  | for (int n: depth) { |
|  | changeCoordinate(n, size, i1, j1, i0, j0); |
|  | i0 = \*i1; |
|  | j0 = \*j1; |
|  | size /= 2; |
|  | } |
|  | } |
|  |  |
|  | void changeSquare(int number, int \*squareSize, int \*numberOfSquare, int i1, int j1) { |
|  | insertList(number); |
|  | incLastInList(); |
|  |  |
|  | while (depth.size() > 1 && depth.back() == 5) { |
|  | depth.pop\_back(); |
|  | incLastInList(); |
|  | } |
|  | insertList(number); |
|  |  |
|  | \*numberOfSquare = \*numberOfSquare + 1; |
|  |  |
|  | if (\*squareSize == 4 && \*numberOfSquare > 4) { |
|  | \*squareSize \*= 2; |
|  | } |
|  |  |
|  | while (\*numberOfSquare > 4 && \*squareSize \* 2 <= number) { |
|  | \*squareSize \*= 2; |
|  | \*numberOfSquare = findBelongingToSquare(i1, j1, \*squareSize); |
|  |  |
|  | if (\*numberOfSquare == 4) { |
|  | \*numberOfSquare = \*numberOfSquare + 1; |
|  | continue; |
|  | } else { |
|  | \*numberOfSquare = 1; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void doTask(int number, int x, int y) { |
|  | int array[number][number]; |
|  | int half = number / 2; |
|  | int k = 1; |
|  | int i1 = 0; |
|  | int j1 = 0; |
|  | int x1; |
|  | int y1; |
|  | int squareSize = number; |
|  | int numberOfSquare = 1; |
|  |  |
|  | //initialize matrix |
|  | for (int i = 0; i < number; i++) { |
|  | for (int j = 0; j < number; j++) { |
|  | if (i != x || j != y) { |
|  | array[i][j] = -1; |
|  | } else { |
|  | array[i][j] = 0; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | //insertCentralSquare |
|  | for (int i = i1 + half - 1; i <= i1 + half; i++) { |
|  | for (int j = j1 + half - 1; j <= j1 + half; j++) { |
|  | if (array[i][j] == -1 && !findBelongingToOneSquare(i, j, x, y, squareSize)) { |
|  | array[i][j] = k; |
|  | } |
|  | } |
|  | } |
|  | k++; |
|  |  |
|  | if (number > 4) { |
|  | while (true) { |
|  | if (squareSize >= 4) { |
|  | if (squareSize > 4) { |
|  | squareSize /= 2; |
|  | } |
|  |  |
|  | //выбрать координаты для квадрата |
|  | setCoordinate(&i1, &j1, number); |
|  |  |
|  | half = squareSize / 2; |
|  |  |
|  | //найти координаты выколотой точки |
|  | if (array[i1][j1] != -1) { |
|  | x1 = i1; |
|  | y1 = j1; |
|  | } else if (array[i1][j1 + squareSize - 1] != -1) { |
|  | x1 = i1; |
|  | y1 = j1 + squareSize - 1; |
|  | } else if (array[i1 + squareSize - 1][j1] != -1) { |
|  | x1 = i1 + squareSize - 1; |
|  | y1 = j1; |
|  | } else if (array[i1 + squareSize - 1][j1 + squareSize - 1] != -1) { |
|  | x1 = i1 + squareSize - 1; |
|  | y1 = j1 + squareSize - 1; |
|  | } else { |
|  | x1 = x; |
|  | y1 = y; |
|  | } |
|  |  |
|  | //insertCentralSquare |
|  | for (int i = i1 + half - 1; i <= i1 + half; i++) { |
|  | for (int j = j1 + half - 1; j <= j1 + half; j++) { |
|  | if (array[i][j] == -1 && !findBelongingToOneSquare(i, j, x1, y1, squareSize)) { |
|  | array[i][j] = k; |
|  | } |
|  | } |
|  | } |
|  | k++; |
|  |  |
|  | if (squareSize == 4) { |
|  | squareSize /= 2; |
|  | } |
|  | } else { |
|  | //смена квадрата внутри квадрата и, если надо, координат внешнего квадрата |
|  | squareSize \*= 2; |
|  |  |
|  | changeSquare(number, &squareSize, &numberOfSquare, i1, j1); |
|  |  |
|  | if (numberOfSquare == 5) { |
|  | break; |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | //дозаполнение квадратов 2 x 2 |
|  | i1 = 0; |
|  | j1 = -2; |
|  |  |
|  | while (i1 + 2 != number || j1 + 2 != number) { |
|  | j1 += 2; |
|  |  |
|  | if (j1 == number) { |
|  | i1 += 2; |
|  | j1 = 0; |
|  | } |
|  |  |
|  | if (array[i1][j1] == -1) { |
|  | array[i1][j1] = k; |
|  | } |
|  |  |
|  | if (array[i1][j1 + 1] == -1) { |
|  | array[i1][j1 + 1] = k; |
|  | } |
|  |  |
|  | if (array[i1 + 1][j1] == -1) { |
|  | array[i1 + 1][j1] = k; |
|  | } |
|  |  |
|  | if (array[i1 + 1][j1 + 1] == -1) { |
|  | array[i1 + 1][j1 + 1] = k; |
|  | } |
|  |  |
|  | k++; |
|  | } |
|  |  |
|  | //write matrix |
|  | cout << "Answer: " << endl; |
|  |  |
|  | for (int i = 0; i < number; i++) { |
|  | for (int j = 0; j < number; j++) { |
|  | cout << setw(6) << array[i][j]; |
|  | } |
|  | cout << endl; |
|  | } |
|  | } |
|  |  |
|  | int main() { |
|  | int n; |
|  | int number; |
|  | int x, y; |
|  |  |
|  | cin >> n; |
|  | cin >> x >> y; |
|  |  |
|  | number = (int) pow(2, n); |
|  |  |
|  | x--; |
|  | y--; |
|  |  |
|  | doTask(number, x, y); |
|  | return 0; |
|  | } |