Университет ИТМО

Факультет программной инженерии и компьютерной техники

# Лабораторная работа №1

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P3211

Болорболд А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

**Задача №1 «Агроном-любитель»**

1. *Краткое описание алгоритма:*

Здесь используется техника «скользящего окна»: ищутся индексы (разница между ними максимальная), где не повторяется один и тот же элемент 3 раза подряд. Если есть 3 совпадении, то индексы раздвигаются соответственно. После того, как делается ещё одна проверка, выводится ответ.

*Исходный код:*

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int maxLeft, maxRight, curLeft, curRight, n, cur, prev, eq;
    maxLeft = maxRight = curLeft = curRight = 1, prev = eq = 0;
    cin >> n;
    while(n--){
        cin >> cur;
        if(cur == prev){
            eq++;
            if(eq == 3) {
                if(maxRight - maxLeft < curRight - curLeft) {
                    maxLeft = curLeft;
                    maxRight = curRight;
                }
                curLeft = curRight - 1;
                eq--;
            }
        } else {
            prev = cur;
            eq = 1;
        }
        curRight++;
    }
    if(maxRight - maxLeft < curRight - curLeft) {
        maxLeft = curLeft;
        maxRight = curRight;
    }
    cout << maxLeft << " " << --maxRight << "\n";
    return 0;
}
```

2. *Корректность применяемого алгоритма:*

Для этого рассматриваем текущий, предыдущий и пред- предыдущий цветки, чтобы понять, когда встретятся 3 подряд одинаковых цветка. Изначально мы воспринимаем текущим началом самого длинного отрезка первый цветок. Когда встречаем три одинаковых цветка — берем за текущее начало локального длинного участка предыдущий цветок (так, чтобы начало нового длинного участка составляли два одинаковых цветка — т.е. в участке нет трёх одинаковых цветков, но и длина благодаря этому максимальная). Если текущая длина участка больше длины предыдущего участка, считаем текущее начало участка началом максимального участка, конец максимального участка — текущим цветком, а максимальную длину —

текущей длиной. Таким образом, обработав все цветки, мы отслеживаем максимальную длину участка и его границы на протяжении всего процесса, значит, алгоритм гарантирует, что в любом текущем отслеживаемом участке не будут трёх одинаковых цветков подряд, а к кону выполнения будет найден конечный максимальный участок, удовлетворяющий условиям.

3. *Оценка сложности по времени:*

Так как количество итерации напрямую зависит от значения n, сложность по времени в **среднем и худшем** должна составлять **O(n)**.

4. *Оценка сложности по памяти:*

Так как здесь используется постоянное число переменных и не происходит дополнительной аллокации памяти (например, создание массива для хранения и обработки последующих значений), сложность по памяти должна составлять **O(1)**.

**Задача №2 «Зоопарк Глеба»**

1. *Краткое описание алгоритма:*

Здесь используется такой тип данных, где можно динамически добавить и убрать буквы (то есть животные и ловушки) из него. В моём случае был стек, но можно использовать и другие структуры данных, как например вектор. Если буквы соответствуют друг другу, то они вытаскиваются из своих стеков. Ещё есть наблюдательный стек, состояние которого отвечает за вывод ответа. Если там есть какое-то значение (то есть животные или ловушки, не нашедшие друг друга либо из-за отсутствия пары, либо из-за невозможности не пересекаться), то выводится «Impossible», а если нет, то выводится элементы массива, то есть индексы животных в ловушках.


*Исходный код:*

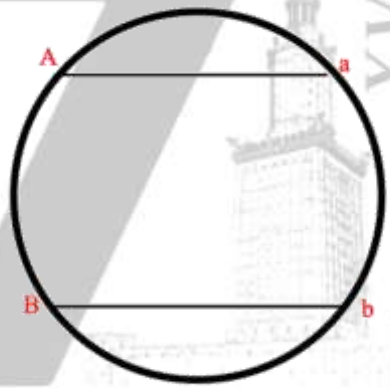
```
#include<bits/stdc++.h>
#include<iostream>
using namespace std;

int main(){
    string str;
    cin >> str;
    bool caught;
    stack<char> st;
    int animal = 0, trap = 0;
    stack<int> animals, traps;
    vector<int> a;
    for(char c : str){
        (islower(c)) ? animals.push(++animal) : traps.push(++trap),
```

```
a.push_back(0);
        if(st.empty()){
            st.push(c);
        } else {
            caught = (isupper(st.top()) && islower(c)
                    || islower(st.top()) && isupper(c))
                    && tolower(st.top()) == tolower(c);
            if(caught){
                st.pop();
                a.at(traps.top()-1) = animals.top();
                traps.pop();
                animals.pop();
            } else {
                st.push(c);
            }
        }
    }
    if(!st.empty()) {
        cout << "Impossible";
    } else {
        cout << "Possible\n";
        for(int i : a){
            if(i != 0){
                cout << i << " ";
            }
        }
    }
    return 0;
}
```

2. *Корректность применяемого алгоритма:*

Для общей простоты я решил доказывать корректность этого алгоритма геометрическим способом. Если рассматривать животные и ловушки как точки на окружности, то для этого надо доказывать, что **хорды**, проведённые по соответствующим парам, **не пересекаются**.




Делим хордой окружность, тогда получаются сегменты Aa и ABba. Это делается до тех пор, когда проведены все хорды. Если предположить, что решение существует, то найдётся *такая* комбинация деления хордами окружностей, а это значит, что

операцию можно повторять, пока не останется какое-то количество сегментов и полусегментов. От этого понятно, что хорды не должны пересекаться, ибо в обратном случае получаются секторoподобные фигуры, что и требовалось доказать.

3. *Оценка сложности по времени:*

Так как алгоритм проходит через массив символов (или же String) только один раз, сложность по времени составляет **O(n)**, где n — длина массива символов. Операции с каждым символом занимают O(1).

4. *Оценка сложности по памяти:*

Так как здесь используются стеки и векторы для хранения животных и ловушек, сложность по памяти составляет **O(n) в худшем случае**.

**Задача №3 «Конфигурационный файл»**

1. *Краткое описание алгоритма:*

Принцип решения подсказывали мне мои соотечественники из старших курсов, но к сожалению, он не смог мне помочь. Поэтому я решил идти своим путём.

Вместо того, чтобы создать период кеширования (до сих пор не могу понять, почему мне подкинули именно такую идею), я решил создать отдельную структуру, состоящий из самого значения и его «глубины». Под «глубиной» разумеется уровень вложенности в блоках. Оттуда я создал мап (ключ и значение, как раз подходит) и стек (из-за знакомства, мог использовать vector вместо этого), где сохраняются изменённые при инициировании блока переменные. В зависимости от символа меняются значение глубины. Дальше просто: я создал 2 отдельных хэшов, которые отвечают за ключ (индекс переменного) и значение переменного. Если не выполнялись все условия в третьем блоке кода, то это означает, что какое-то значение переменного изменялось в блоке, поэтому оно внесётся в стек изменённых переменных, от которого можно будет потом присваивать старое значение переменному.

*Исходный код:*

```
#include<bits/stdc++.h>
using namespace std;

struct val_depth{
    int value;
    int depth;
};

int main(){
    int depth = 0;
    stack<unordered_set<string>> changedVal;
    map<string, stack<val_depth>> hashMap;
    changedVal.emplace(0);
    string input;
    while(cin >> input){
```

```
//         if(input == "&") {
//             break;
//         }
        if(input == "{") {
            depth++;
            changedVal.emplace(0);
        } else if(input == "}"){
            depth--;
            for(const string &k: changedVal.top())
                if(!hashMap[k].empty() && hashMap[k].top().depth > depth){
                    hashMap[k].pop();
                }
            changedVal.top().clear();
            changedVal.pop();
        } else if(!input.empty()){
            char equalSign = (int) input.find('=');
            string lHash = input.substr(0, equalSign);
            string rHash = input.substr(equalSign + 1, input.length());
            if(hashMap[lHash].empty()) {
                hashMap[lHash].push({0, depth});
            }
            if(isdigit(rHash[0]) || rHash[0] == '-'){
                if(hashMap[lHash].top().depth < depth){
                    hashMap[lHash].push({stoi(rHash), depth});
                } else {
                    hashMap[lHash].top() = {stoi(rHash), depth};
                }
            } else {
                if(hashMap[rHash].empty()){
                    hashMap[rHash].push({0, depth});
                }
                cout << hashMap[rHash].top().value << "\n";
                if(hashMap[lHash].top().depth < depth){
                    hashMap[lHash].push({hashMap[rHash].top().value, depth});
                } else {
                    hashMap[lHash].top() = {hashMap[rHash].top().value, depth};
                }
            }
            changedVal.top().insert(lHash);
        }
    }
    return 0;
}
```

2. *<u>Корректность применяемого алгоритма:</u>*

Здесь обрабатывается конфигурация строка за строкой. При открытии блока мы инициализируем новое множество назначений переменных в стеке.

Когда мы встречаем присваивание и это число, мы устанавливаем новое значение на это число, если это другая переменная, то мы берём верхнее значение (текущая глубина, т.е. последнее назначенное) со стека значений мапа конфига и выводим его. После этого мы добавляем имя переменной в верхнее множество (т.е. для текущего блока) стека назначений переменных, а также в конфиге по имени этой переменной мы прикладываем новое значение переменной.

При закрытии блока нужно очистить переменные текущего уровня. Для этого для всех значений верхнего (т.е. последнего открытого) множества стека переменных мы убираем верхнее значение стека (pop) значений переменных в мапе конфига для этой переменной и убираем со стека переменных это множество. Т.е. соответственно всем назначениям этого блока (хранятся во множестве на вершине стека) мы удаляем такое же количество назначений из конфига, оставляя на вершине конфига значение, назначенное до входа в блок.

Рассмотрим пример:

```
a=69
a=420
b=a
{
b=228
a=b
}
a=b
b=1488
```

В строках 1-2 при назначении мы пушим на стек конфига для *a* значения 69 и 420. В строке 3 для *b* пушим верхнее значение со стека для *a*, т.е. 420. Теперь конфиг содержит {a: [69, 420], b: [420]}. Верхнее множество стека переменных: [a, a, b]. Открываем новый блок, теперь стек: [[a, a, b], []]. После строк 5 и 6 структуры данных имеют вид: {a: [69, 420, 228], b: [420, 228]} и [[a, a, b], [b, a]]. Закрываем блок и удаляем значения, присвоенные за время существования блока следующим образом: пока верхнее множество стека (красное) не содержит пустое множество, мы удаляем последнее значение стека из конфига, основываясь на имени переменной в стеке назначений переменных (для *a* удаляем 228, потом для *b* удаляем 228), после чего удаляем верхнее значение (имя переменной) со стека назначений переменных — той, чье значение мы удалили перед этим. Когда мы очистим все назначения из этого блока, мы удаляем верхнее множество со стека назначений переменных, соответствующее данной глубине видимости. После этого значение словаря конфига снова будут {a: [69, 420], b: [420]}. Поэтому в строках 8-9 мы используем значения, установленные для блока. К концу выполнения структуры данных будут содержать следующие: {a: [69, 420, 420], b: [420, 1488]} и [[a, a, b, a, b]].

3. *Оценка сложности по времени:*

Количество выполняемых циклов равно количеству строк конфигурационного файла — N. Если это операция закрытия блока, то она выполняется за **O(K)**, где K —

количество назначений в закрытом блоке. Открытие блока осуществляется за O(1), назначение переменной — за O(log(n)) — при добавлении значения в словарь.

4. *Оценка сложности по памяти:*

В худшем случае в стеке словаря будет храниться L значений (это количество всех назначений за время работы программы, если не было блоков кроме внешнего) значений, а в стеке назначений в таком случае L значений будет только если все эти назначения разные, поэтому сложность по памяти должна составлять **O(L)**.

**Задача №4 «Профессор Хаос»**

1. *Краткое описание алгоритма*:

Сначала казалось, что это довольно простая задача с проверк***ой*** условия. Мне кажется, что здесь почти нечего объяснить (если есть, то прошу прощения за пафосность):

Но не тут-то было, когда мне подсказывали, что эту задачу можно решать за O(1). Моя реализация в любом случае был O(n), поэтому я пошёл искать за ответы.

*Исходный код:*

```
#include<bits/stdc++.h>
using namespace std;

int main() {
    int a, b, c, d;
    long long k;
    cin >> a >> b >> c >> d >> k;
    if(a == a * b - c) {
        cout << a;
    } else if(k > d) {
        cout << ((a * b - c <= a) ? 0 : d);
    } else {
        for(int i = 1; i <= k; i++) {
            a = min(d, max(a * b - c, 0));
        }
        cout << a << endl;
    }
    return 0;
}
```

2. *Корректность применяемого алгоритма:*

По принципе бритвы Оккама сначала я полагал, что O(1) происходит в любом случае. Но такое просто никак невозможно, так как само решение задачи полагается на использовании итерации. Поэтому начал думать про второй вариант: O(1) может проявиться лишь в определённых случаях, а O(n) остаётся в качестве худшего. Следовательно, мне пришлось имплементировать в программу проверки ***частных случаев***.

Частный случай 1: если изначальное количество бактерии не изменилось. Это буквально поворот на 360 градусов, так что вычислить ответ от такого случая циклом просто не имеет смысла. Придумать пример к этому просто:

```
1 3 2 5 2
case 1
1
```

Частный случай 2: если количество дней больше вместимости контейнера. Сначала кажется, что это очень нестандартное условие для проверки. Да и так я не смог найти этого без наводящих подсказок от моих коллег. В чём дело: при таком условии есть только 2 варианта: у профессора остаётся либо бактерии в контейнере, либо не остаётся. Это происходит из-за того, что количество бактерий соответственно либо сводится к случаю аналогичный первому (где количество бактерий остаётся постоянной, но уже в контейнере, ибо излишка уничтожается по условию задачи), либо исчерпается (так как он использует всех сразу же). Значит, у этого случая есть 2 подслучая:

- Частный случай 2а: если количество бактерии после экспериментов меньше чем изначального. В таком случае ответом является 0 в любом случае:

  ```
  1 3 4 5 6
  case 2a
  0
  ```

- Частный случай 2б: если количество бактерии после экспериментов больше чем изначального. Тогда ответ равен вместимости контейнера (причина объяснена выше). Опять же, придумать пример к этому тривиальное дело:

  ```
  1 3 1 5 6
  case 2b
  5
  ```

Случай 3 — худший по времени случай, происходит только когда не выполнялись предыдущие 2 условия. Все примеры из Яндекс Контеста относятся к этому:

```
1 3 1 5 2   1 2 0 4 3   1 2 3 5 2
case 3      case 3      case 3
5           4           0
```

Таким образом я смог найти оптимальное (по моему мнению) решение и могу выручать за корректность данного алгоритма.

3. *Оценка сложности по времени:*

Обычно средняя сложность будет близка к худшему случаю, так как мы уже утвердили, что мы не можем уменьшить количество итераций без информации о том, когда значение бактерий начнёт совпадать с предыдущим. Поэтому, так как число итерации зависит от k, на который мы должны узнать количество бактерий, а узнать момент совпадения количества бактерий мы не можем без хотя бы нескольких итерации, количество итерации напрямую зависит от k, временная сложность должна составлять **O(k)**.

4. *Оценка сложности по памяти:*

Так как этот алгоритм использует только определённое количество переменных и не требует дополнительной аллокации памяти, сложность по памяти должна составлять **O(1)**.

**Задача №1296 «Гиперпрыжок»**

1. *Краткое описание алгоритма:*

Здесь требуется нахождение «максимального гравитационного потенциала», или же, подмассива с максимальной суммой. Решение такой задачи довольно тривиальное, так как существует множество предназначенных алгоритмов. Но если учитывать проблему временной сложности, то эти алгоритмы являются неоптимальными ($O(n^3)$, $O(n^2)$, $O(nlogn)$).

Поэтому я решил использовать алгоритм Кадана. Это такой алгоритм, где за линейное количество времени можно находить подмассив с наибольшей суммой.

Возьмём в качестве примера условие задачи из самого Тимуса:

| | |
|---|---|
| 10<br>31<br>-41<br>59<br>26<br>-53<br>58<br>97<br>-93<br>-23<br>84 | 187 |
| 3<br>-1<br>-5<br>-6 | 0 |

Суть алгоритма Кадана заключается в следующем: составляется новый подмассив T[1…n], где i-тый элемент равен наибольшей сумме подмассива, оканчивающегося в i. Можно отсюда считать T за O(n), а ответом будет наибольшее из T.

Подробнее:

Если мы знаем сумму наибольшего подмассива, оканчивающегося в i, то следующий элемент может либо расширять его, либо начать с нового подмассива.

Действительно, T[i + 1] может быть либо T[i] + x[i + 1], либо x[i + 1], либо 0, если x[i + 1] < 0. Если выразить по формуле, то получается следующее:

*T[0] = 0,*

*T[i + 1] = max{T[i] + x[i + 1], x[i + 1], 0} = max{T[i] + x[i + 1], 0}*

Сначала берётся нулевой элемент (x[0] == 31) в качестве максимальной суммы, а текущая сумма равен 0. После проверки обе равны 31.

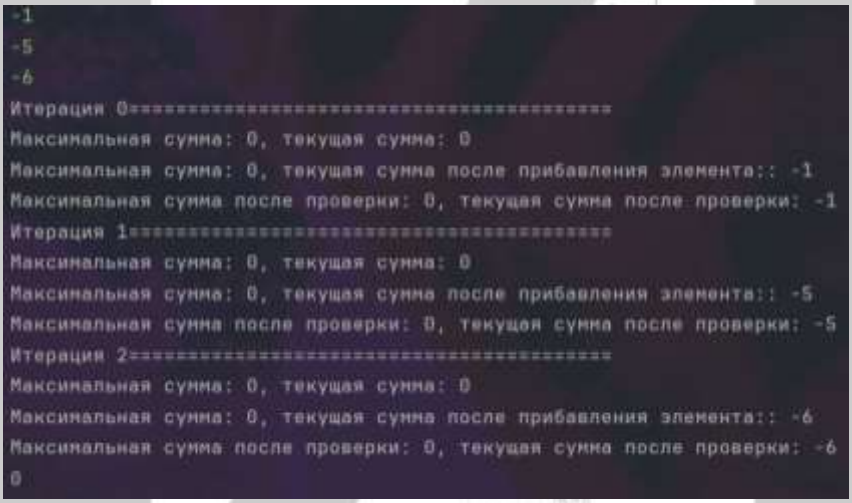
Вторая итерация: x[1] == –41. Здесь повеселее: после прибавления x[1] текущая сумма равняется -10, что меньше максимальной суммы 31, поэтому такое значение не присваивается.

Третья итерация: так как –10 < 0, то такое значение отбрасывается и от текущей суммы и заменяется нулём. Потом к этому прибавляется x[2] == 59, и так как 59 > 31, то такое значение присваивается максимальной сумме.

Это повторяется до тех пор, пока не находится максимальное значение подмассива — 187, а так как это и есть максимальное значение от всего массива, то его уже не вытеснить.

Так как нам нужно вывести только значение максимальной суммы, а не индексы данного подмассива, то наша задача на этом и заканчивается.

Повторяем этот процесс с другим примером:




И в итоге есть не просто решение, а ещё и оптимальное решение данной задачи.

*Исходный код:*

```cpp
#include <bits/stdc++.h>
using namespace std;

int kadanes(const int* x, size_t a){
    int maxSum = (x[0] < 0) ? 0 : x[0], curSum = 0;
    for (int i = 0; i < a; i++){
        curSum = max(curSum, 0);
        curSum += x[i];
```

```
        maxSum = max(maxSum, curSum);
    }
    return maxSum;
}
int main() {
    int a;
    cin >> a;
    int x[a];
    for(int i = 0; i < a; i++){
        cin >> x[i];
    }
    cout << kadanes(x, a) << endl;
    return 0;
}
```

2. *Корректность применяемого алгоритма:*

Корректность данного алгоритма доказывается с помощью математической индукции.

n = 1 — тривиальный случай.

При n = i: допустим, что алгоритм Кадана возвращает значение подмассива с максимальной суммой. Под этим допущением надо доказывать, что такое является действительной и в случае n = i + 1.

Обозначим функцию f(i) = max(sigma(0 -> i), sigma(1 -> i), …, sigma(i -> i)), где sigma(n -> i) — сумма всех элементов по индексу от n до I, а сама функция возвращает максимальное значение из всех сумм подмассивов.

При n = i + 1: Введём дополнительное равенство:

sigma(n -> i + 1) = sigma(n -> i) + element(i + 1),

где element(i + 1) — элемент массива на индексе i + 1.

После введения этого равенства получается следующее:

f(i + 1) = max(sigma(0 -> i) + element(i), sigma(1 -> i) + element(i + 1), …, sigma(i -> i) + element(i + 1), element(i + 1)) = max(max(sigma(0 -> i) + element(i + 1), sigma(1 -> i) + element(i + 1), …, sigma(i -> i) + element(i + 1)), element(i + 1)).

Значит,

f(i + 1) = max(max(sigma(0 -> i) + element(i + 1), sigma(1 -> i) + element(i + 1), …, sigma(i -> i) + element(i + 1)), element(i + 1)) = max(f(i) + element(i + 1), element(i + 1)),

так как

max(a + x, b + x, c + x) = max(a, b, c) + x.

Это является доказательством того, что алгоритм Кадана возвращает значение максимальной суммы подмассива из массива любой длины.

3. *Оценка сложности по времени:*

Так как этому алгоритму требуется только один проход через массив, сложность по времени должна составлять **O(n).**

4. *Оценка сложности по памяти:*

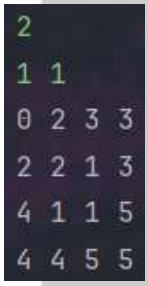
Так как этот алгоритм использует только 2 переменных (maxSum и curSum) и не требует дополнительной аллокации памяти, сложность по памяти должна составлять **O(1)**.

**Задача №1401 «Игроки»**

1. *Краткое объяснение алгоритма:*

Сначала рассматривается тот самый тривиальный случай, потом по принципу «разделяй и властвуй» доска разбивается на 4 до приведения к тривиальному случаю с помощью рекурсии. В соответствии с условием задачи, в программе есть счётчик, который используется в качестве значения фигурки.

К сожалению, мой алгоритм даёт не совсем похожий на пример ответ, но он всё равно прошёл все тесты на Тимусе:




| | |
|---|---|
| 2<br>1 1 | 0 1 3 3<br>1 1 4 3<br>2 4 4 5<br>2 2 5 5 |

*Исходный код:*

```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;

int cnt = 3, arr[512][512];
void place(int sz, int x, int y, int nx, int ny){
    if(sz == 2) {
        for(int i = 0; i < 2; i++){
            for(int j = 0; j < 2; j++){
                if(x + i != nx || y + j != ny){
                    arr[x + i][y + j] = cnt++ / 3;
                }
            }
        }
        return;
    }
    int hsz = sz / 2;
    for(int i = 0; i < 2; i++){
        for(int j = 0; j < 2; j++){
            if(x + i * hsz > nx || nx >= x + hsz * (i + 1) || y + j * hsz > ny
|| ny >= y + hsz * (j + 1)){
                arr[x + hsz + i - 1][y + hsz + j - 1] = cnt++ / 3;
            }
```

```
        }
    }
    for(int i = 0; i < 2; i++){
        for(int j = 0; j < 2; j++){
            (x + i * hsz <= nx && nx < x + hsz * (i + 1) && y + j * hsz <= ny &&
ny < y + hsz * (j + 1)) ?
            place(hsz, x + i * hsz, y + j * hsz, nx, ny) :
            place(hsz, x + i * hsz, y + j * hsz, x + hsz + i - 1, y + hsz + j -
1);
        }
    }
}
int main(){
    int n, x, y;
    cin >> n >> x >> y;
    int s = (int) pow(2, n);
    place(s, 0, 0, x - 1, y - 1);
    for (int i = 0; i < s; i++) {
        for (int j = 0; j < s; j++){
            cout << arr[i][j] << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

2. *Корректность применяемого алгоритма:*

Здесь принцип довольно простая: надо делить всю доску на фрагменты, чтобы их потом можно было по одному заполнить следующими фигурами:

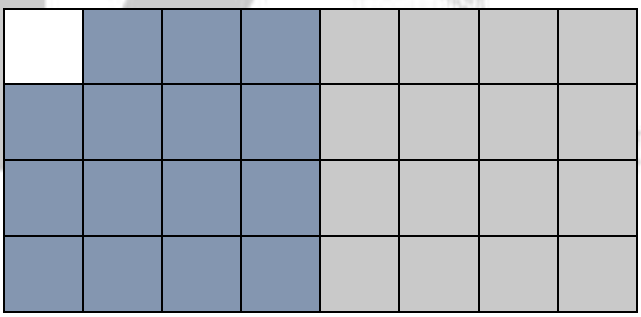
```
X      XX      X      XX
XX     X      XX       X
```

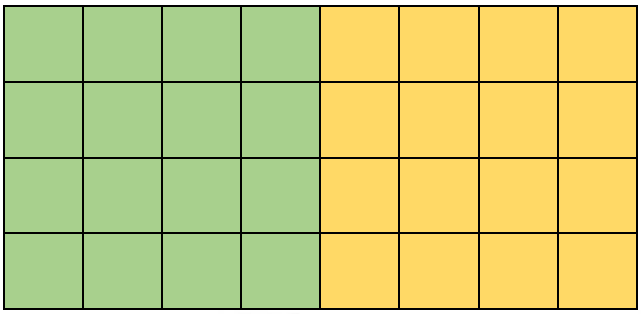
Из задачи видно: чтобы решать задачу, нам надо доказывать утверждение Чичикова математическим образом. Я решил доказать утверждение с помощью индукции:

При n = 1 всё очевидно.

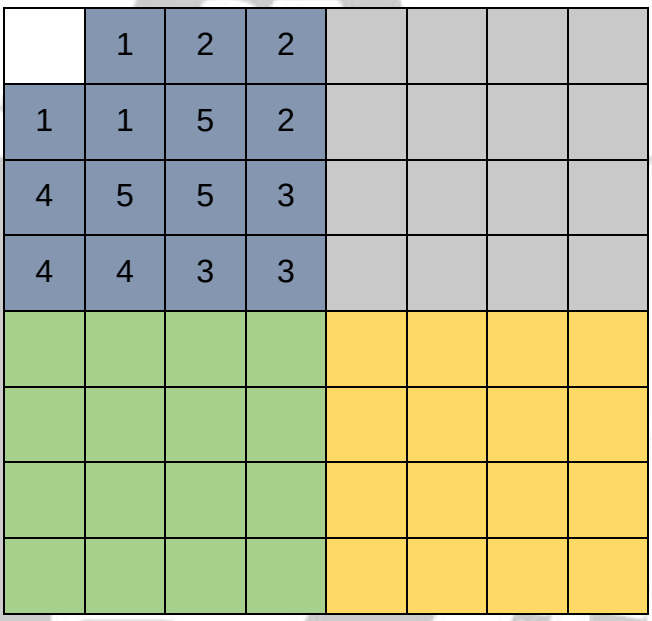
При n = k: допустим, что мы можем заполнить доску $2^k \times 2^k$ вышеупомянутыми фигурами, пропустив одну клетку. Под этим допущением надо доказывать, что такое является действительной и в случае n = k + 1.

При n = k + 1: требуется доказывать, что утверждение является истинной и в случае $2^{k+1} \times 2^{k+1}$. Разделим доску на 4 одинаковых:

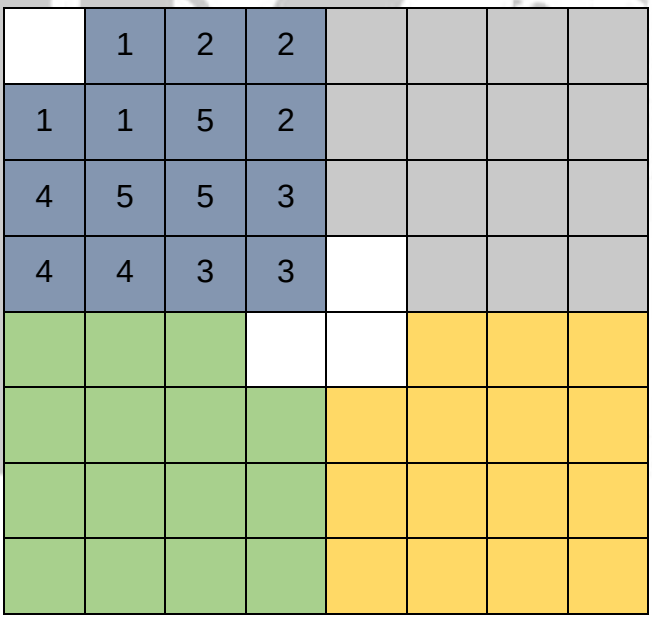




«Дырка» существует только в одном из 4 фрагментов, отсюда из n = k по индукцию будем предположить, что его можно заполнить фигурами:




Временно будем убрать 3 клетки из «целых» фрагментов:




И отсюда по индукцию понятно, что их теперь также можно заполнить фигурами:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 2 | 19 | 19 | 18 | 18 |
| 1 | 1 | 5 | 2 | 19 | 20 | 20 | 18 |
| 4 | 5 | 5 | 3 | 16 | 16 | 20 | 17 |
| 4 | 4 | 3 | 3 | 21 | 16 | 17 | 17 |
| 7 | 7 | 6 | 21 | 21 | 11 | 12 | 12 |
| 7 | 10 | 6 | 6 | 11 | 11 | 15 | 12 |
| 8 | 10 | 10 | 9 | 14 | 15 | 15 | 13 |
| 8 | 8 | 9 | 9 | 14 | 14 | 13 | 13 |

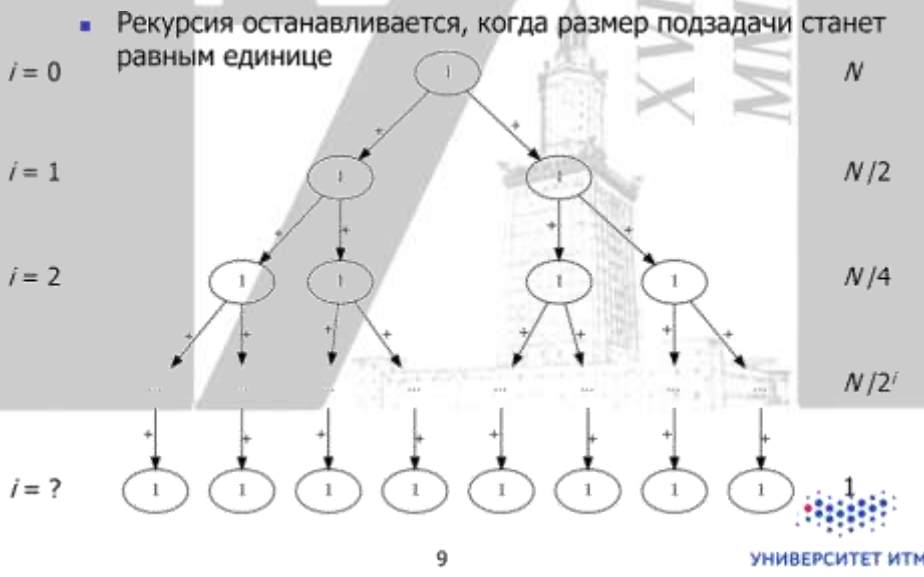
В конце, временная дырка является фигуркой. Следовательно, по математической индукции доказано, что любую доску с размером $2^n \times 2^n$, $n \in N$, можно заполнить фигурами с исключением одной клетки.

Этот алгоритм является имплементацией данного доказательства.

3. *<u>Оценка сложности по времени:</u>*

Здесь используется рекурсивный алгоритм, поэтому для оценки сложности по времени требуется использование особых методов.

Хорошим подходом для решения этой проблемы является составление дерева рекурсий. Мы уже можем найти предварительную оценку времени исходя из самого алгоритма: T(N) = 2T(N/2) + C. Оценка сложности в такой форме ничего не будет представлять из себя, так что теперь сможем обратиться к дереву рекурсии.




Здесь не совсем похожее дерево, у нас константа C, а не 1

Если выразить оценку в соответствии с деревом рекурсии:

$T(N) = 2T(N/2) + C = 4T(N/4) + C = 8T(N/8) + C = \ldots = 2^{N-1} * T(N/2^{N-1}) +$

$+ C => 2^{N-1} * T(1) + C.$

На нижнем уровне N узлов (подзадач) каждая со временем выполнения T (1), поэтому время работы этого уровня Θ(N). В итоге:

$T(N) = C + 2C + 4C + 8C + \ldots + 2^{\log N - 1}C + \Theta(N) = C(2^{\log N} - 1) / (2 - 1) + \Theta(N) =$

$= C*N - 1 + \Theta(N).$

**$T(N) = \Theta(N)$**.

4. *<u>Оценка сложности по памяти:</u>*

Так как мы уже определили сложности по времени через дерево рекурсии, мы можем сделать такую же и со сложностью по памяти. Так как каждый вызов рекурсивной функции занимает определённую память на стеке вызова, и максимальная глубина дерева (т.е. количество, на который мы можем разделить дерево на несколько веток) равна N, то оценка сложности по памяти должна составлять **O(n)**.