ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа бакалавриата «Программная инженерия»

СОГЛАСОВАНО
Программист
ООО «ИНТЕЛЛИДЖЕЙ ЛАБС»
(JetBrains)

_____ ___ С.В. Булгаков
«_13_» ____мая_____ 2022 г.

УТВЕРЖДАЮ
Академический руководитель
образовательной программы
«Программная инженерия»
профессор департамента
программной инженерии, канд. техн.
наук

_____ В.В. Шилов
«_13_» ____мая_____ 2022 г.

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

Генератор карт для Pocket Palm Heroes

Текст программы
ЛИСТ УТВЕРЖДЕНИЯ
RU.17701729.04.01-01 ТЗ 01-1 ЛУ

Исполнитель студент группы БПИ 203
_____ _____ /К. А. Веснин /

Москва 2022 г.

УТВЕРЖДЕН
RU.17701729.04.01-01 ТЗ 01-1ЛУ

Текст программы

RU.17701729.04.01-01 ТЗ 01-1

Листов 23

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

Москва 2022 г.

CellsAndTypes

Castle

```
package cellsAndTypes

class Castle(var type: CastleType, var size: CastleSize): Object(){
    constructor():this(CastleType.CITADEL, CastleSize.SMALL)
    enum class CastleType(val value:Int){
        CITADEL(0), STRONGHOLD(1), TOWER(2), DUNGEON(3), FORTRESS(4),
NECROPOLIS(5);
        companion object {
            fun fromInt(value: Int) = CastleType.values().first { it.value ==
value }
        }
    }
    enum class CastleSize(val value:Int){
        SMALL(0), MEDIUM(1), LARGE(2);
        companion object {
            fun fromInt(value: Int) = CastleSize.values().first { it.value ==
value }
        }
    }

    override fun getInt():Int {
        return type.value * 10 + size.value
    }

    companion object {
        fun getCastleByInt(n:Int) : Castle {
            val size = CastleSize.fromInt(n % 10)
            val type = CastleType.fromInt(n / 10)
            return Castle(type, size)
        }
    }
}
```

Cell

```
package cellsAndTypes

public class Cell(t: TypeOfCell, terr: TypeOfTerrain, creeps: TypeOfCreeps,
xy:Pair<Int,Int>) {
    constructor(xy:Pair<Int,Int>) : this(TypeOfCell.LAND, TypeOfTerrain.NO,
TypeOfCreeps.NO, xy)
    constructor(t: TypeOfCell, terr: TypeOfTerrain, creeps: TypeOfCreeps,
                xy:Pair<Int,Int>, enum:Int):this(t, terr, creeps, xy){
                    obj = when(t){
                        TypeOfCell.ROAD -> Road.getRoadByInt(enum)
                        TypeOfCell.CASTLE -> Castle.getCastleByInt(enum)
                        else -> Ownerable.getOwnerableByInt(enum) //
TypeOfCell.VISITABLE
                    }
                }
    var type: TypeOfCell
    var terr: TypeOfTerrain
    var creeps: TypeOfCreeps
    var obj: Object
    var isTaken:Boolean
```

```
    val xy:Pair<Int,Int>
    init{
        this.type = t
        this.terr = terr
        this.creeps = creeps
        obj = when(t){
            TypeOfCell.ROAD -> Road()
            TypeOfCell.CASTLE -> Castle()
            TypeOfCell.DECORATION -> Decoration()
            TypeOfCell.MAP_ITEM -> MapItem()
            else -> Ownerable() // TypeOfCell.VISITABLE
        }
        this.xy = xy
        isTaken = type != TypeOfCell.LAND
    }

    enum class TypeOfCell{
        ROAD, CASTLE, LAND, OWNERABLE, DECORATION, MAP_ITEM
    }
    enum class TypeOfTerrain(val value:Int){
        NO(0), LIGHT_SAND(1), SOIL(2), GRASS(3), DARK_GRASS(4), VOLCANO(5),
        DIRT(6), SNOW(7), SAND(8), GOLD(9), STONE(10);
        companion object {
            fun fromInt(value: Int) = TypeOfTerrain.values().first { it.value
== value }
        }
    }
    enum class TypeOfCreeps(val value:Int){
        NO(0), VERY_WEEK(1), WEEK(2), NORMAL(3),
        STRONG(4), VERY_STRONG(5), INSANE(6);
        companion object {
            fun fromInt(value: Int) = TypeOfCreeps.values().first { it.value
== value }
        }
    }
}
```

Decoration

```
package cellsAndTypes

class Decoration(var type:TypeOfDecoration):Object() {
    constructor():this(TypeOfDecoration.SMALL)

    enum class TypeOfDecoration(val value:Int){
        SMALL(0), BIG(1);
        companion object {
            fun fromInt(value: Int) = TypeOfDecoration.values().first {
it.value == value }
        }
    }

    override fun getInt():Int {
        return type.value
    }

    companion object {
        fun getDecorationByInt(n:Int) : Decoration {
            val type = TypeOfDecoration.fromInt(n)
            return Decoration(type)
```

```
        }
    }
}
```

MapItem

```
package cellsAndTypes

class MapItem(var type:TypeOfItem):Object() {
    constructor():this(TypeOfItem.RANDOM_RES)
    enum class TypeOfItem(val value:Int){
        RANDOM_RES(0), CAMPFIRE(2), TREASURE(3), ARTIFACT(4);
        companion object {
            fun fromInt(value: Int) = TypeOfItem.values().first { it.value ==
value }
        }
    }

    override fun getInt():Int {
        return type.value
    }

    companion object {
        fun getItemByInt(n:Int) : MapItem{
            val type = TypeOfItem.fromInt(n)
            return MapItem(type)
        }
    }
}
```

Object

```
package cellsAndTypes

abstract class Object() {
    abstract fun getInt():Int
}
```

Ownerable

```
package cellsAndTypes

class Ownerable(var type: TypeOfOwnerable): Object() {
    constructor():this(TypeOfOwnerable.GOLD_MINE)
    enum class TypeOfOwnerable(val value:Int){
        GOLD_MINE(0), ORE_PIT(1), WOOD_SAWMILL(2), ALCHEMIST_LAB(3),
        GEMS_MINE(4), CRYSTAL_MINE(5), SULFUR_MINE(6);
        companion object {
            fun fromInt(value: Int) = TypeOfOwnerable.values().first {
it.value == value }
        }
    }

    override fun getInt(): Int {
        return type.value
    }
```

```
    companion object {
        fun getOwnerableByInt(n:Int) : Ownerable {
            val type = TypeOfOwnerable.fromInt(n)
            return Ownerable(type)
        }
    }
}
```

Road

```
package cellsAndTypes

class Road(var type: TypeOfRoad): Object() {
    constructor():this(TypeOfRoad.OK)
    enum class TypeOfRoad(val value:Int){
        OK(0), HARD(1), VERY_HARD(2);
        companion object {
            fun fromInt(value: Int) = TypeOfRoad.values().first { it.value ==
value }
        }
    }

    override fun getInt():Int {
        return type.value
    }

    companion object {
        fun getRoadByInt(n:Int) : Road {
            val type = TypeOfRoad.fromInt(n)
            return Road(type)
        }
    }
}
```

Generator

BytesBuffer

```
import cellsAndTypes.*
import java.io.File

class BytesBuffer(val field:Field) {
    fun writeToBytes(){
        val bytesBuff:MutableList<Byte> = ArrayList()

        // EMAP_FILE_HDR_KEY
        uIntToByteArray(BytesConstants.EMAP_FILE_HDR_KEY).forEach {
bytesBuff.add(it) }
        // EMAP_FILE_VERSION
        uIntToByteArray(BytesConstants.EMAP_FILE_VERSION).forEach {
bytesBuff.add(it) }
        // map.m_Siz
        bytesBuff.add(getMapSiz())
        // map.m_lngMask
        BytesConstants.m_lngMask.forEach{bytesBuff.add(it)}
        // text resources count
        BytesConstants.textRes.forEach{bytesBuff.add(it)}
        // map.m_MapVersion and map_MapAuthor
        BytesConstants.mapVersAndAuthor.forEach{bytesBuff.add(it)}

        // Players
        val players = field.getPlayersCastles()
        uInt16ToByteArray(players.size.toUInt()).forEach { bytesBuff.add(it)
}
        for(i in 0 until players.size){
            bytesBuff.add(i.toByte()) // m_PlayerId
            bytesBuff.add(2) // m_PlayerTypeMask
            bytesBuff.add(1) // hasMainCastle
            uInt16ToByteArray(players[i].first.toUInt()).forEach {
bytesBuff.add(it) }
            uInt16ToByteArray(players[i].second.toUInt()).forEach {
bytesBuff.add(it) }
            bytesBuff.add(1) // create a hero here
        }

        val heroes_size = 0u
        uInt16ToByteArray(heroes_size).forEach { bytesBuff.add(it) }

        // Map Items
        val mapItems:MutableList<Cell> = field.getMapItemCells()
        uInt16ToByteArray(mapItems.size.toUInt()).forEach { bytesBuff.add(it)
}
        mapItems.forEach { item ->
            bytesBuff.add(item.obj.getInt().toByte())
            uInt16ToByteArray(item.xy.first.toUInt()).forEach {
bytesBuff.add(it) }
            uInt16ToByteArray(item.xy.second.toUInt()).forEach {
bytesBuff.add(it) }
            for(i in 1..7)
                listOf<Byte>(-1, -1, 0, 0, 0, 0).forEach { bytesBuff.add(it)
}
            val lastListOfByte = when(item.obj.getInt()){
```

```
                0 -> listOf<Byte>(0, 0, 0, 0, -1, 0, 0, 0, 0)
                else -> listOf<Byte>(0,0,0,0) // 2,3
            }
            lastListOfByte.forEach { bytesBuff.add(it) }
            if(item.obj.getInt() == MapItem.TypeOfItem.ARTIFACT.value){
                BytesConstants.randomArtId.forEach{bytesBuff.add(it)}
            }
        }

        // Guard
        //uInt16ToByteArray(0u).forEach { bytesBuff.add(it) }
        val guards = field.getGuardCells()
        uInt16ToByteArray(guards.size.toUInt()).forEach { bytesBuff.add(it) }
        guards.forEach{
            getGuardByte(it.creeps, it.xy).forEach{bytesBuff.add(it)}
        }

        val mapEvents_size = 0u
        uInt16ToByteArray(mapEvents_size).forEach { bytesBuff.add(it) }
        val visitables_size = 0u
        uInt16ToByteArray(visitables_size).forEach { bytesBuff.add(it) }

        // Mines
        //uInt16ToByteArray(0u).forEach { bytesBuff.add(it) }
        val ownerables = field.getOwnerableCells()
        uInt16ToByteArray(ownerables.size.toUInt()).forEach {
bytesBuff.add(it) }
        ownerables.forEach {
            getOwnerableByte(it.obj.getInt(), it.xy).forEach {
bytesBuff.add(it) }
        }

        // Castles
        //uInt16ToByteArray(0u).forEach { bytesBuff.add(it) }
        val castles = field.getCastleCells()
        uInt16ToByteArray(castles.size.toUInt()).forEach { bytesBuff.add(it)
}
        castles.forEach{
            var owner:Byte = -1
            if(field.getPlayersCastles().contains(it.xy)) owner =
field.getPlayersCastles().indexOf(it.xy).toByte()
            getCastleByte(it.obj.getInt(), getCastleIdOfType(it), it.xy,
owner).forEach { bytesBuff.add(it) }
        }

        // Map Dump
        getMapDumpByteArray().forEach { bytesBuff.add(it) }

        // Decorations
        val decorations = field.getDecorationCells()
        uIntToByteArray(decorations.size.toUInt()).forEach{bytesBuff.add(it)}
        decorations.forEach { decoration ->
            uInt16ToByteArray(decoration.xy.first.toUInt()).forEach {
bytesBuff.add(it) }
            uInt16ToByteArray(decoration.xy.second.toUInt()).forEach {
bytesBuff.add(it) }
            genDecorationId(decoration).forEach { bytesBuff.add(it) }
        }
        // Roads
        //uIntToByteArray(0u).forEach { bytesBuff.add(it) }
        var allRoadPoints = field.getAllRoadPoints()
        uIntToByteArray(allRoadPoints.size.toUInt()).forEach {
```

```
bytesBuff.add(it) }
        allRoadPoints.forEach {
            uInt16ToByteArray(it.xy.first.toUInt()).forEach {
bytesBuff.add(it) }
            uInt16ToByteArray(it.xy.second.toUInt()).forEach {
bytesBuff.add(it) }
            when(it.obj.getInt()){
                Road.TypeOfRoad.OK.value ->
BytesConstants.roadOK_id.forEach{bytesBuff.add(it)}
                Road.TypeOfRoad.HARD.value, Road.TypeOfRoad.VERY_HARD.value -
>
                    BytesConstants.roadOK_id.forEach{bytesBuff.add(it)}
                else -> BytesConstants.roadOK_id.forEach{bytesBuff.add(it)}
            }
        }
        ////showBytesBuff(bytesBuff)
        val arr:ByteArray = ByteArray(bytesBuff.size)
        for(i in 0 until bytesBuff.size){
            arr[i] = bytesBuff[i]
        }
        File("map.hmm").writeBytes(arr)
    }


    fun showBytesBuff(bytesBuff:MutableList<Byte>){
        for(i in 0 until bytesBuff.size){
            print("${Integer.toHexString(bytesBuff[i].toInt())} ")
            if((i + 1) % 4 == 0)
                print("    ")
            if((i + 1) % 16 == 0)
                println()
        }
    }

    fun genDecorationId(cell:Cell):List<Byte>{
        if(cell.obj.getInt() == Decoration.TypeOfDecoration.SMALL.value){
            return when(cell.terr){
                Cell.TypeOfTerrain.GRASS, Cell.TypeOfTerrain.DARK_GRASS ->
                    listOf(BytesConstants.flowersId,
BytesConstants.pinesId).random()
                Cell.TypeOfTerrain.LIGHT_SAND, Cell.TypeOfTerrain.SAND ->
                    listOf(BytesConstants.palmId,
BytesConstants.cactusId).random()
                Cell.TypeOfTerrain.DIRT, Cell.TypeOfTerrain.SOIL ->
                    //listOf(BytesConstants.rockId,
BytesConstants.rockWithPlantId).random()
                    listOf(BytesConstants.flowersId,
BytesConstants.pinesId).random()
                Cell.TypeOfTerrain.STONE, Cell.TypeOfTerrain.VOLCANO ->
                    listOf(BytesConstants.deadPlantId).random()
                else -> listOf(BytesConstants.snowTreeId).random() // snow
            }
        }
        return BytesConstants.snowTreeId
    }

    fun getOwnerableByte(numType:Int, pos:Pair<Int,Int>):MutableList<Byte>{
        val ownerableBytes:MutableList<Byte> = ArrayList()
        val thisId:List<Byte> =
when(Ownerable.TypeOfOwnerable.fromInt(numType)){
            Ownerable.TypeOfOwnerable.GOLD_MINE -> BytesConstants.goldId
            Ownerable.TypeOfOwnerable.ORE_PIT -> BytesConstants.oreId
```

```
            Ownerable.TypeOfOwnerable.WOOD_SAWMILL -> BytesConstants.woodId
            Ownerable.TypeOfOwnerable.ALCHEMIST_LAB ->
BytesConstants.alchemistId
            Ownerable.TypeOfOwnerable.GEMS_MINE -> BytesConstants.gemsId
            Ownerable.TypeOfOwnerable.CRYSTAL_MINE ->
BytesConstants.crystalId
            Ownerable.TypeOfOwnerable.SULFUR_MINE -> BytesConstants.sulfurId
        }
        thisId.forEach{ ownerableBytes.add(it)}
        ownerableBytes.add(-1) // owner
        uInt16ToByteArray(pos.first.toUInt()).forEach {
ownerableBytes.add(it) }
        uInt16ToByteArray(pos.second.toUInt()).forEach {
ownerableBytes.add(it) }
        for(i in 1..7)
            listOf<Byte>(-1, -1, 0, 0, 0, 0).forEach { ownerableBytes.add(it)
}
        return ownerableBytes
    }



    fun getGuardByte(force: Cell.TypeOfCreeps,
pos:Pair<Int,Int>):MutableList<Byte>{
        val guardBytes:MutableList<Byte> = ArrayList()
        guardBytes.add(force.value.toByte())
        guardBytes.add(15) // random type of creatures
        listOf<Byte>(0, 0, 0, 0).forEach{guardBytes.add(it)}
        guardBytes.add(1) // disposition (true)
        guardBytes.add(0) // notGrow (false)
        uInt16ToByteArray(pos.first.toUInt()).forEach { guardBytes.add(it) }
        uInt16ToByteArray(pos.second.toUInt()).forEach { guardBytes.add(it) }
        listOf<Byte>(0, 0, 0, 0).forEach{guardBytes.add(it)} // message
        return guardBytes
    }

    fun getCastleByte(typeSize:Int, idOfType:UInt, pos:Pair<Int,Int>,
owner:Byte):MutableList<Byte>{
        val castleBytes:MutableList<Byte> = ArrayList()
        val type = Castle.CastleType.fromInt(typeSize / 10)
        val size = Castle.CastleSize.fromInt(typeSize % 10)
        val thisId:List<Byte> = when(size){
            Castle.CastleSize.SMALL -> BytesConstants.castleSmallId
            Castle.CastleSize.MEDIUM -> BytesConstants.castleMediumId
            Castle.CastleSize.LARGE -> BytesConstants.castleLargeId
        }
        thisId.forEach{castleBytes.add(it)}
        uInt16ToByteArray(idOfType).forEach { castleBytes.add(it) }
        castleBytes.add(type.value.toByte())
        castleBytes.add(owner) // owner
        uInt16ToByteArray(pos.first.toUInt()).forEach { castleBytes.add(it) }
// pos
        uInt16ToByteArray(pos.second.toUInt()).forEach { castleBytes.add(it)
}
        for(i in 1..7){ // garrison
            listOf<Byte>(-1, -1, 0, 0, 0, 0).forEach{castleBytes.add(it)}
        }
        listOf<Byte>(0, 0, 0, 0).forEach{castleBytes.add(it)} // custom name
        uInt16ToByteArray(0u).forEach { castleBytes.add(it) } // isCustomized
        return castleBytes
    }
```

```
    fun getMapSiz():Byte{
        if(field.rows != field.cols) return -1
        when(field.rows - 1){
            32 -> return 0
            64 -> return 1
            128 -> return 2
            256 -> return 3
        }
        return -1
    }

    fun getCastleIdOfType(castle: Cell):UInt{
        val x = castle.xy.first
        val y = castle.xy.second
        if(x + 1 < field.rows && field.matr[x + 1][y].type ==
Cell.TypeOfCell.ROAD)
            return 52u
        else if(y + 1 < field.cols && field.matr[x][y+1].type ==
Cell.TypeOfCell.ROAD)
            return 51u
        else if(x - 1 >= 0 && field.matr[x-1][y].type ==
Cell.TypeOfCell.ROAD)
            return 52u
        else
            return 51u
    }

    fun uIntToByteArray(value:UInt):ByteArray{
        val bytes = ByteArray(4)
        bytes[0] = (value and 0xFFFFu).toByte()
        bytes[1] = (value.shr(8) and 0xFFFFu).toByte()
        bytes[2] = (value.shr(16) and 0xFFFFu).toByte()
        bytes[3] = (value.shr(24) and 0xFFFFu).toByte()
        return bytes
    }

    fun uInt16ToByteArray(value:UInt):ByteArray{
        val bytes = ByteArray(2)
        bytes[0] = (value and 0xFFFFu).toByte()
        bytes[1] = (value.shr(8) and 0xFFFFu).toByte()
        return bytes
    }

    fun getMapDumpByteArray():MutableList<Byte>{
        val byteArr:MutableList<Byte> = ArrayList()
        for(i in 0 until field.rows){
            for(j in 0 until field.cols){

uInt16ToByteArray(field.matr[i][j].terr.value.toUInt()).forEach {
byteArr.add(it) }
            }
        }
        return byteArr
    }
}
```

BytesConstants

```
class BytesConstants {
    companion object{
```

```
        val EMAP_FILE_HDR_KEY:UInt = 0x76235278u
        val EMAP_FILE_VERSION = 0x19u
        val m_lngMask = listOf<Byte>(1, 0, 0, 0)
        val textRes = listOf<Byte>(2, 0, 0, 0)
        val mapVersAndAuthor = listOf<Byte>(15, 0, 0, 0, 77, 0, 97, 0, 112,
0, 32, 0, 68, 0, 101, 0,
            115, 0, 99, 0, 114, 0, 105, 0, 112, 0, 116, 0, 105, 0, 111, 0,
            110, 0, 2, 0, 19, 0, 0, 0, 68, 0, 101, 0, 102, 0, 97, 0,
            117, 0, 108, 0, 116, 0, 32, 0, 100, 0, 101, 0, 115, 0, 99, 0,
            116, 0, 105, 0, 112, 0, 116, 0, 105, 0, 111, 0, 110, 0, 8, 0,
            0, 0, 77, 0, 97, 0, 112, 0, 32, 0, 110, 0, 97, 0, 109, 0,
            101, 0, 1, 0, 7, 0, 0, 0, 78, 0, 101, 0, 119, 0, 32, 0,
            77, 0, 97, 0, 112, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            -1, -1, -1, -1, 0, 0, 0, 0)
        val goldId = listOf<Byte>(9, 0, 0, 0, 103, 0, 111, 0, 108, 0, 100,
0,95, 0, 109, 0, 105, 0, 110, 0, 101, 0)
        val oreId = listOf<Byte>(8, 0, 0, 0, 111, 0, 114, 0, 101, 0, 95, 0,
109, 0, 105, 0, 110, 0, 101, 0)
        val woodId = listOf<Byte>(12, 0, 0, 0, 119, 0, 111, 0, 111, 0, 100,
0, 95, 0, 115, 0,
            97, 0, 119, 0, 109, 0, 105, 0, 108, 0, 108, 0)
        val alchemistId = getBytesByStr("0b 00 00 00 6d 00 65 00 72 00 63 00
75 " +
                "00 72 00 79 00 5f 00 6c 00 61 00 62 00")
        val gemsId = getBytesByStr("09 00 00 00 67 00 65 00 6d 00 73 00 5f 00
6d 00 " +
                "69 00 6e 00 65 00")
        val crystalId = getBytesByStr("0c 00 00 00 63 00 72 00 79 00 73 00 74
00 61 00 " +
                "6c 00 5f 00 6d 00 69 00 6e 00 65 00")
        val sulfurId = getBytesByStr("0b 00 00 00 73 00 75 00 6c 00 66 00 75
00 72 00 " +
                "5f 00 6d 00 69 00 6e 00 65 00")
        val roadOK_id = listOf<Byte>(10, 0, 0, 0, 115, 0, 116, 0, 111, 0,
110, 0, 101, 0, 95, 0,
            114, 0, 111, 0, 97, 0, 100, 0)
        val roadHard_id = listOf<Byte>(9, 0, 0, 0, 100, 0, 105, 0, 114, 0,
116, 0, 95, 0, 114, 0,
            111, 0, 97, 0, 100, 0)
        val castleSmallId = listOf<Byte>(8, 0, 0, 0, 115, 0, 109, 0, 97, 0,
108, 0, 108, 0, 95, 0,
            48, 0)
        val castleMediumId = listOf<Byte>(9, 0, 0, 0, 109, 0, 101, 0, 100, 0,
105, 0, 117, 0, 109, 0,
            95, 0, 48, 0)
        val castleLargeId = listOf<Byte>(8, 0, 0, 0, 108, 0, 97, 0, 114, 0,
103, 0, 101, 0, 95, 0,
            48, 0)
        val palmId = getBytesByStr("07 00 00 00 70 00 61 00 6c 00 6d 00 73 00
5f 00 " +
                "38 00")
        val flowersId = getBytesByStr("09 00 00 00 66 00 6c 00 6f 00 77 00 65
00 72 00 " +
                "73 00 5f 00 33 00")
        val pinesId = getBytesByStr("07 00 00 00 70 00 69 00 6e 00 65 00 73
00 5f 00 " +
                "31 00")
        val rockId = getBytesByStr("08 00 00 00 72 00 6f 00 63 00 6b 00 73 00
5f 00 " +
                "32 00 38 00")
        val deadPlantId = getBytesByStr("09 00 00 00 64 00 70 00 6c 00 61 00
6e 00 74 00 73 " +
```

```
                "00 5f 00 32 00")
        val rockWithPlantId = getBytesByStr("08 00 00 00 72 00 6f 00 63 00 6b
00 73 00 5f 00 " +
                "31 00 37 00")
        val snowTreeId = getBytesByStr("09 00 00 00 64 00 70 00 6c 00 61 00
6e 00 74 00 " +
                "73 00 5f 00 39 00")
        val cactusId = getBytesByStr("08 00 00 00 63 00 61 00 63 00 74 00 75
00 73 00 5f " +
                "00 35 00")
        val randomArtId = getBytesByStr("0e 00 00 00 52 00 61 00 6e 00 64 00
6f 00 6d 00 " +
                "41 00 72 00 74 00 69 00 66 00 61 00 63 00 74 00")
        fun getBytesByStr(nums:String):MutableList<Byte>{
            val bytes:MutableList<Byte> = ArrayList()
            for(i in 0 until nums.length - 1 step 3){
                val n =nums.subSequence(i, i+2)
                val sum = getIntBySymHEX(n[1]) + getIntBySymHEX(n[0]) * 16
                bytes.add(sum.toByte())
            }
            return bytes
        }
        fun getIntBySymHEX(c:Char):Int{
            return when(c){
                'a' -> 10
                'b' -> 11
                'c' -> 12
                'd' -> 13
                'e' -> 14
                'f' -> 15
                else -> c.toString().toInt()
            }
        }
    }
}
```

Field

```
import cellsAndTypes.Cell
import cellsAndTypes.Decoration
import cellsAndTypes.Road
import generator.CellTypeGenerator

class Field(rows:Int, cols:Int) {
    val rows:Int
    val cols:Int
    val matr:MutableList<MutableList<Cell>> = ArrayList()
    private val roadPoints:MutableList<Pair<Int,Int>> = ArrayList()
    private val castlePoints:MutableList<Pair<Int,Int>> = ArrayList()
    init{
        this.rows = rows
        this.cols = cols
        for(i in 0 until rows){
            matr.add(ArrayList<Cell>())
            // matr[i] = ArrayList<TypeOfCell>(cols)
            for(j in 0 until cols){
                // matr[i][j] = TypeOfCell.LAND
                matr[i].add(Cell(Pair(i,j)))
            }
        }
```

```
    }

    fun getAllPointsBetween(f:Pair<Int, Int>, s:Pair<Int, Int>,
                            notRoad:Pair<Int,Int> = Pair(-1,-
1)):MutableList<Pair<Int,Int>>{
        val allPoints:MutableList<Pair<Int,Int>> = ArrayList()
        if(f.first > s.first){
            return getAllPointsBetween(s, f, notRoad)
        }
        var jOld = f.second
        if(f.first == s.first){
            if(f.second <= s.second) {
                for (j in f.second..s.second)
                    allPoints.add(Pair(f.first, j))
            } else{
                for (j in s.second..f.second)
                    allPoints.add(Pair(f.first, j))
            }
            //allPoints.removeAll{it.first == notRoad.first && it.second ==
notRoad.second}
            if(allPoints.contains(notRoad))
                allPoints.remove(notRoad)
            return allPoints
        }

        for(i in f.first..s.first){
            var jNew = f.second + ((s.second - f.second) * (i - f.first) /
(s.first - f.first)).toInt()
            if(jOld < jNew) {
                for (j in jNew downTo jOld) {
                    allPoints.add(Pair(i, j))
                }
            } else{
                for (j in jNew..jOld) {
                    allPoints.add(Pair(i, j))
                }
            }
            jOld = jNew
        }
        //allPoints.removeAll{it.first == notRoad.first && it.second ==
notRoad.second}
        if(allPoints.contains(notRoad))
            allPoints.remove(notRoad)
        return allPoints
    }

    fun connectTwoPoints(f:Pair<Int, Int>, s:Pair<Int, Int>, typeOfRoad:
Road.TypeOfRoad,
                         notRoad:Pair<Int,Int> = Pair(-1,-1)){
        getAllPointsBetween(f,s, notRoad).forEach {
            initRoad(it.first,it.second, typeOfRoad)
        }
    }

    fun findClosestRoad(p:Pair<Int,Int>, isDownLeft:Boolean =
false):Pair<Int,Int>{
        val maxDist = distanceSQ(Pair(0,0), Pair(rows - 1, cols - 1))
        var ans = Pair(-1,-1)
        var dist = maxDist
        if(isDownLeft){
            for(i in p.first until rows){
                for(j in p.second until cols){
```

```
                    if(matr[i][j].type == Cell.TypeOfCell.ROAD) {
                        val tempDist = distanceSQ(p, Pair(i,j))
                        if(tempDist < dist){
                            dist = tempDist
                            ans = Pair(i,j)
                        }
                    }
                }
            }
            if(dist != maxDist)
                return ans
        }
        for(i in 0 until rows){
            for(j in 0 until cols){
                if(matr[i][j].type == Cell.TypeOfCell.ROAD) {
                    val tempDist = distanceSQ(p, Pair(i,j))
                    if(tempDist < dist){
                        dist = tempDist
                        ans = Pair(i,j)
                    }
                }
            }
        }
        return ans
    }

    private fun initRoad(i:Int, j:Int, typeOfRoad: Road.TypeOfRoad){
        matr[i][j].type = Cell.TypeOfCell.ROAD
        matr[i][j].obj = Road(typeOfRoad)
        matr[i][j].isTaken = true
    }

    fun getMatrOfIsTaken():MutableList<MutableList<Boolean>>{
        val matrIsTaken:MutableList<MutableList<Boolean>> = ArrayList()
        for(i in 0 until rows){
            matrIsTaken.add(ArrayList())
            for(cell in matr[i]){
                matrIsTaken[i].add(cell.isTaken)
            }
        }
        return matrIsTaken
    }

    fun clear(){
        for(i in 0 until rows){
            matr.add(ArrayList<Cell>())
            // matr[i] = ArrayList<TypeOfCell>(cols)
            for(j in 0 until cols){
                // matr[i][j] = TypeOfCell.LAND
                matr[i][j] = Cell(Pair(i,j))
            }
        }
    }

    fun placeGraph(matrix:MutableList<MutableList<Int>>,
roadPoints:MutableList<Pair<Int,Int>>){
        for(i in 0 until matrix.size){
            this.roadPoints.add(roadPoints[i])
            for(j in matrix[i]){
                connectTwoPoints(roadPoints[i], roadPoints[j],
CellTypeGenerator.genTypeOfRoad())
            }
```

```
        }
    }

    fun setTerrainType(i:Int, j:Int, type:Cell.TypeOfTerrain){
        matr[i][j].terr = type
    }

    fun putAllTerrainType(){
        for(i in 0 until rows){
            for(j in 0 until cols){
                if(matr[i][j].terr == Cell.TypeOfTerrain.NO) {
                    matr[i][j].terr = getClosestTerrainCell(i, j).terr
                }
            }
        }
    }

    fun getClosestTerrainCell(i:Int,j:Int): Cell {
        val thisPair = Pair(i, j)
        var minDist = rows * rows + cols*cols
        var pair = Pair(i, j)
        for(p in castlePoints){
            if(matr[p.first][p.second].terr != Cell.TypeOfTerrain.NO){
                if(minDist > distanceSQ(thisPair,p)){
                    minDist = distanceSQ(thisPair, p)
                    pair = p
                }
            }
        }
        return matr[pair.first][pair.second]
    }

    fun distanceSQ(f:Pair<Int, Int>, s:Pair<Int, Int>):Int{
        val dif = Pair(s.first - f.first, s.second - f.second)
        return dif.first * dif.first + dif.second*dif.second
    }

    fun placeCastles(castlePoints:MutableList<Pair<Int,Int>>){
        for(p in castlePoints){
            val closestRoad = findClosestRoad(Pair(p.first,p.second), true)
            this.castlePoints.add(p)
            matr[p.first][p.second].type = Cell.TypeOfCell.CASTLE
            // for(i in p.first - 4..p.first + 4){
            //     for(j in p.second - 4..p.second +4){
            //         matr[i][j].isTaken = true
            //     }
            // }
            matr[p.first][p.second].obj = CellTypeGenerator.genCastle()

            connectTwoPoints(p, closestRoad,
CellTypeGenerator.genTypeOfRoad(), p)
        }
    }

    fun getClosestPointToTheRoad(start:Pair<Int,Int>,
closestRoad:Pair<Int,Int>, radius:Int):Pair<Int,Int>{
        var minDist = rows*rows + cols*cols
        var ans = Pair(-1,-1)
        val between = getAllPointsBetween(start, closestRoad)
        for(b in between){
            val tempDist = distanceSQ(b,closestRoad)
            if(isZoneOfPointsNotTaken(radius, b) && tempDist < minDist){
```

```
                minDist = tempDist
                ans = b
            }
        }
        return ans
    }

    fun isZoneOfPointsNotTaken(radius:Int, p:Pair<Int,Int>):Boolean{
        val x = p.first
        val y = p.second
        for (i in x - radius..x + radius) {
            for (j in y - radius..y + radius) {
                if (i < 0 || i >= rows || j < 0 || j >= cols  ||
matr[i][j].isTaken)
                    return false
            }
        }
        return true
    }

    fun isPointCorrectForBuilding(p:Pair<Int,Int>, radius:Int):Boolean{
        if(!isZoneOfPointsNotTaken(radius, p)) return false
        val closestRoad = findClosestRoad(p, true)
        val between = getAllPointsBetween(p, closestRoad, closestRoad)
        between.forEach { if(matr[it.first][it.second].isTaken) return false
}
        return true
    }

    fun placeDecorations(points:MutableList<Pair<Int,Int>>,
type:Decoration.TypeOfDecoration){
        points.forEach {
            matr[it.first][it.second].type = Cell.TypeOfCell.DECORATION
            matr[it.first][it.second].obj = Decoration(type)
        }
    }

    fun placeOwnerables(ownerablePoints:MutableList<Pair<Int,Int>>){
        // ownerablePoints.forEach {
        //     val x = it.first
        //     val y = it.second
        //     for (i in x - 2..x + 2) {
        //         for (j in y - 2..y + 2) {
        //             matr[i][j].isTaken = true
        //         }
        //     }
        // }
        for(o in ownerablePoints){
            for (i in o.first - 2..o.first + 2) {
                for (j in o.second - 2..o.second + 2) {
                    matr[i][j].isTaken = false
                }
            }
            val closestRoad = findClosestRoad(o, true)
            val cl = getClosestPointToTheRoad(o, closestRoad, 2)

            matr[cl.first][cl.second].type = Cell.TypeOfCell.OWNERABLE
            for(i in cl.first - 2..cl.first + 2){
                for(j in cl.second - 2..cl.second +2){
                    matr[i][j].isTaken = true
                }
            }
```

```
            matr[cl.first][cl.second].obj = CellTypeGenerator.genOwnerable()
            connectTwoPoints(cl, closestRoad,
CellTypeGenerator.genTypeOfRoad(), cl)
        }
    }

    fun placeCreep(creep:Pair<Int,Int>, type: Cell.TypeOfCreeps){
        matr[creep.first][creep.second].creeps = type
    }

    fun placeMapItems(items:MutableList<Pair<Int,Int>>){
        for(item in items){
            for (i in item.first - 1..item.first + 1) {
                for (j in item.second - 1..item.second + 1) {
                    matr[i][j].isTaken = false
                }
            }
            val closestRoad = findClosestRoad(item, true)
            val cl = getClosestPointToTheRoad(item, closestRoad, 1)

            matr[cl.first][cl.second].type = Cell.TypeOfCell.MAP_ITEM
            for(i in cl.first - 1..cl.first + 1){
                for(j in cl.second - 1..cl.second +1){
                    matr[i][j].isTaken = true
                }
            }
            matr[cl.first][cl.second].obj = CellTypeGenerator.genMapItem()
            connectTwoPoints(cl, closestRoad,
CellTypeGenerator.genTypeOfRoad(), cl)
        }
    }

    fun isNotConnectedWithRoad(p:Pair<Int,Int>):Boolean{
        for(i in maxOf(p.first - 1, 0)..minOf(p.first + 1, rows - 1)){
            for(j in maxOf(p.second - 1, 0)..minOf(p.second + 1, cols - 1)){
                if(matr[i][j].type == Cell.TypeOfCell.ROAD)
                    return false
            }
        }
        return true
    }

    fun getCastleCells():MutableList<Cell>{
        val castleCells:MutableList<Cell> = ArrayList()
        for(i in 0 until rows){
            for(j in 0 until cols){
                if(matr[i][j].type == Cell.TypeOfCell.CASTLE){
                    castleCells.add(matr[i][j])
                }
            }
        }
        return castleCells
    }

    fun getMapItemCells():MutableList<Cell>{
        val mapItemCells:MutableList<Cell> = ArrayList()
        for(i in 0 until rows){
            for(j in 0 until cols){
                if(matr[i][j].type == Cell.TypeOfCell.MAP_ITEM){
                    mapItemCells.add(matr[i][j])
                }
            }
```

```
        }
        return mapItemCells
    }

    fun getOwnerableCells():MutableList<Cell>{
        val ownerableCells:MutableList<Cell> = ArrayList()
        for(i in 0 until rows){
            for(j in 0 until cols){
                if(matr[i][j].type == Cell.TypeOfCell.OWNERABLE){
                    ownerableCells.add(matr[i][j])
                }
            }
        }
        return ownerableCells
    }

    fun getDecorationCells():MutableList<Cell>{
        val decorationCells:MutableList<Cell> = ArrayList()
        for(i in 0 until rows){
            for(j in 0 until cols){
                if(matr[i][j].type == Cell.TypeOfCell.DECORATION){
                    decorationCells.add(matr[i][j])
                }
            }
        }
        return decorationCells
    }
    fun getGuardCells():MutableList<Cell>{
        val guardCells:MutableList<Cell> = ArrayList()
        for(i in 0 until rows){
            for(j in 0 until cols){
                if(matr[i][j].creeps != Cell.TypeOfCreeps.NO){
                    guardCells.add(matr[i][j])
                }
            }
        }
        return guardCells
    }

    fun getAllRoadPoints():MutableList<Cell>{
        val allRoadPoints:MutableList<Cell> = ArrayList()
        for(i in 0 until rows){
            for(j in 0 until cols){
                if(matr[i][j].type == Cell.TypeOfCell.ROAD ||
                    matr[i][j].type == Cell.TypeOfCell.CASTLE ||
matr[i][j].type == Cell.TypeOfCell.OWNERABLE){
                    allRoadPoints.add(matr[i][j])
                }
            }
        }
        return allRoadPoints
    }

    private val playersCastles:MutableList<Pair<Int,Int>> = ArrayList()
    fun setPlayersCastles(castles:MutableList<Pair<Int,Int>>){
        playersCastles.clear()
        castles.forEach { playersCastles.add(it) }
    }
    fun getPlayersCastles():MutableList<Pair<Int,Int>>{
        return playersCastles
    }
```

```
}
```

ForceAlgorithm

```
import kotlin.math.abs
import kotlin.math.sqrt
import kotlin.random.Random

class ForceAlgorithm {
    companion object{
    fun modifyGraph(matr:MutableList<MutableList<Int>>,
points:MutableList<Pair<Int,Int>>, rows:Int,
                    cols:Int, iterations:Int,
dif:Int):MutableList<Pair<Int,Int>>{
        val n = points.size
        val l = sqrt(rows*cols.toDouble() / n)
        var changes:MutableList<Pair<Double,Double>> = ArrayList()
        var tempPoints:MutableList<Pair<Double, Double>> = ArrayList()
        var temperature = l * 2
        for(i in 0 until n){
            tempPoints.add(Pair(points[i].first.toDouble(),
points[i].second.toDouble()))
        }
        for(count in 1..iterations) {
            val center = getCenter(tempPoints, n)
            for (i in 0 until n) {
                // var firstDif = 0.0
                // var secondDif = 0.0
                // for (j in 0 until n) {
                //     if (i != j) {
                //         val changeNow = getSprAndRep(tempPoints[i],
tempPoints[j], n, area)
                //         firstDif += changeNow.first.first
                //         secondDif += changeNow.first.second
                //         // if(count == iterations - 1)
                //         //     println("$i $j     Rep:
${changeNow.first.first}      ${changeNow.first.second}")
                //         if (matr[i].contains(j)) {
                //             firstDif += changeNow.second.first
                //             secondDif += changeNow.second.second
                //             //println("Spr: ${changeNow.second.first}
${changeNow.second.second}")
                //         }
                //         // val newFirstI =
modifyNumInBounds(tempPoints[i].first + firstDif, 0, rows)
                //         // val newSecI =
modifyNumInBounds(tempPoints[i].second + secondDif, 0, cols)
                //         // val newFirstJ =
modifyNumInBounds(tempPoints[j].first - firstDif, 0, rows)
                //         // val newSecJ =
modifyNumInBounds(tempPoints[j].second - secondDif, 0, cols)
                //         // tempPoints[i] = Pair(newFirstI,newSecI)
                //         // tempPoints[j] = Pair(newFirstJ,newSecJ)
                //     }
                // }
                // val newFirstI = modifyNumInBounds(tempPoints[i].first +
firstDif, 0, rows)
                // val newSecI = modifyNumInBounds(tempPoints[i].second +
```

```
secondDif, 0, cols)
                //val newFirstJ = modifyNumInBounds(tempPoints[j].first +
firstDif, 0, rows)
                //val newSecJ = modifyNumInBounds(tempPoints[j].second +
secondDif, 0, cols)
                val changeI = calculateForce(tempPoints, i, center,
temperature, matr, l, n, rows, cols, dif)
                changes.add(changeI)
                //tempPoints[i] = Pair(newFirstI,newSecI)
            }
            tempPoints = changes
            changes = ArrayList()
            temperature /= 1.04

            // readLine()
            // val field = Field(rows,cols)
            // val answ:MutableList<Pair<Int,Int>> = ArrayList()
            // for(i in 0 until n){
            //     answ.add(Pair(tempPoints[i].first.toInt(),
tempPoints[i].second.toInt()))
            // }
            // field.placeGraph(matr,answ)
            // field.show()
        }
        val ans:MutableList<Pair<Int,Int>> = ArrayList()
        for(i in 0 until n){
            ans.add(Pair(tempPoints[i].first.toInt(),
tempPoints[i].second.toInt()))
        }
        return ans
    }

    fun calculateForce(tempPoints:MutableList<Pair<Double,Double>>, i:Int,
center:Pair<Double,Double>, temp:Double,
                       matr:MutableList<MutableList<Int>>, l:Double, n:Int,
rows:Int, cols:Int, dif:Int):Pair<Double,Double>{
        var firstDif = 0.0
        var secondDif = 0.0
        val gravityK = 0.5 + matr[i].size / 2.0
        //val gravityK = matr[i].size.toDouble()
        for (j in 0 until n) {
            if (i != j) {
                var pairDist = Pair(tempPoints[j].first -
tempPoints[i].first,
                    tempPoints[j].second - tempPoints[i].second)
                if(abs(pairDist.first - 0.0)  <= Double.MIN_VALUE &&
abs(pairDist.second - 0.0) <= Double.MIN_VALUE){
                    pairDist = Pair(Random.nextDouble(-0.1,0.1),
Random.nextDouble(-0.1,0.1))
                }
                val distanceSQ = pairDist.first*pairDist.first +
pairDist.second*pairDist.second
                val repulsion = getRep(pairDist, l, distanceSQ)
                firstDif += repulsion.first
                secondDif += repulsion.second
                if (matr[i].contains(j)) {
                    val attraction:Pair<Double,Double>
                    if(matr[i].size == 1){
                        // attraction = Pair(attraction.first * 2 ,
attraction.second * 2)
                        attraction = getAttr(pairDist, l / 20, distanceSQ,
gravityK)
```

```
                    } else{
                        attraction = getAttr(pairDist, l, distanceSQ,
gravityK)
                    }
                    firstDif += attraction.first
                    secondDif += attraction.second
                }
            }
        }
        val gravity = getGravity(i, matr, tempPoints, center, 1.3)
        firstDif += gravity.first
        secondDif += gravity.second
        val koef = 0.5
        firstDif *= koef
        secondDif *= koef
        val force = changeForceForTemparature(Pair(firstDif,secondDif), temp)
        val newFirstI = modifyNumInBounds(tempPoints[i].first + force.first,
dif, rows - dif)
        val newSecI = modifyNumInBounds(tempPoints[i].second + force.second,
dif, cols - dif)
        return Pair(newFirstI, newSecI)
    }

    fun changeForceForTemparature(force:Pair<Double,Double>,
t:Double):Pair<Double,Double>{
        val dist = sqrt(force.first*force.first + force.second*force.second)
        if(dist <=t) return force
        return Pair(force.first / dist * t, force.second / dist * t)
    }

    fun modifyNumInBounds(n:Double, lowerB:Int, upperB:Int):Double{
        if(n < lowerB)
            return lowerB.toDouble()
        if(n > upperB - 1)
            return (upperB - 1).toDouble()
        return n
    }

    fun getRep(pairDist:Pair<Double,Double>, l:Double,
distanceSq:Double):Pair<Double, Double>{
        val kOfRep = (-l * l / distanceSq)
        //val kOfRep = (-l * l / distanceSq / sqrt(distanceSq))
        return Pair((kOfRep * pairDist.first), (kOfRep * pairDist.second))
    }

    fun getAttr(pairDist:Pair<Double,Double>, l:Double, distanceSq:Double,
gravityK:Double):Pair<Double,Double>{
        //val kOfAttr = (sqrt(distanceSq) / l )
        val kOfAttr = (distanceSq / (l * gravityK) / sqrt(distanceSq))
        return Pair((kOfAttr * pairDist.first), (kOfAttr * pairDist.second))
    }

    // correct
    fun getCenter(tempPoints:MutableList<Pair<Double,Double>>,
n:Int):Pair<Double,Double>{
        var sumCoordinates = Pair(0.0,0.0)
        for (j in 0 until n) {
            sumCoordinates = Pair(
                sumCoordinates.first + tempPoints[j].first,
                sumCoordinates.second + tempPoints[j].second
            )
        }
```

```
        return Pair(sumCoordinates.first / n, sumCoordinates.second / n)
    }

    fun getGravity(i:Int, matr:MutableList<MutableList<Int>>, tempPoints:
MutableList<Pair<Double, Double>>,
                   center:Pair<Double,Double>,
koef:Double):Pair<Double,Double>{
        val degK = 2 + matr[i].size / 2.0
        //val degK = matr[i].size.toDouble()
        val outOfCenter = Pair(center.first - tempPoints[i].first,
center.second - tempPoints[i].second)
        return Pair(degK * koef * outOfCenter.first, degK * koef *
outOfCenter.second)
    }
    }

}
```

Main

```
import generator.FieldGenerator
import kotlin.math.pow
import kotlin.random.Random

fun main(args:Array<String>) {
    var nOfPlayers = -1
    var mapSize = -1
    print("Do you want to set options manually? 1 - YES, 0 - NO: ")
    val inp = readLine()?.toIntOrNull() ?: 0
    if (inp == 1) {
        print("Enter the number of players from 1 to 6: ")
        nOfPlayers = readLine()?.toIntOrNull() ?: -1
        if (nOfPlayers < 1 || nOfPlayers > 6) nOfPlayers = -1
        print("Enter the map size, 0 - is 32, 1 - 64, 2 - 128, 3 - 256: ")
        var poww = readLine()?.toIntOrNull() ?: -1
        if (poww < 0 || poww > 3) poww = -1
        mapSize = if (poww != -1) 1 + 32 * 2.0.pow(poww.toDouble()).toInt()
else -1
    }
    if (nOfPlayers == -1 || mapSize == -1) {
        if (inp == 1)
            println("Wrong number of players or map size so the map was
created manually.")
        nOfPlayers = Random.nextInt(2, 6)
        mapSize = 129
    }

    val field = FieldGenerator(mapSize,nOfPlayers).generateField()
    //field.show()
    BytesBuffer(field).writeToBytes()
    println("The result of this program is the file \"map.hmm\" in the
project directory.")
}
```