Lab 1 Report | xuej41 | 400515671

A. GameMechs::setInitBoard(): Input size n – assume boardX = boardY = n when both dimensions become large

Find Θ

```
void GameMechs::setInitBoard()  // private helper function
{
    for(int i = 0; i < boardSizeY; i++)   Θ(n)
    {
        for(int j = 0; j < boardSizeX; j++)   Θ(n)
        {
            if(i == 0 || i == (boardSizeY - 1) || j == 0 || j == (boardSizeX - 1))
                gameBoard[i][j] = '#';                                              Θ(1)
            else
                gameBoard[i][j] = ' ';
        }
    }
}
```

There is a nested for loop that both iterate from i to n (boardSizex/y = n). Therefore the Θ for each for loop is Θ(n).

The if statement within the nested for loops only executes once per cycle, so Θ is Θ(1).

Therefore Θ of the entire GameMechs::setInitBoard() function is **$\Theta(n^2)$**.

B. objPosArrayList::insertHead(): Input size n – list size (a.k.a. size of the Snake)

Find Θbest , Θworst , O, and o

```
void objPosArrayList::insertHead(const objPos &thisPos)
{

    if(listSize == arrayCapacity) return;   Θ(1)

    // The rest will be our estimation of Theta worst case

    for(int i = listSize; i > 0; i--)   Θ(n)
    {
        list[i] = list[i-1];  // can do this because of copy constructor
    }

    list[0].setX(thisPos.getX());
    list[0].setY(thisPos.getY());
    list[0].setNum(thisPos.getNum());
    list[0].setPF(thisPos.getPF());
    list[0].setSym(thisPos.getSym());   Θ(1)

    //list[0].printObjPos();

    listSize++;
}
```

Θbest = **Θ(1)** because the best case scenario is when listSize == arraySize, because then the function returns immediately without executing anything else.

Θworst = **Θ(n)** because in the absolute worst case, listSize < arrayCapacity, which means there is space to insert a head. This causes the for loop to iterate through each element in the list (shifting everything forward). Everything else has a constant time complexity. Therefore the final time complexity is just Θ(n).

O = upper bound on time complexity = Θworst = **O(n)**.

o = time complexity that grows strictly slower than the given function = **o(n^2)**. We cannot claim that the complexity grows slower than n, because the function always takes at least O(n) time to perform the operation in the worst case. Technically o can be anything greater than n (n logn, n^2, n^3 etc).

C. objPosArrayList::removeTail(): Input size n – list size (a.k.a. size of the Snake)

Find Θ

```
objPos objPosArrayList::removeTail()
{
    if(listSize <= 0)
        return objPos(-99,0,0,0,0); // objPos constructor is called

    listSize--;  // lazy delete

    return list[listSize];  // return the lazy-deleted element
    // returning objPos instance through the stack will invoke copy constructor of objPos
}
```

Θ(1)

Θ(1)

Θ(1) shallow copy

Everything in the removeTail() function has a time complexity of Θ(1). The last line of the function invokes the copy constructor of objPos, which uses shallow copy:

```
objPos::objPos(const objPos& thisPos) // copy constructor
{
    x = thisPos.x;
    y = thisPos.y;
    number = thisPos.number;
    prefix = thisPos.prefix;
    symbol = thisPos.symbol;

    reward = thisPos.reward;
}
```

Everything in the copy constructor has a time complexity of Θ(1) as well. Therefore Θ = **Θ(1)**.

D. objPosArrayList::insert(): Input size n – list size (a.k.a. size of the Snake)

Find Θbest , Θworst , O, and o

```cpp
void objPosArrayList::insert(const objPos &thisPos, int index)
{
    if(listSize == arrayCapacity) return;

    if(index > listSize) index = listSize;
    if(index < 0) index = 0;

    for(int i = listSize; i > index; i--)
    {
        list[i] = list[i-1];
    }

    list[index].setX(thisPos.getX());
    list[index].setY(thisPos.getY());
    list[index].setNum(thisPos.getNum());
    list[index].setPF(thisPos.getPF());
    list[index].setSym(thisPos.getSym());

    //list[index].printObjPos();

    listSize++;
}
```

$\Theta(1)$

$O(n)$

$\Theta(1)$

Θbest = **Θ(1)** because if index == listSize, the for loop will not run, resulting in a constant time complexity.

Θworst = **Θ(n)** for the same reasons as Θworst for part B.

O = Θworst = **O(n)**.

o = **o(n^2)** because it grows strictly slower than O, which is at most O(n).

E. Player::drawPlayer() and Player::undrawPlayer(): Input size n – size of the Snake

Find Θ

```cpp
void Player::drawPlayer()
{
    objPos targetPos;
    int scanSize = myPos->getSize();  // get the list size

    myPos->resetReadPos();
    for(int i = 0; i < scanSize; i++)
    {
        targetPos = myPos->getNext();
        boardRef[targetPos.getY()][targetPos.getX()] = targetPos.getSym();
    }
}


void Player::undrawPlayer()  // private helper function
{
    objPos targetPos;
    int scanSize = myPos->getSize();  // get the list size

    myPos->resetReadPos();
    for(int i = 0; i < scanSize; i++)
    {
        targetPos = myPos->getNext();
        boardRef[targetPos.getY()][targetPos.getX()] = ' ';
    }
}
```

default constructor

$\Theta(1)$

$\Theta(n)$

same here

Both functions work similarly. The default constructor is called, which has a time complexity of Θ(1), the function getSize() is called, which simply returns a size value and whose time complexity is also Θ(1), and the function resetReadPos is called which also simply returns a value and whose time complexity is Θ(1).
Then there is a for loop which has time complexity Θ(n) as it iterates through each element in the snake's length, and performs actions of Θ(1) complexity.
Therefore both functions have a total Θ of **Θ(n)**.

F. ScreenDrawer::Draw(): Input size n – assume boardX = boardY = n when both dimensions become large. Assume MacUILib_clearScreen() and MacUILib_printf() have Θ(1).
Find Θ

```
void ScreenDrawer::Draw() const
{
    // Clear the Screen
    MacUILib_clearScreen();                          Θ(1)

    // redraw all items;
    binRef->drawItem();                              Θ(1)
    playerRef->drawPlayer();                         Θ(n)

    // Get the Game Board 2D array
    char** drawTarget = gmRef->getBoardRef();        Θ(1)
    objPos target = binRef->getItem();

    // Draw it on the screen
    for(int i = 0; i < gmRef->getBoardSizeY(); i++)
    {
        for(int j = 0; j < gmRef->getBoardSizeX(); j++)
        {
            MacUILib_printf("%c", drawTarget[i][j]);     Θ(n^2)
        }
        MacUILib_printf("\n");
    }

    // Append any required debugging message below
    MacUILib_printf("Player Score: %d\n", playerRef->getScore());    Θ(1)
    MacUILib_printf("Food Reward: %d\n", target.getReward());
    //MacUILib_printf("Object: <%d, %d>, ID=%c%d\n", target.getX(), target.getY(), target.getPF(), target.getNum());
}
```

As shown by the analysis above, Θ = **Θ(n^2)**. Every other line of code calls on a function of time complexity Θ(1), except for drawPlayer() which has a time complexity of Θ(n) because the function contains a for loop.

G. Player::checkSelfCollision(): Input size n – size of the Snake

Find Θbest , Θworst , O, and o

```cpp
bool Player::checkSelfCollision()  // private
{
    // Make sure snake is long enough to kill itself
    int length = myPos->getSize();
    if(length < 4) return false;

    // Then check for self collision
    myPos->resetReadPos();
    objPos tempPos;
    objPos headPos = myPos->getNext();

    for(int i = 1; i < length; i++)
    {
        tempPos = myPos->getNext();
        if(headPos.isOverlap(&tempPos))
        {
            // set game end.
            return true;
        }
    }

    return false;
}
```

$\Theta(1)$

$\Theta(1)$

$O(n)$

$\Theta(1)$

Θbest = **Θ(1)**. If length < 4, the function returns false immediately and therefore has a time complexity of Θ(1). If length >= 4, the for loop may detect a collision on the first cycle, which therefore also has a time complexity of Θ(1).

Θworst = **Θ(n)**. The function checks each segment of the snake and no collision is detected, so it runs through the entire for loop and returns false at the very end.

O = Θworst = **O(n)**.

o = **o(n^2)**. The time complexity of o grows strictly slower than O, which is at most O(n).

H. Player::movePlayer(): Input size n – size of the Snake. Assume MacUILib_hasChar() and MacUILib_getChar() have Θ(1). Assume checkCollision() always returns false, and assume killable is always false. Find Θ if possible. If not possible, use the most suitable Landau notation to describe its complexity. You may use the runtime estimations from previous determined function calls.

```
67  void Player::movePlayer()
68  {
69      updatePlayerFSM();
70      if(myDir == STOP) return;              Θ(1)
71
72      undrawPlayer();                        Θ(n)
73
74      objPos currHeadPos = myPos->getHead();
75      int inX = currHeadPos.getX();          Θ(1)
76      int inY = currHeadPos.getY();
77
78      switch(myDir)
79      {
80          case UP:
81              if(--inY < 1)
82                  inY = gmRef->getBoardSizeY() - 2;
83              break;
84
85          case DOWN:
86              if(++inY > (gmRef->getBoardSizeY() - 2))
87                  inY = 1;
88              break;
89
90          case LEFT:                                  Θ(1)
91              if(--inX < 1)
92                  inX = gmRef->getBoardSizeX() - 2;
93              break;
94
95          case RIGHT:
96              if(++inX > (gmRef->getBoardSizeX() - 2))
97                  inX = 1;
98              break;
99
100         default:
101             break;
102     }
103
104     currHeadPos.setX(inX);
105     currHeadPos.setY(inY);// TARGET                 Θ(1)
106
107     myPos->insertHead(currHeadPos);  // insert new head   O(n)      O(n)
108
109     if(!checkCollision())            // check collision.  If collision never happened,
110         myPos->removeTail();   Θ(1)  // removeTail.  Otherwise, generate new item.     Ω(1)
111
112     if(killable)
113         if(checkSelfCollision())
114             gmRef->setGameLost();      // If colliding with itself, end game.
```

No, it is not possible to find Θ because on line 70, if myDir == STOP, the function returns immediately. Therefore the best case is **Ω(1)**.

If myDir != STOP, the time complexity is **O(n)** due to the functions noted in the image.

I. ItemBin::generateItem(): Input size n – size of the Snake

Find Θ if possible. If not possible, use the most suitable Landau notation to describe its complexity.

```cpp
53  void ItemBin::generateItem()
54  {
55      // Step 1: Get Player Ref from GameMech for Player pos
56      Player** plList = gmRef->getPlayerListRef();
57      objPosList *playerPos = plList[0]->getPlayerPos();
58
59      //int bitVec[gmRef->getBoardSizeX()][gmRef->getBoardSizeY
60
61      // to prevent stack overflow
62      int xsize = gmRef->getBoardSizeX();
63      int ysize = gmRef->getBoardSizeY();
64      int** bitVec = new int*[xsize];
65      for(int i = 0; i < xsize; i++)
66      {
67          bitVec[i] = new int[ysize];
68          for(int j = 0; j < ysize; j++)
69              bitVec[i][j] = 0;
70      }
71
72
73
74      int playerLength = playerPos->getSize();
75
76      objPos target;
77      playerPos->resetReadPos();
78
79      for(int i = 0; i < playerLength; i++)
80      {
81          target = playerPos->getNext();
82          bitVec[target.getX()][target.getY()] = 1;
83      }
84
85      int randCandidateX = 0;
86      int randCandidateY = 0;
87      int randCandidate = 0;
88
89      // Step 2: Generate another food object
90      // Coordinate Generation
91      // x [2, BoardX-3]
92      // y [2, BoardY-3]
93
94      do
95      {
96          randCandidateX = rand() % (gmRef->getBoardSizeX() - 4
97          randCandidateY = rand() % (gmRef->getBoardSizeY() - 4
98      } while(bitVec[randCandidateX][randCandidateY] != 0);
99
100
101     undrawItem();
102
103     myItem->setX(randCandidateX);
104     myItem->setY(randCandidateY);
105
106     // Prefix Generation
107     // PF   [a-z, A-Z]
108     randCandidate = rand() % 26 + 'A';  // 26 alpabets
109     if(rand() % 2) randCandidate += 32; // randomly lowercase
110     myItem->setPF((char)randCandidate);
111
112     // Number Generation
113     // Number [00, 99]
114     myItem->setNum(rand() % 100);
115
116     drawItem();
117
118     for(int i = 0; i < xsize; i++)
119         delete[] bitVec[i];
120     delete[] bitVec;
121 }
```


No, it is not possible to find Θ. Due to the do while loop on line 94, the worst case time complexity is technically **O(∞)** (or undefined), where it would continuously generate random positions and check the condition without ever finding a valid spot.

The best case Ω is **Ω(n^2)** due to the nested for loop on line 65, given the do while loop runs less than n^2 times.

All the other code here is irrelevant for time complexity but their analysis is in the image above.