MOKSH SHUKLA
180433

Theoretical Assignment 2: PART 2

ESO207A: Data Structures & Algorithms

2021-03-22

Problem 4

The approach to this problem is modified version of count Inversion problem discussed in the lectures, where we recursively keep dividing our problem into halves and then count inversions in both the divided halves plus all the cross inversions that exist across the divided halves. The only modification from count Inversions problem that this solution requires is that we need to keep count of smaller elements to the right of an element for each index of array. So, for this we create an array *count*, in which each index stores the count of smaller elements to the right.

Pseudo Code

```
#define ll long long int

struct data {
    ll key; ll idx;
};

void _mergeInversions(data *arr, ll *count , ll left, ll right) {

    if (left > right)
        return;

    ll mid = (left + right) / 2;
    data temp[right - left + 1];
    ll i = left; ll j = mid + 1; ll k = 0;

    while ((i <= mid) && (j <= right)) {
        if (arr[i].key > arr[j].key) {
            temp[k] = arr[i];
            count[arr[i].idx] += right - j + 1;
            k++;i++;
        }
        else {
            temp[k] = arr[j];
            k++;j++;
        }
    }

    while (i <= mid) {
        temp[k] = arr[i];
        k++;i++;
    }

    while (j <= right) {
        temp[k] = arr[j];
        k++;j++;
    }

    for (ll i = left; i <= right; i++)
        arr[i] = temp[i - left];

    return;
}

void countInversions(data *arr, ll *count, ll left, ll right) {

    if (left >= right)
        return;
    ll mid = (left + right) / 2;

    countInversions(arr, count , left, mid); // T(n/2)
    countInversions(arr, count, mid + 1, right); // T(n/2)
    _mergeInversions(arr, count, left, right); // cn

}
int main()
{
    data arr[n];
    for (ll i = 0; i < n; i++) {
        cin >> arr[i].key;
        arr[i].idx = i;
    }

    ll count[n] = {0};
    countInversions(arr, count, 0, n - 1);
}
```

Time Complexity

The algorithm makes call to recursive function *countInversions* twice and the loop functions in *_mergeInversions* have complexity cn. So, for the algorithm:

$$\begin{aligned}
T(n) &= 2T(n/2) + cn \\
T(n) &= cn + c*2*(n/2) + c*2*(2*(n/4)) + ... + c*2^{\log_2 n}*1 \\
T(n) &= c\,\underbrace{(n+n+n....+n)}_{\text{log n terms}} \\
T(n) &= cn\log n \\
T(n) &= \mathcal{O}(n\log n)
\end{aligned}$$

Problem 5

Let V be the number of vertices and E be the number of edges.

- **Case 1**: We know that if there are V nodes then the number of edges possible such that there does exist any cycle is $V-1$ (as discussed in lectures). Hence, if $E >= V$, then we can definitely say that there exists a cycle.
- **Case 2**: When $E < V$, we use recursive DFS to find out the cycle in the graph. In this case, for every vertex v we go to its neighbour using DFS and subsequently keep on storing the vertices which are visited. While we follow this procedure, if we again reach to a vertex which has already been visited (checked through Boolean condition), then straightaway we know that there exists a cycle.

The data structure used in this algorithm to check for cycles is ***Adjacency List***.

Pseudo Code

```
bool DFS_cycle(G, v, parent, *visit) {
    visit[v] = true; //initialize for first node
    for each neighbour w of v{
    if(w==parent)
        continue;
    else if(visit[w])
        return true;
    else if(DFS_cycle(G,w,v,visit))
        return true;
    }
    return false; // no cycle found
}

// V -> no. of vertices
// E -> no. of edges
bool solve(G,V,E){
    if(E>=V)
        return true;
    visit array initialized false for vertices of G
    for each node w in adj_L{
        if(visit[w])
            continue;
        else if(DFS_cycle(G,w,-1,visit))
            return true;
    }
}
```

Time Complexity

- **Case 1** ($E >= V$): $\mathcal{O}(1)$, since we just need to check for a single condition which operates in constant time.
- **Case 2** ($E < V$): Since we use DFS, each vertex is visited at most once and as the neighbours of these are searched time taken is twice the number of edges connected to it. Therefore, this case runs in $T(n) = 2E + V$. Since, $E < V$, we have $2E < 2V$ and adding V to both sides we have $2E + V < 3V$. Thus, $T(n) = \mathcal{O}(V)$.

Overall Time Complexity is $T(n) = \mathcal{O}(1) + \mathcal{O}(V) = \mathcal{O}(V)$

Problem 6

The knight can move to 8 possible cells(its neighbours)from its current position. So, we start by initializing all grid points to -1 (to signify not visited cells) and starting point to 0. Then the initial position of the knight is pushed inside a queue. Now we start iterating to all the possible positions from the current cell by removing the from the queue last visited cell position. If the new cell position is inside the chess board and is not visited (checked by condition that value of the new cell position is not -1), then the new cell position is pushed inside the queue and distance value is updated by 1 at that position. Finally when endpoint is reached, the final updated value is returned.

Pseudo Code

```
struct data
{
    int x; int y;
};

bool Inside(N, x, y) {
    if (x <= 0 or y <= 0 or x >= N or y >= N)
        return false;
    else
        return true;
}

int minMoves(N, C, D, E, F) {
    posBoard[N][N] < - initialize all cells of board with - 1
    posBoard[C][D] = 0; // initialize starting point with 0
    xMoves = { -2, -2, -1, -1, 1, 1, 2, 2 };
    yMoves = { -1, 1, -2, 2, -2, 2, -1, 1};
    data pos; pos.x = C; pos.y = D;
    q < - CreateEmptyQueue();
    Enqueue(pos , q);
    while (!IsQueueEmpty(q)) {
        knightPos < - Dequeue(q);
        for (i = 0 : 7) { // possible moves from a cell for knight
            knightNext_x = knightPos.x + xMoves[i];
            knightNext_y = knightPos.y + yMoves[i];

            if (Inside(N, knightNext_x , knightNext_y) == false) // checks if piece is inside the board
                continue;
            else if (posBoard[knightNext_x][knightNext_y] >= 0) // checks if cell has already been visited
                continue;

            posBoard[knightNext_x][knightNext_y] = posBoard[knightPos.x][knightPos.y] + 1; // updates distance when reaches not visited cell

            if (knightNext_x == E and knightNext_y == F)
                return posBoard[knightNext_x][knightNext_y];

            pos.x = knightNext_x; pos.y = knightNext_y;
            Enqueue(pos , q);
        }
    }

}
```

Time Complexity

In the worst case our algorithm visits all the cells of the grid and hence we get our time complexity $\mathcal{O}(n^2)$.

Space Complexity

In the worst case our algorithm inserts all the cells of the grid (n^2) into the queue. Hence, the space complexity $\mathcal{O}(n^2)$.