Hermmy Wang

704978214

CS 180 Homework 1

1. Bit complexity

BinaryOnetoN(n):	
X<--0	1
For i <-- 1 to n	n
Starting from right to left in X, **find** the first digit that is 0 and **assume** it is the kth digit	1
X **<-- flip** the kth digit of X to 1 and flip 1,2,...,(k-1)th digit of X to 0	2n
Print X	1

Length of the input: n

Outside the loop: 1 operation

Within the loop of size n: finding the digit takes 1 operation; flipping the digits and assigning them to X take 2n operations; printing X takes 1 operation. In total:

$T(n) = 1+n(1+2n+1) = 2n^2+2n+1$

2.

(a)

PART A: We want to prove that for any favorable table, there exists a move that makes the table unfavorable. In other words, if we represent the number of matches in each row in binary, we need to show: for a table that have some columns with odd number of ones, there is a way to manipulate only *one* row, so that the table contains all columns of even ones.

Proof by induction:

Base case: If there are 1 row of matches on a table and the first player removes all matches from that row, there is 0 match left. The table is now an unfavorable table.

Inductive case: Suppose for a table with k rows, we assume there is one move which can turn all columns to have an even number of ones.

If we add one row to the k-row table, the table now contains (k+1) rows. The $(k+1)^{th}$ row contains m_{k+1} matches, or $(b_1, b_2, \dots, b_j)_2$ in binary. Since we assume that the first k rows of the table have all columns with an even number of ones, for every $b_i = 1$, where $0 < i <= j$, the i^{th} column contains an odd number of ones.

For a column with an odd number of ones, removing or adding one 1 will give the column an even number of ones. Therefore, if all $b_i = 1$ are negated, the result number of matches, which is smaller than the original number, will contain all columns with an even number of ones.

PART B: We want to prove that for any unfavorable table, any move makes the table favorable for one's opponent.

Proof by contradiction:

Suppose there is a move that will not make the table favorable. An unfavorable contains all columns of even 1's. A move in a selected row will change the bits in that row's binary representation of matches, causing one or more bits to flip. However, as shown before, flipping a bit once will definitely change the parity of that column. Hence, the table will *not* maintain the even parity anymore, which is a contradiction.

ALGORITHM:

```
removeMatches(table):
sum <-- 0
For column c <-- 0 to m-1
        For row r <-- 0 to n-1
                sum <-- sum + table[r][c]
        Endfor
        If sum is odd
                parity[c] <-- odd
        Else
                parity[c] <-- even
        Endif
        sum <-- 0
Endfor
For row r <-- 0 to n-1
        For column c <-- 0 to m-1
                If parity[c] is odd
                        result_binary[c] <-- flip table[r][c]
                Else
                        result_binary[c] <-- table[r][c]
                Endif
                If (result_binary)10 < (table[r])10
                        remove_num <-- (table[r])10 – (result_binary)10
                        Return the (r+1)th row and remove_num
                Endif
        Endfor
        result_binary <-- 0
Endfor
Return -1 and -1 if nothing is found
```

Time complexity: $O(n^2)$

There are two 2-level loops in this algorithm, so the time complexity is $O(n^2)$. I first find the parity of each column in the table. Then starting from the first row, I find how I should manipulate each bit in the row in order to make its column even. For the odd column, I need to flip the bit. For the even column, I do not need to do anything. After finishing all the columns, I obtain a binary representation of the desired number of matches in the row. Then I compare this number with the original number of matches: if this number is larger, it is not logical to remove any matches from that row. I will continue the loop to look for a valid solution in the next row. Once I reach a valid solution, I will return which row it is and the number of matches to remove.

(b) We want to find an algorithm that shows there are more than one way to make the table unfavorable. From the algorithm in part(a), if I change the second two-level loop to:

```
...
count <-- 0
For row r <-- 0 to n-1
        For column c <-- 0 to m-1
                If parity[c] is odd
                        result_binary[c] <-- flip table[r][c]
                Else
                        result_binary[c] <-- table[r][c]
                Endif
                If (result_binary)10 < (table[r])10
                        count <-- count+1
                Endif
        Endfor
        result_binary <-- 0
Endfor
Return true if count > 1
```

Time complexity: $O(n^2)$

There are two 2-level loops in this algorithm, so the time complexity is $O(n^2)$. This algorithm works because this algorithm works in the same fashion as (a), except that I will count how many valid solutions I have in the current table. If there are more than one solution, I successfully determined whether there exist multiple ways to make the table favorable.

(c)

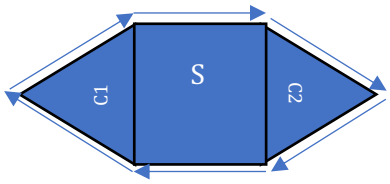
```
winGame(table):
If table = empty
        Return win
Endif
sum <-- 0
For column c <-- 0 to m-1
        For row r <-- 0 to n-1
                sum <-- sum + table[r][c]
        Endfor
        If sum is odd
                parity[c] <-- odd
        Else
                parity[c] <-- even
        Endif
        sum <-- 0
Endfor
For row r <-- 0 to n-1
        For column c <-- 0 to m-1
                If parity[c] is odd
                        result_binary[c] <-- flip table[r][c]
                Else
                        result_binary[c] <-- table[r][c]
                Endif
                If (result_binary)10 < (table[r])10
                        remove_num <-- (table[r] - result)10
                        Return the (r+1)th row and remove_num
                Endif
        Endfor
        result_binary <-- 0
Endfor
Player remove matches
Get next_table from opponent
Return winGame(next_table)
```

Time complexity: $O(n^4)$

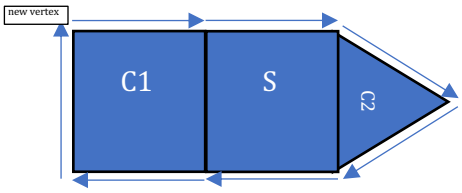
There are two 2-level loops in this function, so the time complexity in each call of the function is $O(n^2)$. This algorithm works in the same fashion as (a), except that it only stops when the game ends. The function waits the player to finish his move after a solution is found. This move will make the current table unfavorable; and then any move from the opponent will make the table favorable for the player. I will pass the current table again to this function. Since there are n*m matches to remove, the function will at most be called ½*n*m times. The final time complexity if $O(n^{2\wedge 2})$, which is $O(n^4)$.

3.

(a) Base case: In the simplest case where we have C1, C2, and S, we have 6 vertices. The single cycle that visits every vertex in G is:



Inductive case: Suppose in a graph G = {V, E}, there exists a single cycle that visits every vertex in G exactly once. Then for G' = {V', E'}, we add one vertex and one edge such that C1, C2, and S still hold. One of the cycles, e.g. C1, now has one more vertex and one more edge. In this case, the new single cycle will follow the edges in C1, including the new vertex and edge. Therefore, there always exists a single cycle that visits all vertices once.

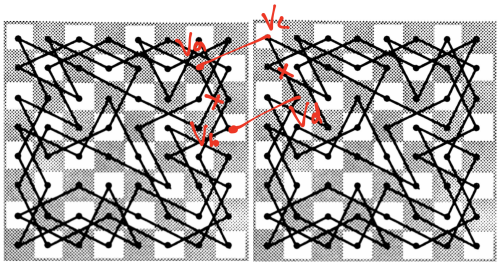


(b)

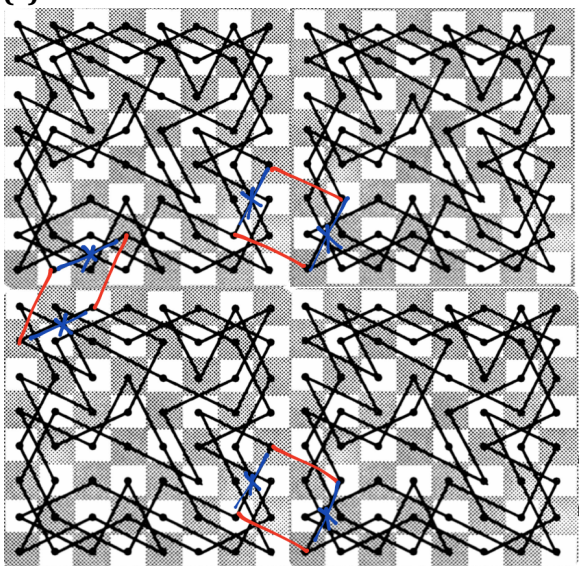
Step1: If we take two 8x8 chessboards with a closed knight's tour, we can find 2 edges on each board, (v_a, v_b), (v_c, v_d) such that (v_a, v_c) and (v_b, v_d) are each a diagonal of a 2x3 grid. Remove (v_a, v_b) and (v_c, v_d), we now have a closed knight's tour on an 8x16 chessboard.

Step2: Repeating the same process for another two 8x8 chessboards will obtain the second 8x16 chessboard.

Step3: Taking these two 8x16 chessboards and repeating the same process again will result in a 16x16 chessboard with a closed knight's tour.



(c)



Edge selection is *not* unique. This figure showing the edges I choose to add (red) and to remove (blue) on a 16x16 board. I express the graph as a set of edges. Each edge is represented as a pair of edges. The vertices are labelled as a point on the coordinate, where the bottom left corner is (0,0) and the upper right corner is $(2^k-1, 2^k-1)$

genClosed(k):
If k = 3
    Return the 8x8 board with closed knight's tour
Endif
On the 2^k x 2^k board, form 6 edges:
    $(2^{k-1},0)$, $(2^{k-1}-2,1)$
    $(2^{k-1}+1,2)$, $(2^{k-1}-1,3)$
    $(2^{k-1}, 2^{k-1})$, $(2^{k-1}-2, 2^{k-1}+1)$
    $(2^{k-1}+1, 2^{k-1}+2)$, $(2^{k-1}-1, 2^{k-1}+3)$
    $(0, 2^{k-1}-2)$, $(1, 2^{k-1})$
    $(2, 2^{k-1}-1)$, $(3, 2^{k-1}+1)$
upleft <-- genClosed(k-1)
upright <-- genClosed(k-1) + x-axis shifts right 2^{k-1}
downleft <-- genClosed(k-1) + y-axis shifts down 2^{k-1}
downright <-- genClosed(k-1) + y-axis shifts down 2^{k-1} and x-axis shifts right 2^{k-1}
current <-- current + upleft + upright + downleft + downright
On the current board, remove edges:
    $(2^{k-1},0)$, $(2^{k-1}+1,2)$
    $(2^{k-1}-2,1)$, $(2^{k-1}-1,3)$
    $(2^{k-1}, 2^{k-1})$, $(2^{k-1}+1, 2^{k-1}+2)$
    $(2^{k-1}-2, 2^{k-1}+1)$, $(2^{k-1}-1, 2^{k-1}+3)$
    $(0, 2^{k-1}-2)$, $(2, 2^{k-1}-1)$
    $(1, 2^{k-1})$, $(3, 2^{k-1}+1)$
Return current

Time complexity: $O(4^{2^{\wedge}n})$

In each call of the function, the time complexity is 2^n because it needs to shift the coordinates of the vertices for every subgraph. Since the function is called 4 times within the function, the total time complexity is $O(4^{2\wedge n})$. The way the algorithm works by dividing the chessboard into 4 subgraphs of $2^{k-1}x2^{k-1}$ chessboards recursively. Two 8x8 chessboard can form a closed loop of knight's tour by connecting two diagonals of a 2*3 grid across the two boards and removing the adjacent edges whose vertices belong to the same 8x8 board (see the figure above). If we keep adding and removing edges repeatedly for every single cycle we found, we will eventually get a big single cycle, which is the close loop of knight's tour.

4.

(a)

```
recurBtoN(B[1...n]):
    If the whole array is finished:
        Return 0
    Endif
    If the last element of the current array is 0:
        Return 2*recurBtoN(B[1...n-1])
    Endif
    Return 2*recurBtoN(B[1...n-1])+1
```

Time complexity: O(n)

The function calls itself once to process every element in the array, so the time complexity is O(n). As the algorithm reads from the last element to the first element of the array, the value will multiply by 2 every time it detects one more bit ahead and increment by the current bit value. This is because when a new bit is detected, all the old bits shift up by 2 and the newest bit is placed on the current 2^0 position. By the time the array is finished, the correct decimal value of the binary representation will be returned.

(b)

```
itBtoN(B[1...n]):
        sum <-- 0
        For i <-- 1 to n
                If the ith element is 0:
                        sum <-- sum*2
                Else
                        sum <-- sum*2+1
                Endif
        Endfor
        Return sum
```

Time complexity: O(n)

The function is dominated by a loop of size n; hence the time complexity is O(n). Similar to part(a), the loop reads the binary array from the first element. Every time the loop checks the next bit, the current sum will shift up by 2 and increment the next bit's value. By the time the array is all checked, the correct decimal value of the binary representation will be returned.