Jason Less
404-640-158
CS180

CS180 Homework 1

Question 1

Assuming there are k total bits to represent a number, a single execution of the loop (i.e. to increment or calculate the next number) would take O(k) in the worst case (i.e. flipping all k bits). As there are n numbers to calculate, the algorithm consists of n increment operations, so in the worst case, the bit complexity is O(nk), where n is the number of increment operations, and k is the number of bits flipped.

However, k bits aren't flipped for every iteration of the loop; and in fact, the 0^{th} bit is flipped every 1 increment, the 1^{st} bit is flipped every 2 increments, the 2^{nd} bit is flipped every 4 increments, and so on, whereas the kth bit is flipped every 2^k increments. To represent n numbers, we need 2^b bits (i.e. if b = 1, we can represent n = 2 numbers, if b = 5, we can represent n = 32 numbers). Solving for b:

$$2^b = n => b\ log\ 2 = \log n => b = \log n\ \ bits$$

Therefore, we only need log n bits to construct the binary representation from 1 to n. So, for k = 0, 1, …, log N, bit k flips $\frac{n}{2^k}$ times for n increments. The total number of bit flips (or bit complexity) for n operations can be represented as a summation series:

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^k} = n\ (1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k})$$

This equation represents the sum of infinite geometric series, which can be represented by the equation: $s = \frac{a}{1-r}$, where a is the first value in the series, and r is the changing ratio. As we are constructing the binary representation starting at 1, a = 1, and the ratio r = ½. Therefore, substituting in values for a and r, we get:

$$n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k}\right) = n\left(\frac{1}{1 - \frac{1}{2}}\right) = 2n$$

Finally, after removing constants, we see that the bit complexity of the BinaryOneToN algorithm is O(n).

Jason Less
404-640-158
CS180

Question 2

(a)

By definition, a favorable table is a table where there exists a column with an odd number of ones in it. It can be shown that given a favorable table, there exists a move to make the table unfavorable by demonstrating such a move. This can be achieved by locating the most significant column that consists of an odd number of ones in it. From that column, change any one of the ones to a zero. Additionally, from whichever row we selected, also change any other columns (to the right of the column selected) to make all columns have an even number of ones (i.e. creating an unfavorable table).

Using the strategy mentioned above, a favorable table can be made into an unfavorable table. Notice that two selections were made when locating the row to remove matches from that created an unfavorable table:

1. The column selected is the most significant column with odd parity
    a. By selecting the most significant column, we know that we are always decreasing the number of matches in a given row (which is a requirement of the game)
2. We selected a column with odd parity
    a. We don't select a column with even parity because altering bits in that column would create odd parity for that column (and we want to create even parity for a column to make an unfavorable table)

Next we want to prove that for any unfavorable table, a move exists to make the table favorable. By definition, an unfavorable table is a table where all columns have an even number of ones. As we can only remove matches from one row, and we must remove at least one match from that row, any number of matches taken will change the bit representation in the row. By doing this, it will make it so that at least one column will have an odd number of ones (creating a favorable table) for mentions already illustrated above.

Algorithm:

- Find most significant column with odd number of ones in it
- Change one of the rows with a one in that column to a zero
- For the least significant columns in the row we selected, change the bits to create columns that have an even number of ones

Jason Less
404-640-158
CS180

Question 2 (cont.)

(b)

Given an input of favorable table, you can determine whether there exists multiple ways to make the table unfavorable if there exists a column with an odd number of ones in which all the elements of the column are ones.

Ex: if we have three rows that consist of 4, 5, and 6 matches respectively, then the most significant column contains an odd number of ones, where all the elements are ones. From this representation, we perform the algorithm mentioned in (a), we can create three unfavorable tables: a table with 3, 5, and 6 matches, a table with 4, 2, and 6 matches, and a table with 4, 5, and 1 match.

(c)

Algorithm

- On your turn, always make the given favorable table into an unfavorable table
    - This will result in a player always winning the game because the winner of the game selects the last match
    - If this player selects the last match, then the other player will see a table where no matches are left, which is an unfavorable table (thus losing the game)

Jason Less
404-640-158
CS180

Question 3

(a)

There exists a single cycle in G that visits every vertex in G exactly once (a Hamiltonian cycle) due to the presents of cycles C_1 and C_2, and the four-edged circle that exists in G. First of all the two cycles C_1 and C_2 each pass through half of the vertices in G and are disjoint sets that don't share any vertices in common. If we assume for a second that there exist two edges a and b that can be used to connect cycles C_1 and C_2 on each side, then we know that we have a complete graph where we can pass through each vertex exactly once (i.e. following path of C_1, taking edge a from C_1 to C_2, following the path of C_2, and then taking edge b from C_2 to C_1). How do we know two such edges a and b exist in graph G? We know this due to the fact that there exists a four edge circle in G s.t. one edge belongs to C_1, one edge belongs to C_2, and two edges belong to neither C_1 nor C_2. Thus, the last two edges of the circle allow us to make the complete graph as described above leading to a single cycle in G that visits every vertex exactly once.

(b)

A closed knight's tour visits every square exactly once and returns to the original square. This is a common problem where the concept of backtracking can be applied (and though not the most efficient, it can be used to complete a closed knight's tour of the chessboard). Incrementally try to add new moves to the existing tree, and if we visit all squares and arrive back at the start, we are done. If we encounter a square we have seen before or run out of valid moves, then backtrack and try a different path.

Algorithm

- If we have visited all of the squares and return to the start square, then we are done
- Else
    - Add one of eight possible moves to the our data structure keeping track of the board and recursively try to complete the closed knight's tour
    - If the move chosen doesn't lead to a solution, then try the other eight possible moves to try to reach a solution
    - If all of the eight moves don't lead to a solution from the given square, then we know that we can't complete the knight's tour from this square having executed the moves we have chosen. Thus, we remove the previous move and must backtrack to try to complete the closed knight's tour from the previous square (and so on)

(c)

The backtracking algorithm can be used to solve the closed knight's tour using any sized board (whether 16x16 or $2^k x 2^k$), where performance gets slower as the size of the board increases.

Jason Less
404-640-158
CS180

Question 4

(a)

An array B that stores the binary representation of a positive number can be evaluated to get the actual value of the number by utilizing the fact that a binary number can be represented in decimal by formula below, where b is the position, and q is the base value:

$$n = b_n q^n + b_{n-1} q^{n-1} + \cdots + b_1 q^1 + b_0 q^0$$

So for example, if B = [1, 1, 0], then this can be evaluated as:

$$1(2^2) + 1(2^1) + 0(2^0) = 4 + 2 + 0 = 6$$

I will call the recursive function evalBinary, and will utilize a position variable pos to keep track of where I am in array B, and will use the power function to calculate the result. If the array B consists of n bits, then the algorithm will make n recursive calls to evaluate each position of array B, thus the time complexity is O(N) and the space complexity is O(N) if we count n recursive calls to the stack.

Algorithm

- If pos is equal to the size of the array, then return 0
- Else
  - return $2^{pos} * B[size - 1 - pos] + evalBinary(pos + 1)$

(b)

The above recursive algorithm can be simplified into an iterative solution by using a simple loop. I will utilize a rs variable to keep track of the value of array B. The time complexity in this case would also be O(N) and the space complexity would be O(1).

Algorithm

- rs = 0, j = 0
- starting at the back of the loop, for each position i in array B for i >= 0
  - rs += $2^j * arr[i]$
  - increment j, decrement i