CS180 Solution 1

1. In the class we showed how to construct the binary representations for numbers $1, 2, 3, ..., n$ by doing binary addition:

```
BINARYONETON(n)
  X ← 0
  for i ← 1 to n
      starting from right to left in X, find the first digit that is 0 and assume it is the k^th digit
      X ← flip the k^th digit of X to 1 and flip 1, 2, ..., (k−1)^th digit of X to 0
      print X
```

The bit complexity of an algorithm counts the number of bit operations in terms of the length of the input. What is the bit complexity of the algorithm BINARYONETON?

Solution:

We count the number of bit operations, i.e., the number of flip bit operations used when $n = 8$. For all iterations with initial value of X that ends with 0, it costs one bit operation to add 1 on X, and moreover, there are $n/2 = 4$ such iterations. For all iterations with initial value of X that ends with 01, it costs 2 operations to add 1 and there are $n/4 = 2$ iterations, For all iterations with initial value of X that ends with 011, it costs 3 operations and there are $n/(2^3) = 1$ such iteration.

iteration	Initial value of X	# of bit operations to add 1
1	0000	1
2	0001	2
3	0010	1
4	0011	3
5	0100	1
6	0101	2
7	0110	1
8	0111	4

Table 1: number of bit operations for each iteration when n= 8

In general, for any iteration with initial value X that ends with $0\underbrace{11.11}_{i-1 \text{ ones}}$, it costs i operations to add 1 on X, and there are $n/(2^i)$ such iterations. It is not hard to see that such a ending pattern partitions all the numbers 1..n into $\log n$ parts by the fact that the length of n is $\log n$. Therefore, the total number of bit operations in terms of the value of the input n is:

$$T(n) = 1 \cdot \frac{n}{2^1} + 2 \cdot \frac{n}{2^2} + 3 \cdot \frac{n}{2^3} + .. + \log n \cdot \frac{n}{2^{\log n}} = \sum_{i=1}^{\log n} i \cdot \frac{n}{2^i} = O(n)$$

The last equality can be shown by deriving $T(n) = 2T(n/2) + \log n$ and then solve the recurrence by master theorem. To get the bit complexity in terms of the length of the input, we can just rewrite the the formula above by the length of the input $|n| = \log n$.

2. Suppose you are playing the game of NIM. This game begins with a placement of n rows of matches on a table. Each row i has m_i matches. Players take turns selecting a row of matches and removing any or all of the matches in that row. Whoever claims the final match from the table wins the game. This game has a winning strategy based on writing the count for each row in binary and lining up the binary numbers (by place value) in columns. We note that a table is *favorable* if there is a column with an odd number of ones in it, and the table is *unfavorable* if all columns have an even number of ones.

   Example: Suppose we start off with three rows of matches of 7, 8 and 9. The binary representations of number of matches are $7 = 0111, 8 = 0111, 9 = 1001$. Therefore, the board will look like this

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

   Since the third row from the right and the second row from the right, each has odd numbers of ones, the table is favorable.

   (a) Prove that, for any favorable table, there exists a move that makes the table unfavorable. Prove also that, for any unfavorable table, any move makes the table favorable for one's opponent. Write the algorithm that on an input of favorable table outputs the row and the number of matches to remove from that row, in order to make the table unfavorable.

   Solution:

   The first thing to prove is that, for any favorable table, there exists a move that makes the table unfavorable for one's opponent. The easiest way to prove that something exists is by construction: that is, show how to take a favorable table and select a move to produce an unfavorable table.

```
FAVORMOVE(matrx reprentation of the table M)
  for j ← 1 to n (column indices of M)
      if column j has an odd count of one
          Select an arbitrary row i with one on columnj, and flip M[i, j] it to 0
          for j ← j + 1 to n
              if column j has an odd count of one, flip M[i, j] to its opposite
          return the operation that remove matches that corresponds the new row i on M
```

   If the table is favorable, this means that there is at least one column that has an odd sum. Inspect the most significant column with an odd count. Select an arbitrary row with a one in that column; this is the pile of matches from which we will remove some. Change the 1 in that column to a 0 (note that this change alone is sufficient to show that the resulting number of matches is smaller than the original count, and thus a valid move, no matter how many other columns we change). Change any other columns, if necessary, to produce even parity. This will produce an unfavorable table for the opponent.

   We also need to prove that, given an unfavorable table, any move produces a favorable table for one's opponent. By definition of unfavorable table, there is even parity for each column. However, we may only select one row, and must remove at least one match from that row. As a result, at least one bit will change, and will change in only one row. Any bit change will modify the parity of the column, and as such, this will produce odd parity in at least one column, and thus a favorable table for the opponent.

(b) Given an input of favorable table, can you determine whether there exist multiple ways to make the table unfavorable? How?

Solution:

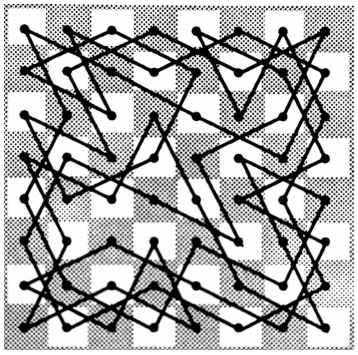
If the most significant column with odd ones has three or more "1"s , then you can chose any row with a "1" and make the table unfavorable.

(c) Design an algorithm that always wins the game if given a favorable table.

Solution:

The loser of the game will see no match on the table, which is a unfavorable table. So, a wining strategy is always leave a unfavorable table to the other by applying the algorithm in part a.

3. Recall that a knight can make one of eight legal moves. A closed knight's tour on an 8×8 chessboard, that is, a circular tour of knight's moves that visits every square of the chessboard exactly once before returning to the first square. Such a tour is also known as a Eulerian tour.



A closed knight tour for 8×8 chessboard

(a) Given a graph $G = \{V, E\}$. G contains two cycles C_1 and C_2 where each cycle visits half of the vertices exactly once and C_1 and C_2 do not share any vertex. Suppose there exists a circle S in G made of four edges: $S = \{(v_a, v_b), (v_b, v_c), (v_c, v_d), (v_d, v_a)\}$, where (v_a, v_b) is an edge in cycle C_1 and (v_c, v_d) is an edge in cycle C_2, while the two other edges are neither in C_1 nor in C_2. Show that then there exists a single cycle in G that visits every vertex in G exactly once.

Solution:

We can concatenate C_1 and C_2 by removing (v_a, v_b) is an edge in cycle C_1 and (v_c, v_d) is an edge in cycle C_2 and then add $(v_b, v_c), (v_d, v_a)$ in the remaining edges of C_1 and C_2. Such modification gives a valid construction of a single cycle in G that visits every vertex exactly once.

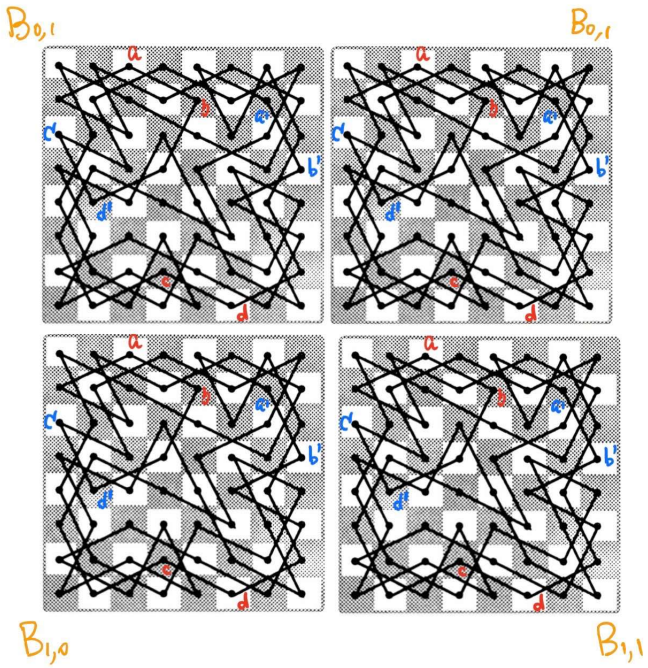
(b) Give an algorithm generates a closed knight's tour on a 16×16 chessboard. (Hint: you may use a closed knight's tour for 8×8 as a starting point)

Solution:

We provide a procedure to construct a 16×16 closed knight tour from the 8×8 closed e knight tour we proved. The idea is arrange four 8×8 board in square and then try to apply the concatenation procedure in part a to connect them as a big cycle. Given four 8×8 board, we arrange them in a square and denote them as $B_{0,0}, B_{0,1}, B_{1,0}, B_{1,1}$ and assume corresponding closed knight tours are $C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}$.

```
CONNECTFOUR(B_{0,0}, B_{0,1}, B_{1,0}, B_{1,1}, C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1})
  for every board pair (B_x, B_y) ∈ {(B_{0,0}, B_{1,0}), (B_{1,0}, B_{1,1}), (B_{1,1}, B_{0,1})}
      let C_x, C_y be the corresponding tour for B_x and B_y
      for all edges in C_{0,0} and C_{1,0}
          find a pair of edges (v_a, v_b) from C_x and (v_c, v_d) such that
              there exists a move from v_a to v_d and a move from v_c to v_b.
          remove (v_a, v_b) and (v_c, v_d) in C_x and C_y
          merge C_x and C_y to be a closed knight tour C_xy by connecting (v_a, v_c) and (v_b, v_d)
```

The algorithm works because there exists edges (v_a, v_b), (v_c, v_d) that satisfies the condition in problem (a) for any pair of $\{(B_{0,0}, B_{1,0}), (B_{1,0}, B_{1,1}), (B_{1,1}, B_{0,1})\}$, which is shown in the following figure.



possible connection points for four 8×8 chessboards

(c) Give an algorithm generates a closed knight's tour on any $2^k \times 2^k$ chessboard for all $k \geq 3$.
In the previous construction of 16×16 closed knight tour, notice the boundary of the merged cycle is not modified since we just need to connect 3 pairs. Hence, we can reuse the previous algorithm to merge any four $2^{k-1} \times 2^{k-1}$ chessboard, and it is guaranteed to find proper edges to do the merge job.

4. Given an array $B[1..n]$ that stores a binary representation of a positive integer, where each $B[i]$ is either 0 or 1. For example, $B = [1, 1, 0]$, which represents 6 in binary.

(a) Design a recursive algorithm that evaluates the actual value of the array B represents. Your algorithm is NOT allowed to use any pre-calculated table for powers of 2. For example, your algorithm cannot use $2^3 = 8$ or $2^4 = 16$ as a known fact without actually calculating it.
Solution:

For notation purpose, we reverse the array B and relabel it to $B[0..n]$, where $B[0]$ stores the least significant bit and $B[n]$ most significant bit. So number 6 corresponding the array $B = [0,1,1]$. By the definition of the binary number, to evaluate the array $B[1..n]$ is equivalent to evaluate to polynomial $p(x) = B[0]\cdot x^0 + B[1]\cdot x^1 + ... + B[n-1]*x^{n-1} + B[n]\cdot x^n$ on the point $x = 2.$, i.e., $\text{EVAL}(B) = B[0]\cdot 2^0 + B[1]\cdot 2^1 + ... + B[n-1]*2^{n-1} + B[n]\cdot 2^n$. A trivial iterative algorithm to evaluate a polynomial B with coefficients $B[0..n]$ at point x according to the definition would be

```
POLYEVAL(B[0..n], x)
  s ← 1
  y ← 0
  for i ← 0 to n
        y ← y + B[i]·s
        s ← s·x
  return y
```

The numbers of arithmetic operations it uses are n addition and $2n$ multiplication. However, we can reduce the number of multiplication by half using Horner's

$$p(x) = B[0] + x(B[1] + x(B[2] + ... + xB[n])$$

and it leads to a natural recursive algorithm:

```
HORNEREVAL(B[0..n], x)
  if n = 0
        return B[0]
  y ← B[0]
  y ← y + x · HORNEREVAL(B[1..n], x)
  return y
```

Since each recursive call uses one addition and multiplication, the algorithm uses n additions and n multiplications in total, and that is a faster than the trivial one.

(b) Unfold the recursive algorithm (in which a procedure call on itself with a different parameter) into an *iterative* algorithm that does not call any procedure.

Solution:

Again, uses Horner's rule:

```
ITEREVAL(B[0..n], x)
  y ← 0
  for i ← n to 0
        y ← y · x + B[i]
  return y
```

To evaluate the binary number B, just set $x = 2$. It uses the same number of addition and multiplication as the recursive version.