Suggested Solutions to Midterm Problems

1. Consider the following theorem regarding Gray codes, for which we have sketched a proof by induction in class.

   There exists a Gray code of length $\lceil \log_2 k \rceil$ for any number $k \geq 2$ of objects. The Gray codes for the *even* values of k are *closed*, and the Gray codes for *odd* values of k are *open*.

   Please complete the proof (giving sufficient details); in particular, try to be precise about the length of a code in the proof. You should assume and use the following facts in your proof:

   (a) Given a closed Gray code for an even number k (≥ 2) of objects, we can construct a closed Gray code with one additional bit for $2k$ objects.

   (b) Given a closed Gray code of length i for 2^i ($i \geq 1$) objects, we can construct an open Gray code of the same length for any odd k, $2^{i-1} < k < 2^i$, of objects.

   (c) Given an open Gray code for an odd number k (≥ 2) of objects, we can construct a closed Gray code with one additional bit for $2k$ objects.

   Solution. We first state and prove two lemmas that will help better structure the main proof.

   - (Lemma 1) There exists a closed Gray code of length i ($= \lceil \log_2 k \rceil$) for any number $k = 2^i$ ($i \geq 1$) of objects.
     The proof is by induction on i.
     Base case ($i = 1$, i.e., $k = 2$): $\{0, 1\}$ constitute a closed Gray code of length 1 for 2 objects.
     Inductive step: Consider $k = 2^i = 2 \times 2^{i-1}$ ($i > 1$) objects. From the Induction Hypothesis and Fact 1a, we can construct a closed Gray code with $(i-1)+1 = i$ bits for the k objects
   - (Lemma 2) There exists an open Gray code of length $\lceil \log_2 k \rceil$ for any odd number $k > 2$ of objects.
     Every odd number $k > 2$ is properly bounded by 2^{i-1} and 2^i for some $i > 1$. From Lemma 1 and Fact 1b, we can construct for the k objects an open Gray code of length i which equals $\lceil \log_2 k \rceil$.

   The main proof is by induction on the number k of objects.

   Base case ($k = 2^i$ for $i \geq 1$): this follows from Lemma 1.

   Inductive step: The case when k is odd follows from Lemma 2. Now, let $k = 2j$. There are two cases:

   - j is even. From the Induction Hypothesis and Fact 1, we can construct a closed Gray code with $\lceil \log_2 j \rceil + 1$ for $2j$ objects. $\lceil \log_2 j \rceil + 1 = \lceil 1 + \log_2 j \rceil = \lceil \log_2 2 + \log_2 j \rceil = \lceil \log_2 2j \rceil = \lceil \log_2 k \rceil$, so the length is as required.

- j is odd. From the Induction Hypothesis and Fact 3, we can construct a closed Gray code with $\lceil \log_2 j \rceil + 1$ for $2j$ objects. Analogous to the previous case, the length is as required.

□

2. Consider the following two-player game: given a positive integer N, player A and player B take turns counting to N. In his turn, a player may advance the count by 1 or 2. For example, player A may start by saying "1, 2", player B follows by saying "3", player A follows by saying "4", etc. The player who eventually has to say the number N loses the game.

   A game is *determined* if one of the two players always has a way to win the game. Prove that the counting game as described is determined for any positive integer N; the winner may differ for different given integers. You must use induction in your proof. (Hint: think about the remainder of the number N divided by 3.)

   Solution. We first prove the following claim:

   > When $N = 3k + 1$ for some $k \geq 0$, player B can always win the game.

   The proof is by induction on k.

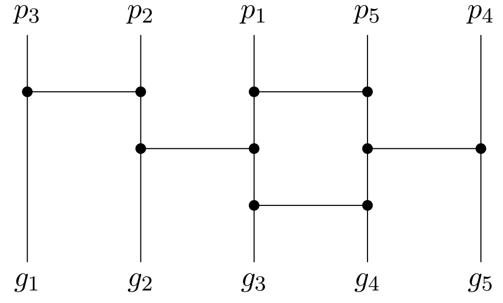
   Base case ($k = 0$, i.e., $N = 1$): player A has no other choice but say 1 and hence player B wins.

   Inductive step ($k \geq 1$, i.e., $N = 3k + 1 \geq 4$): player A starts either by "1" or "1, 2". In both cases, player B can always count to 3. At this point we have the situation analogous to that the two players are to play a game with $N = 3(k - 1) + 1$, in which player B can always win from the induction hypothesis.

   We next show that, when $N = 3k + 2$ or $N = 3(k + 1)$ for some $k \geq 0$, player A can always win the game. In the case when $N = 3k + 2$, player A starts by saying "1", while in the case when $N = 3(k + 1)$, he starts by "1, 2". After player A's first turn, we have the situation analogous to that player B is to start a game with $N = 3k + 1$, playing the role of player A (to start first in the remaining game). From the preceding claim, player A (playing the role of player B in the remaining game) will win the game.

   Now we see that, for every positive integer N, there is always a player that can win the counting game and hence the game is determined. □

3. We sometimes would use a diagram like the following to distribute n gifts (or assign n tasks) to n people. The main part of the diagram covered, each person (without seeing the horizontal line segments) is asked to choose one of the vertical lines. After everyone has made a choice, the whole diagram is revealed. Following the line chosen by p_i, go down along the line and, whenever hitting an intersection, must make a turn (to the left or right). The traced path will eventually reach a gift at the end and the gift is given to p_i.



Prove by induction that such a diagram (with arbitrary numbers of vertical and horizontal line segments) always produces a one-to-one mapping between the people and the gifts (whose number equals that of the vertical lines).

Solution. The proof is by induction on the number m of horizontal line segments. (Note: it should have been stated that no two horizontal line segments share an intersection.)

Base case ($m = 0$): Since there is no horizontal line segment, p_1 is mapped to g_1, p_2 to g_2, ..., and p_n to g_n, which is a one-to-one mapping between the people and the gifts.

Inductive step: Given an arbitrary setting of m (≥ 1) horizontal line segments, we remove the line segment that is highest in position; if there are several such line segments, remove any one of them. From the induction hypothesis, the new setting of $m-1$ horizontal line segments defines a one-to-one mapping between the people and the gifts. Let us refer to the mapping as f, which maps p_i to $g_{f(i)}$ Suppose the removed line segment originally connected vertical lines i and $i+1$. We claim that, with the removed line segment restored, the original setting also defines a one-to-one mapping; call it f'. Clearly, $f'(i) = f(i+1)$, $f'(i+1) = f(i)$, and $f'(j) = f(j)$ for any other j. It follows that, given f is one-to-one, f' is also one-to-one. □

4. For each of the following pairs of functions, determine whether $f(n) = O(g(n))$ and/or $f(n) = \Omega(g(n))$. Justify your answers.

	$f(n)$	$g(n)$
(a)	$(\log n)^{\log n}$	$\frac{n}{\log n}$
(b)	$n^3 2^n$	3^n

Solution.

(a) We assume the base of logarithm is 2. $f(n) = (\log n)^{\log n} = (2^{\log \log n})^{\log n} = 2^{\log \log n \times \log n}$; $g(n) = \frac{n}{\log n} = 2^{\log \frac{n}{\log n}} = 2^{\log n - \log \log n}$.

Therefore, $f(n) \geq g(n)$ and hence $f(n) = \Omega(g(n))$.

(b) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^3 2^n}{3^n} = \lim_{n\to\infty} n^3 (\frac{2}{3})^n = 0$. So, $f(n) = o(g(n))$. It follows that $f(n) = O(g(n))$ (and $g(n) = \Omega(f(n))$), but $f(n) \neq \Omega(g(n))$.

Alternatively, $\forall n \geq 7$, $f(n) = n^3 2^n \leq 7^3 \times 3^n = 343 \times g(n)$, which shows that $f(n) = O(g(n))$. The number 7 was decided by considering the largest number n such that $(\frac{n+1}{n})^3 \geq \frac{3}{2}$, after which point the increase via n^3 can never catch up with the increase via $(\frac{3}{2})^n$. □

5. Solve the following recurrence relation using *generating functions.* This is a very simple recurrence relation, but you must use generating functions in your solution.

$$\begin{cases} T(1) = 1 \\ T(2) = 2 \\ T(n) = 2T(n-1) - T(n-2), \quad n \geq 3 \end{cases}$$

Solution. Let $T(z) = 0 + T_1 z + T_2 z^2 + T_3 z^3 + \cdots + T_n z^n + \cdots$ (a generating function for the sequence $T(n)$, $n \geq 0$, assuming $T(0) = 0$).

$$\begin{array}{rcl} T(z) & = & T_1 z + T_2 z^2 + T_3 z^3 + \cdots + T_n z^n + T_{n+1} z^{n+1} + \cdots \\ 2zT(z) & = & 2T_1 z^2 + 2T_2 z^3 + \cdots + 2T_{n-1} z^n + 2T_n z^{n+1} + \cdots \\ z^2 T(z) & = & T_1 z^3 + T_2 z^4 + \cdots + T_{n-2} z^n + T_{n-1} z^{n+1} + \cdots \\ \hline (1 - 2z + z^2)T(z) & = & z \end{array}$$

$T(z) = \frac{z}{1-2z+z^2} = \frac{z}{(1-z)^2} = z + 2z^2 + 3z^3 + \cdots + nz^n + \cdots$ and, therefore, $T(n) = n$ for $n \geq 1$.

□

6. If $f(x)$ is monotonically *decreasing*, then

$$\sum_{i=1}^{n} f(i) \leq f(1) + \int_1^n f(x)dx.$$

Show that this is indeed the case. (5 points)

Solution. This is easily seen by comparing the areas (on the $R \times R$ plane) defined by the formulae on the two sides. □

7. Consider the Knapsack Problem: Given a set S of n items, where the i-th item has an integer size $S[i]$, and an integer K, find a subset of the items whose sizes sum to exactly K or determine that no such subset exists.

We have discussed in class two approaches to implementing a solution that we designed by induction: one uses dynamic programming (see the Appendix), while the other uses recursive function calls.

Suppose there are 5 items, with sizes $2, 3, 4, 5, 6$, and we are looking for a subset whose sizes sum to 13. Assuming recursive function calls are used, please give the two-dimension table P whose entries are filled with -, O, I, or left blank when the algorithm terminates. Which entries of $P[n, K]$ are visited/computed more than once?

Solution.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		-	-	-	-	-	-	-	-	-	-	-		-
$k_1 = 2$	O	-		-	-	-	-	-	-	-	-			-
$k_2 = 3$				I	-			-	-	-				-
$k_3 = 4$								I	-					-
$k_4 = 5$								O						-
$k_5 = 6$														I

$P[0, 1]$, $P[0, 3..8]$, and $P[1, 4]$ are visited more than once. □

8. Consider a variant of the Knapsack Problem where we want the subset to be as large as possible (i.e., to be with as many items as possible). How will you adapt the algorithm (see the Appendix) that we have studied in class? Your algorithm should collect at the end the items in one of the best solutions if they exist. Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary (you may reuse the

code for the original Knapsack Problem). Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will get for this problem.

Solution. (Yi-Wen Chang)

To find the largest possible subset of items, we modify the *Knapsack* algorithm in the Appendix to obtain *Knapsack_ForMaxSubset*, as shown below. Each element $P[i,k]$ in the result array P now contains a new integer variable *size* which memorizes the size of the current largest subset for k.

```
Algorithm Knapsack_ForMaxSubset($S, K$);
begin
    $P[0,0].exist := true$;
    $P[0,0].size := 0$;
    for $k := 1$ to $K$ do
        $P[0,k].exist := false$;
        $P[0,k].size := 0$;
    for $i := 1$ to $n$ do
        for $k := 0$ to $K$ do
            $P[i,k].exist := false$;
            $P[i,k].size := 0$;
            if $k - S[i] \geq 0$ and $P[i-1, k-S[i]].exist$ then
                if $P[i-1,k].exist$ and $P[i-1,k].size \geq P[i-1,k-S[i]].size + 1$ then
                    $P[i,k].exist := true$;
                    $P[i,k].belong := false$;
                    $P[i,k].size := P[i-1,k].size$;
                else
                    $P[i,k].exist := true$;
                    $P[i,k].belong := true$
                    $P[i,k].size := P[i-1,k-S[i]].size + 1$;
            else if $P[i-1,k].exist$ then
                $P[i,k].exist := true$;
                $P[i,k].belong := false$;
                $P[i,k].size := P[i-1,k].size$;
    if $\neg P[n,K].exist$ then
      print "no solution"
    else $i := n$;
         $k := K$;
         while $k > 0$ do
            if $P[i,k].belong$ then
                print $i$;
                $k := k - S[i]$;
            $i := i - 1$;
end
```

The complexity remains the same, which is $O(nK)$. When a solution (a subset of items whose sizes sum to exactly K) exists, the printed result will be the largest among such subsets. □

9. Let $x_1, x_2, \ldots, x_n$ be a set of integers, and let $S = \sum_{i=1}^{n} x_i$. Design an algorithm to partition the set into two subsets of equal sum, or determine that it is impossible to do

so. When the partitioning is possible, your algorithm should also give the two subsets of integers. The algorithm should run in time $O(nS)$.

Solution. (Jen-Feng Shih)

```
Algorithm Partition_into_Two_Subsets(x);
begin
    sum := $\sum_{i=1}^{n} x_i$;
    if sum is odd then print "no solution."
    else
        K := sum/2;
        Knapsack(x, K);
        if P[n, K].exist := false then
            print "no solution."
        else
            l := 1;
            m := 1;
            for i := n to 1 do
                if P[i, k].belong := true then
                    set1[l] := x[i];
                    l := l + 1;
                    k := k − x[i];
                else
                    set2[m] := x[i];
                    m := m + 1;
            print "set1:"
            for i := 1 to l − 1 do
                print set1[i];
            print "set2:"
            for i := 1 to m − 1 do
                print set2[i];
end
```

The complexity remains the same as in the Knapsack Problem, which is $O(nK) = O(nS)$.

□

10. In the **towers of Hanoi** puzzle, there are three pegs A, B, and C, with n (generalizing the original eight) disks of different sizes stacked in decreasing order on peg A. The objective is to transfer all the disks on peg A to peg B, moving one disk at a time (from one peg to one of the other two) and never having a larger disk stacked upon a smaller one.

    (a) Give an algorithm to solve the puzzle. Compute the total number of moves in the algorithm. (10 points)

    Solution.

```
Algorithm Towers_Hanoi(A,B,C,n);
begin
   if n=1 then
      pop x from A and push x to B
   else
```

```
        Towers_Hanoi(A,C,B,n-1);
        pop x from A and push x to B;
        Towers_Hanoi(C,B,A,n-1);
end;
```

Let $T(n)$ denote the number of moves taken by the algorithm to move the n disks from peg A to peg B. The recurrence relation for $T(n)$ is as follows:

$$\begin{aligned}&T(1) = 1\\&T(n) = 2T(n-1)+1, \text{for } n \geq 2\end{aligned}$$

$$\begin{array}{rl}
T(n) = & 2T(n-1)+1 \\
2T(n-1) = & 2(2T(n-2)+1) = 2^2T(n-2)+2 \\
2^2T(n-2) = & 2^2(2T(n-3)+1) = 2^3T(n-3)+2^2 \\
\cdots & \cdots \\
2^{n-2}T(2) = & 2^{n-1}T(1)+2^{n-2} \\
2^{n-1}T(1) = & 2^{n-1} \\
\hline
T(n) = & 1+2+2^2+\cdots+2^{n-1} \\
= & 2^n-1
\end{array}$$

□

(b) If there is an additional fourth peg D, it is possible to reduce the number of moves. Please give a new algorithm that requires fewer moves. (5 points)

Solution.

```
Algorithm Four_Towers_Hanoi(A,B,C,D,n);
begin
   if n<=2 then
      Towers_Hanoi(A,B,C,n);
   else
      Four_Towers_Hanoi(A,D,B,C,n-2);
      Towers_Hanoi(A,B,C,2);
      Four_Towers_Hanoi(D,B,C,A,n-2);
end;
```

`Towers_Hanoi(A,B,C,1)` takes 1 move, while `Towers_Hanoi(A,B,C,2)` takes 3 moves. A recurrence relation for $T(n)$ is the following:

$$\begin{aligned}&T(1) = 1\\&T(2) = 3\\&T(n) = 2T(n-2)+3, \text{for } n \geq 3\end{aligned}$$

We solve the recurrence relation by considering odd and even n's separately.
When n (≥ 3) is odd,

$$\begin{array}{rl}
T(n) = & 2T(n-2)+3 \\
2T(n-2) = & 2(2T(n-4)+3) = 2^2T(n-4)+2\times 3 \\
2^2T(n-4) = & 2^2(2T(n-6)+3) = 2^3T(n-6)+2^2\times 3 \\
\cdots & \cdots \\
2^{\frac{n-3}{2}}T(3) = & 2^{\frac{n-3}{2}}(2T(1)+3) = 2^{\frac{n-1}{2}}+2^{\frac{n-3}{2}}\times 3 \\
\hline
T(n) = & 2^{\frac{n-1}{2}}+3\times(2^{\frac{n-1}{2}}-1) \\
= & 2^{\frac{n+3}{2}}-3
\end{array}$$

When n (≥ 3) is even,

$$\begin{array}{rl}
T(n) = & 2T(n-2)+3 \\
2T(n-2) = & 2(2T(n-4)+3) = 2^2T(n-4)+2\times 3 \\
2^2T(n-4) = & 2^2(2T(n-6)+3) = 2^3T(n-6)+2^2\times 3 \\
\cdots & \cdots \\
2^{\frac{n-4}{2}}T(4) = & 2^{\frac{n-4}{2}}(2T(2)+3) = 3\times 2^{\frac{n-2}{2}} + 2^{\frac{n-4}{2}}\times 3 \\
\hline
T(n) = & 3\times 2^{\frac{n-2}{2}} + 3\times(2^{\frac{n-2}{2}}-1) \\
= & 3\times 2^{\frac{n}{2}} - 3
\end{array}$$

Apparently in both cases, the number of moves is less than that in the algorithm for the original puzzle.

$\square$