Suggested Solutions to Midterm Problems

(Compiled on May 4, 2000)

1. Find a gray code of length $\lceil \log_2 14 \rceil$ $(= 4)$ for 14 objects. Show how the gray code is constructed systematically from gray codes of smaller lengths.

   Solution. Let $(c_1, c_2, \ldots, c_n)^R$ denote the list $c_n, c_{n-1}, \ldots, c_1$.

   Code of length 1 for 2 objects: $0, 1$.
   Code of length 2 for 2 objects: $00, 01$.
   Code of length 2 for 3 objects: $00, 01, 11$ (which is open).
   Code #1 of length 3 for 3 objects: $000, 001, 011$.
   Code #2 of length 3 for 3 objects: $100, 101, 111$.
   Code of length 3 for 6 objects: $000, 001, 011, (100, 101, 111)^R$.
   Code of length 3 for 7 objects: $000, 001, 011, 111, 101, 100, 110$ (which is open).
   Code #1 of length 4 for 7 objects: $0000, 0001, 0011, 0111, 0101, 0100, 0110$.
   Code #2 of length 4 for 7 objects: $1000, 1001, 1011, 1111, 1101, 1100, 1110$.
   Code of length 4 for 14 objects:
   $0000, 0001, 0011, 0111, 0101, 0100, 0110, (1000, 1001, 1011, 1111, 1101, 1100, 1110)^R$
   $= 0000, 0001, 0011, 0111, 0101, 0100, 0110, 1110, 1100, 1101, 1111, 1011, 1001, 1000.$ □

2. Below is a theorem from Manber's book:

   > For all constants $c > 0$ and $a > 1$, and for all monotonically increasing functions $f(n)$, we have $(f(n))^c = O(a^{f(n)})$.

   Prove, by using the above theorem, that $n^2(\log n)^2 = O(n^{2.25})$. (5 points)

   Solution. If we are able to show that $(\log n)^2 = O(n^{.25})$, then $n^2(\log n)^2 = O(n^2 \cdot n^{.25}) = O(n^{2.25})$.

   Applying the theorem with $f(n) = \log n$, $c = 2$, and $a = 2^{.25}$, we have $(\log n)^2 = O((2^{.25})^{\log n}) = O(2^{.25 \log n}) = O(2^{\log n^{.25}}) = O(n^{.25})$.
   (Note: As usual, we have assumed the base of logarithm is 2. The same result can still be obtained even if a different base is used.) □

3. Show all intermediate and the final AVL trees formed by inserting the numbers 0, 1, 2, 3, 4, 9, 8, 7, 6, and 5 (in this order). If a rotation is performed during an insertion, please also show the tree before the rotation.

   Solution. See the attached. □

4. Write a program (in pseudo code) to merge two skylines.
   An example skyline: (1,**9.2**,5.5,**11**,9,**0**,12.8,**6.6**,18,**15**,22.9).

   Solution. This is part of Homework Assignment #3 (Programming Exercise #1). □

5. The Knapsack Problem is defined as follows: Given a set S of n items, where the ith 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.

   Below is an algorithm for determining whether a solution to the problem exists.

```
Algorithm Knapsack (S, K);
begin
    P[0,0].exist := true;
    for k := 1 to K do
        P[0,k].exist := false;
    for i := 1 to n do
        for k := 0 to K do
            P[i,k].exist := false;
            if P[i-1,k].exist then
               P[i,k].exist := true;
               P[i,k].belong := false
            else if k - S[i] ≥ 0 then
                    if P[i-1,k-S[i]].exist then
                       P[i,k].exist := true;
                       P[i,k].belong := true
end
```

   (a) Modify the algorithm to solve a variation of the knapsack problem where each item has an unlimited supply.

   Solution. It suffices to modify the last five lines before "end" as follows:

```
                P[i,k].belong := 0;
            else P[i,k].belong := 0;
                j := 1;
                while k - S[i] × j ≥ 0 do
                        if P[i-1,k-S[i] × j].exist then
                           P[i,k].exist := true;
                           P[i,k].belong := j;
                           break;
                        j := j + 1;
```

□

(b) Design an algorithm to recover the solution recorded in the array P of the preceding algorithm.

Solution.

Procedure Print_Solution (S, P, n, K);
begin
  if $\neg P[n, K].exist$ **then**
    print "no solution"
  else $i := n$;
    $k := K$;
    while $k > 0$ **do**
      if $P[i, k].belong > 0$ **then**
        print $i, P[i, k].belong$;
        $k := k - S[i] \times P[i, k].belong$;
      $i := i - 1$
end

□

6. Below is the algorithm that we studied in class for determining, given a sorted array A of distinct integers, whether there exists an index i such that $A[i] = i$. The idea of the algorithm is good, but its pseudo code has a bug. Please identify the error and give an example input for which the code produces an incorrect output or fails to terminate.

Algorithm Special_Binary_Search (A, n);
begin
  $Position := Special\_Find(1, n)$;
end

function Special_Find $(Left, Right) : integer$;
begin
  if $Left = Right$ **then**
    if $A[Left] = Left$ **then** $Special\_Find := Left$
    else $Special\_Find := 0$
  else
    $Middle := \lceil \frac{Left+Right}{2} \rceil$;
    if $A[Middle] < Middle$ **then**
      $Special\_Find := Special\_Find(Middle + 1, Right)$

else
$Special\_Find := Special\_Find(Left, Middle)$
end

(5 points)

Solution. The last five lines before "end" are problematic. Suppose we feed the algorithm with a two-element array $A = (-1, 1)$. In the invokation of $Special\_Find(1, 2)$, $Middle$ will become $\lceil \frac{1+2}{2} \rceil = 2$. Since $A[2] = 1 < 2 = Middle$, an erroneous invokation $Special\_Find(3, 2)$ occurs. The final result is unpredictable. Depending on the values of the memory cells following $A[2]$, the program may fail to terminate or return some strange value.

A remedy is to change the assignment "$Middle := \lceil \frac{Left+Right}{2} \rceil$" to "$Middle := \lfloor \frac{Left+Right}{2} \rfloor$". (Another possibility is to change the recursive calls "$Special\_Find(Middle + 1, Right)$" and "$Special\_Find(Left, Middle)$" respectively to "$Special\_Find(Middle, Right)$" and "$Special\_Find(Left, Middle - 1)$".) □

7. Given as input a sorted array A of n numbers and another number x, design an algorithm with running time $O(n)$ to determine whether there are two elements of A whose sum is exactly x. (Hint: Recall the ideas of the $O(n)$ soluton to the Celebrity Problem discussed in class.)

Solution. The basic idea is that, if $A[1] + A[n] < x$, then $A[1]$ cannot be part of the solution and, if $A[1] + A[n] > x$, then $A[n]$ cannot be part of the solution. In either case, we eliminated one element from the array.

Algorithm Find_Two (A, n, x);
begin
  $i := 1$;
  $j := n$;
  while $i < j$ **do**
    if $A[i] + A[j] = x$ **then**
      break;
    if $A[i] + A[j] < x$ **then**
      $i := i + 1$;
    else $j := j - 1$;
  if $i < j$ **then**
    print i, j;
  else print "no solution"
end

The while loop will be executed at most $n - 1$ times, hence the running time of the algorithm is $O(n)$. □

8. Rearrange the following array into a heap using the bottom-up approach.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	1	5	11	10	14	3	9	8	2	13	4	15	12	6

Show the result after each element is added to the part of array that already satisfies the heap property.

Solution.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	1	5	11	10	14	3	9	8	2	13	4	15	12	6
7	1	5	11	10	14	**12**	9	8	2	13	4	15	**3**	6
7	1	5	11	10	**15**	12	9	8	2	13	4	**14**	3	6
7	1	5	11	**13**	15	12	9	8	2	**10**	4	14	3	6
7	1	5	11	13	15	12	9	8	2	10	4	14	3	6
7	1	**15**	11	13	**14**	12	9	8	2	10	4	**5**	3	6
7	**13**	15	11	**10**	14	12	9	8	2	**1**	4	5	3	6
15	13	**14**	11	10	**7**	12	9	8	2	1	4	5	3	6

□

9. A variation of the n-coins problem is defined as follows.

> You are given a set of n coins $\{c_1, c_2, \ldots, c_n\}$, among which at least $n-1$ are identical "true" coins and at most one coin is "false"; a false coin is lighter than the other true coins. Also, you are given a balance scale, which you may use to compare the total weight of any m coins with that of any other m coins. The problem is to find the "false" coin, or show that there is no such coin, by making some sequence of comparisons using the balance scale.

Show that in the worst case it is impossible to solve the n-coins problem with k comparisons (for any n and k) if $n > (3^k - 1)$.

Solution. All examinees will receive the full credit for this problem.

The inequality "$n > (3^k - 1)$" was erroneously stated as "$n > (2^k - 1)$" in the original problem statement. This corrected version is left as an exercise. □

10. Design an algorithm as efficient as possible to determine whether two sets of numbers (represented as arrays) are disjoint. State the time complexity of your algorithm in terms of the sizes m and n of the given sets.

Solution.

Algorithm Disjoint?(A, m, B, n);
begin
$Heapsort(A, m)$;

```
    Heapsort(B, n);
    i := 1;
    j := 1;
    while i ≤ m and j ≤ n do
        if A[i] = B[j] then
            break;
        if A[i] < B[j] then
            i := i + 1;
        else j := j + 1;
    if i > m or j > n then
        print "yes";
    else print "no"
end
```

The invokations of Heapsort take $O(m \log m + n \log n) = O(\max(m, n) \log(\max(m, n)))$ time and the while loop takes $O(m + n) = O(\max(m, n))$ time. So, the time complexity of the algorithm is $O(\max(m, n) \log(\max(m, n)))$. □

11. Draw a Huffman tree for a text that contains eight characters A, B, C, D, E, F, G, and H with frequencies 6, 2, 3, 5, 14, 8, 4, and 2, respectively.

Solution. See the attached. □