Sample Solutions to Homework #2

1. (15)

   (a) See Figure 1.

   Figure 1: Illustration for Problem 1(a).

   (b) See Figure 2.

   (c) See Figure 3.

2. (20)

   (a) See Figure 4.

   (b) Since $Y[1,1]$ and $Y[m,n]$ are the smallest and the largest elements in Y respectively, the two statements are true.

   (c) We first record the smallest value and then replace it with the infinity. The infinity must be moved to proper position for maintaining property the of Young tableau. This goal is achieved by recursion. The recursion compares the current position with right and down positions. We exchange the smaller one with current position, treat the position with the smaller value as the new current position, and then do next recursion. It stops when right and down values are both infinity. After the recursion, the recorded value is returned as the minimum value.

   The method is shown as follows:

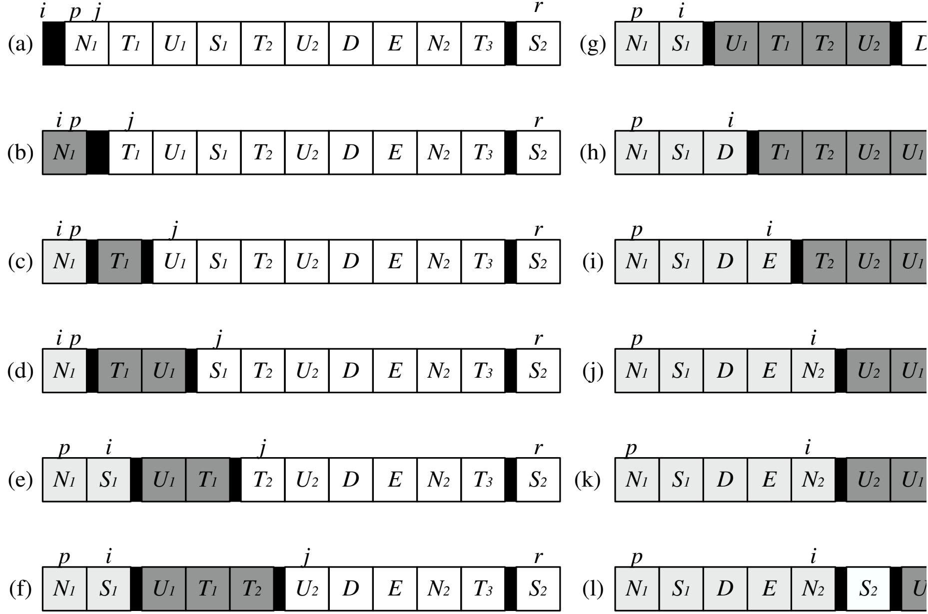

Figure 2: Illustration for Problem 1(b).

Framework:
Step 1. Record $Y[1,1]$, i.e. $v \leftarrow Y[1,1]$
Step 2. Replace $Y[1,1]$ with ∞, then call MOVE-RD$(1,1)$
Step 3. Return v

MOVE-RD(i,j):
assume $Y[i,j] = \infty$ if $i > m$ or $j > n$
if $Y[i+1,j] = \infty$ and $Y[i,j+1] = \infty$ then RETURN
if $Y[i+1,j] < Y[i,j+1]$ then
swap $Y[i,j]$ with $Y[i+1,j]$
call MOVE-RD$(i+1,j)$
else
swap $Y[i,j]$ with $Y[i,j+1]$
call MOVE-RD$(i,j+1)$

Time complexity:
$T(p) = T(p-1) + \Theta(1) = O(p)$
Hence,$T(m+n) = O(m+n)$

(d) We first put the new element at $Y[m,n]$. The added element must be moved to proper position for maintaining the property of Young tableau. This goal is also achieved by recursion.

The recursion compares the current position with up and left positions. If both them are larger than the current position, we exchange current position with the larger one of them; otherwise, we exchange current position with the one whose value is larger than current position. After the exchange, we do next recursion for new position. It stops when up and left values are both smaller than current position.

The method is shown as follows:

Framework:
Step 1. Replace $Y[m,n]$ with the new element
Step 2. Call MOVE-UL(m,n)

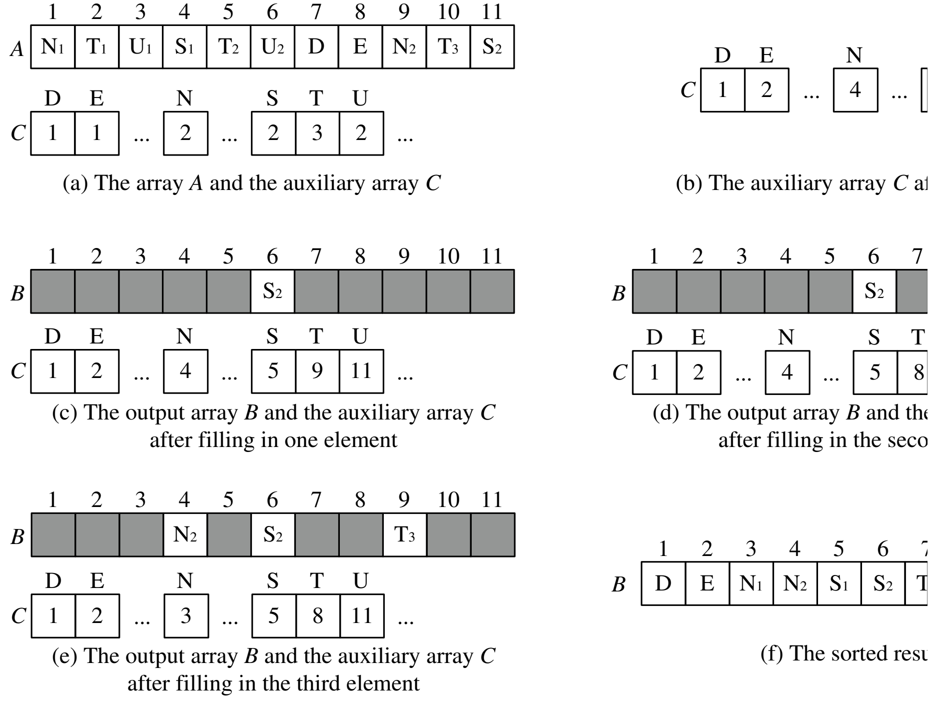

Figure 3: Illustration for Problem 1(c).

2	4	9	∞
3	8	16	∞
5	14	∞	∞
12	∞	∞	∞

Figure 4: Sample table for Problem 2(b).

```
MOVE-UL(i, j):
    assume Y[i, j] = −∞ if i < 1 or j < 1
    if Y[i − 1, j] < Y[i, j] and Y[i, j − 1] < Y[i, j] then RETURN
    if Y[i − 1, j] >= Y[i, j] and Y[i, j − 1] >= Y[i, j] then
        if Y[i − 1, j] > Y[i, j − 1] then
            swap Y[i, j] with Y[i − 1, j]
            call MOVE-UL(i − 1, j)
        else        swap Y[i, j] with Y[i, j − 1]
            call MOVE-UL(i, j − 1)
    else if Y[i − 1, j] >= Y[i, j] then
        swap Y[i, j] with Y[i − 1, j]
        call MOVE-UL(i − 1, j)
    else    swap Y[i, j] with Y[i, j − 1]
        call MOVE-UL(i, j − 1)
```

Time complexity:
$T(p) = T(p-1) + \Theta(1) = O(p)$
Hence,$T(m+n) = O(m+n)$

(e) **Step 1.** Insert n^2 values into table, needs $n^2 \cdot O(n+n) = O(n^3)$
Step 2. Extract n^2 values from table by EXTRACT-MIN, needs $n^2 \cdot O(n+n) = O(n^3)$
According to 1 and 2, we have $O(n^3)$ sorting method.

(f) Key idea:
See Figure 5 for illustration. For any position $[i, j]$, we can see that $\forall p, q, p \leq i, q \leq j$, $Y[p, q] \leq Y[i, j]$. That is, the gray positions are not greater than $Y[p, q]$. To use this property, we can compare target value with $Y[i, j]$, and then determine that the target value falls in gray or white positions.

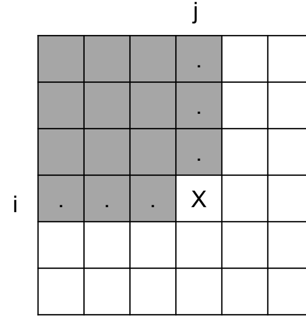

Figure 5: Illustration of Problem 2(f).

Framework:
Step 1. Call Determine($value, m, 1$)

Determine($value, i, j$):
  assume $Y[i, j] = -\infty$ if $i < 1$ or $j < 1$
  assume $Y[i, j] = \infty$ if $i > m$ or $j > n$
  if $Y[i, j] = \infty$ or $Y[i, j] = -\infty$ then RETURN false
  if $value = Y[i, j]$ then RETURN true
  else if $value > Y[i, j]$ then RETURN Determine($value, i, j + 1$)
  else if $value < Y[i, j]$ then RETURN Determine($value, i - 1, j$)

Time complexity:
$T(p) = T(p - 1) + \Theta(1) = O(p)$
Hence,$T(m + n) = O(m + n)$

3. (10)

(a) We will prove by induction that Stooge sort correctly sorts the input array $A[1 \ldots n]$.

The base case is for $n = 2$. In this case the first two lines of Stooge sort guarantee that a bigger element will be returned in a position following that of a smaller element. Thus, Stooge sort correctly sorts A when $n = 2$.

The inductive case happens when $n \geq 3$. Then, assume by inductive hypothesis that the recursive call to Stooge-sort in line 6 correctly sorts the elements in the first two-thirds of the input array A, and let A' be the array thus obtained. Also, assume by inductive hypothesis that the recursive call to Stooge-sort in line 7 correctly sorts the elements in the last two-thirds of the array A', and let A'' be the array thus obtained. Finally, assume by inductive hypothesis that the recursive call to Stooge-sort in line 8 correctly sorts the elements in the first two-thirds of the array A'', and let B be the (final) array thus obtained. Now we need to prove that at the end of the algorithm, the array B is sorted. To prove this, we just need to prove that for any two elements $A[i] < A[j]$, the position of $A[i]$ in B will be smaller than the position of $A[j]$, no matter which is their position in the input array A. Since this will be true for any pair $A[i]$; $A[j]$, then the correctness of Stooge sort will follow.

We divide the proof in two cases. (1) Consider the simple case in which the position of $A[i]$ is smaller than the position of $A[j]$ in the input array A. Then, clearly, from the three facts that we have established using the inductive hypothesis, we have that the position of $A[i]$ in B will be smaller than the position of $A[j]$. (2) Consider the more involved case in which the position of $A[j]$ is smaller than the position of $A[i]$ in the input array A. We need to prove that their order in B will be swapped. Here we have some other cases. First, if $A[j]$ and $A[i]$ are both in the first two-thirds of the input array A, then they will be swapped in the recursive call in line 6 (which is correct by inductive hypothesis) and will not be swapped again in the following recursive calls (again, by the inductive hypothesis). Second, if $A[j]$ and $A[i]$ are both in the last two-thirds of the input array A, then they will be swapped in the recursive call in line 7 (which is correct by inductive hypothesis) and will not be swapped again in the following recursive call (again, by the inductive hypothesis). Finally, if $A[j]$ is in the first one-third of the input array A and $A[i]$ is in the last one-third of the input array A, then we claim that their position are swapped either in the recursive call in line 7 or in the recursive call in line 8. In fact, assume their positions are not swapped in the recursive call in line 7. This means that after the execution of the recursive call in line 6 (which is correct by inductive hypothesis) the position of $A[j]$ in array A' is still in the first one-third and moreover, at least all the elements in the second third of A' are greater than $A[j]$. Then we have that since these elements are greater than $a[j]$, then they are greater than $A[i]$ (since we assumed $A[i] < A[j]$), and after the execution of the recursive call in line 7 (which is correct by inductive hypothesis), $A[j]$ will appear in the first two-thirds of A''. Now, since also $A[i]$ will remain in the first third of A'', their positions will be swapped by the execution of the recursive call in step 8.

(b) The recurrence for the worst-case running time of Stooge-Sort can be written as

$$T(n) = \begin{array}{cc} \Theta(1), & n \leq 2, \\ 3T(\frac{3n}{2}) + \Theta(1), & \text{otherwise.} \end{array}$$

And with the Master theorem, we can see that $T(n)$ can be asymptotically bounded as

$$T(n) = \Theta(n^{log_{3/2}3}).$$

(c) Since $log_{3/2}3 > 2$ and the worst-case running time bounds for the listed sorting algorithms are $\Omega(n^2)$, the professors do not deserve tenure in terms of running time.

4. (10)

(a) One possible algorithm is comparing each red jug with each blud jug. Since there are n red jugs and n blue jugs, it will take $\Theta(n^2)$ comparisons in the worst case.

(b) The bound can be proved with the following two facts.

- There are $(n!)$ possible matchings. Thus there are $n!$ leaves on the decision tree for the matching.
- At each comparison, there are only 3 possible cases $\{\geq, =, \leq\}$. Therefore, the degree of the decision tree is at most 3.

Combine these facts, any algorithm solving this problem must take at least $log_3(n!) = \Omega(n\lg n)$ comparisons.

(c) A possible algorithm works as in Figure 6.

Denote C_{ij} as the event that r_i and b_j is compared. Then the total number of comparisons is $\sum_{\forall i<j} I\{C_{ij} \bigcup C_{ji}\}$ and the proof is the same as the proof for *Quicksort*. Therefore, the expected number of comparisons is $O(n\lg n)$.

5. (10)

- case 1

If the input elements are divided into groups of 7, the lower bound of the number of elements that are greater/less than the partitioning elements x becomes

```
BR_MATCHING(B, R)
    If |B| = |R| = 0
        return;
    Initialize B_lower, B_greater, R_lower, R_greater as ∅;
    b_i ← randomly choose a jug from B;
    For each jug r* in R
        Compare b_i and r*;
        if r* has the same capacity as b_i, r_i ← r*;
        else if r* has larger capacity than b_i, R_greater ← R_greater ⋃{r*};
        else if r* has lower capacity than b_i, R_lower ← R_lower ⋃{r*};
    For each jug b* in (B − {b_i})
        Compare r_i and b*;
        if b* has larger capacity than r_i, B_greater ← B_greater ⋃{b*};
        else if b* has lower capacity than r_i, B_lower ← B_lower ⋃{b*};
    BR_MATCHING(B_lower, R_lower)
    BR_MATCHING(B_greater, R_greater)
```

Figure 6: The pseudo-code to complete jug matching.

$$4(\left\lceil \frac{1}{2} \left\lceil \frac{n}{7} \right\rceil \right\rceil - 2) \geq \frac{2n}{7} - 8.$$

And the recurrence can be written as

$$T(n) \leq T(\left\lceil \frac{n}{7} \right\rceil) + T(\frac{5n}{7} + 8) + O(n),$$

which can be shown to be $O(n)$ by substitution. Therefore, *Select* still run in linear time for this case.

- case 2

  If the input elements are divided into groups of 3, the lower bound of the number of elements that are greater/less than the partitioning elements x becomes

$$2(\left\lceil \frac{1}{2} \left\lceil \frac{n}{3} \right\rceil \right\rceil - 2) \geq \frac{n}{3} - 4.$$

  And the recurrence can be written as

$$T(n) \leq T(\left\lceil \frac{n}{3} \right\rceil) + T(\frac{2n}{3} + 4) + O(n),$$

  which cannot be upper-bounded by any $cn + \Theta(1), c \geq 0$. In other words, *Select* does not run in linear time for this case.

6. (10)

Let's start out by supposing that the median (the lower median, since we know we have an even number of elements) is in X. Let's call the median value m, and let's suppose that it's in $X[k]$. Then k elements of X are less than or equal to m and $n-k$ elements of X are greater than or equal to m. We know that in the two arrays combined, there must be n elements less than or equal to m and n elements greater

than or equal to m, and so there must be $n-k$ elements of Y that are less than or equal to m and $n-(n-k)=k$ elements of Y that are greater than or equal to m.

Thus, we can check that $X[k]$ is the lower median by checking whether $Y[n-k] \leq X[k] \leq Y[n-k+1]$. A boundary case occurs for $k=n$. Then $n-k=0$, and there is no array entry $Y[0]$; we only need to check that $X[n] \leq Y[1]$.

Now, if the median is in X but is not in $X[k]$, then the above condition will not hold. If the median is in $X[k']$, where $k' < k$, then $X[k]$ is above the median, and $Y[n-k+1] < X[k]$. Conversely, if the median is in $X[k'']$, where $k'' > k$, then $X[k]$ is below the median, and $X[k] < Y[n-k]$.

Thus, we can use a binary search to determine whether there is an $X[k]$ such that either $k < n$ and $Y[n-k] \leq X[k] \leq Y[n-k+1]$ or $k=n$ and $X[k] \leq Y[n-k+1]$; if we find such an $X[k]$, then it is the median. Otherwise, we know that the median is in Y, and we use a binary search to find a $Y[k]$ such that either $k < n$ and $X[n-k] \leq Y[k] \leq X[n-k+1]$ or $k=n$ and $Y[k] \leq X[n-k+1]$; such a $Y[k]$ is the median. Since each binary search takes $O(lgn)$ time, we spend a total of $O(lgn)$ time.

Here's how we write the algorithm in pseudocode:

```
TWO-ARRAY-MEDIAN(X, Y)
n ← length[X];                  ▷ n also equals length[Y]
median ← FIND-MEDIAN(X, Y, n, 1, n);
if median = NOT-FOUND
    then median ← FIND-MEDIAN(Y, X, n, 1, n);
return median;
```

```
FIND-MEDIAN(A, B, n, low, high)
if low > high
    then return NOT-FOUND;
    else k ← ⌊(low + high)/2⌋;
        if k = n and A[n] ≤ B[1]
            then return A[n];
        elseif k < n and B[n − k] ≤ A[k] ≤ B[n − k + 1]
            then return A]k];
        elseif A[k] > B[n − k + 1]
            then return FIND-MEDIAN(A, B, n, low, k − 1);
        else return FIND-MEDIAN(A, B, n, k + 1, high);
```

7. (10)

    (b) **O**

    (d) **O**

    (e) **X**

8. (10)

    One possible algorithm works with the following steps.

    (a) Insert the sequences one by one into the radix tree.

    (b) Traverse the radix tree with pre-order tree walk, and write down sequences visited.

    The correctness of the algorithm can be justified by proving that the pre-order tree walk visits sequences in monotonically increasing order. According to the structure of radix trees and the definition of "lexicographically less than," for any node i in a radix tree, we have

(The sequence on i) $<$ (any sequence in the left subtree of i) $<$ (any sequence in the right subtree of i.)

Therefore, the pre-order tree walk does visit sequences in monotonically increasing order.

The timing bound can be derived as follows. Inserting the sequences takes $\Theta(n)$ time and the pre-order tree walk takes $O(n)$ time. In conclusion, the timing of the algorithm is $\Theta(n)$.

9. (10)

(a) See Figure 7.

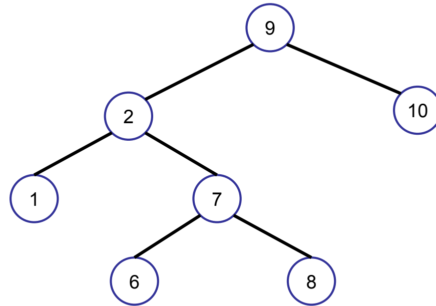

Figure 7: Sample table for Problem 9(a).

(b) See Figure 8.

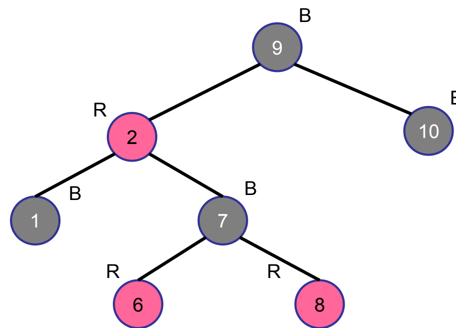

Figure 8: Sample table for Problem 9(b).

(c) See Figure 9.

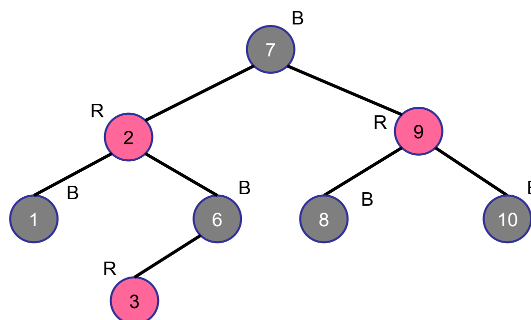

Figure 9: Sample table for Problem 9(c).

10. (10)

The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the sequence of dimensions $< 5, 10, 3, 12, 5, 50, 6 >$ are shown in Figure 10.The minimum number of scalar multiplications to multiply the six matrices is $m[1, 6] = 2010$ and its corresponding parenthesization is $((A_1A_2)((A_3A_4)(A_5A_6)))$.

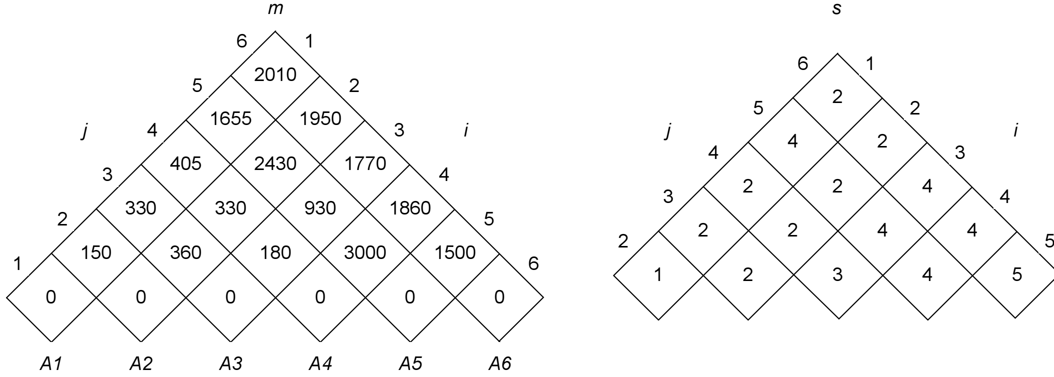

Figure 10: The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the sequence of dimensions $< 5, 10, 3, 12, 5, 50, 6 >$.

11. (15)

(a) Consider $L = 10, d_1 = 2, d_2 = 5$, i.e. $(d_0, d_1, d_2, d_3) = (0, 2, 5, 10)$.
For sequence (1,2), the cost is $10 + 8 = 18$.
For sequence (2,1), the cost is $10 + 5 = 15$.
They are different.

(b) For any wood between marked places i and j, we determine the next cut which would give the smallest payment. Hence, we must check every cut between i and j, and then find out the solution recursively. The recurrence of $c(i, j)$ is defined as follows:

$$c(i,j) = \begin{cases} \min_{i \le k < j}\{c(i,k) + c(k,j)\} + (d_j - d_i), & \text{if } i < j \\ 0, & \text{otherwise.} \end{cases}$$

(c) The algorithm starts from small marked intervals to large marked intervals, and determine the next cut for every interval.
Here's how we write the algorithm in pseudocode (the running time is $O(n^3)$):

```
WOOD-CUT(d, n)
Sisoutputqueue
T[0 : n + 1] ← 0
C[0 : n + 1] ← ∞
for z ← 0 to n + 1
    C[z, z] ← 0
for l ← 1 to n + 1
    for i ← 0 to n + 1 − l
        j ← i + l
        for k ← i to j − 1
            if C[i, j] > (C[i, k] + C[k, j]) then
                C[i, j] ← C[i, k] + C[k, j]
                T[i, j] ← k
        C[i, j] ← C[i, j] + d_j − d_i
TRACE(S, T, 0, n + 1)


TRACE(S, T, i, j)
if j = i + 1 then RETURN
S.push(T[i, j])
TRACE(S, T, i, T[i, j])
TRACE(S, T, T[i, j], j)
```

12. (30) DIY