Sample Solutions to Homework #2

1. (15)
   (a) See Figure 1.

   (a) Heap built by Build-Max-Heap
   (b) First exchange followed by Max-Heapify
   (c)
   (d)
   (e)
   (f)
   (g)
   (h)
   (i)
   (j)
   (k)
   (l)
   (m)
   (n)

   (o) Sorted sequence

   Figure 1: Illustration for Problem 1(a).

   (b) See Figure 2.
   (c) See Figure 3.

2. (20)
   (a) See Figure 4.
Figure 2: Illustration for Problem 1(b).

(a) The array $A$ and the auxiliary array $C$

(b) The auxiliary array $C$ after accumulation

(c) The output array $B$ and auxiliary array $C$ after filling in one element

(d) The output array $B$ and auxiliary array $C$ after filling in the second element

(e) The output array $B$ and auxiliary array $C$ after filling in the third element

(f) The sorted result

Figure 3: Illustration for Problem 1(c).

(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:

Framework:

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

MOVE-RD$(i,j)$:

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

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$(m,n)$
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 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)

Time complexity:
T(p) = T(p − 1) + Θ(1) = O(p)
Hence, T(m + n) = O(m + n)

(e) Step 1. Insert n^2 values into table, needs n^2 · O(n + n) = O(n^3)
Step 2. Extract n^2 values from table by EXTRACT-MIN, needs n^2 · 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 ∀p, q, p ≤ i, q ≤ j, Y[p, q] ≤ 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] = −∞ if i < 1 or j < 1 or i > m or j > n
if Y[i, j] = −∞ 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. (5)
Construct a counting array \( C \) used in counting sort in \( O(n+k) \) time. Since \( C[a] \) denotes the number of integers which fall into the range \([0, a]\), it is obvious to see the answer is \( C[b] - C[a-1] \), where \( C[-1] \) is defined as 0. Therefore, the query then can be answered in \( O(1) \) time.

4. (10)
(a) See Figure 6.

(b) See Figure 7.

5. (20)
(a) Since there are \( n \) red jugs and \( n \) blue jugs, that will take \( \Theta(n^2) \) comparisons to compare each red jug with each blue jug in the worst case.

(b) The computation of the algorithm can be viewed in terms of a decision tree. Every internal node is labelled with two jugs (red and blue), and has three outgoing edges (red jug smaller, same size, or larger than the blue jug). The leaves are labelled with a unique matching of jugs. The height of the decision tree is equal to the worst-case number of comparisons the algorithm has to make to determine the matching. To bound that size, we first compute the number of possible matchings for \( n \) red and \( n \) blue jugs. If we label the red jugs from 1 to \( n \) and we also label the blue jugs from 1 to \( n \) before starting the comparisons, every outcome of the algorithm can be represented as a set
\[ \{(i, \pi(i)) : 1 \leq i \leq n \text{ and } \pi \text{ is a permutation on } \{1, \ldots, n\}\}, \]
which contains the pairs of red jugs (first component) and blue jugs (second component) that are matched up. Since every permutation \( \pi \) corresponds to a different outcome, there must be exactly \( n! \) different results.

Therefore, the height $h$ of the decision tree can be bounded as follows: Every tree with a branching factor of 3 (every inner node has at most three children) has at most $3^h$ leaves, and the decision tree must have at least $n!$ children. This statement follows that

$$3^h \geq n! \geq (n/e)^n \Rightarrow h \geq n \log_3 n - n \log_3 e = \Omega(n \log n).$$

So any algorithm solving the problem must use $\Omega(n \log n)$ comparisons.

(c) $- R$: red jugs with labelled numbers 1, 2, ..., $n$, $R \subseteq \{1, ..., n\}$.
- $B$: blue jugs with labelled number 1, 2, ..., $n$, $B \subseteq \{1, ..., n\}$.
- output: $n$ distinct pairs $(i, j)$ indicate the red jug $i$ and the blue jug $j$ have the same volume.
- Pseudocode:

```plaintext
MATCH-JUGS(R, B)
1 if |R| = 0
2 then return
3 if |R| = 1
4 then let $R = \{r\}$ and $B = \{b\}$
5 output "$(r, b)$"
6 return
7 else $r \leftarrow$ a randomly chosen jug in $R$
8 compare $r$ to every jug of $B$
9 $B_\leq$ the set of jugs in $B$ that are smaller than $r$
10 $B_\geq$ the set of jugs in $B$ that are larger than $r$
11 $b \leftarrow$ the one jug in $B$ with the same size as $r$
12 compare $b$ to every jug of $R - \{r\}$
13 $R_\leq$ the set of jugs in $R$ that are smaller than $b$
14 $R_\geq$ the set of jugs in $R$ that are larger than $b$
15 output "$(r, b)$"
16 MATCH-JUGS($R_\leq$, $B_\leq$)
17 MATCH-JUGS($R_\geq$, $B_\geq$)
```

The procedure will be called only with inputs that can be matched, which means $|R| = |B|$.
The correctness of the algorithm can be proved as follows. Once the jug \( r \) is randomly picked from \( R \), there will be a matching among the jugs in volume smaller than \( r \) which are in the sets \( R_\prec \) and \( B_\prec \), and likewise between the jugs larger than \( r \) which are in the sets \( R_\succ \) and \( B_\succ \). Since 
\(|R_\prec| + |R_\succ| < |R|\) in every recursive step, the size of the first parameter reduces with every recursive call, and the recursion will be terminated when the parameter reaches 0 or 1.

The analysis of the expected number of comparisons is similar to that of the quicksort algorithm. To analyze the expected number of comparisons, we first order the red and blue jugs as \( r_1, \ldots, r_n \) and \( b_1, \ldots, b_n \) where \( r_i < r_{i+1} \) and \( b_1 < b_{i+1} \) for \( i = 1, \ldots, n \), and \( r_i = b_i \). Then, the indicator random variable can be defined as follows.

\[ X_{ij} = I\{\text{red jug } r_i \text{ is compared to blue jug } b_j\} \]

As in quicksort, a given pair \( r_i \) and \( b_i \) is compared at most once. When comparing \( r_i \) to every jug in \( B \), jug \( r_i \) will not be put in either \( R_\prec \) or \( R_\succ \). When comparing \( b_i \) to every jug in \( R - \{r_i\} \), jug \( b_i \) is not put into either \( B_\prec \) or \( B_\succ \). The total number of comparisons is

\[
X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.
\]

Then, the expected value of \( X \) can be calculated as follows.

\[
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{r_i \text{ is compared to } b_j\}
\]

Once we choose a jug \( r_k \) such that \( r_i < r_k < b_j \), we will put \( r_i \) in \( R_\prec \) and \( b_j \) in \( R_\succ \), and so \( r_i \) and \( b_j \) will never be compared again. Jugs \( r_i \) and \( b_j \) will be compared if and only if the first jug in \( R_{ij} = \{r_i, \ldots, r_j\} \) to be chosen is either \( r_i \) or \( r_j \). Any jug in \( R_{ij} \) is equally likely to be first one chosen. Since \( |R_{ij}| = j - i + 1 \), the probability of any given jug being the first one chosen in \( R_{ij} \) is \( 1/(j - i + 1) \). Similar to the analysis of quicksort, we can prove that the expected number of comparisons is \( O(n \lg n) \).

The worst-case number of comparisons is \( \Theta(n^2) \). In the worst-case we always choose the largest or smallest jugs to partition the sets, which reduces the set sizes only by 1. The worst-case running time obeys the recurrence \( T(n) = T(n-1) + \Theta(n) \), and the worst-case number of comparisons is \( T(n) = \Theta(n^2) \).

6. (10)

We can optimize by splitting the input in pairs and comparing each pair. After \( n/2 \) comparisons, we have reduced the potential minimums and potential maximums to \( n/2 \) each. Furthermore, those two sets are disjoint so now we have two problems, one minimum and one maximum, each of size \( n/2 \). The total number of comparisons is

\[ n/2 + 2(n/2 - 1) = n/2 + n - 2 = 3n/2 - 2 \]

This assumes that \( n \) is even. If \( n \) is odd we need one additional comparison in order to determine whether the last element is a potential minimum or maximum. Hence the ceiling.

7. (10)

This problem can be solved by a binary-search approach. Let \( p \) be the median of \( X[1..n] \), and \( q \) be the median of \( Y[1..n] \). If \( p > q \), then the median of the two arrays is in \( X[1..n/2] \) or \( Y[n/2..n] \). If \( p \leq q \), then the median of the two arrays is in \( X[n/2..n] \) or \( Y[1..n/2] \). Therefore, we can use recursion to solve this problem. For any instance with two sorted arrays \( X[1..n] \) and \( Y[1..n] \), we first compare the median of them, and then follow the mentioned rules to divide the two arrays into half of the original size. After dividing, we can let the new smaller arrays as the new instance and apply the same strategy recursively.

Obviously, this binary-search approach divides the problem size into half size every recursion. Thus we have the time complexity, \( T(n) = T(n/2) + \Theta(1) = O(\lg n) \)
8. (10) The optimal y-coordinate for Professor Olay’s east-west oil pipeline is as follows:

- If \( n \) is even, then on either the oil well whose y-coordinate is the lower median or the one whose y-coordinate is the upper median, or anywhere between them.
- If \( n \) is odd, then on the oil well whose y-coordinate is the median.

**Proof:** We examine various cases. In each case, we will start out with the pipeline at a particular y-coordinate and see what happens when we move it. We’ll denote by \( s \) the sum of the north-south spurs with the pipeline at the starting location, and \( s' \) will denote the sum after moving the pipeline.

We start with the case in which \( n \) is even. Let us start with the pipeline somewhere on or between the two oil wells whose y-coordinates are the lower and upper medians. If we move the pipeline by a vertical distance \( d \) without crossing either of the median wells, then \( n/2 \) of the wells become \( d \) farther from the pipeline and \( n/2 \) become \( d \) closer, and \( s' = s + dn/2 - dn/2 = s \); thus, all locations on or between the two medians are equally good.

Now suppose that the pipeline goes through the oil well whose y-coordinate is the upper median. What happens when we increase the y-coordinate of the pipeline by \( d > 0 \) units, so that it moves above the oil well that achieves the upper median? All oil wells whose y-coordinates are at or below the median become \( d \) units farther from the pipeline, and there are at least \( n/2 + 1 \) such oil wells (the upper median, and every well at or below the lower median). There are at most \( n/2 - 1 \) oil wells whose y-coordinates are above the upper median, and each of these oil wells becomes at most \( d \) units closer to the pipeline when it moves up. Thus, we have a lower bound on \( s' \) of \( s' \geq s + d(n/2 + 1) - d(n/2 - 1) = s + 2d > s \). We conclude that moving the pipeline up from the oil well at the upper median increases the total spur length. A symmetric argument shows that if we start with the pipeline going through the oil well whose y-coordinate is the lower median and move it down, then the total spur length increases.

We see, therefore, that when \( n \) is even, an optimal placement of the pipeline is anywhere on or between the two medians.

Now we consider the case when \( n \) is odd. We start with the pipeline going through the oil well whose y-coordinate is the median, and we consider what happens when we move it up by \( d > 0 \) units. All oil wells at or below the median become \( d \) units farther from the pipeline, and there are at least \( (n+1)/2 \) such wells (the one at the median and the \( (n-1)/2 \) at or below the median). There are at most \( (n-1)/2 \) oil wells above the median, and each of these becomes at most \( d \) units closer to the pipeline. We get a lower bound on \( s' \) of \( s' \geq s + d(n + 1)/2 - d(n - 1)/2 = s + d > s \), and we conclude that moving the pipeline up from the oil well at the median increases the total spur length. A symmetric argument shows that moving the pipeline down from the median also increases the total spur length, and so the optimal placement of the pipeline is on the median.

Since we know we are looking for the median, we can use the linear-time median-finding algorithm. Thus, the optimal location can be determined in linear time.

9. (10)

(b) \( O \)

(d) \( O \)

10. (10) One possible algorithm works in 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 $\Theta(n)$ time. In conclusion, the timing of the algorithm is $\Theta(n)$.

11. (20)
   (a) See Figure 8(a).
   (b) See Figure 8(b).
   (c) See Figure 9(a) for the binary search tree.
       Yes, see Figure 9(b) for the red-black tree.
   (d) See Figure 10(a) for the inserted red-black tree.
       See Figure 10(b) for the legal red-black tree.
   (e) See Figure 11(a) for the deleted red-black tree.
       See Figure 11(b) for the legal red-black tree.

12. (20)
   (a) If we know $N_{h-1}$ and $N_{h-2}$, we can determine $N_h$. Since this $N_h$—noded tree must have a height
       $h$, the root must have a child that has height $h - 1$. To minimize the total number of nodes in this
Figure 10: Illustration for Problem 11(d). (a) Inserting key 3 to the original red-black tree. (b) Legal red-black tree.

Figure 11: Illustration for Problem 11(e). (a) Deleting key 6 to the original red-black tree. (b) Legal red-black tree.

tree, we would have this sub-tree contain $N_{h-1}$ nodes. By the property of an AVL tree, if one child has height $h_1$, the minimum height of the other child is $h - 2$. By creating a tree with a root whose left sub-tree has $N_{h-1}$ nodes and whose right sub-tree has $N_{h-2}$ nodes, we have constructed the AVL tree of height $h$ with the least nodes possible. This AVL tree has a total of $N_{h-1} + N_{h-2} + 1$ nodes.

The base cases are $N_0 = 1$ and $N_1 = 2$. From here, we can iteratively construct $N_h$ by using the fact that $N_h = N_{h_1} + N_{h_2} + 1$ that we figured out above.

\[
\begin{align*}
N_h & = N_{h-1} + N_{h-2} + 1 \\
N_{h-1} & = N_{h-2} + N_{h-3} + 1 \\
N_h & = (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1 \\
N_h & > 2N_{h-2} \\
N_h & > 2^h \\
lg N_h & > lg 2^h \\
2 lg N_h & > h
\end{align*}
\]
Therefore, an AVL tree with n nodes has height $O(\lg n)$.

(b) See Figure 12 for four cases for BALANCE($x$).

(c)
- **Pseudocode:**

```plaintext
AVL-INSERT($x$, $z$)
1  TREE-INSERT($x$, $z$
2  BALANCE($x$)
```

(d) AVL-insertions are binary search tree insertions plus at most two rotations. Since binary search tree insertions take $O(h)$ time, rotations are $O(1)$ time, and AVL trees have $h = O(\lg n)$, AVL insertions take $O(\lg n)$ time.

13. (0) Dynamic Programming. Please finish this on HW3.

14. (0) Dynamic Programming. Please finish this on HW3.

15. (40) DIY.