1. (20 points) Sorting and Selection (I). Mark the correct answer(s). Justification is optional.

   a. To sort a general array, which of the following algorithms (in their textbook forms, i.e., no modifications) satisfies the following two criteria: (1) runs in $\Theta(n \log n)$ time in the worst case; (2) is stable?

      (a) Randomized quick sort
      (b) Merge sort
      (c) Insertion sort
      (d) Heap sort
      (e) Counting sort
      (f) None of the above

   b. To sort a general array, which of the following algorithms (in their textbook forms, i.e., no modifications) satisfies the following two criteria: (1) runs in $\Theta(n \log n)$ time in the worst case and (2) is in-place?

      (a) Randomized quick sort
      (b) Merge sort
      (c) Insertion sort
      (d) Heap sort
      (e) Counting sort
      (f) None of the above

   c. Suppose that we have an array of $n$ data records to sort, and that the key of each record has the value either 0 or 1. Which of the following algorithms satisfies the following two criteria: (1) runs in $\Theta(n)$ time; (2) is stable.

      (a) Counting sort
      (b) Merge sort
      (c) Heap sort
      (d) Partition
      (e) None of the above

   d. Suppose that we have an array of $n$ data records to sort, and that the key of each record has the value either 0 or 1. Which of the following algorithms can satisfy the following two criteria: (1) runs in $\Theta(n)$ time; (2) is in place.

      (a) Counting sort
      (b) Merge sort
      (c) Heap sort
      (d) Partition
      (e) None of the above
c. For an already sorted array, which of the following algorithms will run in linear time?
   
   (a) Randomized quick sort
   (b) Merge sort
   (c) Insertion sort
   (d) Selection sort
   (e) None of the above

f. Which of the following statements is CORRECT?

   (a) Quick sort and Randomized quick sort have the same worst-case asymptotic running time.

   (b) Finding median from an unsorted array can be done in $\Theta(n)$ time in theory.

   (c) In the worst case, Randomized SELECT has the same running time as randomized quick sort.

   (d) If an array $A$ is a max heap, then reversing $A$ will result in a min heap.

   (e) The worst-case running time of Randomized SELECT is $\Theta(n \log n)$ while the standard (non-randomized) SELECT is $\Theta(n^2)$.

   (f) Counting sort is usually more efficient than other sorting algorithms for sorting real numbers, because its running time is $\Theta(n)$ while the others are at least $\Theta(n \log n)$.

   (g) None of the above.

2. (10 points) Sorting and Selection (Part II)
   Suppose that you have an array of $n$ numbers, and you would like to get the $k$ smallest numbers in sorted order. You are considering the following three options:
   
   (i) Sort the numbers into non-decreasing order using merge sort and then return the first $k$ numbers.

   (ii) Build a min-heap and then call Extract-Min $k$ times.

   (iii) Use the randomized select algorithm to find the $k$-th smallest number, partition around that number, and sort the $k$ smallest numbers in place with quick sort.
a. Analyze the expected running time of the above three methods in terms of both $n$ and $k$.

(i): $O(n \log n) + O(k) = O(n \log n)$ regardless of $k$

(ii): $O(n) + O(k \log n)$

(iii): $O(n) + O(k \log k)$

b. Which method(s) would you prefer and why, when $k = \Theta(1)$, i.e., $k$ is a relatively small number independent of $n$ (e.g., $k = 10$)?

(i): guaranteed to be $O(n)$

(iii): $O(n)$ expected but could be $O(n^2)$ in worst case.

3. (15 points) Heaps

a. Does the array [10 7 9 6 4 2 8 3 1 5] represent a max heap? Why or why not?

b. Consider the following procedure for building a max heap. The algorithm takes an unsorted array $A$ as an input and make $A$ a heap.

```
BuildHeap(A) {
    heap_size(A) = length(A);
    for (i = floor(length(A)/2) downto 1)
        Heapify(A, i);
}
```
(a) Illustrate how the procedure BuildHeap works using array \( A = [14, 15, 10, 5, 6, 16, 11, 3, 12, 20] \). Show necessary intermediate steps for full credit (e.g., show the content of the tree after each call to Heapify). Make sure your final tree is indeed a heap.

(b) What is the asymptotically tight time complexity of BuildHeap?

(c) Suppose that we change the third line of BuildHeap to “for (i = length[A] downto 1)”, would the algorithm still work? If no, state the reason. If yes, what is the time complexity of the modified algorithm?

Yes. Still \( \Theta(n) \)

(d) Suppose that we change the third line to “for (i = 1 to length[A]/2)”, would the algorithm still work? If no, state the reason. If yes, what is the time complexity of the modified algorithm?

No. Heapify assumes subtrees are heaps.

(c) Describe an efficient, in-place algorithm to detect duplicated elements in a random array and report the number of occurrences for each of these duplicated elements. Your algorithm should run in \( O(n \log n) \) time in the worst case.

1. Heap sort the array.
2. Go over the heap array and count duplicated elements.
4. (10 points) Analysis of randomized algorithm

Consider the following randomized algorithm which takes an integer $n$ as input. Analyze its running time as a function of $n$.

```
RandAlg(n) {
    if (n <= 1) return;
    for (i = 1; i <= n; i++) { print i; }
    r = rand(); // r is a random number between 0 and 1
    p = 0.5;
    if (r <= p) {
        RandAlg(n/3);
        RandAlg(n/3);
    } else
        RandAlg(n/3);
        RandAlg(n/3);
        RandAlg(n/3);
}
```

a. What is the worst-case running time of RandAlg?

$$T(n) = 3 T(n/3) + O(n) \implies T(n) = \Theta(n \log n)$$

b. What is the best-case running time of RandAlg?

$$T(n) = 2 T(n/3) + O(n) \implies T(n) = \Theta(n)$$

c. What is the expected running time of RandAlg?

$$T(n) = 2.5 T(n/3) + O(n) \implies T(n) = \Theta(n)$$

d. If line 5 is changed to “p = 0.1”, what will be the expected running time of RandAlg?

$$T(n) = (0.2 + 2.7) T(n/3) + O(n) \implies T(n) = O(n)$$

$$= 2.9 T(n/3) + O(n)$$

e. When p decreases, how does the expected running time change?

$$T(n) = O(n) \text{ when } p < 0.5$$

$$\implies T(n) = O(n \log n)$$
5. (20 points) Longest common subsequence (LCS).

a. (10 points)
Use dynamic programming to find LCS between two strings BAABAB and ABBAAB. In your trace-back, show only one path (if there are multiple), and the LCS corresponding to that path.

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

b. (10 points) Given two strings that contain only ‘A’s and ‘B’s, revise the LCS algorithm to find a “weighted” LCS, where for each ‘A’ in the common subsequence we receive 1 points and for each ‘B’ in the common subsequence we receive 2 points. Hint: Let WLCS(i, j) represents the maximum score for the weighted LCS between X[1..i] and Y[1..j]. Define WLCS(i, j) recursively and then fill in the table below to solve the problem. Your final solution should include both the score and the weighted LCS.

\[
WLCS(i, j) = \begin{cases} 
\text{WLCS}(i-1, j) + s(X_i, Y_j), & \text{if } X_i = Y_j = A \\
\text{WLCS}(i-1, j), & \text{if } X_i = Y_j = B \\
\text{WLCS}(i, j-1), & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
6. (15 + 10 points) Dynamic programming II (Gas Station Location Problem).

   a. (15 points) You are given a list of $n$ locations to consider building gas stations. Each location has its own estimated profit value $p_i$. Your task is to find an optimal selection that will result in the highest estimated total profit, subject to the policy that any two selected locations may NOT be adjacent to each other (i.e., you cannot select both location $i$ and $i + 1$.)

      (a) Let $S(i)$ be the estimated total profit of the optimal plan that considers only locations $1$ to $i$. Write down the recurrence function for computing $S(i)$.

      $$ S(i) = \max \left\{ S(i-1), S(i-2) + p_i \right\} $$

      (b) Show how to use the recurrence function and dynamic programming to find the optimal plan given the estimated profits below.

<table>
<thead>
<tr>
<th>Location $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit $p_i$</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>select = no</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>24</td>
<td>29</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>select = yes</td>
<td>9</td>
<td>3</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>24</td>
<td>29</td>
<td>30</td>
<td>27</td>
<td>35</td>
</tr>
<tr>
<td>$S(i)$</td>
<td>9</td>
<td>9</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>24</td>
<td>29</td>
<td>30</td>
<td>30</td>
<td>35</td>
</tr>
</tbody>
</table>

   Optimal value:

   Locations (indices) selected in an optimal solution:

   b. (Extra credit: 10 points) Now consider a modification to the policy about the distance between selected locations. The new policy says that you cannot have three selected locations next to each other, i.e., for any three selected locations $i < j < k$, $k - i$ needs to be $> 2$. (For example, you can select 1, 2, 4 or 1, 3, 4, but not 1, 2, 3.) Provide a dynamic programming algorithm to solve the problem. Hint: at each location $i$, consider three possible cases: (1) location $i$ is not selected, (2) location $i$ is selected but location $i - 1$ is not selected, and (3) both location $i$ and location $i - 1$ are selected.

   (a) Let $P(k, i)$ be the optimal total profit for the first $i$ locations, where location $i$ falls into case $k$ ($1 \leq k \leq 3$) defined above. Give the recurrence to calculate $P(k, i)$.

   $$ P(1, i) = \max_{k=1, 2} (P(k, i-1)) $$

   $$ P(2, i) = P_i + P(1, i-1) $$

   $$ P(3, i) = P_i + P(2, i-1) $$
(b) Use the recurrence above and dynamic programming to find an optimal selection for the following problem.

<table>
<thead>
<tr>
<th>Location $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit $p_i$</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>$P(1, i)$</td>
<td>0</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>23</td>
<td>24</td>
<td>30</td>
<td>32</td>
<td>36</td>
<td>39</td>
</tr>
<tr>
<td>$P(2, i)$</td>
<td>0</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>20</td>
<td>19</td>
<td>30</td>
<td>26</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>$P(3, i)$</td>
<td>0</td>
<td>12</td>
<td>23</td>
<td>24</td>
<td>26</td>
<td>32</td>
<td>37</td>
<td>40</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Optimal value:

Locations (indices) selected in an optimal solution:

7. (10 points) Greedy algorithm.

a. (8 points) Use greedy algorithm to solve the fractional knapsack problem. The knapsack has a weight limit of 10LB.

<table>
<thead>
<tr>
<th>Item ID</th>
<th>Weight (LB)</th>
<th>Value ($)</th>
<th>Value / Weight</th>
<th>Weight taken</th>
<th>Value taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>15</td>
<td>5</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>30</td>
<td>6</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>12</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>7</td>
<td>3.5</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>10</td>
<td>57</td>
</tr>
</tbody>
</table>

b. (1 point) If the value of each item in the list above is now increased by the same fixed constant (e.g., $1), will you need to re-run the algorithm to get the optimal selection and why (or why not)?

Yes, order may change.

c. (1 point) If the value of each item in the list above is now increased by the same fixed percentage (e.g., 20%), will you need to re-run the algorithm to get the optimal selection and why (or why not)?

No. Order stays the same.
8. (10 points) Extra credit: Algorithm design.

Dynamic Programming. Design a dynamic programming algorithm to solve the maximum subarray problem: Given a one-dimensional array of real numbers (containing at least one positive number), find a contiguous subarray that has the largest sum. For example, for the array \( A = [-2, 1, -3, 4, -1, 2, 1, -5, 4] \), the contiguous subarray with the largest sum is \([4, -1, 2, 1]\), with sum 6. (Hint: let \( A_i \) be the \( i \) value in the array, \( S(i) \) be the largest sum for the first \( i \) elements. Also define \( P(i, 0) \) as the largest sum for the first \( i \) elements when \( A_i \) is not selected, and \( P(i, 1) \) as the largest sum for the first \( i \) elements when \( A_i \) is selected.)

\[
\begin{align*}
\mathbf{\max}\{ \rho(i, 0) = \rho(i-1, 0), \rho(i-1, 1) \} \\
\rho(i, 1) &= A_i + \mathbf{\max}\{ \rho(i-1, 1) \} \\
S(i) &= \mathbf{\max}(\rho(i, 0), \rho(i, 1))
\end{align*}
\]

-2 1 -3 4 -1 2 1 -5 4
-2 0 0 1 1 4 4 5 6 6
-2 1 -2 4 3 5 6 1 5