1. (18 points) Sorting and Selection (Part I)

Indicate whether the following statement is true or false. Briefly justify your answers.

a. Merge sort is more efficient than heap sort asymptotically, because the former is a stable sorting algorithm while the latter is not.
   
   False. First of all, merge sort and heap sort have the same time and space complexity. Secondly, stability of sorting has nothing to do with efficiency.

b. The running time of counting sort depends on the values of the input and is not always $\Theta(n)$.
   
   True. Counting sort runs in $\Theta(n + k)$ and is in $\Theta(n)$ only if $k \in O(n)$.

c. Quick sort runs in linear time for an already sorted array.
   
   False. Even best case running time of quick sort is $n \log(n)$. In fact, for increasingly sorted array, if the first element is always taken as pivot, quick sort runs in $\Theta(n^2)$.

d. In theory, quick sort can be implemented to run in $\Theta(n \log n)$ time even in the worst case, for example, by using the Linear Time Select algorithm to always find median as pivot.
   
   True. It takes $\Theta(n)$ time to find median, and using median as pivot can result in perfectly balanced partition. Therefore the running time of quick sort can be guaranteed to be $\Theta(n \log n)$.

e. No sorting algorithm can be designed to satisfy the following three requirements simultaneously: (1) has $\Theta(n \log n)$ running time, (2) is stable, and (3) is in-place.
   
   False. While we have not learned any algorithm that satisfies all three requirements simultaneously, heap sort runs in $\Theta(n \log n)$ running time and is in-place. It can be modified to be stable by using the original index of the elements as a secondary key.

f. Quick sort and randomized quick sort have the same asymptotic worst-case running time.
   
   True. The worst-case running time of both quick sort and randomized quick sort are $\Theta(n^2)$.

g. Finding the median from a sorted array takes the same amount of time as from an unsorted array.
   
   False. Finding median from a sorted array takes constant time. (Just return the middle element.)
h. The advantage of a heap is that it enables you to get all elements in an array into sorted order in $\Theta(n)$ time.

False. Heap can be constructed in $\Theta(n)$ time, but each retrieve takes $\log(n)$ time and to completely sort all elements using heap sort takes $\Theta(n \log n)$ time.

i. In the worst case, randomized select has the same asymptotic running time as randomized quick sort.

True. The worst-case running time of both randomized select and randomized quick sort are $\Theta(n^2)$.

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.

a. Analyze the running time of the above three methods in terms of both $n$ and $k$.

Strategy (a) takes $\Theta(n \log n) + \Theta(k) = \Theta(n \log n)$ time. Strategy (b) takes $\Theta(n) + \Theta(k \log n)$ time. Strategy (c) takes $\Theta(n) + \Theta(k \log k)$ expected time, but the worst case could be $\Theta(n^2)$.

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., $k = 10$)?

For $k \in \Theta(1)$, (b) is theoretically most efficient, with a $\Theta(n)$ running time even in the worst case. While (c) can have a worst-case running time $\Theta(n^2)$, its expected or average running time is also $\Theta(n)$. In fact, when $n$ is much larger than $k$, strategy (c) is expected to be more efficient than (b) in practice, because of the relatively larger constant factor associated with buildHeap.

c. Which method(s) would you prefer and why, when $k = \Theta(n)$, i.e., $k$ is proportional to $n$ (e.g., $k = n/4$).

For $k \in \Theta(n)$, strategy (a) and (b) have the same time complexity $\Theta(n \log n)$. Strategy (a) is probably more straightforward and has smaller constant factor if $k$ is close to $n$. While strategy (c) is also expected to run in $\Theta(n \log n)$ on average, in the worst case it can be $\Theta(n^2)$.

3. (16 points) Heaps

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

Yes. All nodes have values smaller than their respective parents.
b. Consider the following two procedures for building a max heap. Both algorithms take an unsorted array \( A \) as an input and make \( A \) a heap.

\[
\begin{align*}
\text{BuildHeap}(A) \{ \\
& \text{heap\_size}(A) = \text{length}(A); \\
& \text{for} \ (i = \text{floor}(\text{length}[A]/2) \ \text{downto} \ 1) \\
& \quad \text{Heapify}(A, i); \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{BuildHeap}'(A) \{ \\
& \text{heap\_size}(A) = 1; \\
& \text{for} \ (i = 2 \ \text{to} \ \text{length}[A]) \\
& \quad \text{HeapInsert}(A, A[i]); \\
\}
\end{align*}
\]

(1) Illustrate how the procedure \text{BuildHeap} works on an array \([1 \ 5 \ 6 \ 4 \ 2 \ 7 \ 3]\). Show necessary intermediate steps for full credit (e.g., show the content of the tree after each \text{Heapify}).

(II) Illustrate how the procedure \text{BuildHeap}' works using the same example. Show necessary intermediate steps for full credit (e.g., show the content of the heap after each \text{HeapInsert}).

(III) Do the two procedures \text{BuildHeap}' and \text{BuildHeap} have the same time complexity asymptotically? Briefly justify your answer.

The time complexity using \text{BuildHeap} is \( \Theta(n) \). In contrast, the worst-case time complexity of \text{BuildHeap}' using \text{HeapInsert} is \( \Theta(n \log n) \). In the worst case, each insertion takes time \( h \), which is the current height of the heap. A heap with \( i \) elements have height \( \lceil \log_2 i \rceil \). Therefore, the overall complexity is \( \sum_{i=2}^{n} \lceil \log_2 i \rceil \), which is in \( \Theta(n \log n) \), as shown below.

\[
\sum_{i=2}^{n} \lceil \log_2 i \rceil \leq \sum_{i=2}^{n} \log_2 i \leq \log_2(n!) = \Theta(n \log n).
\]

\[
\sum_{i=2}^{n} \lceil \log_2 i \rceil \geq \sum_{i=2}^{n} (\log_2 i - 1) = \log_2(n!) - n = \Theta(n \log n - n) = \Theta(n \log n).
\]
4. (18 points) Longest common subsequence (LCS).

a. (15 points) Use dynamic programming to find LCS between two strings ACDBDCA and ABDBCDAC. If there are multiple longest common subsequences, report all of them.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>D</th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
<td>2</td>
<td>(3)</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>(2)</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>(4)</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>(3)</td>
<td>3</td>
<td>(4)</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>(1)</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>(5)</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>(2)</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>(5)</td>
<td>5</td>
</tr>
</tbody>
</table>

There are actually four longest common subsequences of length 5: ABDCA, ADBDA, ADBDC, and ADBCA.

b. (3 points) The running time of LCS is \( \Theta(mn) \), where \( m \) and \( n \) are the lengths of the two strings, respectively. This time complexity is valid only if you are interested in finding a longest common subsequence, even if multiple exist. Argue (or use an intuitive example to show) that if you are interested in returning ALL longest common subsequences, the running time may be much longer.

The key is that the total number of LCSs between two strings could be exponential. For example considering two strings ABCDEFGHIJ and BADCFEHGJI, each having length 10. There are \( 2^5 = 32 \) LCSs of length 5, by randomly picking one char from each pair (A/B)(C/D)(E/F)(G/H)(I/J).

To simply print out these LCSs would take exponential time.
5. (21 points) Restaurant location problem.

a. (15 points) Solve the following optimal restaurant location problem using dynamic programming. The distance constraint is that two selected locations cannot be within 10 miles.

Note that $d_i$ is the distance between location $i$ and location 1.

<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>Distance $d_i$</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>12</td>
<td>15</td>
<td>21</td>
<td>27</td>
<td>33</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>Profit $p_i$</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>select = no</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>20</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>select = yes</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>14</td>
<td>13</td>
<td>11</td>
<td>20</td>
<td>22</td>
<td>26</td>
<td>24</td>
</tr>
<tr>
<td>$S(i)$</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>20</td>
<td>22</td>
<td>26</td>
<td>26</td>
</tr>
</tbody>
</table>

Optimal value: 26

Locations (indices) selected in an optimal solution: 1, 4, 7, 9

b. (6 points) Now consider a special case of the restaurant location problem where all the candidate locations are evenly spaced. (For example, assume the distance between any two consecutive locations is exactly 5 miles.) Design a dynamic programming strategy to solve the problem in linear time. Write down your recurrence, and use it to solve the above problem after changing the distances so that $d_i = 0, 5, 10, 15, 20, ...$

The recurrence is simply

$$S(i) = \max \left\{ S(i - 1), S(i - 2) + p(i) \right\}$$
6. (12 points) Greedy algorithm.

a. (9 points) Use greedy algorithm to solve the fractional knapsack problem. The knapsack has a weight limit of 10 LBs.

<table>
<thead>
<tr>
<th>Item ID</th>
<th>Weight (LB)</th>
<th>Value ($)</th>
<th>Value / Weight ($/LB)</th>
<th>Weight (LB) taken</th>
<th>Value ($) taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>35</td>
<td>7</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>18</td>
<td>3.6</td>
<td>5</td>
<td>35</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>9</td>
<td>4.5</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>10</td>
<td>58</td>
</tr>
</tbody>
</table>

b. (3 points) If the objective of the problem is changed to filling the knapsack with the least (instead of most) total value, can greedy algorithm still be applied? Why or why not?

The answer is yes and the formal argument involves a proof by contradiction.

7. (5 points) Graph basics.

Draw a graph that can be represented by the adjacency matrix below.

```
A  B  C  D  E  F  G
A  0  1  0  0  0  0  0
B  0  0  0  0  0  0  0
C  0  0  0  1  0  0  0
D  0  0  1  0  0  1  0
E  0  0  0  0  1  0  0
F  0  1  0  0  0  0  1
G  0  0  0  1  1  1  0
```

![Graph Diagram]

---

6