CS 3343 (Spring 2015) Exam 2 Solution

1. (15 points) Sorting and Selection I

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Expected time complexity</th>
<th>Worst-case time complexity</th>
<th>Extra memory complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomized Quicksort</td>
<td>Θ(n log n)</td>
<td>Θ(n^2)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>Θ(n log n)</td>
<td>Θ(n log n)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>Θ(n + k)</td>
<td>Θ(n + k)</td>
<td>Θ(n + k)</td>
</tr>
<tr>
<td>Randomized Selection</td>
<td>Θ(n)</td>
<td>Θ(n^2)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Linear-time Selection</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
</tr>
</tbody>
</table>

b. When \( k \in \Theta(n) \), counting sort can be used to give a linear-time selection algorithm. If memory is a concern, then the randomized selection algorithm can be used, whose expected running time is linear in practice, even though its worst-case running time is quadratic.

c. When \( k \) is much larger than \( n \), counting sort loses its main advantage. The randomized selection algorithm should be preferred in practice, although the worst-case linear time selection algorithm has a theoretical advantage.

2. (10 points) Sorting and Selection II.

a. F. The best-case running time for Quick Sort is \( \Theta(n \log n) \).

b. T.

c. T. Finding median from an unsorted array takes at least \( \Theta(n) \). Finding all values \( \leq \) median given the median takes \( \Theta(n) \). So the first step, finding median, is the bottleneck and to return all values \( \leq \) median does not change the time complexity.

d. T. The main advantage of building a heap out an array instead of pre-sorting the array is heap’s efficiency in keeping the array in “partially” sorted order when dynamic updates are expected.

e. F. In theory, both finding median and finding mean can be done in \( \Theta(n) \) time, so asymptotically they are the same.

3. (25 points) Heaps

a. Yes it is a max heap. Child node’s value is never larger than parent node’s value.

b. \( A = [3 \ 8 \ 7 \ 9 \ 20 \ 15 \ 4] \Rightarrow [3 \ 8 \ 15 \ 9 \ 20 \ 7 \ 4] \Rightarrow [3 \ 20 \ 15 \ 9 \ 8 \ 7 \ 4] \Rightarrow [20 \ 9 \ 15 \ 3 \ 8 \ 7 \ 4] \)

c. No, it cannot. The precondition to execute heapify is that the two subtrees are already heaps. When building a heap top-down using heapify, there is no guarantee that the precondition holds.

d. No, there is no guarantee. For example \([3 \ 1 \ 2]\) is a max heap, but \([2 \ 1 \ 3]\) is not a min heap.

e. See figures on next page.
4. (18 points) Longest common subsequence (LCS).

   a. Details skipped.

   b. There are two LCSs (multiple paths), XCXED and YCXED.

5. (7 points) Fractional knapsack problem

<table>
<thead>
<tr>
<th>Item</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>90</td>
<td>30</td>
<td>.33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>60</td>
<td>1.2</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>C</td>
<td>40</td>
<td>20</td>
<td>.5</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>20</td>
<td>35</td>
<td>1.75</td>
<td>20</td>
<td>35</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>15</td>
<td>1.5</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>16</td>
<td>1.6</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Total</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>100</td>
<td>131</td>
</tr>
</tbody>
</table>

6. (15 points) Gas station location problem.

   a. The optimal value $F(i)$ can be computed by the following formula:

   $$F(i) = \max(F(i - 1), F(i - 2) + p(i)) \text{ for } i > 1.$$ 

   Initial condition: $F(0) = 0, F(1) = p(1)$. 


b. The optimal value is $290k and the optimal selection is 1, 4, 6, 9, 11, 13, and 15.

<table>
<thead>
<tr>
<th>Intersection</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit (× $10k)</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$F(i)$</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>16</td>
<td>17</td>
<td>19</td>
<td>19</td>
<td>23</td>
<td>23</td>
<td>28</td>
<td>28</td>
<td>29</td>
</tr>
</tbody>
</table>

c. The time complexity of the algorithm is $\Theta(n)$ for $n$ locations. The algorithm can be implemented by a simple for loop where each $F(i)$ is computed by comparing two values, $F(i-1)$, and $F(i-2) + p(i)$, which can be done in constant time for each $i$.

d. No, it is not necessary. As the estimation is only a constant ratio off, all possible solutions will be affected by the same ratio and therefore the optimal selection would still be the optimal selection, except that its value needs to be adjusted by the corresponding ratio.

e. Yes, you have to rerun the algorithm to get the optimal solution. While the estimation is off by a fixed constant, different solutions could have different number of locations selected and therefore the impact to different solutions could be different. For example, if we had only three locations, and the initial estimated profits were [1 2.5 1], the optimal solution would be selecting the second location with an optimal value 2.5. However, if the corrected estimates were [2 3.5 2], the optimal solution would be changed to selecting the first and third locations with an optimal value 4.

7. (5 points) Graph basics
   Skipped.

8. (20 points) Extra credit: Algorithm design.

   a. Solution: This task can be achieved by the following algorithm:

   (1) randomly choose a bolt.

   (2) Use the nut chosen in step 1 to partition the nuts into three groups: those that are smaller than the bolt (group 1), those that are larger than the bolt (group 2), and those that fit the bolt exactly (group 3).

   (3) Take a nut from the group 3 nuts obtained above in step 2, and use the nut to partition the bolts into three groups: those that are smaller than the nut (group 1), those that are larger than the nut (group 2), and those that fit the nut exactly (group 3).

   (4) Match each group 3 nut with a group 3 bolt.

   (5) Recursively apply steps 1-4 to match group 1 nuts with group 1 bolts, and group 2 nuts with group 2 bolts.

The analysis of the above algorithm follows exactly the analysis of randomized quick sort. The bolt chosen in step 1 can be considered as the pivot element. Step 2 and Step 3 take $\Theta(n)$ time to partition the bolts and nuts. Running time of Step 4 depends on how many bolts have equal sizes, and is in $O(n)$ nevertheless. Step 5 makes two recursive calls, similar to the two recursive calls in Quick Sort.

Thus, by choosing a random bolt (pivot) in step 1, the algorithm’s expected running time is $\Theta(n \log n)$. 

3
b. Let $S$ be a longest palindromic subsequence (LPS) of $X$. Then, we note that (1) $S$ is a sub-
sequence of $X$, and (2) $S$ is also a subsequence of the reverse of $X$. That is, $S$ must be a 
common subsequence of $X$ and reverse($X$). Therefore, to find the LPS of $X$, we can compute 
LCS($X$, reverse($X$)). However, if $X$ and reverse($X$) have several LCSs, then not all LCSs are 
palindromic. For example, if $X$ is TACTA, then LCS(TACTA, ATCAT) can be any of the 
four strings: ACA, TCT, TCA, ACT, only two of which are palindromic. The actual LPS 
can be found easily by taking half of a LCS and reverse it to get the other half.

It is also easy to develop a DP algorithm for this problem directly. Let $LPS(i, j)$ be the length 
of the longest palindromic subsequence for the substring $X[i..j]$. The recursion to compute 
$LPS(i, j)$ is as follows:

$$LPS(i,j) = \begin{cases} 
1 & \text{if } i = j; \\
LPS(i+1, j-1) + 1 & \text{if } x[i] = x[j] \text{ and } j > i; \\
\max\{LPS(i+1, j), LPS(i, j-1)\} & \text{if } x[i] \neq x[j] \text{ and } j > i 
\end{cases}$$

When computing the dynamic programming table, first initialize all diagonal entries to 1. 
Then work on the upper triangle towards the upper-right corner. The length of the LPS is 
stored in the entry $LPS(1, n)$.