NAME:

ID:

CSE 101 FINAL EXAM   Solutions

This final is a closed book, closed notes exam.
Do not open your exam until it is announced that you may do so.
Be prepared to show your id when handing in your final.
Good luck!

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/ 20</td>
</tr>
<tr>
<td>2</td>
<td>/ 20</td>
</tr>
<tr>
<td>3</td>
<td>/ 10</td>
</tr>
<tr>
<td>4</td>
<td>/ 10</td>
</tr>
<tr>
<td>5</td>
<td>/ 10</td>
</tr>
<tr>
<td>6</td>
<td>/ 10</td>
</tr>
<tr>
<td>7</td>
<td>/ 10</td>
</tr>
<tr>
<td>8</td>
<td>/ 10</td>
</tr>
<tr>
<td>Total</td>
<td>/ 100</td>
</tr>
</tbody>
</table>
Q1. State whether the following statements are True or False:
Each correct answer is worth 2 points while each wrong answer is worth –1 point. A blank answer is 0 points.

a) __F__ P is the set of problems solvable with a polynomial time algorithm. If A is in P and problem A reduces to problem B in polynomial time, the B is NP-complete.

b) __T__ There is an algorithm running in \( O(n \log(n^{1/2})) \) time for comparison based sorting.

c) ___T__ Assume \( P=NP \). Then the Independent Set problem can be solved in polynomial time.

d) __T__ If A is NP-hard and A is in NP, then A is NP-complete.

e) __T__ If \( f(n) = O(g(n)) \), then \( g(n) = \Omega(f(n)) \).

f) __F__ If \( f(n) = \Omega(g(n)) \), then \( g(n) = \Theta(f(n)) \).

g) __F__ If an NP problem can be solved in polynomial time then all NP problems can be solved in polynomial time as well.

h) __T__ If any NP-complete problem can be solved in polynomial time then all NP problems could be solved in polynomial time as well.

i) __F__ Every algorithm that runs in \( \Omega(n^2) \) time is always faster than another one that runs in \( O(n^3) \) time, assuming that the size \( n \) of the input is sufficiently large.

j) __F__ Let \( G \) be a weighted graph where each edge has a positive weight. Suppose \( P \) is a shortest weighted path between vertices \( r \) and \( s \) in \( G \). Construct a new weighted graph \( G' \) by adding a positive number \( k \) to each edge weight in \( G \). Then \( P \) always describes a shortest path from \( r \) to \( s \) in \( G' \).
Q2.

a) [5 pts] For the following graph, apply a *minimum spanning tree* algorithm, (Kruskal or Prim). Draw separate updated graphs for each step of the algorithm.

Solution:

The numbers in parenthesis specify the number of step that edge is picked.
b) [5 pts] For the following graph, apply \textit{Breadth First Search} (BFS) to compute the distances of the nodes to the source node, \( s \). Draw separate updated graphs for each step of the algorithm.

Solution:

\[ \begin{array}{c}
\text{Step 1: } \hspace{1cm} \text{Step 2: } \\
\text{Step 3: } \hspace{1cm} \text{Step 4: }
\end{array} \]
c) [5 pts] For the following graph, apply Depth First Search (DFS), starting from node u. Draw separate updated graphs for each step of the algorithm. Show the spanning tree you come up with after applying DFS.

Solution:

The numbers in paranthesis specify the number of step that edge is picked.
d) **[5 pts]** For the following graph, apply Dijkstra’s algorithm to find the shortest path from node s to all other nodes. Draw separate updated graphs for each step of the algorithm.

Solution:
Q3. [10 pts] Below are the definitions for the vertex cover and independent set problems:

**Vertex Cover problem**
Input: A graph \( G = (V, E) \) and integer \( k \leq |V| \)
Output: Is there a subset \( S \) of at most k vertices such that every \( e \) in \( E \) has at least one vertex in \( S \)?

**Independent Set problem**
Input: A graph \( G \) and an integer \( k \leq |V| \)
Output: Does there exist a set of \( k \) vertices in \( G \) such that no two of these vertices are connected by an edge?

Suppose you know that the vertex cover problem is NP-complete. Show that the independent set problem is NP-complete by reduction to the vertex cover problem.

Solution:
We get vertex cover is NP-hard by the fact that a vertex cover of a graph can be complemented to get an independent set in the graph, and vice versa. When I say complemented, I mean, all of the vertices in the independent set are not in the vertex cover. All of the vertices not in the independent set are in the vertex cover.

So we get as code,

\[
\text{Vertex-Cover}(G,k) \\
G' = G \\
k' = |V| - k \\
\text{return Independent-Set}(G', k')
\]

Now, we show that IS is in NP by noticing that it is easy to check if an IS solution. All we must do is go through the list of edges and see if any edge touches two vertices in the independent set. This takes time \( O(|E|) \) which is polynomial.

So, we are done.

Q4. Suppose Mergesort were modified such that it splits the list into 3 sections, and then merges the three lists, as follows:

3-way-Mergesort(A:array of integers) returns array of integers

/* Base cases to the sort */
if (size(A) = 1) then return A
if (size(A) = 2) then
    return A
else
    return A
end if
end if
/* If the list is size 3 or bigger, split the list up */
k = \lfloor\text{size}(A) / 3\rfloor
B = 3-way-Mergesort(A[1..k])
C = 3-way-Mergesort(A[k+1..2k])
D = 3-way-Mergesort(A[2k+1..\text{size}(A)])
/* Now merge the three lists, two at a time */
E = Merge(B, C)
F = Merge(E, D)
/* return the result */
return F

Suppose also that the merge operation, like that used in the normal mergesort, takes time linear to the inputs. Thus, if the merge operation takes lists of size m and n, the merge operation will run in time $O(m+n)$.

a) [5 pts] Inspect the code above and determine a recurrence relation for the algorithm. Be sure to show your work, and explain where you are getting each term from in the recurrence relation.

Solution:
Answer: $T(n) = O(1) + 3T(n/3) + O(2n/3) + O(3n/3) = 3T(n/3) + O(n)$
We get the 3 in front by the fact that the mergesort is called 3 times.
We get the 3 in the denominator by the fact that the mergesort is run on lists of size 1/3rd the original. The $O(n)$ at the end comes from the time it takes to perform the two merge operations ($5n/3$).
b) [5 pts] Solve the recurrence relation above to get the running time for the overall algorithm. Again, show your work (either by unrolling or by drawing the tree). A correct guess of the order of the algorithm can be worth fewer points than an incorrect solution with shown work.

Answer: \( T(n) = 3T(n/3) + O(n) = 9T(n/9) + 3O(n/3) + O(n) = O(n \log_3 n) = O(n \log n) \)
Q5. [10pts] Suppose you are given an array of n sorted numbers that has *been circularly shifted k positions to the right*. For example, \{35,42,5,17,27,29\} is a sorted array that has been circularly shifted k=2 positions to the right. Give an O(log n) algorithm to find the largest number in this array. Note that you don’t know what k is.

Solution:

\[
\text{CS}(l,u)
\]

\[
\text{If } l=u \text{ then return } A[u]
\]

\[
\text{If } u=l+1 \text{ then return } \max(A[u],A[l])
\]

\[
q=(l+u)/2
\]

\[
\text{If } A[q]<A[l] \text{ then } \text{CS}(l,q)
\]

\[
\text{Else if } A[q+1]>A[u] \text{ then } \text{CS}(q+1,u)
\]

\[
\text{Else return } \max(A[q],A[u])
\]

For this algorithm, since \(T(n)=T(n/2)+k\), we have \(T(n)=O(\log n)\)
Q6. [10 pts] Suppose you are given two words, a and b, consisting of only 0's and 1's. Give an algorithm to find the length of the longest common contiguous subsequence (LCCS) of a and b. You should write down the pseudo-code for this. What is the running time of your algorithm? Run your algorithm on a="01101001", b="001101010". (The result should be 6 since the LCCS of a and b is 011010.)

**Hint:** *Use dynamic Programming.*

**Solution:**
We are going to keep track of an array A, and A[i,j] represents the LCCS of the first i bits of a and the first j bits of b. n is the length of a, m is the length of b.

LCCS(a,b)
For i=1 to n
   A[i,1]=(a_i=b_1)
For j=1 to m
   A[1,j]=(a_1=b_j)
For i=2 to n
   For j=2 to m
      If a_i=b_j then A[i,j]=A[i-1,j-1]+1
      Else A[i,j]=0
      Search for max element in A, return it.

The running time of the algorithm is O(n.m). Let’s work on a="01101001" and b="001101010":

<table>
<thead>
<tr>
<th>a/b</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Q7.

a) [5 pts] What is the best possible lower bound for sorting n numbers just using comparisons? Prove that it is not possible to have any comparison based algorithm with better running time.

Solution:

The best possible lower bound for sorting n numbers by comparisons is $\Theta(n \log n)$.

Since there are $n!$ different permutations of $n$ elements, each comparison separates the set of permutations into two parts. So it takes $\log_2(n!) = \Theta(n \log n)$ comparisons at least to reach every permutation.

b) [5 pts] Suppose I developed a data structure in which I claim that I have Insert, Delete, and Extract-Max operations all running in $O(1)$ time. Extract-Max gives you the maximum element in the structure and then removes that element from the structure as well. Prove that developing such a data structure with operations above is impossible, based on your knowledge of lower bound for sorting.

Solution:

If the claim is true, we can sort by using Extract-Max and delete ($n$ times each). This means sorting can be done in $O(n)$ time which contradicts the fact that sorting has a lower bound $\Theta(n \log n)$. 
Q8. A graph is bipartite if the vertex set $V(G)$ of $G$ can be partitioned into two nonempty disjoint sets $A$ and $B$ such that every edge has an endpoint in $A$ and an endpoint in $B$.

(a) [3 pts] Explicitly write a bipartition $V(G) = A \cup B$ of the following graph that proves that the graph is bipartite.

(b) [7 pts] Given a connected graph, give an algorithm to test if $G$ is bipartite. Write a pseudocode for an algorithm that either gives a bipartition of the vertices $V(G)$ if $G$ is bipartite or else terminate and state “$G$ is not bipartite”. What is the running time of your algorithm?

Solution: (a) $A=\{a,c,h,e\}$, $B=\{b,g,d,f\}$

(b)

BCOLOR(G)
Start from an arbitrary node $u$.
Color $u$ blue
Traverse the graph by BFS
For each new vertex $v$ in the traversal,
   Color $v$ different from the color of its parent
Check each backedge of $v$, if a neighbor of $v$ has the same color as $v,
return “G is not bipartite”. 
When all vertices are colored, return “G is bipartite”, the set of vertices in blue and the set of vertices in red.

The running time is $O(n+m)$. 