Tutorial 7: Sorting and Graphs
Example Solutions

(1) (a) i. By the definition of the median and the way that partition works it is clear that the lower half has size ⌈n/2⌉ and the upper half has size ⌊n/2⌋.
ii. The recurrence here is simple:

\[ T(n) = \begin{cases} 
\Theta(1), & \text{if } n \leq 1; \\
T(⌈n/2⌉) + T(⌊n/2⌋) + O(n), & \text{if } n \geq 2. 
\end{cases} \]

iii. Here \( M(n) + \Theta(1) = \Theta(n) \). So in the Master Theorem the critical exponent \( \epsilon = \log_2(2) = 1 \). Thus the middle case applies and we have \( T(n) = \Theta(n \log n) \).
iv. See the chapter on on Medians and Order Statistics in [CLRS]. The algorithm is not particularly complicated (around 2 pages for a careful description and analysis including a diagram).

(2) The following algorithm \texttt{sortFour} does the job (\texttt{mergeSort} also sorts 4 numbers with 5 comparisons).

```plaintext
sortFour(A)
   (s₁, t₁) ← comp(A[0], A[1])
   (s₂, t₂) ← comp(A[2], A[3])
   (s₃, t₃) ← comp(s₁, s₂)
   (s₄, t₄) ← comp(t₁, t₂)
   return t₁, t₂, s₃, s₄
```

We can discover this algorithm as follows. We display elements and their relationship (in terms of relative order) at each stage. An arrow from \( a \) to \( b \) means that we now know \( a > b \). Initially we know nothing about the ordering of the given elements.

\[
\begin{array}{c|c|c|c}
\hline
\end{array}
\]

Compare \( A[0] \) with \( A[1] \)

\[
\begin{array}{c|c|c}
l₁ & l₂ \\
\hline
\hline
\end{array}
\]


\[
\begin{array}{c|c|c}
l₂ & l₂ \\
\hline
s₁ & s₂ \\
\hline
\end{array}
\]

We have now finished.

This is best possible: 5 comparisons is the smallest number for the general case of sorting 4 elements. To see why this is the case, we use the fact that for general inputs of 4 numbers, there are 4! possible different rearrangements of those numbers which correspond to possible outputs (because there are 4! permutations of any 4 items). We can view sorting algorithms as binary decision trees, where vertices are comparisons. As the algorithm is executed on an input we follow a path from the root of the tree to a leaf (so the maximum number of comparisons made is the height of the tree). Each comparison leads us to one of two possibilities. By the time we get to a leaf of the tree (all comparisons have been made for this input) the ordering is fixed. Thus there must be at least as many leaves as there are possible orderings of the input. Since a binary tree of height \( h \) has at most \( 2^h \) leaves we must have \( h \geq \log(24) > 4 \).

This reasoning can be generalised to the case of \( n \) items. We see that we must use at least \( \log(n!) \) comparisons. Since \( (n/2)^{n/2} < n! \leq n^n \) we deduce that \( \Omega(n \log n) \) comparisons are necessary.
A single edge is a bipartite graph and a triangle is not.

We use a search of the graph (e.g., BFS) with the adjacency list representation. We maintain a variable $V$ that alternates between 1 and 2 (to denote $V_1$ or $V_2$). Initialise the variable to 1 (or 2, it doesn’t matter). Each time we visit a vertex we first give $V$ the alternative value (i.e., alternate between 1 and 2). When we visit a vertex for the first time we mark it with the vertex set denoted by $V$. When we revisit a vertex we check if the set indicated by $V$ is consistent with the one recorded. If it is carry on otherwise report that the graph is not bipartite.

Since we do only a constant amount of work at each vertex the time is asymptotically the same as for a straight search, i.e., $\Theta(n + m)$.

Using DFS or BFS has a worst case runtime of $\Theta(m + n)$ where $m$ is the number of edges of the tree and $n$ is the number of vertices. Since we have $m = n - 1$ it follows that the runtime is $\Theta(2n - 1) = \Theta(n)$. But using binary search the runtime is in fact $\Theta(\log n)$ which is exponentially faster.

The answer displays no understanding at all of the purpose of binary search trees and the overwhelming advantage they give us in searching over naive methods such as linear search. Mentioning DFS or BFS might make it look technically informed but in fact the method is just linear search with unnecessary extra overheads [why?]; thus it is in fact technically inept.

We can prove the claim about the number of edges very simply. If $n = 1$ then the tree has just the root and no edges, so the base case is correct. For the induction step, we have $n > 1$ and following any path from the root towards the leaves we will eventually come to a vertex without any internal vertices as children (i.e., both children are leaves). Deleting this vertex and the edge that led us to it gives us a tree with $n - 1 \geq 1$ vertices [since $n > 1$]. By the induction hypothesis this tree has $n - 1 - 1 = n - 2$ edges. Thus the original tree with $n$ vertices has $n - 2 + 1 = n - 1$ edges. This proves the induction step.

An alternative approach to the induction step is to note that if we delete any edge we get two smaller trees with $n_1$ and $n_2$ vertices where $n_1 + n_2 = n$. Note that $n_1 \geq 1$ and $n_2 \geq 1$ since the two endpoint of the edge lie in different subtrees. Applying the induction hypothesis to the subtrees we deduce that they have $n_1 - 1$ and $n_2 - 1$ edges respectively. Thus the original tree has $n - 1 + n_2 - 1 - 1 = n_1 + n_2 - 2 = n - 1$ edges.

The induction step for trees often focuses on the root of the tree and considers the subtrees [two in the case of binary trees] in order to apply the induction hypothesis to them. However we have to be careful here because one subtree might be a leaf and we have excluded these from consideration. So we are lead to two possible cases: (i) neither subtree is a leaf and (ii) one subtree is a leaf [they cannot both be leaves [why?]!]. The induction step goes through easily enough in either case.

An alternative approach is to allow leaves and change our statement to: a tree with $n$ internal vertices has $\max\{0, n - 1\}$ edges. Yet another approach is to allow edges joining leaves to internal vertices [so this reflects the diagrams we often draw that show leaves explicitly] and modify the statement to: a tree with $n$ internal vertices has exactly $n - 1$ edges both of whose endpoints are internal vertices. To some extent which approach we take is a matter of taste, though the main one taken leads to what is arguably the simplest statement and proof.