Solutions
1. For each question below, give a short answer - a few words or symbols, maybe a sentence or two.

(a) (4 points) Give an example of an algorithm which has average time of $O(n^2)$ but is not $\Theta(n^2)$. Explain what sort of special situations would lead to the better runtime.

Solution: Answers may vary. My suggested example is: Insertion Sort runs in $O(n)$ time if the array is already sorted, because, for each element, we look at it, and do not move it.

Another plausible answer: In theory, any sorting algorithm can run in $O(n)$ time, simply by adding a scan through the data at the beginning; if the data is already sorted, then skip all the rest of the work.

(b) (5 points) Give an example of an algorithm which run (on average) in $O(n \lg n)$ time, but which will occasionally do worse.

- Name the sort
- Give the true worst-case runtime
- Explain why it sometimes has worse than $O(n \lg n)$ time; how does this happen?

Solution: Quick Sort
Worst case time: $O(n^2)$
Occasionally, the pivot that is chosen will be very poor, and so almost nothing is in one of the two partitions. If this happens many times in the same sort, then the cost will be $O(n^2)$.

(c) (2 points) Identify two sorting algorithms which are known to have worst case running times of $O(n \lg n)$. (You don’t have to justify why these algorithms run that quickly.)

Solution: Heap Sort, Merge Sort

(d) (2 points) Give an example of a sort algorithm which is a stable sort.

Solution: Give one of the below options, from either list:
$O(n \lg n)$ sorts: Merge Sort
$O(n^2)$ sorts: Insertion Sort, Bubble Sort

Give an example of a sort algorithm which is a not a stable sort.

Solution: Give one of the below options, from either list:
$O(n \lg n)$ sorts: Quicksort, Heap Sort
$O(n^2)$ sorts: Selection Sort

(e) (2 points) Explain why no sorting algorithm can have performance better than $O(n)$.

Solution: The algorithm must read every element once, just to see if the array is already sorted!
2. (a) (10 points) Give the formal definition for $O(g(n))$.

**Solution:**

$$O(g(n)) = \left\{ f(n) : \exists c > 0, n_0 > 0 \quad \forall n > n_0 \quad 0 \leq f(n) \leq cg(n) \right\}$$

(b) (5 points) We think of a heap as a binary tree. How do we actually store it in memory? What is special about a heap which makes this possible?

**Solution:** It doesn’t have any holes! That is, all of the layers are full except for the last; and the last layer always fill from left to right.

(c) (5 points) Explain the difference between keys and satellite data (also known as “values”) in a sort algorithm.

**Solution:** The key is used to sort the data. The satellite data simply “comes along” - it is part of the thing being sorted, but ignored for the sort.
3. (a) (5 points) Suppose that you have a predicate \( P(x, y) \). Write formal quantifiers to express the following English language statements. If negations are required, you are not required to simplify those negations; just write a correct expression.

“\( P(x, y) \) always returns true, no matter what parameters you give it.”

Solution: \( \forall x \forall y P(x, y) \)

“At least one combination of \((x, y)\) will be rejected by \( P \).”

Solution: \( \neg \forall x \forall y P(x, y) \)
- or -
\( \exists x \exists y \neg P(x, y) \)

(b) (5 points) Again, convert English to a formal expression; this time, the quantifier is \( Q(x) \).

For full credit, convert this to a mathematical expression (an English language explanation is not required).

For half credit, break this expression down into simpler expressions, but still express them in English - and skip the mathematical formula.

“\( Q(x) \) is true for only one \( x \).”

Solution: Conceptual Answer:
\( Q(x) \) is true for some \( x \). And also, if any two values \( x, y \) are chosen such that \( Q \) is true for both, then \( x = y \).

Complete Answer:
\( \exists x Q(x) \land \forall x \forall y ((Q(x) \land Q(y)) \implies (x = y)) \)

4. (10 points) On the next page, I’ve given an implementation for the \texttt{merge()} step of Merge Sort. I have inserted quite a few bugs. Edit the code to make the code work correctly.

The parameters are:
- The array of input values
- The index of the start, midpoint, and end (\texttt{beg,mid,end})

The first sub-array to merge goes from \texttt{beg} (inclusive) to \texttt{mid} (exclusive). The second sub-array to merge goes from \texttt{mid} (inclusive) to \texttt{end} (exclusive). (Don’t assume that the two sub-arrays are the same length!)

Place the merged data back into the original array; you may allocate a temporary buffer.

You may assume that all of the parameters are valid: the array is not \texttt{null}, and the indices are all non-negative, and the indices obey the assumption \( \texttt{beg} \leq \texttt{mid} \leq \texttt{end} \leq \texttt{vals.length} \).
void merge(int[] vals, int beg, int mid, int end)
{
    int[] tmp = new int[vals.length];
    int out=0, L=0, R=0;

    while (L < mid || R < end)
    {
        if (vals[L] < vals[R])
        {
            tmp[out] = vals[L];
            L++;
            out++;
        }
        else
        {
            tmp[out] = vals[R];
            R++;
        }
        out++;
    }

    while (L < mid)
    {
        tmp[out] = vals[L];
        L++;
        out++;
    }

    while (R < end)
    {
        tmp[out] = vals[R];
        R++;
        out++;
    }

    for (int i=0; i<tmp.length; i++)
    {
        vals[beg+i] = tmp[i];
    }
}
Solution:

```java
void merge(int[] vals, int beg, int mid, int end)
{
    int[] tmp = new int[vals.length]; // FIX: temporary buffer is too long!
    // ideal length is [end-beg]
    int out=0, L=0, R=0; // FIX: either set L=beg, R=mid, or
    // else change how the bounds
    // are calculated below.

    while (L < mid || R < end) // FIX: &&
    {
        if (vals[L] < vals[R]) // FIX: less-or-equals, or it's not stable!
        {
            tmp[out] = vals[L];
            L++;
            out++;
            // FIX: Double-increment of out!
        }
        else
        {
            tmp[out] = vals[R];
            R++;
        }
        out++;
    }
    while (L < mid)
    {
        tmp[out] = vals[L];
        L++;
        out++;
    }
    while (R < end)
    {
        tmp[out] = vals[R];
        R++;
        out++;
    }
    for (int i=0; i<tmp.length; i++)
    {
        vals[beg+i] = tmp[i];
    }
}
```
5. (15 points) Use the Master Method to solve the following recurrences. If the recurrence cannot be solved by the Master Method, give a short explanation why not.

(a) \( T(n) = 8T\left(\frac{n}{3}\right) + n^2 \)

**Solution:**
\[
\begin{align*}
a &= 8 \\
b &= 3 \\
\log_b a &= \log_3 8 \\
\end{align*}
\]
\[
\text{(this is something less than 2)}
\]
\[
f(n) = n^2
\]
This is Case 3, because \( f(n) = \Omega(n^{\log_3 8 + \epsilon}) \).
\[
T(n) = \Theta(n^2)
\]

(b) \( T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n} \)

**Solution:**
\[
\begin{align*}
a &= 2 \\
b &= 4 \\
\log_b a &= \log_4 2 = \frac{1}{2} \\
\end{align*}
\]
\[
f(n) = \sqrt{n}
\]
This is Case 2, because \( f(n) = \Theta(n^{1/2}) \).
\[
T(n) = \Theta(\sqrt{n} \lg n)
\]

(c) \( T(n) = 2T(n - 100) + \lg n \)

**Solution:** This cannot be solved by the Master Method, because it’s not in the standard form (there is subtraction in the recursive call).

(d) \( T(n) = 27T\left(\frac{n}{3}\right) + n^2 \)

**Solution:**
\[
\begin{align*}
a &= 27 \\
b &= 3 \\
\log_b a &= \log_3 27 = 3 \\
\end{align*}
\]
\[
f(n) = n^2
\]
This is Case 1, because \( f(n) = O(n^{3-\epsilon}) \).
\[
T(n) = \Theta(n^3)
\]
Choose only one of the induction problems to do. Do not do both - if you do, we will only grade one of them!

6. (20 points) Consider a (non-empty) trinary tree, with a special restriction: every internal node has exactly three children (never less).

Using structural induction, prove that the number of leaves is always odd.
(You may use any of the structural induction strategies.)

HINT: Trees of this form can only have certain numbers of nodes: 1,4,7,10...

Solution:

**Base: A single node**

Clearly, a tree with a single node has a single leaf. Thus, the base case holds.

**Inductive: (version 1: Add new leaves at the bottom)**

Assume that the conjecture holds for all conceivable trees of this form, which have $< k$ nodes. We will prove that it also holds for trees with $k$ nodes.

A tree (of this special form) with $k$ nodes is a smaller tree, with $k - 3$ nodes, plus three new nodes, added below one of the old leaves. Obviously, that old leaf becomes an internal node instead.

By the I.H., we know that the smaller tree had an odd number of nodes. We added $3 - 1 = 2$ nodes; thus, the new tree has an odd number of nodes as well.

Thus, the inductive step holds.

**Inductive: (version 2: Root+subtrees)**

Assume that the conjecture holds for all conceivable trees of this form, which have $< k$ nodes. We will prove that it also holds for trees with $k$ nodes.

A tree (of this special form) with $k$ nodes is a root node, plus three subtrees. (We know that it has three subtrees because, by definition, all non-leaf nodes have exactly three children.)

The children might be any size (even single nodes), and are not necessarily all the same size; however, by the I.H., we know that each of them has an odd number of leaves.

The root is clearly not a leaf. Thus, the total number of leaves is $\text{odd+odd+odd}$, and thus odd.

Thus, the inductive step holds.
Choose only one of the induction problems to do. Do not do both - if you do, we will only grade one of them!

Using induction, prove the following conjecture:
\[ \sum_{i=0}^{n-1} ar^i = \frac{a(1 - r^n)}{1 - r}, \text{ where } n \in \mathbb{Z}^+, r \neq 1. \]

Solution:
Base: \( n = 1 \)

\[
\sum_{i=0}^{0} ar^i = \frac{a(1 - r^1)}{1 - r} = a
\]

Thus, the base case holds.

Inductive:
Assume that the conjecture holds for all \( n \leq k \). We will try to prove that it holds for \( n = k + 1 \).

\[
\sum_{i=0}^{k+1-1} ar^i = \sum_{i=0}^{k} ar^i + \sum_{i=0}^{k-1} ar^i
\]

By the I.H.:

\[
\frac{a(1 - r^k)}{1 - r} + \frac{a(1 - r^k)}{1 - r} \quad \frac{a - ar^{k+1}}{1 - r} + \frac{a(1 - r^{k+1})}{1 - r}
\]

Thus, the inductive step holds.
7. (a) (5 points) Perform Radix Sort on the data below; show the complete contents of the array after every pass. As with Project 2, use decimal digits as the “columns” of each key.

82
962
770
962
465
941
769
727

Solution:

770 727 82
941 941 465
82 962 727
962 962 769
962 465 770
465 769 941
727 770 962
769 82 962

(b) (5 points) Use the median-of-3 algorithm to choose a pivot for each of the following arrays. Circle the value that you choose as the pivot.

Solution: Pivots:

300 (middle)
89 (front)
61 (back)
22 (front)
68 (middle)

[149, 741, 40, 300, 1, 941, 402]