1. (a) A correct answer is \(3^n = \Omega(n^{\sqrt{n}})\). However, a better answer would be \(3^n = \omega(n^{\sqrt{n}})\). There are many ways to arrive at this. For example, we can write \(n^{\sqrt{n}}\) as \(3^{\log_3(n^{\sqrt{n}})} = 3^{\sqrt{n}\log_3 n}\). Then,

\[
\lim_{n \to \infty} \frac{3^n}{n^{\sqrt{n}}} = \lim_{n \to \infty} \frac{3^n}{3^{\sqrt{n}\log_3 n}} = \lim_{n \to \infty} 3^{n - \sqrt{n}\log_3 n} = \infty
\]

where the last equality holds because \(n\) is asymptotically strictly larger than \(\sqrt{n}\log_3 n\) (i.e. \(n = \omega(\sqrt{n}\log_3 n)\): Verify this!)

(b) The goal here was clarity of presentation. For example, let us describe the sorting problem (taken from CLR):

**Input:** A sequence of \(n\) numbers \((a_1, \ldots, a_n)\).

**Output:** A permutation (reordering) \(\langle a'_1, \ldots, a'_n \rangle\) of the input sequence such that \(a'_1 \leq a'_2 \leq \cdots \leq a'_n\).

**Complexity:** An algorithm like Heapsort takes \(O(n \log n)\) time and \(O(n)\) space to sort.

(c) We can replace each edge weight by its log. Since all the edge weights are \(> 1\), all the logs are positive. Therefore, we can run our usual Dijkstra algorithm on the modified graph to find the path of smallest weight from source \(s\) to each of the other vertices. (Note: This is an example of a reduction.)

2. (a) Let the running time of the sorting algorithm be \(T(N)\). The recurrence is

\[
T(N) = \sqrt{N}T(\sqrt{N}) + O(N)
\]

There are several ways to solve this recurrence. Let’s divide each term in the recurrence by \(N\). Then we get

\[
\frac{T(N)}{N} = \frac{T(\sqrt{N})}{\sqrt{N}} + O(1)
\]

Substitute \(N = 2^m\).

\[
\frac{T(2^m)}{2^m} = \frac{T(2^{m/2})}{2^{m/2}} + O(1)
\]

Rename \(T(2^m)/2^m\) as \(S(m)\).

\[
S(m) = S(m/2) + O(1)
\]

This is the recurrence for binary search. So we know that \(S(m) = O(\log m)\). Backsubstituting,

\[
\frac{T(2^m)}{2^m} = O(\log m) \Rightarrow \frac{T(N)}{N} = O(\log \log N) \Rightarrow T(N) = O(N \log \log N)
\]

(b) Let \(f(N)\) denote the number of comparisons to merge \(\sqrt{N}\) sorted array of size \(\sqrt{N}\). In Part (a) we have already seen that when \(f(N) = O(N)\), the complexity of our sort becomes \(T(N) = O(N \log \log N)\). But since this is a comparison based sort, we have a lower bound on \(T(N)\), i.e. \(T(N) = \Omega(N \log N)\). Therefore, \(f(N) = \omega(N)\). Let us try a larger function, say \(f(N) = \Omega(N \log N)\). Then, redoing the recurrence as in part (a), we get

\[
\frac{T(N)}{N} = \frac{T(\sqrt{N})}{\sqrt{N}} + \Omega(\log N)
\]

\[
S(m) = S(m/2) + \Omega(m)
\]

\[
S(m) = \sum_{i=1}^{\log m} \Omega \left( \frac{m}{2^i} \right) = \Omega \left( m \sum_{i=1}^{\log m} \left( \frac{1}{2} \right)^i \right) = \Omega(m)
\]

which implies that \(T(N) = \Omega(N \log N)\), thus confirming our guess that a good lower bound for \(f\) is \(f(N) = \Omega(N \log N)\).
3. We can store a copy of the tree $T$, say $T'$, with some changes. Each node will have a pointer to its parent. This can be done in linear time, because when we visit a node $u$ in the tree during preprocessing, we can initialize all the parent pointers in its child nodes to $u$. Also, the children can be stored in order of rank in an array in the parent’s node in $T'$. Thus, both types of queries, Parent($u$) and Ranked-Child($k$, $u$) can be answered in constant time.

4. We are given as input a string $T$. We can store a copy of the tree $A$, and start filling the end of the array $A$, whose elements are initially set to 0. For each center $i$, do the following: Initialize $j = C[i]$. Till $j > 0$, do $A[i - j + 1] = j$ and $j = j - 1$. You can verify the fact that each entry in $A$ is written at most once. Therefore, the time complexity of step 2 is $O(N \log N)$ time. The time complexity of the algorithm is dominated by step 1, therefore it takes $O(N \log N)$ time.

5. (a) Given an interval $I = [i, j]$, let us define $f(x)$ as

$$f(x) = \sum_{k=i}^{j} (A[k] - x)^2$$

From Calculus, we know that since the function $f$ is minimum at $x^*_I$, $f'(x^*_I) = 0$. Thus,

$$\sum_{k=i}^{j} -2(A[k] - x^*_I) = 0 \Rightarrow x^*_I = \frac{\sum_{k=i}^{j} A[k]}{(j - i + 1)}$$

It takes $\Theta(|I|) = \Theta(j - i)$ time to compute $x^*_I$.

(b) Let $m(i, l)$ denote the minimum sum of $x^*_i$s over $l$ nonoverlapping intervals $I$ when the array is $A[i..N]$. Thus, our objective is to find $m(1, L)$. The function $m$ has the following recursive definition:

$$m(i, l) = \begin{cases} x^*_i[N] & \text{if } l = 1 \\ \min_{i \leq j \leq N - l + 1} \left\{ x^*_i[j] + m(j + 1, l - 1) \right\} & \text{otherwise} \end{cases}$$

This enables us to use Dynamic Programming to solve our problem. We need to store all possible values of $m(i, l)$. Therefore, the space required is $O(NL)$. The time complexity of the algorithm is $O(N^3L)$, since calculating each $m(i, L)$ potentially requires $O(N^2)$ time due to the time overhead to calculate $x^*_i[j]$. (We could save on time by a factor of $N$ by using additional $O(N)$ space to store the some sums of the arrays so that the $x^*_i[j]$s can be calculated in constant time.)