Instructions for Q1 through Q4
Assume that $f(n)$ and $g(n)$ are positive-valued functions defined on the set of non-negative integers.

Q1. [8 points] What does the fact given below imply regarding any big-O, big-Ω and/or big-Θ relationships between the functions? Be complete and precise. No justification is needed.

$$\forall n \geq 1000,\ f(n) \geq 7g(n)$$

Answer:

The given fact can be rewritten as: $\forall n \geq 1000,\ g(n) \leq \frac{1}{7} f(n)$

And, by definition, this implies that g is O(f).

Q2. [8 points] What does the fact given below imply regarding any big-O, big-Ω and/or big-Θ relationships between the functions? Be complete and precise. No justification is needed.

$$\forall n \leq 1000000,\ g(n) \geq 7 f(n)$$

Answer:
This implies nothing of interest because the bound on is incorrect for any relevant asymptotic definitions.

Q3. [8 points] What does the fact given below imply regarding any big-O, big-Ω and/or big-Θ relationships between the functions? Be complete and precise. No justification is needed.

$$\forall n \geq 10,\ f(n) = 7g(n)$$

Answer:
The equality relationship implies both that f(n) <= 7g(n) and that 7g(n) <= f(n). Therefore, we have that f is O(g) and that f is Ω(g), and hence that f is Θ(g).

Q4. [8 points] What does the fact given below imply regarding any big-O, big-Ω and/or big-Θ relationships between the functions? Be complete and precise. No justification is needed.

$$\forall n \geq 42,\ 3g(n) \leq f(n) \leq 5g(n)$$

Answer:
Directly from the definition, this implies that f is Θ(g).

Q5. **[16 points]** The following algorithm, known as Horner's Rule, is used to evaluate a polynomial $f(x) = \sum_{i=o}^{N} a_i x^i$.

Assume that the coefficients of the polynomial are stored in an array `a[]` and the value for x is stored in a variable `x`.

```
Poly = 0;                          // 1

for (i=N; i>=0; i--)               // 1 before, 2 per pass, 1 to exit

   Poly = x * Poly + a[i];         // 3 for each pass of the loop
```

Derive a simplified complexity function T(N) for the given algorithm. Show all supporting work.

Answer:
From the notes above:

$$T(N) = 1 + 1 + \sum_{i=0}^{N}(2+3) + 1 = \sum_{i=0}^{N}(5) + 3 = 5(N+1) + 3 = 5N + 8$$

Q6. **[18 points]** Suppose you need to organize a large collection of data objects for which there is a `compareTo()` method, so that the objects can be compared for storage in any of the data structures we've discussed (except the PR quadtree). Suppose also that the key field is the name of a geographic feature, like "Portage des Sioux". There are many records, and you have no idea what names might be included.

And, suppose that you expect that the most common operation on these data objects will be to perform a range search; that is, a search like "find all data objects where the key field is between Pumpkin Center and Truth or Consequences".

Suppose that you have access to generic implementations of a splay tree, a hash table, and a B-tree (or a variant B-tree if you like). To make your most common operation efficient, which of these would be the best choice for organizing these data objects? Explain why.

Answer:
First, note that we have NO useful idea for a set of "candidate" strings that fall between the two given feature names (or any others). So, it's not possible to approach this by searching, one by one, across all possible answers.

A hash table would require that we simply do a linear pass through the table's array, which is hardly using the hash table as a hash table. The cost of that would be O(# table slots).

A splay tree is a BST, so we could perform a range search on it; we just need to take the usual BST search algorithm and limit our traversal to eliminate subtrees that could not possibly contain a suitable element. That can be shown to be O(# nodes in tree). BUT, if the splay tree were to rearrange itself after each "hit", that would probably render our algorithm useless.

A B+ tree would allow us to search for the lower bound for the range, and either find that or find the smallest successor to it in the tree, and then walk the leaf level until we found the first element that's too large.

Q7. [18 points] Suppose you are developing a GIS application, and you are using a PR quadtree to index records stored in a GIS database file, as described in the projects in this course. You may have noticed that the records in our GIS files tend to be stored in a fair approximation of alphabetical order, by feature name.

Of course, the PR quadtree allows you to obtain the file offsets of all the GIS records that match a given pair of longitude/latitude coordinates. Then, you can seek to each of those file offsets and retrieve the matching records from the GIS database file. So far, so good. But the GIS records are stored in a file on disk, and we know that disk accesses are extremely slow, compared to DRAM accesses. So, your implementation will include a buffer pool, and we expect that determining whether a record is in the buffer pool will be 10,000 times faster than retrieving a record from the file.

Your partner, Haskell Hoo IV, argues that users are likely, but not guaranteed, to be interested in features that are nearby features they've already asked about. That does seem to be a reasonable prediction, and we will accept that Haskell is correct about this.

Based on that idea, Haskell suggests you can take advantage of spatial locality in user searches. Haskell says that, when you are given a pair of coordinates for a search, you should actually identify all the records that fall within a small, fixed distance of those coordinates, by using the range search feature of the location index to get their file offsets, and then retrieve all those records and store those records in the buffer pool, so they will be available if a user does ask about nearby features.

Critique Haskell's suggestion. Assuming that users do tend to perform searches for nearby features, will his idea tend to improve performance? Explain why or why not.

Answer:

The term "spatial locality" refers to searches for records that are near to each other in the file on disk; the situation we have here does not fit that at all. There is no reason to think that geographic features that are physically close will correspond to GIS records that will be physically close in the file (since the entries in the file are somewhat in alphabetic order).

With spatial locality, we hope to take advantage of the fact that, at least with traditional HD storage, reading a few thousand bytes costs only a tiny bit more time that reading just a few dozen bytes. So, grabbing a larger chunk, holding several records would be more efficient that retrieving those records one by one.

Basically, Haskell has no understanding of the term spatial locality. If we adopted his suggestion:

- **We'd go to the location index with a longitude/latitude pair, and get back a collection of file offsets.**
- **We'd then have to retrieve those records from the file.**
- **Since the relevant records would probably not be at all close to one another in the file, we'd have to do a separate seek and read for each record. And that does not buy us improved performance.**

As far as the time to retrieve records from the file goes, we might as well just wait and retrieve each record when it's actually requested. That way, we won't waste time retrieving records that are never needed, and, worse, having them bump records that would be needed from the buffer pool.

Most answers missed the point completely… which goes to show how good I am at judging whether a question is easy or not.

Common issues with answers included:

- **The vast majority of answers totally missed the meaning of spatial locality.**
- **Most answers actually concluded Haskell had a good idea.**
- **Some answers may have made some valid points about the potential Haskell's approach would evict useful records from the buffer pool in favor of records that are likely to be irrelevant.**

Q8. [16 points] If we have a BST storing unique integer values, the smallest value will be in the left-most node in the tree; that is, the node reached by starting at the root and taking left-child pointers until there is no left child to go to.

Suppose we have a max-heap storing unique integer values. What is the most precise statement that can be made about the location of the smallest value in the max-heap? Justify your answer.

Answer:
The smallest value must be in a leaf, since in a max-heap the value in each parent node must be larger than the values in its children.

But, it is, in general, to specify exactly where that leaf might be. There is no ordering of the children of an internal node. We can say the leaf must either be in the bottom level, or the level immediately above that, if the bottom level is not completely full.

Common issues with answers included:

- **Only saying the smallest element must be in a leaf, but not that it could, potentially, be in any of a number of positions.**
- **Saying the smallest element had to be somewhere in the last two levels without actually specifying it must be in a leaf; that leaves open the possibility it could be in an internal node in the next to last level.**