READ THIS NOW!

- Print your name in the space provided below.
- Unless a question involves Java code and determining whether that code is syntactically correct, assume that it is, and that any necessary `imports` have been made.
- There are 8 short-answer questions, priced as marked. The maximum score is 100.
- When you have finished, sign the pledge at the bottom of this page and turn in the test and your fact sheet.
- Aside from the allowed one-page fact sheet, this is a closed-book, closed-notes examination.
- No laptops, calculators, cell phones or other electronic devices may be used during this examination.
- Until solutions are posted, you may not discuss this examination with any student who has not taken it.
- Failure to adhere to any of these restrictions is an Honor Code violation.

Name (Last, First) _____

`printed`

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

_____

signed

1. Assume that $f(n)$ and $g(n)$ are positive-valued functions defined on the set of non-negative integers.

a) [6 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 justifications are needed.

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

By definition, this implies that f is $\Omega(g)$. If you divide through by 7, you get

$\forall n \geq 1000, g(n) \leq \frac{1}{7} g(n)$, **and so** g is $O(f)$.

b) [6 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 justifications are needed.

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

The first part of inequality implies that f is $\Omega(g)$ and the second implies that f is $O(g)$.

Therefore, we have that f is $\Theta(g)$. Since xx is an equivalence relation, we also have that g is $\Theta(f)$, and hence that g is $\Omega(f)$ and g is $O(f)$.

---

2. [9 points] For each part, state the simplest function $g(n)$ such that the given function is $\Theta(g)$; you should use theorems from the course notes; no justification is required

a) $a(n) = 14n + 3\log n$ a is $\Theta(n)$

b) $b(n) = 10n^{100} + 2^n$ b is $\Theta(2^n)$

c) $c(n) = 4\log(n^3) + \log^3(n)$ c is $\Theta(\log^3 n)$

3. [15 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];      // 4 per pass
```

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

$$T(N) = 1 + 1 + \sum_{i=0}^{N} (2+4) + 1 = \sum_{i=0}^{N} 6 + 3 = 6(N+1) + 3 = 6N + 9$$

(If you assumed i- - was 2 operations, you should have gotten 7N+10.)

4. [12 points] A developer wants to use a hash table to organize a collection of 1,000,000 data objects, and the key values are strings of the following form:

αβγijkl

where α, β, and γ are lower-case letters [a-z] and i, j, k and l are digits [0-9]. Thus there are 175,760,000 different possible key values.

The developer will use a table of size 2,000,000 and some form of probing to resolve collisions. Since there is no way for the developer to know exactly what key values will actually occur in the data objects, she designs a hash function that is highly uniform; that is, if all 175,760,000 key values were hashed into the table, roughly the same number of keys would hash to each table slot (about 88).

If the chosen hash function were perfectly uniform on the set of actual key values, we would have no collisions at all. That is unlikely to be the case. But if the chosen hash function is even reasonably uniform on the actual key values, we might expect to have only a small number of collisions, and to not have very many collisions in any particular table slot.

Explain why it is still possible, even if the hash function has the properties described above, that there will be table slots that receive a large number of collisions. Your explanation should use specific (but hypothetical) examples of how this could occur.

Even if the function distributes the set of all possible key values uniformly across the table, if the subset of actual keys is not distributed uniformly through the set of possible keys then the resulting hash mapping may be skewed within the table.

For example, suppose that the hash function concatenates the 7-bit ASCII codes of the three characters with the binary representation of the 4-digit number represented by the last part of each key. This will yield a sequence of 38 bits and will be uniformly distributed (if we leave off leading 0s in the representation of the last part).

But, suppose that the actual key values always end with two zeros. Then when we use this hash function, and mod by the table size to get a table index, we will never put a record into a slot whose index does not end with two zeros. That's far from uniform.

(The most common issue here was in not giving a sufficiently detailed, specific example of how this could occur.)

5. [12 points] State the primary theoretical shortcoming of using pure quadratic probing (i.e., k^2) to resolve collisions in a hash table, and explain why this shortcoming may be considered to be insignificant in practice.

Pure quadratic probing will not usually examine all the slots in the hash table before entering a repeating pattern. So, even though there are empty slots in the table, this scheme may not find one of them.

On the other hand, pure quadratic probing will generally examine a healthy subset of the table slots, and we can guarantee it examines at least half of them if we pick a prime table size. Moreover, we can always limit the number of quadratic probe steps and revert to an alternative scheme if quadratic probing fails to find an open slot.

And, even if the probe sequence is somewhat long for a small number of records, that will not ruin the average search performance.

And finally, if the insertion of very many records requires that many probe steps, the cost of subsequent searches for that record will be Θ(N), and that's unacceptable for a hash table since the goal is to beat Θ(log N) for the average search cost.

6. [12 points] In the context of hashing, what is a *collision*? Assume that the table is reasonably sized, say twice the number of data objects that will be stored in it, and that the function is 1-1 from the set of actual key values into the set of integers. If collisions still occur, is that only because of defects in the chosen hash function? If yes, explain why. If no, explain what else could cause a collision to occur.

A collision occurs when a hashing scheme maps two different key values into the same table slot.

Even if the hash function is 1-1as described, if the table size is too small we will get collisions.

And even if the table size is reasonable, when we mod the hash function value by the table size we may get the same table index, and hence a collision. The basic issue here is that being 1-1 from the set of actual keys into the (infinite) set of integers is not the same as being 1-1 into the (very finite) set of valid table indices.

7. Consider using a PR quadtree to organize data objects that lie in some bounded two-dimensional region.

a) [7 points] Suppose that all the data objects actually lie within the square bounded by (0, 0) and (256, 256), but that the developer does not know this and therefore he sets the world boundaries to correspond to the square bounded by (0, 0) and (1024, 1024). Give a precise description of the effect this will have on the resulting PR quadtree, compared to the tree that would result if the developer used the tighter bounds.

Since every actual data point lies in the SW quadrant of the SW quadrant of the selected world, the root node will have only a SW child, and that node will have only a SW child.

If the tighter bounds were used, then we would get the tree that is rooted at the SW child of the SW child of the other tree.

From a performance perspective, every search in the tree based on the larger world would require two extra steps.

b) [7 points] The GIS project requires using a bucketed PR quadtree, in which each leaf node can store several data objects instead of just one. What effect does bucketing have on the performance of the PR quadtree that results when a set of data objects is inserted into the tree? Why?

Suppose that the bucket size is k. It will not be necessary to split a leaf node until we encounter k+1 data points that lie in the region corresponding to that leaf node.

This should result in significantly shorter branches than if a bucket size of 1 were used; the degree of the benefit would depend on how much the actual data points are clustered. Clearly, this will reduce the memory cost of the tree, perhaps substantially.

The search performance effect is less obvious, while searches down shorter branches will require dropping through fewer levels, how does searching within a bucket compare to the cost of traversing the levels that were eliminated when a split was prevented?

That depends on how many levels we are talking about. If you had 4 children under a single internal node, versus the same 4 data values stored in a single leaf node, the advantage probably lies with the former case since comparing key values in a linear search of the bucket will likely cost more than determining which sub-region to descend into. But, if the 4 data values were scattered down a taller subtree, the advantage will probably lie with the bucketed implementation.

8. Suppose that an application performs a large number of searches for data records that are all stored in some structure in memory. There are **N** different data records.

a) [7 points] Do any of the data structures that we have discussed so far in the course guarantee that the worst-case search cost is $\Theta(\log N)$ or better? If so, which?

Only the AVL tree, for dynamic structures. An optimal hash table will achieve this, but optimality cannot be guaranteed unless you know the key values in advance and compute a perfect hash function.

A sorted array will achieve this as well, but the cost of insertions and deletions is much worse than for the AVL tree.

b) [7 points] What aspect of the implementation of the probabilistic skiplist that was discussed in class eliminates it from consideration in the previous question? Be precise.

The levels of the new skiplist node are determined randomly, and therefore it is possible that a long run of nodes in the same level could be created. Note that this is possible even if the desired distribution of nodes is achieved (1/2 in level 0, 1/4 in level 1, etc.

In that case, searches in that part of the skiplist would degenerate to a linear traversal. If that happens often enough then the average search cost will degenerate to worse than Θ(log N).

Note that this is entirely independent of whether the desired proportions of level-0, level-1,etc, nodes are created. It all has to do with how they are arranged.