1 Algorithmic Thinking

For each function $f(n)$ along the left side of the table, and for each function $g(n)$ across the top, write O, Ω, or Θ in the appropriate space, depending on whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, or $f(n) = \Theta(g(n))$. If more than one such relation holds between $f(n)$ and $g(n)$, *write only the strongest one* (which is the only answer considered correct).

The first row is a demo solution for $f(n) = n \log n$. The function $\log$ denotes logarithm to the base two unless otherwise noted. $\binom{n}{2}$ denotes the "n choose 2" symbol.

		$g(n)$		
		$n^{0.01}$	$n^{\log \log n}$	$n \log_{15} n$
$f(n)$	$n \log n$	Ω	O	Θ
	$(\log n)^{\log n}$	Ω	Θ	Ω
	$2^{\sqrt{\log n}}$	O	O	O
	$n \log \binom{n}{2}$	Ω	O	Θ

Useful observations:

$$n^{0.01} = 2^{0.01 \cdot \log n}$$

$$n^{\log \log n} = 2^{\log n \cdot \log \log n}$$

$$(\log n)^{\log n} = 2^{\log n \cdot \log \log n}$$

$$n \log \binom{n}{2} = n \log \frac{n!}{(n-2)!2!} = n \log \frac{n(n-1)}{2} = \Theta(n \log n).$$

2 Peak Finding

Design an algorithm that finds the maximum of a *rotated-sorted* array of n **distinct** elements in $O(\log n)$ time. A rotated-sorted array is formed by circularly shifting an array of integers arranged in increasing order (for example, $[9, 11, 2, 3, 6]$ is one such array.)

Note: You do not need to prove that your algorithm is correct, but you need to provide a **brief** analysis of its running time.

Solution: Denote the array by A and let $x = A[0]$ be the first element of the array. If $n = 1$, then x is trivially the maximum element.

Otherwise, compare the middle element, $m = A[\lfloor n/2 \rfloor]$, with x. If $m < x$, then the maximum element must be strictly before m in the array. In that case, recurse on the subarray of A consisting of all elements before m. Otherwise if $m > x$, then the maximum element must be m or after m, so recurse on the subarray of A from m to the end.

At each step of the algorithm, we perform a single comparison between two elements m and x, then recurse on a subproblem with half the size. The runtime of the algorithm must therefore satisfy the recurrence

$$T(n) = T(n/2) + \Theta(1),$$

which is $\Theta(\log n)$ by the Master Theorem.

Note: This problem is not equivalent to peak finding. The peak finding algorithm is only guaranteed to find *a* peak, but the rightmost element of the array in this case is also a peak.

3 Document distance

Name an operation that was one of the biggest asymptotic "time-sinks" in the original (un-optimized) version of the document distance program. (There may be more than one correct answer;any one will do.) What running time was incurred by this "time sink"?

Solution: There were several such time sinks. One was the computation of word counts by iterating through the whole list of words we have seen before for each new word we encounter,giving a Theta(n^2) running time (The linear time solution is to use Python dictionaries). Another was taking the inner product without sorting the word lists (also Theta(n^2)).

4 Insertion Sort and Merge Sort

True / False Running merge sort on an array of size n which is already correctly sorted takes $O(n)$ time.

Solution: False. The merge sort algorithm presented in class always divides and merges the array $O(\log n)$ times, so the running time is always $O(n \log n)$.

True / False Insertion Sort makes more comparisons than Heap Sort on all inputs.

Solution: *False. If the input is an already sorted list then Insertion Sort will make only $n - 1$ comparisons, while Heap Sort will make strictly more than that (in fact, $\Omega(n \log n)$).*

5 Master theorem

(Fall 14)

$T(n) = T(n/4) + T(3n/4) + \Theta(n)$

Solution: $T(n) = \Theta(n \log n)$. The best way to see this is to draw the recursion tree for a recursion behaving according to $T(n)$, as we saw in the lectures. The root corresponds to problem size n with cn extra work (where c is the constant behind $\Theta(n)$ in the recursion). The next level contains two nodes, one with problem size $n/4$ and the other with problem size $3n/4$, still adding up to n and thus incurring an extra cn work. For each following level, each node divides further and the total problem size at each level keeps adding up to n until we reach the first leaf. Depth of the leaves in the tree vary from between $\log_4 n$ and $\log_{4/3} n$, and at each level the amount of work done is at most cn (and exactly cn up to depth $\log_4 n$). Thus, the total amount of work done is at least $cn \log_4 n$ and at most $cn \log_{4/3} n$, and thus $\Theta(n \log n)$.

(Spring 14) What is the asymptotic runtime of an algorithm with the following recurrence:

$$T(n) = T(\frac{n}{3}) + T(\frac{n}{6}) + \Theta(n)$$

(a) $\Theta(\log n)$

(b) $\Theta(n)$

(c) $\Theta(n \log n)$

(d) $\Theta(n^2)$

(e) $\Theta(n^2 \log n)$

Solution: This recurrence is bounded by $T(n) \leq 2T(\frac{n}{3}) + \Theta(n)$. Solving this gives us $T(n) = O(n)$ as an upper bound. Because the recurrence itself contains a $\Theta(n)$, we know $T(n) = \Omega(n)$ also. Thus $T(n) = \Theta(n)$.

6 Heaps and HeapSort

Suppose a binary max-heap H contains 80 distinct keys. How many distinct positions might contain the *smallest* element in H?

(a) 32

(b) 40

(c) 41

(d) 42

(e) 64

Solution: (b)
There are 63 keys in the top 6 levels of the tree, and 17 keys in the 7th level. Of the 32 keys in the 6th level, 9 of them have children. Only the 17 keys in the 7th level, plus the 23 childless keys in the 6th level can contain the smallest element in H. So there are 40 such positions.

Which of the two algorithms {HeapSort, MergeSort}, implemented as described in class, is a better choice if **space** (memory usage) is the primary concern, rather than running time?

Solution: Heapsort; it sorts "in-place". Merge sort, as described in class, requires additional arrays to be allocated. (As a side note, using methods *not* described in class, in-place merging is possible, but it requires a great deal of work and is not at all practical).

7 Binary Search Trees, BST Sort

(Spring 15)
Given a **sorted list** of n numbers (not necessarily integers), give an algorithm to construct a **balanced binary search tree** containing the same numbers in time $O(n)$, or argue why it's not possible.

Solution: Given a sorted list A with n elements, note that for an index i, all elements to the right of i are greater than or equal to $A[i]$ and all elements to the left are less than or equal to $A[i]$. This is to say, $A[h] \leq A[i] \leq A[j]\ \forall h < i < j$. Thus, if we were to make a BST node with $A[i]$ as the root and all elements to the left of i as the left sub tree, and all elements to the right of i in the right sub tree then it would be a valid BST. The overall algorithm is as follows: for a list with n elements, create a node r from $A[n/2]$, recurse on the left half of A, take the tree generated and make it the left sub tree of r; the right side is symmetric.

The recursion is $T(n) = 2T(\frac{n}{2}) + O(1)$ because there are two sub-problems of size $\frac{n}{2}$ processed and attaching the trees takes constant time. This is $O(n)$ by Case 1 of the Master Theorem.

(alternatively:)

Describe (in clear English or pseudocode) an $O(n)$ algorithm for balancing an arbitrary binary search tree, that is, for producing a new binary search tree with the same elements as the original one but with height $O(\log(n))$. Include a time complexity analysis.

Solution: We first use an in-order traversal to generate a sorted list of the elements in the given BST. Once we have all the elements in sorted order, we can use the following recursive algorithm to construct a well balanced BST:

Time complexity analysis: The algorithm has time complexity of $O(n)$ for the inorder traversal and $O(n)$ for the construction of the balanced BST - this follows from the recurrence $T(n) = 2T(n/2) + O(1)$.

8 AVL Trees, AVL Sort

(Fall 15) Superconductivity Data Collection (4 parts)

In a series of experiments on superconductivity, you are collecting resistance measurements that were observed at different temperatures. Each data point contains a temperature reading and the resistance measured at that temperature. For example, the data point $(10, 3)$ was observed at temperature 10 (kelvins) and has resistance 3 (nano-ohms). At any time during your experiments, you would like to be able to insert a new data point, and to find the data point with the minimum resistance that was observed at or below temperature t kelvins.

Create a data structure that will allow you to insert data points, INSERT$((t, r))$ where t is the temperature and r is the measured resistance in $O(\log n)$ runtime. Your data structure must also allow finding the data point with the minimum resistance at or below a given temperature t, FIND-MINIMUM-RESISTANCE(t), in $O(\log n)$ runtime. You may assume that the set of data points is initially empty. The following table shows an example of a sequence of operations, along with their desired behaviors. Clarification: The temperatures of all data points are different.

operation	**desired behavior**
INSERT$((10, 3))$	update the set of data points to $\{(10, 3)\}$
INSERT$((2, 5))$	update the set of data points to $\{(10, 3), (2, 5)\}$
FIND-MINIMUM-RESISTANCE(10)	return the data point $(10, 3)$
FIND-MINIMUM-RESISTANCE(5)	return the data point $(2, 5)$
FIND-MINIMUM-RESISTANCE(1)	return `None`
INSERT$((4.5, 0.2))$	update the set of data points to $\{(10, 3), (2, 5), (4.5, 0.2)\}$
FIND-MINIMUM-RESISTANCE(5)	return the data point $(4.5, 0.2)$

(a) What data structure that we learned in class could be useful here? Describe how this data structure can be used to store the data.

Solution: We create a balanced binary search tree (such as an AVL tree) indexed by temperature. The nodes are sorted by the temperature of the data points. We also store the resistances of the data points in the nodes.

(b) How can this data structure be augmented to enable efficient FIND-MINIMUM-RESISTANCE(t) queries?

Solution: We augment each node with the (data point with) minimum resistance in its subtree (including that of the node itself).

(c) Describe how to insert into the data structure, INSERT$((t, r))$, while maintaining this augmentation in $O(\log n)$ time.

Solution: When we insert a new data point (t, r), we use t as our key. We update the augmented minimum resistance by comparing r with the minimum resistance stored in every node we pass while performing the insertion. Updating the minimum resistance takes 1 time per node. As the height of our AVL tree is bounded by $O(\log n)$, inserting a new node and updating the augmented data takes $O(\log n)$ time.

Next, we may need to perform rotations during rebalance. Using our augmentation, we may compute the minimum resistance of a subtree by comparing the resistance of the parent with the minimum resistance of each child's subtree, which is readily available at that children. Each rotation only requires $O(1)$ nodes to be updated with the new minimum value. Since up to $O(\log n)$ nodes are candidates for rotations, the total running time for rebalancing the tree is $O(\log n)$.

(d) Describe how to perform FIND-MINIMUM-RESISTANCE(t) in $O(\log n)$ time on the data structure you described above.

Solution: We search for the key t in our BBST. We make the modification to our SEARCH by keeping the data point with minimum resistance we have found so far (originally set to NONE). Suppose that we reach node x during our search. If the temperature at node x is at most t, we compare our current minimum resistance with the data point at x, as well as the minimum resistance on the left subtree stored at $x.left$; we update our data point accordingly. This algorithm is correct because the union of all the minima considered spans all data points with temperature less than or equal to t. This procedure takes time that is proportional to the length of the search path. Thus, the procedure runs in $O(\log n)$ time on a BBST.

9 Sorting Lower Bounds

Problem Consider the following multi-dictionary problem. Let $A[1..n]$ be a fixed array of distinct integers. Given an array $X[1..k]$, we want to find the position (if any) of each integer $X[i]$ in the array A. In other words, we want to compute an array $I[1..k]$ where for each i, either $I[i] = 0$ (so

zero means none) or $A[I[i]] = X[i]$. Determine the exact complexity of this problem, as a function of n and k in the binary decision tree model.

Solution: We will first show a lower bound for this problem in the binary decision tree model. We know that each leaf of the binary decision tree should correspond to a single solution and also that the running time lower bound corresponds to the height of the tree. Thus, the lower bound is $\Omega(\log(\sharp possible\ answers)) = \Omega(\log(\frac{n!}{(n-k)!})) = \Omega(k \log n)$. This lower bound is tight since there exists an easy $O(k \log n)$ algorithm that solves this problem. Namely, the algorithm is to do a binary search for each of the k elements in the array X and for each of them return the index $I[i]$ of A such that $A[I[i]] = X[i]$ or 0 if there is no such index. Each binary search takes $O(\log n)$ time, so the algorithm takes $O(k \log n)$ time in total.

(Fall 10) We can sort 7 numbers with 10 comparisons.

Solution: False. To sort 7 numbers, the binary tree must have $7! = 5040$ leaves. The number of leaves of a complete binary tree of height 10 is $2^{10} = 1024$. This is not enough.

10 Counting Sort, Radix Sort

Ben Bitdiddle modifies RADIX-SORT to use INSERTION-SORT to sort by digits, instead of COUNTING-SORT. Would the resulting algorithm still work correctly? What is the complexity of this new algorithm? The complexity of conventional RADIX-SORT on n numbers in base b with at most d digits is $O((n + b) \cdot d)$.

Solution: Insertion sort is a stable sort, so the new Radix Sort algorithm would still work correctly. The new complexity becomes $O(n^2 \cdot d)$, because now we are using insertion sort, an $O(n^2)$ algorithm, d times in our Radix Sort.

11 Hashing with Chaining

Given an array A of n integers and an integer k, detect if there is an entry $A[i]$ that is equal to one of the k previous entries $A[i-1] \dots A[i-k]$. Your algorithm should run in time $O(n)$. You can assume you have access to a hash function which satisfies the simple uniform hashing assumption (SUHA).

Example: Given an array $A = [1, 3, 5, 7, 6, 5, 2]$ and $k = 4$, the algorithm should output YES since $A[3] = A[6] = 5$.

Note: You do not need to prove that your algorithm is correct, but you need to provide a **brief** analysis of its running time.

Solution: We will use hashing to solve this problem. We create a hash table H of size k which is initially empty. As we go through the array A, whenever we are at position i, H will contain the values of the k previous entries $A[i-1], ..., A[i-k]$. More precisely, for $i = 1$ to n, we do the following:

- Check whether H contains element $A[i]$. If yes, we are done!
- Remove element $A[i-k]$ from H (if it exists) and add element $A[i]$.

The last step makes sure, that H will contain elements $A[i]...A[i-k+1]$ when we visit the entry $A[i+1]$.

The algorithm above runs in $O(n)$ expected time, since we perform at most n iterations and each iteration takes $O(1)$ in expection. This is because, every time we do at most one look-up in H, one insertion and one deletion and every such operation takes $O(1+\alpha)$ in expectation. This is $O(1)$ because there always at most k elements in the hash table and the load factor is $\alpha = \frac{k}{k} = 1$.

An alternative solution is to have a hash table (dictionary) of size n and everytime keep the most recent occurrence of an element. Then, after adding all elements of the array up to $i-1$ in the hash table, we can check if $A[i]$ has a duplicate by looking up its most recent occurrence in the hash table and checking if it was during the previous k steps, $i-k, ..., i-1$.

Common Mistakes: A common mistake among people that implemented the alternative solution was to keep an entry in the hash table for each occurrence. This can result in $O(n^2)$ time since many collisions can occur everytime. The uniform hashing assumption protects from false collisions, but not from true collisions. Consider for example the array $[1, 2, 1, 2, 1, 2, 1, 2, ...]$ for $k = 1$. In this case, all the 1s will go to the same entry in the hash table, and if we were to keep an entry for each 1 we encounter, we would have $O(n)$ entries. Searching through them everytime would take O(n) time which would give a $O(n^2)$ runtime.

Another common approach was to not use hashing but use a regular array instead. This would work if the numbers were bounded but is impossible to implement if the numbers are huge. Solutions of this sort as well as solutions based on counting/radix sort, which suffer from the same problem, received lower scores for this problem.

12 Table Doubling, Karp-Rabin

(Spring 16) Consider a modification to the table doubling procedure in which whenever the number of keys n is at least $\frac{m}{2}$, where m is the current size of the table, we set its new size m' to be $m' = 2m + \lfloor\sqrt{m}\rfloor$. If we never delete any elements from our table, then the resulting amortized overhead of this modified table doubling is still only $O(1)$ per each hash table operation.

Solution: True

Remarks: The new size m' is still at least twice the size of the old table and at most three times that size. So, the $O(1)$ per each hash table operation amortized analysis we covered still works.

(Fall 12) When using "Table Doubling" to maintain the size of a hash table (with insertions but no deletions), the size of the table grows exponentially with the number n of keys inserted.

Solution: False. The table size is at most a constant times the number of keys. There is no reason to have exponentially many buckets.

(Fall 12) Recall the Rabin-Karp string-matching algorithm, which uses a rolling hash to search for a pattern of length m in a text of length n. Suppose it is run until the first match, if any, is found. Then the expected running time is $\Theta(nm)$.

Solution: False. The running time is $O(n + m)$, which is effectively $O(n)$ if the pattern is shorter than the text.