Quiz 1

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet. Circle your recitation at the bottom of this page.
- You have 100 minutes to earn a maximum of 100 points. Do not spend too much time on any one problem. Read them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed one-sided letter-sized sheet with your own notes**. No calculators or programmable devices are permitted. No cell phones or other communication devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points	Grade	Grader
Name	0	2		
1	7	14		
2	1	9		
3	3	12		
4	4	24		
5	2	20		
6	2	19		
Total		100		

Name: _____

Circle your recitation:	R01 Brando Miranda **10AM**	R02 Brando Miranda **11AM**	R03 Parker Zhao **12PM**	R04 Alex Jaffe **12PM**	R05/R07 Danil Tyulmankov **1PM 2PM**	R06 Peinan Chen **1PM**	R08 Kevin Tian **2PM**	R09 Allen Park **3PM**	R10 Daniel Manesh **4PM**

Problem 0. What is Your Name? [2 points] (2 parts)

(a) [1 point] Flip back to the cover page. Write your name and circle your recitation section.

(b) [1 point] Write your name on top of each page.

Problem 1. True or False [14 points] (7 parts)

For each of the following questions, circle either **T** (True) or **F** (False). There is no need to justify the answers. Each problem is worth 2 points.

(a) **T** **F** Every algorithm with $\Theta(n^2)$ running time is slower than any $\Theta(\sqrt{n})$-time algorithm on all inputs.

(b) **T** **F** If $\log f(n)$ is $O(\log g(n))$ then $f(n)$ is $O(g(n))$.

(c) **T** **F** Insertion Sort makes more comparisons than Heap Sort on all inputs.

(d) **T** **F** An in-order traversal of a Max Heap (which runs just like an In-Order-Tree-Walk for a BST) always produces a sorted list of its elements.

(e) **T** **F** Let v be the only node of a Max Heap that does *not* satisfy the Max Heap Property. Running the MAX-HEAPIFY() procedure on v will always result in a Max Heap for which all nodes satisfy the Max Heap Property.

(f) **T** **F** The successor search in a Binary Search Tree always runs in $\Omega(\log n)$ time.

(g) **T** **F** For every node u in an AVL tree, the number of nodes in the left subtree of u and the number of nodes in the right subtree of u can differ by a factor of at most two.

Problem 2. Asymptotic Notation [9 points] (1 part)

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
	$(\log n)^{\log n}$			
	$2^{\sqrt{\log n}}$			
	$n \log \binom{n}{2}$			

Problem 3. Fun with Recurrences [12 points] (3 parts)

Derive the asymptotic growth of the following functions $T(n)$ in Θ notation, and briefly describe your work.

(a) [4 points] $T(n) = 3 \cdot T(\frac{n}{3}) + \Theta(n \log n)$

(b) [4 points] $T(n) = T(\frac{n}{3}) + \Theta(3^n)$

(c) [4 points] $T(n) = T(n-1) + \Theta(\log n)$

Problem 4. 6.006 Adventures of Ben Bitdiddle [24 points] (4 parts)

(a) [6 points] Ben Bitdiddle is taking 6.006 and after hearing about Max Heaps he decides to implement one. However, as he was not paying enough attention in class, his implementation of the MAX-HEAPIFY() procedure is flawed. Specifically, the pseudo-code of his implementation is as follows:

```
BB-Max-Heapify(H,x):
  If H[Left(x)].key > H[x].key:
     Swap elements H[x] and H[Left(x)]
     BB-Max-Heapify(H,Left(x))
  Elseif H[Right(x)].key > H[x].key:
     Swap elements H[x] and H[Right(x)]
     BB-Max-Heapify(H,Right(x))
```

Provide an example of a heap H on which the above implementation fails. That is, H should be a Max Heap in which the root $H[1]$ is the only node violating the Max Heap Property, and calling the above procedure BB-MAX-HEAPIFY$(H, 1)$ will *NOT* result in a valid Max Heap. What will that resulting heap be?

Draw the heap H before *and* after running BB-MAX-HEAPIFY$(H, 1)$ on it. No other explanation is needed. (Note: Your example can be made quite small.)

(b) [6 points] Ben finally fixed the above bug and now has at his disposal a correct implementation of a Max Heap that supports all the standard operations: EXTRACT-MAX(H), INSERT(H,K), INCREASE-KEY(H,X,K), in time $O(\log n)$; and MAX(H), in time $O(1)$, where n is the number of stored elements.

Ben wants now to add a new capability to this Max Heap. Namely, he would like to implement a DELETE-ELEMENT(H,X) operation that removes element $H[x]$ from an n-element Max Heap H in time $O(\log n)$. (Note that the $H[x]$ does not need to be the maximum-key element in H. Also, naturally, H has to remain a Max Heap after deletion of $H[x]$.)

Help Ben by providing an implementation of such a DELETE-ELEMENT(H,X) operation with a *brief* analysis of its running time.

(c) [6 points] After learning about hashing in the class, Ben has decided to implement a hash table to use it as the ultimate storage for all his data. Unfortunately, he forgot to implement a collision resolution scheme. As a result, whenever a new element is inserted into a particular cell of such table, it overwrites any element that was in that cell before (if any).

Assume that the hash function that Ben uses satisfies the Simple Uniform Hashing Assumption (SUHA), and that n distinct elements were inserted into such a "collision resolution free" hash table of size m. If Ben now calls a SEARCH-KEY(x) operation, with x being the *first* element inserted, what is the probability that the answer to that query will be correct?

(d) [6 points] Ben Bitdiddle is also helping out with the gravitational waves detection project. He is responsible for collecting a large number of measurements from one of the devices. Each measurement is a four-dimensional point (x, y, z, t), where x, y, z, t are integers such that $-M \leq x, y, z, t \leq M$.

Unfortunately, due to Ben's mistake, some of the measurements were recorded multiple times! He needs now to devise a fast algorithm that removes all duplicate measurements. Can you devise an algorithm that given a list of n such four-dimensional measurements, finds all the duplicated entries in $O(n + M)$ time? (Assume you do *not* have access to randomness. So, you cannot use a hashing technique.)

Problem 5. Nearest-Neighbor Search on a Line [20 points] (2 parts)

Alyssa P. Hacker wants to perform nearest neighbor search on a line. Specifically, given an array A of n different (not necessarily integer) points on a line and a query point q as well as a number $1 \leq k \leq n$, Alyssa would like to compute k *nearest neighbors* of q in A. That is, she would like to output a list of k points in A that are closest to q with respect to their distance on a line (breaking ties arbitrarily), sorted non-decreasingly by their distance from q. (Recall that the distance between points p and r on a line is $|p - r|$.)

For example, if the input is $A = [-1.3, 6.4, 4, 9.1, 7.3, -11]$, then on a query $q = 1.1$ and $k = 3$ the output should be $[-1.3, 4, 6.4]$. On the other hand, on a query $q = 3.2$ and $k = 4$, the output should be either $[4, 6.4, -1.3, 7.3]$ or $[4, 6.4, 7.3, -1.3]$.

(a) [12 points]

Help Alyssa by devising a data structure that can be constructed out of a size n input array A in $O(n \log n)$ time and then is able to find for *any* query point q and *any* number $1 \leq k \leq n$, a list of q's k nearest neighbors in A in $O(k + \log n)$ time. (Note that the query point q does not need to be in A.)

(b) [8 points]

Argue *briefly* why devising such a data structure would be impossible if we insisted on the construction time be only $O(n)$. Specifically, provide a reasoning that shows that existence of such a hypothetical nearest neighbor data structure with $O(n)$ construction time would enable us to solve another algorithmic task faster than we know is possible.

Problem 6. Find the Unbaked Cookies! [19 points] (2 parts)

Prof. Madry is getting the cookies for the whole class (since they aced the Part B challenge again!). After he had already picked up the cookies from the bakery, the owner called to tell him that somehow they included an unbaked cookie in the batch!

Now, Prof. Madry must find this unbaked cookie, and do it quickly because the class starts soon. The problem is that, without tasting it, the only difference between a baked and unbaked cookie is that the unbaked cookie is slightly heavier.

To solve this problem, Prof. Madry wants to use a scale that can compare the weight of one set of cookies against the weight of another set of cookies. (The scale does not enable one to measure absolute weights of cookies.)

Can you help Prof. Madry find the unbaked cookie?

(a) [9 points] Design an algorithm that is guaranteed to find the unbaked cookie among all n cookies with $O(\log n)$ weighings on the balance scale.

Note: You do not need to prove that your algorithm is correct (but, of course, it should be), but you need to provide a *brief* analysis that bounds the number of scale uses.

(b) [10 points] What if there are *two* unbaked cookies? Extend the algorithm from part (a) to an algorithm that is guaranteed to find *both* unbaked cookies among all the n cookies after $O(\log n)$ weighings with the scale. Assume that both unbaked cookies have exactly the same weight and that n is a power of 2.

Note: You do not need to prove that your algorithm is correct (but, of course, it should be), but you need to provide a *brief* analysis that bounds the number of scale uses.

SCRATCH PAPER

SCRATCH PAPER