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.

Solution: *False*

Remarks: *Consider two algorithms: one with running time $T_1(n) = 1 + n^2$, and a one with running time $T_2(n) = 3 + 10 \cdot \sqrt{n}$. On inputs of size $n = 1$, we have that $T_1(1) = 2$, which is strictly smaller than $T_2(1) = 13$.*

Also, our running times denote the worst-case running times. It might happened that one algorithm is slower than another in terms of worst-case behavior, but still is faster on certain inputs. (Good example: Insert sort vs. Heap sort - see point (c) below.)

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

Solution: *False*

Remarks: *Consider $f(n) = n^2$ and $g(n) = n$.*

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

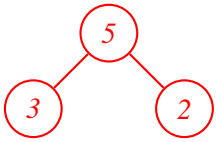
Solution: *False*

Remarks: *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, $\Omega(n \log n)$).*

(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.

Solution: *False*

Remarks: *In-order traversal of the following Max Heap will produce $[3, 5, 2]$, which evidently is not sorted.*

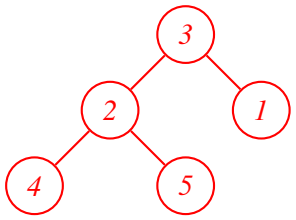


(e) **T** **F** Let v be the only node of a Max Heap that does *not* satisfy the Max Heap Property. Running the MAX-HEAPIFY() procedure on v will always result in a Max Heap for which all nodes satisfy the Max Heap Property.

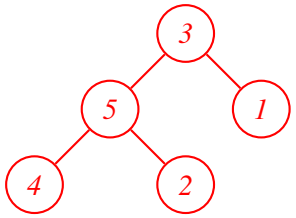
Solution: *False*

Remarks: *Recall that the* MAX-HEAPIFY() *procedure is designed to fix any violations of the Max Heap Property only in the* subtree *rooted at the vertex it is run on.*

More precisely, consider the following heap:



Clearly, the only node v that violates the Max Heap Property is the node with the key value 2. However, after we run the MAX-HEAPIFY() *procedure on that node, the resulting heap will be:*



In this heap, the root, i.e., the node with key value 3, is violating Max Heap Property. (Even though it was not violating it before.)

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

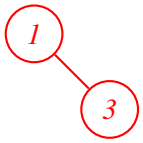
Solution: *False*

Remarks: *Consider an n-element BST in which each node (except the unique leaf) has only a right child. Clearly, successor of a given internal node is exactly its right child. Also, the successor search will always run in $O(1)$ time in that case, as it boils down to calling* FIND-MIN *in the subtree rooted at the right child and this procedure will terminate in $O(1)$ time too since that right child does not have any left child.*

(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.

Solution: *False*

Remarks: *One simple counter-example here is the following AVL tree.*



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

Useful observations:

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

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

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

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

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

Solution: *The number L of leaves in the recursion tree is $L = \Theta(n^{\log_3 3}) = \Theta(n)$. So, $\Theta(n \log n) = \Theta(L \log n)$.*
By Case 2' of Master theorem, we have that

$$T(n) = \Theta(L \log^2 n) = \Theta(n \log^2 n).$$

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

Solution: *The number L of leaves in the recursion tree is $L = \Theta(n^{\log_3 1}) = \Theta(1)$. So, $f(n) = \Theta(3^n) = \Omega(L^{1+\varepsilon})$, for some (in fact, any) constant $\varepsilon > 0$, and $f(n/3) \leq \frac{1}{3} \cdot f(n/3)$, for sufficiently large n.*
Thus, by Case 3 of Master theorem, we have that

$$T(n) = \Theta(f(n)) = \Theta(3^n).$$

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

Solution: *Observe that the recursion tree here is a path of height n. Each level i contributes $\Theta(\log i)$. Consequently,*

$$T(n) = \sum_{k=1}^{n} \Theta(\log k) = \Theta(n \log n).$$

Alternatively, one can use substitution method. That is, guess that, for sufficiently large n, $T(n) = C \cdot n \log n$ and verify that this is indeed the case by induction. (Note that the constant C here needs to depend also on the leading constant hidden in the $\Theta(\cdot)$ notation.)

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

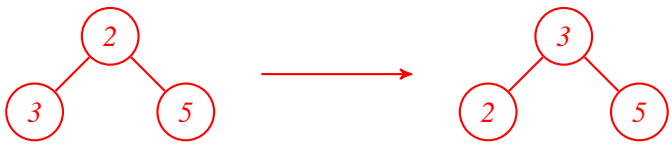
(a) [6 points] Ben Bitdiddle is taking 6.006 and after hearing about Max Heaps he decides to implement one. However, as he was not paying enough attention in class, his implementation of the MAX-HEAPIFY() procedure is flawed. Specifically, the pseudo-code of his implementation is as follows:

```
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.)

Solution: *The mistake that Ben made was that in his implementation of the* Max_Heapify() *procedure if both children of the node $H[x]$ have larger keys then this procedure always swaps that node with the left child instead of swapping it with the child that has the larger key value. Consequently, an example of a correct answer is the following heap before and after running* BB-MAX-HEAPIFY$(H, 1)$ *on it:*



Clearly, the root of the resulting heap violates the Max Heap Property and thus that heap is not a valid Max Heap.

(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.

Solution: *Consider the following implementation of* DELETE-ELEMENT(H,X)*:*

1. *Set* MAX $\leftarrow$ MAX(H)
2. *Call* INCREASE-KEY(H,X,MAX+1)
3. *Execute* EXTRACT-MAX(H).

Observe that since MAX $=$ MAX(H) *is the key value of the maximum key value element in the heap, the* INCREASE-KEY(H,X,MAX+1) *procedure ensures that the element $H[x]$ becomes the maximum key value element now. Consequently, calling* EXTRACT-MAX(H) *will remove it from the heap, leaving it a valid Max Heap that contains all the elements (with the original key values) but the element we wanted to remove.*

Since MAX(H) *runs in $O(1)$ time and both* INCREASE-KEY(H,X,MAX+1) *and* EXTRACT-MAX(H) *take $O(\log n)$ time, the overall running time of our implementation of* DELETE-ELEMENT(H,X) *is $O(\log n)$, as desired.*

(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?

Solution: *Since there is no collision handling, the* SEARCH-KEY(x) *call correctly reports that the element x is in the hash table only if none of the $n-1$ elements inserted into that table after inserting x collides with it, that is, is hashed to the same cell as x.*

By SUHA, whenever each of these $n-1$ elements is being inserted, the cell to which these elements ends up being hashed to is chosen uniformly at random from all the m possible cells. As a result, the probability that such element does not *collide with x is exactly*

$$\frac{m-1}{m} = \left(1 - \frac{1}{m}\right).$$

Since, by SUHA, all these events are independent, the probability that none of these $n-1$ elements collides with x is exactly

$$\left(1 - \frac{1}{m}\right)^{n-1}.$$

(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.)

Solution: *Consider the following procedure:*

1. *Map all four coordinates in each measurement to* $[0, 2M]$ *range by adding* M *to them*
2. *Use Radix sort to sort all the measurements, treating each measurement* (x, y', z', t') *as a four-digit number base* $(2M + 1)$
3. *Duplicate measurements have to appear as contiguous blocks in such sorted list, so one can "trim" these blocks down to a single entry in just a single pass. Resulting list will have no duplicate entries anymore*
4. *Map all the measurements back to original ranges by subtracting* M *from each of their four coordinates*

The time needed to execute Steps 1, 3, and 4 is only $O(n)$*, while sorting all the measurements in Step 2 takes*

$$O((n + (2M + 1))d) = O(n + M),$$

since the number of digits d *is 4 here.*
Overall running time is thus $O(n + M)$*, as needed.*

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.)

Solution: *One way to implement such a data structure is to simply keep the array A sorted. That is, as a construction step we sort the array A. If we use Merge sort, this will take $O(n \log n)$, as desired.*

Now, to find k nearest neighbors of a query point q, we perform the following procedure.

First, use binary search to find in the (sorted) array A an index i such that all points in the subarray A[i:n-1] are all either greater or equal to q, and all the points in the subarray A[0:i-1] are strictly smaller than q. This takes $O(\log n)$ time.

Then, note that now, in each of these two subarrays, the points are sorted according to their distance to q. (A[0:i-1] is sorted non-increasingly, while A[i:n-1] is sorted non-decreasingly.) Consequently, k nearest neighbors of q have to be contained in the union of the length-k suffix A[i-k-1:i-1] of A[0:i-1] and length-k prefix A[i:i+k] of A[i:n-1]. To identify these k nearest neighbors in $O(k)$ time, merge these two sorted length-k subarrays into a single sorted length-$2k$ array and return the first k elements of it.

Therefore, the overall running time of answering such a query is $O(k + \log n)$, as desired.

(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.

Solution: *If such an $O(n)$ construction time data structure existed, it would enable us to sort n arbitrary real numbers in $O(n)$ time. From the class though, we know that this is impossible.*

To see how such a hypothetical data structure could be used for sorting an array A of n arbitrary real numbers in $O(n)$ time, consider the following procedure:

1. *Find a minimum element x in A in $O(n)$ time*
2. *Construct the data structure for A in $O(n)$ time*
3. *Query the data structure with $q = x$ and $k = n$. This would return all the elements of A in an non-decreasing order with respect to their distance to x. But, as x is the minimum element of A, this order would be exactly the sorted order of elements in A*

Consequently, we would indeed be able to sort A in $O(n)$ time, which is impossible.

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.

Solution: *We apply a divide and conquer approach. Let n be the total number of cookies.*

If $n = 1$ then, clearly, the single cookie has to be the unbaked one.

If $n > 1$, assume first that n is even. Then, we use the scale once to compare the total weight of the first half of the cookies against the other half. As the unbaked cookie is slightly heavier than the baked ones, one of these halves has to be heavier and we recurse on that half.

If $n > 1$ and n is odd, we set one (arbitrarily chosen) cookie aside and use the scale once to weigh the two halves of the remaining cookies against each other. If one of these halves ends up being heavier, we recurse on it. Otherwise, i.e., if these two halves weigh exactly the same, it means that the cookie we set aside is the unbaked one and we can terminate.

Observe that in the above algorithm one use of the scale enables us to reduce the size of our problem by at least a half. So, the total number $S(n)$ of scale uses of this algorithm on n cookies obeys the following recurrence relation:

$$S(n) \leq S(n/2) + 1.$$

By Case 2 of Master Theorem, we know that $S(n) = O(\log n)$, as desired.

(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.

Solution: *Again, we apply a divide and conquer approach. Let* $n = 2^k$ *(as we are allowed to assume that here).*

If $n = 2$*, i.e.,* $k = 1$*, we know that these two cookies are the unbaked ones.*

If $n = 2^k$*, for* $k > 1$*, we use the scale once to weigh the first half of cookies against the other half. If one of these halves ends up being heavier, we know that* both *unbaked cookies are in that half and we can recurse on it.*

However, if the weight of these two halves is equal it means that we have exactly one *of the two unbaked cookies in each of the halves. We thus run the algorithm from Point (a) on each one of these two halves to find these cookies.*

To bound the total number $S(n)$ *of scale uses needed by this algorithm when run on* n *cookies, observe that whenever we recurse, we used the scale once to reduce the size of our problem by a half. On the other hand, whenever we do not recurse, we find both unbaked cookies after two calls to algorithm from Point (a), which uses* $O(\log n)$ *scale weightings overall.*

Consequently, we can write the following recurrence relation for $S(n)$*:*

$$S(n) \leq \max\{S(n/2) + 1, O(\log n)\}.$$

As by Case 2 of Master Theorem, we know that the recurrence relation $T(n) \leq T(n/2) + 1$ *implies that* $T(n) = O(\log n)$*. We know that we always have that* $S(n) = O(\log n)$*, as desired.*

SCRATCH PAPER

SCRATCH PAPER