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 every page of this quiz booklet.
- You have 120 minutes to earn 120 points. Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress.
- This quiz is closed book. You may use **one** $8\frac{1}{2}'' \times 11''$ or A4 crib sheet (both sides). No calculators or programmable devices are permitted. No cell phones or other communications 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. It is sufficient to cite known results.
- When asked for an algorithm, your algorithm should have the time complexity specified in the problem. If you cannot find such an algorithm, you will generally receive partial credit for a slower algorithm **if you analyze your algorithm correctly**.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem	Parts	Points	Grade	Grader
1	2	2		
2	9	18		
3	3	20		
4	1	20		
5	2	25		
6	3	20		
7	2	15		
Total		120		

Name: _____

Friday	Zuzana	Debmalya	Ning	Matthew	Alina	Alex
Recitation:	**10 AM**	**11 AM**	**12 PM**	**1 PM**	**2 PM**	**3 PM**

Problem 1. What is Your Name? [2 points] (2 parts)

(a) [1 point] Flip back to the cover page. Write your name there.

(b) [1 point] Flip back to the cover page. Circle your recitation section.

Problem 2. True or False [18 points] (9 parts)

For each of the following questions, circle either T (True) or F (False). There is no penalty for incorrect answers.

(a) T F [2 points] Given two heaps with n elements each, it is possible to construct a single heap comprising all $2n$ elements in $O(n)$ time.

Solution: TRUE. Simply traverse each heap and read off all $2n$ elements into a new array. Then, make the array into a heap in $O(n)$ time by calling MAX-HEAPIFY for $i = 2n$ down to 1.

(b) T F [2 points] Building a heap with n elements can always be done in $O(n \log n)$ time.

Solution: TRUE. In fact, we can build a heap in $O(n)$ time by putting the elements in an array and then calling MAX-HEAPIFY for $i = n$ down to 1.

(c) T F [2 points] Given a hash table of size n with n elements, using chaining, the minimum element can always be found in $O(1)$ time.

Solution: FALSE. We will need to scan the entire table; this takes $\Omega(n)$ time.

(d) T F [2 points] 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 $O(n \log n)$.

(e) T F [2 points] We can always find the maximum in a min-heap in $O(\log n)$ time.

Solution: FALSE. The *maximum* element in a min-heap can be anywhere in the bottom level of the heap. There are up to $n/2$ elements in the bottom level, so finding the maximum can take up to $O(n)$ time.

(f) T F [2 points] In a heap of depth d, there must be at least 2^d elements. (Assume the depth of the first element (or root) is zero).

Solution: TRUE. The minimum number of elements in a heap of depth d is one more than the maximum number of elements in a heap of depth $(d-1)$. Since a level at depth d in a binary heap can have up to 2^d elements, the number of elements at depth $(d-1)$ is $\sum_{i=0}^{d-1} 2^i = 2^d - 1$. So the minimum number of elements in a heap of depth d is $(2^d - 1) + 1 = 2^d$.

(g) T F [2 points] Consider a family of hash functions $\{h_a\}$ hashing an r-bit number to an s-bit number ($r > s$). This family is defined by $h_a(x) = (x \operatorname{xor} a) \bmod 2^s$, where "xor" represents binary exclusive or and where a is a randomly chosen r-bit number.

This hash function family is 2-universal.

Solution: FALSE. Note that for all $(r-s)$-bit numbers k, the keys $k \cdot 2^s$ all hash to $a \bmod 2^s$. This means that the keys 2^s, $2 \cdot 2^s$, $3 \cdot 2^s$, etc. always collide regardless of the value of a. This means that the hash values are not pairwise independent and thus that the hash function family is not 2-universal.

(h) T F [2 points] Inserting an element into a binary search tree of size n always takes $O(\log n)$ time.

Solution: FALSE. Inserting an element into a binary search tree takes $O(h)$ time, where h is the height of the tree. If the tree is not balanced, h may be much larger than $\log n$ (as large as $n-1$).

(i) T F [2 points] Any two (possibly unbalanced) BSTs containing n elements each can be merged into a single balanced BST in $O(n)$ time.

Solution: TRUE. Use in-order traversal of the two BSTs to create two sorted lists of length n in $O(n)$ time, merge them into a single sorted list of length $2n$ in $O(n)$ time, and then create a balanced BST from the sorted lists in $O(n)$ time.

Problem 3. Asymptotics & Recurrences [20 points] (3 parts)

(a) [10 points] Rank the following functions by *increasing* order of growth. That is, find any arrangement $g_1, g_2, g_3, g_4, g_5, g_6, g_7$ of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, $g_3 = O(g_4)$, $g_4 = O(g_5)$, $g_5 = O(g_6)$, $g_6 = O(g_7)$.

$$f_1(n) = n^4 + \log n \qquad f_2(n) = n + \log^4 n \qquad f_3(n) = n \log n \qquad f_4(n) = \binom{n}{3}$$
$$f_5(n) = \binom{n}{n/2} \qquad f_6(n) = 2^n \qquad f_7(n) = n^{\log n}$$

Solution:

$$f_1(n) = O\left(n^4\right) \qquad f_2(n) = O\left(n\right) \qquad f_3(n) = O\left(n \log n\right)$$
$$f_4(n) = \frac{n(n-1)(n-2)}{6} = O\left(n^3\right) \qquad f_6(n) = O\left(2^n\right) \qquad f_7(n) = n^{\log n} = 2^{(\log n)^2}$$

We can determine the asymptotic complexity of f_5 by using Stirling's approximation, $n! \approx \sqrt{2\pi n}\,(n/e)^n$:

$$f_5(n) = \frac{n!}{((n/2)!)^2} \approx \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\left(\sqrt{2\pi n}\left(\frac{n/2}{e}\right)^{n/2}\right)^2} = \frac{2^n}{\sqrt{2\pi n}} = O\left(2^n/\sqrt{n}\right)$$

Thus, the correct order is (from slowest to fastest growing):

$$f_2, f_3, f_4, f_1, f_7, f_5, f_6$$

(b) [5 points] Find an asymptotic solution of the following functional recurrence. Express your answer using Θ-notation.

$$T(n) = 9 \cdot T(n/3) + n^3$$

Solution: Using the master theorem, $a = 9$, $b = 3$, and $\log_b a = \log_3 9 = 2$. Thus, we compare $n^{\log_b a} = n^2$ to $f(n) = n^3$. Since $n^3 = \Omega\left(n^{2+\epsilon}\right)$, $f(n)$ dominates the recurrence and we are in Case 3 of the master theorem. Thus, $T(n) = \Theta\left(n^3\right)$.

(c) [5 points] Find an asymptotic solution of the following functional recurrence. Express your answer using Θ-notation.

$$T(n) = 2 \cdot T(n/4) + \sqrt{n}$$

Solution: Using the master theorem, $a = 2$, $b = 4$, and $\log_b a = \log_4 2 = 1/2$. Thus, we compare $n^{\log_b a} = n^{1/2}$ to $f(n) = \sqrt{n}$. Since these are asymptotically equivalent, we are in Case 2 of the master theorem. This means that we gain an additional $\log n$ factor and $T(n) = \Theta\left(\sqrt{n} \log n\right)$.

Problem 4. Median of Lists [20 points]

You are given two lists of integers, each of which is sorted in ascending order and each of which has length n. All integers in the two lists are different. You wish to find the n-th smallest element of the union of the two lists. (That is, if you concatenated the lists and sorted the resulting list in ascending order, the element which would be at the n-th position.) Present an algorithm to find this element in $O(\log n)$ time. You will receive half credit for an $O\left((\log n)^2\right)$-time algorithm.

Solution: Assuming the lists $l1$ and $l2$ have constant access time to any of their elements, we can use a modified version of binary search to get an $O(\log n)$ solution. The easiest approach is to search for an index $p1$ in just one of the lists and calculate the corresponding index $p2$ in the other list so that $p1 + p2 = n$ at all times. (This assumes the lists are indexed from 1.)

First we check for the special case when all elements of one list are smaller than any element in the other list:

If $l1[n] < l2[0]$ return $l1[n]$.
If $l2[n] < l1[0]$ return $l2[n]$.

If we do not find the n-th smallest element after this step, call $findNth(1, n)$ with the approximate pseudocode:

```
findNth(start,end)
p1 = ⌊(start + end)/2⌋
p2 = n − p1
if l1[p1] < l2[p2]:
  if l1[p1 + 1] > l2[p2]:
      return l2[p2]
  else:
      return findNth(p1+1, end)
else:
  if l2[p2 + 1] > l1[p1]:
      return l1[p1]
  else:
      return findNth(start,p1-1)
```

Element $l2[p2]$ is returned when $l2[p2]$ is greater than exactly $p1 + p2 - 1 = n - 1$ elements (and therefore is the n-th smallest). $l1[p1]$ is returned under the same but symmetric conditions. If $l1[p1] < l2[p2]$ and $l1[p1+1] < l2[p2]$, the rank of $l2[p2]$ is greater than n, so we need to take more elements from $l1$ and less from $l2$. Therefore we search for $p1$ in the upper half of the previous search interval. On the other hand, if $l2[p2] < l1[p1]$ and $l2[p2 + 1] < l1[p1]$, the rank of $l1[p1]$ is greater than n. Therefore the real $p1$ will lie in the bottom half of our current search interval.

Since we are halving the size of the problem at each call to $findNth$ and we need to do only constant work to halve the problem size, the recurrence for this algorithm is $T(n) = T(n/2) + \Theta(1)$, which has an $O(\log n)$-time solution.

The $O(\log^2 n)$ solution (awarded at most 10 points) uses alternates binary search on each list. In short, it takes the middle element in the current search interval in the first list ($l1[p1]$) and searches for it in $l2$. Since the elements are unique, we will find at most 2 values closest to $l1[p1]$. Depending on their values relative to $l1[p1-1]$ and $l1[p1+1]$ and their indices $p2_1$ and $p2_2$, we either return the n-th element or recurse: If the sum of any out of the (at most) 3 indices in $l1$ can be combined with one of the (at most) 2 indices in $l2$ so that $l1[p1']$ and $l2[p2']$ would be right next to each other in the sorted union of the two lists and $p1' + p2' = n$ or $p1' + p2' = n + 1$, we return one of the 5 elements. If $p1 + p2 > n$, we recurse to left half of the search interval in $l1$, otherwise we recurse to the right interval. This way, for out of the $O(\log n)$ possible midpoints in $l1$ we do an $O(\log n)$ binary search in $l2$. Therefore the running time is $O(\log^2 n)$.

The $O(n)$ solution (awarded at most 5 points) does a modified merge operation (without the need to copy the elements into a new array) and stops when it reaches the n-th element.

Problem 5. Party Conversation [25 points] (2 parts)

You are attending a party with n other people. Each other person i arrives at the party at some time s_i and leaves the party at some time t_i (where $s_i < t_i$). Once a person leaves the party, they do not return.

Additionally, each person i has some *coolness* c_i. At all times during the party, you choose to talk to the coolest person currently at the party. (All coolness values are distinct.) If you are talking to someone, and someone else cooler arrives at the party, you leave your current conversation partner and talk to the new person. If the person you are talking to leaves the party, you go talk to the coolest person remaining at the party. (This might or might not be a person with whom you have already talked.)

You are the first to arrive at the party and the last to leave. Additionally, you are the most popular person at the party, so everyone wants to talk with you.

(a) [15 points] Describe a data structure which allows you to decide in $O(1)$ time to whom you should talk at any moment. You should be able to update this data structure in $O(\log n)$ time when someone arrives or leaves.

Solution: Build a max-heap H_{cool} with each person's coolness as the key. Build another min-heap H_{time} with each person's leaving time as the key (so the coolest person at that moment will be at the root of H_{cool}, and the person leaving the earliest will be at the root H_{time}). Storing with the keys in the two heaps, we also store the index of the person. We also maintain an array links of size n which stores the current index of person i in H_{cool} and index in H_{time}. (This is important. Most students did not do this, and without this there is no way to find the node to delete from the heap when somebody leaves).

We use $Heap - Max(H_{cool})$ to decide with person you should talk to. If somebody leaves, use $Extrac - Min(H_{time})$ to find who is leaving. Suppose the person to leave is i. Use the array $links$ to get the index of person i in H_{cool} and $Heap - Delete(H_{cool}, index(i))$.

When a person j comes, perform $Heap-Insert(H_{cool}, c_j)$, $Heap-Insert(H_{time}, t_j)$ and store the indices of person j in the two heaps into array $links$.

Since $Heap - Max()$ can be done in $O(1)$ time, and both $Heap - Insert()$ and $Heap-Delete()$ can be done in $O(h) = O(\log n)$ time, so this data structure satisfies the requirements.

(b) [10 points] If you know in advance when each person will arrive at and depart from the party, describe an $O(n \log n)$-time algorithm to compute the total amount of time that you will spend talking with each person.

Solution: Given $\{s_i, t_i\}$, we first sort them in increasing order in $O(n \log n)$ time (you may instead assume the times are given as sorted, but you must explicitly say

so). While sorting these numbers, we keep track of the index of the person to which each time belong to and also if this is a arrival time or a departure time. We use an array of size n to store the start talking time and cumulative duration of each person. Initially, the cumulative duration is set to 0 and start talking time is set to NIL for each person. We use the dynamic data structure built in part (a) and initially we set the two heaps to be empty. We then linearly scan the sorted times of arrival/departure and after reading each number, we update the data structure built in part(a) to decide who you are talking to. When you start talking to a person i, we update the start talking time of i. When you stop talking to person i (this can happen if i leaves or some cooler person arrives), we subtract i's start talking time from the current time to get the duration of this talk and increment the cumulative duration of i (since you may talk to i again at a later time). Since each such update takes at most $O(\log n)$ time and there are at most $2n$ such updates (that is, we update only when someone comes or leaves), so the total time to compute the values in the array is $O(n \log n)$.

Problem 6. Space Efficient Sets [20 points] (3 parts)

In this problem, we will use hashing to test membership of an element in a set. Suppose we have a set S of n elements and we need to build a data structure that efficiently answers queries of the form: *is x in S?* Ben Bitdiddle comes up with the following idea: construct a hash table T of size $m > n$, where each entry of the hash table is a single bit. Now, use a hash function h to map each element in S to an index in the hash table, and set the corresponding bit of T to 1. Thus, $T[i] = 1$ iff $i = h(y)$ for some $y \in S$. His search strategy is simply to output *yes* iff $T[h(x)] = 1$.

(a) [7 points] Assume simple uniform hashing. If $x \notin S$, what is the probability that the algorithm answers *yes*? (This is called the probability of a *false positive*.)

Solution:

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

We do not want to find the probability of a given space already being filled directly because we cannot assume that all n elements went to different buckets. Rather, we should think of this in terms of the probability that the bucket which x hashes to has *not* been filled and then subtract this from 1.

Answers of $\frac{n}{m}$, which would be correct if this were an open addressing system or if one ignored independence, were given partial credit with appropriate explanation.

(b) [3 points] If $x \in S$, what is the probability that the algorithm answers *no*? (This is called the probability of a *false negative*.)

Solution: Zero

Given that $x \in S$, the corresponding bit, $T[h(x)]$ has been set to 1. There is no method given to set any bit to 0, so the algorithm will always find that bit set to 1 and return *yes*.

(c) [10 points] Ben Bitdiddle changes the algorithm to use k hash functions $h_1, h_2, \ldots, h_k$ as follows: we now set $T[i] = 1$ iff $i = h_j(y)$ for some $y \in S, j \in \{1, 2, \ldots, k\}$ and answer *yes* iff $T[h_j(x)] = 1$ for all $j \in \{1, 2, \ldots, k\}$. Assuming that hash functions are independent and each hash function satisfies the assumption of part (a), what is the probability of a false positive now?

Solution:

$$(1 - (1 - \frac{1}{m})^{nk})^k$$

This is the probability of a false positive for all k hashes of x. This is similar to the solution of part (a) but because each element of n is hashed k times we have k times as many hashes in the table.

Solutions neglecting the additional k factor or again neglecting independence were given partial credit with appropriate explanation.

Problem 7. Modal Weights [15 points] (2 parts)

You are given a collection of n iron weights of varying masses. Some weights have the same mass and some weights have different masses. You would like to find the size of the largest possible sub-collection of weights that all have the same mass.

(a) [8 points] Assume that the weights are unlabeled and that your only tool for comparing them is a balance scale which will compare two weights. The balance will tell you which weight is more massive, or tell you that they have the same mass. If there are k distinct mass values represented by the n weights, give an $O(n \log k)$-time algorithm for finding the size (the number of weights) of the largest collection of equal weights.

Solution: We construct an AVL tree on the weights. Each node of the AVL tree represents a unique mass corresponding to at least one weight in the collection. Observe that the *value* of the mass represented by a node is unknown; instead, a node (say, representing mass m) comprises a pointer to some weight having mass m (call this weight the *witness* for the node). Additionally, the node has a field where the number of weights with mass m is stored (call it the *frequency* of the node). Recall that the construction of an AVL tree from an unordered list of length n is a sequence of n insertions. So, it is sufficient to define the insertion procedure for a single weight in the AVL tree. To insert a new weight w, we compare the weight with the witness for the root node (call it r) using the balance, and do one of the following depending on the result of the comparison:

- if w is lighter than r, then recursively insert w in the left sub-tree,
- if w is heavier than r, then recursively insert w in the right sub-tree, and
- if w and r have equal weight, increment the frequency of the root node by 1.

If the sub-tree we are inserting into is empty, then create a new node as the root of the sub-tree, set w as its witness, initialize its frequency to 1, and rebalance the tree if necessary. Since the AVL tree has a unique node for each mass, and there are k different masses, the depth of the AVL tree is $O(\log k)$. Thus, the n insertions take $O(n \log k)$ time.

Once all weights have been inserted, traverse the AVL tree pre-/post-/in-order keeping a running maximum over the frequency values. The maximum frequency is returned by the algorithm. This traversal takes time proportional to the number of nodes in the tree, i.e. $O(k)$. Thus, the overall time complexity is $O(n \log k) + O(k) = O(n \log k)$.

(Note: Alternatively, we can maintain a running max while constructing the AVL tree as well.)

(b) [7 points] Now assume that you find a digital scale, which can tell you the exact mass of a weight. Now give an $O(n)$-time algorithm to accomplish the same task.

Solution: We use a hash table of size $2k$ where we store the unique mass values corresponding to the weights. Additionally, each hash table bucket has a *frequency* field that stores the number of weights whose mass is represented by the bucket. We weigh each weight, and hash its mass value into this table. If there is a collision, there are two possibilities: (1) if the masses are identical, we increment the frequency value, and (2) if the masses are unequal, we resolve collisions by open addressing. Finally, we scan each entry of the hash table to find the mass value with the maximum frequency. Since there are k distinct mass values and the size of the table is $2k$, the expected time for each insertion is $O(\frac{1}{1-0.5}) = O(1)$. Using linearity of expectation, the expected time for all insertions is $O(n)$. The final scan takes $O(k)$ time; so the overall time complexity is $O(n)$ in expectation.

(Note: As in part (a), the scan at the end is not required, we can maintain a running max during the insertions.)

SCRATCH PAPER

SCRATCH PAPER