Quiz 1 Solutions

Problem 1. We hope you mastered this question. Your name is that thing you include at the top of your problem set when you submit. Found yourself tongue-tied? This question is bound to show up on future quizes, so feel free to put it on your crib sheet for free points. Not that two points is enough to really dominate the quiz, but it's a start.

Problem 2. Asymptotics & Recurrences [20 points] (3 parts)

(a) [10 points] Rank the following functions by *increasing* order of growth. That is, find any arrangement $g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_8$ of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, $g_3 = O(g_4)$, $g_4 = O(g_5)$, $g_5 = O(g_6)$, $g_6 = O(g_7)$, $g_7 = O(g_8)$.

$$f_1(n) = n\log^2 n \qquad f_2(n) = n + \sqrt{n}\log^4 n \qquad f_3(n) = \binom{n}{4} \qquad f_4(n) = \binom{n}{n/3}$$
$$f_5(n) = \binom{n}{n-2} \qquad f_6(n) = 2^{\log^2 n} \qquad f_7(n) = n^{\sqrt{\log n}} \qquad f_8(n) = n^3\binom{n}{2}$$

Solution: $f_2, f_1, f_5, f_3, f_8, f_7, f_6, f_4$

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

$$T(n) = 16 \cdot T(n/4) + n^2\log^3 n$$

Solution: Using Master Theorem, we compare $n^2\log^3 n$ with $n^{\log_4 16} = n^2$. This is case 2 of the generalized version of the theorem as treated in class, so we increment the $\log^k n$ for $\Theta(n^2\log^4 n)$.

(c) [5 points] Find an asymptotic solution of the following recurrence. Express your answer using Θ-notation, and give a brief justification. (Note that $n^{\frac{1}{\log n}} = 1$.)

$$T(n) = T(\sqrt{n}) + 1$$

Solution: $T(n) = \Theta(\log\log n)$.

To see this, note that $\underbrace{\sqrt{\ldots\sqrt{n}}}_{i \text{ times}} = n^{1/2^i}$. So, once i becomes $\log\log n$ we will have $n^{1/2^i} = n^{1/\log n} = 1$. Thus the recursion stops after $\log\log n$ levels and each level contributes 1, hence $T(n) = \Theta(\log\log n)$.

Problem 3. True/False [18 points] (9 parts)

Circle (T)rue or (F)alse. You don't need to justify your choice.

(a) T F [2 points] Inserting into an AVL tree can take $o(\log n)$ time.

Solution: False. To answer this question, we need to know the length of the shortest possible path from the root to a leaf node in an AVL tree with n elements. In the best possible case, for each node we pass, the heights of its two children differ by 1, and we move to the child with the lower height. The child's height is then 2 less than the current node's height. So in the best case, each time we move to a new node, the height decreases by 2. The number of times we do this to get to height 0 is then the height of the root divided by 2. The height of the root is $\Theta(\log n)$, so it takes $1/2 \cdot \Theta(\log n) = \Theta(\log n)$ time to insert into an AVL tree, in the best case. Therefore it cannot take $o(\log n)$ time.

(b) T F [2 points] If you know the numbers stored in a BST and you know the structure of the tree, you can determine the value stored in each node.

Solution: True. You can do an inorder walk of the tree, which would order the nodes from smallest key to largest key. You can then match them with the values.

(c) T F [2 points] In max-heaps, the operations insert, max-heapify, find-max, and find-min all take $O(\log n)$ time.

Solution: False. The minimum can be any of the nodes without children. There are $n/2$ such nodes, so it would take $\Theta(n)$ time to find it in the worst case.

(d) T F [2 points] When you double the size of a hash table, you can keep using the same hash function.

Solution: False. If you double the size of the table, you need to change the hash function so that it maps keys to $\{0, 2m\text{-}1\}$ instead of mapping them to $\{0, m\text{-}1\}$. However, some people answered True, but explained that this would be inefficient. This answer also received full credit.

(e) T F [2 points] 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.

(f) T F [2 points] Merge sort can be implemented to be stable.

Solution: True. Whether it is stable or not depends on which element is chosen next in case there is a tie during the merge step. If the element from the left list (the list of elements that came earlier in the original array) is always chosen, then the merge sort is stable.

(g) T F [2 points] If we were to extend our $O(n)$ 2D peak finding algorithm to four dimensions, it would take $O(n^3)$ time.

Solution: True. The algorithm would break the 4-dimensional array into 16 subarrays, find the maximum element among 12 3-D planes ($O(n^3)$ elements), and recurse into a subarray with a higher neighbor. The recurrence is
$T(n) = T(n/2) + O(n^3) = O(n^3)$.

(h) T F [2 points] A $\Theta(n^2)$ algorithm always takes longer to run than a $\Theta(\log n)$ algorithm.

Solution: False. The constant of the $\Theta(\log n)$ algorithm could be a lot higher than the constant of the $\Theta(n^2)$ algorithm, so for small n, the $\Theta(\log n)$ algorithm could take longer to run.

(i) T F [2 points] Assume it takes $\Theta(k)$ time to hash a string of length k. We have a string of length n, and we want to find the hash values of all its substrings that have length k using a division hash. We can do this in $O(n)$ time.

Solution: True. Use a rolling hash.

Problem 4. Runway Reservation Modifications [20 points] (1 part)

Recall the Runway Reservation system used in Problem Set 2. We would like to expand the functionality of this system to deal with more types of requests.

Suppose there is a plane which needs to request an emergency landing at some time t_1 but there is already a flight scheduled to land there. Rather than shift this existing flight to the next open space, we would like to just push each subsequent flight back as much as we're able to until a big enough gap is created.

For example, assuming a window of 3 minutes with flights at times 28, 31, 35, 38, 42, 46, 49, 53, 57, 60, attempting to insert another flight at time 31 would cause a new set of flights at times 28, 31, 34, 37, 40, 43, 46, 49, 53, 57, 60. In this case we would say four flights needed adjusting - 31, 35, 38, and 42 - to accommodate the emergency landing.

Give an algorithm to find the minimum number of adjustments needed. This algorithm should find only the number of flight times requiring adjusting, it does not need to perform the flight updates. Full points will be given for an algorithm which operates well on a very tight schedule and has an asymptotic running time independent of the number of flights needing adjustment. You may use data structure augmentations provided that you explain the augmentation. Its maintenance may not increase the asymptotic running time of other operations but you are not required to prove this.

Solution: Augment the tree with Rank and do binary search on Select, comparing the total amount of time since t_1 with the number of flights which have landed until the boundary of enough time to insert another flight is found. Running time of $\log^2 n$ since each Select takes $\log n$ time and we do $\log n$ of these.

A more efficient solution is to augment the tree with Rank and simply walk down the tree, again comparing the number of flights since the insertion with the amount of time that has passed and recursing on the right child if there has not been enough time or the left child if there already has until the boundary is found. This means going down the tree only once, for a total of $\log n$ time.

Solutions running in time which was not polynomial in $\log n$ and potentially touched every node in the tree were given half-credit if correct. Solutions with incorrect or missing time analysis were also given half-credit. Common incorrect solutions looked only at children of the flight being collided with and didn't actually traverse the tree, thus failing to find the correct place to stop if the collided flight was a leaf.

Some students provided a clevel solution of "Augment each node with the number of flights which need to be moved if there is a collision at that node". Sadly, the course staff was unable to find an augmentation capable of doing this without increasing the asymptotic running time of other operations, since any insertion may cause all n nodes to need to have their augmentation updated.

Problem 5. Computing Fibonacci numbers [20 points] (3 parts)

The Fibonacci numbers are defined by the following recurrence: $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$, yielding the sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ There is a closed-form formula for F_n given by $F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618$ is the golden ratio. However, this formula isn't practical for computing the exact value of F_n as it would require increasing precision on $\sqrt{5}$ and ϕ as n increases. In this problem we are interested in obtaining practical algorithms for computing the n^{th} Fibonacci number F_n for any given n. Assume that the cost of adding, subtracting, or multiplying two integers is $O(1)$, independent of the size of the integers we are dealing with.

(a) [5 points] From the recurrence definition of the Fibonacci sequence, one can use the following simple recursive algorithm:

```
$FIB1(n)$:
  **if** $n \leq 1$ **then**
    **return** $n$
  **else**
    $x \leftarrow FIB1(n-1)$
    $y \leftarrow FIB1(n-2)$
    **return** $x + y$
  **end if**
```

Give the running time of this algorithm. Express your answer using Θ-notation.

Solution: Let $T(n)$ be the time taken to compute F_n. We have $T(n) = T(n-1) + T(n-2) + \Theta(1)$, and so $T(n) = \Theta(F_n) = \Theta(\phi^n)$.

(b) [5 points] Give an algorithm that computes F_n in $\Theta(n)$ and justify its running time.

Solution: To obtain the desired algorithm we develop an iterative (non-recursive) version of the previous algorithm in which we memorize the last two computed Fibonacci numbers, so we are able to compute the next number in constant time. An example code looks as follows:

```
$FIB2(n)$:
  $i \leftarrow 1$
  $j \leftarrow 0$
  for $k = 1$ to $n$ do
    $temp \leftarrow i + j$
    $j \leftarrow i$
    $i \leftarrow temp$
  end for
  return $j$
```

To analyze the complexity of this algorithm it is sufficient to note that we have n iterations of the loop and each iteration can be performed in $O(1)$ time.

(c) [10 points] Consider the matrix $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Show that for $i \geq 1$, $A^i = \begin{pmatrix} F_{i-1} & F_i \\ F_i & F_{i+1} \end{pmatrix}$. From the fact that $A^{2i} = (A^i)^2$, and $A^{2i+1} = A*(A^i)^2$, use divide and conquer to show that A^n can be calculated in time $\Theta(\log n)$. Conclude by giving a $\Theta(\log n)$ algorithm for computing F_n.

Solution: We first show how to compute A^n in $\Theta(\log n)$ time.

Consider the following algorithm:

```
$Matrix\_power(A, n)$:
  $C \leftarrow Matrix\_power(A, \lfloor n/2 \rfloor)$
  $C \leftarrow C \cdot C$
  **if** $n$ odd **then**
    $C \leftarrow C \cdot A$
  **end if**
  **return** $C$
```

The recurrence describing the running time $T(n)$ of this algorithm is $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. (Note that we work here with 4-by-4 matrices so multiplying them takes $O(1)$ time.) Using Master Theorem we get $T(n) = \Theta(\log n)$.

Now, to compute F_n we just call $C := Matrix\_power(A, n)$ and return the entry $C[1, 2]$ of the computed answer. Clearly, this takes $\Theta(\log n)$ time.

Problem 6. One Hash, Two Hash [20 points] (4 parts)

We talked in class about two methods for dealing with collisions: chaining and linear probing. Cornelius Beefe decided to come up with his own method. He creates two hash tables T_1 and T_2, each of size m, and two different hash functions h_1 and h_2 and decides to use T_2 to resolve collisions in T_1. Specifically, given an element x, he first calculates $h_1(x)$ and tries to insert x into T_1. If there is a collision, he calculates $h_2(x)$ and inserts x into T_2.

(a) [4 points] Assume T_2 uses chaining to deal with collisions. Given an element y, give an algorithm for deciding whether or not y is in the hash table.

Solution: Hash y using $h_1(y)$ and check if $T_1[h_1(y)]$ contains an element. If this element is y, return true. Otherwise, hash y using $h_2(y)$ and check the list at $T_2[h_2(y)]$.

Scoring:

4 points: For anything like the above solution

2 points: If you forgot to say something about hashing first to T_1

(b) [2 points] Cornelius tries using the following hash functions for each table:

1) $h_1(x) = (a_1)^x \bmod m$ for table T_1
2) $h_2(x) = (a_2)^x \bmod m$ for table T_2

He first tries $a_1 = a_2$ and uses chaining in table T_2. Is this likely to result in fewer collisions than if he had just used one hash table of size m with chaining?

Solution: No. Items that collide in T_1 also collide in T_2.

Scoring:

2 points: If you said No.

1 point: If you said yes, but showed you understood something of how the two tables interact.

(c) [4 points] Assume $m = 21$. Circle the set of values that will result in the fewest collisions in T_2 and explain why you chose it.

(1) $a_1 = 9, a_2 = 5$

(2) $a_1 = 5, a_2 = 7$

(3) $a_1 = 4, a_2 = 16$

(4) $a_1 = 13, a_2 = 11$

Solution: There were two correct solutions. Full credit was given for saying $a_1 = 13$ and $a_2 = 11$ because neither is a power of the other (also gave credit for saying a_1 and a_2 were relatively prime to each other) and both are relatively prime to 21.

Full credit was also given if you worked out the residuals modulo 21 and saw that $a_1 = 9, a_2 = 5$ had the most possible residuals.

Scoring:

4 points: If you gave either of the two above answers with a correct explanation.

3 points: If you argued that a_1 and a_2 were relatively prime to the size of the table, but did not mention their relationship to each other.

2 points: If you said a_1 and a_2 were prime and/or relatively prime, but did not say that they were relatively prime to the size of the table.

1 point: If you circled one of the two correct answers with no justification.

(d) [10 points] Consider the case in which T_2 uses linear probing to deal with collisions. Also, assume that h_1 satisfies the simple uniform hashing assumption and h_2 satisfies the uniform hashing assumption. We insert 4 elements. What is the probability that while inserting the fourth element there are at least two collisions?

Solution: There must be one collision in the first hash table and (at least) one in the second.

There are four ways this can occur:

1) k_1 and k_2 mapped to the same position in T_1 $(1/m)$, k_3 did not collide in T_1 $((m-1)/m)$, k_4 collided with k_1 or k_3 in T_1 $(2/m)$, and k_4 collided with k_2 in T_2 $(1/m)$. The probability of this case is $(1/m)((m-1)/m)(2/m)(1/m) = (2(m-1))/m^4$.
2) k_1 and k_3 collide in T_1, while k_2 does not. Has the same probability as the first case (just different order), $(2(m-1))/m^4$.
3) k_2 and k_3 collide in T_1. Again the same probablity of $(2(m-1))/m^4$
4) k_1, k_2, and k_3 mapped to the same position in T_1 $(1/m^2)$, k_4 collides with k_1 in T_1 $(1/m)$, and with k_2 or k_3 in T_2 $(2/m)$. The probability is $(1/m^2)(1/m)(2/m) = 2/m^4$

The total probability is therefore

$$\frac{6m-4}{m^4}$$

Scoring: Many people solved the incorrect problem in this part. Rather than solving for two collisions during the fourth insertion they solved for two collisions over all four insertions. This is a much easier problem so one point was taken off. If you tried to solve this problem and made a mistake, you were generally penalized more harshly because it was an easier problem.

10 points: For the correct answer

9 points: If you came close to the correct answer but missed a case, evaluated a probability wrong, or made some other fairly major math mistake. If you solved the wrong problem correctly, you also received 9 points.

7-8 points: If you had the right idea, but missed a lot of cases or evaluated most of the probabilities incorrectly. 7 points was also given if you explained exactly how to solve the problem but showed no math.

5-6 points: If you understood that one collision needed to occur in T_1 before the fourth insertion and/or that only one collision could occur in T_1 during the fourth insertion, but did not explain your answer clearly.

3-4 points: If you showed some understanding of the simple uniform hashing assumption, but did not demonstrate much understanding of how the two tables worked together.

1-2 points: If you realized that the probability should decrease with table size, but showed little other work or explanation.

Problem 7. Extreme Temperatures [20 points] (1 part) Professor Daskalakis is interested in studying extreme temperatures on the Arctic Cap. He placed temperature-measuring devices at m locations, and programmed each of these devices to record the temperature of the corresponding location at noon of every day, for a period of n days. Moreover, using techniques that he learned while preparing the Heapsort lecture, he decided to program each device to store the recorded temperatures in a max-heap. To cut a long story short, Prof. Daskalakis now has m devices that he collected from the Arctic Cap, each of which contains in its hard-drive a max-heap of n elements. He now wants to compute the ℓ largest temperatures that were recorded by any device, e.g., if $m = 2$, $n = 5$, $\ell = 5$, and the two devices recorded temperatures $(-10, -20, -5, -34, -7)$ and $(-13, -19, -2, -3, -4)$ respectively, the desired output would be $(-2, -3, -4, -5, -7)$. Can you help your professor find the ℓ largest elements in $O(m + \ell \log \ell)$ time? Partial credit will be given for less efficient algorithms, as long as the run-time analysis is correct.

Solution: Build a max-heap $\mathcal{H}$ containing the numbers at the root of every heap. This takes $O(m)$ time. Now go through the nodes of $\mathcal{H}$. Each of these nodes v is a root of a max-heap H_v from some device. v has at most 2 subtrees rooted at it in $\mathcal{H}$ and two subtrees T_{vL} and T_{vR} rooted at it in H_v. "Add" the subtrees T_{vL} and T_{vR} to node v of heap $\mathcal{H}$, by adding a record at node v remembering the indices of the roots of these subtrees in the array representing heap H_v. This preprocessing step takes overall time $O(m)$, the resulting max-heap $\mathcal{H}$ contains all collected data, and all nodes of $\mathcal{H}$ have at least 2 and at most 4 children. Now in $O(\ell \log \ell)$ time find the ℓ biggest elements of $\mathcal{H}$ as follows: Start at the root of $\mathcal{H}$. This is the biggest among all elements, so you can output it as the biggest number. Now insert all of its children in a new max-heap $\mathcal{H}'$; this takes constant time since there are at most 4 of them. For each element that is inserted into $\mathcal{H}'$ throughout the execution of the algorithm, we are going to remember its location in $\mathcal{H}$. Now looking at the root of $\mathcal{H}'$, we can find the second biggest element among all collected elements. We output it, extract it from the heap $\mathcal{H}'$, and add to $\mathcal{H}'$ its at most 4 children in $\mathcal{H}$. Then we extract the new root of $\mathcal{H}'$, and so on. Every time we output an element we insert at most 4 elements in $\mathcal{H}'$, hence the size of $\mathcal{H}'$ won't grow beyond 4ℓ. Hence, every time we insert into $\mathcal{H}'$ we pay at most $O(\log \ell)$. So, our running time is $\ell \log \ell$. It remains to argue that the proposed algorithm outputs the largest ℓ elements. Notice that the largest elements of $\mathcal{H}$ form a subtree rooted at the root of $\mathcal{H}$. Argue by induction that a supertree of this subtree is inserted into $\mathcal{H}'$; indeed, all children of every output element are inserted into $\mathcal{H}'$.