Problem 1. Recurrence

Fall 2010 1(c): Find an asymptotic solution of the following recurrence. Express your answer using Θ-notation, and give a brief justification. (Note that $n^{\frac{1}{\log n}} = 2 = \Theta(1)$.)

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

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 $n^{1/2^i} = n^{1/\log n} = \Theta(1)$. Thus the recursion stops after $\log \log n$ levels and each level contributes $\Theta(1)$, hence $T(n) = \Theta(\log \log n)$.

Problem 2. Asymptotics

Spring 2010 3(a): 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, $n! \approx \sqrt{2\pi n}\,(n/e)^n$:

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

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

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

Problem 3. Divide and Conquer

Spring 2010 4: You are given two Python lists (arrays) of integers, each of which is sorted in ascending order and each of which has length n. All the 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:

Assume that lists are 0-indexed.

$O(n)$ solution

A straight-forward solution (worth $\frac{1}{4}^{\text{th}}$ of the credit) is to observe that the arrays are already sorted, so we can merge them, and stop after n steps. The first $n-1$ elements do not need to be copied into a new array, so this solution takes $O(n)$ time and $O(1)$ memory.

$O(\log^2 n)$ solution

The $O(\log^2 n)$ solution (worth $\frac{1}{2}$ credit) uses alternating 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 $p1' + p2' = n-1$ or $p1' + p2' = n$, we return one of the 5 elements. If $p1 + p2 \geq n$, 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 $O(\log^2 n)$.

$O(\log n)$ solution

The best solution comes from what is effectively a binary search modified to account for two separate inputs simultaneously. We assume that the lists support constant-time access to their elements. (Perhaps they are Python lists.)

There are two trivial cases, wherein all of the elements of one list are smaller than all the elements of the other; if $l1[n-1] < l2[0]$ our answer is $l1[n-1]$, and if $l2[n-1] < l1[0]$ it is $l2[n-1]$. But what if the two lists are interleaved?

Suppose we are searching the interval (a, b). Let i and j be indices into $l1$ and $l2$, respectively. We'll start with $i = (a + b)/2$ (halfway through the search range) and then $j = n - i - 1$. This way, if we've chosen i and j properly, we're looking at the nth-smallest element.

Compare $l1[i]$, $l1[i + 1]$, $l2[j]$, and $l2[j + 1]$. If $l1[i] < l2[j]$ and $l1[i + 1] > l2[j]$ then we've found the right interleaving of the two lists; $l2[j]$ is greater than exactly $i + 1 + j - 1$ = $n - 1$ elements (and therefore is the nth-smallest), and is our answer. Symmetrically, if $l2[j] < l1[i]$ and $l2[j + 1] > l1[i]$, then we return $l1[i]$.

If both $l1[i]$ and $l1[i + 1]$ are less than $l2[j]$, then the rank of $l2[j]$ is greater than n, and we need to take more elements from $l1$ and fewer from $l2$. We recurse on the upper half of the search interval, letting $a = i + 1$. In the opposite case, we recurse on the lower half, and let $b = i - 1$.

```
findNth(a, b)
$i = \lfloor (a + b)/2 \rfloor$
$j = n - i$
if $l1[i] < l2[j]$:
  if $l1[i + 1] > l2[j]$:
      return $l2[j]$
  else:
      return findNth($i + 1$, $b$)
else:
  if $l2[j + 1] > l1[i]$:
      return l1[i]
  else:
      return findNth($a$, $i - 1$)
```

Since we are halving the size of the problem each time we recurse, and each call involves constant work, this is a logarithmic-time solution. The relevant reccurence is $T(n) = T(n/2) + \Theta(1)$.

Problem 4. Bunker hill

Given a $W \times H$ 2-D document (the hill), e.g.

1	2	1	2	3	0	0	2	1	3
2	1	2	1	0	0	1	2	1	2
1	2	1	2	1	3	2	1	2	3
3	3	3	1	2	1	1	2	1	1
0	0	0	2	1	2	0	1	0	3
0	3	1	3	2	1	3	2	1	0

Find all $w \times h$ 2-D patterns (the bunkers), e.g.

1	2	1
2	1	2
1	2	1

Solution: An optimal solution runs in $O(WH)$ time (bounded by the need to read in the input array) and uses the Rabin-Karp algorithm, modified for 2-D documents.

We want to slide a window of size $w \times h$ over the $W \times H$ hill, compute its hash, and compare it with the hash of the bunker. If the hashes match, we'll do brute-force comparisons. Assuming a good hash function, this won't happen too often. In order to make this work in $O(WH)$ time, we need to be able to slide the window in $O(1)$ time.

We could use a single rolling hash of $w \times h$ elements, and Slide h times every time we move the window to the right, but this would yield a running time of $O(WHh)$.

Instead, we use W rolling hashes, which map to 1-column mini-windows of size h. Each of these mini-windows can slide down 1 row in $O(1)$ time, so we can slide all of them down in $O(W)$ time. Accounting for initialization, sliding the windows $H - h + 1$ times will take $O(WH)$ time. When the W rolling hashes are "in position", we use their hash values as the document for the standard 1-D Rabin-Karp string matching algorithm, where the pattern is the w hashes of the w columns in the bunker. The W hashes represent a $W \times h$ window in the hill, so a 1-D Rabin-Karp match translates in a $w \times h$-column match against the bunker. $H - h + 1$ rounds of 1-D Rabin-Karp run in $O(W(H - h + 1)) = O(WH)$ time. So the total running time is $O(WH)$.

Problem 5. Fake coin

You have N (e.g., 12) coins that are visually identical. The coins are supposed to be made of gold, but one of them is fake. The fake coin is made of a less dense material, and therefore is lighter. You have one ideal balancing scale of infinite precision, and want to find the fake coin with the minimum number of weighing experiments. How many experiments? What is the strategy?

Solution: $\Omega(\log_3 N)$, obtained by applying the decision tree method used to prove lower-bounds in the comparison-based model. ($\Omega(\log N)$ for searching, and $\Omega(N \log N)$ for sorting).

A weighing experiment will place some coins on the left side of the scale, some other coins on the right side of the scale, and then read the scale's outcome, which can be $<$, $=$, or $>$. The fact that there are 3 possible outcomes rules out binary search from the set of optimal solutions, because binary search will never use the $=$ outcome.

Any algorithm can be mapped out to a decision tree that starts out with no information and makes weighing experiments. Because experiments have 3 possible outcomes, each node in an optimal algorithm's decision tree will have 3 children. So the number of nodes in a tree of height h (corresponding to h experiments) is 3^h. A correct algorithm will have at least N leaves, with the outcomes "coin i is fake" for $1 \leq i \leq N$. Therefore, the minimum height of an algorithm that is both correct and optimal is $\geq \log_3 N$. Note that we can't use Ω here, because $\log_3 N$ and $\log_2 N$ only differ by constant factors.

Once we have the lower bound, we can use intuition to figure out the correct algorithm. The base 3 in the logarithm suggests that we should split the available coins into 3 piles at each experiment. After weighing two piles, we can draw the following conclusions, based on the outcome:

$<$: the fake coin is in the first pile

$>$: the fake coin is in the second pile

$=$: the fake coin is in the third pile that was not weighed

Problem 6. Basic Concepts

Spring 2010 2: For each of the following questions, circle either T (True) or F (False). There is no penalty for incorrect answers.

(a) T F 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 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 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 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 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 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 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 $n-1$).

Problem 7. 3D Peak Finding

Spring 2011 4: Consider a 3D matrix B of integers of size $n \times n \times n$. Define the ***neighborhood*** of an element $x = B[i][j][k]$ to consist of

$$\begin{array}{lll} B[i+1][j][k], & B[i][j+1][k], & B[i][j][k+1], \\ B[i-1][j][k], & B[i][j-1][k], & B[i][j][k-1], \end{array}$$

that is, the six elements bordering the one in question, not including diagonals. (For elements on a face we consider only five neighbors; for elements on an edge we consider only four neighbors; and for the eight corner elements we consider only three.) An element x is a ***peak*** of B if it is greater than or equal to all of its neighbors. Consider this algorithm for finding a peak in matrix B:

1. Consider the nine planes $i = 0$, $i = \frac{n}{2}$, $i = n-1$, $j = 0$, $j = \frac{n}{2}$, $j = n-1$, $k = 0$, $k = \frac{n}{2}$, $k = n-1$, which divide the matrix into eight submatrices.
2. Find the maximum element m on any of the nine planes.
3. If m is a peak, return it.
4. Otherwise, m has a larger neighbor element. Recurse into the submatrix that contains that larger neighbor element (including elements in bordering planes).

What is the running time of this 3D peak finding algorithm?

Solution: Size of original problem of finding a peak in a $n \times n \times n$ 3D matrix is $T(n)$. Time taken to divide is finding the maximum element of nine $n \times n$ planes, which takes $O(n^2)$. Size of subproblem to recurse to is $T(n/2)$, and there is only one subproblem. Recurrence is $T(n) = T(n/2) + O(n^2)$, which is $O(n^2)$ by Master Theorem Case 3.

Each of the following were awarded 2 points:

1. Correct size of the subproblem ($n/2$). A common mistake was to think that subproblem sizes were $n/8$. The reason for the confusion was that the total volume is indeed $1/8$th, as $(n/2)^3 = n^3/8$.
2. Correct number of subproblems (which is 1). A common mistake was to think that there were 8 subproblems, but in fact, we recurse on only one of the subcubes.
3. Correct time for max finding ($f(n) = O(n^2)$. A common mistake was to think that this could be done with 2D peak finding, but in fact we're looking for the maximum, not the peak, of each array, hence we need to examine all n^2 elements).
4. Correct case of the master theorem (case 3) and explanation of why this case applies.
5. Correct final runtime ($O(n^2)$). If this number appeared magically with very little or no explanation, a maximum of 4 points were awarded, depending on the strength of the explanation, as all the above points were missing.